TEXAS INSTRUMENTS

TMS34010

User's Guide

User's Guide

TMS34010

1988

1988

Graphics Products

TMS34010 User's Guide

TEXAS
INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to or to discontinue any semiconductor product or service identified in this publication without notice. TI advises its customers to obtain the latest version of the relevant information to verify, before placing orders, that the information being relied upon is current.

TI warrants performance of its semiconductor products to current specifications in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Unless mandated by government requirements, specific testing of all parameters of each device is not necessarily performed.

TI assumes no liability for TI applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Contents

Illustrations

Tables

Table *Page*

Section 1

Introduction

The TMS34010 Graphics System Processor (**GSP**) is an advanced 32-bit microprocessor, optimized for graphics systems. The TMS34010 is a member of the TMS340 family of computer graphics products from Texas Instruments.

A single TMS34010 provides a cost-effective solution in applications that require efficient data manipulation. The TMS34010 can be configured to serve in either a host-based or a stand-alone environment. Systems based on multiple TMS34010 devices are implemented using special features of the TMS34010's local and host interfaces.

The TMS34010 is well supported by a full set of hardware and software development tools, including a full-speed emulator, a software simulator, an IBM-PC development board, a C compiler, predeveloped software libraries, and assembly language tools.

Topics covered in this introductory section include:

1.1 TMS34010 Overview

The TMS34010 combines the best features of general-purpose processors and graphics controllers to create a powerful and flexible Graphics System Processor. Key features of the TMS34010 are its speed, high degree of programmability, and efficient manipulation of hardware-supported data types such as pixels and two-dimensional pixel arrays.

The TMS34010's unique memory interface reduces the time needed to perform tasks such as bit alignment and masking. The 32-bit architecture supplies the large blocks of continuously-addressable memory that are necessary in graphics applications. TMS34010 system designs can take advantage of video RAM (such as the TMS4461) technology to facilitate applications such as high-bandwidth frame buffers; this circumvents the bottleneck often encountered when using conventional DRAMs in graphics systems.

The TMS34010 instruction set includes a full complement of general-purpose instructions, as well as graphics functions, from which you can construct efficient high-level functions. The instructions support arithmetic and Boolean operations, data moves, conditional jumps, and subroutine calls and returns.

The TMS34010 architecture supports a variety of pixel sizes, frame buffer sizes, and screen sizes. On-chip functions have been carefully selected so that no functions tie the TMS34010 to a particular display resolution. This enhances the portability of graphics software, and allows the TMS34010 to adapt to graphics standards such as MIT's X, CGI/CGM, GKS, NAPLPS, PHIGS, and evolving industry and display management standards.

1.2 Key Features

- Fully programmable 32-bit general-purpose processor
- 128-megabyte address range
- Instruction cycle times:
  - 132 ns (TMS34010-60)
  - 160 ns (TMS34010-50)
  - 200 ns (TMS34010-40)
- On-chip peripheral functions include:
  - Programmable CRT control (horizontal sync, vertical sync, and blanking)
  - Direct interfacing to conventional DRAMs and multiport video RAMs
  - Automatic CRT display refresh
  - Direct communications with an external (host) processor
- Instruction set includes special graphics functions such as pixel processing, XY addressing, and window clip/hit
- Programmable 1, 2, 4, 8, or 16-bit pixel size with 16 Boolean and 6 arithmetic pixel-processing options
- 30 general-purpose 32-bit registers
- 256-byte on-chip instruction cache
- Dedicated 8/16-bit host-processor interface and HOLD/HLDA interface
- 32-bit and 64-bit integer arithmetic
- High-level language support
- Full line of hardware and software development tools including:
  - C compiler
  - Macro assembler
  - Linker
  - Archiver
  - Software application libraries
  - XDS (Extended Development Support) in-circuit emulator
  - Software development board (SDB)
  - ROM utility
  - Simulator
  - Symbolic debugger
- 68-pin PLCC package
- 5-V CMOS technology

1.3 Architectural Overview

Figure 1-1 illustrates the TMS34010's major internal functions and its interfaces to external devices. The on-chip processor executes both graphics instructions and general-purpose instructions. The TMS34010 is a true 32-bit processor, with 32-bit internal data paths, a 32-bit ALU, and a large address space. Thirty 32-bit general-purpose registers, a 32-bit stack pointer, and a 256-byte instruction cache increase performance. Nonprocessor functions included on the chip include CRT timing, screen refresh, and DRAM refresh. Separate physical interfaces are provided for communicating with a host processor, for providing the video timing signals necessary to control a CRT monitor, and for connecting directly to dynamic RAMs (like the TMS4256 or TMS4C1024) and video RAMs (such as the TMS4461).

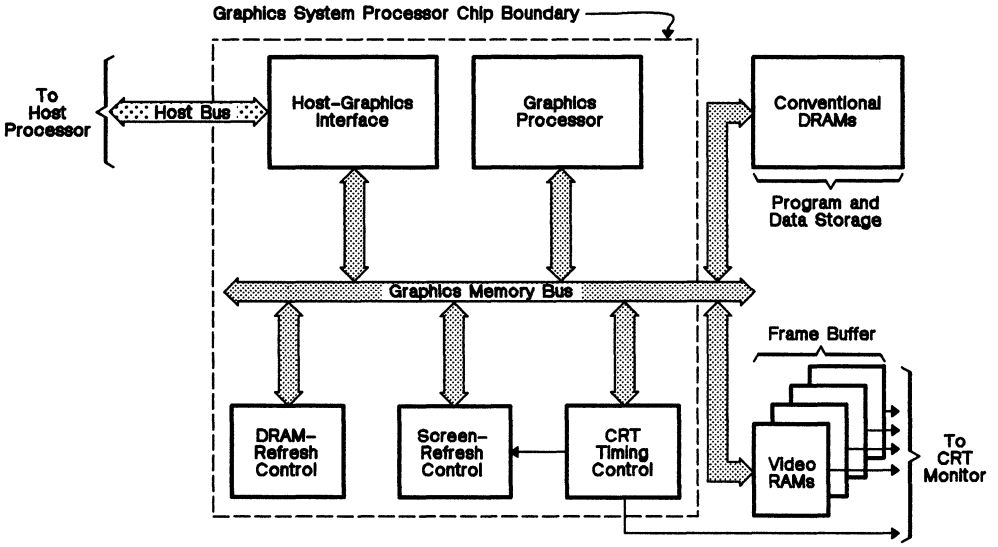


Figure 1-1. System Block Diagram

1.3.1 TMS34010 Block Diagram

Figure 1-2 illustrates the internal architecture of the TMS34010; the following subsections describe the individual blocks shown in Figure 1-2.

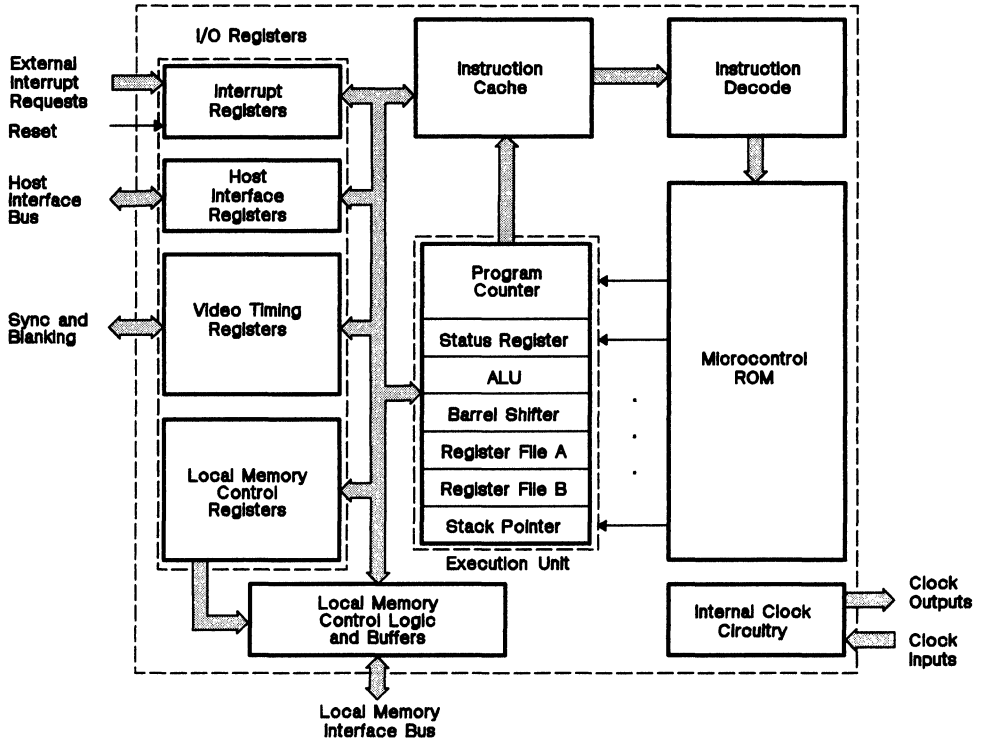


Figure 1-2. Internal Architecture Block Diagram

1.3.1.1 CPU Internal Functions

The center portion of Figure 1-2 highlights the main internal functions of the TMS34010:

- The 32-bit **program counter** (PC) points to the next instruction word to be fetched. The PC's four LSBs are always 0. Section 5.3 (page 5-18) discusses the program counter.
- The 32-bit **status register** (ST) specifies the status of the TMS34010 processor. It contains the sign, carry, zero, overflow, interrupt enable, and PixBlt execution status bits. It also specifies the lengths and field extension modes of fields 0 and 1. Section 5.2 (page 5-17) discusses the status register.

- **Register files A and B** each contain 15 general-purpose registers, A0–A14 and B0–B14, respectively. The B-file registers are also used as implied operands for the graphics instructions. Section 5.1 (page 5-2) discusses the register files.

  The general-purpose register files are dual ported to support parallel data movement. Two separate internal buses route data from the registers to the ALU, and a third bus routes results back to the registers.

- The **stack pointer**, or **SP**, is available to instructions that operate on either register file.

- The 32-bit **barrel shifter** shifts or rotates 32-bit operands from 1 to 32 bit positions in a single machine state.

- The 32-bit **ALU** is connected to the other CPU components by 32-bit data paths. This allows most register-to-register operations to be performed in a single machine state. (Accessing external memory requires a minimum of two states.) The following actions occur in parallel during a single state:

  1) Two operands are transferred from the selected general-purpose register file to the ALU.
  2) The ALU performs the specified operation on the operands.
  3) The result is routed back to the general-purpose register file.

1.3.1.2 Instruction Cache

The TMS34010 contains a 256-byte instruction cache that can contain up to 128 instruction words (an instruction word may be an entire single-word instruction or 16 bits of a multiple-word instruction). Section 5.4 (page 5-19) describes instruction cache operation.

1.3.1.3 I/O Registers

Twenty-eight 16-bit, on-chip I/O registers are dedicated to peripheral control functions. The I/O registers are divided into four categories:

- Seven **local memory interface registers** are dedicated to memory interface control and configure the memory controller.

- Fourteen **video timing and screen refresh registers** generate the sync and blanking signals used to drive a CRT, and schedule screen-refresh cycles.

- Five **host interface registers** are accessible to external host processors as well as to the TMS34010. Status information can be communicated directly through these registers. Large blocks of data in TMS34010 memory can be accessed indirectly through pointer registers.

- Two **interrupt control registers** provide status information about interrupt requests.

Section 6 provides individual descriptions of each I/O register.

1.3.1.4 Microcontrol ROM

The TMS34010 transfers decoded instructions to the microcontrol ROM for interpretation. The microcontrol ROM has 166 control outputs and 808 microstates.

1.3.1.5 Clock Timing Logic

The clock timing logic converts the clock input signals to internal timing signals and generates the clock output signals, LCLK1 and LCLK2, used by external devices. The machine state is a fundamental time unit of the graphics processor in the TMS34010; it is the time interval during which the processor is in a particular microinstruction state. The instruction timing for each assembly language instruction is specified in multiples of machine states. The TMS34010's machine state is a single local clock period (the time from one LCLK1 low-to-high transition to the next) in duration.

1.3.2 Other Special Processing Hardware

The TMS34010 CPU also supports the following special processing functions in hardware:

- Detecting whether a pixel lies within a specified display window
- Detecting the leftmost one in a 32-bit register
- Expanding a black-and-white pattern to a variable pixel-depth pattern

1.4 Typical Applications

The TMS34010's 32-bit processing power and its ability to handle complex data structures make it well suited for a variety of applications. These include display systems, imaging systems, mass storage, communications, high-speed controllers, and peripheral processing. The TMS34010's efficient bit manipulation facilitates demanding tasks such as high-quality, proportionally-spaced text; this capability makes it especially useful in applications such as desktop publishing. In graphics display systems, the TMS34010 provides cost-effective performance for color or black-and-white bit-mapped displays. Table 1-1 lists typical end uses of the TMS34010.

Table 1-1. Typical Applications of the TMS34010

Computers

- Terminals and CRTs
- Windowing systems
- Electronic publishing
- Laser printers
- Personal computers
- Printers and plotters
- Engineering workstations
- Copiers
- Document readers
- FAX
- Imaging
- Data processing

Industrial Control

- Robotics
- Process control
- Instrumentation
- Motor control
- Navigation

Telecommunications

- Video phones
- PBX

Consumer Electronics

- Automotive displays
- Information terminals
- Cable TV
- Home control
- Video games

1.5 Manual Organization

The *TMS34010 User's Guide* describes TMS34010 operation, focusing on the TMS34010's role in applications that involve CRT-based, bit-mapped, graphics systems. The user's guide is divided into four major sections:

1) General information (Section 1)
2) Architecture (Sections 2–8)
3) Timing (Sections 9–11)
4) Instruction set (Sections 7, 12, and 13)

A glossary, an index, and a reference card are also provided.

Section 1 Introduction

Provides an overview of the TMS34010 and TMS34010 architecture, including key features, a block diagram, and typical applications. Discusses manual organization and lists suggested reading.

Section 2 Pin Functions

Illustrates the TMS34010 pinout and contains general pin descriptions. Also describes specific pin functions regarding the host interface, the local bus interface, video timing signals, hold and emulator interface pins, and power, ground, and reset pins.

Section 3 Memory Organization

Discusses 32-bit addressing methods, the TMS34010 memory map, and the stack.

Section 4 Hardware-Supported Data Structures

Discusses hardware-supported data structures (such as fields and pixels) and XY addressing.

Section 5 CPU Registers and Instruction Cache

Describes general-purpose register files A and B (including a reference of the B registers' graphics functions), the status register, the program counter, and the instruction cache.

Section 6 I/O Registers

Provides a detailed discussion of host interface registers, memory-interface control registers, video timing and screen refresh registers, interrupt interface registers, and I/O register addressing. Includes an alphabetical reference of the I/O registers.

Section 7 Graphics Operations

Discusses graphics instructions such as PixBlts, PIXTs, and related topics such as 2-dimensional arrays of pixels, window checking, XY-to-linear conversion, and plane masking.

Section 8 Interrupts, Traps, and Reset

Describes external and internal interrupts, interrupt processing, and reset.

Section 9 Screen Refresh and Video Timing

Describes the horizontal sync, vertical sync, and blanking signals, horizontal and vertical timing, and video RAM control.

Section 10 Host Interface Bus

Discusses host interface pins, registers, and timing.

Section 11 Local Memory Interface Bus

Discusses local memory interface timing, addressing mechanisms, and data manipulation at the local memory interface.

Section 12 Assembly Language Instruction Set

Discusses addressing modes, summarizes MOVE, PIXBLT, and PIXT instruction variations, and presents the entire TMS34010 assembly language instruction set in alphabetical order.

Section 13 Instruction Timings

Contains an overview of timing for general instructions, and specific timing information for move and graphics instructions.

Appendix A TMS34010 Data Sheet

Appendix B Emulation Guidelines for Prototyping

Appendix C Software Compatibility with Future GSPs

Appendix D Glossary

1.6 Related Documentation, References, and Suggested Reading

The following books and articles provide further background in graphics and system concepts associated with graphics.

Artwick, Bruce A. *Applied Concepts in Microcomputer Graphics*. Englewood Cliffs, New Jersey: Prentice-Hall, 1984.

Asal, Short, Preston, Simpson, Roskell, and Guttag. "The Texas Instruments 34010 Graphics System Processor." *IEEE Computer Graphics and Applications* vol.6 no.10, pp. 24–39.

Bresenham, J.E. "Algorithm for Computer Control of a Digital Plotter." *IBM Systems Journal* 4 No.1 (1965): 25–30.

Bresenham, J.E. "A Linear Algorithm for Incremental Display of Digital Arcs." *Communications of the ACM* 20 (Feb. 1977): 100–106.

Cody, William J. Jr., and William Waite. *Software Manual for the Elementary Functions*. Englewood Cliffs, New Jersey: Prentice-Hall, 1980.

Foley, James, and Andries van Dam. *Fundamentals of Interactive Computer Graphics*. Reading, Massachussetts: Addison-Wesley, 1982.

Gupta, Satish. "Architectures and Algorithms for Parallel Updates of Raster Scan Displays." Tech. Report CMU-CS-82-111, Computer Science Dept., Carnegie Mellon University, 1981.

Ingalls, D.H. "The Smalltalk Graphics Kernel." Special issue on Smalltalk, *Byte*, August 1981, pp. 168-194.

Kernighan, B., and D. Ritchie *The "C" Programming Language*. Englewood Cliffs, New Jersey: Prentice-Hall, 1978.

Killebrew, C.R. Jr., "The TMS34010 Graphics System Processor." *BYTE*, December 1986, pp. 193–204.

Kochan, Stephen G. *Programming in C*. Hasbrouck Heights, New Jersey: Hayden Book Company, 1983.

Newman, W.M., and R.F. Sproull. *Principles of Interactive Computer Graphics*. 2nd ed. New York: McGraw-Hill, 1979.

Pike, Rob. "Graphics in Overlapping Bitmap Layers." *ACM Transactions On Graphics* 2 (April 1983): 135–160.

Pinkham, R., M. Novak, and K. Guttag. "Video RAM Excels at Fast Graphics." *Electronic Design*, August 18, 1983, pp. 161–168.

Pitteway, M.L.V. "Algorithm for Drawing Ellipses or Hyperbolae with a Digital Plotter." *Computer Journal* 10 (November 1967): 24–35.

Porter, T. and T. Duff. "Composing Digital Images." *Computer Graphics*, July 1984, pp. 253–259.

Sproull, R.F. and I.E. Sutherland. "A Clipping Divider." *Fall Joint Computer Conference* Washington, DC: Thompson Books, 1968.

Van Aken, Jerry R. "An Efficient Ellipse-Drawing Algorithm." *IEEE Computer Graphics & Applications* 4 (Sept. 1984): 24–35.

Wientjes, Guttag, and Roskell. "First Graphics Processor Takes Complex Orders to Run Bit-Mapped Displays." *Electronic Design* Vol. 34, No.2 (January 23, 1986): 73–80.

The following TMS34010 documents are available from Texas Instruments. To obtain a copy of any of the TI documents listed below, please call the Texas Instruments Customer Response Center (CRC) at 1-800-232-3200.

- The ***TMS34010 Application Guide*** (literature number SPVA007) is a collection of individual application reports. Each application report discusses a specific TMS34010 application; for example, using a TMS34010 in a 512×512-pixel minimum-chip system, designing TMS34010-based systems that are compatible with various graphics standards, and interfacing the TMS34010 to a variety of host processors.

- The ***TMS34010 Assembly Language Tools User's Guide*** (literature number SPVU004) tells you how to use the TMS34010 assembler, linker, archiver, object format converter, and simulator.

- The ***TMS34010 C Compiler User's Guide*** (literature number SPVU005) tells you how to use the TMS34010 C compiler. This C compiler accepts standard Kernighan and Ritchie C source code and produces TMS34010 assembly language source code. We suggest that you use *The C Programming Language* (written by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall) as a companion to the *TMS34010 C Compiler User's Guide*.

- The ***TMS34010 Math/Graphics Function Library User's Guide*** (literature number SPVS006) describes a collection of mathematics and graphics functions that can be called from C programs.

- The ***TMS34010 Software Development Board User's Guide*** (literature number SPVU002) describes using the TMS34010 software development board (a high-performance, PC-based graphics card) for testing and developing TMS34010-based graphics systems.

- The ***TMS34010 Software Development Board Schematics*** (literature number SPVU003) is a companion to the *TMS34010 Software Development Board User's Guide*.

- The ***TMS34010 Font Library User's Guide*** (literature number SPVU007) describes a set of fonts that are available for use in a TMS34010-based graphics system.

Section 2

Pin Functions

This section discusses the TMS34010 pin functions. Section 2.1 contains a TMS34010 pinout, summarizes the pin functions, and categorizes the signals by function; Section 2.2 through Section 2.6 describe the functional categories.

Topics in this section include:

2.1 Pinout and Pin Descriptions

The TMS34010 is packaged as a 68-pin plastic leaded chip carrier (PLCC). Figure 2-1 shows a pinout of the TMS34010 processor, and Table 2-1 summarizes the pin functions at each interface. Appendix A contains mechanical information.

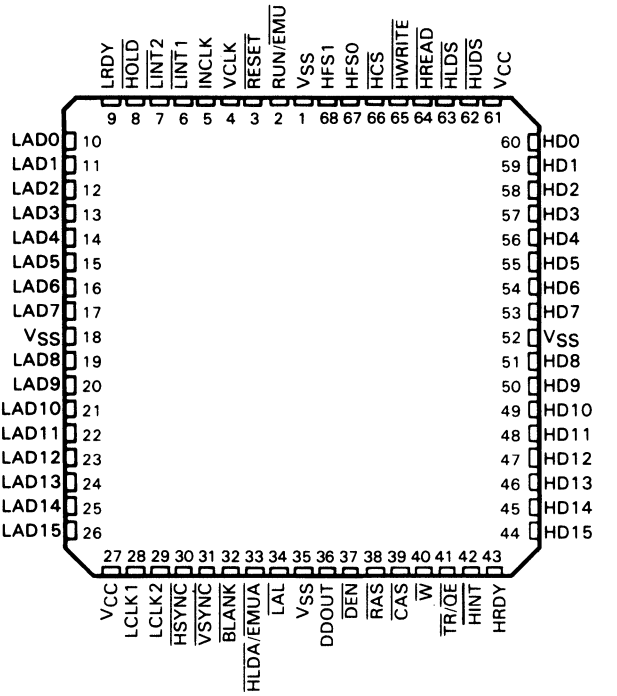


Figure 2-1. TMS34010 Pinout (Top View)

As Figure 2-2 shows, the TMS34010's 68 pins are divided among several interfaces:

Host interface	25 pins
Local memory interface	29 pins
Video timing interface	4 pins
Hold and emulator interfaces	3 pins
Power and reset	7 pins
Total:	68 pins

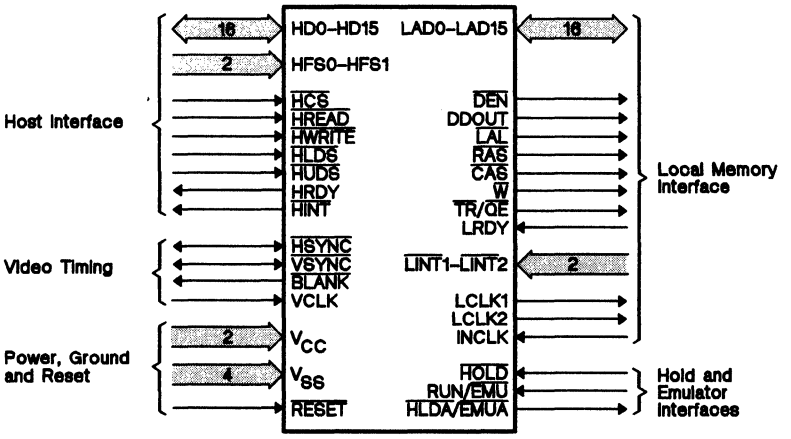


Figure 2-2. TMS34010 Major Interfaces

Table 2-1. Pin Descriptions

Host Interface Bus Pins			
Name	**Pin**	**I/O**	**Description**
$\overline{\text{HCS}}$	66	I	Host chip select
HD0–HD15	44–51,53–60	I/O	Host bidirectional data bus
HFS0,HFS1	67,68	I	Host function select
$\overline{\text{HINT}}$	42	O	Host interrupt request
$\overline{\text{HLDS}}$	63	I	Host lower data select
$\overline{\text{HUDS}}$	62	I	Host upper data select
HRDY	43	O	Host ready
$\overline{\text{HREAD}}$	64	I	Host read strobe
$\overline{\text{HWRITE}}$	65	I	Host write strobe

Table 2-1. Pin Descriptions (Concluded)

Local Interface Bus Pins			
Name	**Pin**	**I/O**	**Description**
$\overline{RAS}$	38	O	Local row-address strobe
$\overline{CAS}$	39	O	Local column-address strobe
DDOUT	36	O	Local data direction out
$\overline{DEN}$	37	O	Local data enable
LAD0–LAD15	10–17,19–26	I/O	Local address/data bus
$\overline{LAL}$	34	O	Local address latched
LCLK1,LCLK2	28,29	O	Local output clocks
$\overline{LINT1}$,$\overline{LINT2}$	6,7	I	Local interrupt request pins
LRDY	9	I	Local ready
$\overline{TR}/\overline{OE}$	41	O	Local shift-register transfer or output enable
$\overline{W}$	40	O	Local write strobe
INCLK	5	I	Input clock
Hold and Emulation			
Name	**Pin**	**I/O**	**Description**
$\overline{HOLD}$	8	I	Hold request
RUN/$\overline{EMU}$	2	I	Run/Emulate
$\overline{HLDA}$/$\overline{EMUA}$	33	O	Hold acknowledge or emulate acknowledge
Video Timing Signals			
Name	**Pin**	**I/O**	**Description**
$\overline{BLANK}$	32	O	Blanking
$\overline{HSYNC}$	30	I/O	Horizontal sync
VCLK	4	I	Video clock
$\overline{VSYNC}$	31	I/O	Vertical sync
Power, Ground, and Reset Signals			
Name	**Pin**	**I/O**	**Description**
$\overline{RESET}$	3	I	Device reset
V_{CC}	27,61	I	Nominal 5-volt power supply
V_{SS}	1,18,35,52	I	Ground

2.2 Host Interface Bus Signals

The host interface pins are used for communication between the TMS34010 and a host processor. Signals output on these pins are assumed to be asynchronous with respect to local clocks LCLK1 and LCLK2. To software running on a host processor, the TMS34010's host interface appears as a peripheral device containing a block of four 16-bit registers. Table 2-2 describes the host interface pins. Section 6 describes the host interface registers, and Section 10 discusses host interface operation.

Table 2-2. Host Interface Signals

Signal	I/O	Description
$\overline{\text{HCS}}$	I	**Host Chip Select.** $\overline{\text{HCS}}$ is driven active low to enable access to the 16-bit host interface register that is selected by HFS0 and HFS1. During the low-to-high transition of $\overline{\text{RESET}}$, the level on the $\overline{\text{HCS}}$ input determines whether the TMS34010 is halted (if $\overline{\text{HCS}}$ is high), or begins immediately executing its reset service routine (if $\overline{\text{HCS}}$ is low). In the second case, the $\overline{\text{HCS}}$ and $\overline{\text{RESET}}$ pins may be tied directly together.
HFS0, HFS1	I	**Host Function Select.** HFS0 and HFS1 determine which of the four 16-bit host interface registers is selected during a read or write cycle that is initiated by the host processor. **HFS1 HFS0 Register Description** 0 0 HSTADRL LSBs of pointer address 0 1 HSTADRH MSBs of pointer address 1 0 HSTDATA Data buffer register 1 1 HSTCTL Control register
$\overline{\text{HREAD}}$	I	**Host Read Strobe.** $\overline{\text{HREAD}}$ is driven active low during a read cycle that is initiated by the host processor. This enables the contents of the selected host interface register to be output on HD0–HD15. $\overline{\text{HREAD}}$ should not be active low at the same time that $\overline{\text{HWRITE}}$ is active low.
$\overline{\text{HWRITE}}$	I	**Host Write Strobe.** $\overline{\text{HWRITE}}$ is driven active low during a write cycle that is initiated by the host processor. This enables the contents of HD0–HD15 to be written to the selected host interface register. $\overline{\text{HWRITE}}$ should not be active low at the same time that $\overline{\text{HREAD}}$ is active low.
$\overline{\text{HLDS}}$	I	**Host Lower Data Select.** $\overline{\text{HLDS}}$ is driven active low during a read or write cycle that is initiated by the host. This enables the lower byte (bits 0–7) of the selected host interface register to be accessed.
$\overline{\text{HUDS}}$	I	**Host Upper Data Select.** $\overline{\text{HUDS}}$ is driven active low during a read or write cycle that is initiated by the host processor. This enables the upper byte (bits 8–15) of the selected host interface register to be accessed.

† In systems that do not use the host interface, it may be desirable to pull these inputs up to the $+V_{CC}$ level.

Table 2-2. Host Interface Signals (Concluded)

Signal	I/O	Description
HRDY	O	**Host Ready**. HRDY indicates when the TMS34010 is ready to complete a read or write cycle that is initiated by the host. Except during an access of a host interface register, HRDY is always high. HRDY is driven low if the host processor attempts to initiate an access of a host interface register before the TMS34010 has had sufficient time to complete all processing resulting from an access initiated previously by the host. HRDY always goes low briefly at the start of a HSTCTL register access. When HRDY is driven low, the host must wait to complete the access until HRDY is again driven high. While $\overline{\text{HCS}}$ is high, HRDY is driven high.
$\overline{\text{HINT}}$	O	**Host Interrupt Request**. $\overline{\text{HINT}}$ follows the INTOUT bit in the HSTCTL register; it is typically used to transmit interrupt requests from the TMS34010 to the host processor. When INTOUT is set to 1 by the TMS34010, $\overline{\text{HINT}}$ is driven active low. $\overline{\text{HINT}}$ remains active low until the host writes a 0 to INTOUT, at which time $\overline{\text{HINT}}$ becomes inactive high.
HD0–HD15	I/O	**Host Bidirectional Data Bus**. The host data pins, HD0–HD15, form a bidirectional 16-bit bus which is used to transfer data between the selected 16-bit host interface register and the host processor. HD0 is the LSB and HD15 is the MSB.

2.3 Local Memory Interface Signals

The TMS34010 uses the local bus interface pins to communicate with external memory and with memory-mapped I/O devices. The signals at this interface are used directly to control DRAMs (dynamic RAMs) and VRAMs (video RAMs). Section 11 discusses local memory interface operation.

Table 2-3. Local Bus Interface Signals

Signal	I/O	Description
$\overline{DEN}$	O	**Local Data Enable.** $\overline{DEN}$ is an active-low output; it drives the active-low output-enable inputs on the bidirectional transceivers (such as the 74ALS245) which are used to buffer data input and output on the LAD0–LAD15 pins. External buffering may be required on the LAD0–LAD15 pins when the TMS34010 is interfaced to a large number of local memory devices.
DDOUT	O	**Local Data Direction Out.** DDOUT drives the direction control inputs on the bidirectional transceivers (such as the 74ALS245) which are used to buffer data input and output on the LAD0–LAD15 pins. External buffering may be required on the LAD0–LAD15 pins when the TMS34010 is interfaced to a large number of local memory devices. During write cycles, DDOUT is driven high to enable data to be output from the LAD0–LAD15 pins while $\overline{DEN}$ is driven active low. During read cycles, DDOUT goes low to enable data to be input to the LAD0–LAD15 pins while $\overline{DEN}$ is driven active low. At all other times, DDOUT remains driven to the default high level.
$\overline{LAL}$	O	**Local Address Latched.** An external latch can use the high-to-low transition of $\overline{LAL}$ to capture the column address from the LAD0–LAD15 pins. When a transparent latch such as a 74ALS373 is used, the address remains latched as long as $\overline{LAL}$ remains active low.
$\overline{RAS}$	O	**Local Row Address Strobe.** The $\overline{RAS}$ output drives the $\overline{RAS}$ inputs of DRAMs and VRAMs.
$\overline{CAS}$	O	**Local Column Address Strobe.** The $\overline{CAS}$ output drives the $\overline{CAS}$ inputs of DRAMs and VRAMs.
$\overline{W}$	O	**Local Write Strobe.** The active-low $\overline{W}$ output drives the $\overline{W}$ inputs of DRAMs and VRAMs. $\overline{W}$ can also be used as the active-low write enable to static memories and other devices connected to the TMS34010 local interface. During a local memory read cycle, $\overline{W}$ remains inactive high while $\overline{CAS}$ is strobed active low. During a local memory write cycle, $\overline{W}$ is strobed active low while $\overline{CAS}$ is low. During shift-register-transfer cycles, the state of $\overline{W}$ indicates whether the transfer is from shift register to memory ($\overline{W}$ is low) or memory to shift register ($\overline{W}$ is high). At all other times, $\overline{W}$ is driven to the default high level.
$\overline{TR}/\overline{QE}$	O	**Local Shift Register Transfer or Output Enable.** This pin connects directly to a VRAM's $\overline{TR}/\overline{QE}$ (or $\overline{DT}/\overline{OE}$) pin. During local memory read cycles, the $\overline{TR}/\overline{QE}$ pin functions as an active-low output enable to gate data from memory to the LAD0–LAD15 pins. During VRAM shift-register-transfer cycles, $\overline{TR}/\overline{QE}$ is driven active low during the high-to-low transition of $\overline{RAS}$.
INCLK	I	**Input Clock.** INCLK is the input clock used to generate the LCLK1 and LCLK2 outputs, to which all processor functions in the TMS34010 are synchronous. A separate input clock, VCLK, controls the video timing registers.

Table 2-3. Local Bus Interface Signals (Concluded)

Signal	I/O	Description
LCLK1,LCLK2	O	**Local Output Clocks.** These two output clocks, 90 degrees out of phase with each other, provide convenient synchronous control of external circuitry to the TMS34010's internal timing. All clocked signals output from the TMS34010, with the exception of the CRT timing signals, are synchronous to these clocks.
LRDY	I	**Local Ready.** LRDY is driven low by external circuitry to inhibit the TMS34010 from completing a local memory cycle it has initiated. While LRDY remains low, the TMS34010 continues to wait. When LRDY is again driven high, the TMS34010 completes the cycle. While LRDY is low, the TMS34010 generates internal wait states in increments of one full LCLK1 cycle in duration. LRDY can be driven low to extend local memory read and write cycles, shift-register-transfer cycles, and DRAM refresh cycles. During internal cycles, the TMS34010 ignores LRDY.
$\overline{\text{LINT1}}$,$\overline{\text{LINT2}}$	I	**Local Interrupt Request Pins.** Interrupt requests from external devices are transmitted to the TMS34010 on the $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$ pins. Each pin activates the request for one of two external interrupt request levels. An external device generates an interrupt request by driving the appropriate interrupt request pin to its active-low state. The pin should remain active low until the TMS34010 has recognized the request. Transitions on the two interrupt request pins are assumed to be asynchronous with respect to local clocks LCLK1 and LCLK2; the signals on these pins are synchronized internally before being used internally.
LAD0–LAD15	I/O	**Local Address/Data Bus.** LAD0–LAD15 form the local multiplexed address/data bus. At the start of a memory cycle, two addresses (row and column) are output on LAD0–LAD15. During a read cycle, data are input on LAD0–LAD15 during the latter part of the cycle. During a write cycle, data are output on LAD0–LAD15 during the latter part of the cycle. LAD0 is the LSB, and LAD15 is the MSB. During the time the row address is output on LAD0–LAD14, status bit $\overline{\text{RF}}$ is output on LAD15. $\overline{\text{RF}}$ is active low at the start of a DRAM-refresh cycle (either $\overline{\text{RAS}}$-only or $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$). During the time that the column address is output on LAD0–LAD13, status bits $\overline{\text{TR}}$ and IAQ are output on LAD15 and LAD14, respectively. IAQ is active high during a read cycle in which the TMS34010 fetches an instruction word from the local memory. During all other cycles, IAQ is inactive low. $\overline{\text{TR}}$ is active low during shift-register-transfer cycles. (The level output on LAD14 during the high-to-low transition of $\overline{\text{CAS}}$ is always the same as the level output on $\overline{\text{TR}}$/$\overline{\text{QE}}$ during the high-to-low transition of $\overline{\text{RAS}}$.)

Notes: 1) The system designer must ensure that LRDY is not held low for so long that the TMS34010 is prevented from performing the necessary number of DRAM refresh cycles or is prevented from refreshing the display by performing a VRAM memory-to-shift-register cycle during horizontal retrace.

2) The operation of $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$ is affected by the RUN/$\overline{\text{EMU}}$ pin. Make sure this pin is in the proper state.

2.4 Video Timing Signals

The video timing signals ($\overline{\text{BLANK}}$, $\overline{\text{HSYNC}}$, and $\overline{\text{VSYNC}}$) control the horizontal and vertical sweep rates of the video monitor. They also synchronize the display on the monitor to video data that is output from the VRAMs. Section 9 discusses video timing and screen refresh operations.

Table 2-4. Video Timing Signals

Signal	I/O	Description
$\overline{\text{HSYNC}}$	I/O	**Horizontal Sync**. $\overline{\text{HSYNC}}$ is the horizontal sync signal used to control external video circuitry. It is programmed as either an input or an output by means of two control bits in the DPYCTL register. When configured as an output, the active-low horizontal sync signal is generated by the TMS34010's on-chip video timers. When configured as an input, the TMS34010 synchronizes its video timers to externally-generated horizontal sync pulses. Immediately following reset, $\overline{\text{HSYNC}}$ is configured as an input.
$\overline{\text{VSYNC}}$	I/O	**Vertical Sync**. $\overline{\text{VSYNC}}$ is the vertical sync signal used to control external video circuitry. It is programmed as either an input or an output by means of a control bit in the DPYCTL register. When configured as an output, the active-low vertical sync signal is generated by the TMS34010's on-chip video timers. When configured as an input, the TMS34010 synchronizes its video timers to externally-generated vertical sync pulses. Immediately following reset, $\overline{\text{VSYNC}}$ is configured as an input.
$\overline{\text{BLANK}}$	O	**Blanking**. $\overline{\text{BLANK}}$ is a composite blanking signal used to turn off the electron beam of a CRT during both horizontal and vertical retrace intervals. This signal may also be used to control the starting and stopping of the VRAM shift registers.
VCLK	I	**Video Clock**. VCLK is derived from the dot clock of the external video system and is used internally to drive the TMS34010's video timing logic. The signals output at the $\overline{\text{BLANK}}$, $\overline{\text{HSYNC}}$, and $\overline{\text{VSYNC}}$ pins are synchronous to VCLK. VCLK is not required to have any timing relationship with respect to INCLK; that is, VCLK and INCLK can be asynchronous. In order to read HCOUNT and VCOUNT registers reliably, VCLK should be held high during the read. In systems which do not use the video timing registers or require automatic screen refreshing, VCLK can be strapped high.

Note: The operation of $\overline{\text{HSYNC}}$ and $\overline{\text{VSYNC}}$ is affected by the RUN/$\overline{\text{EMU}}$ pin. Make sure this pin is in the proper state.

2.5 Hold and Emulator Interface Signals

The TMS34010 hold interface permits other devices to request and be granted control of the local interface bus.

The emulator interface is used to control the TMS34010 when it is used for emulation. The RUN/$\overline{\text{EMU}}$ pin may remain unconnected in nonemulation applications.

Table 2-5. Hold and Emulator Interface Signals

Signal	I/O	Description
$\overline{\text{HOLD}}$	I	**Hold Request**. The $\overline{\text{HOLD}}$ pin is driven active low by an external device to signal a request that the TMS34010 release ownership of the local memory bus. Once the TMS34010 has acknowledged the hold request via a hold acknowledge signal, the external device assumes ownership of the bus. The device must continue to assert its hold request until it has released the bus.
$\overline{\text{HLDA}}$/$\overline{\text{EMUA}}$	O	**Hold Acknowledge and Emulate Acknowledge**. The $\overline{\text{HLDA}}$/$\overline{\text{EMUA}}$ pin is multiplexed between two functions: (1) acknowledgment of hold requests and (2) acknowledgment of emulation requests. The hold acknowledge signal ($\overline{\text{HLDA}}$) is output during phases Q3 and Q4 of the local clock cycle. The emulate acknowledge signal ($\overline{\text{EMUA}}$) is output during phases Q1 and Q2. $\overline{\text{HLDA}}$ is driven active low in response to a hold request from an external device, but not until the TMS34010 has released the bus to the requesting device. The device must delay taking possession of the bus until it has received an active $\overline{\text{HLDA}}$ signal. Once an active-low hold acknowledge signal has been transmitted during Q3–Q4, it will continue to be transmitted during Q3–Q4 of each local clock period until the external device ceases to assert its hold request. $\overline{\text{EMUA}}$ is driven active low to indicate to external circuitry that the TMS34010 has halted in response to an $\overline{\text{EMU}}$ command input on the RUN/$\overline{\text{EMU}}$ pin. $\overline{\text{HLDA}}$/$\overline{\text{EMUA}}$ is also driven low when an EMU opcode is executed by the TMS34010, but only during phases Q1 and Q2 of a single LCLK1 cycle. Execution of an EMU opcode causes an active-low signal to be output at the $\overline{\text{HLDA}}$/$\overline{\text{EMUA}}$ pin during phases Q1 and Q2, so external devices that generate hold requests should avoid interpreting these signals as hold acknowledgment.
RUN/$\overline{\text{EMU}}$	I	**Run/Emulate**. This pin is defined as a no-connect during normal system operation. The RUN/$\overline{\text{EMU}}$ pin should **not** be pulled low except during factor testing or chip emulation. An internal pull-up load permits RUN/$\overline{\text{EMU}}$ to remain unconnected during normal use. If RUN/$\overline{\text{EMU}}$ is pulled low, $\overline{\text{RESET}}$, $\overline{\text{LINT1}}$, $\overline{\text{LINT2}}$, $\overline{\text{HSYNC}}$, and $\overline{\text{VSYNC}}$ are reconfigured to perform special functions used only during emulation and factory testing.

2.6 Power, Ground, and Reset Signals

Six TMS34010 pins are dedicated to ground and power supply. Section 8 provides more details about $\overline{\text{RESET}}$.

Table 2-6. Power, Ground, and Reset Signals

Signal	I/O	Description
V_{CC}	I	V_{CC} (2 pins). Two +5-volt power supply inputs.
V_{SS}	I	V_{SS} (4 pins). Four electrical ground inputs.
$\overline{\text{RESET}}$	I	**Reset**. $\overline{\text{RESET}}$ is pulled low to reset the device during normal operation. While $\overline{\text{RESET}}$ is asserted low, the internal registers of the TMS34010 are set to an initial known state, and all output and bidirectional pins are driven either to inactive levels or to high impedance. The behavior of the TMS34010 chip following reset depends on the level of the $\overline{\text{HCS}}$ input just prior to the low-to-high transition of $\overline{\text{RESET}}$. If $\overline{\text{HCS}}$ is low, the TMS34010 begins executing the instructions pointed to by the reset vector. If $\overline{\text{HCS}}$ is high, the TMS34010 is halted until a host processor writes a 0 to the HLT bit in the HSTCTL register. Transitions on the $\overline{\text{RESET}}$ pin are assumed to be asynchronous with respect to local clocks LCLK1 and LCLK2; the signal input on this pin is synchronized internally before it is used internally.

Section 3

Memory Organization

This section presents details of physical and logical addresses, illustrates the TMS34010 memory map, and describes stack operation.

3.1 Memory Addressing

The TMS34010 is a bit-addressable machine with a 32-bit internal memory address. The total memory capacity is four gigabits (or 512 megabytes); the TMS34010 supports external addressing of 128 megabytes.

Memory is accessed as a continuously addressable string of bits. Each 32-bit address points to an individual bit within memory. Groups of adjacent bits form data structures called **fields**. A field is specified by its starting bit address and its length. The TMS34010 supports field lengths from 1 to 32 bits. Bit addresses range from 00000000h to 0FFFFFFFFh.

Figure 3-1 illustrates the logical memory structure.

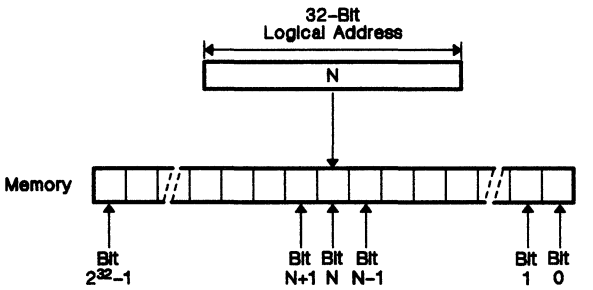


Figure 3-1. Logical Memory Address Space

Figure 3-2 illustrates physical memory organization. The TMS34010 communicates with memory over a 16-bit data bus, and always reads or writes a complete 16-bit word from or to memory. A word accessed during a memory cycle always begins on an even 16-bit boundary; thus, the four LSBs of the 32-bit starting address of the word are 0s. Bits within a word are numbered from 0 to 15; bit 15 is the MSB and bit 0 is the LSB. A word is identified by the address of its LSB. In this document, the LSB of a memory word is depicted as the rightmost bit in the word.

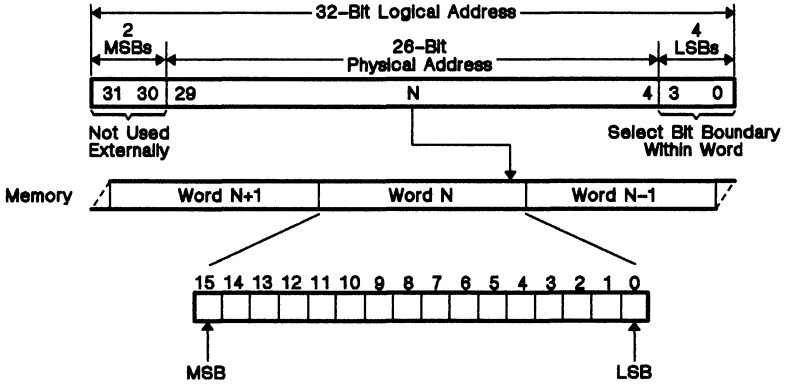


Figure 3-2. Physical Memory Addressing

The four LSBs of the 32-bit logical address in Figure 3-2 do not appear on the local memory bus. When the TMS34010 extracts a data structure that does not begin and end on even word boundaries, these four LSBs are used internally to indicate a bit boundary within an accessed word. Control logic at the local memory interface automatically performs the bit alignment and masking necessary to extract a data structure from physical memory; this is completely transparent to software. If the data structure being extracted straddles word boundaries, multiple read cycles are required. Similarly, inserting a data structure into memory may require a series of read and write cycles, accompanied by the internal masking and shifting of data to properly align the data structure within memory. The memory-control logic performs these tasks automatically.

The two MSBs of the 32-bit logical address are not output. The TMS34010 supports an external address range of 128 megabytes of physical memory.

3.2 Memory Map

Figure 3-3 illustrates the TMS34010 memory map. Memory is logically organized as four gigabits, but is physically accessed 16 bits at a time. Locations are shown as 16-bit words, identified by 32-bit addresses whose four LSBs are 0s. Word addresses range from 00000000h to FFFFFFF0h (bit address 00000000h is the rightmost bit in the word at the bottom of Figure 3-3, and bit address FFFFFFFFh is the leftmost bit in the word at the top.) Reading or writing to an address in the range C0000000h to C00001F0h accesses an internal I/O register. Reading or writing to any address outside this range accesses off-chip memory (or a memory-mapped device) external to the TMS34010.

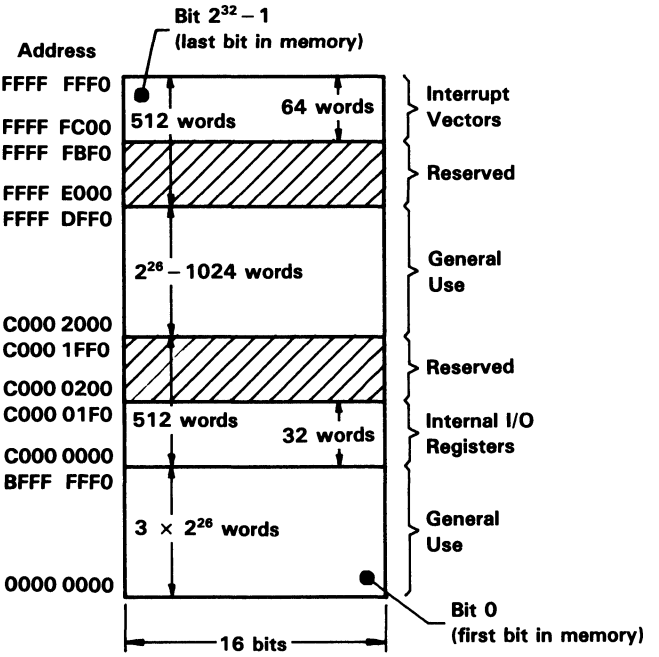


Figure 3-3. TMS34010 Memory Map

As Figure 3-3 shows, memory is divided into several regions:

- **General use**

  Addresses ranges 0h–BFFFFFF0h and C0002000h–FFFFDFF0h are for general use (executable code, data tables, etc.).

- **I/O registers**

  Addresses C0000000h–C00001F0h are reserved for the 16-bit I/O registers. Section 6 discusses the I/O registers; it contains a map of this

memory area which associates each I/O register with the appropriate address.

- **Interrupt, Reset, and Trap Vectors**

  Addresses FFFFFC00h–FFFFFFE0h are reserved for 32 interrupt, reset, and trap vectors. A vector is a 32-bit address that points to the starting location in memory of the appropriate interrupt, reset, or trap service routine. Each address is stored in physical memory as two consecutive 16-bit words, with the 16 LSBs at the lower address. Section 8 contains more information about interrupts and traps.

- **Reserved memory**

  Addresses C0000200h–C0001FF0h are reserved for future expansion of the I/O registers.

  Addresses FFFFE000h–FFFFFBF0h are reserved for future expansion of the interrupt vectors.

3.3 Stacks

The TMS34010's system stack is implemented in local memory and managed in hardware. The stack is used to store return addresses and processor status information during interrupts, traps, and subroutine calls. The contents of general-purpose registers can be pushed onto the stack and popped off the stack. The system stack can also be used for dynamically allocated data storage.

The stack is accessed through a dedicated 32-bit internal register, called the *stack pointer*, or **SP**. The SP points to the top of the system stack; it can be accessed as register 15 in either register file.

In addition to the system stack, you can define your own auxiliary stacks. The system stack always grows toward lower memory addresses; an auxiliary stack can be defined to grow toward either lower or higher addresses. The MOVE and MOVB instructions, combined with the automatic predecrement and postincrement addressing modes, facilitate pushing and popping auxiliary stack data. One or more registers in the A or B files can be used by software as auxiliary stack pointers and frame pointers. The indexed addressing modes can be used in conjunction with a frame pointer to access variables embedded within the stack.

3.3.1 System Stack

Figure 3-4 shows the structure of the system stack, which grows in the direction of lower memory addresses.

The SP points to the top of the stack; it contains the 32-bit address of the LSB (bit 0) of the value on top of the stack. The SP can contain any 32-bit address; however, stack operations execute more efficiently when the four LSBs of the SP are 0s. This aligns the SP to word boundaries in memory, reducing the number of memory cycles necessary to push values onto the stack or pop values off the stack.

Any instruction that manipulates general-purpose registers (A0–A14 or B0–B14) can also be used to manipulate the SP. The SP can be specified as the source or destination operand in any instruction that operates on the general-purpose registers. Instructions that manipulate the SP include:

Instructions that Push Values on the Stack	**Instructions that Pop Values from the Stack**
MMTM SP, *register list*	MMFM SP, *register list*
CALL *Rs*	RETI
CALLA *absolute address*	RETS
CALLR *relative address*	POPST
TRAP *number*	MOVE *SP+, *Rd*
PUSHST	
MOVE *Rs*, -*SP	

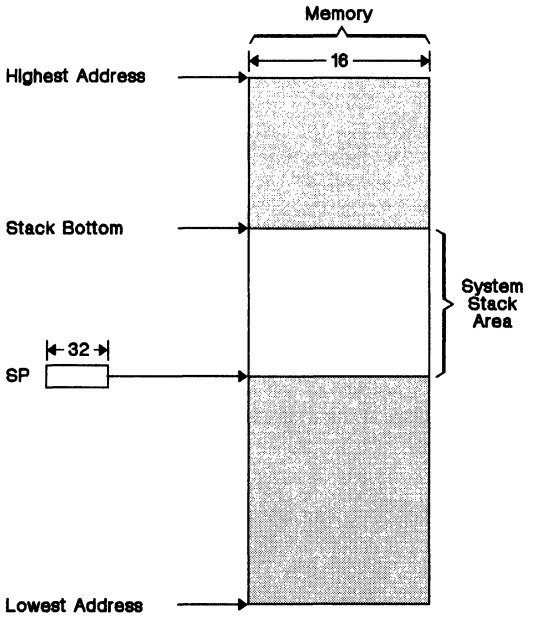


Figure 3-4. System Stack

3.3.1.1 Saving Registers on the System Stack

Register information can be stored on the stack during an interrupt or a subroutine call. This frees up the register for use by an interrupt routine or a subroutine, and allows you to restore the original register values from the stack when the routine finishes executing.

During an interrupt, the contents of the PC and ST are automatically saved on the stack; if you want to save values that are in general-purpose registers, you can use the MMTM and MMFM instructions. MMTM pushes multiple general-purpose registers onto the stack, and MMFM pops multiple general-purpose registers from the stack.

When the contents of a 32-bit register are pushed onto the stack, they are stored in two consecutive 16-bit words. The 16 MSBs are stored at the higher memory address, and the 16 LSBs are stored at the lower address. This is shown in Figure 3-5, which demonstrates the effects of the following instruction sequence:

```
MMTM SP, A0    ; Push register A0 onto stack
MMFM SP, A1    ; Pop stack into register A1
```

- Figure 3-5 *a* shows the original state of the stack and registers.
- Figure 3-5 *b* illustrates the state after A0 is pushed onto the stack.
- Figure 3-5 *c* shows the result of popping the top of the stack into A1.

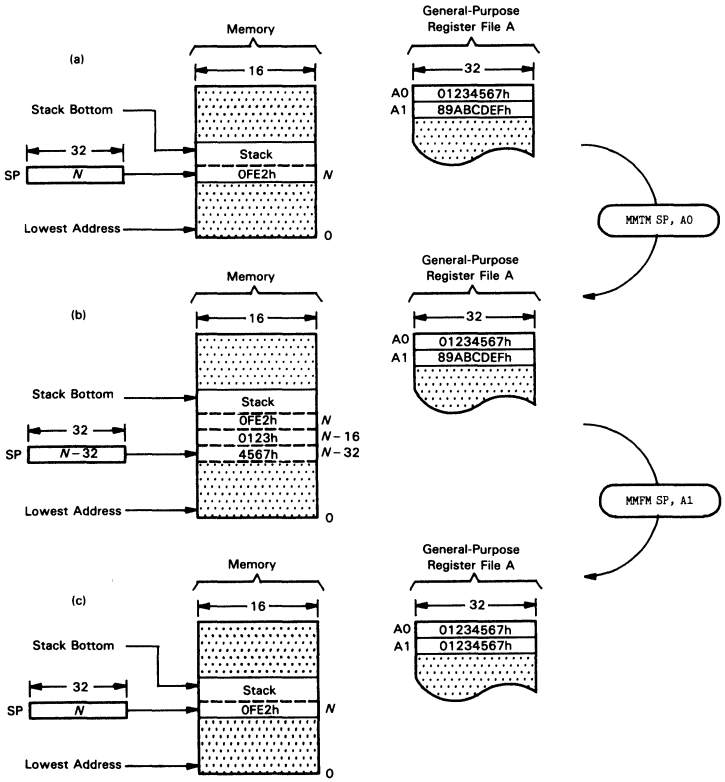


Figure 3-5. Stack Operations

The TMS34010 performs two steps to push the contents of a 32-bit register onto the top of the stack:

1) Decrement the PC by 32.
2) Push the register contents onto the stack.

The TMS34010 performs two steps to pop the top of stack into a 32-bit register:

1) Pop the 32 bits at the top of the stack into the register.
2) Increment the SP by 32.

3.3.1.2 Saving Information On the System Stack During an Interrupt

During an interrupt, the TMS34010 pushes the PC and ST onto the stack; this allows the interrupted routine to resume execution when the interrupt processing is completed. An interrupt routine performs the following actions:

1) Decrement the SP by 32.
2) Push the PC onto the stack.
3) Decrement the SP again by 32.
4) Push the ST onto the stack.

During a return from an interrupt:

1) Pop the 32 bits at the top of the stack into the ST.
2) Increment the SP by 32.
3) Pop the 32 bits at the top of the stack into the PC.
4) Increment the SP again by 32.

3.3.1.3 Saving Information On the System Stack During a Subroutine Call

A subroutine call saves the state of the calling routine on the stack; this allows the routine to resume execution when the subroutine completes. A subroutine call performs the following actions:

1) Decrement the SP by 32.
2) Push the PC onto the stack.

During a return from a subroutine:

1) Pop the 32 bits at the top of the stack into the PC.
2) Increment the SP by 32.

3.3.2 Auxiliary Stacks

Auxiliary stacks can be managed in software. Any A- or B-file register, except the SP, can be used as the auxiliary stack pointer. Auxiliary stacks are typically used to contain dynamically allocated data storage.

In the following discussion, *STK* represents the auxiliary stack pointer. STK is a symbol that must be equated to one of the general-purpose registers; for example:

```
STK     .set    A0
```

The STK may contain any 32-bit value; however, stack operations execute more efficiently when the four LSBs of the STK are 0s. This aligns the STK to word boundaries in memory, reducing the number of memory cycles necessary to push values onto the stack or pop values off the stack.

As Figure 3-6 and Figure 3-7 show, the auxiliary stack can be configured to grow in either direction in memory. The memory is shown in these figures as a string of continuously addressable bits.

3.3.2.1 An Auxiliary Stack that Grows Toward Lower Addresses

Figure 3-6 shows a stack that grows toward lower memory addresses:

- Figure 3-6 *a* shows the original stack.
- In Figure 3-6 *b*, a field of arbitrary size is pushed onto the stack with this instruction:

  `MOVE` *Rs*, `*-STK`

  (*Rs* and STK represent general-purpose registers.)

- In Figure 3-6 *c*, the field is popped off the stack with this instruction:

  `MOVE *STK-,` *Rd*

  (*Rd* and STK represent general-purpose registers.)

Between instructions, the STK always points to the lowest bit address in the stack – this corresponds to the very top of the stack. You can use the **MMTM STK**,*register list* instruction to save multiple registers on the stack in Figure 3-6. Later, you can restore the registers to their former values with an **MMFM STK**,*register list* instruction.

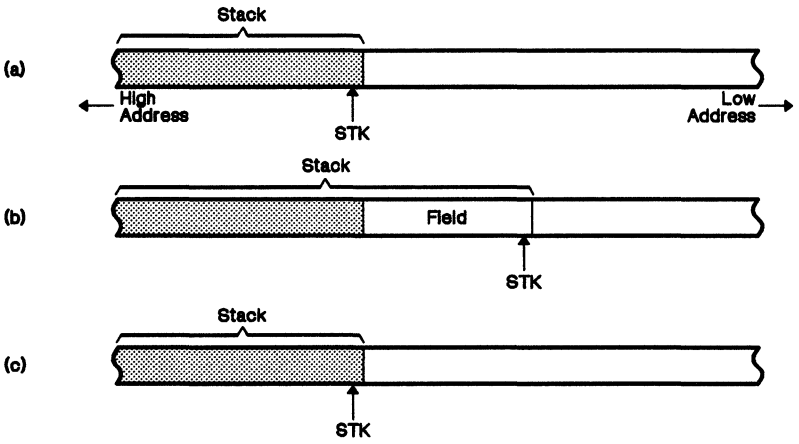


Figure 3-6. An Auxiliary Stack that Grows Toward Lower Addresses

3.3.2.2 An Auxiliary Stack that Grows Toward Higher Addresses

Figure 3-7 shows a stack that grows toward higher memory addresses:

- Figure 3-7 *a* shows the original stack.
- In Figure 3-7 *b*, a field of arbitrary size is pushed onto the stack using the following instruction:

  `MOVE` *Rs*, `*STK+`

- In Figure 3-7 *c*, the field is popped off the stack with this instruction:

  `MOVE *-STK,` *Rd*

Between instructions, the STK always points to one plus the highest bit address in the stack – this location is one bit beyond the very top of the stack.

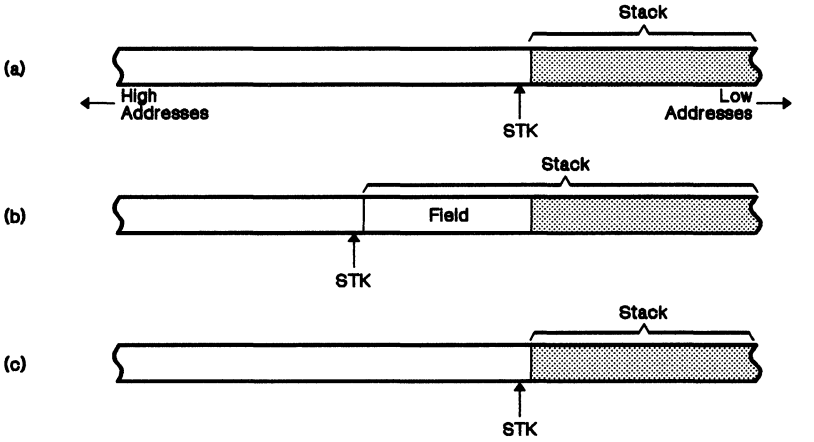


Figure 3-7. An Auxiliary Stack that Grows Toward Higher Addresses

Section 4

Hardware-Supported Data Structures

The TMS34010 supports several data structures at the machine level:

- **Fields** are configurable data structures whose length can be defined within the range 1 to 32 bits. Two field sizes can be defined simultaneously. A field can begin and end at arbitrary bit addresses.

- **Bytes** are a special case of field in which the field length is fixed at eight bits and is sign extended. Bytes can begin on any bit boundary within a word.

- **Pixels** are configurable data structures; pixel length can be programmed to be 1, 2, 4, 8, or 16 bits (always a power of two). Pixels are aligned so that they do not cross word boundaries in memory.

- Two-dimensional **pixel arrays**, or pixel blocks, are rectangular groups of pixels that are manipulated using the PIXBLT (pixel block transfer) and FILL (pixel block fill) instructions. A pixel array can be moved from one area of memory to another in a single PixBlt operation. It can be combined with another array of the same size by performing Boolean or arithmetic operations on the corresponding pixels of the two arrays.

The number of bits in a pixel, field, or array is programmable, but byte length is fixed. Two field sizes and one pixel size can be specified simultaneously. The size and starting addresses of the pixel arrays that are manipulated during a PixBlt operation are specified by the values loaded into dedicated hardware registers.

Topics in this section include:

4.1 Fields

The TMS34010 supports two software-configurable field types, *field 0* and *field 1*. A field in memory is defined by two parameters:

- Starting address **and**
- Field size (1 to 32 bits)

A field's starting address is the address of the field's LSB. A field can begin at an arbitrary bit address in memory. When a field is moved from memory to a general-purpose register, the field is right justified within the register; that is, the field's LSB coincides with the register's rightmost bit (bit 0). The register bits to the left of the field are all 1s or all 0s, depending on the values of both the appropriate FE (field extension) bit in the status register, and the sign bit (MSB) of the field. If FE=1, the field is sign extended; if FE=0, the field is zero extended.

Field size can range from 1 to 32 bits. The lengths of fields 0 and 1 are defined by two 5-bit fields in the status register, FS0 and FS1.

Figure 4-1 illustrates a field in memory. In this example, the field straddles the boundary between words *N* and *N+1* in memory. Field extraction and insertion is performed by on-chip hardware:

- To move the field to a general-purpose register, the TMS34010 extracts the field from memory by reading word *N* and word *N+1* in separate cycles.

- To move the field from a general-purpose register, the TMS34010 inserts the field into memory by reading and writing word *N*, and reading and writing word *N+1*.

The memory operations necessary to insert or extract a field are performed automatically by special hardware, and are transparent to software.

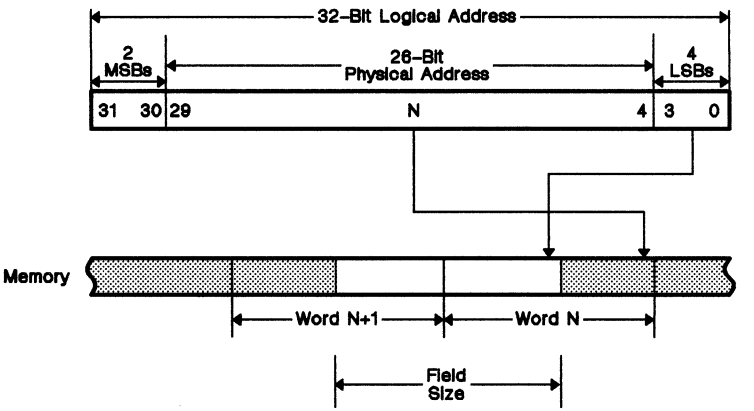


Figure 4-1. Field Storage in External Memory

In Figure 4-1, word *N* is pointed to by a 26-bit physical address output by the TMS34010 to memory. This 26-bit address corresponds to bits 4–29 of the field's 32-bit logical address. The four LSBs of the logical address point to the beginning of the field within word *N*.

The number of memory cycles required to extract or insert a field depends on how the field is aligned in memory. Field manipulation is more rapid when fields are stored in memory so that they do not cross word boundaries. Figure 4-2 illustrates various cases of alignment and nonalignment of fields to word boundaries in memory. Given a field starting address and field length, the memory controller will recognize the specified field alignment as one of the seven cases in Figure 4-2. Field extraction and field insertion are performed in a manner that requires the minimum number of memory cycles.

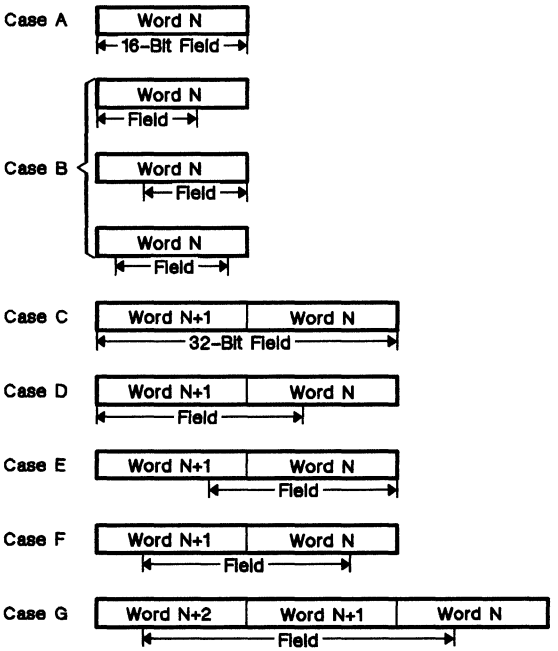


Figure 4-2. Field Alignment in Memory

Case A A 16-bit field is aligned on word boundaries. Field extraction requires a single read cycle, and field insertion requires a single write cycle.

Cases B1–B3 The field length is less than 16 bits.

- In **Case B1**, the field starting address is not aligned to a word boundary, although the end of the field coincides with the end of the word.
- In **Case B2**, the field starting address is aligned to a word boundary, but the end of the field does not coincide with the end of the word.
- In **Case B3**, the field length is 14 bits or less, and neither the start nor the end of the field is aligned to a word boundary.

For Cases B1–B3, a field extraction requires a single read cycle. A field insertion requires the following sequence of memory cycles:

1) Read word *N*
2) Write word *N*

Case C A 32-bit field is aligned on word boundaries. A field extraction requires the following sequence of memory cycles:

1) Read word *N*
2) Read word *N+1*

A field insertion requires the following sequence of memory cycles:

1) Write word *N*
2) Write word *N+1*

Case D The field size is greater than 16 bits. The field starting address is not aligned to a word boundary, but the end of the field coincides with the end of the word. A field extraction requires the following sequence of memory cycles:

1) Read word *N*
2) Read word *N+1*

A field insertion requires the following sequence of memory cycles:

1) Read word *N*
2) Write word *N*
3) Write word *N+1*

Case E The field size is greater than 16 bits. The field starting address is aligned to a word boundary, but the end of the field does not coincide with the end of the word. A field extraction requires the following sequence of memory cycles:

1) Read word *N*
2) Read word *N+1*

A field insertion requires the following sequence of memory cycles:

1) Write word *N*
2) Read word *N+1*
3) Write word *N+1*

Case F The field straddles the boundary between two words. Neither the start nor the end of the field is aligned to a word boundary. A field extraction requires the following sequence of memory cycles:

1) Read word *N*
2) Read word *N+1*

A field insertion requires the following sequence of memory cycles:

1) Read word *N*
2) Write word *N*
3) Read word *N+1*
4) Write word *N+1*

Case G The field size ranges from 18 to 32 bits, and the field straddles two word boundaries. Neither the start nor the end of the field is aligned to a word boundary. A field extraction requires the following sequence of memory cycles:

1) Read word *N*
2) Read word *N+1*
3) Read word *N+2*

A field insertion requires the following sequence of memory cycles:

1) Read word *N*
2) Write word *N*
3) Write word *N+1*
4) Read word *N+2*
5) Write word *N+2*

A field insertion modifies only the portion of a word that lies within a field. The TMS34010 memory controller must perform a read-modify-write operation when a field that does not begin and end on even 16-bit word boundaries is to be written to memory. This occurs when the four LSBs of the address are not 0, or when the specified field size is a value other than 16 or 32. The memory controller uses these two parameters (address LSBs and field size) to produce a mask that identifies the bits in the word corresponding to the field. Hardware uses the mask to perform the read-modify-write cycle. The TMS34010's local memory control logic automatically generates the mask and executes the read-modify-write operation; this is transparent to software.

Figure 4-3 shows an example of inserting a 5-bit field stored in a register to logical address 00000008h.

- In Figure 4-3 *a*, the field to be inserted is shown right-justified in the 16 LSBs of the designated general-purpose register.
- In *b*, memory controller hardware has rotated the field to align it with the destination in memory.
- In *c*, the TMS34010 reads the original word from the destination in memory.
- In *d*, the mask is generated to designate the bits to be modified.
- In *e*, the field is inserted into the word from memory, and the result is written back to the destination address in memory.

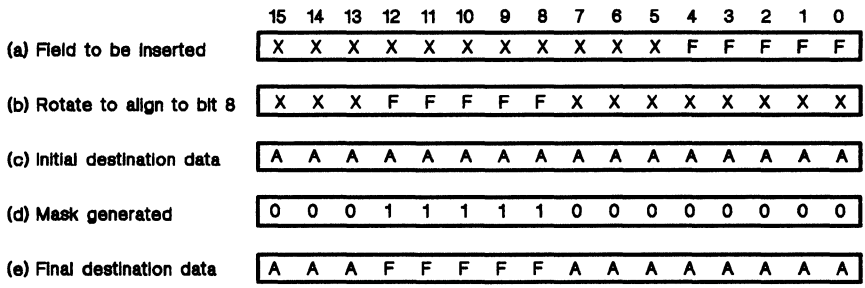


Figure 4-3. Field Insertion

In the more complex case in which a field straddles one or two word boundaries in memory, the portion of the field lying within each word is inserted into that word using the methods described above.

4.2 Pixels

The term pixel has two meanings in the context of a TMS34010-based graphics system. Outside the TMS34010, a pixel is a picture element on a display surface. Inside the TMS34010, a logical pixel is a software-configurable data structure supported by the TMS34010 instruction set. The logical pixel data structure in TMS34010 memory contains the information needed to specify the attributes of a picture element visible on a screen. The information for a horizontal line of pixels on the screen is usually stored in consecutive words in memory.

4.2.1 Pixels in Memory

Within TMS34010 memory, the pixel data structure is defined by two parameters:

- Starting address **and**
- Pixel size

A pixel's starting address is the address of the LSB of the pixel.

Pixel size (the number of bits per pixel) is defined in the PSIZE register. A pixel can be 1, 2, 4, 8, or 16 bits long. The TMS34010 treats pixels as a special case of a field in which the field size is constrained to be a power of two. However, pixels do not cross word boundaries within memory; they are aligned within memory so that an integral number of pixels is contained within the boundaries of a memory word. For example, a 2-bit pixel should begin at an even bit address whose LSB is 0, a 4-bit pixel should begin at a bit address whose two LSBs are 0s, and so forth.

When a pixel is moved from memory to a general-purpose register, the pixel is right justified within the register. That is, the LSB of the pixel coincides with the rightmost bit (bit 0) of the register. Register bits to the left of the pixel are loaded with 0s.

Figure 4-4 illustrates pixel storage in memory. The pixel is located within the word pointed to by the 26-bit physical address corresponding to bits 4–29 of the 32-bit logical address of the pixel. The four LSBs of the logical address specify the displacement of the pixel within the word. When the pixel length is less than 16, each word contains two or more pixels.

Pixel extraction and insertion is performed by on-chip hardware in a manner that requires the minimum number of memory cycles. (The operations are transparent to the programmer.) In the worst case, two memory cycles (a read followed by a write) are required to insert a pixel of less than 16 bits. Inserting a 16-bit pixel requires a single write cycle, and extracting a pixel (1 to 16 bits) requires a single read cycle.

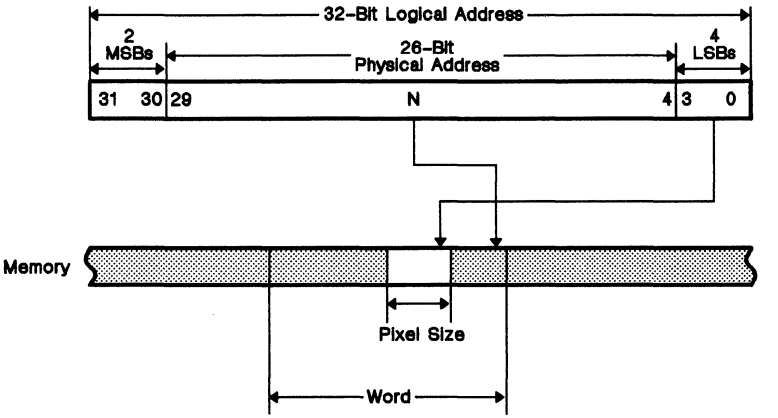


Figure 4-4. Pixel Storage in External Memory

4.2.2 Pixels on the Screen

Figure 4-5 illustrates the mapping of pixels from memory to a display screen. The screen refresh function outputs pixels in the sequence of ascending pixel addresses. However, the electron beam sweeps from the left edge of the screen to the right edge during each horizontal scan interval, so pixels appear on the screen in the opposite order of their representation in memory. That is, the least significant pixel (in terms of bit address) appears on the left, and the most significant pixel appears on the right.

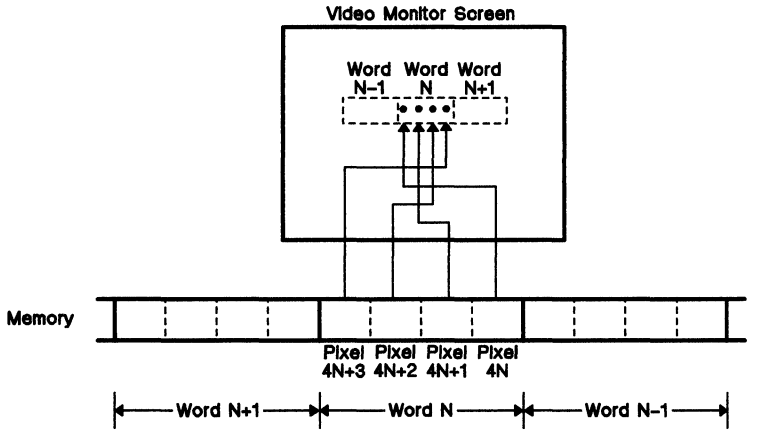


Figure 4-5. Mapping of Pixels to Monitor Screen

The TMS34010 allows a pixel to be identified either in terms of its XY coordinates on the screen, or in terms of the address of the logical pixel in memory. These two methods are called **XY addressing** and **linear addressing**, respectively.

When XY addressing is used, the origin can be selected to lie in either the upper left or lower left corner of the screen. The position of the origin is controlled by the ORG bit in the DPYCTL register. Figure 4-6 *a* illustrates the default coordinate system (ORG=0), in which the origin of the two coordinate axes is located in the upper left corner of the screen. Figure 4-6 *b* shows the alternate coordinate system (ORG=1) in which the origin is located in the lower left corner of the screen.

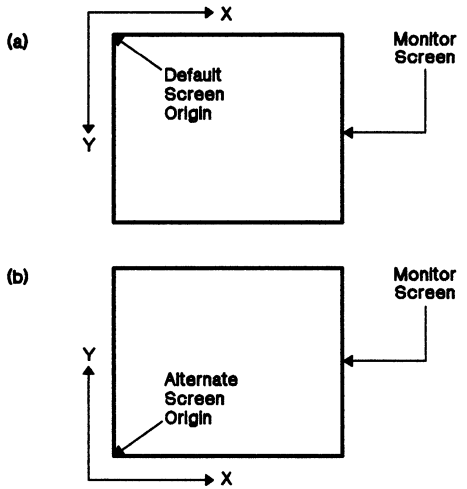


Figure 4-6. Configurable Screen Origin

Using the default screen origin, Figure 4-7 illustrates the mapping of pixels from memory to the screen. In Figure 4-7, horizontal movement represents travel in the X direction on the screen. Vertical movement represents travel in the Y direction. The depth of the buffer represents the pixel size. The "on-screen memory" contains the pixels that appear on the screen.

The display memory shown in Figure 4-7 is shown in terms of a "screen format" rather than the "memory format" used in the memory map shown in Figure 3-3 on page 3-4. The screen format places the lowest pixel address at the upper left corner of the memory map. This is the same relative orientation in which pixels appear on the screen. Compare this to the memory format shown in Figure 3-3, which places the lowest bit address at the lower right corner of the memory map. This convention is frequently used in industry to represent the relative location of addresses in memory. In this document, assume the standard memory format is used unless the screen format is indicated.

Figure 4-8 illustrates the mapping of XY coordinates to the on-screen memory. For simplicity, assume that the screen origin coincides with the upper left

corner of the display memory. *P* represents the X extent of the display memory and *N* represents the Y extent. Each box represents a pixel within the memory. The number within the box represents the pixel's memory location, relative to the beginning of the on-screen memory. The number in the box is multiplied by the number of bits per pixel to produce the address offset of the pixel from the start of the display memory. Since the pixel size is constrained to be a power of two, the multiply can be replaced by a simple shift operation.

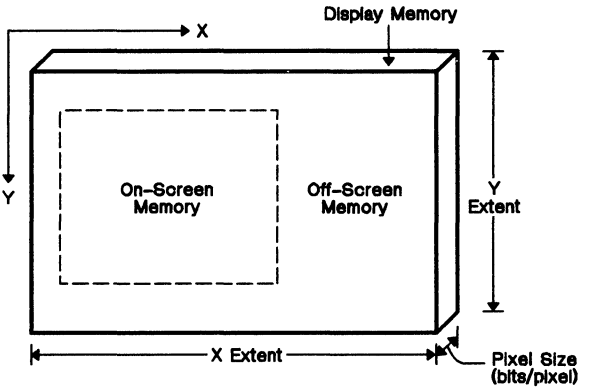
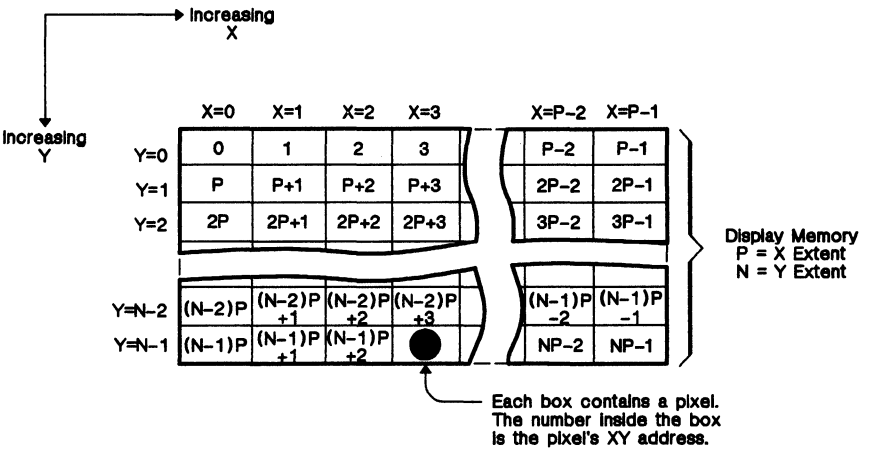


Figure 4-7. Display Memory Dimensions



Display Pitch = (X extent) × (pixel size)
= Differences in 32-bit memory addresses of two vertically adjacent pixels

Figure 4-8. Display Memory Coordinates

4.2.3 Display Pitch

The term *display pitch* refers to the difference in memory addresses between two pixels that appear in vertically adjacent positions (one directly above the other) on the screen. In Figure 4-8, the pitch is calculated as *P* times the pixel size, where *P* is the X extent of the display memory.

The display pitch must be a power of two in order to support XY addressing of pixels on the screen. Linear addressing of pixels on the screen imposes fewer restrictions. In particular, the display pitch for linear addressing may be any value that is a multiple of 16; that is, the four LSBs of the address must be 0s. Features such as automatic window checking are available with XY addressing, but are not available with linear addressing.

The pitch of a pixel array is the difference in memory addresses of two vertically adjacent pixels in the array. If the array occupies a rectangular area of the screen, the array pitch is the same as the display pitch.

During a pixel operation such as a PixBlt, the source and destination array pitches are specified in separate dedicated hardware registers. This facilitates the transfer of pixel arrays between on-screen and off-screen memory, which may have different pitches.

A sample display pitch calculation is shown below. In this example, the pixel size is four bits and the X extent of the pixel display is 640 pixels. However, since XY addressing and windowing are to be used, the physical memory is organized so that there are 1024 pixels between successive scan lines. Thus, the X extent of physical display memory is 1024, and the display pitch is:

$$\begin{aligned} \text{Display Pitch} &= (1024 \text{ pixels/line}) \times (4 \text{ bits/pixel}) \\ &= 4096 \text{ (which is } 2^{12}) \end{aligned}$$

4.3 XY Addressing

The TMS34010 allows pixel addresses to be specified in terms of two-dimensional XY coordinates that correspond to locations on the screen. This is referred to as XY addressing. XY addressing has several benefits:

- TMS34010 software can be easily ported from one display configuration to another. System-dependent details such as the number of bits per pixel and the X extent of the display memory are transparent to the software, but are used by the machine to automatically convert the XY coordinates to the address of a pixel in memory.

- XY addressing allows you to think in terms of the high-level concept of XY coordinates rather than in terms of the machine-level mapping of pixels into memory.

- XY addressing facilitates such functions as window clipping.

Figure 4-9 illustrates XY addressing format. The XY address is stored in a 32-bit general-purpose register. The X and Y components are each treated as 16-bit signed integers. The X component resides in the 16 LSBs of the register, and is right justified to bit 0 of the register. The Y component occupies the 16 MSBs of the register, and is right justified to bit 16 of the register. XY coordinates in the range (-32768,-32768) to (+32767,+32767) can be represented. The clipping window, which identifies the pixels that can be altered during drawing operations, is restricted to positive X and Y coordinate values, (0,0) to (+32767,+32767). Thus, pixels identified by negative X or Y coordinates must always lie outside the window.

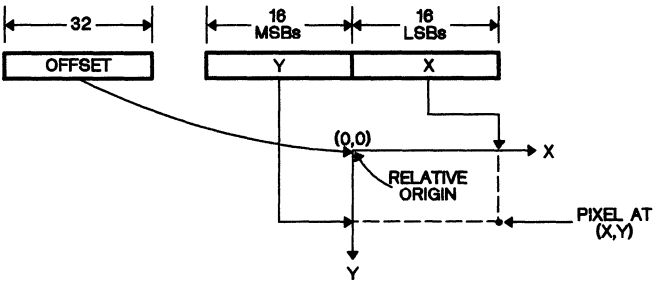


Figure 4-9. Pixel Addressing in Terms of XY Coordinates

4.3.1 XY-to-Linear Conversion

The TMS34010 automatically converts a pixel's XY address to a 32-bit logical address (linear address) for all instructions that use XY addressing. Three parameters are used to perform XY-to-linear conversion:

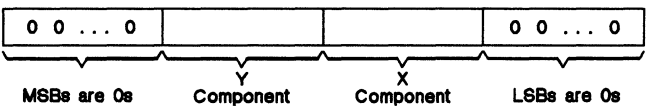
- The logical pixel size (stored in the PSIZE register)
- A pitch conversion factor (stored in the CONVSP or CONVDP registers)
- An offset defining the XY origin (stored in the OFFSET register)

The TMS34010 uses the following formula to calculate the physical address associated with the XY address:

$$\text{Address} = [(Y \times \text{display pitch}) \text{ OR } (X \times \text{pixel size})] + \text{offset}$$

Since the display pitch and pixel size are both powers of two, the calculation is performed using only shift, OR, and add operations. Window clipping may be used to detect out-of-bounds (negative) X or Y values before this calculation is performed.

Linear addresses are formed from XY addresses by simply concatenating the binary numbers that represent the X and Y coordinate values, as shown in Figure 4-10. The number of 0s to the right of the X component of the address depends on the number of bits per pixel, and equals $\log_2$(pixel size). The displacement of the Y component within the 32-bit logical address in Figure 4-10 is equal to $\log_2$(display pitch). Finally, a 32-bit offset is added to the address in Figure 4-10 to calculate the address in memory of the pixel at coordinates (X,Y). The offset corresponds to the linear address in memory of the pixel at (0,0).



Note: The shift value for the Y component is contained in CONVSP or CONVDP register, depending on the instruction being executed.

Figure 4-10. Concatenation of XY Coordinates in Address

The TMS34010 uses the pitch conversion factors **CONVSP** and **CONVDP** to compute the displacement of the Y component within the address, as shown in Figure 4-10. The Y component is displaced from bit 0 of the address by an amount equal to $\log_2$(pitch), which the hardware obtains by inverting the five LSBs of the appropriate CONVSP or CONVDP register. These values must be loaded through software before executing an instruction that uses XY addressing. CONVSP (source address pitch) is used if the XY address points to a *source* pixel or pixel array; CONVDP (destination address pitch) is used if the XY address points to a *destination* pixel or pixel array. The pixel size stored in the PSIZE register is used similarly to determine the displacement of the X component, as shown in Figure 4-10.

The OFFSET register contains the linear memory address of the pixel located at coordinates (0,0) on the monitor screen. The OFFSET register is used in translating XY coordinates into linear addresses, but does not control which region of the display memory is output to refresh the video screen. It is a virtual screen origin. It allows the coordinate axes of the XY address to be translated to an arbitrary position in memory. The OFFSET register supports the use of "window relative" addressing in which the X and Y coordinates are specified relative to coordinate offsets in the display memory. The position and size of a window can be specified arbitrarily. A new offset specified in terms of XY coordinates can be converted to a linear address using the CVXYL instruction. CVXYL converts an XY address to a linear address for the purpose of absolute memory addressing, or to use special features available to instructions that use linear addressing. Figure 4-11 illustrates the XY-to-linear conversion process.

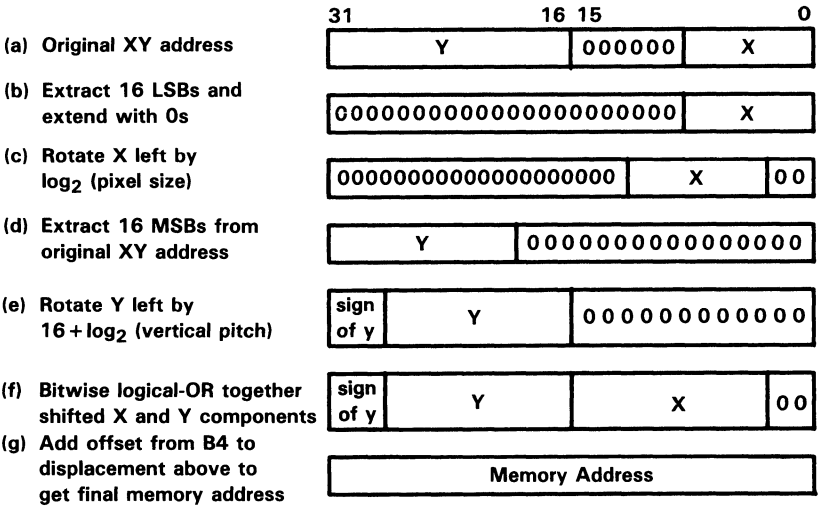


Figure 4-11. Conversion from XY Coordinates to Memory Address

- Step *a* shows the original XY address.
- The X component is extracted in step *b*.
- In step *c*, the X component is shifted left by $\log_2$(pixel size). The result of step *c* represents the product of the X component and the pixel size.
- The Y component is extracted in step *d*.
- In step *e*, the Y component is rotated left by 16+$\log_2$(display pitch). The result of step *e* is Y multiplied by the display pitch.
- In step *f*, the results of steps *c* and *e* are bitwise-ORed to form the displacement in memory of the pixel at (X,Y) from the pixel at the origin.
- In step *G*, the offset is added to produce the final memory address.

The example of Figure 4-11 corresponds to a pixel size of four bits and a pitch of 4,096. The six MSBs of the X half of the XY address (bits 10–15) in Figure 4-11 must be 0s to produce a valid memory address. For this example, the

clipping window should be set to disable writes to pixels having X coordinate values outside the range 0 to +1023.

Generally, given a display with a pitch of 2^n, a valid memory address is produced by the XY translation process shown in Figure 4-11 when only the *n* LSBs of the X half of the XY address are nonzero (that is, when the 16-*n* MSBs are 0). X values may be in the range -32768 to +32767 before clipping. However, after clipping, the X value should be a positive number in the range 0 to (X_{extent} -1), where X_{extent} = pitch/pixel size. The TMS34010's automatic window clipping can be configured to clip pixels lying outside the window; hence, no software overhead is incurred in clipping. Y values lying outside the window are clipped in a similar fashion.

4.4 Pixel Arrays

A rectangular area of the screen that is *DX* pixels wide and *DY* pixels high is an example of a data structure called a **two-dimensional pixel array**. The array contains DX x DY pixels, but can be manipulated by the TMS34010 as one structure. The TMS34010's instruction set includes a powerful set of raster operations, called PixBlts, that manipulate pixel arrays on the screen and elsewhere in memory.

Figure 4-12 shows a pixel array occupying a rectangular region in display memory. The DX pixels in each row of the array are packed together into adjacent cells in the display memory. Rows do not generally occupy adjacent areas of memory, but are separated from each other by a constant displacement called the array pitch. The array pitch is the difference in memory addresses between the start of one row and the start of the row directly beneath it. In the Figure 4-12 example, the array pitch is equal to the display pitch. The product of the array width DX and the pixel size must be less than or equal to the pitch.

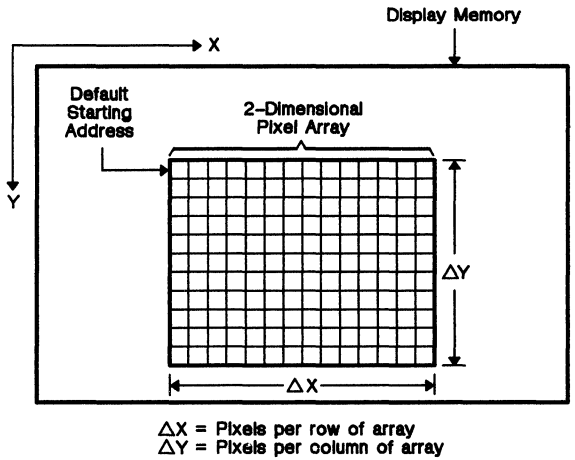


Figure 4-12. Pixel Array

A pixel array is specified in terms of its width, height, pitch, and starting address. The starting address is the address of the first pixel to be moved during a PixBlt. The default starting address is simply the base address of the array; that is, the address of the pixel that has the lowest address in the array.

In Figure 4-12, the XY origin is located in its default position at the upper left corner of the screen. The default starting address is the address of the pixel located in the upper left corner of the array. When a PixBlt operation moves the pixels from a source pixel array to a destination array, the pixels in each row are moved in sequence from left to right, and the rows are moved in sequence from top to bottom.

Certain PixBlt operations allow the starting pixel to be specified as one of the pixels in the other three corners of the array. This feature is provided so that when the source and destination arrays overlap, the appropriate starting corner can be selected to ensure that no data is lost by being overwritten during PixBlt execution. The order in which pixels in the array are moved can be altered to be from right to left or from bottom to top as appropriate to accommodate the change in starting corner.

The starting address of a pixel array can be specified either in terms of the XY coordinates of the starting pixel (XY address), or the memory address of the starting pixel (linear address):

- An array whose starting location is specified as an XY address is referred to as an *XY array*. In this format, the starting location of the array is identified by the XY coordinates of the first pixel in the array.

- A pixel array whose starting location is specified as a memory address is referred to as a *linear array*. In this format, the location of the array is identified by the memory address of the first pixel (the pixel that has the lowest bit address) in the array.

The XY array format has two advantages. First, the starting location of the array is specified in system-independent Cartesian coordinates rather than as a system-dependent memory address. Second, the TMS34010's window checking (which allows it to automatically detect an attempt to write a pixel inside or outside a specified window) can only be used in conjunction with XY addressing.

The linear format's main advantage is that the array pitch does not have to be a power of two. This supports a wider variety of memory organizations. Using XY format, the array pitch is constrained to be a power of two.

The general rules governing array pitch are as follows. When an array is specified in XY format, the pitch **must** be a power of two. The pitch for an array specified in linear format may be any multiple of 16; that is, the four LSBs of the pitch must be 0s. There are a few important exceptions to the second rule which are discussed below.

For the special case of a PIXBLT B,XY or PIXBLT B,L instruction, the source pitch may be any value. This feature supports efficient use of memory by allowing adjacent rows of the source array to be packed together with no intervening gaps. The destination pitch must still be a multiple of 16.

Under certain conditions the linear source array specified for a PIXBLT L,XY or PIXBLT B,XY must have a pitch that is a power of two. This is necessary when the linear start address for the array has to be adjusted in the Y direction due to one of the following conditions:

- The source array is automatically preclipped to lie within a rectangular window.

- One of the lower two corners of the source array (refer to Figure 4-12) is selected to be the start address.

In either case, the start addresses specified for both the source and destination arrays are automatically adjusted, and for this purpose the conversion factors specified in the CONVSP and CONVDP registers must be valid.

While PixBlts are useful for moving arrays from one area of the screen to another, they can also be used to move arrays to the screen from other parts of memory, and vice versa. The pitch for the off-screen pixel array can be specified independently of the pitch for the on-screen array. This permits off-screen data to make efficient use of storage, regardless of the display pitch. On-screen objects may be defined as XY arrays but may be more efficiently stored as linear arrays in off-screen memory. The PIXBLT instructions support the transfer of a linear array to an XY array, and vice versa. PIXBLT instructions can also be used to rapidly move blocks of non-pixel data (ASCII characters, for example) from one location in memory to another.

Section 5

CPU Registers and Instruction Cache

The TMS34010 has two on-chip general-purpose register files, file A and file B. Each register file contains 15 32-bit registers. The two files share a 32-bit hardware stack pointer (SP) that automatically manages the system stack during interrupts and subroutine calls. The TMS34010 also has two dedicated 32-bit registers – a program counter and a status register. An on-chip cache holds up to 128 instruction words, and is transparent to software. The CPU registers and instruction cache are discussed in the following sections:

In addition to the CPU registers, the TMS34010 contains 28 memory-mapped registers that are dedicated to I/O functions; Section 6 discusses the I/O registers.

5.1 General-Purpose Registers

The TMS34010 has 30 32-bit general-purpose registers, divided into register files A and B. In addition, a single stack pointer (SP) is common to both register files.

The multiple internal data paths that link the ALU and general-purpose registers provide single machine state execution of most register-to-register instructions. Single-state instructions include add, subtract, Boolean operations, and shifts (1 to 32 bits). During a single-state instruction, the following actions occur:

1) Two 32-bit operands are read in parallel from the general-purpose registers.
2) The ALU performs the specified operation.
3) The 32-bit result is stored in the specified general-purpose register.

The general-purpose registers are dual-ported to permit operands to be read from two independent registers at the same time.

5.1.1 Register File A

Fifteen of the 30 general-purpose registers, A0–A14, form register file A. These registers can be used for data storage and manipulation. No hardware-dedicated functions are associated with these general-purpose registers.

All register-to-register instructions (except MOVE *Rs*, *Rd*) require both registers to be in the same file. Instructions that manipulate registers A0–A14 can also manipulate the stack pointer. The SP can be specified in place of an A-file register in any of these instructions. Figure 5-1 illustrates register file A.

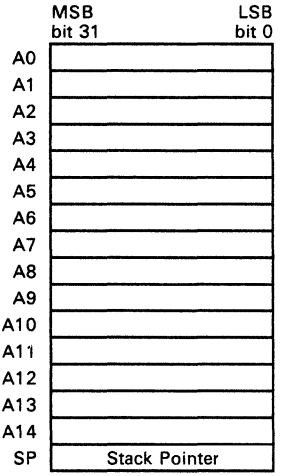


Figure 5-1. Register File A

5.1.2 Register File B

Register file B consists of 15 general-purpose registers, B0–B14. All register-to-register instructions (except MOVE *Rs,Rd*) require both registers to be in the same file. Instructions that manipulate registers B0–B14 can also manipulate the stack pointer. The SP can be specified in place of a B-file register in any of these instructions.

Registers B0–B14 can be used for general-purpose functions such as data storage and manipulation. During PixBlt and other pixel operations, however, these registers are assigned hardware-dedicated functions.

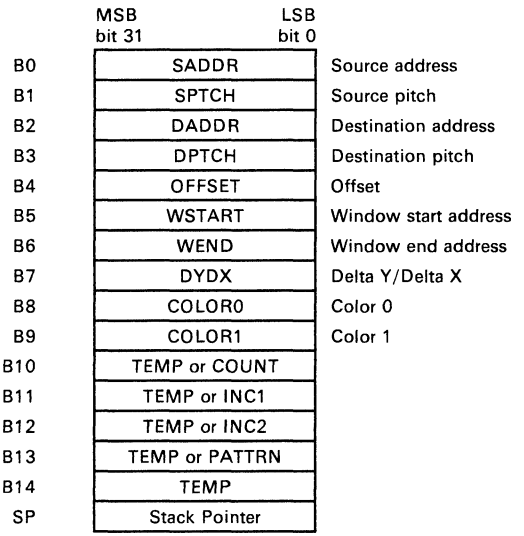


Figure 5-2. Register File B

As Figure 5-2 shows, registers B0–B9 are used as special-purpose registers during pixel operations. These registers must be loaded with specific parameters before execution of pixel operations. Registers B10–B14 are used as special-purpose registers for the LINE instruction. During pixel operations, registers B10–B14 are used for temporary storage; their previous contents are destroyed. Register functions may vary for individual instructions.

Section 5.1.4 describes the B-file registers in detail.

5.1.3 Stack Pointer

The stack pointer (SP) is a 32-bit register that contains the bit address of the top of the system stack. *The TMS34010 contains only a single SP.* However, this SP can be addressed as a member of *either* register file, as register A15 or register B15. Any instruction that uses a general-purpose register as an operand can also use the SP as an operand.

Figure 5-3 illustrates the stack pointer; Section 3.3 (page 3-6) describes stack operation in detail.

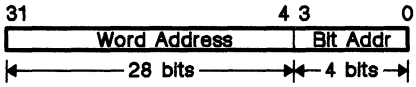


Figure 5-3. Stack Pointer Register

The system stack grows in the direction of smaller addresses. During an interrupt, the PC and ST are pushed onto the stack to permit the interrupted routine to resume execution when interrupt processing is completed. A subroutine call saves the PC on the stack to allow the calling routine to resume execution when subroutine execution is completed.

The stack pointer always points to the value at the top of the stack. Specifically, the SP contains the 32-bit address of the LSB of that value. While the four LSBs of the SP may be set to an arbitrary value, stack operations execute more efficiently when the four LSBs are 0s. Setting these bits to 0s aligns the stack pointer to 16-bit word boundaries in memory, reducing to two the number of memory cycles necessary to push or pop the contents of a 32-bit register.

The SP can be specified as the source or destination operand in any instruction that operates on the general-purpose registers. The SP can be accessed as register 15 in file A or B. Refer to the descriptions of the specific instructions for details.

5.1.4 Implied Graphics Operands

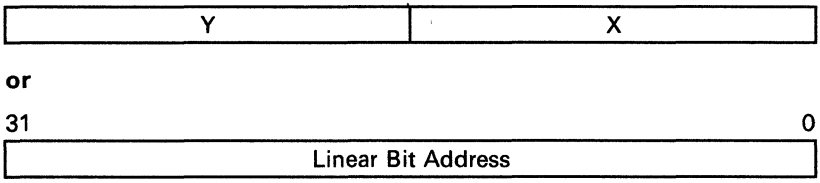
Table 5-1 summarizes the B-file register functions during graphics operations. These registers are referred to as *implied graphics operands*. Several I/O registers, described in Section 6, are also implied graphics operands.

Table 5-1. B-File Registers Summary

Reg.	Function	Description
B0	SADDR	**Source Address**. Address of the upper left corner of the source pixel array (lowest pixel address in the array). SADDR is a linear or XY address, depending on the instruction which uses it.
B1	SPTCH	**Source Pitch**. Difference in linear start addresses between adjacent rows of a source pixel array.
B2	DADDR	**Destination Address**. Address of the upper left corner of the destination pixel array (lowest pixel address in the array). DADDR is a linear or XY address, depending on the instruction which uses it.
B3	DPTCH	**Destination Pitch**. Difference in linear start addresses between adjacent rows of a destination pixel array.
B4	OFFSET	**Offset**. Linear bit address corresponding to XY-coordinate origin (X=0, Y=0).
B5	WSTART	**Window Start Address**. XY address of the upper left corner of the window (smallest X and Y coordinate values in the array).
B6	WEND	**Window End Address**. XY address of the lower right corner of the window (largest X and Y coordinate values in the array).
B7	DYDX	**Delta Y/Delta X**. The 16 LSBs of this register specify the width (X dimension) of the destination array. The 16 MSBs specify the height (Y dimension) of the destination array. If either DY = 0 or DX = 0, then nothing is moved.
B8	COLOR0	**Pixel value corresponding to "color 0"**. COLOR0 contains the source background color to be used during a color-expand operation (PIXBLT B,XY or PIXBLT B,L). The pixel value should be replicated throughout the 16 LSBs of register B8 (see note below). Non-replicated patterns may be entered for dithering effects. The 16 MSBs are ignored during the expand operation. For example, at four bits per pixel, COLOR0 contains four identical pixel values, as shown below.
B9	COLOR1	**Pixel value corresponding to "color 1"**. COLOR1 contains the source foreground color to be used during a color-expand, fill, or draw-and-advance operation. The pixel value should be replicated throughout the 16 LSBs of register B9 (see note below). Nonreplicated patterns may be entered for dithering effects. The 16 MSBs are ignored during the expand operation. For example, at four bits per pixel, COLOR1 contains four identical pixel values, as shown below.
B10–B14		**PixBlt temporary registers**. PixBlt instructions use these registers for storing temporary values and context information necessary to resume execution of a partially-completed PixBlt operation in the event of an interrupt.
SP	SP	**Stack pointer**. SP contains the bit address of the top of the stack.

Notes: To provide upward compatibility with future versions of the GSP, replicate the pixel value throughout all 32 bits of COLOR0 or COLOR1, as shown.

Format



Description SADDR contains the source array address for PIXBLTs. Generally, SADDR points to the pixel with the lowest address in the source array. When the selected starting corner is not the upper left corner, the TMS34010 automatically adjusts SADDR to point to the selected starting corner of the source array. (For PIXBLT L,L, however, you must manually adjust SADDR to point to the starting corner. This feature allows you to use PIXBLT L,L for manipulating pixel arrays with pitches that are not powers of two.)

SADDR is in either XY or linear format. If the first operand of a PIXBLT instruction is an ***L*** (such as PIXBLT L,XY), then SADDR is in linear format. If the first operand of a PIXBLT instruction is an ***XY*** (such as PIXBLT XY,L), then SADDR is in XY format.

During PIXBLT operations, SADDR is used in linear format. When the PIXBLT is completed, SADDR points to the starting location of the row that follows the last row in the array. If a PIXBLT is interrupted, SADDR points to the next word of pixels to be read.

During LINE operation, SADDR contains the current decision variable value.

The following instructions use SADDR as an implied operand:

Instruction	SADDR Format and Function
LINE	Contains $d=2b-a$, used for the line draw.
PIXBLT B,L	Linear address; points to the beginning of a binary source array (a bit map).
PIXBLT B,XY	Linear address; points to the beginning of a binary source array (a bit map).
PIXBLT L,L	Linear address with special requirements when PBH = 1 or PBV = 1. Refer to the PIXBLT L,L for a description of its unique requirements.
PIXBLT L,XY	Linear address; points to the beginning of a source array.
PIXBLT XY,L	XY address; points to the beginning of a source array.
PIXBLT XY,XY	XY address; points to the beginning of a source array.

Example

```
SADDR   .set  B0

        MOVI  00080015h, SADDR   ; Move XY value 15h,8h
                                 ; into B0
        MOVI  00010AFCh, SADDR   ; Move linear value
                                 ; 10AFCh into B0
```

Format

31	0
Linear Bit Address	

Description SPTCH specifies the linear difference in the start addresses of adjacent rows of the source array for PIXBLT and FILL instructions. The TMS34010 uses the value in SPTCH to move from row to row through the source array. SPTCH **must** be an integer multiple of 16 (except for the special cases of PIXBLT B,L and PIXBLT B,XY). SPTCH is constrained in some cases to be a power of two; this allows XY addressing and allows SADDR to be automatically adjusted to point to an alternate starting corner.

The following instructions use SPTCH as an implied operand.

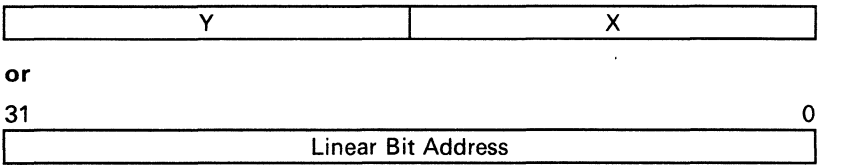
Instruction	SPTCH Format and Function
PIXBLT B,L	Linear; any value.
PIXBLT B,XY	Linear; power of two for windowing, any value otherwise.
PIXBLT L,L	Linear; multiple of 16.
PIXBLT L,XY	Linear; power of two $\geq$ 16 for windowing or PBV = 1, multiple of 16 otherwise.
PIXBLT XY,L	Linear; power of two $\geq$ 16.
PIXBLT XY,XY	Linear; power of two $\geq$ 16.

Example

```
SPTCH   .set  B1

        MOVI  00001000h, SPTCH   ; Power of two for
                                 ; PIXBLT XY,L
        MOVI  00010AFCh, SPTCH   ; Any value for
                                 ; PIXBLT B,L
```

Format



Description

DADDR contains the destination array address for PIXBLTs. Generally, DADDR points to the pixel with the lowest address in the destination array. When the selected starting corner is not the upper left corner, the TMS34010 automatically adjusts DADDR to point to the selected starting corner of the destination array. (For PIXBLT L,L, however, you must manually adjust DADDR to point to the starting corner. This feature allows you to use PIXBLT L,L for manipulating pixel arrays with pitches that are not powers of two.)

DADDR is also used in conjunction with DYDX to perform a common rectangle function for some instructions (FILL XY, PIXBLT B,XY, PIXBLT L,XY, and PIXBLT XY,XY, with window option 1). In these cases, DADDR is set to the starting XY address of the common rectangle that represents the intersection of the original destination array and the clipping window indicated by WSTART and WEND. No drawing is performed. If the array and the window do not intersect, the V bit is not set and the contents of DADDR are undefined.

DADDR is in either XY or linear format. If the second operand of the PIXBLT instruction is an ***L*** (such as PIXBLT XY,L), then DADDR is in linear format. If the second operand of the PIXBLT instruction is an ***XY*** (such as PIXBLT XY,XY), then DADDR is in XY format.

If DADDR is specified in XY format, the PIXBLT converts it to the corresponding linear address prior to the start of the pixel array transfer. During PIXBLT operation, DADDR is maintained in linear format. When the PIXBLT completes, DADDR points to the linear starting address of the row following the last row in the array. If a PIXBLT is interrupted, DADDR points to the next word of pixels to be read.

For the LINE instruction, DADDR contains the XY address of the next point on the line.

The following instructions use DADDR as an implied operand.

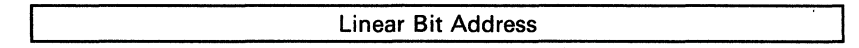
Instruction	DADDR Format and Function
FILL L	Linear; points to the beginning of the destination array.
FILL XY	XY; points to the beginning of the destination array.
LINE	XY; points to the current pixel.
PIXBLT B,L	Linear; points to the beginning of the destination array.
PIXBLT B,XY	XY; points to the beginning of the destination array.
PIXBLT L,L	Linear with special requirements when PBH=1 or PBV=1. Refer to the PIXBLT L,L for a description of its unique requirements.
PIXBLT L,XY	XY; points to the beginning of the destination array.
PIXBLT XY,L	Linear; points to the beginning of the destination array.
PIXBLT XY,XY	XY; points to the beginning of the destination array.

Example

```
DADDR   .set  B2

        MOVI  00080015h, DADDR    ; Move XY value 15h,8h
                                  ; into B2
        MOVI  00010AFCh, DADDR    ; Move linear value
                                  ; 10AFCh into B2
```

Format



Description DPTCH specifies the linear difference in the starting memory addresses of adjacent rows of the destination array for PIXBLT and FILL instructions. The TMS34010 uses the value in DPTCH to move from row to row through the destination array. DPTCH **must** be an integer multiple of 16 (except for FILL L when DX=1). DPTCH is also constrained in some cases to be a power of two; this allows XY addressing and allows DADDR to be automatically adjusted to point to an alternate starting corner.

The following instructions use DPTCH as an implied operand.

Instruction	DPTCH Format and Function
FILL L	Linear; unused when DY=1.
FILL XY	Linear; power of two ≥ 16.
PIXBLT B,L	Linear; multiple of 16.
PIXBLT B,XY	Linear; power of two ≥ 16 for windowing, multiple of 16 otherwise.
PIXBLT L,L	Linear; multiple of 16.
PIXBLT L,XY	Linear; power of two ≥ 16.
PIXBLT XY,L	Linear; power of two us.> 16 for PBV = 1, multiple of 16 otherwise.
PIXBLT XY,XY	Linear; power of two ≥ 16.

Example

```
DPTCH   .set  B3

        MOVI  00001000h, DPTCH    ; Power of two for
                                  ; PIXBLT XY,L
        MOVI  00010AFCh, DPTCH    ; Any value for
                                  ; PIXBLT L,L
```

Format

Description OFFSET contains the linear address of the first pixel in the XY coordinate space for instructions using XY addressing. This corresponds to the linear address of the XY origin (X=0,Y=0). This value is used as the memory base for performing XY to linear address conversions.

OFFSET is always in linear format. It may be placed at any position in the TMS34010 linear address space and should contain a pixel-aligned value for proper XY address conversions, transparency, pixel processing, and plane masking. Instructions that use OFFSET as an implied operand do not modify the contents of OFFSET.

The following instructions use OFFSET as an implied operand.

Instruction	OFFSET Format and Function
CVXYL *Rs,Rd*	Linear address of XY origin
DRAV *Rs,Rd*	Linear address of XY origin
FILL XY	Linear address of XY origin
LINE	Linear address of XY origin
PIXBLT B,XY	Linear address of XY origin
PIXBLT L,XY	Linear address of XY origin
PIXBLT XY,L	Linear address of XY origin
PIXBLT XY,XY	Linear address of XY origin
PIXT *Rs,Rd*.XY	Linear address of XY origin
PIXT *Rs*.XY,*Rd*	Linear address of XY origin
PIXT *Rs*.XY,*Rd*.XY	Linear address of XY origin

Example

```
OFFSET  .set  B4

        MOVI  00042000h, OFFSET    ; Linear value on
                                   ; pixel boundary
```

Format

31 16	15 0
Window start Y	Window start X

Description

WSTART specifies the XY address of the least significant pixel contained in the rectangular destination clipping window. WSTART must be valid for instructions that use an XY destination address and a window option. The least significant pixel is the pixel with the lowest address in the array. For a screen with the ORG bit of the DPYCTL register set to 0, this corresponds to the pixel in the upper left corner of the pixel array.

WSTART may be placed at any position in the positive quadrant of the XY address space. It describes an inclusive pixel; that is, the pixel at the XY location contained in WSTART is included in the window. The value in WSTART is used with WEND, DADDR, and DYDX to preclip pixels, lines, and pixel arrays. WSTART is not modified by instruction execution.

The following instructions use WSTART as an implied operand.

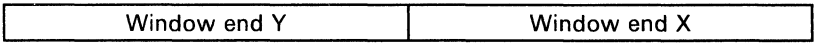
Instruction	WSTART Format and Function
CPW *Rs,Rd*	XY value of least significant window corner
DRAV *Rs,Rd*	XY value of least significant window corner
FILL XY	XY value of least significant window corner
LINE	XY value of least significant window corner
PIXBLT B,XY	XY value of least significant window corner
PIXBLT L,XY	XY value of least significant window corner
PIXBLT XY,XY	XY value of least significant window corner
PIXT *Rs,Rd*.XY	XY value of least significant window corner
PIXT *Rs*.XY,*Rd*.XY	XY value of least significant window corner

Example

```
WSTART  .set  B5

        MOVI  00400100h, WSTART    ; XY value (256,64)
                                   ; stored in WSTART
```

Format



Description WEND specifies the XY address of the most significant pixel contained in the rectangular destination clipping window. WEND must be valid for instructions that use an XY destination address and a window option. The most significant pixel is the pixel with the highest address in the array. For a screen with the ORG bit of the DPYCTL register set to 0, this corresponds to the pixel in the lower right corner of the pixel array.

WEND may be placed at any position in the positive quadrant of the XY address space. It describes an inclusive pixel; that is, the pixel at the XY location contained in WEND is included in the window. The value in WEND is used with WSTART, DADDR, and DYDX to preclip pixels, lines, and pixel arrays. WEND is not modified by instruction execution.

The following instructions use WEND as an implied operand.

Instruction	WEND Format and Function
CPW *Rs,Rd*	XY value of most significant window corner
DRAV *Rs,Rd*	XY value of most significant window corner
FILL XY	XY value of most significant window corner
LINE	XY value of most significant window corner
PIXBLT B,XY	XY value of most significant window corner
PIXBLT L,XY	XY value of most significant window corner
PIXBLT XY,XY	XY value of most significant window corner
PIXT *Rs,Rd*.XY	XY value of most significant window corner
PIXT *Rs*.XY,*Rd*.XY	XY value of most significant window corner

Example

```
WEND    .set  B6

        MOVI  00400100h, WEND     ; XY value (256,64) stored
                                  ; in WEND
```

Format

31 16	15 0
Delta Y	Delta X

Description

DYDX specifies the X and Y dimensions of the rectangular destination array for PIXBLT and FILL instructions. Both the X and Y dimensions are in pixels; that is, the DX value is number of pixels in width of the array, and DY is the number of rows of pixels in the destination array.

When the window clipping option is selected, the pixel block dimensions for the transfer are determined by the relationships between WSTART, WEND, DADDR, and DYDX. If either the X or Y dimension is 0, then the block is interpreted as having a dimension of 0; no transfer is performed.

The values for DY and DX may range up to the coordinate extent of the display (up to 65,535, depending on the display pitch and pixel size selected). For window operations, the relationship between DYDX, WSTART, and WEND is such that DY = Y_{end} - Y_{start} + 1 and DX = X_{end} - X_{start} + 1. The value in DYDX is used with WSTART, DADDR, and DYDX to preclip pixels, lines, and pixel arrays.

Most graphics instructions do not modify the contents of DYDX. For FILL XY, PIXBLT B,XY, PIXBLT L,XY, and PIXBLT XY,XY, with window option 1, however, DYDX is used with DADDR to perform a common rectangle function. In this case, the instruction sets DYDX to the dimensions of the common pixel block described by the intersection of the original destination array and the window identified by WSTART and WEND. No drawing is performed. If there is no common rectangle, the V bit is not set and the value of DYDX is indeterminate. See these instructions for further information.

The following instructions use DYDX as an implied operand.

Instruction	DYDX Format and Function
FILL L	Array dimensions in XY format.
FILL XY	Array dimensions in XY format; special results when W=1 is selected, as previously noted.
LINE	Dimensions of the rectangle described by the line to be drawn.
PIXBLT B,L	Array dimensions in XY format
PIXBLT B,XY	Array dimensions in XY format; special results when pick is selected, as previously noted.
PIXBLT L,L	Array dimensions in XY format.
PIXBLT L,XY	Array dimensions in XY format; special results when pick is selected, as previously noted.
PIXBLT XY,L	Array dimensions in XY format.
PIXBLT XY,XY	Array dimensions in XY format; special results when pick is selected, as previously noted.

Example

This example illustrates the relationship of DYDX to WSTART and WEND.

```
WSTART  .set   B5
WEND    .set   B6
DYDX    .set   B7

        MOVE   WEND, DYDX       ; Put WEND into DYDX
        SUBXY  WSTART, DYDX     ; Generate (WEND - WSTART)
        ADDI   10001h, DYDX     ; Increment by 1 in each
                                ; dimension
```

Format

31	0
Replicated Pixel Value	

Description COLOR0 specifies the replacement color for 0 bits in the source array for PIXBLT B,L and PIXBLT B,XY instructions. These two instructions transform binary pixel array information to multiple bits per pixel arrays using the color information in COLOR1 and COLOR0. The lower 16 bits of COLOR0 are used for the 0 or background color. There is a direct correspondence between the alignment of pixels within the COLOR0 register and pixels within memory words to be altered. That is, individual pixels within COLOR0 are used as they align with destination pixels in the destination word.

Execution of graphics instructions does not modify COLOR0.

To provide upward compatibility with future versions of the GSP, the plane mask should be replicated through all 32 bits of COLOR0.

The following instructions use COLOR0 as an implied operand.

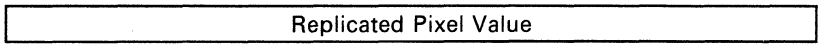
Instruction	COLOR0 Contents
PIXBLT B,L	Background pixel color for color-expanded array
PIXBLT B,XY	Background pixel color for color-expanded array

Example This example is for 4-bit pixels. A pixel value of 5 is replicated throughout the register.

```
COLOR0  .set  B8

        MOVI  55555555h, COLOR0 ; store uniform pixel
                                ; value in COLOR0
```

Format



Description

COLOR1 specifies the replacement color for pixels to be altered at the destination pixel or pixel block for FILL, DRAV and LINE instructions.

For PIXBLT B,L and PIXBLT B,XY instructions, COLOR1 specifies the replacement color for 1 bits in the source array. These two instructions transform binary pixel array information to multiple-plane pixel arrays using color information in COLOR1 and COLOR0. There is a direct correspondence between the alignment of pixels within the COLOR1 register and pixels within memory words to be altered. That is, individual pixels within COLOR1 are used as they align with destination pixels in the destination word.

Execution of graphics instructions does not modify COLOR1.

To provide upward compatibility with future versions of the GSP, the plane mask should be replicated through all 32 bits of COLOR1.

The following instructions use COLOR1 as an implied operand.

Instruction	COLOR1 Contents
DRAV *Rs,Rd*	Pixel color for pixel draw
FILL L	Pixel color for filled array
FILL XY	Pixel color for filled array
LINE	Pixel color for line draw
PIXBLT B,L	Foreground pixel color for color-expanded array
PIXBLT B,XY	Foreground pixel color for color-expanded array

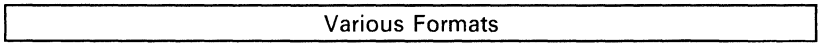
Example

This example is for 4-bit pixels. A pixel value of 3 is replicated throughout the register.

```
COLOR1  .set  B9

        MOVI  33333333h, COLOR1  ; Store uniform pixel
                                 ; value in COLOR1
```

Format



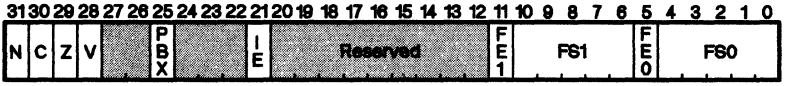
Description The functions of these registers depend on which instruction uses them:

- PIXBLT and FILL instructions use registers B10 through B14 as temporary registers that hold intermediate values.
- The LINE instruction uses these registers as implied operands with the following functions:
  - B11 is the INC1 register; it specifies the X and Y increments for a diagonal step.
  - B12 is the INC2 register; it specifies the X and Y increments for a nondiagonal step.
  - B10 is the COUNT register; it specifies the number of pixels to be drawn in the line.
  - B13 is the PATTRN register; it is reserved for future LINE draw enhancement. It should be set to 0FFFFFFFFh before executing the LINE instruction to ensure software compatibility.
  - B14 is a temporary register (TEMP) that holds intermediate values.

5.2 Status Register

The status register (ST) is a special-purpose, 32-bit register that specifies the processor status. The ST also contains several parameters that specify the characteristics of two programmable data types, fields 0 and 1. The ST is initialized to 00000010h at reset.

Figure 5-4 illustrates the status register. Table 5-2 lists the functions associated with the status bits. Table 5-3 describes the encoding of the field size bits in FS0 and FS1.



Note: The status register bits marked **reserved** (bits 12–20, 22–24, and 26–27) are currently unused. When read, a reserved bit returns the last value written to it. At reset, all reserved bits are forced to 0s.

Figure 5-4. Status Register

Table 5-2. Definition of Bits in Status Register

Bit No.	Field Name	Function
0–4	FS0	**Field Size 0.** Length in bits of first memory data field (see Table 5-3 for values).
5	FE0	**Field Extend 0.** Bit determines whether field from memory is extended with 0s or with the sign bit when loaded into 32-bit general-purpose register. FE0 = 0 selects zero extension for field 0 FE0 = 1 selects sign extension for field 0
6–10	FS1	**Field Size 1.** Length in bits of second memory data field (see Table 5-3 for values).
11	FE1	**Field Extend 1.** Bit determines whether field from memory is extended with 0s or with the sign bit when loaded into 32-bit general-purpose register. FE1 = 0 selects zero extension for field 1 FE1 = 1 selects sign extension for field 1
21	IE	**Interrupt Enable.** Master interrupt enable/disable bit. IE = 0 disables all maskable interrupts IE = 1 enables all maskable interrupts
25	PBX	**PixBlt Executing.** Indicates upon return from an interrupt that the interrupt occurred between instructions or in the middle of a PIXBLT or FILL instruction. 0 = Indicates interrupt occurred at PIXBLT or FILL instruction boundary 1 = Indicates interrupt occurred in the middle of a PIXBLT or FILL instruction
28	V	**Overflow.** Set according to instruction execution.
29	Z	**Zero.** Set according to instruction execution.
30	C	**Carry.** Set according to instruction execution.
31	N	**Negative.** Set according to instruction execution.
12 20 22–24 26–27	–	**Reserved**

Table 5-3. Decoding of Field-Size Bits in Status Register

Five FS Bits	Field Size†	Five FS Bits	Field Size†	Five FS Bits	Field Size†	Five FS Bits	Field Size†
00001	1	01001	9	10001	17	11001	25
00010	2	01010	10	10010	18	11010	26
00011	3	01011	11	10011	19	11011	27
00100	4	01100	12	10100	20	11100	28
00101	5	01101	13	10101	21	11101	29
00110	6	01110	14	10110	22	11110	30
00111	7	01111	15	10111	23	11111	31
01000	8	10000	16	11000	24	**00000**	**32**

† In bits

5.3 Program Counter

The program counter (PC) is a dedicated 32-bit register that points to the next instruction word to be executed. Instructions are always aligned on even 16-bit word boundaries, and as shown in Figure 5-5, the four LSBs of the PC are always 0s.

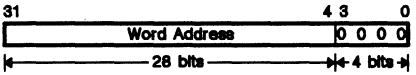


Figure 5-5. Program Counter

An instruction consists of one or more instruction words. The first word contains the opcode for the instruction. Additional words may be required for immediate data, displacements, or absolute addresses. As each instruction word is fetched, the PC is incremented by 16 to point to the next instruction word. The PC contents are replaced during a branch instruction, subroutine call instruction, return instruction, or interrupt. Instructions may be categorized according to their effect on the PC, as indicated in Table 5-4.

Table 5-4. Instruction Effects on the PC

Category	Description
Non-branch	The PC is incremented by 16 at the end of the instruction, allowing execution to proceed sequentially to the next instruction.
Absolute Branch (TRAP, CALL, JAcc)	The PC is loaded with an absolute address; the four LSBs of the address are set to 0s.
Relative Branch (JRcc, DSJxx)	The signed displacement (8 or 16 bits) is added to the current contents of the PC. The signed displacement is treated as a word displacement; that is, it is shifted left four bit positions before it is added to the PC.
Indirect Branch (JUMP, CALL, EXGPC)	The PC is loaded with the register contents. The four LSBs are set to 0s.

5.4 Instruction Cache

Most program execution time is spent on repeated execution of a few main procedures or loops. Program execution can be speeded up by placing these often used code segments in a fast memory. The TMS34010 uses a 256-byte instruction cache for this purpose.

Only instruction words (memory words that are pointed to by the PC) can be accessed from the cache. This includes opcodes, immediate operands, displacements, and absolute addresses. Instructions and data may reside in the same area of memory; therefore, data may occasionally be copied into the instruction cache along with instruction words. However, the processor cannot access data from the cache. All reads and writes of data in memory bypass the cache.

5.4.1 Cache Hardware

The instruction cache contains 256 bytes of RAM, used to store up to 128 16-bit instruction words. Each instruction word in cache is aligned on an even word boundary. Figure 5-6 illustrates cache organization.

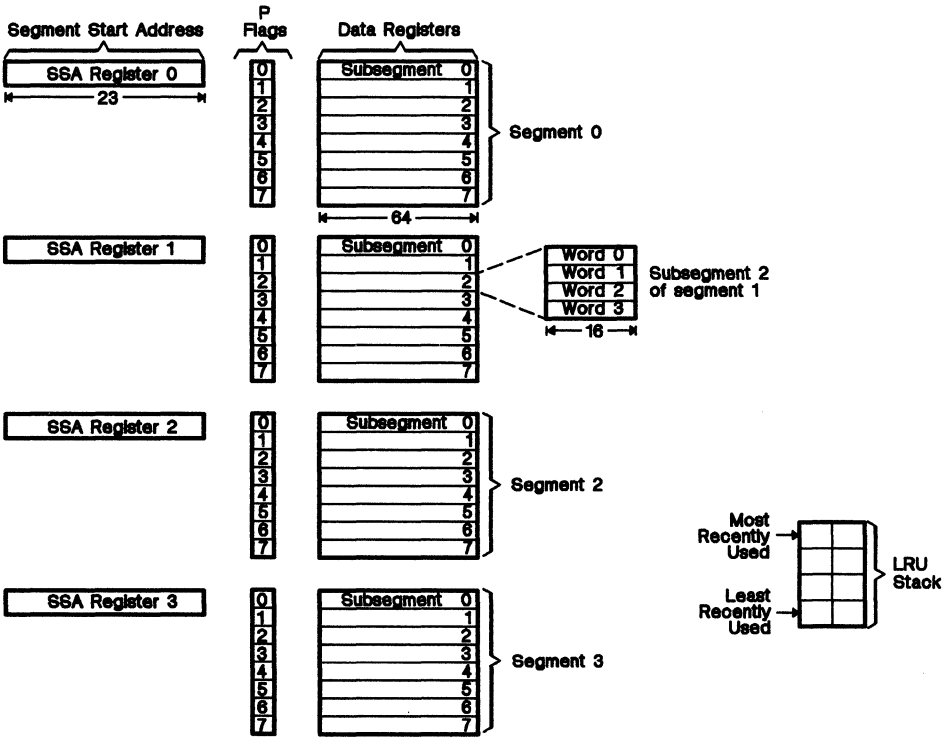


Figure 5-6. TMS34010 Instruction Cache

The cache is divided into four 32-word segments. Each cache segment may contain up to 32 words of a 32-word segment in memory. This memory seg-

ment is a block of 32 contiguous words beginning at an even 32-word boundary in memory.

Each cache segment is divided into eight subsegments; each subsegment contains four words. Dividing each segment into subsegments reduces the number of word fetches required from memory when fewer than 32 words of a memory segment are used. Each of the four cache segments is associated with a segment start address (SSA) register. Figure 5-7 shows how an instruction word is partitioned into the components used by the cache control algorithm.

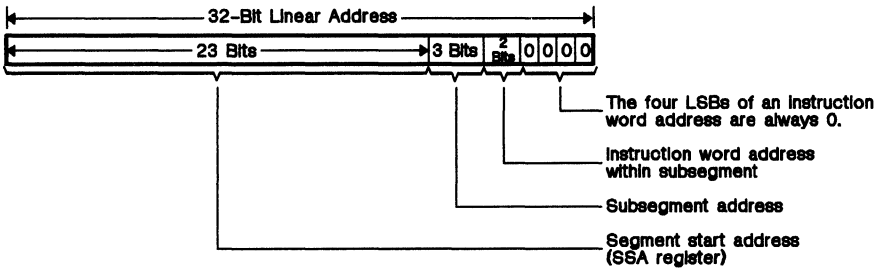


Figure 5-7. Segment Start Address

The 23 bits of the SSA register correspond to the 23 MSBs of the segment's memory address. These 23 MSBs are common to all eight subsegments within a segment. The next three bits (bits 6–8) identify one of the eight subsegments. Bits 4 and 5 identify one of the four words contained in a subsegment. The four LSBs are always 0s because instructions are aligned on word boundaries.

5.4.2 Cache Replacement Algorithm

When the TMS34010 requests an instruction word from a segment that is not in the cache, the contents of one of the four cache-resident segments must be discarded to make room for the segment that contains the requested word. A modified form of the least-recently-used (LRU) replacement algorithm is used to select the segment to be discarded.

The LRU segment manager (part of the cache control logic) maintains an LRU stack to track use of the four segments. The LRU stack contains a queue of segment numbers, 0 through 3. Each time a segment is accessed, its segment number is moved to the top of the stack, pushing the other segment numbers down as necessary to make room at the top. Thus, the number at the top of the LRU stack identifies the most-recently-used segment and the number at the bottom identifies the least-recently-used segment.

When a new segment must be loaded into cache, the least-recently-used segment is discarded. The eight P flags (described in Section 5.4.3) of the selected segment are set to 0s, and the segment's SSA register is loaded with the new segment address. After the requested subsegment has been loaded from memory, its P flag is set to 1, and the requested instruction fetch is allowed to complete.

Following a reset, all P flags in the cache are set to 0 and the four segment numbers in the LRU stack are stored in numerical order (0–1–2–3).

5.4.3 Cache Operation

When the TMS34010 requests an instruction word, it checks to see if the word is contained in cache. First, it compares the 23 MSBs of the instruction address to the four SSA registers. If a match is found, the processor searches for the appropriate subsegment. A present (P) flag, associated with each subsegment, indicates the presence of a particular subsegment within a cache segment. P=1 indicates that the requested word is in cache; this is called a cache hit. If there is no match, or if there is a match and P=0, the word is not in cache; this is called a cache miss.

5.4.3.1 Cache Hit

The cache contains the requested instruction word. The processor performs the following actions:

1) A short (one machine state) access cycle reads the instruction word from cache.

2) The segment number is moved to the top of the LRU stack, pushing the other three segment numbers toward the bottom of the stack.

Due to pipelining, instruction fetches from the cache frequently overlap completion of preceding instructions. The overhead due to instruction fetches in such cases is effectively zero.

5.4.3.2 Cache Miss

The cache does not contain the instruction word. There are two types of cache miss - subsegment miss and segment miss.

- **Subsegment Miss.** The 23 MSBs of the instruction word address match one of the four SSA registers' 23 MSBs; that is, the appropriate segment is in the cache. However, the P flag for the requested subsegment is not set. The processor performs the following actions:

  1) The four-word subsegment containing the requested instruction word is read from local memory into the cache.

  2) The segment number is moved to the top of the LRU stack, pushing the other three segment numbers toward the bottom of the stack.

  3) The subsegment's P flag is set.

  4) The instruction word is read from the cache.

- **Segment Miss.** The instruction word address does not match any of the SSA registers. The processor performs the following actions:

  1) The least-recently-used segment is selected for replacement; the P flags of all eight subsegments are cleared.

  2) The SSA register for the selected segment is loaded with the 23 MSBs of the address of the requested instruction word.

3) The four-word subsegment in memory that contains the requested instruction word is read into the cache. It is placed in the appropriate subsegment of the least-recently-used segment. The subsegment's P flag is set to 1.

4) The LRU stack is adjusted by moving the number of the new segment from the bottom (indicating that it is least recently used) to the top (indicating that it is most recently used). This pushes the other three segment numbers in the stack down one position.

5) The instruction word is read from the cache.

5.4.4 Self-Modifying Code

Avoid using self-modifying code; it can cause unpredictable results. When a program modifies its own instructions, only the copy of the instruction that resides in external memory is affected. Copies of the instructions that reside in cache are not modified, and the internal control logic does not attempt to detect this situation.

5.4.5 Flushing the Cache

Flushing the cache sets it to an initial state which is identical to the state of the cache following reset. The cache is empty and all 32 P flags are set to 0.

The cache is flushed by setting the CF (cache flush) bit in the HSTCTL register to 1. The CF bit retains the last value loaded until a new value is loaded or until the TMS34010 is reset. The contents of the cache remain flushed as long as the CF bit is set to 1. All instruction fetches bypass the cache and are accessed directly from memory.

Unless the cache is disabled, normal cache operation will resume when the CF bit is set to 0.

One use for flushing the cache is to facilitate downloading new code from a host processor to TMS34010 local memory. The host typically halts the TMS34010 during downloading by writing a 1 to the HLT bit in the HSTCTL register. Before allowing the TMS34010 to execute downloaded code, the host should flush the cache to purge it of stale instructions.

For performance reasons, the CD bit should not remain set to 1 for long periods. While CD=1, each instruction-word fetch is interpreted as a cache miss, causing the four words in the subsegment to be fetched from memory. Though the word pointed to by the PC is executed, none of the four words is preserved in cache.

5.4.6 Cache Disable

Disabling the cache facilitates program debugging and emulation. The cache is disabled by setting the CD (cache disable) bit in the CONTROL register to 1. While disabled, the cache is bypassed and all instructions are fetched from external memory.

CD=1 is similar in effect to CF=1, with several exceptions:

- While CD=1 and CF=0, data already in the cache are protected from change. When the CD bit is set back to 0, the state of the cache prior to setting the CD bit to 1 is restored. The instructions in the cache are once again available for execution. If the contents of the cache become invalid while CD=1, they can be flushed by setting CF to 1.

- While CD=1 and CF=0, each instruction word is fetched from memory as it is requested, but the other three words in the subsegment are not fetched. In contrast, if CF=1 and CD=0, all four words in the subsegment that contain the requested instruction word are fetched, although all but the requested word are immediately discarded.

The CD bit can be manipulated to preserve code in the cache for faster execution in some time-critical applications. For example, if an inner loop just exceeds 256 bytes, most of the loop, but not all of it, can fit in the cache. During execution of the few instructions that are not in the cache, the CD bit can be set to 1 to prevent the code in the cache from being replaced. In this instance, the loop's execution speed is improved by eliminating the thrashing of cache contents. Use this technique carefully; in some cases, it can negatively affect execution speed.

5.4.7 Performance with Cache Enabled versus Cache Disabled

When the instruction cache is disabled, instruction words are fetched from external memory. Assuming no wait states are necessary, each instruction fetch from external memory adds 3 machine cycles to the access time. This is considerably slower than a program which uses the cache efficiently (when each word in cache is used several times before it is replaced).

A less efficient use of cache occurs when words in cache are used only once before replacement. This produces a cache miss every fourth word (even in this case, operation is usually much better than operation when the cache is disabled). With the cache enabled, the time penalty due to cache misses in this case is 2.25 machine states per single-word instruction (compare this to 3 states when the cache is disabled), which is calculated as follows:

- Eight machine cycles are required to load four words into cache from memory.
- An additional machine state is required to start processing the instruction.
- Dividing the total of nine machine states by four instruction words yields an average of 2.25 machine states per instruction word.

Performance using the cache is nearly always better than performance with the cache disabled. There are two exceptions. The first occurs when the code contains so many jumps that only a portion of each subsegment is executed before control is transferred to another subsegment. The second occurs when

an inner loop is larger than the cache, in which case only some portion of the instructions in the inner loop can be contained in the cache at any time. In this case, performance may be improved by manipulating the CD bit as described in Section 5.4.6.

While the cache is disabled, the TMS34010's internal memory controller fetches each instruction word from memory only as it is requested by the internal execution unit. This differs from operation with the cache enabled, in which case a cache miss causes the entire four-word subsegment containing the requested instruction word to be loaded into the cache at once.

5.5 Internal Parallelism

Figure 5-8 illustrates the internal data paths associated with TMS34010 processor functions. The TMS34010 has a single, logical memory space for storage of both data and instructions. However, internal parallelism provides the TMS34010 with the benefits found in architectures which contain separate data and instruction storage (sometimes referred to as *Harvard architectures*). The ability to fetch instructions from cache in parallel with data accesses from memory greatly enhances execution speed. Hardware parallelism allows the following three storage areas to be accessed simultaneously:

- Instruction cache
- Dual-ported, general-purpose register files A and B
- External memory

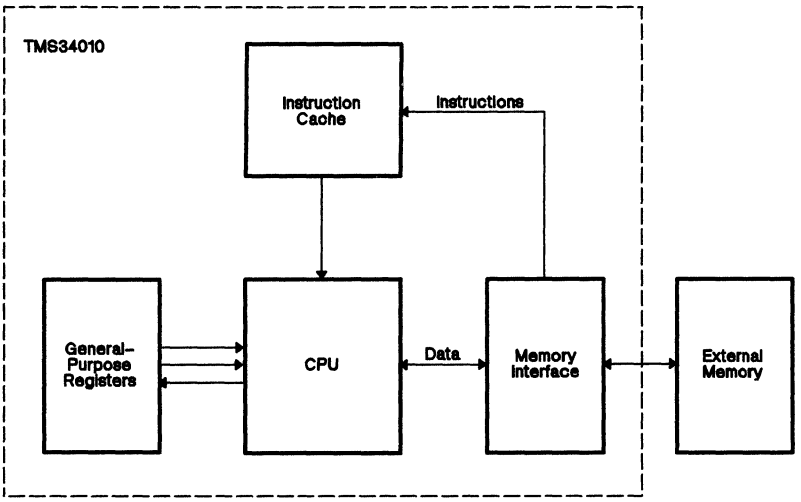


Figure 5-8. Internal Data Paths

Each storage area can also be accessed independently of the other two. This allows the TMS34010 to perform the following actions in parallel during a pair of machine states:

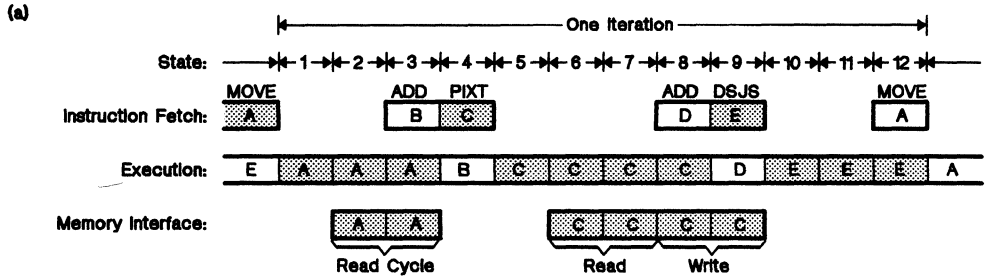
- One external memory cycle
- Two instruction fetches from cache
- Four reads and two writes to the general-purpose register files

The need to schedule conflicting internal operations can limit the TMS34010's ability to perform these actions in parallel. For example, an instruction which requires the memory controller to perform a read must finish executing before the next instruction can be executed.

Figure 5-9 illustrates an example of internal parallelism. Figure 5-9 *a* shows three activities occurring in parallel:

- Instructions are fetched from cache.
- Instructions are executed through the general-purpose registers and the ALU.
- The local memory interface controller performs memory accesses.

Figure 5-9 *a* represents execution of the code in Figure 5-9 *b*, which is the inner loop of a graphics routine. The memory controller accesses pixels while the ALU fetches instructions from cache. The memory controller completes a write cycle while execution begins on the next instruction.



(b)

```
A     L1:   MOVE    *B1+,B10,0    Get DELTAX
B           ADD     B10,B8        Adjust pixel pointer
C           PIXT    *B1,*B8       Draw next pixel
D           ADD     B0,B1         Add field size
E           DSJS    B11,L1        Loop N Times
```

Figure 5-9. Parallel Operation of Cache, Execution Unit, and Memory Interface

Section 6

I/O Registers

The TMS34010's 28 on-chip I/O registers control and monitor the following functions:

- Host interface communications
- Local memory interface control
- Interrupt control
- Video timing and screen refresh

This section describes these functions, I/O register addressing, and then provides an alphabetical presentation of the I/O registers:

Section **Page**

6.1 I/O Register Addressing

TMS34010 I/O registers occupy addresses C0000000h to C00001FFh. These registers can be directly accessed by the TMS34010; they can also be indirectly accessed by a host processor through the host interface registers. For example, the host processor can indirectly read the contents of the PSIZE register by loading the address C0000150h into the HSTADRL and HSTADRH registers, and reading the HSTDATA register. Figure 6-1 illustrates the I/O register memory map.

C00001F0h	REFCNT	DRAM Refresh Count
C00001E0h	DPYADR	Display Address
C00001D0h	VCOUNT	Vertical Count
C00001C0h	HCOUNT	Horizontal Count
C00001B0h	DPYTAP	Display Tap Point
C00001A0h C0000190h C0000180h C0000170h	Reserved	
C0000160h	PMASK	Plane Mask
C0000150h	PSIZE	Pixel Size
C0000140h	CONVDP	Destination Conversion Pitch
C0000130h	CONVSP	Source Conversion Pitch
C0000120h	INTPEND	Interrupt Pending
C0000110h	INTENB	Interrupt Enable
C0000100h	HSTCTLH	Host Control (MSBs)
C00000F0h	HSTCTLL	Host Control (LSBs)
C00000E0h	HSTADRH	Host Address (MSBs)
C00000D0h	HSTADRL	Host Address (LSBs)
C00000C0h	HSTDATA	Host Data
C00000B0h	CONTROL	Control
C00000A0h	DPYINT	Display Interrupt
C0000090h	DPYSTRT	Display Start
C0000080h	DPYCTL	Display Control
C0000070h	VTOTAL	Vertical Total
C0000060h	VSBLNK	Vertical Start Blank
C0000050h	VEBLNK	Vertical End Blank
C0000040h	VESYNC	Vertical End Sync
C0000030h	HTOTAL	Horizontal Total
C0000020h	HSBLNK	Horizontal Start Blank
C0000010h	HEBLNK	Horizontal End Blank
C0000000h	HESYNC	Horizontal End Sync

Figure 6-1. I/O Register Memory Map

The two MSBs of an I/O register's 32-bit internal address are not output on the TMS34010 pins; however, the address is fully decoded internally. Thus, the two MSBs of a 32-bit address must both be 1s for an address to be recognized as that of an I/O register. When an I/O register is accessed, the accompanying memory cycle (as seen at the TMS34010 pins) is altered so that the row address strobe is output, but the column address strobe is inhibited. This is true whether the access is initiated directly by the TMS34010 or indirectly by a host processor.

An access of any address in the range C0000000h–C00001FFh is decoded as an access of an on-chip register location, and the column address strobe remains inactive high through the cycle. An access of any location outside this range is treated as an access of an external memory location.

All I/O registers, with one exception, are cleared to 0 at reset. The exception is the HLT (halt) bit in the HSTCTL register, which is set depending on the value at the $\overline{HCS}$ input pin at the end of the reset pulse:

- If $\overline{HCS}$ is high at reset, the HLT bit is set to 1
- If $\overline{HCS}$ is low at reset, the HLT bit is set to 0

6.2 Latency of Writes to I/O Registers

When an instruction alters the contents of an I/O register, the memory write cycle that modifies the register may not be completed before execution of the next instruction begins. If the second instruction relies on the I/O register value loaded by the first instruction, the second instruction may cause incorrect results. This type of problem could occur, for example, if a PIXBLT instruction were immediately preceded by a MOVE register-to-memory instruction that modified the CONTROL register. This situation is easily avoided by ensuring that the write to the I/O register is allowed to complete before the I/O register value is used as an implied operand by a subsequent instruction. For example, by immediately following a write to an I/O register with a read of the register, the write is certain to have been completed by the time subsequent instructions begin execution.

Internal to the TMS34010, the memory controller operates semi-autonomously with respect to the execution unit that processes instructions. Parallelism between the execution unit and memory controller may allow a write initiated by an instruction to be completed only after one or more subsequent instructions have been executed. An instruction that alters an I/O register (or any other address in memory) transmits its request for a write cycle to the memory controller. Once the request is accepted, the memory controller is responsible for completing the write cycle; in the meantime, execution of the next instruction can begin.

A field insertion request submitted to the memory controller can take as many as five cycles to complete in the case in which a field of 18 or more bits straddles two word boundaries. This case requires a read-modify-write operation to one word, a write to a second word, and a read-modify-write operation to a third word. Although this would be an unusual way of altering locations in the I/O register file, it represents the theoretical worst case number of memory cycles for a field insertion.

The start of a pending field-insertion cycle may be delayed by the following conditions:

- Screen-refresh cycle
- DRAM-refresh cycle
- Host-indirect read or write cycle
- Wait states required for slower memories
- Hold request from an external device

Any uncertainty as to whether a pending write to memory has been completed can be eliminated by making use of the fact that only one field insertion request can be queued at the memory controller at a time. An instruction that requests a second memory access before the earlier field insertion has been completed will be forced to wait. Hence, by following an instruction that alters an I/O register with an instruction that requests a second memory access (*any* memory access), the I/O register is certain to have been updated before the second instruction finishes executing.

6.3 I/O Registers Summary

Table 6-1 summarizes the I/O registers. Descriptions of the four categories of I/O registers follow the table.

Table 6-1. I/O Registers Summary

Host Interface Registers		
Register	**Address**	**Description**
HSTADRH	C00000E0h	**Host interface address, high word.** Contains the 16 MSBs of a 32-bit pointer address used by a host processor for indirect accesses of TMS34010 local memory.
HSTADRL	C00000D0h	**Host interface address, low word.** Contains the 16 LSBs of a 32-bit pointer address used by a host processor for indirect accesses of TMS34010 local memory.
HSTCTLH	C0000100h	**Host interface control, high byte** Contains seven programmable bits that control host interface functions: NMI (bit 8) – Nonmaskable interrupt NMIM (bit 9) – NMI mode bit INCW (bit 11) – Increment pointer address on write INCR (bit 12) – Increment pointer address on read LBL (bit 13) – Lower byte last CF (bit 14) – Cache flush HLT (bit 15) – Halt TMS34010 execution Bits 0 through 7 and 10 are reserved
HSTCTLL	C00000F0h	**Host interface control, low byte.** Contains eight programmable bits that control host interface functions: MSGIN (bits 0–2) – Input message buffer INTIN (bit 3) – Input interrupt bit MSGOUT (bits 4–6) – Output message buffer INTOUT (bit 7) – Output interrupt bit Bits 8 through 15 are reserved
HSTDATA	C00000C0h	**Host interface data.** Buffers data transferred between TMS34010 local memory and a host processor.
Local Memory Interface Registers		
Register	**Address**	**Description**
CONTROL†	C00000B0h	**Memory control.** Contains several parameters that control local memory interface operation: RM (bit 2) – DRAM refresh mode RR (bits 3–4) – DRAM refresh rate T (bit 5) – Transparency enable W (bits 6–7) – Window violation detection mode PBH (bit 8) – PixBlt horizontal direction PBV (bit 9) – PixBlt vertical direction PPOP (bits 10–14) – Pixel processing operation select CD (bit 15) – Cache disable Bits 0 and 1 are reserved
CONVDP†	C0000140h	**Destination pitch conversion factor.** Used during XY to linear conversion of a destination memory address.
CONVSP†	C0000130h	**Source pitch conversion factor.** Used during XY to linear conversion of a source memory address.

† Implied graphics operands

Table 6-1. I/O Registers Summary (Continued)

Local Memory Interface Registers (Continued)		
Register	**Address**	**Description**
PMASK†	C0000160h	**Plane mask register**. Selectively enables/disables the various planes in the bit map of a display system in which each pixel is represented by multiple bits.
PSIZE†	C0000150h	**Pixel size register**. Specifies the pixel size (in bits). Possible pixel sizes include 1, 2, 4, 8, and 16 bits.
REFCNT	C00001F0h	**Refresh count register**. Generates the addresses output during DRAM refresh cycles and counts the intervals between successive DRAM refresh cycles: RINTVL (bits 2–7) – Refresh interval counter ROWADR (bits 8–15) – Row address Bits 0 and 1 are reserved
Interrupt Control Registers		
Register	**Address**	**Description**
INTENB	C0000110h	**Interrupt enable**. Contains the interrupt mask used to selectively enable/disable the three internal and two external interrupts: X1E (bit 1) – External interrupt 1 enable X2E (bit 2) – External interrupt 2 enable HIE (bit 9) – Host interrupt enable DIE (bit 10) – Display interrupt enable WVE (bit 11) – Window violation interrupt enable Bits 0, 3 through 8, and 12 through 15 are reserved
INTPEND	C0000120h	**Interrupt pending**. Indicates which interrupt requests are currently pending: X1P (bit 1) – External interrupt 1 pending X2P (bit 2) – External interrupt 2 pending HIP (bit 9) – Host interrupt pending DIP (bit 10) – Display interrupt pending WVP (bit 11) – Window violation interrupt pending Bits 0, 3 through 8, and 12 through 15 are reserved
Video Timing and Screen Refresh Registers		
Register	**Address**	**Description**
DPYADR	C00001E0h	**Display address**. Counts the number of scan lines output between successive screen refresh cycles and contains the source of the row and column addresses output during a screen refresh cycle: LNCNT (bits 0–1) – Scan line counter SRFADR (bits 2–15) – Screen refresh address
DPYCTL	C0000080h	**Display control**. Contains several parameters that control video timing signals: HSD (bit 0) – Horizontal sync direction DUDATE (bits 2–9) – Display address update ORG (bit 10) – Screen origin select SRT (bit 11) – VRAM serial-register transfer enable SRE (bit 12) – Screen refresh enable DXV (bit 13) – Disable external video NIL (bit 14) – Noninterlaced video enable ENV (bit 15) – Enable video Bit 1 is reserved.
DPYINT	C00000A0h	**Display interrupt**. Specifies the next scan line that will cause a display interrupt request.

† Implied graphics operands

Table 6-1. I/O Registers Summary (Concluded)

Video Timing and Screen Refresh Registers (Continued)		
Register	**Address**	**Description**
DPYSTRT	C0000090h	**Display start address**. Provides control of the automatic memory-to-register cycles necessary to refresh a screen: LCSTRT (bits 0–1) – Specifies the number of scan lines to be displayed between screen refreshes SRSTRT (bits 2–15)– Starting screen-refresh address
DPYTAP	C00001B0h	**Display tap point address**. Contains a VRAM tap point address output during shift register transfer cycles.
HCOUNT	C00001C0h	**Horizontal count**. Counts the number of VCLK periods per horizontal scan line.
HEBLNK	C0000010h	**Horizontal end blank**. Designates the endpoint for horizontal blanking.
HESYNC	C0000000h	**Horizontal end sync**. Specifies the endpoint of the horizontal sync interval.
HSBLNK	C0000020h	**Horizontal start blank**. Specifies the starting point of the horizontal blanking interval.
HTOTAL	C0000030h	**Horizontal total**. Specifies the total number of VCLK periods per horizontal scan line.
VCOUNT	C00001D0h	**Vertical count**. Counts the horizontal scan lines in a video display.
VEBLNK	C0000050h	**Vertical end blank**. Specifies the endpoint of the vertical blanking interval.
VESYNC	C0000040h	**Vertical end sync**. Specifies the endpoint of the vertical sync pulse.
VSBLNK	C0000060h	**Vertical start blank**. Specifies the starting point of the vertical blanking interval.
VTOTAL	C0000070h	**Vertical total**. Specifies the value of VCOUNT at which the vertical sync pulse begins.

6.3.1 Host Interface Registers

Five I/O registers are dedicated to host interface communications, allowing the TMS34010 to:

- Directly transfer status messages or command information
- Indirectly transfer large blocks of data through local memory
- Receive interrupt requests from a host processor
- Transfer interrupt requests to a host processor

The ability to indirectly transfer large blocks of data makes the host interface extremely flexible. For example, a host can transfer blocks of commands to the TMS34010, can halt the TMS34010 temporarily to download a new program for the TMS34010 to execute, or can read blocks of graphics data generated by the TMS34010.

The host interface registers occupy five TMS34010 register locations, and are typically mapped into four consecutive 16-bit locations in the memory or I/O address space of the host processor. The host processor accesses the

HSTCTLL and HSTCTLH registers as the eight LSBs and eight MSBs, respectively, of a single location (the HSTCTL register).

The HSTCTL (host control) register controls functions such as the transfer of interrupt requests and 3-bit status codes between a host processor and the TMS34010. These requests are typically used by software to coordinate the transfer of large blocks of data through TMS34010 local memory. The HSTCTL register also allows the host to flush the instruction cache, halt TMS34010 execution, and transmit nonmaskable interrupt requests to the TMS34010.

The host processor uses the remaining three host interface registers to indirectly access selected data blocks within TMS34010 local memory. The HSTADRL and HSTADRH registers contain a 32-bit address that points to the current word location in memory. The HSTDATA register buffers data transferred to and from the memory under control of the host processor. The host interface can be programmed to automatically increment the address pointer following each transfer, providing the host with rapid access to a block of sequential locations.

6.3.2 Local Memory Interface Registers

Six of the I/O registers support local memory interface functions such as:

- Frequency of DRAM refresh cycles
- Type of DRAM refresh cycles
- Pixel size
- Color plane masking
- Various pixel access control parameters

6.3.3 Interrupt Interface Registers

Two I/O registers monitor and mask interrupt requests to the TMS34010. These include two external and three internal interrupts. External interrupt requests are transmitted to the TMS34010 via input pins $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$. The TMS34010 can be programmed to generate an internal interrupt request in response to any of the following conditions:

- *Window violation* – an attempt is made to write a pixel to a location inside or outside a specified window, depending on the selected windowing mode.
- *Host interrupt* – the host processor sets the INTIN interrupt request bit in the HSTCTL register.
- *Display interrupt* – the specified line number in a frame is displayed on the monitor.

A nonmaskable interrupt occurs when the host processor sets the NMI bit in the HSTCTL host interface register. Reset is controlled by a dedicated pin.

6.3.4 Video Timing and Screen Refresh Registers

Fifteen I/O registers support video timing and screen refresh functions. The TMS34010's on-chip CRT timing generator creates the sync and blanking signals used to drive the CRT monitor in a bit-mapped display system. The timing of these signals can be controlled through the appropriate I/O registers, allowing the TMS34010 to support various screen resolutions and interlaced or noninterlaced video.

The TMS34010 directly supports VRAMs (such as the TMS4461) by generating the memory-to-register cycles necessary to refresh the screen of a CRT monitor. Programmable features include the locations in memory to be displayed on the monitor, as well as the number of horizontal scan lines displayed between individual screen-refresh cycles.

The TMS34010 can optionally be programmed to synchronize to externally generated sync signals. This permits TMS34010-created graphics images to be superimposed upon externally-created images. This external sync mode can also be used to synchronize the video timing of two or more TMS34010 devices in a multiple-TMS34010 display system.

6.4 Alphabetical Listing of I/O Registers

The remainder of this section describes the I/O registers individually; they are listed in alphabetical order. Fields within each register are identified and functions associated with each register are discussed.

Bits within I/O registers that are identified as *reserved* are not used by the TMS34010. When read, a reserved bit returns the last value written to it. No control function, however, is affected by this value. All reserved bits are loaded with 0s at reset. A good software practice is to maintain 0s in these bits.

Address C00000B0h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CD	PPOP					PBV	PBH	W		T	RR		RM	reserved	

Fields

Bits	Name	Function
0–1	Reserved	Not used
2	RM	DRAM refresh mode
3–4	RR	DRAM refresh rate
5	T	Pixel transparency enable
6–7	W	Window violation detection mode
8	PBH	PixBlt horizontal direction
9	PBV	PixBlt vertical direction
10–14	PPOP	Pixel processing operation select
15	CD	Instruction cache disable

Description The CONTROL register contains several control parameters used to configure local memory interface operation.

- **RM** (*DRAM refresh mode, bit 2*)

  The RM bit selects the type of DRAM refresh cycle to be performed. Depending on the value of this bit, the TMS34010 performs each DRAM-refresh cycle as either a $\overline{\text{RAS}}$-only cycle or as a $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ cycle. DRAMs and VRAMs that rely on the TMS34010 to generate an 8-bit row address during a refresh cycle typically use the $\overline{\text{RAS}}$-only refresh cycle, while those that generate their own 9-bit row address internally use the $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ refresh cycle.

RM	Description
0	Selects $\overline{\text{RAS}}$-only refresh cycle
1	Selects $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ refresh cycle

- **RR** (*DRAM refresh rate, bits 3 and 4*)

  The RR field controls the frequency of DRAM refresh cycles. The TMS34010 automatically generates DRAM refresh cycles at regular intervals. The duration of the interval is specified by the value of RR. If required, DRAM refreshing can be disabled by setting RR to the appropriate value.

  The initial value of RR after reset is 00_2. No DRAM refresh cycles are performed while the TMS34010 $\overline{\text{RESET}}$ signal is active.

RR	Description
00	Refresh every 32 local clock periods
01	Refresh every 64 local clock periods
10	Reserved code
11	No DRAM refreshing

- **T** (*Pixel transparency, bit 5*)

  The T bit enables or disables the pixel attribute of transparency. When transparency is enabled, a value of 0 resulting from a pixel operation on source and destination pixels is inhibited from overwriting the destination pixel. In the case of a replace operation (PPOP = 0), a source pixel value of 0 is inhibited from overwriting the destination pixel. Disabling transparency allows a pixel value of 0 to be written to the destination.

T	Effect
0	Disable transparency
1	Enable transparency

- **W** (*Window checking, bits 6 and 7*)

  The W field selects the course of action to be taken when a pixel operation will cause a pixel to be written to a location lying either inside or outside the specified window limits. Window checking applies only to attempts to write to pixel locations defined by XY addresses; writes to pixel locations defined by linear memory addresses are not affected. Nonpixel data writes are not affected.

W	Description
00	No pixel writes are inhibited, and no interrupt requests are generated
01	Generate interrupt request on attempt to write to pixel lying inside window, and inhibit all pixel writes
10	Generate interrupt request on attempt to write to pixel lying outside window
11	Inhibit pixel writes outside window, but do not request interrupt

  A request for a window violation interrupt can occur when $W=01_2$ or $W=10_2$. The WVP bit in the INTPEND register is set to 1 to indicate that a window violation has occurred. This in turn causes the TMS34010 to be interrupted if the WVE bit in the INTENB register and the status IE bit are set to 1.

- **PBH** (*PixBlt horizontal direction, bit 8*)

  The PBH bit determines the horizontal direction (increasing X or decreasing X) of pixel processing for the following instructions:

  - PIXBLT XY,XY
  - PIXBLT L,XY

- PIXBLT XY,L
- PIXBLT L,L

PBH	Effect
0	Increment X (move from left to right)
1	Decrement X (move from right to left)

- **PBV** (*PixBlt vertical direction, bit 9*)

The PBV bit determines the vertical direction (increasing Y or decreasing Y) of pixel processing for the following instructions:

- PIXBLT XY,XY
- PIXBLT L,XY
- PIXBLT XY,L
- PIXBLT L,L

PBV	Effect†
0	Increment Y (move from top to bottom)
1	Decrement Y (move from bottom to top)

† Default screen origin assumed

- **PPOP** (*Pixel processing operation, bits 10–14*)

The PPOP field selects the operation to be performed on the source and destination pixels during a pixel operation. The following 16 PPOP codes perform Boolean operations on pixels of 1, 2, 4, 8, and 16 bits.

PPOP	Operation	Description
00000	S → D	Replace destination with source
00001	S AND D → D	AND source with destination
00010	S AND $\overline{D}$ → D	AND source with NOT destination
00011	0 → D	Replace destination with 0s
00100	S OR $\overline{D}$ → D	OR source with NOT destination
00101	S XNOR D → D	XNOR source with destination
00110	$\overline{D}$ → D	Negate destination
00111	S NOR D → D	NOR source with destination
01000	S OR D → D	OR source with destination
01001	D → D	No change in destination†
01010	S XOR D → D	XOR source with destination
01011	$\overline{S}$ AND D → D	AND NOT source with destination
01100	1 → D	Replace destination with 1s
01101	$\overline{S}$ OR D → D	OR NOT source with destination
01110	S NAND D → D	NAND source with destination
01111	$\overline{S}$ → D	Replace destination with NOT source

† Although the destination array is not changed by this operation, memory cycles still occur.

The following six PPOP codes perform arithmetic operations on 4-, 8-, and 16-bit pixels (but not 1 or 2 bits).

PPOP	Operation	Description
10000	D + S → D	Add source to destination
10001	ADDS(D,S) → D	Add S to D with saturation
10010	D - S → D	Subtract source from destination
10011	SUBS(D,S) → D	Subtract S from D with saturation
10100	MAX(D,S) → D	Maximum of source and destination
10101	MIN(D,S) → D	Minimum of source and destination

PPOP codes 10110_2 through 11111_2 are reserved.

Standard addition and subtraction allow the result of the operation to overflow. However, add-with-saturation and subtract-with-saturation (ADDS and SUBS) do not allow overflow or underflow. In cases in which addition would allow an overflow, ADDS produces a result whose value is all 1s. In cases in which subtraction would allow an underflow, SUBS produces a result whose value is all 0s.

- **CD** (*Cache disable, bit 15*)

The CD bit selectively enables or disables the instruction cache.

CD	Effect
0	Enable instruction cache
1	Disable instruction cache

When the cache is disabled, cache contents (including data, P flags, SSA registers, and so on) remain undisturbed. While the cache remains disabled, all instructions are fetched from memory rather than cache. When the cache is subsequently enabled, its previous state (before it was disabled) is restored. The instructions retained within the cache are once again available for execution.

Address C0000140h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CONVDP															

Description CONVDP is a full 16-bit register that contains a control parameter used during execution of a pixel operation instruction. CONVDP is used with:

- XY addressing
- Window clipping
- PIXBLTs or FILLS (except for PIXBLT L,L) that process pixels from the bottom of the array to the top (PBV=1)

CONVDP is calculated as the result of an LMO instruction whose input operand is the destination pitch value in register B3 (DPTCH). The following assembly code calculates the CONVDP value.

```
LMO  B3,A0          ; Convert DPTCH value
MOVE A0,@CONVDP,0   ; Place result in CONVDP register
```

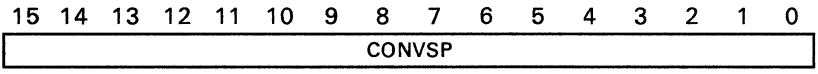
In this example, A0 is used as a scratch register. Constant CONVDP has the value 0C0000140h, and the size of Field 0 is 16 bits.

TMS34010 internal hardware uses the CONVDP value during XY-to-linear conversion of a destination address. PIXBLT and FILL instructions which specify the destination address in XY format use the DPTCH and CONVDP values to convert the XY coordinates to a linear memory address before actually beginning the pixel block move. During a PIXBLT or FILL instruction that requires preclipping of the destination array in the Y direction, the TMS34010 uses the CONVDP value to calculate the effect of the clipped starting Y coordinate on the starting linear address of the destination array. When a PIXBLT instruction's starting Y coordinate is specified to lie in one of the lower two corners of the destination array (when PBV=1), the TMS34010 uses CONVDP to calculate the linear address corresponding to the specified starting coordinates.

The value contained in the five LSBs of CONVDP should be the 1s complement of $\log_2$(DPTCH). When an XY address is specified for the destination, DPTCH must be a power of two; thus, $\log_2$(DPTCH) is an integer. During XY-to-linear conversion, the product of the Y value and the destination pitch is calculated by shifting Y left by $\log_2$(DPTCH).

One instruction, the PIXBLT XY,L instruction, specifies the destination address in linear format but also requires DPTCH to be a power of two. This restriction is necessary when the PBV bit is set to 1.

Address C0000130h



Description CONVSP is a full 16-bit register that contains a control parameter used during execution of a pixel operation instruction. CONVSP is used with:

- XY addressing
- Window clipping
- PIXBLTs or FILLS (except for PIXBLT L,L) that process pixels from the bottom of the array to the top (PBV=1)

CONVSP is calculated as the result of an LMO instruction whose input operand is the source pitch value in register B1 (SPTCH). The following assembly code calculates the CONVSP value

```
LMO  B1,A0          ; Convert SPTCH value
MOVE A0,@CONVSP     ; Place result in CONVSP register
```

In this example, A0 is used as a scratch register. Constant CONVSP has the value 0C0000130h, and the size of Field 0 is 16 bits.

TMS34010 internal hardware uses the CONVSP value during XY-to-linear conversion of a source address. PIXBLT and FILL instructions which specify the source address in XY format use the SPTCH and CONVSP values to convert the XY coordinates to a linear memory address before actually beginning the pixel block move. During a PIXBLT or FILL instruction that requires preclipping of the destination array in the Y direction, the starting source address is modified to accommodate the resulting changes to the starting destination address. When a PIXBLT instruction's starting Y coordinate is specified to lie in one of the lower two corners of the destination array (when PBV=1), the TMS34010 uses CONVSP to calculate the linear address at the corresponding corner of the source array.

The value contained in the five LSBs of CONVSP should be the 1s complement of $\log_2$(SPTCH). When an XY address is specified for the source, SPTCH must be a power of two; thus, $\log_2$(SPTCH) is an integer. During XY-to-linear conversion, the product of the Y value and the source pitch is calculated by shifting Y left by $\log_2$(SPTCH).

Two instructions that specify the source address in linear format also require SPTCH to be a power of two. This is necessary when window clipping is required during execution of either of the following instructions:

- PIXBLT B,XY
- PIXBLT L,XY

It is also necessary when either of these two instructions is executed and the PBV bit in the CONTROL register is set to 1. If PBV=0 and window clipping is disabled, or if window clipping is enabled but the specified array does not require preclipping in the Y dimension, CONVSP is not used, and SPTCH is not required to be a power of two.

Address C00001E0h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRFADR														LNCNT	

Fields

Bits	Name	Function
0–1	LNCNT	Scan line counter
2–15	SRFADR	Screen refresh address

Description The 16-bit DPYADR register contains two separate counters that control the generation of screen-refresh cycles. A screen-refresh cycle transfers the video data for a new scan line to the VRAMs' serial data registers.

- **LNCNT** (*Scan line counter, bits 0 and 1*)

  LNCNT counts the number of scan lines output to the screen between successive screen-refresh cycles. Providing explicit control over the line count permits the implementation of systems that do not reload the VRAMs' internal serial data register on every horizontal scan line. The two-bit LNCNT field is loaded from the two-bit LCSTRT field of the DPYSTRT register at the end of each screen-refresh cycle. The value loaded determines whether the next screen-refresh cycle occurs after 1, 2, 3 or 4 scan lines:

  - When LCSTRT = 0, a screen-refresh cycle occurs after every line.
  - When LCSTRT = 1, 2 or 3, a screen-refresh cycle occurs after every 2, 3 or 4 lines, respectively.

- **SRFADR** (*Screen refresh address, bits 2–15*)

  SRFADR is the source of the row and column addresses output during a screen-refresh cycle. The 14 bits of SRFADR are output as logical address bits 10–23 during screen-refresh cycles. During row address time, DPYADR4–DPYADR15 are output on LAD0–LAD11, and 0s are output on the remaining LAD pins (except as modified by the contents of the DPYTAP register). During column address time, DPYADR2–DPYADR7 are output on LAD6–LAD11 and 0s are output on the remaining LAD lines. Following the completion of each screen-refresh cycle, the value in SRFADR is decremented by the amount indicated in the DUDATE field of the DPYCTL register.

  The following diagrams illustrate the mapping of bits to LAD0–LAD15 from

  1) The logical address as seen by the programmer **and**
  2) The bits of the DPYADR register

  The bits of a 32-bit logical address are numbered 0 to 31, beginning with the LSB. The 14 MSBs of DPYADR, shown in Figure 6-2, are output as logical address bits 10–23 during a screen-refresh cycle. DPYADR2 corresponds to logical address bit 10, DPYADR3 corresponds to logical address bit 11, and so on.

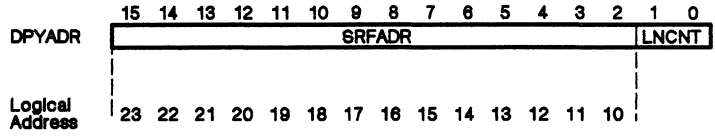


Figure 6-2. Correlation Between SRFADR and Logical Address Bits

Figure 6-3 shows the mapping of logical addresses to LAD0–LAD15 during the row and column address times of the cycle. The symbol *xx* indicates status information output with the row and column addresses.

	LAD Pin Number																
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Logical Row Address Bits	*xx*	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	**Row Address Time**
Corresponding DPYADR bits					15	14	13	12	11	10	9	8	7	6	5	4	
Logical Column Address Bits	*xx*	*xx*	29	28	27	14	13	12	11	10	9	8	7	6	5	4	**Column Address Time**
Corresponding DPYADR bits					7	6	5	4	3	2							

Figure 6-3. Correlation Between DPYADR Bits and Row/Column Addresses

A board designer typically selects eight consecutive address lines from LAD0–LAD11 to connect to the multiplexed address inputs of the VRAMs. For example, by selecting the eight lines LAD2–LAD9, bits 14–21 of the logical address become the row address bits output to the RAMs, and bits 6–13 of the logical address become the column address bits. This means that during a screen-refresh cycle, bits 6–13 of DPYADR become the row address bits output to the RAMs, and bits 4–5 of DPYADR become the two MSBs of the tap point address.

Address C0000080h

Bit Assignments

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ENV	NIL	DXV	SRE	SRT	ORG	DUDATE								Res	HSD

Fields

Bits	Name	Function
0	HSD	Horizontal sync direction
1	Reserved	Not used
2–9	DUDATE	Display address update
10	ORG	Screen origin select
11	SRT	Shift register transfer enable
12	SRE	Screen refresh enable
13	DXV	Disable external video
14	NIL	Noninterlaced video enable
15	ENV	Enable video

Description The DPYCTL register contains several parameters that control video timing signals and serial-register transfer cycles using VRAMs.

- **HSD** (*Horizontal sync direction, bit 0*)

  The HSD bit controls the direction (input or output) of the $\overline{\text{HSYNC}}$ (horizontal sync) pin when the TMS34010 is in external video mode (DXV=0). If HSD=0, $\overline{\text{HSYNC}}$ is configured as an input, the same as $\overline{\text{VSYNC}}$. In this case, the on-chip horizontal sync interval begins when either:

  - The start of the external horizontal sync pulse input at the $\overline{\text{HSYNC}}$ pin is detected, **or**
  - HCOUNT = HTOTAL,

  whichever condition occurs first. $\overline{\text{VSYNC}}$ and $\overline{\text{HSYNC}}$ are configured as inputs or outputs according to the values of the HSD and DXV bits:

HSD	DXV	$\overline{\text{HSYNC}}$	$\overline{\text{VSYNC}}$
0	0	Input	Input
0	1	Output	Output
1	0	Output	Input
1	1	Undefined	

  When $\overline{\text{VSYNC}}$ and $\overline{\text{HSYNC}}$ are both configured as inputs, the on-chip vertical sync interval begins when any of the following conditions occur:

  - The start of the external vertical sync pulse input at the $\overline{\text{VSYNC}}$ pin is detected, **or**
  - VCOUNT=VTOTAL, and the start of the horizontal sync pulse input at the $\overline{\text{HSYNC}}$ pin is detected, **or**
  - VCOUNT=VTOTAL and HCOUNT=HTOTAL.

When $\overline{\text{VSYNC}}$ is an input and $\overline{\text{HSYNC}}$ is an output, the vertical sync interval begins when either the first or third of the listed conditions occurs.

- **DUDATE** (*Display update amount, bits 2–9*)

The DUDATE field indicates the amount by which the SRFADR field in the DPYADR register is incremented (if ORG=0) or decremented (ORG=1) following completion of each memory-to-register cycle used to refresh the screen. DUDATE is loaded with a value containing seven 0s and a single 1. The 1 indicates the bit position at which DPYADR is to be incremented (or decremented if ORG=1).

DUDATE	Increment Size
00000000	0
00000001	1
00000010	2
00000100	4
00001000	8
00010000	16
00100000	32
01000000	64
10000000	128

The increment size is undefined when more than one bit in the DUDATE field is a 1. When interlaced scan mode is enabled, SRFADR is incremented/decremented by half the value indicated in DUDATE at the start of a vertical blanking interval preceding the start of an even field, just after DPYADR2–DPYADR15 have been loaded from DPYSTRT2–DPYSTRT15.

For noninterlaced scanning, DUDATE is programmed to increment the screen address by one scan line. For interlaced scanning, DUDATE is programmed to increment the screen address by two scan lines. Larger increments are typically not used since screen-refresh cycles do not occur more often than once per active scan line.

- **ORG** (*Screen origin select, bit 10*)

The ORG bit controls the origin of the screen coordinate system.

ORG	Effect
0	XY coordinate origin located in upper left corner of screen
1	XY coordinate origin located in lower left corner of screen

If ORG=0 then DPYADR is updated by being incremented by the value in the DUDATE field. If ORG=1 then DPYADR is updated by being decremented by the value in the DUDATE field. Unless explicitly stated otherwise, the discussion in this document assumes that the default origin (ORG=0) is used.

- **SRT** (*Shift-register-transfer enable, bit 11*)

  The SRT bit enables conversion of an ordinary pixel access into a VRAM serial-register transfer cycle.

SRT	Effect
0	Pixel access cycles occur normally
1	Pixel access cycles are converted into VRAM shift-register-transfer cycles

  The TMS34010 instruction set includes several instructions (DRAV, PIXT, LINE, FILL, and PIXBLT) that operate specifically on pixels. By default, SRT=0 and memory accesses performed during accesses of pixel data are the usual memory read and write cycles. When SRT=1, however, accesses of pixel data are converted to shift-register-transfer cycles:

  - A pixel read cycle is converted to a memory-to-register cycle
  - A pixel write cycle is converted to a register-to-memory cycle

  This register-transfer cycle is performed under explicit program control, as opposed to the screen-refresh cycles enabled by the SRE bit, which are automatically generated at regular intervals.

  Uses of the SRT bit include bulk initialization of the entire VRAM array; the entire screen can be cleared to a specified background color in only 256 memory cycles. (While the TMS4461 has this capability, not all VRAMs support this function.) Only pixel accesses are affected by the state of the SRT bit. Instruction fetches and non-pixel data accesses are not altered in any way.

- **SRE** (*Screen-refresh enable, bit 12*)

  The SRE bit enables automatic screen refreshing. Screen refreshes are performed by means of the VRAM memory-to-register cycles which the TMS34010 performs automatically during selected horizontal blanking intervals. The frequency of screen-refresh cycles and the generation of the addresses output during these cycles are programmed by means of the DPYSTRT and DPYCTL registers.

SRE	Effect
0	Disable screen refresh
1	Enable screen refresh

  Changing the value of the SRE bit affects screen refreshes with the start of the next horizontal blanking interval. When SRE changes from 0 to 1, the first screen-refresh cycle occurs at the start of the next horizontal blanking level. When SRE changes from 1 to 0, screen-refresh cycles are disabled beginning at the start of the next horizontal blanking level.

- **DXV** (*Disable external video, bit 13*)

  The DXV bit selects between internally generated or externally generated video timing.

DXV	Effect
0	Selects external video source
1	Selects internally generated video timing

  When DXV=0, the TMS34010 video timing circuitry is programmed to lock onto an external video source. The $\overline{\text{VSYNC}}$ pin is configured as an input and is connected to an external vertical sync signal. If HSD=0, $\overline{\text{HSYNC}}$ is also configured as an input and is connected to an external horizontal sync signal.

  When DXV=1, the TMS34010 generates its own video timing, according to the values loaded into the video timing registers. The $\overline{\text{HSYNC}}$ and $\overline{\text{VSYNC}}$ pins are configured as outputs, and provide the horizontal and vertical sync signals required to drive the video monitor.

- **NIL** (*Noninterlaced video enable, bit 14*)

  The NIL bit selects between an interlaced or a noninterlaced display. The video timing signals output by the TMS34010 are modified according to this selection. The timing differences between interlaced and noninterlaced displays are described in Section 9.

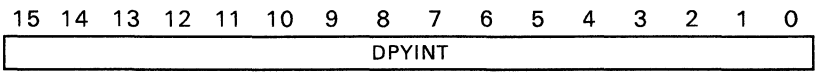
NIL	Effect
0	Selects interlaced video timing
1	Selects noninterlaced video timing

- **ENV** (*Enable video, bit 15*)

  The ENV bit enables or disables the video display. The display remains blanked when ENV=0. During this time, the signal output at the $\overline{\text{BLANK}}$ pin is forced to remain at its active-low level throughout the frame, and setting of the DIP (display interrupt) bit in the INTPEND register is inhibited. (If DIP is already set at the time the ENV is changed from 1 to 0, DIP remains set until explicitly cleared.) When ENV=1, the video display is enabled. The $\overline{\text{BLANK}}$ output signal is controlled according to the parameters contained in the video timing registers, and the DIP bit becomes set when the condition VCOUNT = DPYINT occurs.

ENV	Effect
0	Blank entire screen
1	Enable video

Address C00000A0h

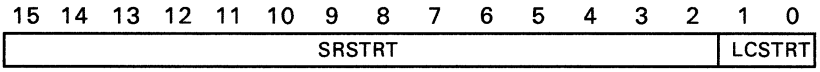


Description The DPYINT register designates the next scan line at which a display interrupt will be requested. This register facilitates the coordination of software activity with the refreshing of selected horizontal lines on the screen of a video monitor.

The contents of DPYINT are compared to the VCOUNT register. When VCOUNT = DPYINT, a display interrupt is requested and the DIP bit in the INTPEND register is set to 1. This coincides with the start of the horizontal blanking interval that marks the end of the line designated by the value contained in DPYINT.

For split-screen applications, a new value can be loaded into the DPYADR register immediately following detection of the 0-to-1 transition of DIP. The new DPYADR value will not affect the line that immediately follows the end of the current horizontal blanking interval, but will affect the next line. The details of this timing are as follows. A screen-refresh cycle may be scheduled to occur at the start of the same horizontal blanking interval during which DIP becomes set. At the end of the screen-refresh cycle, the screen-refresh address in the DPYADR register will be automatically incremented. Requests for screen-refresh cycles have a higher priority than requests for cycles initiated by the on-chip processor. Hence, if the processor loads a new value into DPYADR immediately following detection of DIP's transition from 0 to 1, the value will become the address used for the next screen-refresh cycle, which cannot occur before the next horizontal blanking interval. Between the time that DIP becomes set to 1 and the completion of the next screen-refresh cycle at least one full scan line later, the DPYADR register is guaranteed not to be incremented. Its contents will change during this interval only if it is loaded with a new value under explicit program control. The display interrupt is disabled when the ENV bit in the DPYCTL register is 0.

Address C0000090h



Fields

Bits	Name	Function
0–1	LCSTRT	Starting line count
2–15	SRSTRT	Starting screen-refresh address

Description The DPYSTRT register contains two parameters that control the automatic memory-to-register cycles necessary to refresh the screen.

- **LCSTRT** (*Starting line count, bits 0 and 1*)

  LCSTRT is a two-bit code designating the number of scan lines to be displayed between screen refreshes.

LCSTRT Value	Scan Lines Between Refresh Cycles
0 0	1
0 1	2
1 0	3
1 1	4

  LCSTRT is loaded into the LNCNT field of the DPYADR register at the end of each screen-refresh cycle. LCSTRT is also loaded into LNCNT at the start of the last horizontal blanking interval preceding the first active scan line of a new frame.

- **SRSTRT** (*Starting screen-refresh address, bits 2–15*)

  The 14-bit SRSTRT field contains the starting address loaded into the DPYADR register at the start of each frame. Its value identifies the start of the region of the graphics bit map to be displayed on the screen. SRSTRT is loaded into the SRFADR field of the DPYADR register at the beginning of each vertical blanking interval. (Loading occurs coincides with the start of the horizontal blanking interval at the end of the last active scan line in the frame.)

  The sense of the SRSTRT value depends on the value of the ORG (origin select) bit in the DPYCTL register. When ORG=0, SRSTRT is loaded with the **1's complement** of the starting address. When ORG=1, SRSTRT is loaded with the unmodified starting address. Regardless of the value of the ORG bit, the starting address points to the location in memory of the first pixel output to the screen during each frame. For a typical CRT display, the first pixel of each frame is output to the top left corner of the screen. Refer to the description of the DPYADR register for more information on the generation of screen-refresh addresses.

Address C00001B0h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		DPYTAP													

Fields

Bits	Name	Function
0–13	DPYTAP	Display tap point address
14–15	Reserved	Not used

Description The DPYTAP register contains a VRAM tap point address output during a screen-refresh (memory-to-register) cycle. (The contents of DPYTAP are **not** output during a serial-register transfer initiated under program control while the SRT bit in the DPYCTL register is set to 1.) During a screen-refresh cycle, the 16 bits of the DPYTAP register are bitwise-ORed with the value output at the LAD0–LAD15 pins during the column address time. DPYTAP bit 0 is ORed with LAD0, DPYTAP bit 1 is ORed with LAD1, and so on. This means that the column address output during the cycle is the OR of bits 2–7 of DPYADR and bits 0–15 of DPYTAP.

One application of the DPYTAP register is to permit horizontal panning of the screen over a frame buffer that is wider than the screen. A DPYTAP value of 0 locates the screen at its leftmost position within the frame buffer. Incrementing DPYTAP causes the display to pan to the right through the frame buffer.

DPYTAP is typically used to alter (set to a value other than all 0s) only those column address bits of the SRFADR field of DPYADR that are never incremented. For instance, given a VRAM that requires an 8-bit column address, assume that SRFADR alternately sets the two MSBs of the column address to 00_2, 01_2, 10_2, and 11_2. In this case, DPYTAP should contain 1s only in the bit positions corresponding to the six LSBs of the column address.

Address C00001C0h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HCOUNT															

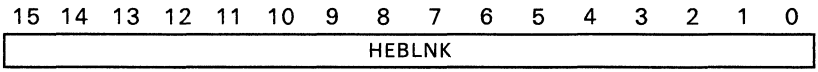
Description The HCOUNT register is a 16-bit counter used in the generation of the horizontal sync and blanking signals. HCOUNT is incremented on the falling edge of the video input clock, and is used to count the number of video clock periods per horizontal scan line. To generate horizontal sync and blanking signals, the value of HCOUNT is compared to the value of the four horizontal timing registers: HESYNC, HEBLNK, HSBLNK, and HTOTAL. When external sync mode is disabled and the value in HCOUNT = HTOTAL, HCOUNT is reset to 0 on the next VCLK falling edge and the $\overline{\text{HSYNC}}$ output is driven active low. HCOUNT is also reset to 0 if the external sync mode is enabled and the input signal $\overline{\text{HSYNC}}$ is driven low.

Two separate, asynchronous elements of the TMS34010 logic can access the HCOUNT register:

- The internal processor, which runs synchronously to local clocks LCLK1 and LCLK2, can access HCOUNT as an I/O register.
- The video timing control logic, which runs synchronously to the video clock VCLK, increments and clears HCOUNT in generating the sync and blanking signals.

No synchronization between these two subsystems is provided, and HCOUNT can only be reliably read or written to while VCLK is held at the logic-high level. HCOUNT is typically not read or written to except during chip test.

Address C0000010h



Description The HEBLNK register is used during the generation of the blanking signal output to the video monitor. The 16-bit value loaded into HEBLNK is compared to HCOUNT, and designates the point at which the horizontal blanking interval ends. The blanking signal output at the $\overline{\text{BLANK}}$ pin is a composite of the internal horizontal and vertical blanking signals. When the value in HCOUNT = HEBLNK, the $\overline{\text{BLANK}}$ output is driven inactive high unless vertical blanking is currently active. Most video monitors require HEBLNK to be set to a value that is less than the value in HSBLNK, but greater than the value in HESYNC.

Address C0000000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HESYNC															

Description The HESYNC register is used during generation of the horizontal sync signal output to the video monitor. The 16-bit value loaded into HESYNC determines the point at which the horizontal sync pulse ends. When the value in HCOUNT = HESYNC, the signal output from the $\overline{\text{HSYNC}}$ pin is driven inactive high to signal the end of the horizontal sync interval. Typical monitors require that HESYNC be set to a value less than the value contained in the HEBLNK register. (However, the HESYNC value is not *required* to be less than the HEBLNK value.) The minimum value of HESYNC is 0.

When external video is enabled and the $\overline{\text{HSYNC}}$ pin is configured as an input, HESYNC should be loaded with a value that ensures that the condition HCOUNT = HESYNC occurs after the external $\overline{\text{HSYNC}}$ signal has gone inactive-high, but before $\overline{\text{HSYNC}}$ goes active low again. For example, a good HESYNC value might be the average of the values in HEBLNK and HSBLNK.

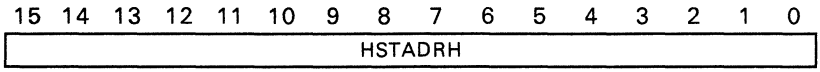
Address C0000020h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HSBLNK															

Description The HSBLNK register is used during generation of the blanking signal output to the video monitor. The 16-bit value in HSBLNK is compared to HCOUNT, and designates the point at which the horizontal blanking interval begins. The blanking signal output at the $\overline{\text{BLANK}}$ pin is a composite of the internal horizontal and vertical blanking signals. When the condition HCOUNT = HSBLNK occurs, the $\overline{\text{BLANK}}$ output is driven from its inactive-high level to its active-low level (unless it is already low due to vertical blanking being active).

Several internal events coincide with the start of horizontal blanking. First, when a screen-refresh cycle is programmed to occur during a particular horizontal scan line, a request for the cycle is sent to the memory controller at the beginning of the horizontal blanking interval that occurs at the end of the line. Second, if a display interrupt request is programmed to occur during a particular horizontal scan line, the request is generated at the start of horizontal blanking. Typical monitors require that HSBLNK be set to a value that is less than the value in HTOTAL, but greater than the value in HEBLNK.

Address C00000E0h



Description The HSTADRH register contains the 16 MSBs of a 32-bit pointer address; the 16 LSBs are contained in HSTADRL. The contents of HSTADRL and HSTADRH are concatenated to form a single 32-bit address during an indirect access by a host processor. The pointer address can be accessed by both the host processor and the TMS34010. The host accesses the pointer address through two 16-bit host interface registers that are mapped into the host's memory or I/O address space.

The four LSBs of the 32-bit pointer address are forced to 0 to point to an even word boundary in memory. If the address pointer is incremented past the largest word address in memory, it will wrap around to the lowest address (all 0s).

When you use the HSTADRH and HSTADRL registers to read data indirectly from the host, be sure that you access them in the correct order. If LBL=0, HSTADRH should be written last. If LBL=1, HSTADRL should be written last.

Note:

When the TMS34010's on-chip processor writes to HSTADRH or HSTADRL, the referenced data is **not** automatically read into HSTDATA. For more information about the host interface, refer to Section 10.

Address C00000D0h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HSTADRL															

Description The HSTADRL register contains the 16 LSBs of a 32-bit pointer address; the 16 MSBs are contained in HSTADRH. The contents of HSTADRL and HSTADRH are concatenated to form a single 32-bit address during an indirect access by a host processor. The pointer address can be accessed by both the host processor and the TMS34010. The host accesses the pointer address through two 16-bit host interface registers that are mapped into the host's memory or I/O address space.

The four LSBs of the 32-bit pointer address are forced to 0 to point to an even word boundary in memory. If the address pointer is incremented past the largest word address in memory, it will wrap around to the lowest address (all 0s).

When you use the HSTADRH and HSTADRL registers to read data indirectly from the host, be sure that you access them in the correct order. If LBL=0, HSTADRH should be written last. If LBL=1, HSTADRL should be written last.

Note:

When the TMS34010's on-chip processor writes to HSTADRH or HSTADRL, the referenced data is **not** automatically read into HSTDATA. For more information about the host interface, refer to Section 10.

Address C0000100h

Bit Assignments

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HLT	CF	LBL	INCR	INCW	Res	NMIM	NMI	Reserved							

Fields

Bits	Name	Function
0–7	Reserved	Not used
8	NMI	Nonmaskable interrupt
9	NMIM	Mode bit for NMI
10	Reserved	Not used
11	INCW	Increment pointer address on write
12	INCR	Increment pointer address on read
13	LBL	Lower byte last
14	CF	Cache flush
15	HLT	Halt TMS34010 processing

Description The HSTCTLH register contains seven programmable bits used to control host interface communications. A host processor can access the control bits in the HSTCTLL and HSTCTLH registers as a single host interface register, HSTCTL. The bits of the host interface's HSTCTL register are mapped into two separate I/O register locations in the TMS34010's memory map, HSTCTLL and HSTCTLH, to allow the TMS34010 to alter the bits in one location without affecting the bits in the other.

The HSTCTLH bits can be both written to and read by both the host processor and the TMS34010. Unpredictable results occur if the TMS34010 and host simultaneously write different values to the HSTCTLH bits. Typically only the host alters the bits in HSTCTLH.

- **NMI** (*Nonmaskable interrupt, host to TMS34010, bit 8*)

  The nonmaskable interrupt allows the host processor to redirect the execution flow of TMS34010 processing to an NMI routine, regardless of the current state of the interrupt mask flags. The host writes a 1 to the NMI bit to send a nonmaskable interrupt request to the TMS34010. The interrupt request cannot be disabled, and will always be executed (unless the TMS34010 is reset before it can complete interrupt execution). The interrupt is initiated immediately upon NMI becoming set (at the time the current instruction completes execution, or in the case of a pixel array instruction, at the next interruptible point in the instruction). Once the interrupt is taken, internal logic automatically clears the NMI bit to 0.

  One use of the NMI is to generate a soft reset after the host downloads new program code into TMS34010 memory. Following execution of a nonmaskable interrupt, screen-refresh and DRAM-refresh functions continue unaffected. The contents of internal registers other than the HSTCTL register are not altered by the interrupt, although they can be modified by the NMI service routine.

- **NMIM** (*Nonmaskable interrupt mode, bit 9*)

  The NMI mode bit determines whether or not the context of the interrupted program is saved when a nonmaskable interrupt occurs. When NMIM=0, the context is saved on the system stack before the NMI service routine is executed. When NMIM=1, the context is discarded when the NMI service routine is executed.

  The NMIM=0 mode supports applications such as single stepping of instructions where the status and PC must be preserved between consecutive nonmaskable interrupts. When NMIM=1, a nonmaskable interrupt can be used to simulate a hardware reset in software (using the NMI vector). Saving the context may be of no benefit if either:

  - Control is never to be returned to the interrupt program **or**
  - The integrity of the stack pointer is suspect.

  The nonmaskable interrupt does not cause the I/O registers to be reset. Consequently, if an NMI is used to simulate a hardware reset, the I/O registers should be reset by software within the NMI service routine.

NMI	NMIM	Effect
0	0	No effect
0	1	Undefined
1	0	NMI (save context on stack)
1	1	NMI (discard previous context)

- **CF** (*Cache flush, bit 14*)

  While CF is set to 1, the contents of the instruction cache are flushed. All four P (present) flags in the cache control logic remain forced to 0 as long as CF remains 1. When CF=1, the cache is disabled; instruction words are fetched from local memory one at a time as they are needed for execution by the TMS34010. Normal cache operation resumes when CF is set to 0, assuming the CD bit in the CONTROL register is also 0. When the value of CF is changed from 1 to 0, the cache begins operation in the same initial state as that which immediately follows reset.

  One use of the CF bit is during downloads of new software from the host processor to TMS34010 local memory. By setting CF to 1 and then to 0 again, the host processor forces the TMS34010 to begin to load new instructions into the cache from memory rather than continue execution of stale instructions already contained in the cache. A 0 must be loaded into CF for normal cache operation to resume.

CF	Effect
0	No effect
1	Flush and disable cache

- **LBL** (*Lower byte last, bit 13*)

  The LBL bit specifies whether an indirect access of TMS34010 memory, initiated by a host register access, begins when the upper or lower byte of the register is accessed by the host processor.

  LBL is provided to accommodate host processors with 8-bit data paths. An 8-bit processor must access a 16-bit TMS34010 host interface register as a series of two 8-bit bytes. Processors which access the lower byte (bits 0–7) first and the upper byte (bits 8–15) second should typically set LBL to 0, and those that access bytes in the opposite sequence should set LBL to 1.

  When LBL is 0, a local bus cycle is initiated if:

  - The host writes to the upper byte of HSTADRH, **or**
  - The host reads from or writes to the upper byte of HSTDATA.

  If LBL is 1, a local bus cycle is initiated if

  - The host accesses the lower byte of HSTDATA, **or**
  - The host writes to the lower byte of HSTADRL

  With this capability, the TMS34010 is capable of automatically resolving so called "Little-Endian/Big-Endian" byte addressing incompatibilities between various processors, and promotes software transparency between 8- and 16-bit versions of the same processor architecture (such as the 8088 and 8086).

LBL	Effect
0	Initiate 16-bit local bus cycle on host access of upper byte of HSTDATA, or on load of upper byte of HSTADRH
1	Initiate 16-bit local bus cycle on host access of lower byte of HSTDATA, or on load of lower byte of HSTADRL

- **INCR** (*Increment address before local read, bit 12*)

  The INCR bit controls whether or not the 32-bit address pointer contained in the HSTADRL and HSTADRH registers is incremented before each read.

INCR	Effect
0	Do not increment address pointer before read cycle on local memory bus
1	Increment address pointer before read cycle on local memory bus

  When INCR=1, the 32-bit address contained in registers HSTADRL and HSTADRH is incremented by 16 before being used for the next read of the TMS34010 memory. This means that HSTDATA is updated to the contents of the next sequential word in the local memory in preparation for the next anticipated read of HSTDATA by the host processor. A local read cycle also occurs when the host loads a new address into the HSTADRL and HSTADRH registers, but the address is not incremented in this case. When incrementing is enabled, repeated reads of the HSTDATA register by the

host result in a series of adjacent words in TMS34010 memory being read; otherwise, the same memory word is read each time. Regardless of the value of the INCR bit, each time HSTDATA is read by the host, a new word is automatically read into HSTDATA from the TMS34010's memory.

- **INCW** (*Increment address after local write, bit 11*)

The INCW bit controls whether or not the 32-bit address pointer contained in the HSTADRL and HSTADRH registers is incremented after each write.

INCW	Effect
0	Do not increment address pointer after write cycle on local memory bus
1	Increment address pointer after write cycle on local memory bus

When INCW=1, the 32-bit address contained in registers HSTADRL and HSTADRH is incremented by 16 after being used as the memory write address. When incrementing is enabled, repeated writes to the HSTDATA register by the host cause a series of adjacent words in TMS34010 memory to be modified; otherwise, the same memory word is modified repeatedly. Regardless of the value of the INCW bit, each time HSTDATA is written to by the host, a new cycle is initiated to write the contents of HSTDATA to the TMS34010's memory.

- **HLT** (*Halt TMS34010 program execution, bit 11*)

When the HLT bit is set to 1, the TMS34010 suspends instruction processing at the next instruction boundary. Once halted, the TMS34010 does **not** respond to interrupt requests (including NMI). Local memory refresh and video timing functions continue unaffected while the TMS34010 is halted. When HLT is again set to 0, the TMS34010 continues execution.

While the TMS34010 is halted, external bus-master devices can arbitrate for, obtain, and release control of the local bus via the TMS34010 hold interface. While the TMS34010 is in the hold state, it cannot perform DRAM-refresh or screen-refresh cycles.

The state of the HLT bit immediately following reset is determined by the state of the $\overline{\text{HCS}}$ pin at the time of the low-to-high transition of $\overline{\text{RESET}}$:

- If $\overline{\text{HCS}}$ is low, HLT is set to 0, and the TMS34010 is enabled to begin executing its reset routine.
- If $\overline{\text{HCS}}$ is high, HLT is set to 1, and the TMS34010 is halted.

Both the host processor and TMS34010 can write to the HLT bit; this means the TMS34010 can halt itself by loading a 1 into HLT.

HLT	Effect
0	Allow TMS34010 to run
1	Halt TMS34010 instruction execution

Address C00000F0h

Bit Assignments

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								INT OUT	MSGOUT			INT IN	MSGIN		

Fields

Bits	Name	Function
0–2	MSGIN	Input message buffer
3	INTIN	Input interrupt bit
4–6	MSGOUT	Output message buffer
7	INTOUT	Output interrupt bit
8–15	Reserved	Not used

Description The HSTCTLL register contains eight programmable bits used to control host interface communications. A host processor can access the control bits in the HSTCTLL and HSTCTLH registers as a single host interface register, HSTCTL. The bits of the host interface's HSTCTL register are mapped into two separate I/O register locations in the TMS34010's memory map, HSTCTLL and HSTCTLH, to allow the TMS34010 to alter the bits in one location without affecting the bits in the other.

The HSTCTLH bits can be read by both the host processor and the TMS34010. The following restrictions apply to writes:

- The MSGOUT field can be modified only by the TMS34010.
- The MSGIN field can be modified only by the host.
- The host can write a 1 to the INTIN bit, but writing a 0 has no effect.
- The TMS34010 can write a 0 to the INTIN bit, but writing a 1 has no effect.
- The TMS34010 can write a 1 to the INTOUT bit, but writing a 0 has no effect.
- The host can write a 0 to the INTOUT bit, but writing a 1 has no effect.

Internal arbitration logic permits the TMS34010 and host processor to access HSTCTLL at the same time without hazard. Synchronization of asynchronous signals at the host interface pins is performed internally.

- **MSGIN** (*Message in, host to TMS34010, bits 0–2*)

The MSGIN field buffers a 3-bit interrupt message to the TMS34010 from the host. The MSGIN field can be both written to and read by the host, but only read by the TMS34010. The MSGIN field typically contains a command or status code from the host, which is read by the TMS34010 in response to a host-generated interrupt (INTIN=1). The meaning of this code is defined in the software of the host and TMS34010.

- **INTIN** (*Interrupt in, host to TMS34010, bit 3*)

  The INTIN bit controls the interrupt request to the TMS34010 from the host. To generate an interrupt request, the host processor loads a 1 to INTIN. The TMS34010 deactivates the request by loading a 0 to INTIN. An attempt by the host to load a 0 to INTIN has no effect. Similarly, an attempt by the TMS34010 to load a 1 to INTIN has no effect. A read-only copy of the INTIN bit is available as the HIP bit in the INTPEND register. The HIP bit faithfully represents the state of the INTIN bit at all times.

INTIN	Effect
0	No interrupt request to TMS34010
1	Send interrupt request to TMS34010

- **MSGOUT** (*Message out, TMS34010 to host, bits 4–6*)

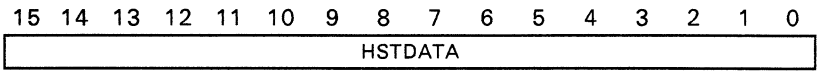
  The MSGOUT field buffers a 3-bit interrupt message to the host from the TMS34010. The MSGOUT field can be both written to and read by the TMS34010, but only read by the host. The MSGOUT field permits an interrupt request generated by means of the INTOUT bit to be qualified by an additional command or status code, the meaning of which is defined in the software of the host and TMS34010.

- **INTOUT** (*Interrupt out, TMS34010 to host, bit 7*)

  The INTOUT bit controls the interrupt request to the host processor from the TMS34010. An interrupt request is transmitted to the host by means of an active-low level on the $\overline{\text{HINT}}$ pin. When INTOUT is 1, $\overline{\text{HINT}}$ is driven active low; when INTOUT is 0, $\overline{\text{HINT}}$ is driven inactive high. The TMS34010 activates the interrupt request by loading a 1 to INTOUT, and the host deactivates the interrupt request by loading a 0 to INTOUT. An attempt by the TMS34010 to load a 0 to INTOUT has no effect. Similarly, an attempt by the host to load a 1 to INTOUT has no effect.

INTOUT	Effect
0	No interrupt request to host
1	Send interrupt request to host

Address C00000C0h



Description The HSTDATA register buffers data transferred through the host interface between TMS34010 local memory and a host processor. HSTDATA can be accessed by the TMS34010 at address C00000C0h. It is one of the four 16-bit registers that can be accessed by the host register through the TMS34010 host interface. HSTDATA is typically accessed by the host rather than the TMS34010. Using the HSTDATA register, the host can either read the TMS34010's memory or write to it. The host initiates the indirect access through the host interface using the 32-bit pointer address in the HSTADRL and HSTADRH registers. During each indirect access, a 16-bit word is transferred between the HSTDATA register and TMS34010 memory. The host processor can access the contents of the HSTDATA register in one 16-bit data transfer or two 8-bit transfers. When the TMS34010's on-chip processor reads from or writes to HSTDATA, **no** automatic read or write cycle takes place between HSTDATA and the memory word pointed to by HSTADRL and HSTADRH.

Address C0000030h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HTOTAL															

Description The HTOTAL register is used during generation of the horizontal sync signal output to the video monitor from the TMS34010. It determines the duration of each horizontal scan line on the screen in terms of the number of VCLK (video clock) periods. The contents of HTOTAL are compared with the horizontal count in HCOUNT to determine the point at which the horizontal sync pulse begins, which also represents the beginning of a new scan line. HCOUNT counts from 0 to the value contained in HTOTAL. When HCOUNT = HTOTAL, the $\overline{\text{HSYNC}}$ output is driven active low on the next falling edge of the VCLK signal, and HCOUNT is reset to 0 on the same clock edge.

HTOTAL is loaded with a 16-bit value greater than that contained in HSBLNK, but less than or equal to 65535. In interlaced scan mode, the value in HTOTAL should be an odd number (LSB=1) to achieve equal spacing between adjacent scan lines. The total number of VCLK video clocks in each horizontal scan line is calculated as HTOTAL + 1. When external sync mode is enabled (DXV=0) and $\overline{\text{HSYNC}}$ is configured as an input (HSD=0), HTOTAL should be loaded with a value greater than the value of HCOUNT at the point at which the external sync pulse is expected. If the external sync pulse does not occur, HCOUNT will be reset when HCOUNT = HTOTAL.

Address C0000110h

Bit Assignments

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				WVE	DIE	HIE	Reserved						X2E	X1E	Res

Fields

Bits	Name	Function
0	Reserved	Not used
1	X1E	External interrupt 1 enable
2	X2E	External interrupt 2 enable
3–8	Reserved	Not used
9	HIE	Host interrupt enable
10	DIE	Display interrupt enable
11	WVE	Window-violation interrupt enable
12–15	Reserved	Not used

Description The INTENB register contains the interrupt mask used to selectively enable the three internally and two externally generated interrupt requests. The following interrupts are enabled by the INTENB register:

- External interrupts 1 and 2 are generated by active-low signals on the input pins $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$, respectively.
- The host interrupt is generated when the host processor sets the INTIN bit in the HSTCTL register to 1.
- The display interrupt is generated when the vertical count in the VCOUNT register reaches the value contained in the DPYINT register.
- The window-violation interrupt is caused by an attempt to write a pixel to a region of the bit map lying outside the limits of the currently-defined window.

The status register contains a global interrupt enable bit, IE. The INTENB register contains individual interrupt enable bits associated with each of the interrupts (X1E, X2E, HIE, DIE, and WVE). Interrupts are enabled through a combination of setting the IE bit and the appropriate bit in the INTENB register. When IE=0, all interrupts are disabled regardless of the values of the bits in the INTENB register. When IE=1, each interrupt is enabled or disabled according to the corresponding enable bit in the INTENB register (1 enables the interrupt, 0 disables it).

Address C0000120h

Bit Assignments

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				WVP	DIP	HIP	Reserved						X2P	X1P	Res

Fields

Bits	Name	Function
0	Reserved	Not used
1	X1P	External interrupt 1 pending
2	X2P	External interrupt 2 pending
3–8	Reserved	Not used
9	HIP	Host interrupt pending
10	DIP	Display interrupt pending
11	WVP	Window-violation interrupt pending
15–12	Reserved	Not used

Description The INTPEND register indicates which interrupt requests are currently pending. INTPEND's six active bits indicate the status of the following interrupts:

- External interrupts 1 and 2 are generated by active-low signals on the input pins $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$, respectively.
- The host interrupt request is generated when the host processor sets the INTIN bit in the HSTCTL register to 1.
- The display interrupt request is generated when the vertical count in the VCOUNT register reaches the value contained in the DPYINT register.
- The window-violation interrupt request is caused by an attempt to write a pixel to a region of the bit map lying inside or outside the limits of the currently-defined window, depending on the selected windowing mode.

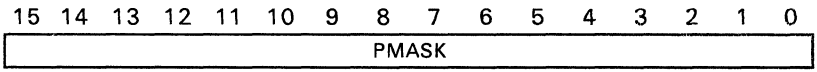
The individual pending bits in the INTPEND register reflect the status of interrupt requests. The interrupt is requested if the corresponding pending bit is 1. There is no request if the pending bit is 0. The status of each interrupt request is reflected in the INTPEND register regardless of whether the interrupt is enabled or not; this allows the TMS34010 to poll interrupts.

The X1P and X2P bits of INTPEND are read only. They reflect the input levels on the $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$ pins, and are not affected when the INTPEND register is written to. The $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$ pins are asynchronous inputs, but the signals to these pins are synchronized internally so that the X1P and X2P bits in the INTPEND register may be reliably read at any time. If an external interrupt is disabled, the interrupt request is ignored, even though the corresponding pending flag in INTPEND is set. The interrupt will be taken by the TMS34010 only if the external request is maintained at the corresponding interrupt request pin until the interrupt is again enabled.

The DIP and WVP bits in the INTPEND register reflect the status of interrupt requests generated by conditions internal to the TMS34010. These two bits are implemented as latches. Once set, DIP or WVP will remain set until a 0 is written to it (or the TMS34010 is reset). Writing a 1 to either of these bits has no effect at any time. While an internal interrupt is disabled, the interrupt request is ignored, even though the corresponding pending flag in INTPEND is set. If the interrupt is subsequently enabled while the interrupt pending flag remains set (because of a prior interrupt request) then the interrupt will be taken by the TMS34010.

The HIP bit in the INTPEND register is a read-only bit that always displays the current contents of the INTIN bit in the HSTCTL register. Writing to the INTPEND register has no effect on the HIP bit. A host interrupt request is generated when the host processor writes a 1 to the INTIN bit of the HSTCTL register. The TMS34010 clears the interrupt request by writing a 0 to the INTIN bit.

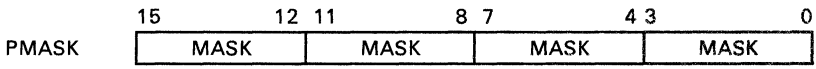
Address C0000160h



Description The PMASK register selectively enables or disables various planes in the bit map of a display system in which each pixel is represented by multiple bits. PMASK contains a 16-bit value that determines which bits of each pixel can be modified during execution of a DRAV, PIXT, FILL, LINE, or PIXBLT instruction. Via the PMASK register, the programmer specifies which bits within each pixel are protected (mask bit=1) and not protected (mask bit=0) from modification. During a pixel write operation, the 0s in the plane mask represent bit positions within the destination pixel that are to be modified by the pixel operation. The 1s in the plane mask represent bit positions in the destination pixel that are protected from modification. During a pixel read operation, the 0s in the mask indicate which bits within a pixel may be read; bits corresponding to 1s in the mask are always read as 0s.

The organization of a display memory is sometimes described in terms of bit planes. If the pixel size is four bits, for example, and the bits in each pixel are numbered from 0 to 3, the display memory is said to be composed of four bit planes, numbered from 0 to 3. Plane 0 contains all the bits numbered 0 from all the pixels, plane 1 contains all the bits numbered 1 from all the pixels, and so on. A 4-bit mask is constructed such that bit 0 of the mask enables (if 0) or disables (if 1) writes to the bits in plane 0, mask bit 1 enables or disables writes to plane 1, and so on.

The plane mask for a 4-bit pixel is four bits; the plane mask for an 8-bit pixel is eight bits; and so on. The plane mask must be replicated throughout the 16 bits of the PMASK register. For example, with four bits per pixel, the PMASK register is loaded with four identical copies of the corresponding 4-bit plane mask, as indicated below.



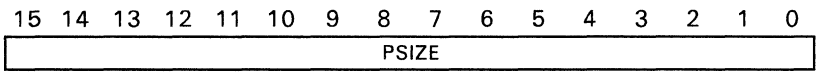
With a pixel size of eight bits, the corresponding 8-bit plane mask is replicated twice – once in bits 0–7 of PMASK, and again in bits 8–15. In general, all 16 bits of the register are used, and a mask for a pixel size of less than 16 bits must be duplicated *n* times, where *n* is 16 divided by the pixel size.

The individual bits of the PMASK register are associated with the corresponding bits of the 16-bit local data bus (data are in fact multiplexed over the same LAD0–LAD15 pins as addresses). PMASK register bit 0 is associated with bit 0 of the data bus (the bit transferred on LAD0), PMASK bit 1 is associated with bit 1 of the data bus, and so on. In general, if PMASK bit *n* is a 0, then bit *n* of the data bus is enabled by the mask; if PMASK bit *n* is a 1, bit *n* is disabled by the mask.

Plane masking is effectively disabled (allowing all bits of each pixel to be modified) by loading all 0s into the PMASK register. This is the default state of PMASK following reset.

To maintain upward compatibility with future versions of the GSP, software drivers should treat the PMASK register as a 32-bit register beginning at address C0000160h. In other words, software should write the plane mask value not only to the 16-bit word at address C0000160h, but also to the word at C0000170h. Writing the second word will have no effect on the TMS34010, but will ensure software compatibility with future graphics processors which may extend the PMASK register from 16 to 32 bits.

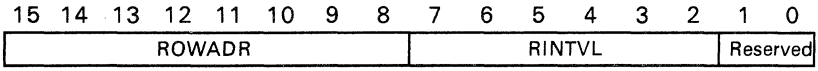
Address C0000150h



Description The PSIZE register is used to specify the pixel size in bits. If the pixel size is four, for example, PSIZE is loaded with the value four. If the pixel size is eight, PSIZE is loaded with the value eight, and so on. All 16 bits of the PSIZE register can be written to or read. Legal pixel sizes are 1, 2, 4, 8, and 16 bits; any other value of PSIZE is undefined.

PSIZE	Pixel Size
0001h	1 bit/pixel
0002h	2 bits/pixel
0004h	4 bits/pixel
0008h	8 bits/pixel
0010h	16 bits/pixel

Address C00001F0h



Fields

Bits	Name	Function
0–1	Reserved	Not used
2–7	RINTVL	Refresh interval
8–15	ROWADR	Row address

Description The REFCNT register generates the addresses output during DRAM refresh cycles and counts the intervals between successive DRAM refresh cycles.

DRAMs require periodic refreshing to retain their data. The TMS34010 automatically generates DRAM refresh cycles at regular intervals. The interval between refresh cycles is programmable. The DRAM refresh mode is selected by loading the appropriate value to the two-bit RR (refresh rate) field in the CONTROL register. DRAM refreshing can be disabled in systems that do not require it. The modes are defined as follows.

RR	Description
00	Refresh every 32 local clock periods
01	Refresh every 64 local clock periods
10	Reserved for future expansion
11	No DRAM refreshing

At reset, the RR field is set to the initial value 00_2. During the time that the reset signal to the TMS34010 is active, no DRAM-refresh cycles are performed.

Bits 2–15 of REFCNT form a continuous binary counter. Bits 2–7 form the RINTVL field, which counts the intervals between successive requests for DRAM-refresh cycles. When RR=01_2, the RINTVL field is decremented by 1 every local clock cycle; that is, the register is decremented at bit 2. This means that RINTVL underflows into ROWADR (a borrow ripples from bit 7 to bit 8 of REFCNT) every 64 local clock cycles. The underflow has two effects:

- ROWADR is decremented by 1 **and**
- A request for a DRAM-refresh cycle is sent to the memory control logic.

When RR=00_2, the RINTVL field is decremented by 2 every local clock period. This means that a DRAM-refresh cycle is generated every 32 local clock periods, twice the rate that results when RR=01_2. When RR=11_2, DRAM refreshing is disabled and no DRAM-refresh cycles occur.

During a DRAM-refresh cycle, the row address output to memory is taken from the 8-bit ROWADR field of REFCNT. Specifically, bits 8–15 of REFCNT are output on LAD0–LAD7. REFCNT bits 8–14 are simultaneously output on LAD8–LAD14. (The $\overline{RF}$ bus status signal is output as a low level on LAD15.) This means that the 8-bit row address needed to refresh a DRAM can be taken from any eight adjacent LAD pins in the range LAD0–LAD14. Note that as ROWADR counts from 255 to 0, the refresh addresses output at the selected eight LAD pins will sequence through all 256 values in the range 255 to 0, though not necessarily in the same order as ROWADR.

REFCNT is set to 0 at reset; after that, refresh address generation is automatic. Typically there is no reason to read this register or write to it, although it can be accessed similarly to the way other I/O registers are accessed. In order to reliably write a value to REFCNT, DRAM refresh should be disabled (by setting RR to 11_2) before writing to REFCNT.

Address C00001D0h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VCOUNT															

Description The VCOUNT register is a 16-bit counter used during generation of the vertical sync and blanking signals. VCOUNT counts the horizontal lines in the video display, incrementing at the same clock edge at which HCOUNT is internally reset to 0. This causes the falling edges of $\overline{\text{HSYNC}}$ and $\overline{\text{VSYNC}}$ to coincide.

In order to generate vertical sync and blanking signals, the value of VCOUNT is compared to the value of the four vertical timing registers, VESYNC, VEBLNK, VSBLNK, and VTOTAL. When HCOUNT = HTOTAL and VCOUNT = VTOTAL at the same time, VCOUNT is reset to 0 on the next VCLK falling edge and the $\overline{\text{VSYNC}}$ output is driven active low.

If interlaced scan mode is enabled and the current field is *even*, and if VCOUNT = VTOTAL and HCOUNT = HTOTAL/2, then VCOUNT is reset to 0 and $\overline{\text{VSYNC}}$ goes low (HCOUNT is not reset until it reaches the value HCOUNT = HTOTAL). When external sync mode is enabled, VCOUNT is reset to 0 when the $\overline{\text{VSYNC}}$ input signal goes active low.

A display interrupt request is generated when VCOUNT = DPYINT. This can be used to coordinate software activity with the refreshing of selected lines on the screen.

Two separate, asynchronous elements of the TMS34010 internal logic can access VCOUNT:

- The internal processor, which runs synchronously to local clocks LCLK1 and LCLK2, can access VCOUNT as an I/O register.
- The video timing control logic, which runs synchronously to the video clock VCLK, increments and clears VCOUNT in the course of generating the sync and blanking signals.

No synchronization between these two subsystems is provided, and VCOUNT can only be reliably read or written while VCLK is held at the logic-high level. VCOUNT is typically not read or written to except during chip test.

Address C0000050h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VEBLNK															

Description VEBLNK is a video timing register that designates the time at which the vertical blanking interval ends. The 16-bit value contained in VEBLNK is compared to VCOUNT to determine when to end the vertical blanking interval. The vertical blanking interval ends when the following conditions are satisfied:

- VCOUNT = VEBLNK
- HCOUNT = HTOTAL

The end of the vertical blanking interval coincides with the start of the horizontal sync, occurring at a time when the internal horizontal blanking signal is active. The blanking signal output from the $\overline{\text{BLANK}}$ pin is a composite of the horizontal and vertical blanking signals generated internally, and will not reach its inactive-high level until both internal blanking signals have become inactive.

When external video is enabled (DXV=0) and the $\overline{\text{HSYNC}}$ pin is configured as an input (HSD=0), the vertical blanking interval ends when the following conditions are satisfied:

- VCOUNT = VEBLNK **and**
- The leading edge of the external horizontal sync pulse is detected

The beginning of the sync pulse is seen as a high-to-low transition at the $\overline{\text{HSYNC}}$ pin.

Typical video monitors require VEBLNK to be set to a value less than the value in VSBLNK, and greater than the value in VESYNC.

Address C0000040h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VESYNC															

Description VESYNC is a video timing register that designates the time at which the vertical sync pulse ends. The 16-bit value contained in VESYNC is compared to VCOUNT to determine when to end the vertical sync pulse. The sync pulse ends when the following conditions are satisfied:

- VCOUNT = VESYNC
- HCOUNT = HTOTAL

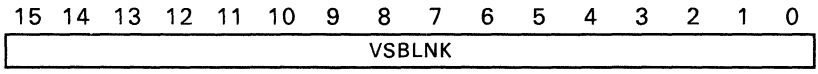
The $\overline{\text{VSYNC}}$ output is driven inactive high to signal the end of the vertical sync interval.

When interlaced mode is enabled and the next vertical field is odd, $\overline{\text{VSYNC}}$ is driven high when VCOUNT = VESYNC and HCOUNT = HTOTAL/2.

Typical video monitors require VESYNC to be set to a value less than the value contained in the VEBLNK register; the minimum value of VESYNC is 0.

When external sync mode is enabled (DXV=0), the end of the external vertical sync pulse is detected as a low-to-high transition at the $\overline{\text{VSYNC}}$ pin, which is configured as an input. VESYNC should be loaded with a value greater than the value in VCOUNT at the point at which the external $\overline{\text{VSYNC}}$ input signal should go inactive high, but lower than the value in VCOUNT when the external $\overline{\text{VSYNC}}$ should again become active low. For example, VESYNC could be loaded with the sum of the values in VEBLNK and VSBLNK divided by two.

Address C0000060h



Description VSBLNK is a video timing register that designates the time at which the vertical blanking interval starts. The 16-bit value contained in VSBLNK is compared to VCOUNT to determine when to start the vertical blanking interval. The vertical blanking interval starts when the following conditions are satisfied:

- VCOUNT = VSBLNK
- HCOUNT = HTOTAL

The start of the vertical blanking interval coincides with the start of the horizontal sync, occurring at a time when the internal horizontal blanking signal is active. The blanking signal output from the $\overline{\text{BLANK}}$ pin is a composite of the horizontal and vertical blanking signals generated internally, and reaches its active-low level when either or both internal blanking signals are active.

When external video is enabled (DXV=0) and the $\overline{\text{HSYNC}}$ pin is configured as an input (HSD=0), the vertical blanking interval starts when the following conditions are satisfied:

- VCOUNT = VSBLNK
- The leading edge of the external horizontal sync pulse is detected

The beginning of the horizontal sync pulse is seen as a high-to-low transition at the $\overline{\text{HSYNC}}$ pin.

VSBLNK should be set to a value less than the value in VTOTAL, and greater than the value in VEBLNK.

Address C0000070h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VTOTAL															

Description VTOTAL contains a 16-bit value that designates the value of VCOUNT at which the vertical sync pulse begins. The contents of VTOTAL are compared to VCOUNT to determine when to start the vertical sync pulse. Vertical sync begins when the following two conditions are satisfied:

- VCOUNT = VTOTAL
- HCOUNT = HTOTAL

These conditions cause HCOUNT to begin counting from 0 again.

The $\overline{\text{VSYNC}}$ output is driven active low to signal the start of the vertical sync interval. The high-to-low transitions of $\overline{\text{VSYNC}}$ and $\overline{\text{HSYNC}}$ occur at the same clock edge.

When interlaced mode is enabled and the next vertical field is odd, $\overline{\text{VSYNC}}$ is driven low when VCOUNT = VESYNC and HCOUNT = HTOTAL/2. The total number of horizontal lines in each vertical field is calculated as VTOTAL + 1. In interlaced mode the total number of horizontal lines in both fields of the vertical frame is calculated as 2 x VTOTAL-1.

When external video is enabled (DXV=0), the $\overline{\text{VSYNC}}$ pin is configured as an input rather than an output. The high-to-low transition of $\overline{\text{VSYNC}}$ is recognized as the beginning of the vertical sync pulse, unless the condition VCOUNT = VTOTAL and the start of horizontal sync are detected first. VTOTAL should be loaded with a value at least as large as the value of VCOUNT at which the external sync pulse should begin. Should the external sync pulse not occur, VCOUNT will be reset one VCLK period after the conditions VCOUNT = VTOTAL and HCOUNT = HTOTAL occur.

VTOTAL should be set to a value greater than the value in VSBLNK. The maximum value that can be loaded into VTOTAL is 65535.

Section 7

Graphics Operations

This section provides an overview of the graphics drawing capabilities of the TMS34010. Topics in this section include:

7.1 Graphics Operations Overview

The TMS34010 instruction set provides several fundamental graphics drawing operations:

- The PIXBLT and FILL instructions manipulate two-dimensional arrays of pixels.
- The LINE instruction implements the fast inner loop of the Bresenham algorithm for drawing lines.
- The DRAV (draw and advance) instruction draws a pixel and increments the pixel address by a specified amount. This function supports the implementation of incremental algorithms for drawing circles, ellipses, arcs, and other curves.
- The PIXT (pixel transfer) instruction transfers individual pixels from one location to another.

The PIXBLT instruction plays an important role in rapidly drawing high-quality, bit-mapped text. In particular, the PIXBLT B,XY and PIXBLT B,L instructions expand character patterns stored as bit maps (at one bit per pixel) into color or gray-scale characters of 1, 2, 4, 8 or 16 bits per pixel. This allows character shape information to be stored independently of attributes such as color and intensity, providing greater storage efficiency.

The TMS34010 provides several methods for processing the values of the source and destination pixels before the result is written to the destination. These operations include:

- Boolean and arithmetic pixel processing operations for combining source pixels with destination pixels.
- A plane mask which specifies which bits within pixels can be altered during pixel operations.
- Transparency, an option which permits objects written onto the screen to have transparent regions through which the background is visible.

Pixel processing, plane masking, and transparency can be used simultaneously. These operations on pixel values can be used in combination with any of the pixel drawing instructions listed above. The arithmetic operations are especially important in displays that use multiple bits per pixel to encode color or intensity information. For example, the MAX and MIN operations allow two objects with antialiased edges to be smoothly merged into a single image.

The TMS34010 has features such as automatic window checking to support windowed graphics environments. Three window-checking modes are provided:

- Clipping a figure to fit a rectangular window.
- Requesting an interrupt on an attempt to write to a pixel *outside* of a window.

- Requesting an interrupt on an attempt to write to a pixel *inside* of a window.

The last of these modes can be used to identify screen objects that are pointed to by a cursor. The window checking modes can be used with any of the pixel drawing instructions that use XY addressing. Window checking is optional and can be turned off.

The TMS34010 provides further support for windowed environments by rapidly detecting the following conditions:

- Whether a *point* lies inside or outside a rectangular window.
- Whether a *line* lies entirely inside or entirely outside a window.

Lines that lie entirely outside a window can be trivially rejected, meaning that they take no further processing time. These conditions are detected via the CPW (compare point to window) instruction, which takes only one machine state to compare the XY coordinates of a point to all four sides of a window.

Another operation that occurs frequently in windowed environments is calculating the region where two rectangles intersect. This is a feature available with the PIXBLT and FILL instructions. Based on the window-checking mode, one of two methods can be selected to calculate the region of intersection:

- The destination pixel array is preclipped to a rectangular window before the PixBlt or fill operation begins.
- The intersection of the destination pixel array with a rectangular window is calculated, but no pixels are transferred.

7.2 Pixel Block Transfers

The TMS34010 supports a powerful set of raster operations, known as **PixBlts** (pixel block transfers), that manipulate two-dimensional arrays of bits or pixels. A pixel array is defined by the following parameters:

- A starting address (by default, the address of the pixel with the lowest address in the array)
- A width *DX* (the number of pixels per row)
- A height *DY* (the number of rows of pixels)
- A pitch (the difference between the starting addresses of two successive rows)

A pixel array appears as a rectangular area on the screen. The array pitch is the same in this case as the pitch of the display. The default starting address is the address of the pixel in the upper left corner of the rectangle. (This assumes that the ORG bit in the DPYCTL register and the PBH and PBV bits in the CONTROL register are all set to their default values of 0.)

Two operands must be specified for a PIXBLT instruction:

- A *source* pixel array **and**
- A *destination* pixel array

The two arrays must have the same width and height, although they may have different pitches. Each pixel in the source array is combined with the corresponding pixel of the destination array. A Boolean or arithmetic *pixel processing operation* is selected and applied to the PIXBLT operation. The default pixel processing operation is *replace*. If replace is selected, source pixel values are simply copied into destination pixels.

Before executing a PIXBLT instruction, load the following parameters into the appropriate GSP internal registers:

DYDX Composed of two portions: DX, which specifies the width of the array, and DY, which specifies the height of the array.

PSIZE Pixel size (number of bits per pixel).

SADDR Starting address of source array (XY or linear address).

DADDR Starting address of destination array (XY or linear address).

SPTCH Source pitch, or difference in memory addresses of two vertically adjacent pixels in the source array.

DPTCH Destination pitch, or difference in memory addresses of two vertically adjacent pixels in the destination array.

If either the source or destination array is specified in XY format, the contents of the CONVSP and CONVDP registers will be used in instances in which the Y component of the starting address must be adjusted prior to the start of the

PixBlt. The Y component may require adjustment, either to preclip the array or to select a starting pixel in one of the lower two corners of the array.

Pitches and starting addresses must be specified separately for the two arrays (source and destination). The width, height, and pixel size are common to both arrays. (During a color expand operation, only the destination pixel size is specified; the source pixel size is assumed to be one bit.)

The starting address of a pixel array can be specified as a linear (memory) address or as an XY address. Window checking can be used only when the destination array is pointed to by an XY address.

On-screen objects may be defined as XY arrays but may be more efficiently stored as linear arrays in off-screen memory. An array specified in linear format can be transferred to an array specified in XY format (and vice versa) by means of the PIXBLT L,XY and PIXBLT XY,L instructions.

The FILL instruction fills a specified destination pixel array with the pixel value specified in the COLOR1 register. A fill operation can be thought of as a special type of PixBlt that does not use a source pixel array. The source pixel value used in pixel processing is the value in the COLOR1 register. The destination array of a FILL instruction can be specified in either XY or linear format.

7.2.1 Color-Expand Operation

The TMS34010 allows shape information to be stored separately from attributes such as color and intensity. A shape can be stored in compressed form as a bit map containing 1s and 0s. The color information is added as the shape is drawn to the screen; the 1s in the bit map are expanded to the specified Color 1 value, and the 0s are expanded to the Color 0 value. This saves a significant amount of memory when the pixel size in the display memory is two bits or more.

Two PIXBLT instructions, PIXBLT B,XY and PIXBLT B,L, provide the color-expand capability. The source array for either instruction is a bit map (one bit per pixel) stored off-screen in linear format for greater storage efficiency. The destination array can be specified in either XY or linear format. The pixel size for the destination array is governed by the value in the PSIZE register. The colors to which the 1s and 0s in the source array are expanded are specified in the COLOR1 and COLOR0 registers.

A primary benefit of the color-expand capability is the reduction in table area needed to store text fonts. Font bit maps are stored in compressed form at one bit per pixel. The color-expand operation adds color to a character shape at draw time, allowing color to be treated as an attribute separate from the shape of the character. The alternative would be to store the fonts in expanded form, which can be costly. The amount of table storage necessary to store red letters A–Z, blue letters A–Z, and so on, multiplied by the number of font styles needed for an application program, would be prohibitive. Furthermore, the color-expand operation is inherently faster than using pre-expanded fonts because far fewer bits of character shape information have to be read from the font table when a character is drawn to the screen.

Figure 7-1 shows the expansion of a bit map, one bit per pixel and four bits wide, into four 4-bit pixels (transforming 0-1-1-0 into yellow-red-red-yellow,

for example). Before transferring the expanded source array to the destination array, any of the Boolean or arithmetic pixel processing operations can be applied.

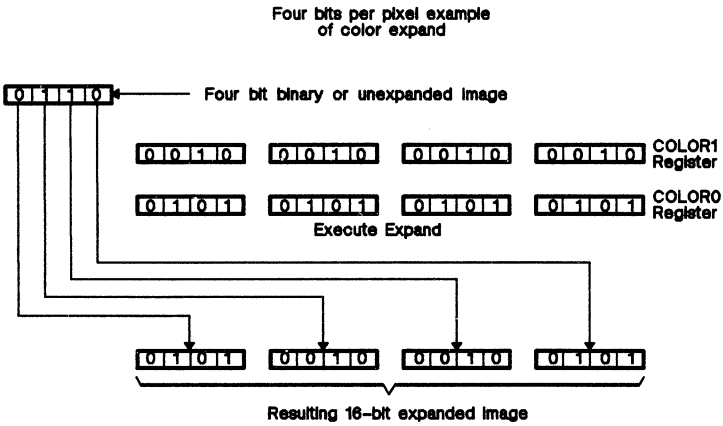


Figure 7-1. Color-Expand Operation

The expand function is also useful in applications that generate shapes or patterns dynamically. During the first stage of this process, a compressed image is constructed in an off-screen buffer area at one bit per pixel. The image is built up of geometric objects such as rectangles, circles or polygons. Patterns can also be added. When complete, the compressed image is color-expanded onto the screen. This method defers the application of color and intensity attributes until the final stage.

Combining color expand with the replace-with-transparency operation yields a new operation that is particularly useful in drawing overlapping or kerned text. The color value used to replace the 0s in the source array is selected by the programmer as all 0s, which is the transparency code. The GSP defers the check for transparency until after the color-expand operation has been performed. As the color-expand operation is performed, the 0s in the source array are expanded to all 0s. Only the pixels in the destination array that correspond to nontransparent pixels in the resulting source array are replaced.

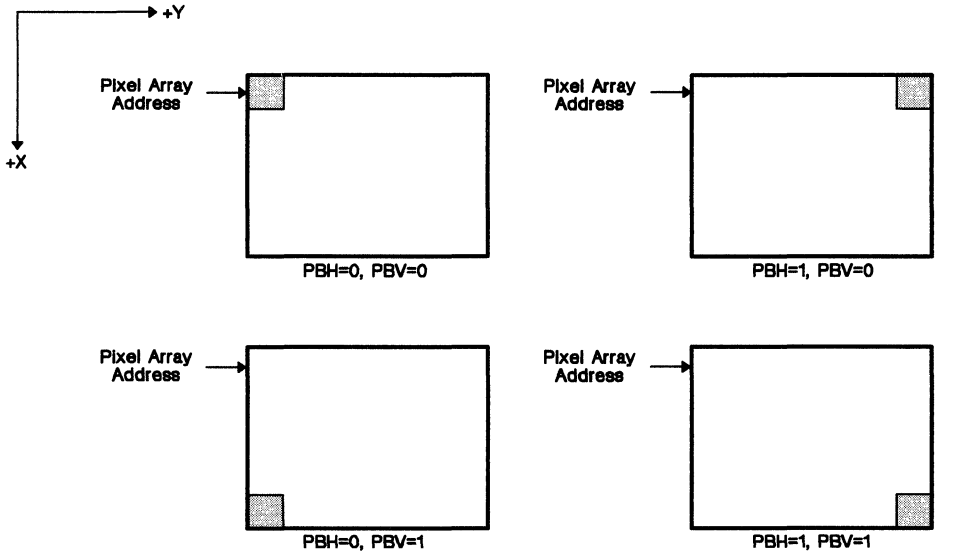
The PIXBLT B,XY and PIXBLT B,L instructions can be used in conjunction with pixel processing, transparency and plane masking. Source pixels are expanded before being processed. Window checking can be used with PIXBLT B,XY.

7.2.2 Starting Corner Selection

The default starting address of a pixel array is the lowest pixel address in the array. When an array is displayed on the screen, as shown in Figure 7-2 *a*, the starting address is the address of the pixel in the upper left corner of the array. (The XY origin is located in its default position at the upper left corner of the screen.) During a PixBlt operation, this pixel is processed first. The PixBlt processes pixels from left to right within each row, beginning at the top row and moving toward the bottom row. The pixel at the lower right corner of the array is processed last.

Certain PixBlt operations allow any of the other three corners to be used as the starting location. This may be necessary, for instance, if the source and destination arrays overlap. The sequence in which pixels are moved when the arrays overlap should be controlled so as to not overwrite the pixels in the source array before they are written to the destination array.

Figure 7-2 shows how the PBV and PBH bits in the CONTROL register determine the starting corner for the PixBlt operation. The starting corner is indicated for each of four cases. PBH selects movement in the X direction, from left to right or right to left. PBV selects movement in the Y direction, from top to bottom or bottom to top.



Note: Starting corners are shaded.

Figure 7-2. Starting Corner Selection

PBH=0 The PixBlt processes pixels from left to right; that is, in the direction of *increasing* X.

PBH=1 The PixBlt processes pixels from right to left; that is, in the direction of *decreasing* X.

PBV=0 The PixBlt processes rows from top to bottom; that is, in the direction of *increasing* Y.

PBV=1 The PixBlt processes rows from bottom to top; that is, in the direction of *decreasing* Y.

All the pixels in one row are processed before moving to the next row.

When one or both of the arrays is specified in XY format, the GSP automatically calculates the actual starting address (specified by PBH and PBV) from the default starting address (that is, the lowest pixel address in the array) and the width and height of the array. Automatic starting address adjustment is available with the following instructions:

- PIXBLT L,XY
- PIXBLT XY,L
- PIXBLT XY,XY

The programmer supplies the *default* starting addresses for these PixBlts in the SADDR and DADDR registers. During the course of instruction execution, SADDR and DADDR are automatically adjusted to the address of the corner selected by PBH and PBV.

When *both arrays are specified in linear format*, the starting addresses of the appropriate corner pixels must be provided by the programmer. The PIXBLT L,L instruction allows any of the four corners to be used as the starting location, but in this case the programmer must adjust the addresses in SADDR and DADDR to the corner selected by PBH and PBV.

7.2.3 Interrupting PixBlts and Fills

PIXBLT and FILL are interruptible instructions. An interrupt can occur during execution of one of these instructions; when interrupt processing is completed, execution of the PIXBLT or FILL resumes at the point at which the interruption occurred.

The execution time of a PIXBLT or FILL instruction depends on the specified pixel array size. In order to prevent high-priority interrupts from being delayed until completion of PixBlts and fills of large arrays, the PIXBLT and FILL instructions check for interrupts at regular intervals during their execution.

When a PIXBLT or FILL instruction is interrupted the PBX (PixBlt executing) status bit is set to 1. This records the fact that the interrupt occurred during a pixel array operation. The PC and the ST are pushed onto the stack, and control is transferred to the appropriate interrupt service routine. At the end of the interrupt service routine, an RETI (return from interrupt) instruction is executed to return control to the interrupted program. The RETI instruction

pops the ST and PC from the stack. When the PBX bit is detected, execution of the interrupted PIXBLT or FILL instruction resumes.

At the time of the interrupt, the state of the PIXBLT or FILL instruction is saved in certain B-file registers. The source and destination address registers contain intermediate values. The source and destination pitches may also contain intermediate values, depending on the instruction. The SADDR, SPTCH, DADDR, DPTCH registers and registers B10–B14 (as well as the original set of implied operands) contain the information necessary to resume the instruction upon return from an interrupt.

If the interrupt routine uses any of these registers, they should be saved on the stack and restored when interrupt processing is complete. By following this procedure, PIXBLT or FILL instructions can be safely executed within interrupt service routines.

Note:

The PBX bit is not set to 1 when a PIXBLT or FILL instruction is aborted due to a window violation.

7.3 Pixel Transfers

The TMS34010 uses the PIXT (pixel transfer) instructions to transfer individual pixels from one location to another. The following pixel transfers can be performed:

- From an A- or B-file register to memory,
- From memory to an A- or B-file register, **or**
- From one memory location to another.

The address of a pixel in memory can be specified in XY or linear format. Linear addresses must be pixel aligned.

The pixel size for all PIXTs is specified by the value in the PSIZE register. Pixel sizes are restricted to 1, 2, 4, 8, or 16 bits to facilitate XY address computations, window checking, transparency, and arithmetic pixel processing.

The PIXT instruction can be used in conjunction with window checking, Boolean or arithmetic pixel processing, plane masking, and transparency.

7.4 Incremental Algorithm Support

The TMS34010 supports incremental drawing algorithms via its DRAV (draw and advance) and LINE instructions. The DRAV instruction is used primarily in the construction of algorithms for incrementally drawing circles, ellipses, arcs, and other curves. The DRAV instruction can also be used in the inner loop of algorithms for drawing straight lines incrementally. Lines, however, are treated as a special case by the TMS34010 in order to achieve even faster drawing rates. A separate instruction, LINE, implements the entire inner loop of the Bresenham algorithm for drawing lines.

The DRAV (draw and advance) instruction draws a pixel to a location pointed to by a register; the pointer register is then incremented to point to the next pixel. The pointer is specified as an XY address. The X and Y portions of the address are incremented independently, but in parallel. The value written to the destination pixel in memory is taken from the COLOR1 register.

The DRAV instruction is embedded in the inner loop of an incremental algorithm to speed up its execution. As an incremental algorithm plots each pixel on a curve, it also determines where the next pixel will be drawn. The next pixel is typically one of the eight pixels immediately surrounding the pixel just plotted on the screen. Advancing in this manner, the algorithm tracks the curve from one end to the other.

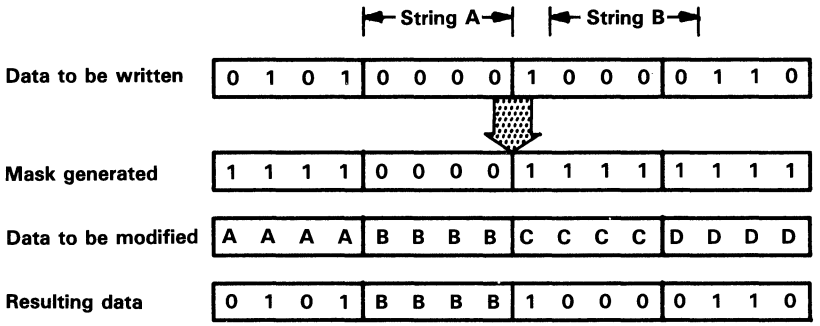
The DRAV and LINE instructions may be used in conjunction with Boolean or arithmetic pixel processing operations, window checking, plane masking and transparency.

7.5 Transparency

When a PixBlt is used to draw an object to the screen, some of the pixels in the rectangular pixel array that contains the object may not be part of the object itself. **Transparency** is a mechanism that allows surrounding pixels in the array to be specified as invisible. This is useful for ensuring that only the object, and not the rectangle surrounding it, is written to the screen.

Transparency is enabled by setting the T bit in the CONTROL register to 1, or disabled by setting the T bit to 0. When enabled, a pixel that has a value of 0 is considered transparent, and will not overwrite a destination pixel. *Transparency detection is applied not to the source pixel values, but to the pixel values resulting from plane masking and pixel processing.* When an operation performed on a pair of source and destination pixels yields a 0 result, the GSP detects this and prevents the destination pixel from being altered. In the case of pixel processing operations such as AND, MIN, and replace, a source pixel value of 0 ensures that the result of the operation will be a transparent pixel.

Figure 7-3 illustrates how transparency works in the GSP. Assuming four bits per pixel, the hardware must detect strings of 0s of length four falling between pixel boundaries. While bit strings ***A*** and ***B*** are both of pixel length, only string ***A*** is detected as transparent. String ***B*** crosses the pixel boundary. The memory interface logic generates an internal mask to govern which bits are modified during a write cycle. This mask contains 0s in the bits corresponding to the transparent pixel. Only destination bits corresponding to 1s in the mask will be modified.



Note: This example assumes four bits per pixel.

Figure 7-3. Transparency

Figure 7-7 (page 7-17) and Figure 7-8 (page 7-20) illustrate several pixel processing operations. Figure 7-8 *h* shows an example of a replace operation performed with transparency enabled. The pixels surrounding the letter **A** pattern in the source array are transparent (all 0s). Compare Figure 7-8 *h* with Figure 7-7 *d*; this replace-with-transparency operation is analogous to the logical OR operation in a one-bit-per-pixel display.

Transparency can be used with any instruction that writes to pixels, including the PIXBLT, FILL, DRAV, LINE, and PIXT instructions. Transparency does not affect writes to non-pixel data.

7.6 Plane Masking

The plane mask is a hardware mechanism for protecting specified bits within pixels. Mask-protected pixels will not be modified during graphics instructions. The plane mask allows the bits within pixels to be manipulated as though the display memory were organized into *bit planes* (or *color planes*) that can selectively be protected from modification. The number of planes equals the number of bits per pixel.

Consider an example in which the pixel size is four bits. The bits within each pixel are numbered 0–3, and belong to planes 0–3, respectively. All the bits numbered 0 in all the pixels form plane 0, all the bits numbered 1 in all the pixels form plane 1, and so on.

The plane mask allows one or more planes to be manipulated independently of the other planes. Given four planes of display memory, for example, three of the planes can be dedicated to eight-color graphics, while the fourth plane can be used to overlay text in a single color. The plane mask can be set so that the text plane can be modified without affecting the graphics planes, and vice versa.

The PMASK register contains the plane mask. Each bit in the plane mask corresponds to a bit position in a pixel. The 1s in the mask designate pixel bits that are protected, while 0s in the mask designate pixel bits that can be modified. Those pixel bits that are protected by the plane mask are always read as 0s during read cycles, and are protected from alteration during write cycles. While no single control bit enables or disables plane masking, it is effectively disabled by setting PMASK to all 0s; this is the default condition following reset.

The logical width of a quantity in the plane mask is the same as the pixel size. However, in order to maintain a consistent effect on all of the pixels within a destination region, regardless of their position within the destination words, you should replicate the mask for a single pixel to fill the entire 16-bit PMASK register. (To provide upward compatibility with future versions of the GSP, you should replicate the plane mask through the 32 bits beginning at address C0000170h.) For example, if the pixel size is four bits, the 4-bit mask is replicated four times within PMASK; in bits 0–3, 4–7, 8–11, and 12–15. These four copies of the mask are applied to the four pixels in a word written to or read from memory. A 16-bit PMASK value for pixels of 1, 2, 8, or 16 bits is constructed similarly by replicating the mask 16, 8, 2, or 1 times, respectively.

The plane mask affects only pixel accesses performed during execution of the PIXBLT, FILL, PIXT, DRAV, and LINE instructions. Data accesses by non-graphics instructions are not affected.

The following list summarizes operation of the PMASK register during pixel reads and writes:

- **Pixel Read:**

  The **0s** in PMASK correspond to unprotected bits in the source pixel that are seen by the GSP to contain the actual values read from memory.

  The **1s** in PMASK correspond to protected bits in the source pixel that are seen as 0s by the GSP, regardless of the values read from memory.

- **Pixel Write:**

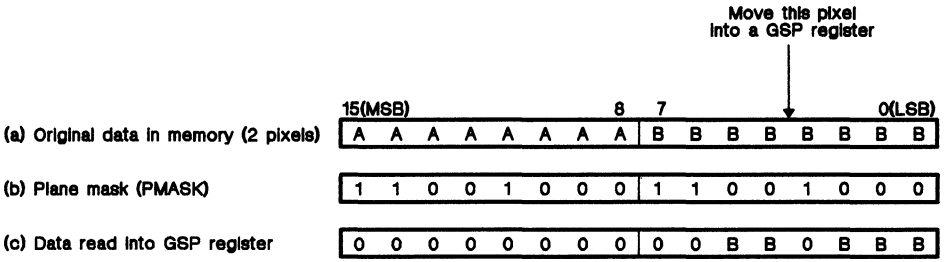
  The **0s** in PMASK specify those bits in the destination pixel in memory which may be altered.

  The **1s** in PMASK specify protected bits in the destination pixel which cannot be altered.

When a pixel is being transferred from a source to a destination location, plane masking is applied to the values read from the source and destination before pixel processing is applied. As the operands are read from memory, the bits protected by the plane mask are replaced with 0s *before* the specified Boolean or arithmetic pixel processing operation is performed. and destination before pixel processing is applied. Transparency detection is performed on the result of this operation. When the result is written back to the destination, those bits of the destination that are protected by the plane mask are not modified.

Source pixels that originate from registers are not affected by the plane mask, and undergo pixel processing in unmodified form. The FILL, DRAV, LINE, PIXT *Rs,*Rd*, and PIXT *Rs,*Rd*.XY instructions obtain their source pixels from registers.

Figure 7-4 shows how special hardware in the local memory interface of the TMS34010 applies the plane mask to pixel data during a read cycle. The pixel size for this example is eight bits per pixel. This could represent the execution of a PIXT **Rs*.XY,*Rd* instruction, for instance.



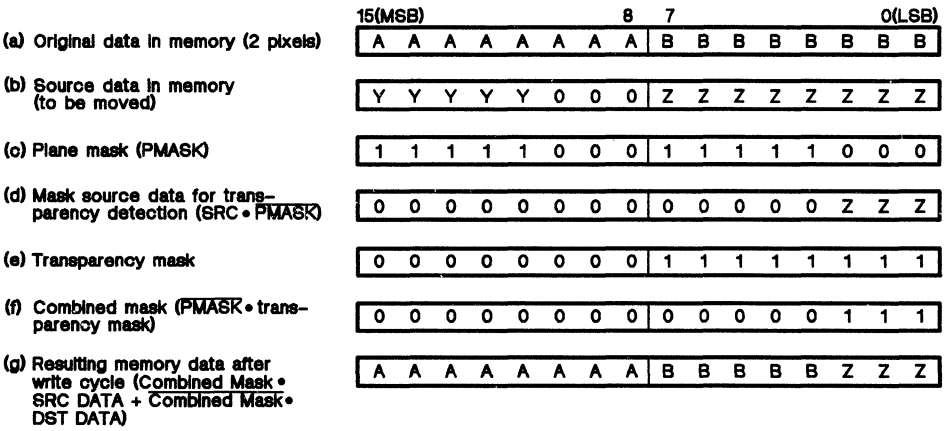
Notes: 1. This example assumes eight bits per pixel.
2. The pixel moved into the GSP register is left justified. All register bits to the left of the pixel are zero filled.

Figure 7-4. Read Cycle With Plane Masking

- Figure 7-4 *a* shows the 16-bit word containing the pixel as it is read from memory.
- The word is ANDed with the inverse of the plane mask shown in *b*.
- The result in Figure 7-4 *c* shows that the bits within the data word that correspond to 1s in the mask have been set to 0s.

After plane masking, the designated pixel is loaded into the eight LSBs of the 32-bit destination register, and the 24 MSBs of the register are filled with 0s.

Figure 7-5 shows the effect of combining plane masking with pixel transparency. Again, the performance of the special hardware in the local memory interface controller is demonstrated. The example shows the transfer of two pixels during the course of a PixBlt operation with transparency enabled, the pixel size set at eight bits, and the *replace* pixel processing operation. The inverse of PMASK is ANDed with the source data, and transparency detection is applied to the resulting entire pixel. In other words, the result is used to control the write in the manner described in the previous discussion of pixel transparency. Since the three LSBs of the source pixel in bits 8–15 are 0s, and the rest of the pixel is masked off, the entire source pixel is interpreted as transparent. The memory interface logic generates an internal mask to govern which bits are modified during a write cycle. This mask contains 0s in the bits corresponding to the transparent pixel.



Note: This example assumes eight bits per pixel.

Figure 7-5. Write Cycle With Transparency and Plane Masking

- Figure 7-5 *a* shows the original data at the destination location.
- *b* shows the source data.
- In *c*, the source data is ANDed with the inverse of the plane mask.
- *d* shows the intermediate result produced by *c*.
- This result is used to generate the transparency mask in *e*, which is ANDed with the inverse of the plane mask in *c* to produce the composite mask shown in *f*.
- The result in *G* is produced by replacing with the source only those bits of the destination corresponding to 1s in the composite mask in *f*.

7.7 Pixel Processing

Source and destination pixel values can be combined according to the *pixel processing* operation (or raster operation) selected. The TMS34010's pixel processing operations include 16 Boolean and 6 arithmetic operations. The Booleans are performed in bitwise fashion on operand pixels of 1, 2, 4, 8, or 16 bits. The arithmetic operations treat operand pixels of 4, 8, or 16 bits as unsigned binary numbers.

When a pixel is read from its source location, it is arithmetically combined with the corresponding destination pixel according to the Boolean or arithmetic pixel processing option selected, and the result is written to the destination pixel. The pixel processing operation is selected by the PPOP field in the CONTROL register. Table 7-1 and Table 7-2 list the 22 PPOP codes and their meanings.

Table 7-1. Boolean Pixel Processing Options

PPOP Field	Operation
00000	Source → Destination
00001	Source AND Destination → Destination
00010	Source AND ~Destination → Destination
00011	0s → Destination
00100	Source OR ~Destination → Destination
00101	Source XNOR Destination → Destination
00110	~Destination → Destination
00111	Source NOR Destination → Destination
01000	Source OR Destination → Destination
01001	Destination → Destination
01010	Source XOR Destination → Destination
01011	~Source AND Destination → Destination
01100	1s → Destination
01101	~Source OR Destination → Destination
01110	Source NAND Destination → Destination
01111	~Source → Destination

Table 7-2. Arithmetic (or Color) Pixel Processing Options

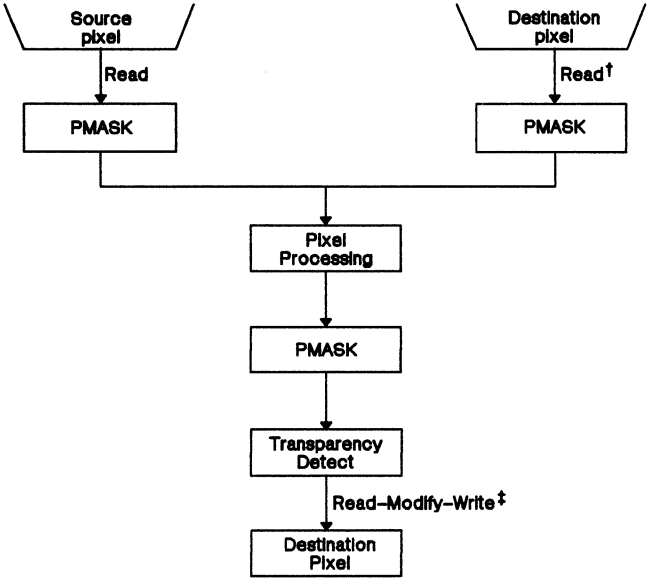
PPOP Field	Operation
10000	Source + Destination → Destination
10001	ADDS(Source, Destination) → Destination
10010	Destination - Source → Destination
10011	SUBS(Source, Destination) → Destination
10100	MAX(Source, Destination) → Destination
10101	MIN(Source, Destination) → Destination
10110–11111	Reserved

In Table 7-2, pixel processing codes 10000_2 and 10010_2 correspond to standard 2s complement addition and subtraction. A result that overflows the specified pixel size causes the pixel value to wrap around within its 4, 8, or 16-bit range. Carry bits are, however, prevented from propagating to adjacent pixels.

The ADDS (add with saturation) and SUBS (subtract with saturation) operations shown in Table 7-2 produce results identical to those of standard addition or subtraction, except when arithmetic overflow occurs. When the ADDS operation would produce an overflow result, the result is replaced with all 1s. When the SUBS operation would produce an underflow result, the result is replaced with all 0s.

The MAX operation shown in Table 7-2 compares the source and destination pixels and then writes the greater value to the destination location. The MIN operation is similar, but writes the lesser value to the destination.

Figure 7-6 depicts the interaction of pixel processing with other graphics operations when a source pixel is transferred to a destination pixel. Note that this is a general description; some of these operations do not occur if they are not selected. Pixels are first read from memory and modified by the plane mask. Pixel processing is then performed on the modified pixel values. The plane mask is applied to the result. Bits which are 1s in the PMASK produce 0 bits in the result of this process. Thus, some processed pixels may become transparent as the result of plane masking. Next, transparency detection is applied to the data, and finally, a read-modify-write operation is invoked.



† Not performed if *replace* is selected.
‡ Only performed when plane masking or transparency is active and the pixel size is not 16, or the data being written is not word-aligned.

Figure 7-6. Graphics Operations Interaction

7.8 Boolean Processing Examples

Figure 7-7 illustrates the effects of five commonly used Boolean operations when applied to one-bit pixels. Black regions contain 0s, and white regions contain 1s. Figure 7-7 *a* and *b* show the original source and destination arrays. The source operand in *a* is the letter **A**, and the destination in *b* is a calligraphic-style **X**.

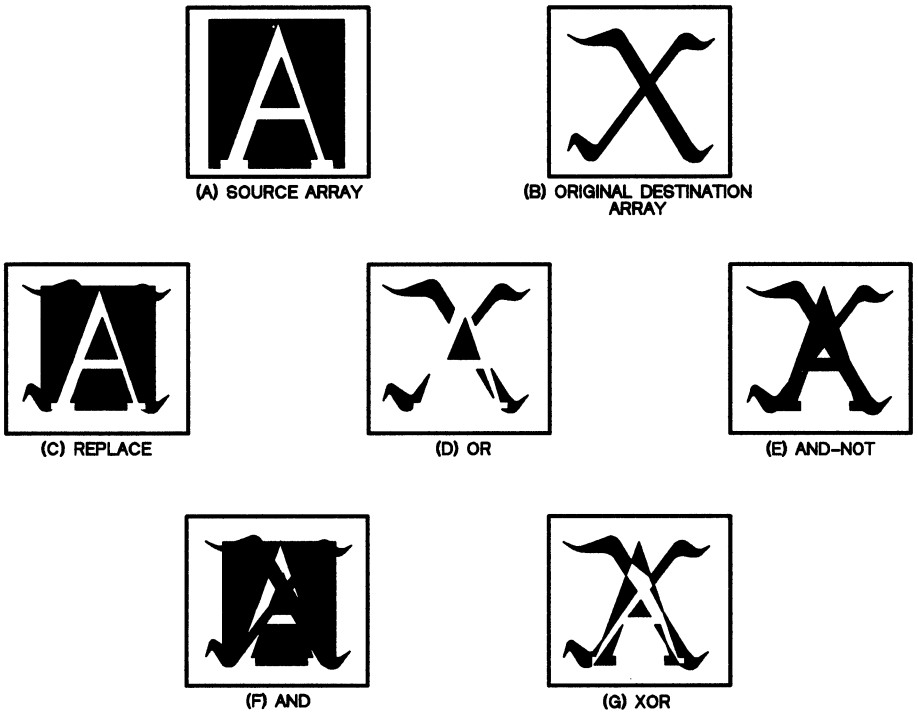


Figure 7-7. Examples of Operations on Single-Bit Pixels

7.8.1 Replace Destination with Source

A simple replacement operation overwrites the pixels of the destination array with those of the source. Figure 7-7 *c* shows the letter **A** written over the center portion of a larger **X** using the replace operation. The rectangular region around the letter **A** obscures a portion of the **X** lying outside the **A** pattern. Other operations allow only those pixels corresponding to the **A** pattern within the rectangle to be replaced, permitting the background pattern to show through. These are the logical OR and logical AND-NOT (NOT source AND destination) operations. The replace-with-transparency operation performs similarly in color systems.

7.8.2 Logical OR of Source with Destination

Figure 7-7 *d* illustrates the use of the logical OR operation during a PixBlt. For a one-bit-per-pixel display, the OR function leaves the destination pixels unaltered in locations corresponding to 0s in the source pixel array. Destination pixels in positions corresponding to 1s in the source are forced to 1s.

7.8.3 Logical AND of NOT Source with Destination

Logically ANDing the negated source with the destination is complementary to the logical OR operation. Destination pixels corresponding to 1s in the source array remain unaltered, but those corresponding to 0s in the source are forced to 0s. Figure 7-7 *e* is an example of the AND-NOT PixBlt operation (notice the negative image of the letter **A**). For comparison, Figure 7-7 *f* shows the result of simply ANDing the source and destination.

7.8.4 Exclusive OR of Source with Destination

The XOR operation is useful in making patterns stand out on a screen in instances where it is not known in advance whether the background will be 1s or 0s. At every point at which the source array contains a pixel value of 1, the corresponding pixel of the destination array is flipped – a 1 is converted to a 0, and vice versa. XOR is a reversible operation; by XORing the same source to the same destination twice, the original destination is restored. These properties make the XOR operation useful for placing and removing temporary objects such as cursors, and in "rubberbanding" lines. As seen in the example of Figure 7-7 *g*, however, the object may be difficult to see if both the source and destination arrays contain intricate shapes.

7.9 Multiple-Bit Pixel Operations

The Boolean operations described in Section 7.8 are sufficient for single-bit pixel operations, but they may be inappropriate for multiple-bit pixel operations, especially when color is involved. For example, the result of a bit-wise-OR operation on a black-and-white (one bit per pixel) display is easily predicted – ORing black and white yields white. However, the meaning of this operation is less intuitive when it is applied to multiple-bit pixels. For example, in a population-density map, colors may be used to represent numeric values. If one color, such as red, represents one level of population density, and blue represents another, what happens when the two colors are bit-wise-ORed? When pixels represent numeric values, numerical operations such as addition and subtraction yield more useful results.

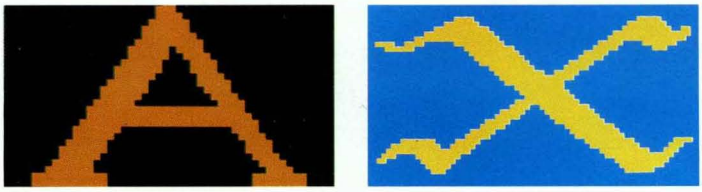
Boolean operations are usually inadequate for merging antialiased objects into a single bit-mapped image. Older graphics systems that are limited to Boolean operations on pixels are incapable of supporting many practical applications on multiple-bit-per-pixel images. For instance, where two antialiased lines cross, AND and OR operations yield chaotic pixel intensities that defeat the purpose of the antialiasing. However, merging the two lines by means of the GSP's MAX operation (for white on black) or MIN operation (for black on white) yields a smooth and aesthetically pleasing image.

7.9.1 Examples of Boolean and Arithmetic Operations

Figure 7-8 illustrates Boolean and arithmetic operations on multiple-bit pixels. Figure 7-8 *a* illustrates a source array that contains a red letter **A**; the red pixels have the value 8 (1000_2) and the black background pixels have the value 0 (0000_2). Figure 7-8 *b* shows the destination array, a yellow **X**; the yellow pixels have the value 12 (1100_2) and the pixels in the blue background pixels have the value 2 (0010_2).

Boolean operations can be applied to multiple-bit pixels by combining the corresponding bits of each pair of source and destination pixels on a bit-by-bit basis according to the specified Boolean operation. Figure 7-8 *c* through *g* show the effects of combining the source and destination arrays using the replace, OR, AND-NOT, AND, and XOR PixBlt operations. Compare these to Figure 7-7 (page 7-17).

Arithmetic operations treat 4-bit, 8-bit, and 16-bit pixels as unsigned binary numbers. An *n*-bit pixel represents a positive integer in the range 0 to 2^n-1 (all 1s). Examples of arithmetic operations on source and destination pixels are shown in Figure 7-8 *i* through *n* and discussed in Section 7.9.1.1 through Section 7.9.1.4.



(a) Source **(b) Destination**

Figure 7-8. Examples of Boolean and Arithmetic Operations

(c) Src Replaces Dst

(d) Src OR Dst

(e) $\overline{\text{Src}}$ AND Dst

(f) Src AND Dst

(g) Src XOR Dst

(h) Replace with Transparency

(i) Add

(j) Subtract

(k) Add with Saturation

(l) Subtract with Saturation

(m) MAX

(n) MIN

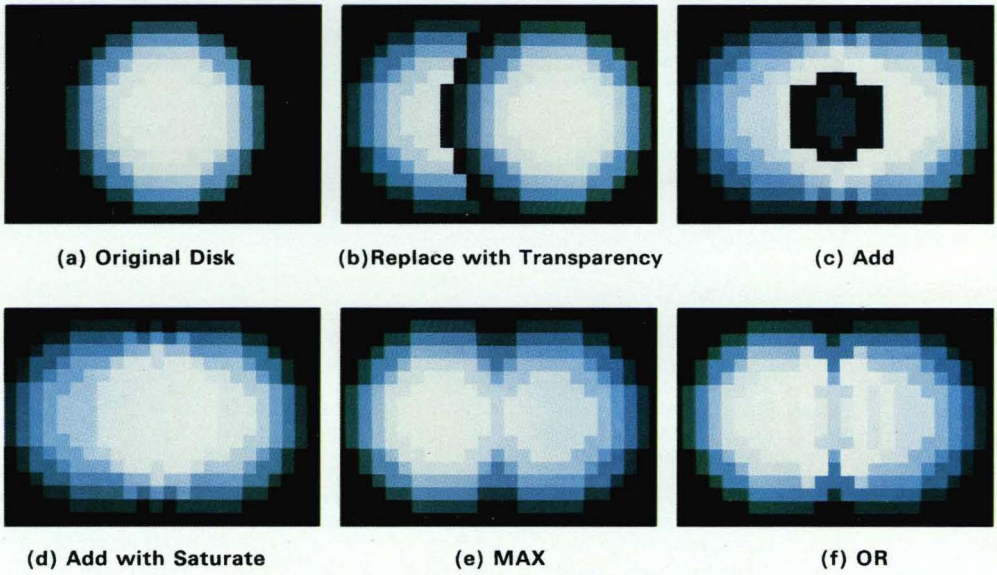
Figure 7-8. Examples of Boolean and Arithmetic Operations (Concluded)

7.9.1.4 Figure 7-8 n - Minimum

Figure 7-8 *n* illustrates the results of the MIN operation on the source and destination arrays. MIN compares two pixel values and replaces the destination pixel with the smaller value. MIN is similar to the Boolean AND function. MIN can be used with priority-encoded pixel values, similar to MAX, but the effect is reversed. In Figure 7-8 *n*, the priorities of the two objects are reversed from that of the MAX example shown in Figure 7-8 *m*. The MIN operation also has uses similar to those of MAX in smoothly combining antialiased objects that overlap.

7.9.2 Operations on Pixel Intensity

Figure 7-9 illustrates the visual effects of various PixBlt operations on two intersecting disks. In these examples, each pixel is a four-bit value representing an intensity from 0 (black) to 15 (white). Before the PixBlt operation, only a single disk resides on the screen, as shown in Figure 7-9 *a*. The intensity of the disk is greatest at the center (where the value is 12), and gradually falls off as the distance from the center increases. Figure 7-9 *b* through *f* show the effects of combining a second, identical disk with the first. Figure 7-9 *b* through *e* are produced using arithmetic operations; *f* is the result of a logical OR of the source and destination. These operations are discussed in Section 7.9.2.1 through Section 7.9.2.4.



(a) Original Disk **(b)Replace with Transparency** **(c) Add**

(d) Add with Saturate **(e) MAX** **(f) OR**

Figure 7-9. Examples of Operations on Pixel Intensity

7.9.1.4 Figure 7-8 n - Minimum

Figure 7-8 *n* illustrates the results of the MIN operation on the source and destination arrays. MIN compares two pixel values and replaces the destination pixel with the smaller value. MIN is similar to the Boolean AND function. MIN can be used with priority-encoded pixel values, similar to MAX, but the effect is reversed. In Figure 7-8 *n*, the priorities of the two objects are reversed from that of the MAX example shown in Figure 7-8 *m*. The MIN operation also has uses similar to those of MAX in smoothly combining antialiased objects that overlap.

7.9.2 Operations on Pixel Intensity

Figure 7-9 illustrates the visual effects of various PixBlt operations on two intersecting disks. In these examples, each pixel is a four-bit value representing an intensity from 0 (black) to 15 (white). Before the PixBlt operation, only a single disk resides on the screen, as shown in Figure 7-9 *a*. The intensity of the disk is greatest at the center (where the value is 12), and gradually falls off as the distance from the center increases. Figure 7-9 *b* through *f* show the effects of combining a second, identical disk with the first. Figure 7-9 *b* through *e* are produced using arithmetic operations; *f* is the result of a logical OR of the source and destination. These operations are discussed in Section 7.9.2.1 through Section 7.9.2.4.

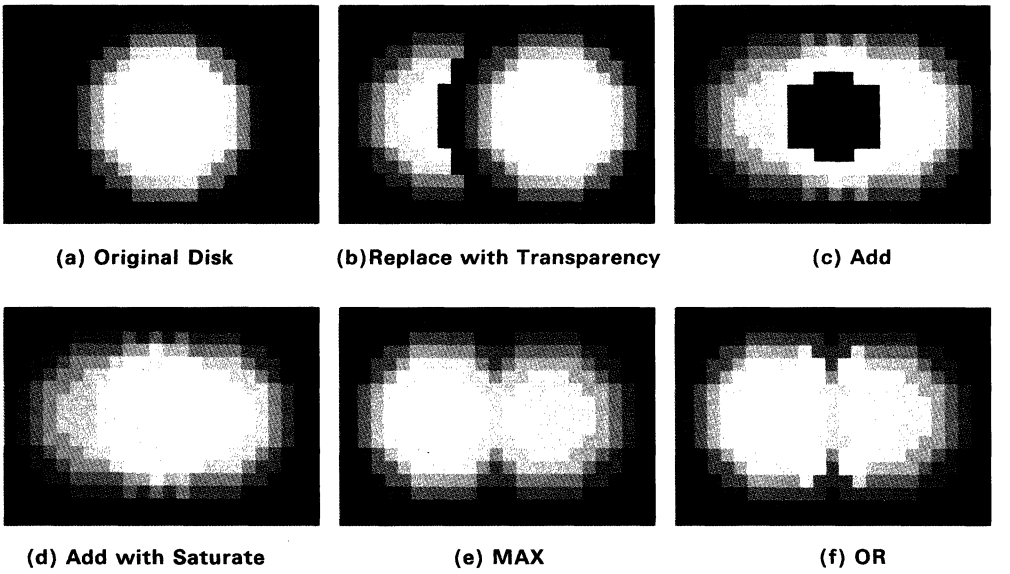


Figure 7-9. Examples of Operations on Pixel Intensity

The gradual change in intensity at the edge of the disk in Figure 7-9 *a* is similar to the result produced by certain antialiasing techniques whose purpose is to reduce jagged-edge effects. A text font might be stored in antialiased form, for example, to give the text a smoother appearance. When two characters from the font table are PixBlt'd to adjacent positions on the screen, they may overlap slightly. The particular arithmetic or Boolean operation selected for the PixBlt determines the way in which the antialiased edges of the characters are combined within regions of overlap.

7.9.2.1 Figure 7-9 b - Replace with Transparency

In Figure 7-9 *b*, a second disk is PixBlt'd into a position near the first disk. A replace-with-transparency operation is performed. Those pixels of the first disk that lie within the rectangular region containing the second disk, but are not part of the second disk, remain intact. The visual effect is that the second disk (at the right) appears to lie in front of the original disk (at the left). However, assuming that the gradual change in intensity at the perimeter of the disks is done for the purpose of antialiasing, the sharp edge that results where the second disk covers the first defeats this purpose. In other applications, this sharp edge may be desirable; for example, it might be used to make a text character or a cursor stand out from the background. The replace-with-transparency operation also supports object priority by writing objects to the screen in ascending order of priority.

7.9.2.2 Figure 7-9 c - Add with Overflow and Subtract with Underflow

In Figure 7-9 *c*, a second disk is PixBlt'd into an area overlapping the first disk, using an add-with-overflow operation. In this example, when 1 is added to an intensity of 15, the sum is truncated to four bits to produce the result 0. The effect of arithmetic overflow is visible at the intersection of the two disks as discontinuities in intensity.

This effect is useful for making objects stand out against a cluttered background. Add with overflow has an additional benefit – the object can be removed by subtracting (with underflow) the object image from the screen.

7.9.2.3 Figure 7-9 d - Add and Subtract with Saturation

In Figure 7-9 *d*, the original disk is on the left. A second disk is PixBlt'd into a region overlapping the original disk, using an add-with-saturate operation. Whenever the sum of two pixels exceeds the maximum intensity value, which is 15 for this example, the sum is replaced with 15. The bright region that occurs where the two disks intersect is produced when the corresponding pixels of the two disks are added in this manner. Subtract-with-saturate is the complementary operation; when the difference of the two pixel values is negative, the sum is replaced by the minimum intensity value, 0.

The add-with-saturate operation shown in Figure 7-9 *d* approximates the effect of two light beams striking the same surface; the surface is brightest in the area in which the two beams overlap.

These operations can be used to achieve an effect similar to that of an airbrush in painting. Consider a display system that represents each pixel as 12 bits, and dedicates four bits each to represent the intensities of the three color

components, red, green, and blue. This method permits the intensity of each component to be directly manipulated. With each pass of the simulated airbrush over the same area of the screen, the color changes gradually toward the color of the paint in the airbrush. For example, assume that the paint is yellow (a mixture of red and green). Each time a pixel is touched by the airbrush, the intensity of the red and green components is increased by 1, and the intensity of the blue component is decreased by 1. With each sweep of the airbrush, the affected area of the screen turns more yellow until the red and green components reach the maximum intensity value (and are not allowed to overflow), and the blue component reaches 0 (and is not allowed to underflow).

7.9.2.4 Figure 7-9 e - MAX and MIN Operations

In Figure 7-9 *e*, the original disk is on the left. A second disk is PixBlt'd into the rectangular region to its right using the MAX operation. In the region in which the disks overlap, each pair of corresponding pixels from the two disks is compared and the greater value is selected. This produces a relatively smooth blending of the two disks. Unlike add with saturate, the MAX function does not generate a "hot spot" where two objects intersect.

The visual effect achieved using the MAX operation is desirable in an application, for instance, in which white antialiased lines are constructed on top of each other over a black background. MAX also smooths out places in which the lines are overlapped by antialiased text. MAX is successful in maintaining two visually distinct antialiased objects, while the add-with-saturate tends to run them together.

MIN, which is complementary to MAX, can be used similarly to smooth the appearance of intersecting black antialiased lines and text on a white background.

The MAX and MIN operations are particularly useful in color applications in which the number of bits per color gun is small (eight bits or less). Other operators could also be used to smooth the transition between the two overlapping antialiased objects in Figure 7-9 *e*, but any additional accuracy attained by using a more complex smoothing function would probably be lost in truncating the result to the resolution of the integer used to represent the intensity at each point.

7.10 Window Checking

The TMS34010's hardware *window clipping* confines graphics drawing operations to a specified rectangular window in the XY address space. Other window checking modes cause an interrupt to be requested on a window *hit* or a window *miss*.

Window checking affects only pixel writes performed by the following graphics instructions:

- PIXBLT
- FILL
- LINE
- DRAV
- PIXT

Data writes by non-graphics instructions are not affected.

A *window* is a rectangular region of display memory specified in terms of the XY coordinates of the pixels in its two extreme corners (minimum X and Y, and maximum X and Y). The corner pixels are considered to lie within the window. Window checking is available only in conjunction with XY addressing; it is not available with linear addressing. Specifically, the destination pixel address must be an XY address.

One of four window checking modes is selected by the value loaded into the W field of the CONTROL register:

W=0: *Window checking disabled.* No window checking is performed.

W=1: *Window hit detection.* Request interrupt on attempt to write *inside* window.

W=2: *Window miss detection.* Request interrupt on attempt to write *outside* window.

W=3: *Window clipping.* Clip all pixel writes to window.

When window checking is enabled (modes 1, 2 or 3), an attempt to write to a pixel outside the window causes the V (overflow) bit in the status register to be set to 1; a write (or attempt to write) to a pixel inside the window sets V to 0. When window checking is turned off (mode 0), the V bit is unaffected during pixel writes.

7.10.1 W=1 Mode - Window Hit Detection

The W=1 mode detects attempts to write to pixels within the window. This form of window checking supports applications which permit objects on the screen to be *picked* by pointing to them with a cursor. In this mode, **all** pixel writes are inhibited, whether they address locations inside or outside the window. A window violation interrupt is requested on an attempt to write to a pixel inside the window.

For the PIXBLT and FILL instructions, the V (overflow) bit is set to 1 if the destination array lies completely outside the window. No interrupt request is generated (the WVP bit in the INTPEND register is not affected) in this case. However, if any pixel in the destination array lies within the window, the V bit is set to 0 and a window violation interrupt is requested (the WVP bit is set to 1). If the interrupt is enabled, the saved PC points to the instruction that follows the PIXBLT or FILL that caused the interrupt. If the interrupt is disabled, execution of the next instruction begins.

While no pixel transfers occur during the PIXBLT and FILL instructions executed in this mode, the specified destination array is clipped to lie within the window. In other words, the DADDR and DYDX registers are adjusted to be the starting address, width, and height of the reduced array that is the intersection of the two rectangles represented by the destination array and the window. This function can be adapted to determine the intersection of two arbitrary rectangles on the screen – a calculation that is often performed in windowed graphics systems.

In the case of a DRAV or PIXT instruction, an attempt to write to a pixel outside the window causes the V bit to be set to 1. No interrupt request is generated (the WVP bit is not affected). An attempt to write to a pixel inside the window causes the V bit to be set to 0, and a window violation interrupt request is generated (the WVP bit is set to 1).

At the end of a LINE instruction, the V bit is 0 if any destination pixel processed by the instruction lies within the window; otherwise, V is 1. Attempts to write to pixels outside the window do not cause interrupt requests to be generated (the WVP bit is not affected). An attempt to write to a pixel inside the window causes a window violation interrupt to be requested (the WVP bit is set to 1) and the LINE instruction aborts. If the interrupt is enabled, the PC saved during the interrupt points to the instruction that follows the LINE instruction. If the interrupt is disabled, execution of the next instruction begins.

The W=1 mode can be used to pick an object on the screen by means of the following simple algorithm. An object previously drawn on the screen is picked by moving the cursor to the object's position and selecting it. To determine which object is pointed to, the software first sets the window to a small region surrounding the position of the cursor. The software next steps a second time through the same display list used to draw the current screen until one of the objects causes a window interrupt to occur. This should be the object pointed to by the cursor. If no object causes an interrupt, the pick window can be enlarged and the process repeated until the object is found. If two objects cause interrupts, the size of the pick window can be reduced until only one object causes an interrupt.

7.10.2 W=2 Mode - Window Miss Detection

The W=2 mode permits a PIXBLT or FILL instruction to be aborted if any pixel in the destination array lies outside the window. The destination array is written only if the array lies entirely within the window, in which case the V (overflow) bit is set to 0, and no interrupt request is generated (the WVP bit is not affected). If any pixel in the destination array lies outside the window, the V bit is set to 1, and a window violation interrupt is requested (the WVP bit is set to 1).

For the DRAV and PIXT instructions, the destination pixel is drawn only if it lies within the window. In this case, the V bit is set to 0, and no interrupt request is generated (the WVP bit is not affected). If the destination location lies outside the window, the pixel write is inhibited, the V bit is set to 1, and a window violation interrupt is requested (the WVP bit is set to 1).

At the end of a LINE instruction, the V bit is 0 if the last destination pixel processed by the instruction lies within the window; otherwise, V is 1. Attempts to write to pixels inside the window do not cause interrupt requests to be generated (the WVP bit is not affected). An attempt to write to a pixel outside the window causes a window violation interrupt to be requested (the WVP bit is set to 1) and the instruction aborts. If the interrupt is enabled, the PC saved during the interrupt points to the instruction that follows the LINE instruction. If the interrupt is disabled, execution of the next instruction begins.

7.10.3 W=3 Mode - Window Clipping

In the W=3 mode, only writes to pixels within the window are permitted; writes to pixels outside the window are inhibited. No interrupt request is generated for any case.

For a PIXBLT or FILL instruction, only the portion of the destination array lying within the window is drawn. At the start of instruction execution, the specified destination array is automatically preclipped to lie within the window before the first pixel is transferred. Hence, no execution time is lost attempting to write destination pixels which lie outside the window. In the case of a PIXBLT, the source array is preclipped to fit the adjusted dimensions of the destination array before the transfer begins.

During execution of a DRAV or PIXT instruction, a write to a pixel inside the window is permitted, and the V bit is set to 0. An attempted write to a pixel outside the window is inhibited, and the V bit is set to 1.

For the LINE instruction, writes to pixels outside the window are inhibited at drawing time; no preclipping is performed. The value of the V bit at the end of a LINE instruction is determined by whether the last pixel calculated by the instruction fell inside (V=0) or outside (V=1) the window.

7.10.4 Specifying Window Limits

The limits of the current window are specified in the WSTART (window start) and WEND (window end) registers. WSTART specifies the minimum XY coordinates in the window, and WEND specifies the maximum XY coordinates.

As Figure 7-10 shows, WSTART specifies the XY coordinates (X_{start},Y_{start}) at the upper left corner of the window, and WEND specified the XY coordinates (X_{end},Y_{end}) at the bottom right corner of the window. The origin is located in its default position in the top left corner of the screen.

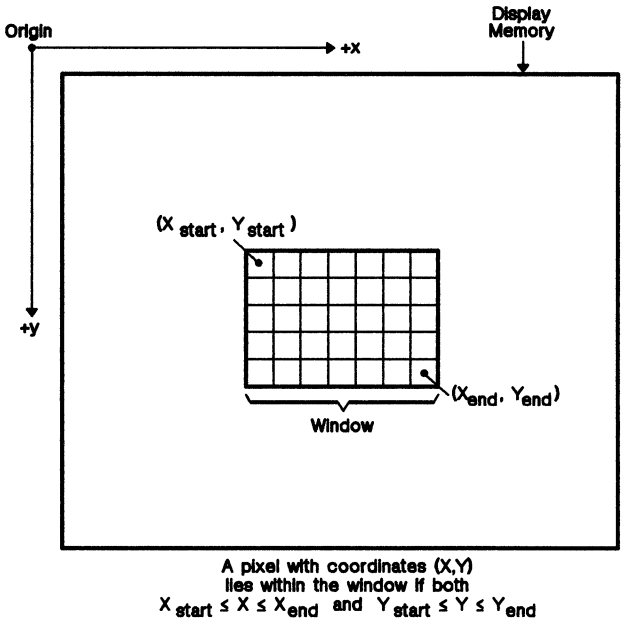


Figure 7-10. Specifying Window Limits

Figure 7-10 shows that a pixel that has coordinates (X,Y) lies within the window if $X_{start} \leq X \leq X_{end}$ **and** $Y_{start} \leq Y \leq Y_{end}$. If a pixel does not meet these conditions, it lies outside the window.

When $X_{start} > X_{end}$ **or** $Y_{start} > Y_{end}$, the window is empty; that is, it contains no pixels. Under these conditions, the window checking hardware detects all destination pixel addresses as lying outside the window. Note that the conditions $X_{start} = X_{end}$ **and** $Y_{start} = Y_{end}$ together specify a window containing a single pixel.

Window start and end coordinates must lie in the range (0,0) to (+32767,+32767). A window cannot contain pixels with negative X or Y coordinates.

7.10.5 Window Violation Interrupt

A window violation (WV) interrupt is requested (the WVP bit in the INTPEND register is set to 1) when:

- W=1 and an attempt is made to write to a pixel **inside** the window **or**
- W=2 and an attempt is made to write to a pixel **outside** the window

The interrupt occurs if it is enabled by the following conditions:

- The WVE bit in the INTENB register is 1
- The IE bit in the status register is 1

Alternatively, if the WV interrupt is disabled (IE=0 or WVE=0), the window violation can be detected by testing the value of either the V bit in the status register or the WVP bit following the operation.

When a WV interrupt occurs, the registers that change during the LINE, PIXBLT and FILL instructions contain their intermediate values at the time the violation was detected.

7.10.6 Line Clipping

The TMS34010 supports two methods for clipping straight lines to the boundaries of a rectangular window: postclipping and preclipping. Postclipping means that just before each pixel on the line is drawn, it is compared with the window limits. If it lies outside the window, the write is inhibited. In contrast, preclipping involves determining in advance of any drawing operations which pixels in the line lie within the window. The algorithm draws only these pixels, and makes no attempt to write to pixels outside the window. A preclipped line may take less time to draw since no calculations are performed for pixels lying outside the window. In contrast, postclipping spends the same amount of time calculating the position of a pixel outside the window as it does calculating a pixel inside the window.

When postclipping is used, special window comparison hardware compares the coordinates of the pixel being drawn against all four sides of the window at once. The W=3 window-checking mode is selected, and window checking is performed in parallel with execution of the LINE instruction, so no overhead is added to the time to draw a pixel. However, unless this form of clipping is used carefully, another type of overhead may become significant. For example, in a CAD (computer-aided design) environment where only a small portion of a system diagram is to be displayed at once, potentially a great deal of time could be spent performing calculations for points (or entire lines) lying off-screen.

Preclipping is generally faster than postclipping, depending on how likely a line is to lie outside the window. The first step in preclipping a series of lines is to identify those that lie either entirely inside or outside the window. This is accomplished by using an "outcode" technique similar to that of the Cohen-Sutherland algorithm. Those lines lying entirely outside are "trivially rejected" and consume no more processing time. Those lines lying entirely within are drawn from one endpoint to the other with no clipping required. This still leaves a third category of lines that may cross a window boundary, and these require intersection calculations. However, this technique is pow-

erful for reducing the number of lines that require such calculations. While the calculation of outcodes could be performed in software, this would represent significant overhead for each line considered. The TMS34010 provides a more efficient implementation via its CPW (compare point to window) instruction, which compares a point to all four sides of the window at once.

The outcode technique classifies a line according to where its endpoints fall in relation to the current clipping window. The area surrounding the window is partitioned into eight regions, as indicated in Figure 7-11. Each region is assigned a 4-bit code called an *outcode*. The outcode within the window is 0000_2. When an endpoint of a line falls within a particular region, it is assigned the outcode for that region. If the two endpoints of a line both have outcodes 0000_2, the line lies entirely within the window. If the bitwise AND of the outcodes of the two endpoints yields a value other than 0000_2, the line lies entirely outside the window. Lines that fall into neither of these categories may or may not be partially visible within the window.

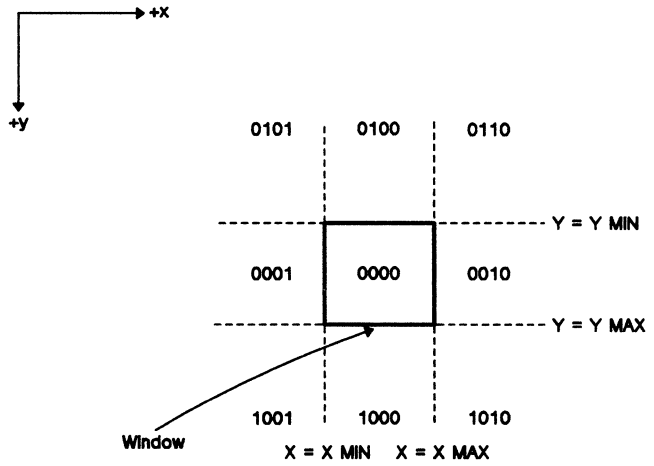


Figure 7-11. Outcodes for Line Endpoints

For those lines that require intersection calculations after the outcodes have been determined, midpoint subdivision is an efficient means of preclipping. The object again is to ensure that drawing calculations are performed only for pixels lying within the window. An example of the midpoint subdivision technique is illustrated in Figure 7-12. The line *AB* lies partially within the window. The first step is to determine the coordinates of the line's midpoint at *C*. These are calculated as follows:

$$(X_C, Y_C) = \left(\frac{X_A + X_B}{2}, \frac{Y_A + Y_B}{2}\right)$$

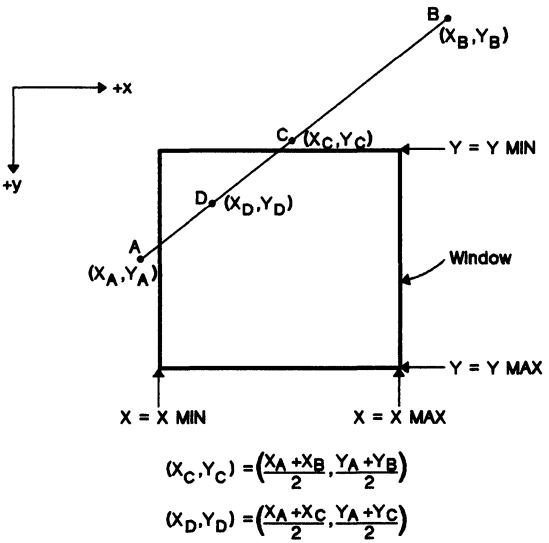


Figure 7-12. Midpoint Subdivision Method

Comparing the outcodes of *B* and *C*, segment *BC* lies entirely outside the window and can be trivially rejected. Segment *AC* still lies partially within the window and will be subdivided again. The coordinates of point *D*, the midpoint of *AC*, are calculated as before. Point *D* is determined to lie within the window. The LINE instruction is now invoked two times, for segments *DC* and *DA*, with *D* selected as the starting point in each case. For each segment the W=2 window-checking mode is selected, but the window violation interrupt is disabled. When each line crosses the window boundary, the window-checking hardware detects this and the LINE instruction aborts. In this way the LINE instruction performs drawing calculations only for portions of *DA* and *DC* lying within the window.

Section 8

Interrupts, Traps, and Reset

The TMS34010 supports eight interrupts, including reset. Memory addresses FFFFFC00h to FFFFFFFFh contain the 32 vector addresses used during interrupts, software traps and reset. Each vector is a 32-bit address that points to the beginning of the appropriate interrupt service routine.

This section includes the following topics:

8.1 Interrupt Priorities and Vector Addresses

Table 8-1 and Figure 8-1 summarize the TMS34010 interrupt vector addresses and the interrupt priorities. $\overline{\text{RESET}}$ has the highest priority, and the illegal opcode interrupt has the lowest. If two interrupts are requested at the same time, the highest priority interrupt is serviced first (assuming it is enabled). $\overline{\text{RESET}}$ and the nonmaskable interrupt cannot be disabled.

Table 8-1. Interrupt Priorities

Int.	Priority	Internal/ External	Description and Source
Reset	1	I	Device reset. Taken when the input signal at the $\overline{\text{RESET}}$ pin is asserted low.
NMI	2	I	Nonmaskable interrupt. Generated by a host processor.
HI	3	I	Host interrupt. Generated by a host processor.
DI	4	I	Display interrupt. Generated by the TMS34010.
WV	5	I	Window violation interrupt. Generated by the TMS34010.
INT1	6	E	External interrupts 1 and 2. Generated by external devices.
INT2	7	E	
ILLOP	8	I	Illegal opcode interrupt. Generated by the TMS34010 when an illegal opcode is encountered.

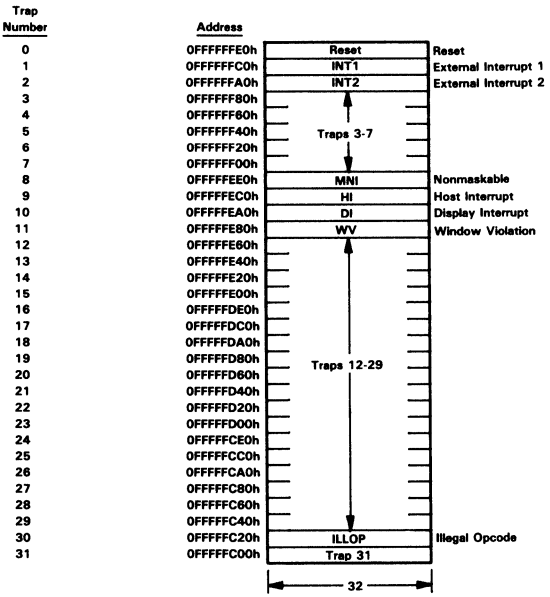


Figure 8-1. Vector Address Map

8.2 Interrupt Interface Registers

Two registers, a subset of the I/O registers discussed in Section 6, monitor and mask interrupt requests. These registers are summarized below; for more information, please refer to the register descriptions in Section 6.

The interrupt enable register, **INTENB**, contains the interrupt mask that selectively enables various interrupts. An interrupt is enabled when the status IE (global interrupt enable) bit and the appropriate bit in the INTENB register are *both* set to 1.

- *X1E* (bit 1) enables external interrupt 1.
- *X2E* (bit 2) enables external interrupt 2.
- *HIE* (bit 9) enables the host interrupt.
- *DIE* (bit 10) enables the display interrupt.
- *WVE* (bit 11) enables the window violation interrupt.

The interrupt pending register, **INTPEND**, indicates which interrupts are currently pending. When an interrupt is requested, the appropriate bit in the INTPEND register is set.

- *X1P* (bit 1) indicates that external interrupt 1 is pending.
- *X2P* (bit 2) indicates that external interrupt 2 is pending.
- *HIP* (bit 9) indicates that the host interrupt is pending.
- *DIP* (bit 10) indicates that the display interrupt is pending.
- *WVP* (bit 11) indicates that the window violation interrupt is pending.

8.3 External Interrupts

External interrupt requests are received through input pins $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$. The two request pins are level-sensitive, active-low inputs. Each pin is dedicated to an individual interrupt, allowing two independent interrupt requests to be generated. (The pins are not encoded.) The state of the $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$ inputs is reflected in the X1P and X2P bits in the INTPEND register. The register bit is 1 if the corresponding request is active.

The interrupts generated by requests at the $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$ inputs are referred to as INT1 and INT2. Interrupts INT1 and INT2 are selectively enabled by means of the X1E and X2E bits in the INTENB register. If external interrupt requests become active at $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$ at the same time, and both interrupts are enabled, INT1 will be serviced first. If one or both of these interrupts is disabled, the state of the $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$ inputs continues to be reflected in the X1P and X2P bits. These bits may be polled by software to detect transitions at the interrupt inputs.

Table 8-2 shows the interrupt trap vectors for INT1 and INT2.

Table 8-2. External Interrupt Vectors

Name	Input Pin	Vector Address
INT1	$\overline{\text{LINT1}}$	FFFFFFC0h
INT2	$\overline{\text{LINT2}}$	FFFFFFA0h

Once an interrupt request has been initiated by driving an interrupt request pin low, the input should continue to be driven low until the interrupt service routine can respond to the interrupting device. If the interrupt pin is permitted to go inactive high before it has been recognized by the interrupt service routine, the request may be missed. If the active level is maintained after returning from the interrupt service routine, however, the interrupt will be taken once again.

The RETI instruction restores the ST (status) and PC (program counter) registers to their original state just prior to the interrupt. (This would not be the case, however, if for some reason the values for these registers, saved on the stack, were altered by the interrupt service routine). Assuming that the IE bit in the restored ST is a 1, interrupts are again enabled by the time the RETI instruction finishes executing. If an interrupt request is active during the last state of the RETI instruction, and the interrupt is enabled in the INTENB register, the interrupt will be taken immediately following the RETI.

The interrupt service routine typically writes to the interrupting device to clear the interrupt request before executing an RETI (return from interrupt) instruction. An example of the last three instructions in a typical interrupt service routine is shown below, where DEVICE is the symbolic address of the interrupting device:

```
CLR     A0
MOVE    A0,@DEVICE
RETI
```

The interrupt request is cleared by the MOVE instruction above, which writes a 0 to the device address. The maximum asynchronous delay from the end of the write cycle (measured from the low-to-high transition of $\overline{\text{W}}$) to the resulting low-to-high transition at the GSP's interrupt request input should be no more than six local clock periods.

Signals input to the local interrupt pins are assumed to be asynchronous to the GSP local clocks, and are synchronized internally by the GSP before they are processed. The GSP samples the state of the $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$ inputs at each high-to-low transition of LCLK1, and updates the X1P and X2P bits in the INTPEND register accordingly (an active-low input is seen as a one in the appropriate register bit). The delay from the transition at the input to the corresponding change in the X1P or X2P bit is from one to two states, depending on the transition's phase relationship to LCLK1.

8.4 Internal Interrupts

Several internal conditions are associated with specific interrupts. Table 8-3 summarizes these interrupts. If two internal interrupts are requested simultaneously, or if two or more internal interrupt requests are pending, the highest priority interrupt is serviced first; NMI has the highest priority, followed by HI, DI, and WV. When internal *and* external interrupts are pending, the internal interrupts are serviced first (with the exception of the ILLOP interrupt).

Table 8-3. Interrupts Associated with Internal Events

Name	Function	Level	Vector Location	Description
NMI	Nonmaskable interrupt	8	FFFFFEE0h	The host processor sets the NMI bit in the HSTCTL register to a 1.
HI	Host interrupt	9	FFFFFEC0h	The host processor sets the INTIN bit in the HSTCTL register to a 1.
DI	Display interrupt	10	FFFFFEA0h	A particular horizontal line on the video display is being refreshed. The line number is specified in the DPYINT register.
WV	Window violation interrupt	11	FFFFFE80h	An attempt has been made to move a pixel to a destination location that lies inside or outside a specified window, depending on the selected windowing mode.
ILLOP	Illegal operand interrupt	30	FFFFFC20h	See Section 8.7.

The nonmaskable interrupt, or NMI, occurs when a host processor requests an interrupt by writing a 1 to the NMI bit in the HSTCTL register. This interrupt cannot be disabled, and always occurs as soon as possible following the request. The NMI is delayed only for completion of an instruction already in progress, or until the next interruptible point of an interruptible instruction such as a PIXBLT is reached.

The NMI mode bit in the HSTCTL register determines whether or not context information is saved on the stack when a nonmaskable interrupt occurs:

- If NMIM = 0, the PC and ST are pushed on the stack before the interrupt is serviced.
- If NMIM = 1, nothing is saved on the stack before the interrupt is serviced.

The TMS34010 automatically clears the NMI bit at the time it takes the interrupt. After setting the NMI bit, the host processor can determine when the TMS34010 has taken the interrupt by polling the NMI bit until it changes from a 1 to a 0.

The display interrupt (DI) is used to coordinate processing activity with the refreshing of particular areas of the display. The display interrupt request becomes active when a particular display line, specified in the DPYINT register, is output to the monitor screen. At the start of each horizontal blanking period, the VCOUNT register is compared to the DPYINT register. When the vertical count value in VCOUNT = DPYINT, a display interrupt request is generated. If enabled, the interrupt is taken.

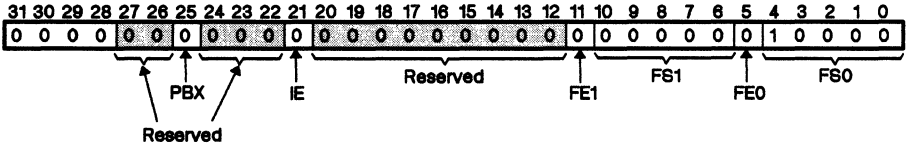
8.5 Interrupt Processing

An interrupt is said to be *pending* if it has been requested but has not yet been processed. If a pending interrupt is enabled, and no interrupt of higher priority is pending at the same time, the interrupt is accepted by the TMS34010 at the end of the current instruction (or at the next interruptible point in the middle of a PIXBLT or FILL instruction). When the TMS34010 takes an interrupt, it performs the following actions:

1) The TMS34010 pushes the PC on the stack.

2) The TMS34010 pushes the ST on the stack. PIXBLT and FILL instructions that are interrupted by external, host, and nonmaskable (if NMIM=0) interrupts set the PBX bit in the ST before pushing the ST.

3) The TMS34010 modifies the contents of the ST as follows:



4) The TMS34010 fetches the interrupt vector from external memory into the PC.

5) The TMS34010 begins executing the instruction pointed to by the new PC value.

In step 5, the TMS34010 resumes instruction execution at the entry point of the interrupt service routine. At the time the first instruction of the service routine begins execution, the new status register contents imply the following conditions:

- All interrupts are disabled (except NMI and reset)
- Field 0 is 16 bits long and is zero extended
- Field 1 is 32 bits long and is zero extended

The service routine can allow itself to be interrupted by loading a new interrupt-enable mask into the INTENB register and setting status bit IE to 1. The INTENB mask value is selected to determine which interrupts can interrupt the currently executing service routine. The service routine can also load new field sizes if values other than the defaults are required.

The last instruction in any interrupt service routine must be RETI (return from interrupt). Unlike the RETS (return from subroutine) instruction, which only pops the PC from the stack, RETI pops both the ST and PC. This restores the original state of the interrupted program so that execution can proceed from the point at which the interrupt occurred.

8.5.1 Interrupt Latency

An external interrupt, host interrupt request, or NMI request is delayed by an amount of time that depends on the instruction in progress and on the local memory bus traffic at the time of the request.

The delay from an interrupt request to the time that the first instruction of the interrupt service routine begins execution is the sum of six potential sources of delay:

1) Interrupt request recognition
2) Screen-refresh cycle
3) DRAM-refresh cycle
4) Host-indirect cycle
5) Instruction interrupt
6) Interrupt context switch

In the best case, items 2 through 5 cause no delay. The minimum delay due to items 1 and 6 is 17 machine states.

- The **interrupt request recognition** delay is the time required for a request to be internally synchronized to the local clock. In the case of an external interrupt request, the delay is measured from the high-to-low transition of the $\overline{\text{INT1}}$ or $\overline{\text{INT2}}$ pin. In the case of a host interrupt or NMI request, the delay is measured from completion of the host's write to the INTIN or NMI pin.

- The **screen-refresh** and **DRAM-refresh cycles** are a potential source of delay, but in fact occur rarely and are unlikely to delay an interrupt.

- The likelihood of a delay caused by a **host-indirect cycle** is small in most instances, but this depends on the application. The delay due to a single host-indirect cycle is two machine states, assuming no wait states, but multiple host-indirect cycles occurring within a brief period of time could cause additional delays. Theoretically, a fast host processor could generate so many local memory cycles that the TMS34010 would be prevented from servicing interrupts for an indefinite period.

- The **instruction interrupt** time refers to the time required for an instruction that was already executing at the time the interrupt request was received to either complete or to reach the next interruptible point in an instruction (such as a PIXBLT, FILL, or LINE).

- The **interrupt context switch** operation pushes the PC and ST onto the stack, and fetches the PC for the interrupt service routine from the appropriate vector in memory.

Table 8-4 shows the minimum and maximum times for each of the six operations listed. The interrupt latency is calculated as the sum of the numbers in the six rows. In the best case, the interrupt latency is only 17 machine states. The worst-case latency can be as high as 22 machine states plus the delays due to host-indirect cycles and instruction completion. Table 8-5 shows instruction interrupt times for some of the longer, noninterruptible instructions. Table 8-5 also shows the instruction completion time for a JRUC instruction

that jumps to itself - the TMS34010 may be executing this instruction if the software is simply waiting for an interrupt.

Table 8-4. Six Sources of Interrupt Delay

Operation	Latency (In States)	
	Min	Max
Interrupt recognition	1	2
Instruction interrupt	0	See Table 8-5
DRAM-refresh cycle	0	2 See Note 2
Screen-refresh cycle	0	2 See Note 2
Host-indirect cycle	0	See Note 1
Interrupt context switch	16	16

Notes: 1) The latency due to host-indirect cycles depends on both the hardware system and the application. Theoretically, a host processor could generate so many local memory cycles that the TMS34010 could effectively be prevented from servicing interrupts. The delay due to a single host-indirect cycle is two machine states, assuming no wait states.
2) DRAM-refresh and screen-refresh cycle times assume no wait states.
3) Context switch time assumes that the SP is aligned to a word boundary; that is, the four LSBs of the SP are 0s. If the SP is not aligned, the delay is 28 states.

Table 8-5. Sample Instruction Completion Times

Instruction	Worst-Case Instruction Interrupt Time (In States)	
	SP Aligned	SP Not Aligned
DIVS A0,A2	43	43
MMFM SP,ALL	72	144
MMTM SP,ALL	73	169
Wait: JRUC wait	1	1

Notes: 1) The worst-case instruction interrupt time is equal to the instruction execution time less one machine state (except for PIXBLTs, FILLs, and LINE).
2) The SP-aligned case assumes that the SP is aligned to a word boundary in memory.

8.6 Traps

The TMS34010 supports 32 software traps, numbered 0 through 31. Software traps behave similarly to interrupts, except that they are initiated when the TMS34010 executes a TRAP instruction. Unlike an interrupt, a software trap cannot be disabled.

When the TMS34010 executes a TRAP instruction, it performs the same sequence of actions that it performs for interrupts. The TRAP 1 through TRAP 31 instructions cause the status register and the PC to be pushed onto the stack. TRAP 0 is similar to a hardware reset because it does not push the status register or PC onto the stack; it differs from a hardware reset because it does not cause the TMS34010's internal registers to be set to a known initial state. TRAP 8 is similar to an NMI interrupt, except that the NMIM (NMI mode) bit in the HSTCTLL register has no effect on instruction execution; the status register and PC are stacked unconditionally when TRAP 8 is executed.

A 32-bit vector address is associated with each software trap. To determine the vector address for a trap number *N*, where *N* = 0 through 31, subtract 32*N* from FFFFFFE0h. Figure 8-1 on page 8-2 shows the vector addresses for the software traps.

8.7 Illegal Opcode Interrupts

The TMS34010 recognizes several reserved opcodes as illegal. When one of these opcodes is encountered in the instruction stream, the TMS34010 traps to vector number 30, located at memory address FFFFFC20h. An illegal opcode is similar in effect to a TRAP 30 instruction. The illegal opcode interrupt cannot be disabled. Table 8-6 lists ranges of illegal opcodes.

Table 8-6. Illegal Opcodes Ranges

0200h through 02FFh
0400h through 04FFh
0800h through 08FFh
0A00h through 0AFFh
0C00h through 0CFFh
0E00h through 0EFFh
3400h through 37FFh
7000h through 7FFFh
9E00h through 9FFFh
BE00h through BFFFh
D800h through DEFFh
FE00h through FFFFh

8.8 Reset

Reset puts the TMS34010 into a known initial state that is entered when the input signal at the $\overline{\text{RESET}}$ pin is asserted low. $\overline{\text{RESET}}$ must remain active low for a minimum of 40 local clock (LCLK1 or LCLK2) periods to ensure that the TMS34010 has sufficient time to establish its initial internal state. While the reset signal remains asserted, all outputs are in a known state, no DRAM-refresh cycles take place, and no screen-refresh cycles are performed.

At the low-to-high transition of the $\overline{\text{RESET}}$ signal, the state of the $\overline{\text{HCS}}$ input determines whether the TMS34010 is halted (*host-present mode*) or whether it begins executing instructions (*self-bootstrap mode*):

- **Host-Present Mode**

  If $\overline{\text{HCS}}$ is high at the end of reset, TMS34010 instruction execution is halted and remains halted until the host clears the HLT (halt) bit in HSTCTL (host control register). Following reset, the eight $\overline{\text{RAS}}$-only refresh cycles required to initialize the dynamic RAMs are performed automatically by the TMS34010 memory control logic. As soon as the eight $\overline{\text{RAS}}$-only cycles are completed, the host is allowed access to TMS34010 memory. At this time, the TMS34010 begins to automatically perform DRAM refresh cycles at regular intervals. The TMS34010 remains halted until the host clears the HLT bit. Only then does the TMS34010 fetch the level-0 vector address from location FFFFFFE0h and begin executing its reset service routine.

- **Self-Bootstrap Mode**

  If $\overline{\text{HCS}}$ is low at the end of reset, the TMS34010 first performs the eight $\overline{\text{RAS}}$-only refresh cycles required to initialize the DRAMs. Immediately following the eight $\overline{\text{RAS}}$-only cycles, the TMS34010 fetches the level-0 vector address from location FFFFFFE0h, and begins executing its reset service routine.

Unlike other interrupts and software traps, reset does not save previous ST or PC values. This is because the value of the stack pointer just before a reset is generally not valid, and saving its value on the stack is unnecessary. A TRAP 0 instruction, which uses the same vector address as reset, similarly does not save the ST or PC values.

8.8.1 Asserting Reset

A reset is initiated by asserting the $\overline{\text{RESET}}$ input pin at its active-low level. To reset the TMS34010 at power up, $\overline{\text{RESET}}$ must remain active low for a minimum of 40 local clock periods after power levels have become stable. At times other than power up, the TMS34010 is also reset by holding $\overline{\text{RESET}}$ low for a minimum of 40 clock periods. The 40-clock interval is required to bring TMS34010 internal circuitry to a known initial state. While $\overline{\text{RESET}}$ remains asserted, the output and bidirectional signals are driven to a known state.

The TMS34010 drives its $\overline{\text{RAS}}$ signal inactive high as long as $\overline{\text{RESET}}$ remains low. The specifications for certain DRAM and VRAM devices, including the TMS4161, TMS4164 and TMS4464 devices, require that the $\overline{\text{RAS}}$ signal be driven inactive-high for 100 microseconds during system reset. Holding the $\overline{\text{RESET}}$ signal low for 150 microseconds causes the $\overline{\text{RAS}}$ signal to remain high

for the 100 microseconds required to bring the memory devices to their initial states. DRAMs such as the TMS4256 specify an initial $\overline{RAS}$ high time of 200 microseconds, requiring that $\overline{RESET}$ be held low for 250 microseconds. In general, holding $\overline{RESET}$ low for *t* microseconds ensures that $\overline{RAS}$ remains high initially for *t* - 50 microseconds.

8.8.2 Suspension of DRAM-Refresh Cycles During Reset

An active-low level at the $\overline{RESET}$ pin is considered to be a power-up condition, and DRAM refresh is not performed until $\overline{RESET}$ goes inactive high. Consequently, the previous contents of the local memory may not be valid after a reset.

8.8.3 State of VCLK During Reset

In many systems, the VCLK pin continues to be clocked during reset. However, a system in which VCLK is not clocked during reset should maintain VCLK at the logic high level while it is not being clocked. This is necessary to ensure that the video counters are reset properly. In fact, VCLK should be held at the logic high level when it is not being clocked regardless of whether the device is being reset. While VCLK is low, storage nodes in the VCOUNT and HCOUNT registers rely on their internal capacitance to maintain their state. If VCLK remains low for a sufficiently long period, these registers are subject to bit errors due to charge leakage.

8.8.4 Initial State Following Reset

While the $\overline{RESET}$ pin is asserted low, the TMS34010's output and bidirectional pins are forced to the states listed in Table 8-7.

Table 8-7. State of Pins During a Reset

Outputs Driven To High level	Outputs Driven To Low Level	Bidirectional Pins Driven to High Impedance
DDOUT HRDY † $\overline{DEN}$ $\overline{LAL}$ $\overline{TR/QE}$ $\overline{RAS}$ $\overline{CAS}$ $\overline{W}$ $\overline{HINT}$ $\overline{HLDA/EMUA}$	$\overline{BLANK}$	$\overline{HSYNC}$ $\overline{VSYNC}$ HD0–HD15 LAD0–LAD15

† HRDY will stay high during reset if the $\overline{HCS}$ input is also high.

Immediately following reset, all I/O registers are cleared (set to 0000h), with the possible exception of the HLT bit in the HSTCTL register. The HLT bit is set to 1 if $\overline{HCS}$ is high just before the low-to-high transition of $\overline{RESET}$.

Just before execution of the first instruction in the reset routine, the TMS34010's internal registers are in the following state:

- General-purpose register files A and B are uninitialized.
- The ST is set to 00000010h.
- The PC contains the 32-bit vector fetched from memory address FFFFFFE0h.

The instruction cache is in the following state at this time:

- The SSA (segment start address) registers are uninitialized.
- The LRU (least recently used) stack is set to the initial sequence 0,1,2,3, where 0 occupies the most-recently-used position, and 3 occupies the least-recently-used position.
- All P (present) flags are cleared to 0s.

8.8.5 Activity Following Reset

Immediately following the low-to-high transition of $\overline{\text{RESET}}$, the TMS34010 performs a series of eight $\overline{\text{RAS}}$-only memory cycles to bring the DRAMs and VRAMs to their initial operating states. These cycles are completed before any accesses of the TMS34010's memory (by either the TMS34010 or host processor) are allowed to occur. If the host processor attempts to access the TMS34010 memory indirectly before the eight $\overline{\text{RAS}}$-only cycles have completed, it receives a not-ready signal from the TMS34010 until the cycles have completed. The eight $\overline{\text{RAS}}$-only cycles occur regardless of the initial value to which the HLT bit in the HSTCTL register is set.

Each of the eight $\overline{\text{RAS}}$-only cycles is a standard DRAM-refresh cycle. The $\overline{\text{RF}}$ bus status signal output with the row address is active low. The row address is all 0s.

Following the eight $\overline{\text{RAS}}$-only cycles, the TMS34010 automatically begins to initiate a new DRAM-refresh cycle every 32 TMS34010 local clock cycles. The first DRAM refresh cycle begins approximately 32 local clock periods after the end of reset. A DRAM-refresh cycle is initiated every 32 TMS34010 clock cycles until the DRAM-refresh rate is changed by the TMS34010 or host processor.

The TMS34010 is configured by means of an external signal input on the $\overline{\text{HCS}}$ pin to either:

- Begin executing instructions immediately after reset is completed (self-bootstrap mode), **or**
- Halt until the host processor instructs it to begin executing (host-present mode).

8.8.5.1 Self-Bootstrap Mode

In self-bootstrap mode, the TMS34010 begins executing instructions immediately following reset. This mode is typically used in a system in which the reset vector and reset service routine are contained in nonvolatile memory, such as a bootstrap ROM. This type of system does not necessarily require a host processor, and the TMS34010 may be responsible for performing host processor functions for the system.

The TMS34010 is configured in self-bootstrap mode when the $\overline{\text{HCS}}$ pin is low just before the low-to-high transition of $\overline{\text{RESET}}$. The low $\overline{\text{HCS}}$ level forces the HLT bit to 0. Immediately following the end of reset and the eight $\overline{\text{RAS}}$-only cycles, the TMS34010 fetches the level-0 vector address and begins executing the reset interrupt routine.

At the low-to-high transition of $\overline{\text{RESET}}$, the $\overline{\text{HCS}}$ input is internally delayed before being checked to determine how to set the HLT bit. In a system without a host processor, for instance, this permits the $\overline{\text{HCS}}$ and $\overline{\text{RESET}}$ pins to be tied together, eliminating the need for additional external logic.

Transitions of the $\overline{\text{HCS}}$ and $\overline{\text{RESET}}$ signals are assumed to be asynchronous with respect to the TMS34010 local clock. $\overline{\text{HCS}}$ and $\overline{\text{RESET}}$ are internally synchronized to the local clock by being held in latches for at least one clock period before being used by the TMS34010. The delay through the synchronizer latch is from one to two local clock periods, depending on the phase of the signal transitions relative to the clock. To permit the $\overline{\text{HCS}}$ and $\overline{\text{RESET}}$ pins to be wired together, TMS34010 on-chip logic delays the $\overline{\text{HCS}}$ low-to-high transition to ensure that it is detected **after** the $\overline{\text{RESET}}$ low-to-high transition. The level of the delayed $\overline{\text{HCS}}$ signal at the time the low-to-high $\overline{\text{RESET}}$ transition is detected determines the setting of the HLT bit.

8.8.5.2 Host-Present Mode

Host-present mode assumes that a host processor is connected to the TMS34010's host interface pins. In this mode, the TMS34010 local memory can be composed entirely of RAM (no ROM). Following reset, the host processor must download the initial program code, interrupt vectors, and so on, before allowing the TMS34010 to begin executing instructions.

The TMS34010 is configured in host-present mode as follows. On the trailing edge of $\overline{\text{RESET}}$, the $\overline{\text{HCS}}$ (host interface chip select) input is sampled. If the $\overline{\text{HCS}}$ pin is inactive high, internal logic forces the HLT (halt) bit to a 1. In this fashion, the TMS34010 is automatically halted following reset, and does not begin execution of its reset service routine until the host processor loads a 0 to HLT. In the meantime, the host processor is able to load the memory and I/O registers with the appropriate initial values before the TMS34010 begins executing instructions. This may include writing the reset vector and reset service routine into the TMS34010's memory, for example.

No additional external logic is required to force $\overline{\text{HCS}}$ high before the low-to-high transition of $\overline{\text{RESET}}$. The simple external decode logic typically used drives the $\overline{\text{HCS}}$ input active low only when one of the TMS34010's host interface registers is addressed by the host processor. Assuming that the host processor is not actively chip-selecting the TMS34010 at the end of reset, $\overline{\text{HCS}}$ is high.

Section 9

Screen Refresh and Video Timing

The TMS34010 generates the synchronization and blanking signals used to drive a video screen in a graphics system. The GSP can be programmed to support a variety of screen resolutions and interlaced or noninterlaced video. If desired, the GSP can be programmed to synchronize to externally generated video signals. The GSP also supports the use of video RAMs by generating the memory-to-register cycles necessary to refresh a screen.

This section includes the following topics:

9.1 Screen Sizes

The TMS34010's 26-bit word address provides direct addressing of up to 128 megabytes of external memory. This address reach supports very high-resolution displays. For example, the designer of a large TMS34010-based system could decide to use the lower half of the address space for display memory, and use the upper half for storing programs and data. Half of this memory space, for example, could be used as a display memory, and the remaining memory can be used for programs and data. The 64-megabyte display memory in this example could support the following display sizes:

- 8192 by 4096 pixels at 16 bits per pixel
- 8192 by 8192 pixels at 8 bits per pixel
- 16,384 by 8192 pixels at 4 bits per pixel
- 16,384 by 16,384 pixels at 2 bits per pixel
- 32,768 by 16,384 pixels at 1 bit per pixel

The video timing registers also support high-resolution displays. The 16-bit vertical counter register, VCOUNT, directly supports screen lengths of up to 65,536 lines. The 16-bit horizontal counter register, HCOUNT, does not directly limit the horizontal resolution. Each horizontal line can be programmed to be up to 65,536 VCLK (video clock) periods long. The VCLK period, however, is an arbitrary number of dot-clock periods in length, depending on the external divide-down logic used to produce the VCLK signal from the dot clock. Thus, the number of pixels per line supported by the GSP horizontal timing registers is limited only by the amount of video memory that is present.

Note that frame buffers in excess of 2^{24} bits may require an external counter to determine which VRAM serial outputs should be enabled during a scan line. This external counter would increment upon detecting a 1-to-0 transition of the logical address bit 23 during successive screen-refresh cycles. To support applications requiring panning and scrolling of the frame buffer, the initial value of this counter immediately following vertical retrace should be capable of being loaded under program control.

9.2 Video Timing Signals

The TMS34010 generates horizontal sync, vertical sync, and blanking signals ($\overline{\text{HSYNC}}$, $\overline{\text{VSYNC}}$, and $\overline{\text{BLANK}}$) on chip. The GSP's video timing logic is driven by the video input clock (VCLK). The sync and blanking signals control the horizontal and vertical sweep rates of the screen and synchronize the screen display to data output by the VRAMs.

$\overline{\text{HSYNC}}$ is the horizontal sync signal used to control external video circuitry. It may be configured as an input or an output via the DXV and HSD bits in the DPYCTL register. When DXV=0 and HDS=0, external video is selected and $\overline{\text{HSYNC}}$ is an input. Otherwise, $\overline{\text{HSYNC}}$ is an output.

$\overline{\text{VSYNC}}$ is the vertical sync signal used to control external video circuitry. It may be configured as an input or an output via the DXV bit in the DPYCTL register. If DXV=1, internal video is selected and $\overline{\text{VSYNC}}$ is an output. If DXV=0, external video is selected and $\overline{\text{VSYNC}}$ is an input.

$\overline{\text{BLANK}}$ is used to turn off a CRT's electron beam during horizontal and vertical retrace intervals. The signal output at the $\overline{\text{BLANK}}$ pin is a composite of the internally generated horizontal and vertical blanking signals. $\overline{\text{BLANK}}$ can also be used to control starting and stopping of the VRAM shift registers.

VCLK is derived from the dot clock of the external video system. VCLK drives the internal video timing logic.

Holding VCLK low for long periods may cause video counter errors. When VCLK is not being clocked for long periods, it should be held at the logic high level. While VCLK is low, the storage nodes within the device rely on their internal capacitance to maintain state information, and if VCLK is held low for a sufficiently long time, charge leakage may cause bit errors.

9.3 Video Timing Registers

The video timing registers are a subset of the I/O registers described in Section 6. The values in the video timing registers control the video timing signals. These registers are divided into two groups:

- ***Horizontal timing registers*** control the timing of the $\overline{\text{HSYNC}}$ signal and the internal horizontal blanking signal.
  - **HCOUNT** counts the number of VCLK periods per horizontal scan line.
  - **HESYNC** specifies the point in a horizontal scan line at which the $\overline{\text{HSYNC}}$ signal ends.
  - **HEBLNK** specifies the endpoint of the horizontal blanking interval.
  - **HSBLNK** specifies the starting point of the horizontal blanking interval.
  - **HTOTAL** defines the number of VCLK periods allowed per horizontal scan line.
- ***Vertical timing registers*** control the timing of the $\overline{\text{VSYNC}}$ signal and the internal vertical blanking signal.
  - **VCOUNT** counts the horizontal scan lines in the screen display.
  - **VESYNC** specifies the endpoint of the $\overline{\text{VSYNC}}$ signal.
  - **VEBLNK** specifies the endpoint of the vertical blanking interval.
  - **VSBLNK** specifies the starting point of the vertical blanking interval.
  - **VTOTAL** specifies the value of VCOUNT at which $\overline{\text{VSYNC}}$ may begin.

9.4 Relationship Between Horizontal and Vertical Timing Signals

Figure 9-1 illustrates the relationship between the horizontal and vertical timing signals in the construction of a two-dimensional raster display pattern. The vertical sync and blanking signals span an entire frame. The horizontal sync and blanking signals span a single horizontal scan line within the frame.

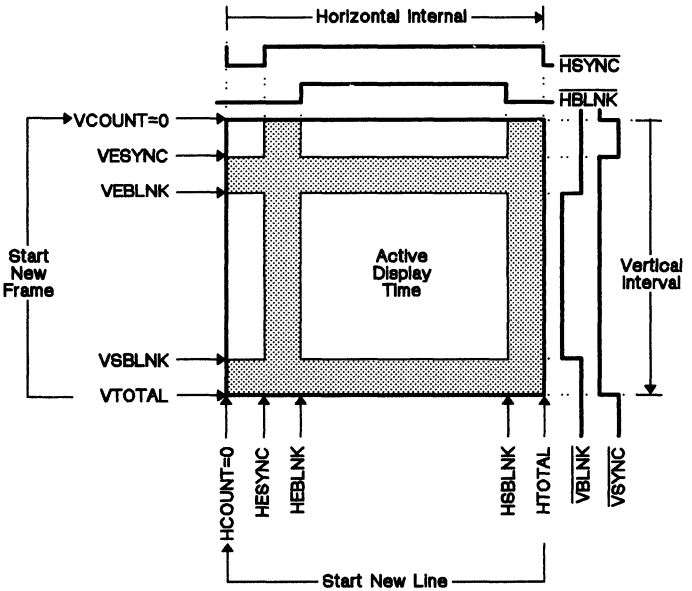


Figure 9-1. Horizontal and Vertical Timing Relationship

Figure 9-1 illustrates the following terms and phrases, which are used throughout this section:

- **$\overline{\text{HBLNK}}$** and **$\overline{\text{VBLNK}}$** are *internal* horizontal and vertical blanking signals that combine to form the $\overline{\text{BLANK}}$ signal output. ($\overline{\text{HBLNK}}$ and $\overline{\text{VBLNK}}$ *cannot* be accessed at TMS34010 pins.) The display is active (not blanked) only when $\overline{\text{HBLNK}}$ and $\overline{\text{VBLNK}}$ are both inactive high.
- **Horizontal front porch** refers to the interval between the beginning of horizontal blanking and the beginning of the horizontal sync signal.
- **Horizontal back porch** is the interval between the end of the horizontal sync signal and the end of horizontal blanking.
- **Vertical front porch** refers to the interval between the beginning of vertical blanking and the beginning of the vertical sync signal.
- **Vertical back porch** is the interval between the end of the vertical sync signal and the end of vertical blanking.

9.5 Horizontal Video Timing

The following discussion applies to internally generated video timing (the DXV and HSD bits in the DPYCTL register are set to 1 and 0, respectively). Horizontal timing signals are the same for interlaced and noninterlaced video.

The HESYNC, HEBLNK, HSBLNK, and HTOTAL registers control horizontal signal timing as shown in Figure 9-2. All horizontal timing parameters are specified as multiples of VCLK. The time between the start of two successive $\overline{\text{HSYNC}}$ pulses is specified by HTOTAL. HCOUNT counts from 0 to the value in HTOTAL and then repeats. The value in HTOTAL represents the number of VCLK periods, minus one, per horizontal scan line. The value in HESYNC represents the duration of the sync pulse, minus one. The values in HEBLNK and HSBLNK specify the beginning and end points of the horizontal blanking interval.

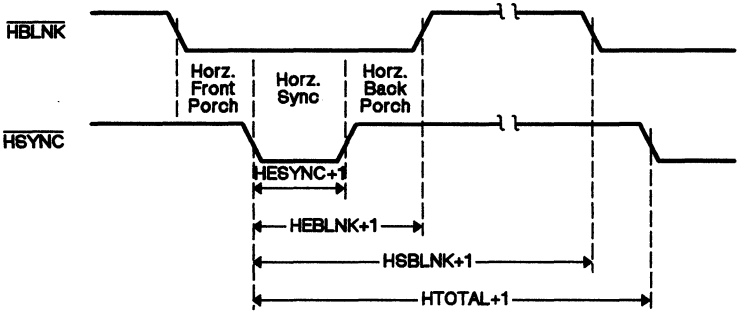


Figure 9-2. Horizontal Timing

Figure 9-3 shows the internal logic used to generate the horizontal timing signals. HCOUNT is incremented once each VCLK period (on the high-to-low transition) until it equals the value in HTOTAL. On the next VCLK period following HCOUNT=HTOTAL, HCOUNT is reset to 0, and begins counting again.

The limits of the horizontal sync pulse are defined by the values in HESYNC and HTOTAL. $\overline{\text{HSYNC}}$ is driven active low when HCOUNT=HTOTAL; it is driven inactive high when HCOUNT=HESYNC. After HCOUNT becomes equal to HTOTAL or HESYNC, there is a one-clock delay before the active/inactive transition takes place at the $\overline{\text{HSYNC}}$ pin.

The internal $\overline{\text{HBLNK}}$ signal is driven active low after HCOUNT=HSBLNK; it is driven inactive high after HCOUNT=HEBLNK. $\overline{\text{HBLNK}}$ is logically ORed (negative logic) with $\overline{\text{VBLNK}}$ to produce the $\overline{\text{BLANK}}$ signal; that is, $\overline{\text{BLANK}}$ goes low when either $\overline{\text{HBLNK}}$ or $\overline{\text{VBLNK}}$ is low. After HCOUNT becomes equal to HSBLNK or HEBLNK, there is a one-clock delay before the transition takes place at the $\overline{\text{BLANK}}$ pin.

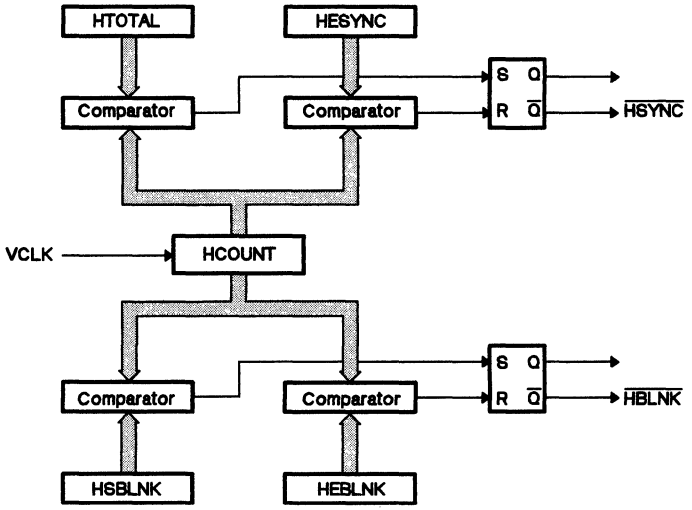
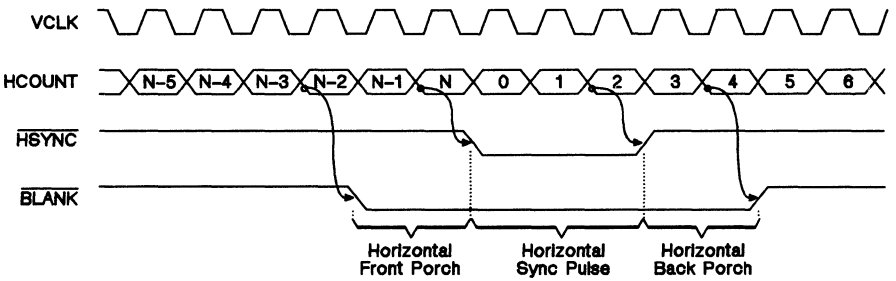


Figure 9-3. Horizontal Timing Logic - Equivalent Circuit

Figure 9-4 illustrates horizontal signal generation. In this example, HTOTAL=N, HSBLNK=N-2, HESYNC=2, and HEBLNK=4. Signal transitions at the $\overline{\text{HSYNC}}$ and $\overline{\text{BLANK}}$ pins occur at high-to-low VCLK transitions. After HCOUNT becomes equal to HTOTAL, HSBLNK, HESYNC, or HEBLNK, there is a one-clock delay before the transition takes place at the $\overline{\text{HSYNC}}$ or $\overline{\text{BLANK}}$ pin. When HCOUNT=HSBLNK (shortly before the end of the horizontal scan), horizontal blanking begins. At this time, the DIP (display interrupt) bit in the INTPEND register is set to 1 if VCOUNT=DPYINT. The next screen-refresh cycle may also occur at this time - the GSP can be programmed to refresh the screen after one, two, three, or four scan lines.



HSBLNK = N-2 HTOTAL = N
HESYNC = 2 HEBLNK = 4

Figure 9-4. Example of Horizontal Signal Generation

9.6 Vertical Video Timing

The following discussion applies to internally generated video timing (the DXV bit in the DPYCTL register is set to 1).

The VESYNC, VEBLNK, VSBLNK, and VTOTAL registers control vertical signal timing as shown in Figure 9-5. All vertical timing parameters are specified as multiples of the horizontal sweep time H, where

$$H = (HTOTAL + 1) \times (VCLK\ period)$$

VTOTAL specifies the time interval between the start of two successive vertical sync pulses; this value is the number of H intervals, less one, in each vertical frame. VESYNC represents the duration of the $\overline{VSYNC}$ pulse, less one, in each vertical frame. $\overline{VSYNC}$'s high-to-low and low-to-high transitions coincide with high-to-low transitions at the $\overline{HSYNC}$ pin.

VSBLNK and VEBLNK specify the starting and ending points of vertical blanking. Blanking begins when VCOUNT=VSBLNK and ends when VCOUNT=VEBLNK. Assuming that horizontal blanking is active at the start of each $\overline{HSYNC}$ pulse, transitions of the internal vertical blanking signal, $\overline{VBLNK}$, occur while horizontal blanking is active.

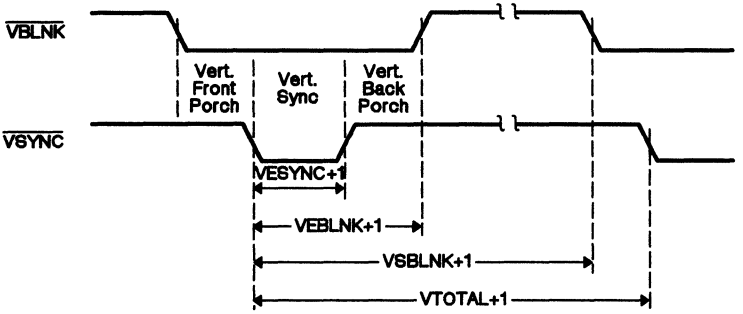


Figure 9-5. Vertical Timing for Noninterlaced Display

Figure 9-6 shows the internal logic that generates the vertical timing signals. VCOUNT increments at the beginning of each $\overline{HSYNC}$ pulse until it equals the value in VTOTAL. When VCOUNT=VTOTAL, VCOUNT is reset to 0 and begins counting again. $\overline{VSYNC}$ is driven active low after VCOUNT=VTOTAL; it is driven inactive high after VCOUNT=VESYNC. The internal $\overline{VBLNK}$ signal is driven active low after VCOUNT=VSBLNK; it is driven inactive high after VCOUNT=VEBLNK. $\overline{VBLNK}$ is logically ORed (negative logic) with $\overline{HBLNK}$ to produce the $\overline{BLANK}$ signal. This description applies to a noninterlaced display. The vertical timing changes slightly for an interlaced display.

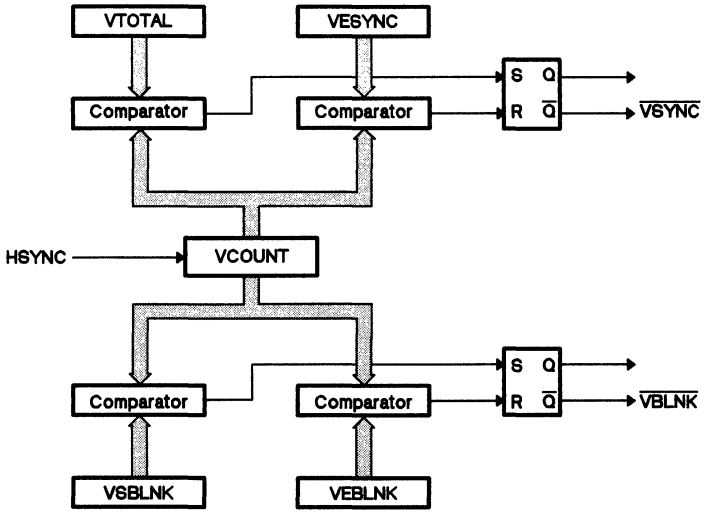


Figure 9-6. Vertical Timing Logic - Equivalent Circuit

9.6.1 Noninterlaced Video Timing

Noninterlaced scan mode is selected by setting the NIL bit in the DPYCTL register to 1. In this mode, each video frame consists of a single vertical field. Figure 9-7 shows the path traced by the electron beam on the screen. Box A shows the vertical retrace, which is an integral number of horizontal scan lines in duration. Box B shows the active portion of the frame. Solid lines represent lines that are displayed; dashed lines are blanked.

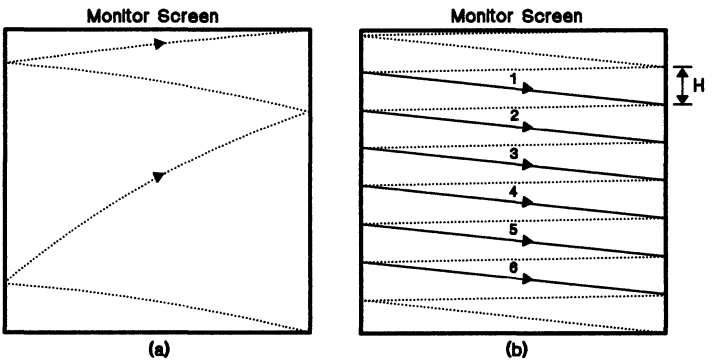


Figure 9-7. Electron Beam Pattern for Noninterlaced Video

Figure 9-8 illustrates the video timing signals that generate the display. In this example, VSBLNK=8, VTOTAL=9, VESYNC=1, and VEBLNK=2. (In actual

applications, much larger values are used; these values were chosen for illustration only.) Each horizontal scan line is preceded by a horizontal retrace. The horizontal scan pattern repeats until VCOUNT=VTOTAL; VCOUNT is then reset to 0, and vertical retrace returns the beam to the top of the screen. $\overline{\text{BLANK}}$ is active low during both horizontal and vertical retrace intervals.

VCOUNT is incremented each time HCOUNT is reset to 0 at the end of a scan line. The $\overline{\text{VSYNC}}$ output begins when VCOUNT=VTOTAL, coinciding with the start of $\overline{\text{HSYNC}}$. The $\overline{\text{VSYNC}}$ output ends when VCOUNT=VESYNC; this also coincides with the start of an $\overline{\text{HSYNC}}$ pulse.

The starting screen-refresh address is loaded from DPYSTRT into DPYADR at the end of the last active horizontal scan line preceding vertical retrace. This load is triggered when HCOUNT=HSBLNK and VCOUNT=VSBLNK.

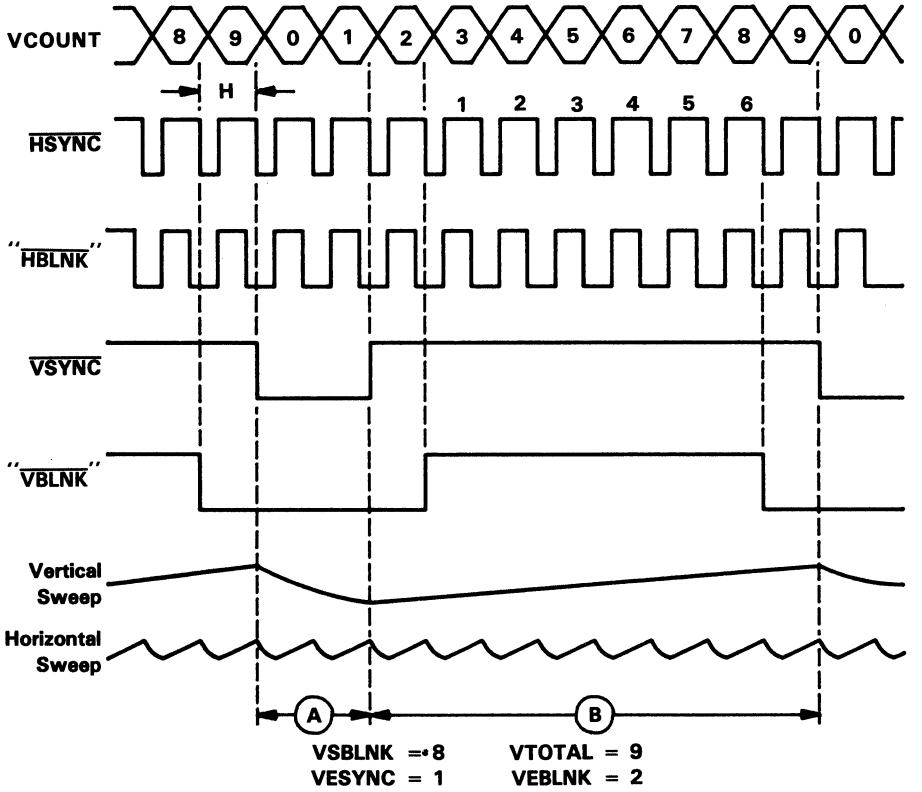


Figure 9-8. Noninterlaced Video Timing Waveform Example

9.6.1.1 Interlaced Video Timing

Interlaced scan mode is selected when the NIL bit in the DPYCTL register is set to 0. In this mode, each display frame is composed of two fields of horizontal scan lines. The display consists of alternate lines from the two fields. This doubles the display resolution while only slightly increasing the frequency with which data is supplied to the screen.

Figure 9-9 illustrates the path traced by the electron beam on the screen. Figure 9-10 shows the timing waveforms used to generate the display in Figure 9-9. In this example, VSBLNK=6, VTOTAL=7, VESYNC=1, and VEBLNK=2. (In actual applications, much larger values are used; these values were chosen for illustration only.)

In interlaced mode, two separate vertical scans are performed for each frame – one for the even line numbers (even field) and one for the odd line numbers (odd field). The even field is scanned first, starting at the top left of the screen (see Figure 9-9 *b*). When VCOUNT=VTOTAL, the vertical retrace returns the beam to the top of the screen, and the odd field is scanned (Figure 9-9 *d*). The electron beam starts scanning the odd and even fields at different points. The reason for this is illustrated in Figure 9-10. The end of the $\overline{\text{VSYNC}}$ pulse that precedes the even field coincides with start of an $\overline{\text{HSYNC}}$ pulse; however, the $\overline{\text{VSYNC}}$ pulse that precedes the odd field ends exactly halfway between two $\overline{\text{HSYNC}}$ pulses

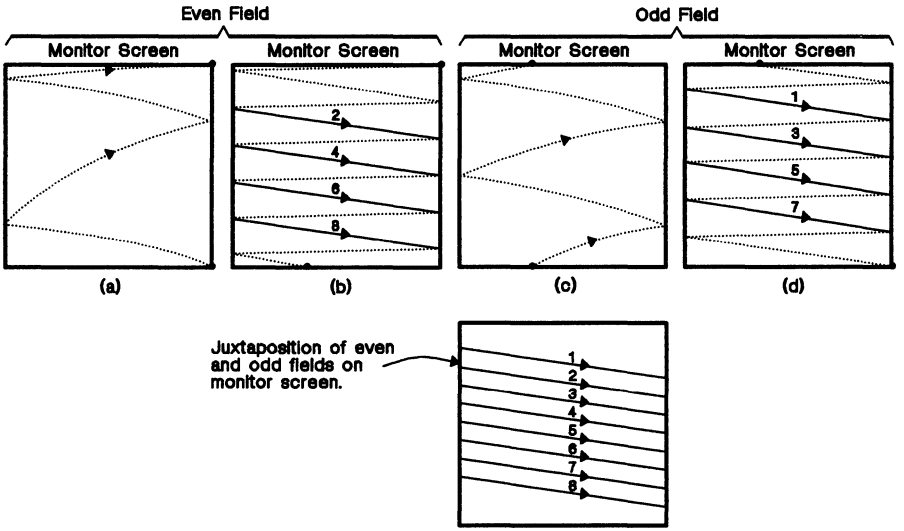


Figure 9-9. Electron Beam Pattern for Interlaced Video

In interlaced mode, video timing logic operation is altered so that the odd field begins when HCOUNT=HTOTAL/2. The beam is thus positioned so that horizontal scan lines in the odd field fall between horizontal scan lines in the even field. To place each line of the odd field precisely between two lines of the even field, load HTOTAL with an odd number.

The transition from *d* to *a* in Figure 9-9 shows that the vertical retrace at the end of the odd field begins at the end of a horizontal scan line; that is, it coincides with the start of an $\overline{\text{HSYNC}}$ pulse, which results from the condition HCOUNT=HTOTAL. The $\overline{\text{VSYNC}}$ pulse duration is an integral number of horizontal scan retrace intervals. When vertical retrace ends and the active portion of the next even field begins, the beam is positioned at the beginning of a horizontal scan line.

Horizontal timing is similar for interlaced and noninterlaced displays. HCOUNT is reset to 0 at the end of each horizontal scan line. A screen-refresh cycle begins before the end of the line, coinciding with the start of the horizontal blanking interval. Assuming that the starting corner of the display is the upper left corner, the DUDATE field of the DPYCTL register is added to the screen-refresh address (SRFADR in the DPYADR register) to generate the row address for the next screen-refresh cycle. In interlaced mode, the DUDATE value must be twice that of the value needed to produce the same display in noninterlaced mode (that is, two times the difference in addresses between consecutive scan lines). This causes the screen refresh to skip alternate lines during the odd and even fields.

At the beginning of each vertical blanking interval, the screen-refresh address (SRFADR in the DPYADR register) is loaded with the starting value specified by the DPYSTRT register. When vertical blanking precedes an even field, the new DPYADR row address is incremented by half the value in the DUDATE field. This is in preparation to display line 2 (Figure 9-9 *b*). When vertical blanking precedes an odd field, the row address loaded into DPYADR from DPYSTRT is not incremented. In this case, the starting row address in DPYSTRT points to the beginning of line 1 (Figure 9-9 *d*).

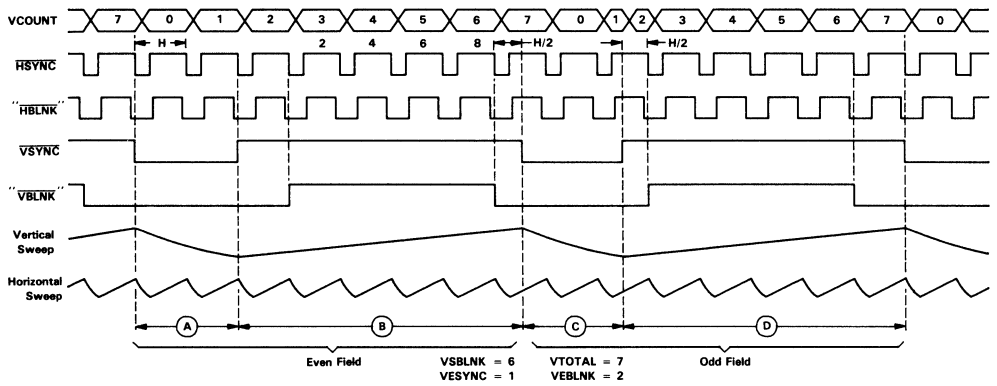


Figure 9-10. Interlaced Video Timing Waveform Example

9.7 Display Interrupt

The TMS34010 can be programmed to interrupt the display when a specified line is displayed on the screen. This is called a **display interrupt**. It is enabled by setting the DIE bit in the INTENB register to 1 and loading the DPYINT register with the desired horizontal scan line number. When VCOUNT = DPYINT, the interrupt request is generated to coincide with the start of horizontal blanking at the end of the specified line.

The display interrupt request can be polled by disabling the interrupt (setting DIE=0) and checking the value of the DIP bit in the INTPEND register. Writing a 0 to DIP clears the request.

The display interrupt has several applications. It can be used to coordinate modifications of the bit map with the display of the bit map's contents, for example. While the bottom half of the screen is displayed, the GSP can modify the bit map of the top half of the screen, and vice versa.

Another use for the display interrupt is in maintaining a cursor on the monitor screen. The cursor image resides in the on-screen memory only during the time the electron beam is scanning the lines containing the cursor. The cursor remains free from flicker even during periods in which the TMS34010 busy drawing to the screen. The technique is to load the DPYINT register with the VCOUNT value of a scan line just above where the top of the cursor is to appear. When the display interrupt occurs, the interrupt service routine performs the following tasks:

- Sets DPYINT to the scan line just below the cursor,
- Saves the portion of the screen where the cursor is to appear, **and**
- PixBlts the cursor onto the screen.

The cursor remains on the screen until the electron beam reaches the bottom of the cursor, at which time a second interrupt request occurs. The original screen is then restored, and the TMS34010 can resume drawing to the screen.

The display interrupt is also useful in split screen applications. By modifying the contents of the DPYADR register halfway through a frame, different parts of the bit map can be displayed on the top and bottom halves of the screen. No special steps are necessary to ensure that loading a new value to DPYADR does not interfere with an ongoing screen-refresh cycle. The display interrupt is requested at the beginning of the horizontal blanking interval. If a screen-refresh cycle occurs during the same horizontal blanking interval, the GSP cannot respond to the interrupt request until the refresh cycle and subsequent updating of DPYADR are complete. This is true whether the interrupt is taken or the GSP simply polls the DIP bit and detects a 0-to-1 transition. After DIP has been set to 1, DPYADR can be loaded with a new value to achieve the split screen anytime before the next screen-refresh cycle.

In interlaced mode, the display interrupt can be used to detect the start of the even field. For this purpose, the DPYINT register is loaded with the value from the VESYNC register. Figure 9-10 (page 9-12) shows that during the odd field, VCOUNT is incremented by 1 halfway through the horizontal interval when the condition VCOUNT=VESYNC is detected. Assuming that HSBLNK=HTOTAL/2, VCOUNT contains the value VESYNC+1 by the time horizontal blanking begins. This means that if DPYINT=VESYNC, the display interrupt is effectively prevented from occurring during the odd field.

9.8 Dot Rate

A typical screen must be refreshed 60 times per second for a noninterlaced scan or 30 times per second for an interlaced scan. For a noninterlaced display, the dot period (time to refresh one pixel) is estimated as:

$$\text{Dot Period} = \frac{(0.8)(1/60 \text{ second})}{(\text{pixels/line}) \times (\text{lines/frame})}$$

For an interlaced display, the dot period is estimated as

$$\text{Dot Period} = \frac{(0.8)(1/30 \text{ second})}{(\text{pixels/line}) \times (\text{lines/frame})}$$

The 0.8 factor in the numerator accounts for the fact that the display is typically blanked for about 20% of the duration of each frame. This factor varies somewhat from monitor to monitor.

During each dot period, the complete information for one pixel must be obtained from the display memory (or frame buffer). Thus, the rate at which video data must be supplied from the display memory (which is usually the limiting factor for large systems) is a function of pixel size as well as screen dimensions.

9.9 External Sync Mode

External sync mode allows the TMS34010 to use horizontal and vertical sync signals from an external source. This permits graphics images generated by the GSP to be superimposed upon or mixed with images from external sources.

External sync mode is selected by setting the DXV and HSD bits in the DPYCTL register to 0. $\overline{\text{HSYNC}}$ and $\overline{\text{VSYNC}}$ are now configured as inputs. (Alternately, $\overline{\text{HSYNC}}$ can be configured as an output and $\overline{\text{VSYNC}}$ as an input by setting DXV=0 and HSD=1.) When an active-low sync pulse is input to one of these pins, the corresponding counter (HCOUNT or VCOUNT) is forced to all 0s. By forcing the counters to follow the external sync signals, the blanking intervals and screen-refresh cycles are also forced to follow the external video signals.

The $\overline{\text{HSYNC}}$ and $\overline{\text{VSYNC}}$ inputs are sampled on each VCLK rising edge. HCOUNT or VCOUNT are cleared 2.5 clock periods (on a VCLK falling edge) following a high-to-low transition at the $\overline{\text{HSYNC}}$ or $\overline{\text{VSYNC}}$ pin, respectively. $\overline{\text{BLANK}}$ remains an output, but its timing is affected because the point at which HCOUNT and VCOUNT are cleared is controlled by the external sync signals. The 2.5-clock delay must be considered when selecting values for the HSBLNK and HEBLNK registers.

9.9.1 A Two-GSP System

One GSP can generate video timing for two GSPs. As Figure 9-11 shows, GSP #1 is configured for internal sync mode (DXV=1) and generates the sync timing. GSP #2 is configured for external sync mode (DXV=0 and HSD=0), and receives the $\overline{\text{HSYNC}}$ and $\overline{\text{VSYNC}}$ inputs from GSP #1. Assume that the video timing registers of the two devices are named as follows:

GSP #1	GSP#2
HCOUNT1	HCOUNT2
HESYNC1	HESYNC2
HSBLNK1	HSBLNK2
HEBLNK1	HEBLNK2
HTOTAL1	HTOTAL2
VCOUNT1	VCOUNT2
VESYNC1	VESYNC2
VSBLNK1	VSBLNK2
VEBLNK1	VEBLNK2
VTOTAL1	VTOTAL2

GSP #2's registers should be programmed in terms of the values in GSP #1's registers, as shown in Table 9-1. The $\overline{\text{BLANK}}$ signals from GSP #1 and GSP #2 are the same, and switch in unison on the same VCLK edges. When HCOUNT1 is cleared on a VCLK falling edge, HCOUNT2 is cleared three full VCLK periods later. For short horizontal blanking periods, HEBLNK2 may need to be loaded with a value that is less than zero. For example, assume that HSBLNK1=HTOTAL1-4 and HEBLNK1=1 (that is, the horizontal blanking interval is six VCLK periods). To ensure that GSP #2's horizontal blanking interval begins and ends at the same time as GSP #1's, GSP #2's registers must be loaded with values so that HSBLNK2=HTOTAL1-8 and HEBLNK2=HTOTAL1-2.

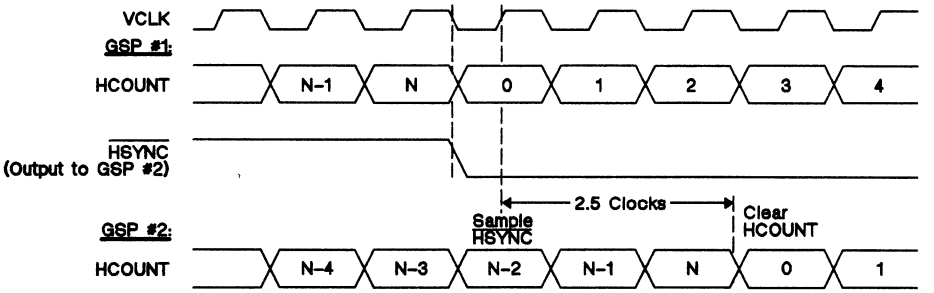


Figure 9-11. External Sync Timing - Two GSP Chips

The values in HTOTAL2 and VTOTAL2 must be large enough so that the conditions HCOUNT=HTOTAL and VCOUNT=VTOTAL do not cause HCOUNT and VCOUNT, respectively, to be cleared before the leading edges of the external horizontal and vertical sync pulses occur. In the example in Table 9-1, HTOTAL2 and VTOTAL2 are set to their maximum values. The value of HESYNC2 must be such that HCOUNT=HESYNC2 occurs between the end of an external $\overline{\text{HSYNC}}$ pulse and the beginning of the next external HSYNC pulse. The value of VESYNC2 must be such that VCOUNT=VESYNC2 occurs between the end of an external $\overline{\text{VSYNC}}$ pulse and the beginning of the next external $\overline{\text{VSYNC}}$ pulse.

Table 9-1. Programming GSP #2 For External Sync Mode

HEBLNK2	=	HEBLNK1 - 3
HSBLNK2	=	HSBLNK1 - 3
HTOTAL2	=	65535
HESYNC2	=	(HEBLNK2 + HSBLNK2)/2 †
VEBLNK2	=	VEBLNK1
VSBLNK2	=	VSBLNK1
VTOTAL2	=	65535
VESYNC2	=	(VEBLNK2 + VSBLNK2)/2 †

† Suggested value; see description in text.

Since the internal counter can only be resolved to the nearest VCLK edge, precise synchronization with an external video source can be achieved only when VCLK is harmonically related to the external horizontal sync signal. In general, however, the $\overline{\text{HSYNC}}$ and $\overline{\text{VSYNC}}$ inputs are allowed to change asynchronously with respect to VCLK, although the precise VCLK edge at which an external sync pulse is recognized can be guaranteed only if the setup and hold times specified for sync inputs are met.

9.9.2 External Interlaced Video

External sync mode can be used for both interlaced and noninterlaced displays. When locking onto external interlaced sync signals, the GSP discriminates between the odd and even fields of the external video signals based on whether its internal horizontal blanking is active at the time that the start of the external vertical sync pulse is detected. In Figure 9-10, for example, the even field begins at a point where $\overline{\text{HBLNK}}$ is active low, and the odd field begins while $\overline{\text{HBLNK}}$ is high.

In interlaced mode, the discrimination between the even and odd fields of an external video source is based on the value of HCOUNT at a point two VCLK periods past the rising VCLK edge at which the GSP detects the $\overline{\text{VSYNC}}$ input's high-to-low transition. If HCOUNT contains a value greater than the value in HEBLNK, but less than or equal to the value in HSBLNK, the GSP assumes that the vertical sync pulse precedes the start of an odd field. Otherwise, the next field is assumed to be even. Alternatively, the GSP can be placed in noninterlaced mode, even though the external sync signals it is locking onto are for an interlaced display. In this case, the GSP simply causes identical display information to be output to the monitor during the odd and even fields.

The program can determine at any time whether an even or odd field is being scanned by inspecting the least significant bits of the DPYADR register to determine whether they have been incremented by DUDATE/2. Recall that that at the start of an even field, the initial address loaded into DPYADR from the DPYSTRT register is automatically incremented by DUDATE/2 (that is, incremented by half the value specified in the DUDATE field of the DPYCTL register). At all other times, DPYADR is incremented by DUDATE rather than DUDATE/2.

9.10 Video RAM Control

The TMS34010 automatically schedules the VRAM (video RAM) memory-to-register cycles needed to refresh a video monitor screen. These cycles are referred to as *screen-refresh* cycles.

In addition to automatic screen-refresh cycles, the GSP can be configured to perform memory-to-register and register-to-memory cycles under the explicit control of software executing on the GSP's internal processor. One of the primary uses for this capability is to facilitate nearly instantaneous clearing of the screen. The screen is cleared in 256 memory cycles or less by means of a technique referred to here as *bulk initialization* of the display memory.

9.10.1 Screen Refresh

A screen-refresh cycle loads the VRAM shift registers with a portion of the display memory corresponding to a scan line of the display. The internal requests for these cycles occur at regular intervals coinciding with the start of the horizontal blanking intervals defined by the video timing registers. When horizontal blanking ends, the contents of the shift registers are clocked out serially to drive the video inputs of a monitor. A screen-refresh cycle typically occurs prior to each active line of the display.

9.10.1.1 Display Memory

The *display memory* is the area of memory which holds the graphics image output to the video monitor. This memory is typically implemented with VRAMs. During a screen-refresh cycle, a portion of the display memory corresponding to one (or possibly more) scan lines of the display are loaded into the VRAM shift registers. Depending on the screen dimensions selected, not all portions of the display memory are necessarily output to the monitor.

The width of the display memory is referred to as the *screen pitch*, which is the difference in 32-bit memory addresses between two vertically-adjacent pixels on the screen. The screen pitch is also the difference in starting memory addresses of the video data for two consecutive scan lines. When XY addressing is used, the screen pitch must be a power of two to facilitate the conversion of XY addresses to memory addresses. The value loaded into the DUDATE field of the DPYCTL register represents the screen pitch, and is the amount by which the screen-refresh address is incremented (or decremented) following each screen-refresh cycle.

The portion of display memory that is actually output to the monitor is referred to as the *on-screen memory*. The starting location of the on-screen memory is specified by the SRFADR field in the DPYSTRT register.

The starting screen-refresh address is output during the screen-refresh cycle that occurs at the start of each frame. At the end of the screen-refresh cycle, the address is incremented to point to the area of memory containing the pixels for the second scan line. The process is repeated for each subsequent scan line of the frame.

A screen-refresh cycle typically affects all video RAMs in the system. A memory-to-register cycle transfers data from a selected row of memory to the internal shift register of each VRAM. The data is then shifted out to refresh the display.

A screen-refresh cycle takes place during the horizontal blanking interval that precedes a scan line to be displayed. Typically, the shift registers containing the video data for the line are clocked only during the active portion of the scan line, that is, when the $\overline{\text{BLANK}}$ output is high. At higher dot rates, the pixel clock or dot clock used to shift video data to the monitor is run through a frequency divider to create the VCLK signal input to the GSP.

The 8-bit row address output during the screen-refresh cycle specifies the row in memory to be loaded into the shift register internal to the VRAM. The number of bits of video data transferred to the shift registers of all the VRAMs in the system during a single screen-refresh cycle is calculated by multiplying the number of VRAMs times the length of the shift register in each VRAM. For example, 64 TMS4161 (64K-by-1) VRAM devices are sufficient to contain the bit map for a 1024-by-1024-pixel display with four bits per pixel. The length of the shift register in each TMS4161 is 256 bits. Thus, in a single screen-refresh cycle, a total of 64 times 256, or 16,384, bits are loaded. This is enough data to refresh four complete scan lines of the display. In general, a single screen-refresh cycle performed during a horizontal blanking interval is sufficient to supply one or more complete scan lines worth of data to the video monitor screen.

9.10.1.2 Generation of Screen-Refresh Addresses

The DPYADR, DPYCTL, DPYSTRT, and DPYTAP registers are used to generate the addresses output during screen-refresh cycles. Figure 9-12 shows these four registers, and indicates the register fields which determine the way in which screen-refresh addresses are generated.

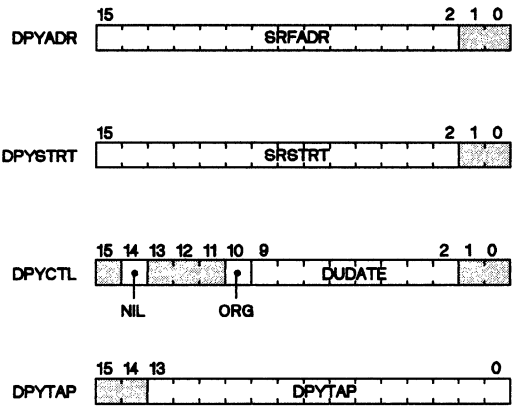


Figure 9-12. Screen-Refresh Address Registers

- DPYADR contains the SRFADR field, which is a counter that generates the addresses output during screen-refresh cycles.

- DPYSTRT contains the SRSTRT field, the starting address loaded into SRFADR at the beginning of each frame.

- DPYCTL contains several fields that affect screen-refresh addresses. The 8-bit DUDATE field is loaded with seven 0s and a single 1 that points to the bit position within SRFADR (bits 2–9 of DPYADR) at which the address is to be incremented (or decremented) at the end of each screen-refresh cycle. The ORG bit determines whether the screen-refresh address is incremented or decremented. If ORG=0, the screen origin is located at the top left corner of the screen and the address is incremented; otherwise, it is decremented. The NIL bit determines whether the GSP is configured to generated an interlaced (NIL=0) or noninterlaced (NIL=1) display. The generation of screen-refresh addresses can be modified to accommodate either type of display.

- The DPYTAP register is used to specify screen-refresh address bits to right of the position at which DUDATE increments the address. DPYTAP provides the additional control over screen-refresh address generation necessary to allow the screen to pan through the display memory.

Bits not directly involved in address generation are shaded in Figure 9-12.

The address output during a screen-refresh cycle identifies the starting pixel on the scan line about to be output to the monitor. Figure 9-13 (page 9-21) shows a 32-bit logical address of the first pixel on one of the scan lines appearing on the screen. The screen-refresh address consists of bits 4–23 of the logical address, which are generated by combining the values contained in SRFADR and DPYTAP. Where SRFADR and DPYTAP overlap (bits 10–17 of the logical address), the address bits are generated by logical ORing the corresponding bits of SRFADR and DPYTAP. The 8-bit DUDATE value contains seven 0s and a single 1 pointing to the position at which SRFADR is to be incremented (or decremented). The DPYTAP register should be loaded with the portion of the pixel address in Figure 9-13 lying to the right of the position indicated by the DUDATE pointer bit. SRFADR contains the portion of the pixel address that is incremented by the DUDATE pointer bit.

Following system power up, the software should load the starting screen-refresh address into the SRSTRT field of the DPYSTRT register, and load the increment to the screen-refresh address into the DPYCTL register. For a typical CRT display, the starting address is the address in memory of the pixel that appears in the upper left corner of the display. If ORG bit in DPYCTL is 0, the *1s complement* of the starting address should be loaded into DPYSTRT. If ORG=1, the starting address loaded into DPYSTRT should *not* be complemented.

DPYADR is automatically loaded with the starting display address from DPYSTRT prior to the start of each frame. As shown in Figure 9-14 *a*, bits 2–15 of DPYSTRT (SRSTRT) are loaded into bits 2–15 of DPYADR (SRFADR). The load occurs coincident with the start of the horizontal blanking interval that occurs just at the end of the last active scan line of the preceding frame.

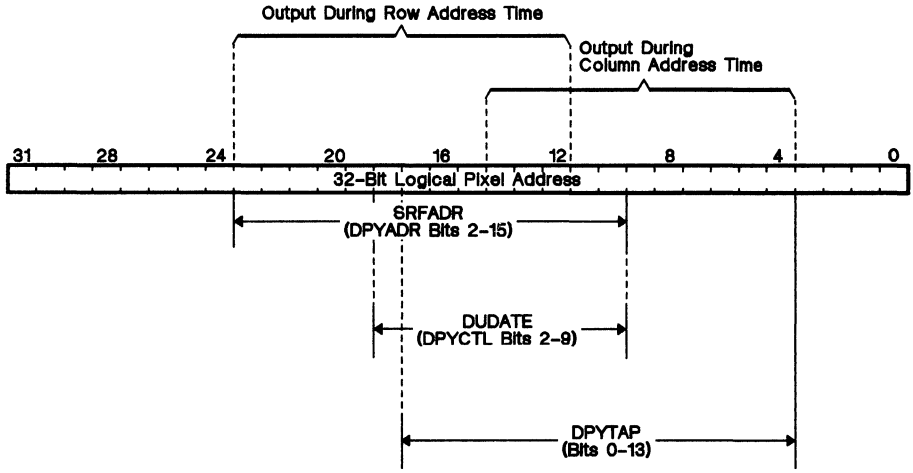
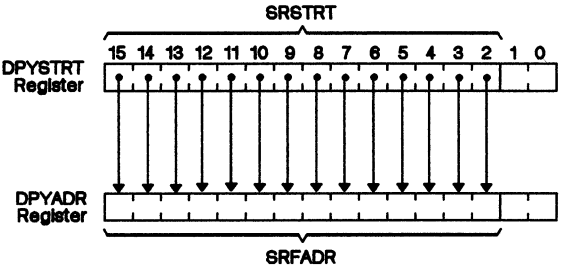


Figure 9-13. Logical Pixel Address

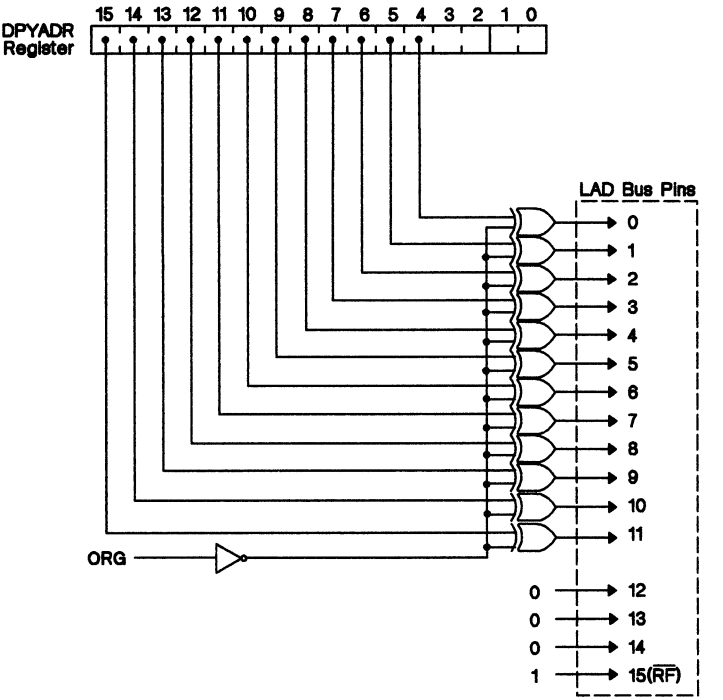
The address output during each screen-refresh cycle is contained in bits 2 through 15 of the DPYADR register (the 14-bit SRFADR field). As shown in Figure 9-14 *b*, DPYADR bits 4–15 are output at the LAD0–LAD11 pins during the row address time of the screen-refresh cycle. If ORG=0, the DPYADR bits are inverted before being output; otherwise, they are output unaltered. Zeros (logic-low level) are output on LAD12–LAD14, and a one (logic-high level) is output on LAD15; this is the $\overline{RF}$ status bit.

During the column address time of the screen-refresh cycle, bits 2–6 of DPYADR are output at LAD6–LAD10. If ORG=0, the DPYADR bits are inverted before being output. DPYTAP bits 6–11 are ORed with DPYADR bits 2–7 and output at LAD6–LAD11. Bits 0–5 and 12–13 of DPYTAP are output at LAD0–LAD5 and LAD11–LAD13, respectively. Zeros are output at LAD14–LAD15 (the $\overline{TR}$ and IAQ status bits).

After the row and column addresses have been output, the address in DPYADR bits 2–15 is decremented by the 8-bit value in DPYCTL bits 2–9 (the DUDATE field). This is done in preparation for the next screen-refresh cycle. The 8-bit DUDATE value is a binary number consisting of seven 0s and a single 1. This single 1 indicates the position at which DPYADR is decremented. If ORG=0, the screen-refresh address in DPYADR is effectively incremented; the 1s complement of the address contained in DPYADR is decremented by the DUDATE amount, but is inverted before being output. This is equivalent to incrementing the address. If ORG=1, the address is decremented.

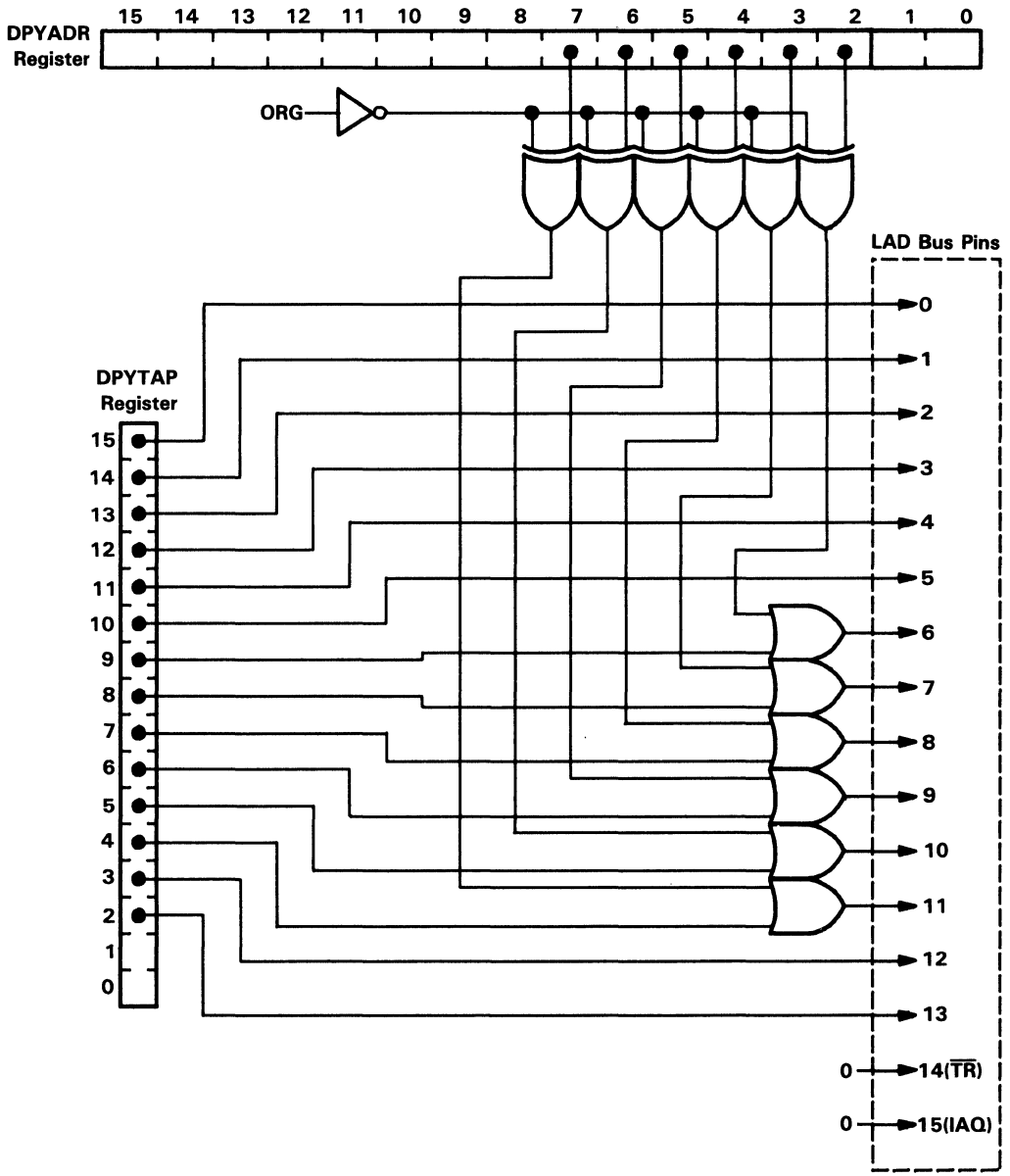


(a) Display-Address Initial Value



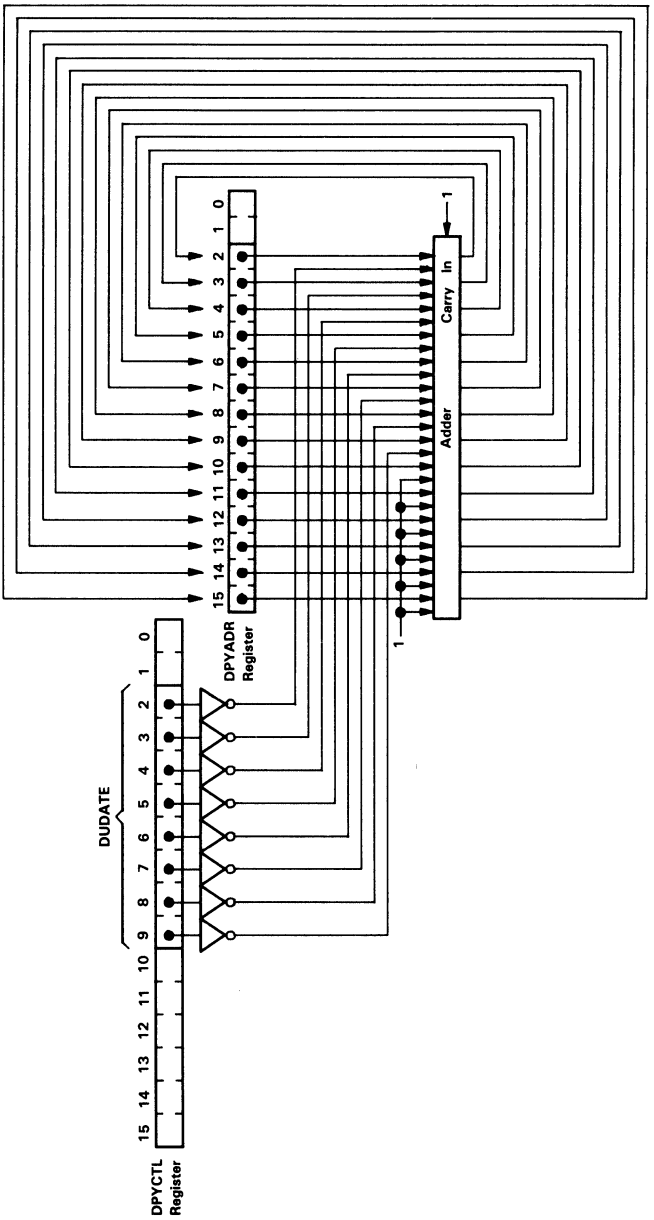
(b) Row-Address Time

Figure 9-14. Screen-Refresh Address Generation



(c) Column-Address Time

Figure 9-14. Screen-Refresh Address Generation (Continued)



(d) Display-Address Update

Figure 9-14. Screen-Refresh Address Generation (Concluded)

9.10.1.3 Screen Refresh for Interlaced Displays

The size of the DUDATE increment specified for an interlaced display should be twice that required for a noninterlaced display of the same dimensions. This allows every other line to be skipped during the even or odd field of an interlaced frame. Before the start of the even field, half the value of the DUDATE increment is added to the starting address loaded into DPYADR to obtain the necessary starting displacement. The SRSTRT field in DPYSTRT points to the area of memory containing the video data for scan line 1 in the example of Figure 9-9 on page 9-11.

9.10.1.4 Panning the Display

The DPYTAP register supports horizontal panning of the screen across a display memory that is larger than the screen. The value contained in the low-order bits of DPYTAP furnish the LSBs of the column address output during the screen-refresh cycle. Incrementing this value results in panning to the right; decrementing this value results in panning to the left.

9.10.1.5 Scheduling Screen-Refresh Cycles

The internal request for a screen-refresh cycle is generated when horizontal blanking begins. This gives the GSP essentially the entire horizontal blanking interval in which to perform the screen-refresh cycle. The delay from the start of horizontal blanking to the start of the screen-refresh cycle is called the *screen-refresh latency*, and is determined by the internal memory controller.

The best and worst case screen-refresh latencies are given in Table 9-2. In the best case, the delay from the high-to-low transition of the $\overline{\text{BLANK}}$ output to the start of the screen-refresh cycle (the time the row address is output) is only 3.25 machine states (or local clock periods). In the worst case, the delay is (7.25 + 2*W*) states, where *W* represents the number of wait states required per memory cycle. The worst case number is based on the fact that the start of the screen-refresh cycle can be delayed by up to three states if a read-modify-write operation began one state before the memory controller received the request for the screen-refresh cycle. A screen-refresh request is given higher priority than requests for DRAM-refresh, host-indirect or GSP CPU cycles; hence, no further delays occur unless an external device generates a hold request.

Table 9-2. Screen-Refresh Latency

Min	Max
3.25 states	(7.25 + 2*W*) states

Note: *W* is the number of wait states per memory cycle.

The horizontal blanking interval should be sufficiently long in duration for the screen-refresh cycle to be completed before blanking ends. The required minimum blanking interval is therefore about (9.25 + 3*W*) machine states, depending on how soon after the end of blanking the external video logic begins clocking the VRAM shift registers. Of course, this time must be translated from

machine states (local clock periods) to VCLK periods to program the HEBLNK register.

The horizontal sync pulse is permitted to be as small as a single VCLK period in duration.

No screen-refresh cycles are performed during vertical blanking until nearly the end of vertical blanking – at the start of the horizontal blanking interval that precedes the first active scan line of the new frame.

The screen-refresh latency specified in Table 9-2 assumes that a local bus hold request ($\overline{\text{HOLD}}$ low) is not asserted between the start of blanking and the start of the screen-refresh cycle. If a hold request prevents the TMS34010 from initiating a scheduled screen-refresh cycle during this time, the TMS34010 is forced to delay its screen-refresh cycle until the bus is released by the external device asserting the hold request. A hold request occurring during the horizontal blanking interval preceding an active scan line on the display should be deasserted in time to allow the TMS34010 to complete the pending screen-refresh cycle before blanking ends. If a screen-refresh cycle is pending at the time the external device releases the bus, the screen-refresh cycle is the first cycle performed by the TMS34010 after it regains control of the bus.

9.10.2 Video Memory Bulk Initialization

VRAMs may be rapidly loaded with an initial value using a special GSP feature that converts pixel accesses to register transfers. This rapid loading method is referred to as bulk initialization of the video memory, and can be used with VRAMs such as the TMS4461. When the SRT (shift register transfer) bit in the DPYCTL register is set to a 1, all reads and writes of pixel data are converted at the memory interface of the GSP to register-transfer cycles. When SRT=0, pixel accesses are performed in normal fashion.

When SRT=1, the processor can initiate register-transfer cycles under explicit program control. By performing a series of such cycles, some or all of the display memory can be set to an initial background color or pattern very rapidly (in a small fraction of one frame time). First, the VRAM shift registers are loaded with the initial value. The video memory is then set to the initial color or pattern one row at a time by writing the shift register contents to the memory.

During a register-transfer cycle (when SRT=1), the row and column addresses are output in unaltered form; that is, the address is not affected by the state of SRT. The 8-bit row address output during the cycle designates which row in memory is involved in the transfer. The direction of the transfer is determined by whether the cycle is a read or a write. A write cycle such as a PIXT transfer from a general-purpose register to memory is converted to a VRAM register-to-memory cycle. Similarly, a read cycle such as a PIXT transfer from memory to a general-purpose register is converted to a VRAM memory-to-register cycle.

Only pixel transfers are affected by the SRT bit. The manner in which all other data accesses and instruction fetches are performed is not affected.

Before bulk initialization of the display memory, the VRAM shift registers are loaded with the solid color or pattern with which the display memory is loaded. This can be done in one of two ways, by either:

- Serially shifting bits into the shift register

  or

- First loading a row of display memory with the color or pattern using a series of "normal" pixel writes (when SRT=0), and then loading the contents of this row into the shift register by means of a PIXT memory-to-register instruction (executed while SRT=1).

To speed up the bulk initialization operation further, a series of transfers can be made more rapidly by using a single FILL instruction in place of a series of PIXT instructions. The fill region is selected so that each pixel write cycle generates a new row address. The fill region is specified to be precisely 16 bits wide, the width of the memory data bus. Also, plane masking is disabled, transparency is turned off, and the pixel processing *replace* operation is selected. This ensures that each row is addressed only once during the course of the fill operation.

The number of bits of the display memory that are altered by a single register-to-memory transfer cycle is calculated by multiplying the number of VRAM devices by the number of shift register bits in each device. The entire frame buffer is loaded with the initial color or pattern in 256 memory cycles.

Section 10

Host Interface Bus

A host processor can communicate with the TMS34010 by means of an interface bus consisting of a 16-bit data path and several transfer-control signals. The TMS34010's host interface provides a host with access to four programmable 16-bit registers (resident on the TMS34010), which are mapped into four locations in the host processor's memory or I/O address space. Through this interface, commands, status information, and data are transferred between the TMS34010 and host processor.

A host processor may read from or write to TMS34010 local memory indirectly via an autoincrementing address register and data port. This optional autoincrement feature supports efficient block moves. The TMS34010 and host can send interrupt requests to each other. A pin is dedicated to the interrupt request from the TMS34010 to the host. To allow block moves initiated by a host to take place more efficiently, the host may suspend TMS34010 program execution to eliminate contention with the TMS34010 for local memory. DRAM-refresh and screen-refresh cycles continue to occur while the TMS34010 is halted.

This section includes the following topics:

10.1 Host Interface Bus Pins

The TMS34010's host interface bus consists of a 16-bit bidirectional data bus and nine control lines. These signals are described in detail in Section 2.

HD0–HD15
form a 16-bit bidirectional bus, used to transfer data between the TMS34010 and a host processor.

$\overline{\text{HCS}}$ is the host chip select signal. It is driven active low to allow a host processor to access one of the host interface registers.

HFS0, HFS1
are function select pins. They specify which of four host interface registers a host can access (see Section 10.2).

$\overline{\text{HREAD}}$ is driven active low to allow a host processor to read the contents of the selected host interface register, output on HD0–HD15.

$\overline{\text{HWRITE}}$ is driven active low to allow a host processor to write the contents of HD0–HD15 to the selected host interface register.

$\overline{\text{HLDS}}$ is driven low to enable a host processor to access the lower byte of the selected host interface register.

$\overline{\text{HUDS}}$ is driven low to enable a host processor to access the upper byte of the selected host interface register.

HRDY informs a host processor when the TMS34010 is ready to complete an access cycle initiated by the host.

$\overline{\text{HINT}}$ transmits interrupt requests from the TMS34010 to a host processor.

10.2 Host Interface Registers

The host interface registers are a subset of the I/O registers discussed in Section 6. The host interface registers can be accessed by both the TMS34010 and the host processor. These registers occupy four 16-bit locations in the host processor's memory or I/O address map. One of these four locations is selected by placing a particular code on the two function select inputs, HFS0 and HFS1, as shown in Table 10-1.

Table 10-1. Host Interface Register Selection

HFS1	HFS0	Selected Register
0	0	HSTADRL
0	1	HSTADRH
1	0	HSTDATA
1	1	HSTCTL

A 16-bit host processor typically connects two of its low-order address lines to HFS0 and HFS1. An 8-bit processor typically connects two low-order address lines to HFS0–HFS1 and uses a third low-order address bit to enable either the upper or lower byte of the selected register by activating one of the

byte select inputs, $\overline{\text{HUDS}}$ or $\overline{\text{HLDS}}$. In the second case, the registers occupy eight 8-bit locations in the host processor's memory map.

- The **HSTADRL** and **HSTADRH** registers contain the 16 LSBs and 16 MSBs, respectively, of a 32-bit pointer address. A host processor uses this address to indirectly access TMS34010 local memory.

- The **HSTDATA** register buffers data that is transferred through the host interface between TMS34010 local memory and a host processor. HSTDATA contains the contents of the address pointed to by the HSTADRL and HSTADRH registers.

- The **HSTCTL** register is accessible to the TMS34010 as two separate I/O registers, HSTCTLL and HSTCTLH, but is accessed by a host processor as a single 16-bit register. HSTCTL contains several programmable fields that control host interface functions.

  - *NMI* (nonmaskable interrupt, bit 8): Allows a host processor to interrupt TMS34010 execution.

  - *NMIM* (NMI mode, bit 9): Specifies if the context of an interrupted program is saved when a nonmaskable interrupt occurs.

  - *CF* (cache flush, bit 14): Setting this bit flushes the contents of the TMS34010 instruction cache. A host processor can force the TMS34010 to execute new code after a download by flushing old instructions out of cache.

  - *LBL* (lower byte last, bit 13): Specifies which byte of a register an 8-bit host processor accesses first.

  - *INCR* (increment address before local read, bit 12): Controls whether the 32-bit pointer in the HSTADR registers is incremented before being used in a local read cycle that updates the HSTDATA register.

  - *INCW* (increment address after local write, bit 11): Controls whether the 32-bit pointer in the HSTADR registers is incremented after being used in a local write cycle that transfers the contents of the HSTDATA register to memory.

  - *HLT* (halt TMS34010 program execution, bit 15): A host processor can halt the TMS34010's on-chip processor by setting this bit to 1.

  - *MSGIN* (message in, bits 0–2): Buffers a 3-bit interrupt message from a host processor to the TMS34010.

  - *INTIN* (input interrupt bit, bit 3): A host must load a 1 into this bit to generate an interrupt request to the TMS34010.

  - *MSGOUT* (message out, bits 4–6): Buffers a 3-bit interrupt message from the TMS34010 to a host.

  - *INTOUT* (Interrupt out, bit 7): The TMS34010 must load a 1 to this bit to send an interrupt request to a host processor.

10.3 Host Register Reads and Writes

Host interface read and write cycles are initiated by the host processor and are controlled by means of the *HCS*, *HWRITE*, *HREAD*, *HUDS*, and *HLDS* signals. Host-initiated accesses of the register selected by the function-select code input on HFS0 and HFS1 are controlled as follows:

- While HCS, HLDS, and HWRITE are active low, the contents of HD0–HD7 are latched into the lower byte of the selected register.
- While HCS, HUDS, and HWRITE are active low, the contents of HD8–HD15 are latched into the upper byte of the selected register.
- While HCS, HLDS, and HREAD are active low, the contents of the lower byte of the selected register are driven onto HD0–HD7.
- While HCS, HUDS, and HREAD are active low, the contents of the upper byte of the selected register are driven onto HD8–HD15.

As this list indicates, at least three control signals *must* be active at the same time to initiate an access. The last of the three signals to become active begins the access, and the first of the three signals to become inactive signals the end of the access. A signal that begins or completes an access is referred to in the following discussion as the *strobe* signal for the cycle. Any of the signals listed above may be a strobe. Figure 10-1 shows a functional representation of the logic that controls the TMS34010's host interface.

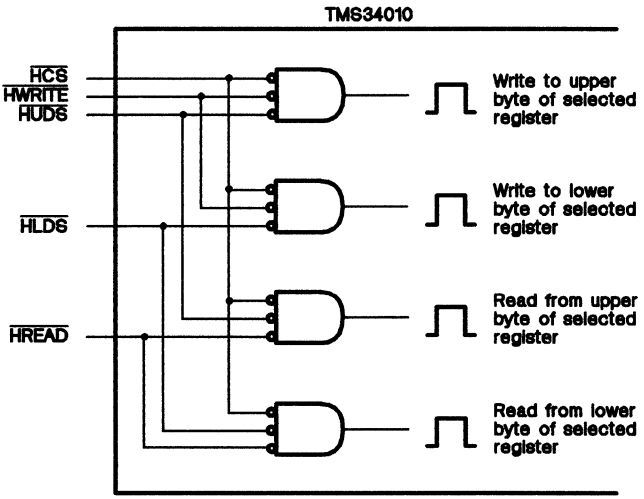


Figure 10-1. Equivalent Circuit of Host Interface Control Signals

The designer must ensure that $\overline{\text{HREAD}}$ and $\overline{\text{HWRITE}}$ are never active low simultaneously during an access of a host interface register; this may cause internal damage to the device.

10.3.1 Functional Timing Examples

The functional timing examples in this section are based on the circuit shown in Figure 10-1.

- The $\overline{\text{HCS}}$ input is the strobe in Figure 10-2 and Figure 10-3.
- The $\overline{\text{HWRITE}}$ signal is the strobe in Figure 10-4.
- The $\overline{\text{HREAD}}$ signal is the strobe in Figure 10-5.
- The $\overline{\text{HUDS}}$ and $\overline{\text{HLDS}}$ signals are strobes in Figure 10-6 and Figure 10-7.

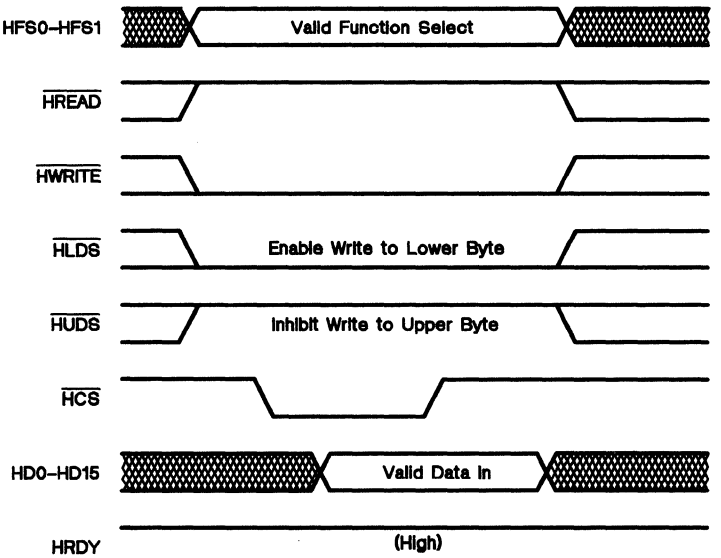


Figure 10-2. Host 8-Bit Write with $\overline{\text{HCS}}$ Used as Strobe

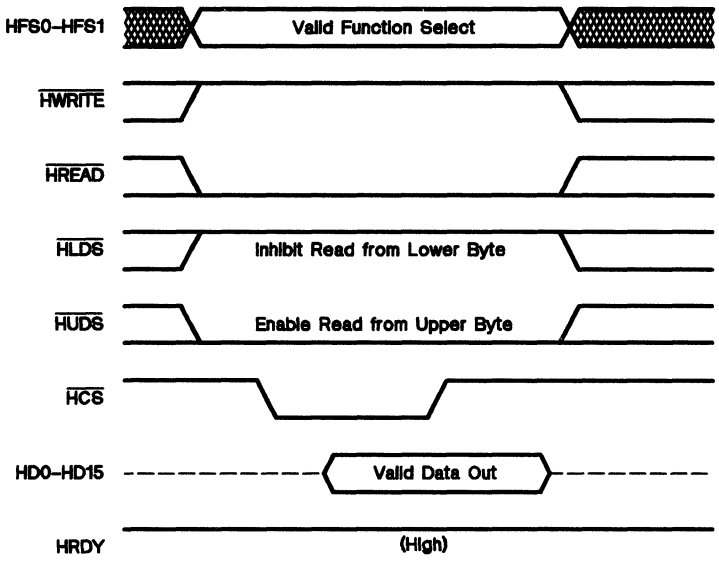


Figure 10-3. Host 8-Bit Read with HCS Used as Strobe

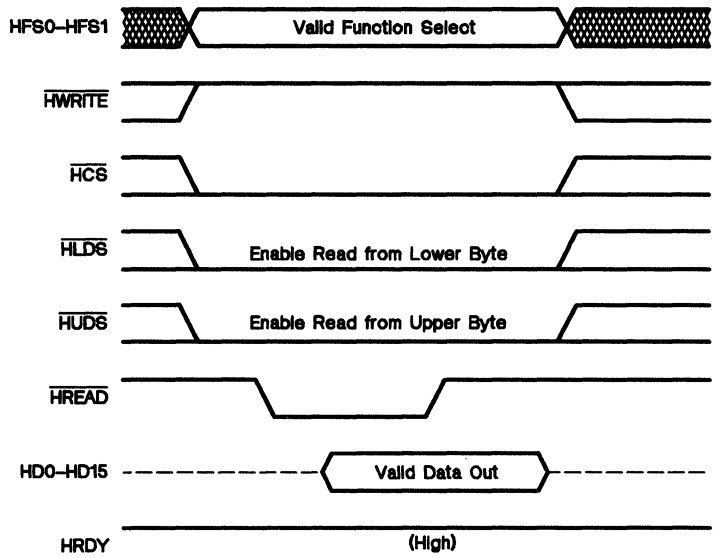


Figure 10-4. Host 16-Bit Read with HREAD Used as Strobe

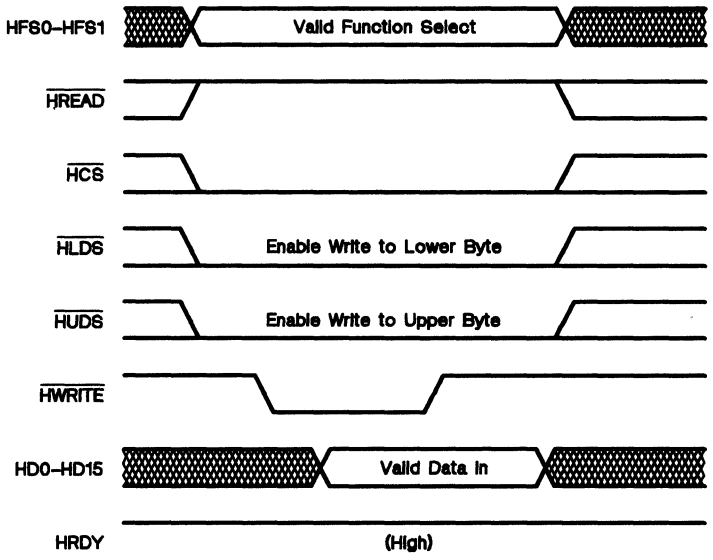


Figure 10-5. Host 16-Bit Write with HWRITE Used as Strobe

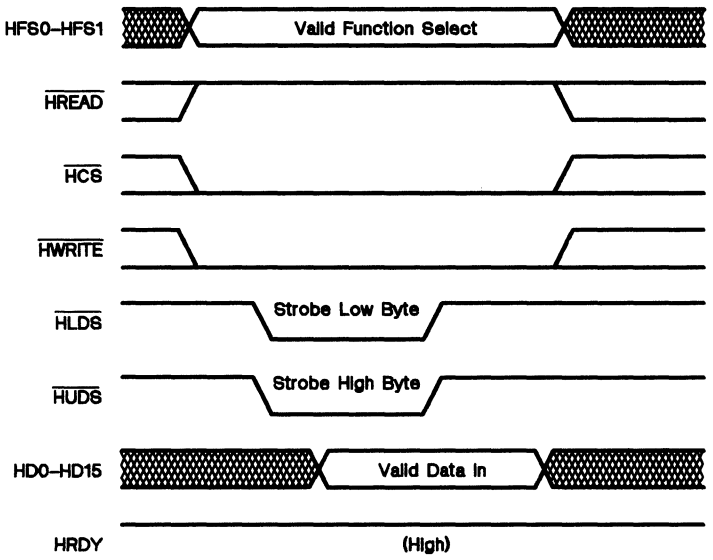


Figure 10-6. Host 16-Bit Write with HLDS, HUDS Used as Strobes

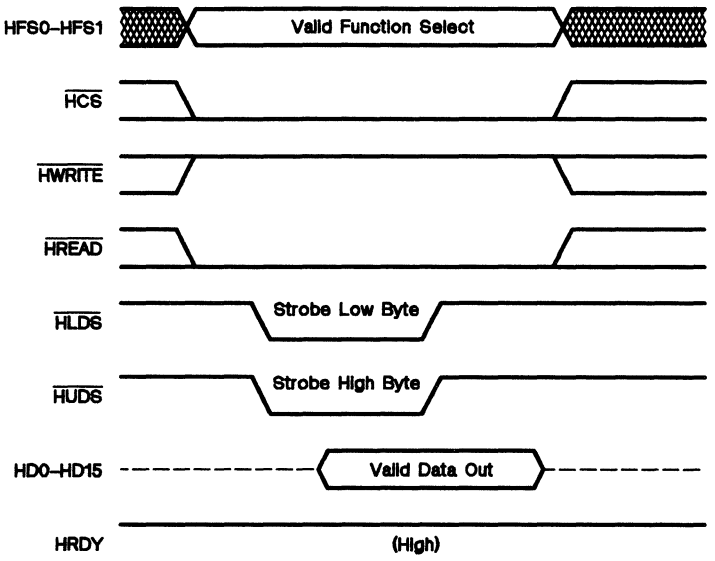


Figure 10-7. Host 16-Bit Read with $\overline{\text{HLDS}}$, $\overline{\text{HUDS}}$ Used as Strobes

10.3.2 Ready Signal to Host

The default state of the bus ready output pin, HRDY, is active high. HRDY is driven inactive low to force the host processor to wait in circumstances in which the TMS34010 is not prepared to allow a host-initiated register access to be completed immediately.

HRDY is always driven low for a brief period at the beginning of a read or write access of the HSTCTL register. When the host attempts to read from or write to the HSTCTL register, HRDY is driven low at the beginning of the access, and is driven high again after a brief interval of one to two local clock cycles.

When the host processor performs certain types of host interface register accesses, a local memory cycle results. For example, in reading from or writing to the HSTDATA register, a read or write cycle on the local bus results. If the host processor attempts to perform an access that initiates a second local memory cycle before the TMS34010 has had sufficient time to complete the first, the TMS34010 drives its HRDY output low to indicate that the host must wait before completing the access. When the TMS34010 has completed the local memory cycle resulting from the previous access, it drives HRDY high to indicate that the host processor can now complete its second access.

A data transfer through the host interface takes place only when some combination of $\overline{\text{HCS}}$, $\overline{\text{HREAD}}$, $\overline{\text{HWRITE}}$, $\overline{\text{HUDS}}$, and $\overline{\text{HLDS}}$ are active simultaneously; however, the HRDY signal is activated by the $\overline{\text{HCS}}$ input alone. HRDY can be active-low only while the TMS34010 is chip-selected by the host processor,

that is, while $\overline{HCS}$ is active low. A high-to-low transition on HRDY follows a high-to-low transition on $\overline{HCS}$. The benefit of this mode of operation is that HRDY becomes valid as soon as $\overline{HCS}$ goes low, which typically is early in the cycle. HRDY is always driven high when $\overline{HCS}$ is inactive high.

A transient low level on the $\overline{HCS}$ input may cause a corresponding low pulse on the HRDY output. Systems that cannot tolerate such transient signals must be designed to prevent $\overline{HCS}$ from going low except during a valid host interface access.

In summary, the following rules govern the HRDY output:

1) If a high-to-low $\overline{HCS}$ transition occurs while the TMS34010 is still completing a local memory cycle resulting from a previous host-indirect access, HRDY goes low. If the register selected is HSTDATA, HSTADRL or HSTADRH, HRDY remains low until the local memory cycle is completed. If the register selected is HSTCTL, the HRDY output remains low for one to two local clock periods.

2) If the host is given a ready signal (HRDY high) to allow it to complete a register access that causes a local memory read or write cycle, HRDY stays high to the end of the access. The access ends when the *strobe* for the cycle ends. The strobe ends when $\overline{HREAD}$ and $\overline{HWRITE}$ are both inactive high, or when $\overline{HLDS}$ and $\overline{HUDS}$ are both inactive high, or when $\overline{HCS}$ is inactive high, whichever is the first to occur. As soon as the strobe ends, a low level on $\overline{HCS}$ allows HRDY to go low again. If the strobe is an input other than $\overline{HCS}$, and $\overline{HCS}$ remains low after the strobe ends, HRDY can go low as a delay from the end of the strobe. If $\overline{HCS}$ is the strobe for the access, the access ends when $\overline{HCS}$ goes high, and HRDY can go low again as soon as $\overline{HCS}$ goes low again.

3) If HSTCTL is selected (FS0 = FS1 = 1) at the high-to-low transition of $\overline{HCS}$, HRDY goes low as a delay from the fall of $\overline{HCS}$, and remains low for one to two local clock periods. To avoid a low-going pulse on HRDY when accessing a register other than HSTCTL, FS0 and FS1 should be valid prior to the high-to-low transition of $\overline{HCS}$.

Figure 10-8 and Figure 10-9 (page 10-10) show examples of host interface register accesses in which HRDY is driven low.

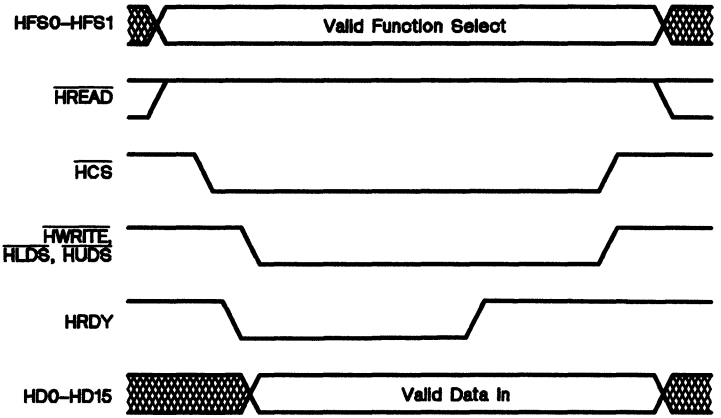


Figure 10-8. Host Interface Timing - Write Cycle With Wait

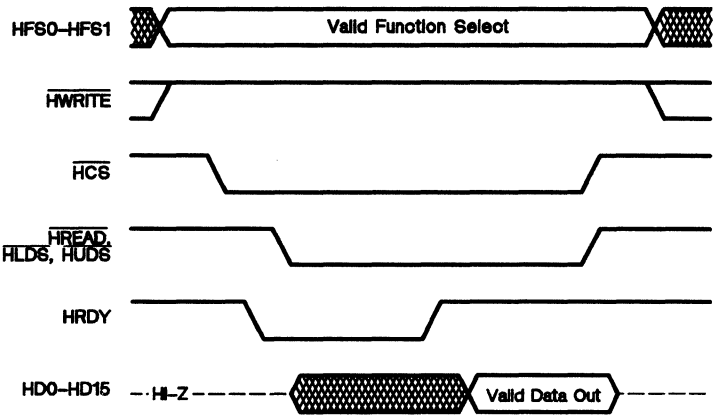


Figure 10-9. Host Interface Timing - Read Cycle With Wait

10.3.3 Indirect Accesses of Local Memory

The host processor indirectly accesses TMS34010 local memory by reading from or writing to the HSTDATA register. HSTDATA buffers data written to or read from the local memory. The word in local memory that is accessed is the word pointed to by the 32-bit address contained in the HSTADRL and HSTADRH registers. The pointer address is loaded into HSTADRL and HSTADRH by the host processor before performing one or more indirect accesses of local memory using the HSTDATA register.

The four LSBs of HSTADRL are forced to 0s internally so that the address formed by HSTADRL and HSTADRH always points to a word boundary in local memory. Between successive indirect accesses of local memory using the HSTDATA register, the local memory address contained in the HSTADR registers can be autoincremented by 16. This allows the host processor to access a block of sequential words in local memory without the overhead of loading a new address prior to each access.

During a sequence of one or more indirect reads of local memory by the host, the TMS34010 maintains in HSTDATA a copy of the local memory word currently addressed by the HSTADRL and HSTADRH registers. Reading from HSTDATA returns the word prefetched from the local memory location pointed to by the HSTADRL and HSTADRH registers, and causes HSTDATA to be updated from local memory again. Writing to HSTDATA causes the word written to HSTDATA to subsequently be written to the location in local memory pointed to by the HSTADRL and HSTADRH registers.

Two increment-control bits, INCR and INCW (contained in the HSTCTL register), are set to 1 to cause the pointer address in HSTADRL and HSTADRH to be incremented by 16 during reads and writes, respectively. In preparing to use the autoincrement feature, the appropriate increment-control bit, INCR or INCW, is loaded with a 1, and the HSTADRL and HSTADRH registers are set up to point to the first location of a buffer region in the local memory.

- When **INCR=1**, a read of HSTDATA causes the address in HSTADRL and HSTADRH to be incremented *before* it is used in the local memory read cycle that updates HSTDATA.

- When **INCW=1**, a write to HSTDATA causes the address in HSTADRL and HSTADRH to be incremented *after* it is used in the local memory read cycle that writes the new contents of HSTDATA to local memory.

Loading the pointer address automatically triggers an update of HSTDATA to the contents of the local memory word pointed to. No increment of HSTADRL and HSTADRH takes place at this time regardless of the state of the increment bits. Each subsequent host access of HSTDATA causes HSTADRL and HSTADRH to be automatically incremented (assuming INCR or INCW is set) to point to the next word location in the local memory. In this manner, a series of contiguous words in local memory can be accessed following a single load of the HSTADRL and HSTADRH registers without additional pointer-management overhead.

10.3.3.1 Indirectly Reading from a Buffer

Figure 10-10 illustrates the procedure for reading a block of words beginning at local memory address *N*. Assume that the INCR bit in the HSTCTL register is set to 1 and the LBL bit in HSTCTL is set to 0.

- In Figure 10-10 *a*, the host processor loads the 32-bit address *N* into HSTADRL and HSTADRH.
- The loading of the second half of the address into HSTADRH causes the TMS34010 host interface control logic to automatically initiate a read cycle on the local bus. This read cycle, shown in Figure 10-10 *b*, transfers the contents of memory address *N* to the HSTDATA register.
- In *c*, the host processor reads the HSTDATA register, fetching the data previously read from address *N*.
- The read of HSTDATA by the host processor causes the TMS34010 to automatically increment the contents of HSTADRL and HSTADRH by 16, as shown in *d*.
- The contents of the new address are read into HSTDATA, as shown in Figure 10-10 *e*. This data will be available in HSTDATA the next time it is read by the host processor.

The process shown in *c* through *e* repeats for every word read from TMS34010 local memory.

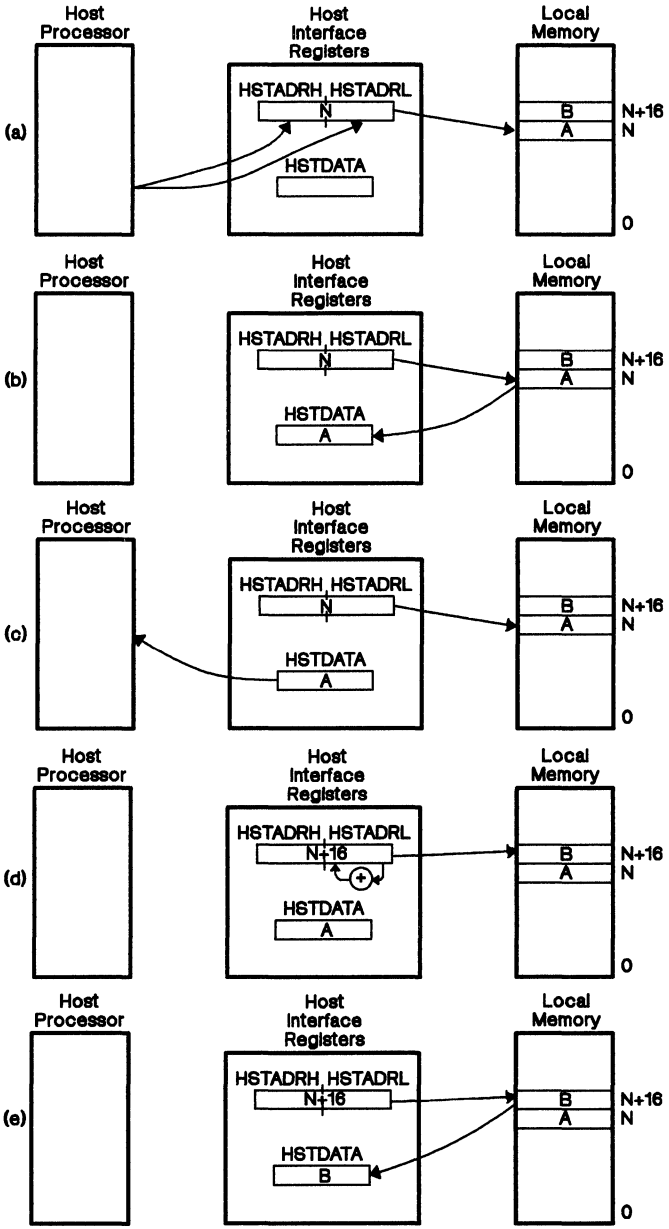


Figure 10-10. Host Indirect Read from Local Memory (INCR=1)

10.3.3.2 Indirectly Writing to a Buffer

Figure 10-11 illustrates the procedure for writing a block of words to TMS34010 local memory. The block begins at address *N*. Assume that the INCW bit is set to 1 and the LBL bit is set to 0.

- In Figure 10-11 *a*, the host processor loads the 32-bit address *N* into HSTADRL and HSTADRH.
- The loading of the second half of the address into HSTADRH causes the TMS34010 host interface control logic to automatically initiate a read cycle on the local bus. This read cycle, which takes place in Figure 10-11 *b*, fetches the contents of memory address *N* into HSTDATA.
- The data loaded into this register is not used, however. Instead, the host processor writes to the HSTDATA register in Figure 10-11 *c*, overwriting its previous contents.
- In response to the host's write to HSTDATA, the TMS34010 automatically initiates a write cycle to transfer the contents of HSTDATA to the local memory address *N* as shown in *d*.
- Following the write, the TMS34010 automatically increments the address in HSTADRL and HSTADRH to point to the next word, as shown in *e*. At this point the host interface registers are ready for the host processor to write the next word to HSTDATA.

The process shown in *c* through *e* repeats for every word written to TMS34010 local memory.

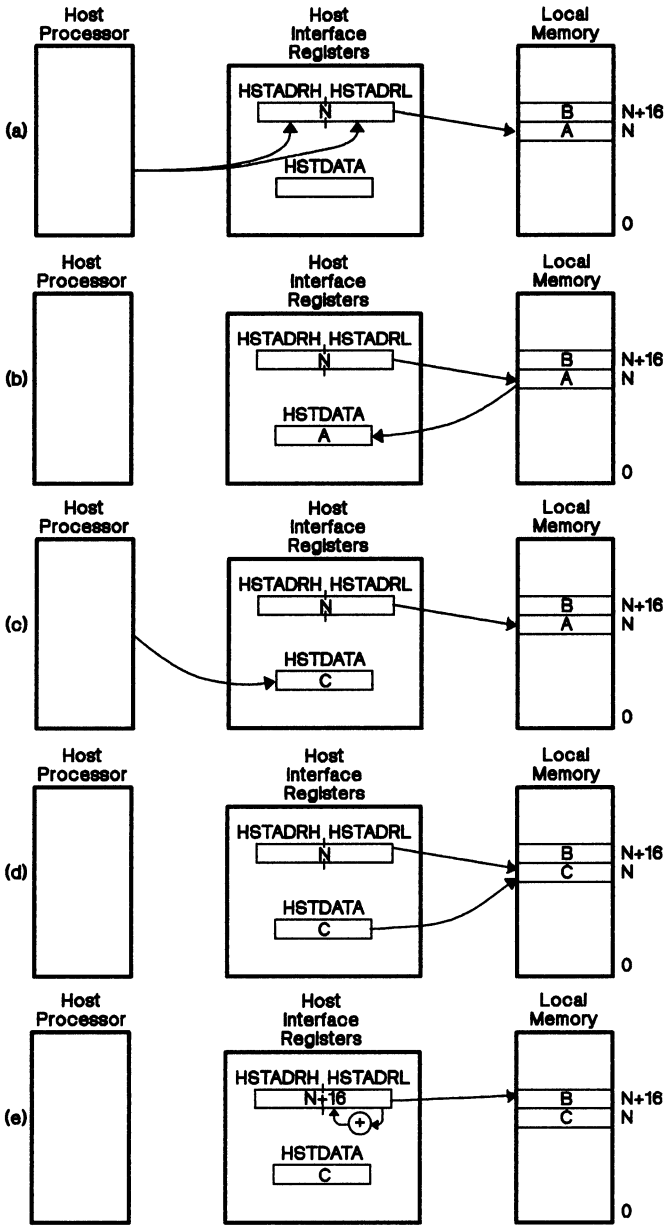


Figure 10-11. Host Indirect Write to Local Memory (INCW=1)

10.3.3.3 Combining Indirect Reads and Writes

If the HSTDATA register in Figure 10-11 is read by the host processor following step *e*, the value returned is the value that the host previously loaded into the register. The host must read HSTDATA a second time to access data from TMS34010 local memory. This principle is illustrated in Figure 10-12, which shows how the host interface performs when a write is followed by two reads. For this example, INCW=1 and INCR=0.

- In Figure 10-12 *a*, HSTADRL and HSTADRH together point to location *N* in the TMS34010's local memory. The host processor is shown writing to HSTDATA.
- In *b*, the data buffered in HSTDATA is written to location *N* in memory.
- The address registers are incremented in *c*.
- In *d*, the host processor reads the HSTDATA register, which returns the value that the host loaded into the register in step *a*.
- Reading HSTDATA causes a memory read cycle to take place in *e*, which loads the value from memory address *N*+16 into HSTDATA.
- In *f*, a second read of HSTDATA by the host processor returns the value from memory address *N*+16.

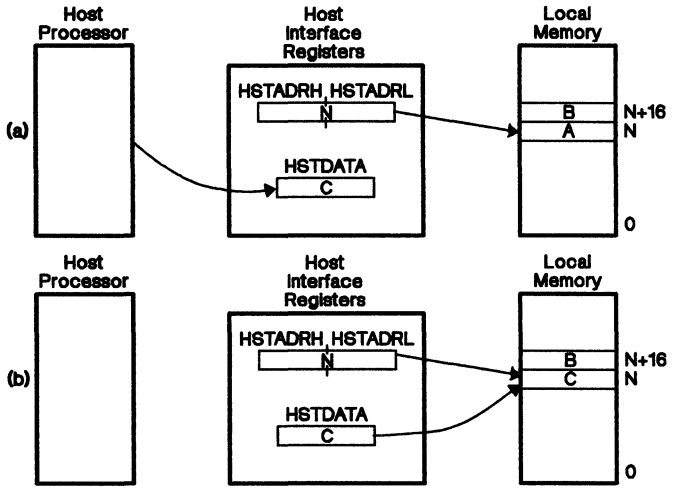


Figure 10-12. Indirect Write Followed by Two Indirect Reads (INCW=1, INCR=0)

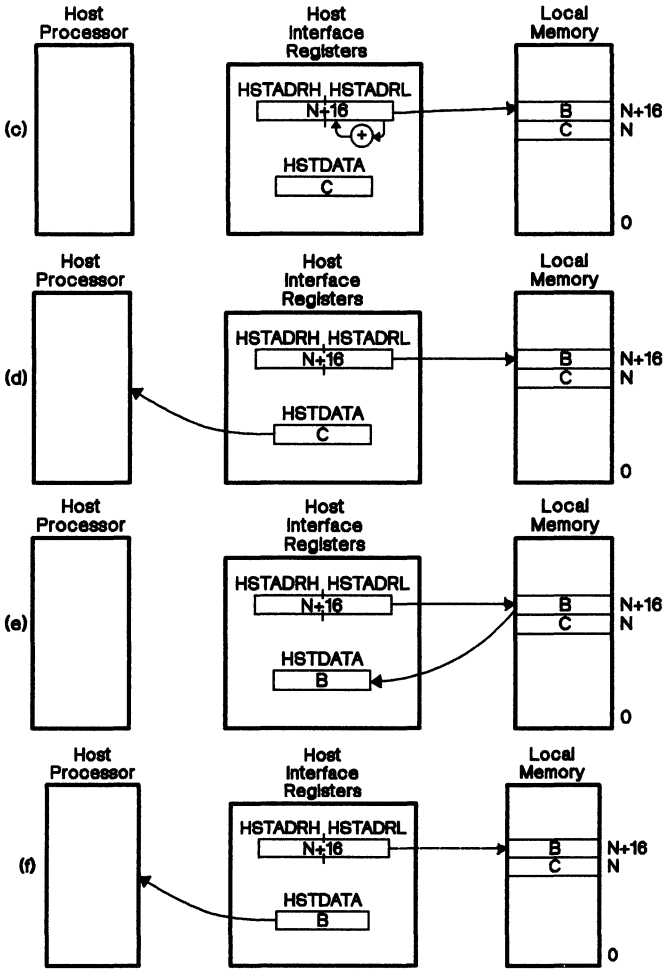


Figure 10-12. Indirect Write Followed by Two Indirect Reads (INCW=1, INCR=0) (Concluded)

10.3.3.4 Accessing Host Data and Address Registers

When the TMS34010 internal processor accesses the HSTDATA, HSTADRL, or HSTADRH register, no subsequent cycle occurs to transfer data between HSTDATA and local memory. Also, the address in HSTADRL and HSTADRH is not incremented, regardless of the state of the INCR and INCW bits.

The host processor can indirectly access any register in the TMS34010's internal I/O register file by first loading HSTADRL and HSTADRH with the address of the register, and they writing to or reading from HSTDATA.

No hardware mechanism is provided to prevent simultaneous accesses of the HSTDATA, HSTADRL and HSTADRH registers by the host processor and by the TMS34010 internal processor. Software must be written to avoid simultaneous accesses, which can result in invalid data being read from or written to these registers.

10.3.3.5 Downloading New Code

The TMS34010 host interface provides a means of efficiently downloading new code from a host processor to TMS34010 local memory. The host initiates this operation through the following process:

- Before downloading, the host interrupts and halts the TMS34010 by writing 1s to the HLT and NMI bits in the HSTCTL register. The host processor should then wait for a period of time equal to the TMS34010 interrupt latency. (TMS34010 hardware resets the NMI bit if the nonmaskable interrupt is initiated before the halt occurs.)

- The code is then downloaded using the auto-increment features of the host interface registers.

- After downloading the code, the host should flush the cache as described in Section 5.4.5, Flushing the Cache (page 5-23).

- The nonmaskable interrupt vector is written through the host port to location FFFFFEE0h so that the new code begins execution at the vectored address.

- The NMI bit in the HSTCTL register should be set to 1 to initiate a nonmaskable interrupt. At the same time, the NMIM bit in the HSTCTL register should be set to 1. If the host does not need the current context to be stored on the stack, or if the nonmaskable interrupt was taken in the first step, the NMIM bit should be set to 1. Otherwise, NMIM should be set to 0.

- The host restarts the TMS34010 by writing a 0 to the HLT bit in the HSTCTL register.

Setting the HLT and NMI bits to 1 simultaneously reduces the worst-case delay (compared to setting HLT only). NMI latency is the delay from the 0-to-1 transition of the NMI bit and the start of execution of the first instruction of the NMI service routine. Halt latency is the delay from the 0-to-1 transition of the HLT bit and the time at which the TMS34010 actually halts (see Section 10.3.4). The maximum NMI latency may be much less than the halt la-

tency if a PIXBLT, FILL, or LINE instruction is in progress at the time of the NMI or halt request. An NMI request interrupts instruction execution at the next interruptible point, but a halt request is ignored until the executing instruction completes or is interrupted. When NMI and HLT are set to 1 simultaneously, the TMS34010 halts before beginning execution of the first instruction in the NMI service routine. Therefore, the delay from the setting the NMI and HLT bits to the time that the TMS34010 actually halts is simply the NMI latency.

10.3.4 Halt Latency

The TMS34010 may be halted by a host processor via the HLT bit in the HSTCTL register. The delay from the receipt of a halt request to the time that the TMS34010 actually halts is the sum of five potential sources of delay:

1) Halt request recognition
2) Screen-refresh cycle
3) DRAM-refresh cycle
4) Host-indirect cycle
5) Instruction completion

In the best case, items 2 through 5 cause no delay. The minimum delay to due to item 1 is one machine state.

- The **halt request recognition** delay is the time required for the setting of the $\overline{\text{HLT}}$ bit to be internally synchronized after the low-to-high transition of the HRDY pin.

- The **screen-refresh** and **DRAM-refresh cycles** are a potential source of delay, but in fact occur rarely and are unlikely to delay a halt.

- The likelihood of a delay caused by a **host-indirect cycle** is small in most instances, but this depends largely on the application. It would only occur if the host had written to the data register just prior to writing to the HLT bit. The delay due to a single host-indirect cycle is two machine states, assuming no wait states.

- The instruction completion time refers to the time required for an instruction that was already executing at the time the halt request was received to complete. Note that the TMS34010 halt condition is entered only on **instruction** boundaries. This means that a PIXBLT, FILL, or LINE instruction that is already in progress runs to completion before the TMS34010 halts.

Table 10-2 shows the minimum and maximum times for each of the five operations listed. The halt latency is calculated as the sum of the numbers in the five rows. In the best case, the halt latency is only one machine state. The worst-case latency is six machine states plus the delays due to host-indirect cycles and instruction completion. Table 10-3 shows instruction completion times for some of the longer instructions. However, a PIXBLT, FILL, or LINE instruction may take longer than the times shown in Table 10-3, depending on the size of the pixel array or line specified. Table 10-3 also shows the instruction completion time for a JRUC instruction that jumps to itself – the TMS34010 may be executing this instruction if the software is simply waiting for a halt.

Table 10-2. Five Sources of Halt Delay

Operation	Latency (In States)	
	Min	Max
Halt recognition	1	2
Instruction completion	0	See Table 10-3
DRAM-refresh cycle	0	2 See Note 2
Screen-refresh cycle	0	2 See Note 2
Host-indirect cycle	0	See Note 1

Notes: 1) The latency due to host-indirect cycles depends on both the hardware system and the application. The delay due to a single host-indirect cycle is two machine states, assuming no wait states.
2) DRAM-refresh and screen-refresh cycle times assume no wait states.

Table 10-3. Sample Instruction Completion Times

Instruction	Worst-Case Instruction Completion Time (In States)	
	SP Aligned	SP Not Aligned
DIVS A0,A2	43	43
MMFM SP,ALL	72	144
MMTM SP,ALL	73	169
PIXBLT, FILL, and LINE	See Note 1	See Note 1
Wait: JRUC wait	1	1

Notes: 1) The worst-case instruction completion time is equal to the instruction execution time less one machine state.
2) The SP-aligned case assumes that the SP is aligned to a word boundary in memory.

10.3.5 Accommodating Host Byte-Addressing Conventions

Processor architectures differ in the manner in which they assign addresses to bytes. The TMS34010 host interface logic can be programmed to accommodate the particular byte-addressing conventions used by a host processor.

This ability is important in ensuring software compatibility between 8- and 16-bit versions of the same processor, such as the 8088 and 8086 or the 68008 and 68000. The 8088 transfers a 16-bit word as a series of two 8-bit bytes, low byte first, high byte second. The 68008 transfers the high byte first, and low byte second.

The HSTCTL register's LBL bit is used to configure the TMS34010 host interface to accommodate different byte-accessing methods. The host interface is configured to operate according to the following two principles:

1) First, when a host processor with an 8-bit data bus reads from or writes to the HSTDATA register, it accesses the high and low bytes of the register in separate cycles. The TMS34010 does not initiate its local memory access until both bytes of HSTDATA have been accessed.

2) Second, when HSTADRH and HSTADRL are loaded by the host, the TMS34010 must not initiate its read of the local memory until the complete pointer address has been loaded into HSTADRL and HSTADRH.

When LBL=0:

- A local memory read cycle is intitiated by the TMS34010 when the host processor reads the high byte of HSTDATA, or writes to the high byte of HSTADRH.

- A local memory write cycle is initiated by the TMS34010 when the host processor writes to the high byte of HSTDATA.

When LBL=1:

- A local memory read cycle is initiated by the TMS34010 when the host processor reads the low byte of HSTDATA, or writes to the low byte of HSTADRL.

- A local memory write cycle is initiated by the TMS34010 when the host processor writes to the low byte of HSTDATA.

When the host processor is an 8088, for example, the TMS34010 is typically configured by setting the LBL bit of the HSTCTL register to 0. When configured in this manner, the TMS34010 expects the HSTADRL register to be loaded first, and HSTADRH loaded second. Furthermore, the high byte of the HSTADRH register is expected to be loaded after the low byte. When LBL is set to 0, a local read cycle is initiated when the upper byte of the HSTADRH register is written to by the host processor. This permits the lower byte of HSTADRH to be loaded first without causing side effects.

10.4 Bandwidth

One measure of the performance of the host interface is its data rate, or bandwidth. The bandwidth is the number of bits per second that can be transferred through the host interface during a block transfer of data to or from TMS34010 memory. Assume that the host interface address register is programmed to autoincrement. The maximum data rate through the host interface can be expected to approach the bandwidth of the TMS34010's memory. For example, assume a 50-MHz TMS34010 and a memory requiring no wait states. The memory cycle time is about 320 nanoseconds (bandwidth = 50 megabits/second). The host's access cycle time at the host interface is somewhat longer than this due to certain additional delays inherent in the operation of the TMS34010's internal host interface logic. Also, the throughput of the host interface may depend on whether or not the TMS34010 is halted.

The bandwidth is calculated as the width of the host data path (16 bits) times the frequency of access cycles through the host interface. Given a continuous series of word accesses, with successive accesses occurring at regular intervals, what is the minimum interval between host accesses that the interface can sustain without having to send not-ready signals to the host? (The TMS34010 drives its HRDY output low temporarily to inform the host when the TMS34010 is not yet ready to complete the host's current access.)

First, when the TMS34010 is halted, the host interface should support continuous accesses occurring at regular intervals no less than about 400 nanoseconds apart. As long as the host attempts to maintain a throughput no greater than this limit, delays due to not-ready signals occur rarely, if at all. The bandwidth for this case is calculated in Table 10-4 *a* as approximately 40 megabits per second. This value can be expected to vary slightly with system-dependent conditions such as the frequency of DRAM-refresh and screen-refresh cycles.

When the TMS34010 is running, the host interface should support continuous accesses occurring at regular intervals no less than approximately 550 nanoseconds. The bandwidth for this case is calculated in Table 10-4 as approximately 29 megabits per second. This value varies slightly with conditions such as the frequency of DRAM-refresh and screen-refresh cycles, and also with the characteristics of the program being executed by the TMS34010.

Table 10-4. Host Interface Estimated Bandwidth

Assumptions	Approximate Throughput
TMS34010 halted 50-MHz TMS34010 No wait states	$\frac{\text{16 bits/transfer}}{\text{400 ns/transfer}}$ = 40 megabits/s
TMS34010 running 50-MHz TMS34010 No wait states	$\frac{\text{16 bits/transfer}}{\text{550 ns/transfer}}$ = 29 megabits/s

10.5 Worst-Case Delay

In some applications, designers must determine not only the effective throughput of the host interface, but also the delays that can occur under worst-case conditions. These conditions occur too rarely to affect overall throughput, but the important consideration here is not how often they occur, but that they can occur at all. First, with the TMS34010 halted, the worst delay is given by the formula $(6 + 2N)T$, where N is the number of wait states per TMS34010 memory cycle, and T is the local clock period (nominally 160 nanoseconds for a 50-MHz TMS34010). Second, with the TMS34010 running, the worst delay is given by the formula $(9 + 4N)T$. The derivation of these formulas, summarized in Figure 10-13, may be helpful in illustrating the mechanisms of the host interface.

$2T$	Synchronization delay
$(2 + N)T$	Screen-refresh cycle
$+ (2 + N)T$	DRAM-refresh cycle
$(6 + 2N)T$	Worst-case delay (total)

(a) Worst-Case Delay with TMS34010 Halted

$2T$	Synchronization delay
$(1 + N)T$	TMS34010 CPU read
$(2 + N)T$	TMS34010 CPU write
$(2 + N)T$	Screen-refresh cycle
$+ (2 + N)T$	DRAM-refresh cycle
$(9 + 4N)T$	Worst-case delay (total)

(b) Worst-Case Delay with TMS34010 Running

N = Number of wait states per memory cycle
T = Local clock period (nominal 160 nanoseconds for 50-MHz device)

Note: These are worst-case delays and have negligible effect on performance. The case shown in *a*, for example, could be expected to occur less than once per thousand (0.1 percent of) host accesses in a typical system.

Figure 10-13. Calculation of Worst-Case Host Interface Delay

Consider case *a*, in which the TMS34010 is halted, first; the worst-case delay is calculated as the sum of the three delays. The first of these delays is the time required to internally synchronize the host interface cycle to the TMS34010 local clock. The host's signals are generally not synchronous to the TMS34010 local clocks. A signal from the host must therefore be passed through a synchronizer latch (part of the TMS34010 on-chip host interface logic) before being used by the TMS34010. The delay through the synchronizer is from one to two local clock periods ($1T$ to $2T$), depending on the phase of the host clock relative to the TMS34010's local clock. The second and third delays in Figure 10-13 represent the time needed to perform a screen-refresh cycle followed by a DRAM-refresh cycle. The arbitration logic internal to the TMS34010 assigns these two types of cycles higher priorities than host-requested indirect accesses. (Screen refresh has a higher priority than DRAM refresh.) Thus, a host access requested at the same time as one of these cycles must wait. The worst-case assumption is that a screen-refresh cycle is generated internal to the TMS34010 on the same clock edge at which the request for the host access arrives. Furthermore, a DRAM-refresh cycle is

requested during this same clock edge or during the next 1 + *N* clock edges. An equivalent delay occurs in the case in which a DRAM refresh and host access are requested on the same clock edge (the DRAM refresh wins), and a screen refresh is requested on a later clock edge before the host access can begin. This case is not shown in Figure 10-13, but the delay in this instance is also $(6 + 2N)T$. In a typical system, DRAM-refresh cycles consume about 2 percent of the available memory bandwidth, and screen-refresh cycles take about 1.5 percent (using VRAMs). The probability of either sequence of events is therefore very small (less than one in a thousand, assuming $N = 0$; that is, no wait states), and the performance degradation due to these unlikely events is negligible.

Now consider the case in which the TMS34010 is running. Host accesses are of higher priority than TMS34010 instruction fetches and data accesses, but still of lower priority than DRAM-refresh or screen-refresh cycles. The worst-case delay is calculated as the sum of the five delays indicated in Figure 10-13 *b*. This assumes that the TMS34010 begins a read-modify-write operation on a memory word (this is performed as a read cycle followed by a separate write cycle) just one clock before the TMS34010 receives the host access request. The TMS34010 CPU read cycle is actually $(2 + N)T$ in duration, but since it begins one clock before the host access is requested, only $(1 + N)T$ is left in the cycle. The TMS34010's local memory controller treats a read-modify-write operation as indivisible; once the read has started, no other request can be granted until the write completes. The write cycle is $(2 + N)T$ in duration. Again, assume that sometime before the write cycle does complete, screen-refresh and DRAM-refresh cycles are also requested. The probability of this case is somewhat more difficult to calculate than that of Figure 10-13 *a*, since the frequency of read-modify-write operations is very program dependent. This sequence of events rarely occurs, however.

Section 11

Local Memory Interface

The TMS34010 local memory interface consists of a triple-multiplexed address/data bus and associated control signals. Several types of memory cycles, including read, write, screen-refresh, and DRAM-refresh cycles are supported. During a memory cycle, the row address, column address, and data are transmitted over the same physical bus lines. The row and column addresses necessary to address DRAMs and VRAMs are available directly at the address/ data pins, eliminating the need for external multiplexing hardware.

The TMS34010 interfaces directly to DRAMs (such as the TMS4256 and TMS4C1024) and VRAMs (such as the TMS4461), and can be programmed to perform DRAM-refresh cycles at regular intervals. $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ or $\overline{\text{RAS}}$-only refresh cycles may be selected. The TMS34010 can also be programmed to perform screen refresh by scheduling VRAM register-transfer cycles to occur at regular intervals.

The local memory interface provides a hold/hold acknowledge capability that allows external devices to request control of the bus. After acknowledging a hold request, the TMS34010 releases the bus by driving its address/data bus and control outputs into high impedance.

Section **Page**

11.1 Local Memory Interface Pins

Section 2 describes TMS34010 pin functions in detail. This section briefly summarizes the local memory interface pins.

LAD0–LAD15
These pins form the local multiplexed address/data bus.

$\overline{\text{DEN}}$ The local data enable signal is driven active low to allow data to be written to or read from LAD0–LAD15. (Connects to the $\overline{\text{G}}$ pins of a pair of optional '245-type octal bus transceivers.)

DDOUT The local data direction out signal is driven high to enable data to be output on LAD0–LAD15. It is driven low to enable data to be input on LAD0–LAD15. (Connects to the DIR pins of a pair of optional '245-type octal bus transceivers.)

$\overline{\text{LAL}}$ The high-to-low transition of the local address latched signal is used by an external '373-type latch to capture the column address from LAD0–LAD15.

$\overline{\text{RAS}}$ The local row address strobe signal drives the $\overline{\text{RAS}}$ inputs of DRAMs and VRAMs.

$\overline{\text{CAS}}$ The local column address strobe signal drives the $\overline{\text{CAS}}$ inputs of DRAMs and VRAMs.

$\overline{\text{W}}$ The local write strobe signal drives the $\overline{\text{W}}$ inputs of DRAMs and VRAMs.

$\overline{\text{TR}}/\overline{\text{QE}}$ The local register transfer/output enable signal connects to the $\overline{\text{TR}}/\overline{\text{QE}}$ (or $\overline{\text{DT}}/\overline{\text{OE}}$) pins of a VRAM.

LRDY The local ready signal is driven low by external circuitry to inhibit the TMS34010 from completing a local memory cycle.

INCLK TMS34010 processor functions are synchronous to this input clock signal. (Video timing is controlled by VCLK.)

LCLK1, LCLK2 These output clocks are available to the board designer for synchronous control of external circuitry.

$\overline{\text{LINT1}}$, $\overline{\text{LINT2}}$ Interrupt requests are transmitted to the TMS34010 on these pins.

11.2 Local Memory Interface Registers

The local memory interface registers are summarized below. These registers are a subset of the I/O registers which are detailed in Section 6.

- The memory **CONTROL** register contains several programmable parameters that provide control of the local memory interface:
  - *RM* (DRAM refresh mode, bit 2): Selects $\overline{\text{RAS}}$-only or $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ refresh cycles.
  - *RR* (DRAM refresh rate, bits 3 and 4): Controls the frequency of DRAM refresh cycles.
  - *T* (transparency enable, bit 5): Enables or disables the pixel attribute of transparency.
  - *W* (window violation detection mode, bits 6 and 7): Selects the course of action the TMS34010 follows when it detects a window violation.
  - *PBH* (PIXBLT horizontal direction, bit 8): Determines the horizontal direction (increasing X or decreasing X) for pixel operations.
  - *PBV* (PIXBLT vertical direction, bit 9): Determines the vertical direction (increasing Y or decreasing Y) for pixel operations.
  - *PPOP* (pixel processing operation select, bits 10–14): Selects among several Boolean and arithmetic pixel processing options.
  - *CD* (instruction cache disable, bit 15): Enables or disables the instruction cache.
- The **CONVDP** register contains the destination pitch conversion factor that is used during XY-to-linear conversion of a destination pixel address.
- The **CONVSP** register contains the source pitch conversion factor that is used during XY-to-linear conversion of a source pixel address.
- The **PMASK** (plane mask) register selectively disables or enables various planes in a multiple-bit-per-pixel bit map display.
- The **PSIZE** (pixel size) register specifies the number of bits per pixel.
- The **REFCNT** (refresh count) register generates the addresses output during DRAM-refresh cycles and counts the intervals between successive DRAM-refresh cycles.

11.3 Memory Bus Request Priorities

The TMS34010's local memory interface controller assigns priorities to requests from various sources, both on and off chip, for local memory cycles. Table 11-1 lists these priorities (priority 1 is highest).

Table 11-1. Priorities for Memory Cycle Requests

Priority	Memory Cycle Requested
1	Hold request from external bus master device
2	Screen-refresh cycle
3	DRAM-refresh cycle
4	Host-initiated indirect read or write cycle
5	TMS34010 CPU memory cycle

A TMS34010 CPU memory cycle is a read or write performed by the TMS34010's on-chip 32-bit processor. Insertion of a field (or a portion of a field spanning multiple words) into a word requires *two* CPU memory cycles when the field does not begin and end on word boundaries. The two cycles are a read followed by a write. This sequence is called a read-modify-write operation. The read and write are performed as separate memory cycles, but are treated as indivisible; that is, the memory controller does not permit another memory request to be serviced between the read and its accompanying write. The only exception to this statement is the hold request. If a read-modify-write is interrupted by a hold, the entire read-modify-write operation is restarted after the hold is released.

While a read-modify-write operation on an individual memory word is indivisible, the accesses necessary to extract or insert a field spanning multiple memory words are not. For example, if a field spans portions of two memory words, a higher priority access such as a host-indirect cycle can occur between the two read-modify-write operations required to insert the field.

The hold request has the highest priority. An external device requests control of the bus by signalling a hold request to the TMS34010. The external device may perform multiple memory cycles following acknowledgment from the TMS34010. However, the device should not control the bus for so long that necessary screen-refresh and DRAM-refresh cycles are prevented from occurring. Indirect accesses initiated by a host processor are blocked as long as the external device continues to control the bus. If the host processor attempts to initiate another indirect access during this time, the host is forced to wait at the host interface (the TMS34010 sends it a not-ready signal) until the external device releases the local bus.

A memory cycle already in progress is always permitted to complete, even if a higher priority request is received while the cycle is still in progress.

11.4 Local Memory Interface Timing

The TMS34010 memory interface contains a triple-multiplexed address/data bus on which row addresses, column addresses and data are transmitted. Figure 11-1 illustrates multiplexing of addresses and data.

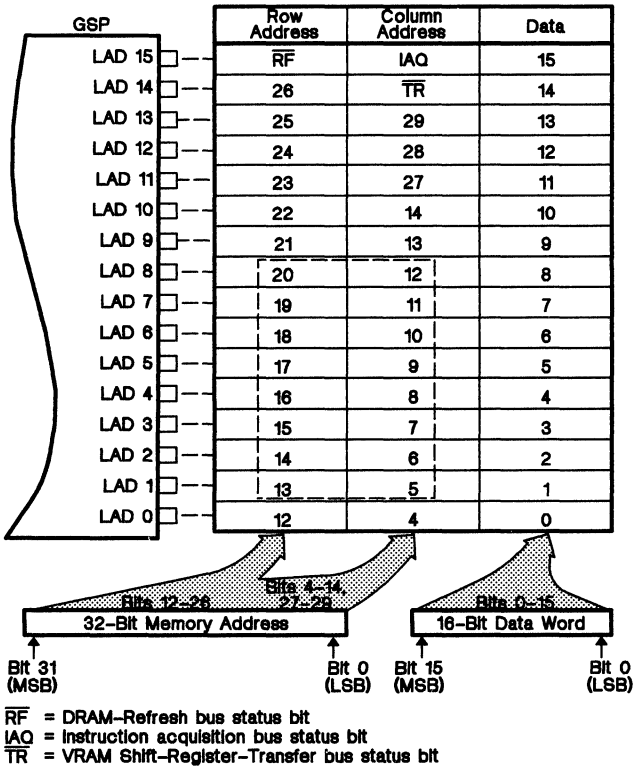


Figure 11-1. Triple Multiplexing of Addresses and Data

The TMS34010 LAD pins directly provide the multiplexed row and column addresses needed to drive dynamic RAMs (like the TMS4256) and video RAMs (such as the TMS4461). Any eight adjacent pins in the range LAD0–LAD10 provide 16 contiguous logical address bits; the eight MSBs are output as part of the row address, and the eight LSBs are output as part of the column address. For example, Figure 11-1 shows that logical address bits 5–20 are output at LAD1–LAD8.

The control signals output to memory support direct interfacing to DRAMs and VRAMs. At the beginning of a memory cycle, the address is output in multiplexed fashion as a row address followed by a column address. The remainder of the cycle is used to transfer data between the TMS34010 and memory. Figure 11-2 (page 11-6) illustrates general timing (the local address/data pins are identified as the *LAD Bus*)

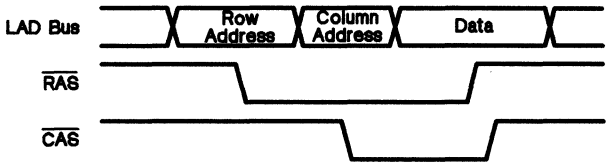


Figure 11-2. Row and Column Address Phases of Memory Cycle

Figure 11-3 through Figure 11-8 show functional timing of the local memory interface. Several timing features are common to the memory read and write cycles in Figure 11-3 and Figure 11-4, and to the register-transfer cycles in Figure 11-6 and Figure 11-7. A row address is output on LAD0–LAD15 at the start of the cycle, and is valid before and after $\overline{RAS}$ falls. A column address is then output on LAD0–LAD15. The column address is valid briefly before and after the falling edge of $\overline{LAL}$, but is not valid at the falling edge of $\overline{CAS}$. The column address is clocked into an external transparent latch (such as a 74AS373 octal latch) on the falling edge of $\overline{LAL}$ to provide the hold time on the column address required for DRAMs and VRAMs. A transparent latch is required so that the row address is available at the outputs of the latch during the start of the cycle.

11.4.1 Local Memory Write Cycle Timing

Figure 11-3 illustrates a memory write cycle. Data are output on LAD0–LAD15 following the latching of the column address. $\overline{DEN}$ goes active low at the same time the data become valid, and remains low as long as the data remain valid. In a large system that requires buffering of the data bus to memory, $\overline{DEN}$ is typically used as the enable signal to an external bidirectional buffer (such as a 74AS245 octal buffer). DDOUT is used as the direction control signal to the buffer. The write strobe, $\overline{W}$, goes active low after the data have become valid and $\overline{CAS}$ is low. This is interpreted as a "late write" cycle by the DRAMs and VRAMs, which are prevented by the inactive-high $\overline{TR/QE}$ signal from enabling their read drivers. Because the data are valid on both sides of the $\overline{W}$ write strobe, external devices can latch the data on either the high-to-low or low-to-high edge of $\overline{W}$.

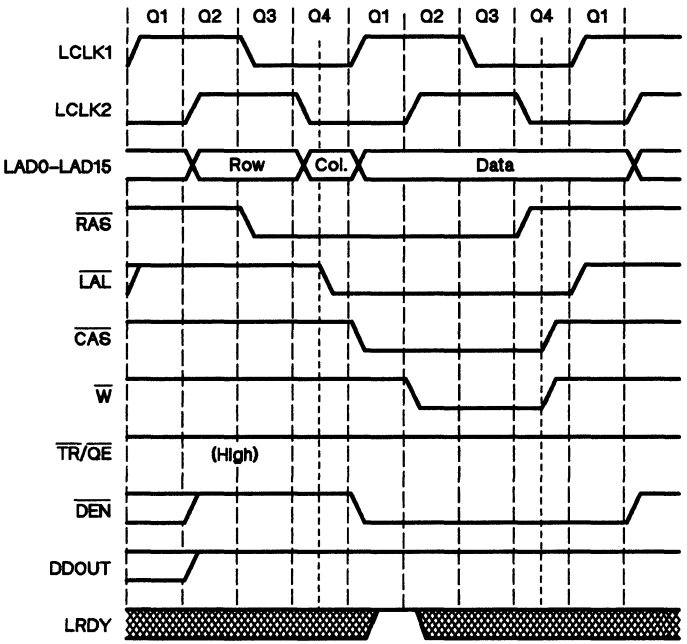


Figure 11-3. Local Bus Write Cycle Timing

11.4.2 Local Memory Read Cycle Timing

Figure 11-4 illustrates a memory read cycle. LAD0–LAD15 are forced to high impedance following the latching of the column address. $\overline{DEN}$ and $\overline{TR/QE}$ both go active low after $\overline{CAS}$ becomes low in order to enable read data from the memory to the LAD pins. $\overline{TR/QE}$ enables the output drivers of the DRAMs and VRAMs. $\overline{DEN}$ enables the external bidirectional buffers needed with memories so large that external buffering (using a device such as a 74AS245 octal buffer) of the data bus is required. The DDOUT signal serves as the direction control for the external bidirectional buffers, and is low well in advance of the high-to-low transition of $\overline{DEN}$, and remains low well after the low-to-high transition of $\overline{DEN}$. The data that is read from memory must be valid during the middle of the Q4 clock phase, as indicated in Figure 11-4. The low-to-high transitions of $\overline{TR/QE}$ and $\overline{DEN}$ occur well in advance of the time at which the LAD drivers turn on to output the row address of the next cycle. This prevents bus conflicts.

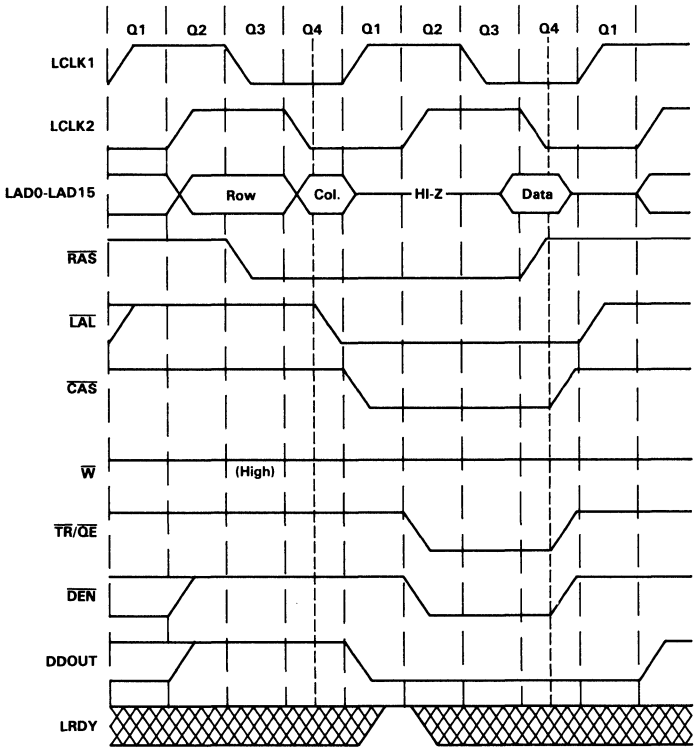


Figure 11-4. Local Bus Read Cycle Timing

11.4.3 Local Register-to-Memory Cycle Timing

A register-to-memory cycle is a special type of cycle used in systems with VRAMs. The cycle transfers the contents of the VRAM's internal serial-data register to a selected row of its internal memory array. The cycle typically affects all VRAMs in the system. During the register-to-memory cycle shown in Figure 11-5, both $\overline{TR}/\overline{QE}$ and $\overline{W}$ are low during the fall of $\overline{RAS}$. VRAMs recognize this timing as the beginning of a register-to-memory cycle. Conventional DRAMs may need to be de-selected (by withholding the row or column address strobe, for example) to prevent them from interpreting the cycle as a conventional read cycle. Alternately, the output enable signal required by a DRAM such as the TMS4464 can be synthesized by connecting $\overline{DEN}$ and DDOUT to the inputs of a two-input OR gate. (In fact, any pair of the signals $\overline{DEN}$, DDOUT, and $\overline{TR}/\overline{QE}$ will work.) The low-to-high transition of $\overline{TR}/\overline{QE}$ occurs after the fall of $\overline{CAS}$ but prior to the rising edge of $\overline{RAS}$. This timing provides compatibility with a variety of VRAMs.

The TMS34010 performs a register-to-memory cycle when writing to a pixel while the DPYCTL register's SRT bit is set to 1. For example, the instruction `PIXT A0,*A1` writes the pixel in A0 to the address pointed to by A1. The PSIZE register should contain the value 16 so that the write cycle is not preceded by a read cycle. When SRT is set to 1, this write is converted to the register-to-memory cycle shown in Figure 11-5. The row address is selected from bits 12–26 of A1, which are output on LAD0–LAD14 during the cycle.

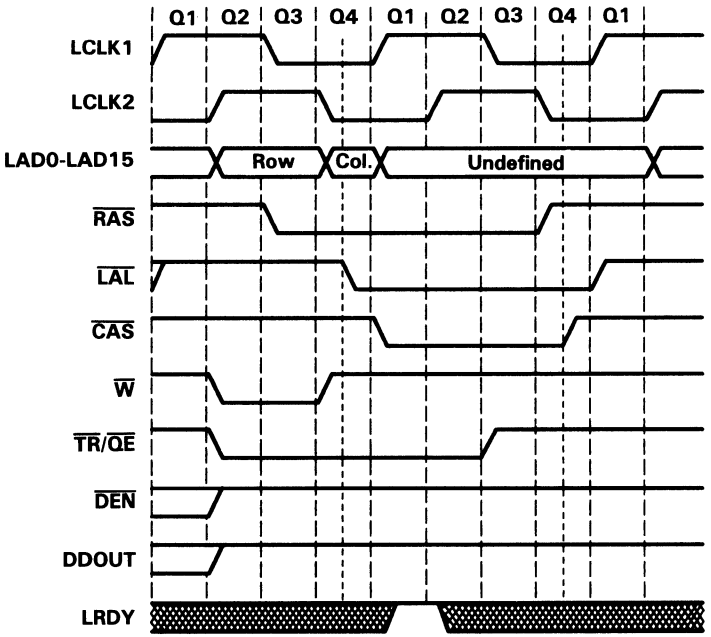


Figure 11-5. Local Bus Register-to-Memory Cycle Timing

11.4.4 Local Memory-to-Register Cycle Timing

A memory-to-register cycle is a special type of cycle used in systems with VRAMs. The cycle transfers the contents of a selected row of a video RAM's memory array to its internal shift register.

VRAM memory-to-register cycles are primarily used to refresh the screen of a CRT monitor. The cycles referred to elsewhere in this document as *screen-refresh cycles* are in fact memory-to-register cycles. The TMS34010 also performs a memory-to-register cycle when reading a pixel (for example, by executing a `PIXT *A0,A1` instruction) while the SRT bit of the DPYCTL register is set to 1.

During the memory-to-register cycle shown in Figure 11-6, $\overline{TR}/\overline{QE}$ is low during the fall of $\overline{RAS}$, but $\overline{W}$ remains high. VRAMs recognize this timing as the beginning of a memory-to-register cycle, and their data outputs remain in high impedance. Conventional DRAMs may need to be de-selected to prevent them from interpreting the cycle as a memory read cycle. Alternately, the output enable signal required by a DRAM such as the TMS4464 can be synthesized by connecting $\overline{DEN}$ and DDOUT to the inputs of a two-input OR gate. The low-to-high transition of $\overline{TR}/\overline{QE}$ occurs after the fall of $\overline{CAS}$ but prior to the rising edge of $\overline{RAS}$. This timing provides compatibility with a variety of VRAMs.

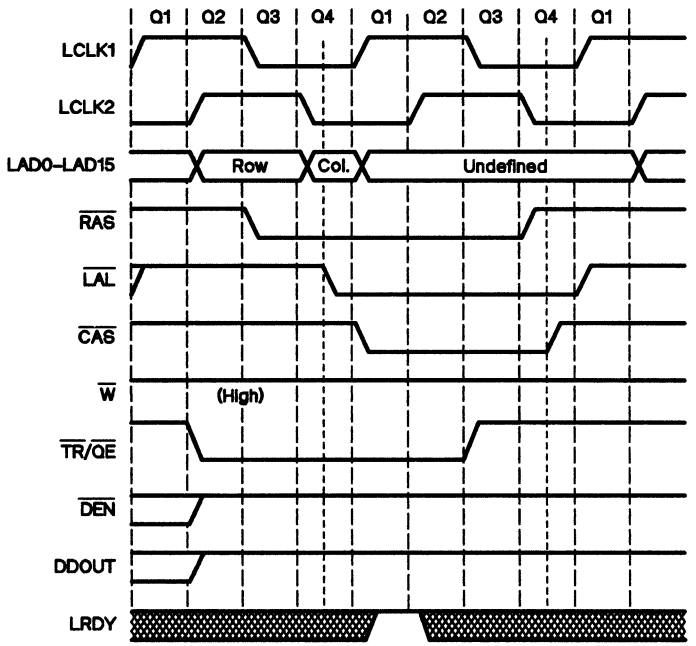


Figure 11-6. Local Bus Memory-to-Register Cycle Timing

11.4.5 Local Memory RAS-Only DRAM Refresh Cycle Timing

During the $\overline{RAS}$-only DRAM refresh cycle shown in Figure 11-7, $\overline{RAS}$ and $\overline{LAL}$ are the only active control signals. All other signals, including $\overline{CAS}$, $\overline{W}$, and $\overline{TR}/\overline{QE}$, remain inactive high through the cycle. The row address, output on the LAD pins during the high-to-low transition of $\overline{RAS}$, is generated by the REFCNT (DRAM-refresh counter) register.

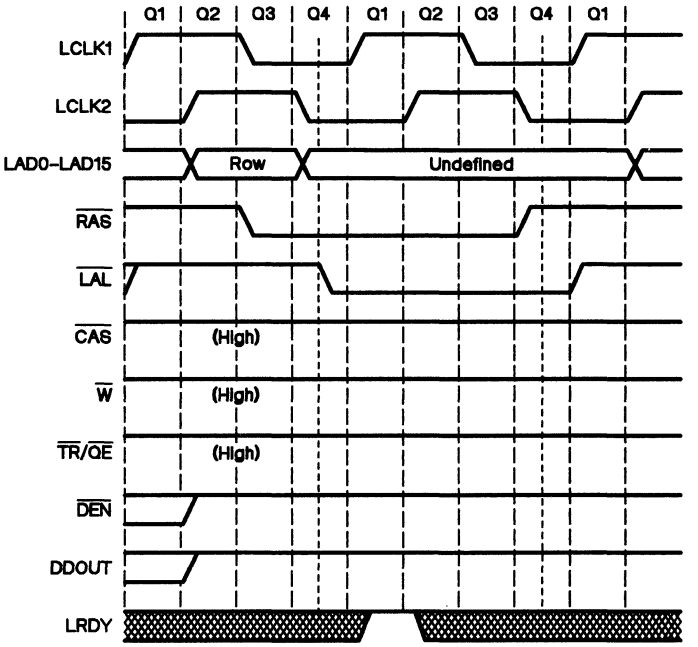


Figure 11-7. Local Bus $\overline{RAS}$-Only DRAM-Refresh Cycle Timing

11.4.6 Local Memory $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ DRAM Refresh Cycle Timing

During the $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ DRAM-refresh cycle shown in Figure 11-8, $\overline{\text{CAS}}$ goes low before $\overline{\text{RAS}}$ goes low. Certain types of DRAMs (like the TMS4256 and TMS4C1024) and VRAMs (such as the TMS4461) recognize this as the beginning of a DRAM-refresh cycle in which the address of the row to be refreshed is generated by a counter on the RAM chip itself, rather than by an external device such as the TMS34010. The row address output by the TMS34010 during the cycle is the same as would be output if the TMS34010 were configured to perform a $\overline{\text{RAS}}$-only refresh cycle rather than a $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ cycle. The address bits output on LAD0–LAD15 remain stable from the start of the row address time (start of Q2) to the end of the column address time (end of Q4). LAD15, on which the $\overline{\text{RF}}$ bus status bit is output, is low during the row address times. In contrast to other types of cycles in which $\overline{\text{RAS}}$ goes low, the $\overline{\text{LAL}}$ output goes low at the start of Q3, after the fall of $\overline{\text{CAS}}$ and before the fall of $\overline{\text{RAS}}$. The timing of $\overline{\text{LAL}}$ is designed to support the use of decode circuitry which latches the state of selected address/data pins during the fall of $\overline{\text{LAL}}$, and which recognizes a $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ cycle by detecting a high level at the $\overline{\text{RAS}}$ output during the fall of $\overline{\text{LAL}}$.

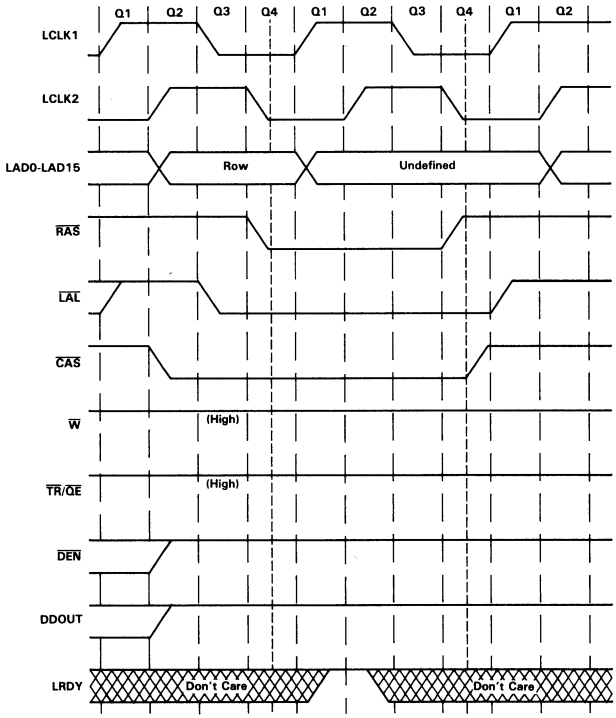


Figure 11-8. Local Bus $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ DRAM-Refresh Cycle Timing

11.4.7 Local Memory Internal Cycles

When the TMS34010 is not performing one of the memory operations shown in Figure 11-3 through Figure 11-8, its memory interface control signals remain inactive, as shown in Figure 11-9. This is called an internal cycle. Figure 11-9 shows two sequential internal cycles. During internal cycles, the LRDY input is ignored.

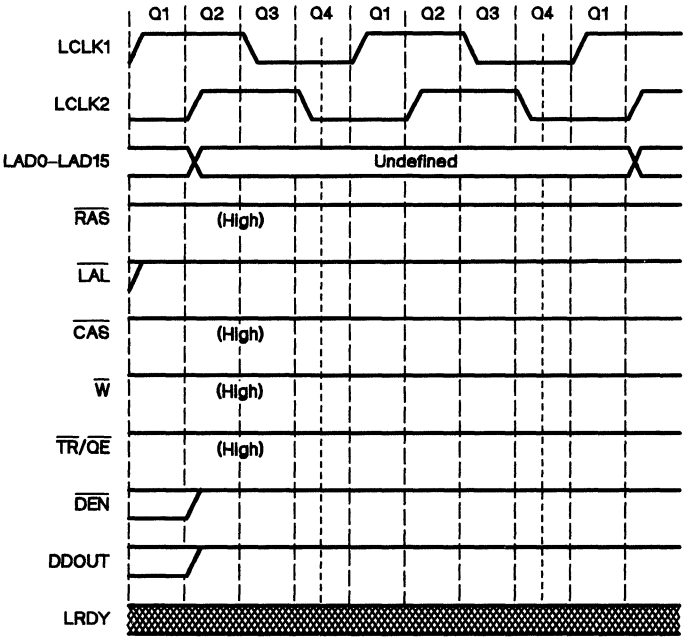


Figure 11-9. Local Bus Internal Cycles Back to Back

11.4.8 I/O Register Access Cycles

A special memory read or write cycle is performed when the TMS34010 addresses an on-chip I/O register. During this cycle, the external $\overline{RAS}$ signal falls, but the external $\overline{CAS}$ remains inactive high for the duration of the cycle. I/O register locations begin at address C0000000h, and all 32 bits of the I/O register address are decoded internally. The two MSBs of the 32-bit logical address are not output at the LAD0–LAD15 pins.

Figure 11-10 shows an I/O register read cycle and Figure 11-11 shows an I/O register write cycle. These cycles occur when one of the TMS34010's on-chip I/O registers is accessed by the on-chip processor or by the host processor via a host-indirect access. An address in the range C0000000h to C00001FFh is interpreted as an I/O register access by on-chip decode logic, and the read or write cycle is modified as shown in Figure 11-10 or Figure 11-11. The two

MSBs of the internal address (bits 30 and 31) are available internally and are included in the internal decoding operation.

An I/O register read or write cycle is always two clock periods in duration, and LRDY is ignored. The only control outputs that are active low during the cycle are $\overline{RAS}$ and $\overline{LAL}$. The $\overline{CAS}$, $\overline{W}$, $\overline{TR}/\overline{QE}$, $\overline{DEN}$ and DDOUT outputs all remain inactive high. The row and column addresses output at the LAD0–LAD15 pins are all 0s. All three bus status bits are inactive ($\overline{RF}$ is high, IAQ is low, and $\overline{TR}$ is high). During the read cycle shown in Figure 11-10, the LAD0–LAD15 pins are driven to high impedance during the data phase of the cycle. During the write cycle shown in Figure 11-11, the LAD0–LAD15 pins contain the valid data being written to the I/O register.

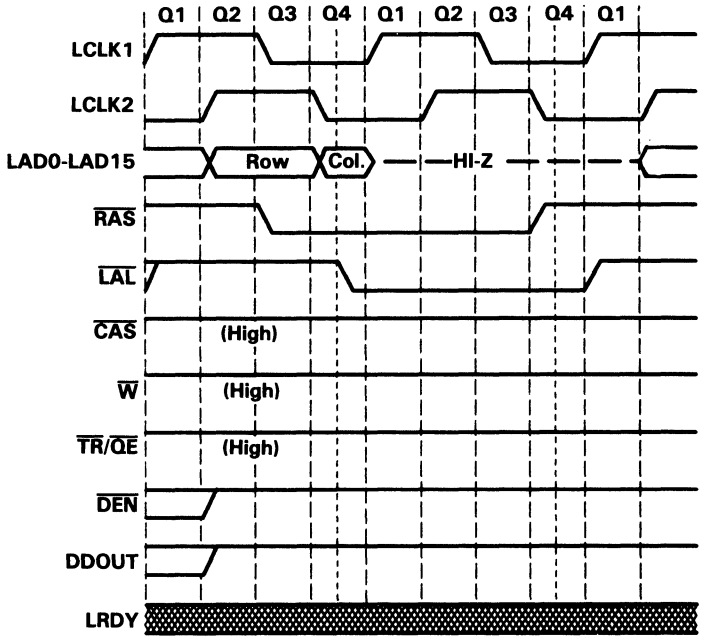


Figure 11-10. I/O Register Read Cycle Timing

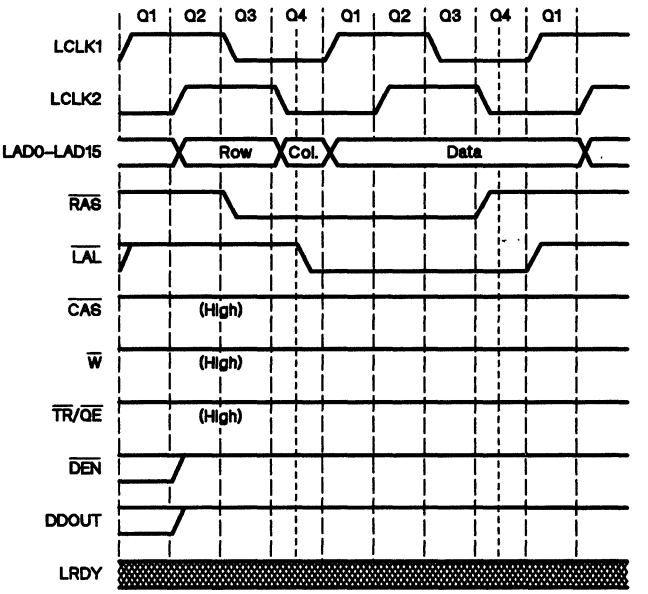


Figure 11-11. I/O Register Write Cycle Timing

11.4.9 Read-Modify-Write Operations

The TMS34010's read-modify-write operation, which consists of separate read and write cycles, is not the same as the read-modify-write cycle specified for some DRAMs. As explained in Section 5, when inserting a field into memory that is not aligned to 16-bit word boundaries, the TMS34010 memory interface logic may be required to perform read-modify-write (RMW) operations on one or more words in memory. A RMW operation consists of the following sequence of steps:

1) A word is read from memory.
2) The portion of that word corresponding to the field being inserted is loaded with the new value.
3) The modified word is written back to memory.

The read cycle is as shown in Figure 11-4 (page 11-8), and the write cycle is as shown in Figure 11-3 (page 11-7).

A local bus request ($\overline{\text{HOLD}}$ low) may cause the TMS34010 to release the bus after the read cycle of a RMW operation has completed, but before the accompanying write cycle has begun. When the TMS34010 later regains control of the bus, it performs both the read and the write cycles of the RMW operation. The RMW operation is performed only when it is the highest priority bus operation pending. Any pending screen-refresh, DRAM-refresh, or host-indirect cycle has higher priority, and is performed first.

11.4.10 Local Memory Wait States

The timing shown in Figure 11-3 through Figure 11-8 assumes that the LRDY input remains high during the cycle. The LRDY pin is pulled low by slower memories requiring a longer cycle time. The TMS34010 samples the LRDY input at the end of Q1, as indicated in the figures. If LRDY is low, the TMS34010 inserts an additional state, called a **wait state**, into the cycle. Wait states continue to be inserted until LRDY is sampled at a high level. The cycle then completes in the manner indicated in Figure 11-3 through Figure 11-8.

The LRDY input is ignored by the TMS34010 during internal cycles, as indicated in Figure 11-9.

Figure 11-12 shows an example of a read cycle extended by one wait state. The first time LRDY signal is sampled, a low level is detected by the TMS34010, causing the cycle to be delayed by a wait state. When LRDY is sampled again one local clock period later, a high level is detected, permitting the cycle to complete. The time during which $\overline{RAS}$, $\overline{CAS}$, $\overline{LAL}$, $\overline{TR/QE}$, $\overline{DEN}$, and DDOUT remain low is extended by one state (one local bus clock period).

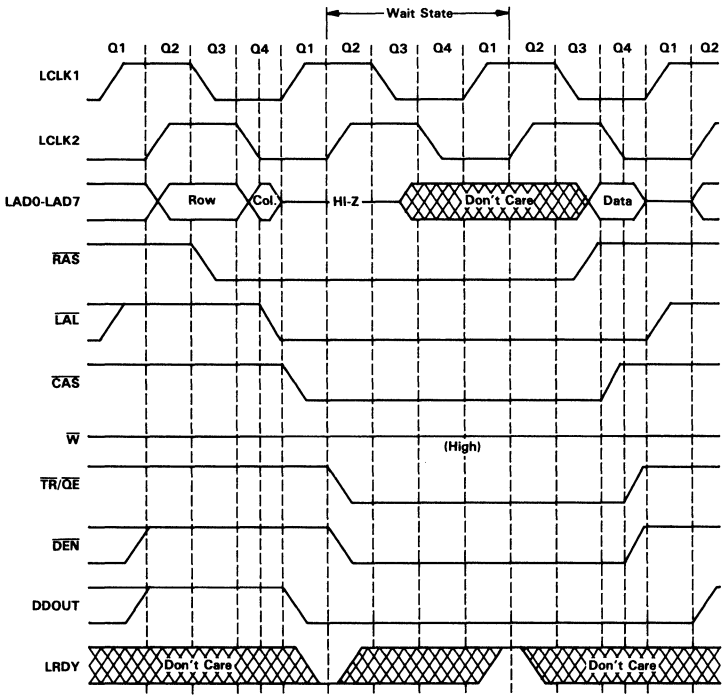


Figure 11-12. Local Bus Read Cycle with One Wait State

Figure 11-13 is an example of a write cycle extended by one wait state. The first time LRDY signal is sampled, a low level is detected by the TMS34010, causing the cycle to be delayed by a wait state. When LRDY is sampled again one local clock period later, a high level is detected, permitting the cycle to complete. The time during which $\overline{RAS}$, $\overline{CAS}$, $\overline{LAL}$, $\overline{W}$ and $\overline{DEN}$ remain low is extended by one state.

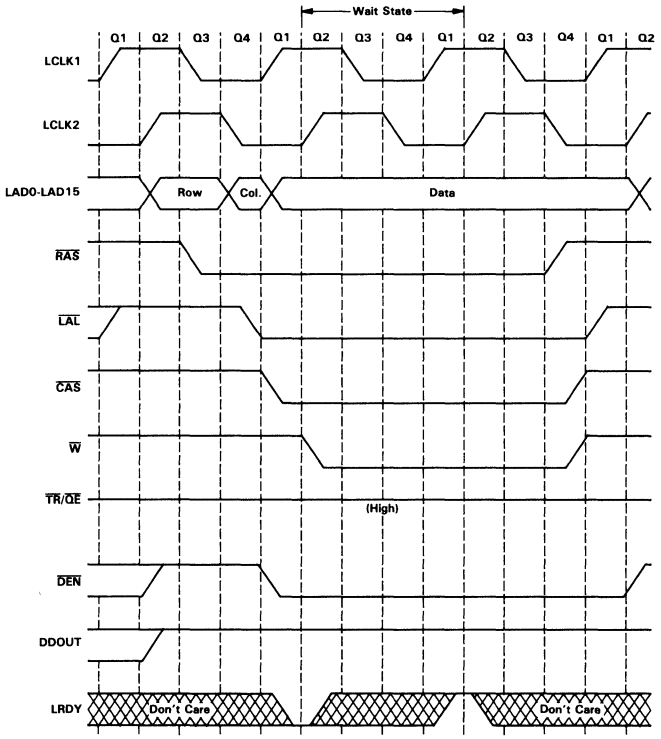


Figure 11-13. Local Bus Write Cycle with One Wait State

Figure 11-14 is an example of a register-to-memory cycle extended by one wait state. The first time the LRDY signal is sampled, a low level is detected by the TMS34010, causing the cycle to be delayed by a wait state. When LRDY is sampled again one local clock period later, a high level is detected, permitting the cycle to complete. The time during which $\overline{RAS}$, $\overline{CAS}$, and $\overline{LAL}$ remain low is extended by one state. The $\overline{W}$ and $\overline{TR}/\overline{QE}$ low times are not extended, however. Similarly, during a memory-to-register cycle, $\overline{TR}/\overline{QE}$ is not extended.

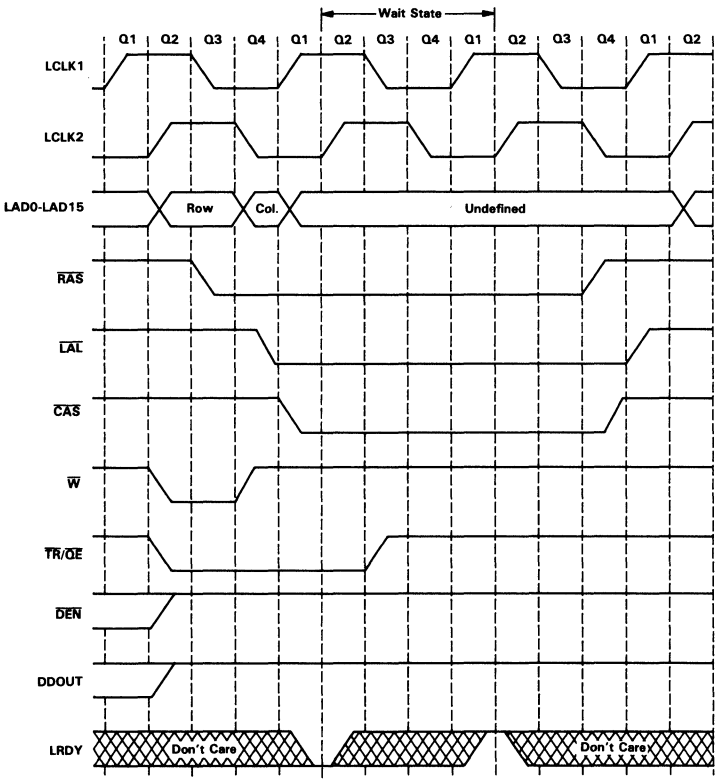


Figure 11-14. Local Bus Register-to-Memory Cycle with One Wait State

11.4.11 Hold Interface Timing

The TMS34010 includes a hold interface through which external bus-master devices can request control of the local memory bus. When the TMS34010 grants a hold request, it drives its local memory address/data bus and control outputs to high impedance, and the requesting device becomes the new bus master. When the requesting device no longer requires the bus, it removes its hold request, and the TMS34010 again assumes control of the local bus.

Figure 11-15 shows the TMS34010 releasing control of the local bus in response to a hold request. The TMS34010 samples the state of the $\overline{\text{HOLD}}$ input at each LCLK2 rising edge (at the end of the Q1 phase of the clock). $\overline{\text{HOLD}}$ is a synchronous input, and must not change during the time that the TMS34010 samples it; refer to the *TMS34010 Data Sheet* for $\overline{\text{HOLD}}$ setup and hold times. The state of the hold acknowledge signal (active or inactive) is output on the $\overline{\text{HLDA}}$/$\overline{\text{EMUA}}$ pin during the Q3 and Q4 clock phases (LCLK1 low). During the first (or leftmost) LCLK2 rising edge, the hold request is inactive. Consequently, the hold acknowledge signal remains inactive during

the first LCLK1 low phase. By the second LCLK2 rising edge, the hold request has been activated, and the TMS34010 responds by acknowledging the hold request during the next LCLK1 low phase. The address/data lines and majority of the control lines are driven to high impedance at the start of the next Q2 phase (LCLK2 rising edge). The DDOUT and $\overline{DEN}$ pins are driven to high impedance a quarter clock later.

Figure 11-16 shows the TMS34010 resuming control of the local bus after deactivation of the hold request. Again, the TMS34010 samples the state of the $\overline{HOLD}$ input at each LCLK2 rising edge. During the first LCLK2 rising edge, the hold request is still active, and the TMS34010 responds during the next LCLK1 low phase with an active hold acknowledge signal. By the second LCLK2 rising edge, the hold request has been removed. The TMS34010 responds by outputting an inactive hold acknowledge signal during the next LCLK1 low phase. At the next LCLK2 rising edge, the TMS34010 begins to drive its address/data pins and the majority of its control pins to logic-high or logic-low levels. The $\overline{DEN}$ and DDOUT signals remain in high impedance for one additional quarter clock before they too begin to be driven.

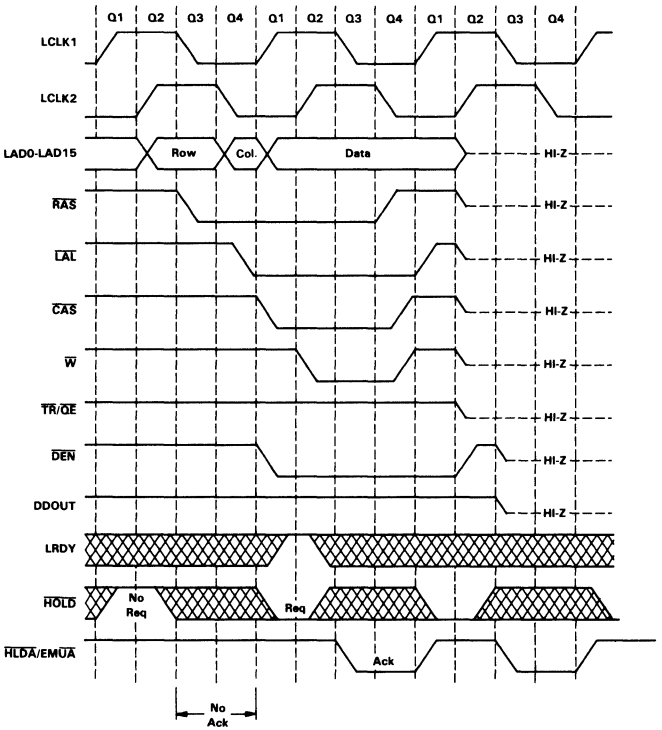


Figure 11-15. TMS34010 Releases Control of Local Bus

In Figure 11-15, the first active-low pulse of the $\overline{HLDA}/\overline{EMUA}$ output is an early acknowledgment, and the bus is not released for another three quarters

of a clock. The early acknowledgment gives advance warning to the device requesting the hold that the bus is about to be released by the TMS34010, allowing the device time to prepare to become the new bus master. The TMS34010 outputs the active hold acknowledge signal only when it is prepared to release the bus within the next clock period. If the TMS34010 must wait longer than this to release the bus, its hold acknowledgment is withheld until it can release the bus.

For instance, if the LRDY signal in Figure 11-15 were low instead of high at the second rising edge of LCLK2, the TMS34010 would be forced to wait, and would therefore not acknowledge the hold request until later, when the not-ready condition was removed. Also, if the hold request in Figure 11-15 was asserted initially during the first LCLK2 rising edge rather than the second, the TMS34010 would delay its hold acknowledgment until the second LCLK1 low clock phase, knowing that the cycle in progress would not be completed until the third Q2 phase in the diagram.

A hold request has a higher priority than any internally generated memory cycle requests, including:

- Screen refresh
- DRAM refresh
- Indirect access by the host processor
- TMS34010 instruction fetch or data access

A hold request is delayed only to allow a memory cycle already in progress to complete.

External devices can activate or deactivate the $\overline{\text{HOLD}}$ input during any clock of an ongoing cycle, as long as the input is stable during the rising edge of LCLK2. The $\overline{\text{HOLD}}$ input is synchronous and is required to meet specified setup and hold times to ensure that the TMS34010 operates correctly. After the TMS34010 grants the bus to an external device (via an active-low level on the $\overline{\text{HLDA}}$/$\overline{\text{EMUA}}$ output during the Q3 clock phase), it continues to acknowledge the hold request during the Q3 phases of subsequent clock cycles. The external device retains control of the bus until it deactivates its hold request.

External devices should avoid placing the TMS34010 in hold for long periods. While the TMS34010 is in hold, it can perform neither screen-refresh nor DRAM-refresh cycles. Furthermore, a host processor attempting to access the TMS34010's local memory through the host interface registers while the TMS34010 is in hold may receive a not-ready signal. When this occurs, the host is forced to wait to complete its access until the TMS34010 leaves the hold state. (Refer to Section 9.10.1.5, Scheduling Screen-Refresh Cycles, on page 9-27 for more information.)

If a request for a DRAM-refresh or screen-refresh cycle is generated within the TMS34010 while an external device controls the bus, the TMS34010 retains the request and perform the DRAM-refresh or screen-refresh cycle after the external device has returned control of the bus to the TMS34010. However, if a requested DRAM-refresh cycle is prevented from occurring for so long that a second DRAM-refresh cycle is requested before the first DRAM-refresh cycle can occur, the first DRAM-refresh request is lost. Similarly, if a screen-refresh request is prevented from occurring for so long that a second

screen-refresh cycle is requested before the first screen-refresh cycle can occur, the first screen-refresh request is lost.

The $\overline{\text{HLDA}}/\overline{\text{EMUA}}$ output is multiplexed between the hold acknowledge ($\overline{\text{HLDA}}$) and emulate acknowledge ($\overline{\text{EMUA}}$) signals. The $\overline{\text{HLDA}}$ signal is output during the LCLK1 low phase, and the $\overline{\text{EMUA}}$ signal is output during the LCLK1 high phase.

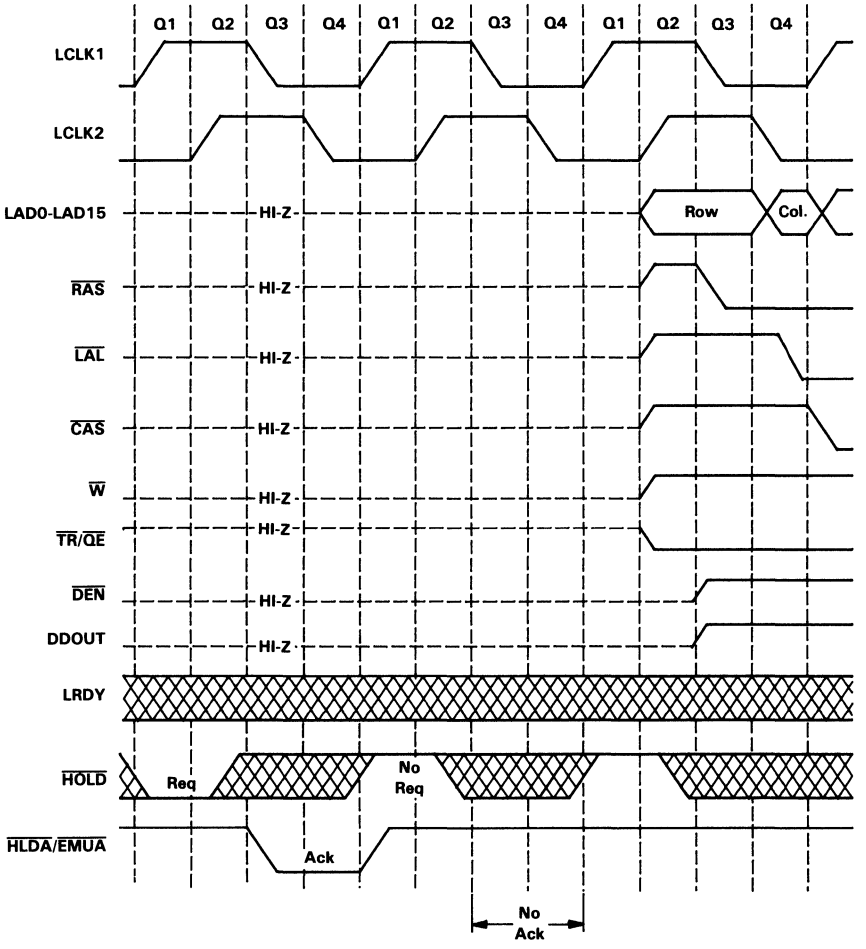


Figure 11-16. TMS34010 Resumes Control of Local Bus

11.4.12 Local Bus Timing Following Reset

Figure 11-17 shows the timing of the local bus signals following reset. At the end of reset, the TMS34010 automatically performs a series of eight $\overline{RAS}$-only refresh cycles, as required to initialize certain DRAMs (such as the TMS4256 and TMS4464) and VRAMs (such as the TMS4461) following power-up. The asynchronous low-to-high transition of $\overline{RESET}$ is sampled at the second high-to-low LCLK1 transition in Figure 11-17. In less than two local clock periods following this LCLK1 transition, the first of the eight $\overline{RAS}$-only cycles begins, as shown at the right side of Figure 11-17.

Each of the eight $\overline{RAS}$ cycles following reset is two clock periods in duration, but can be extended by a not-ready signal (LRDY low). The timing for each cycle is identical to that of a $\overline{RAS}$-only DRAM-refresh cycle, including the bus status codes output during the row and column address times. The row address for each of the eight $\overline{RAS}$-only cycles is all 0s.

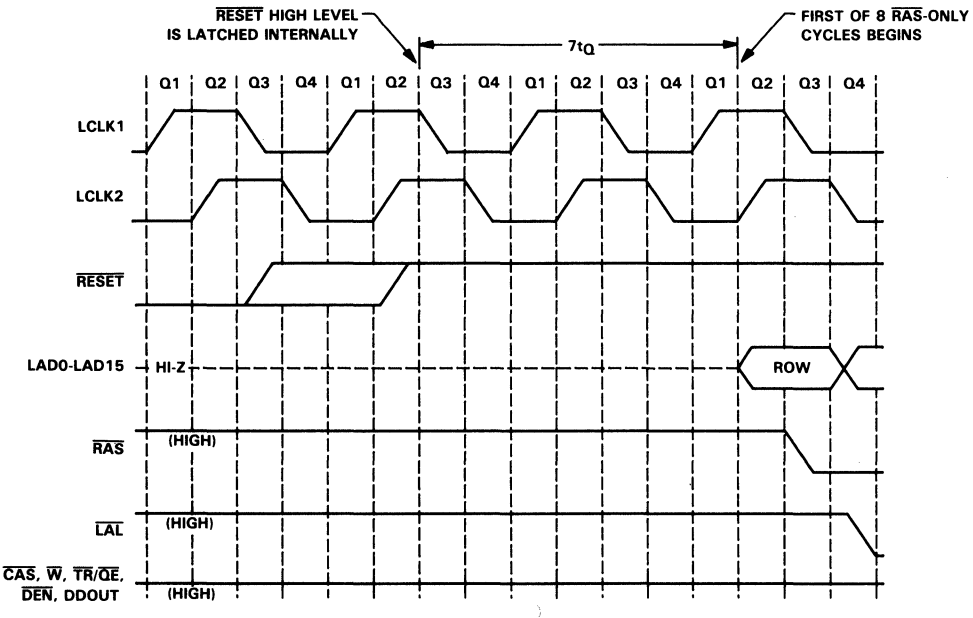


Figure 11-17. Local Bus Timing Following Reset

11.5 Addressing Mechanisms

The TMS34010 addresses memory by means of a 32-bit logical address. As explained in Section 3, each 32-bit logical address points to a bit in memory.

Logical address bits are numbered from 0 to 31, where bit 0 is the LSB and bit 31 is the MSB. Figure 11-18 illustrates the manner in which address bits 4–29 are output to physical memory. Each column in the figure indicates an address/data bus pin, LAD0–LAD15, and below it is the corresponding bit of the logical address output at the LAD pin during the fall of $\overline{RAS}$ and during the fall of $\overline{CAS}$. Bus status bits $\overline{RF}$, $\overline{TR}$, and IAQ are output on LAD14–LAD15.

		LAD Pin Numbers															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TMS34010 Logical Address Bits†	At Fall of $\overline{RAS}$	$\overline{RF}$	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12
	At Fall of $\overline{CAS}$	IAQ	$\overline{TR}$	29	28	27	14	13	12	11	10	9	8	7	6	5	4

† Bus status signals:
$\overline{RF}$ – DRAM refresh cycle
IAQ – Instruction acquisition cycle
$\overline{TR}$ – Register-transfer cycle

Figure 11-18. External Address Format

Key features of the local bus addressing mechanism include the following:

- The two MSBs of the 32-bit logical address (bits 30 and 31) are not output.
- The four LSBs of the 32-bit logical address (bits 0 to 3) are not output, but are used internally to designate a bit boundary within a 16-bit word accessed in the external memory.
- The address bits output on LAD0–LAD10 during the falling edges of $\overline{RAS}$ and $\overline{CAS}$ are aligned so that 16 consecutive bits from the logical address are available at any eight consecutive pins in the range LAD0 to LAD10. The address bits are output in this way in order that the 8-bit row address and 8-bit column address presented to the dynamic RAMs can always be taken from the same eight address/data pins. This eliminates the need for external address multiplexers.
- Logical address bits 12–14 are output twice during a memory cycle – during both the $\overline{RAS}$ and $\overline{CAS}$ falling edges – but at different pins. This allows a variety of memory organizations and decoding schemes to be used.

Pins LAD0–LAD10 form an 11-bit zone in which logical address bits 12–14 are overlapped (that is, they are issued in both cycles, but on different pins). The row and column address bus is connected to any eight consecutive pins within this zone. The actual position is determined by the bank-decoding scheme selected for a particular memory organization.

Output along with the address are three bus status signals:

- The $\overline{RF}$ (DRAM refresh) bit is output on LAD15 during the fall of $\overline{RAS}$. It is low if the cycle that is just beginning is a DRAM-refresh cycle (either $\overline{RAS}$-only or $\overline{CAS}$-before-$\overline{RAS}$); otherwise, $\overline{RF}$ is high.

- The $\overline{TR}$ (VRAM register transfer) bit is output on LAD14 during the fall of $\overline{CAS}$, and is low if the cycle in progress is a video RAM register transfer. Otherwise, $\overline{TR}$ is high. In either event, the state of the $\overline{TR}$ bit reflects the state of the $\overline{TR}/\overline{QE}$ output during the falling edge of $\overline{RAS}$ within the same cycle.

- The IAQ bit is output on LAD15 during the fall of $\overline{CAS}$, and is high if the cycle is an instruction fetch; otherwise, IAQ remains low. The term *instruction fetch* includes not only reads of opcodes, but also immediate data, immediate addresses, and so on.

IAQ is active high when words are fetched from memory to load the instruction cache. A cache subsegment (a block of four words) is loaded in a series of read cycles, during which IAQ is active high. The PC points to an instruction word within the block, but the block may contain data as well as instruction words (opcodes, immediate addresses, immediate data, and so on). Only during execution can the TMS34010 distinguish instruction words from data words residing in the cache. Instruction words are fetched from the cache as they are needed, but data inadvertently loaded into the cache is ignored and all memory data reads or writes result in accesses of the memory rather than the cache.

When the cache is disabled, IAQ is active high only when the first word of an instruction is fetched; in the case of a multiple-word instruction, IAQ is inactive while the additional words are fetched.

11.5.1 Display Memory Hardware Requirements

The minimum number of bits of memory required to implement the display memory is the product of the total number of pixels (on-screen and off-screen areas combined) and the number of bits per pixel. The minimum number of VRAMs required to contain the display memory is calculated as follows:

$$\text{Number of VRAMs} = \frac{(\text{pixels per line}) \times (\text{lines per frame}) \times (\text{bits per pixel})}{\text{Number of bits per VRAM}}$$

This calculation yields the minimum number of VRAMs needed, but additional VRAMs may be required in some applications. For instance, XY addressing can be supported by making the number of pixels per line of the display memory a power of two, but this may require more than the minimum number of VRAMs needed to contain the display.

11.5.2 Memory Organization and Bank Selecting

During a single local memory cycle, one data word (16 bits) is transferred between the TMS34010 and the selected bank of memory. The memory is partitioned into a number of banks, where each bank contains the number of memory devices that can be accessed in a single memory cycle. The number of devices per bank is therefore determined by dividing the width of the data bus by the number of data pins per device. The TMS34010 data bus is 16 bits wide, and can access 16 memory data pins during a single cycle. This means, for example, that a bank composed of 64K-by-1 RAMs contains 16 RAM devices. A bank composed of 64K-by-4 RAMs contains 4 RAM devices.

In a typical system, the local memory is divided into two parts, one consisting of the display memory and the other consisting of additional DRAMs needed to store programs and data. This additional RAM can be called the *system* memory. A high-order address bit is typically used to select between the display memory and system memory. Within the display memory or system memory, some address bits are provided as the row and column addresses to the selected bank, while other address bits are used to select one of the banks.

The number of banks of VRAM needed for the display memory is calculated by dividing the total number of VRAMs by the number of VRAMs per bank. This in turn determines how many bank select bits must be decoded.

11.5.3 Dynamic RAM Refresh Addresses

DRAMs (and VRAMs) require periodic refreshing to retain their data. The TMS34010 automatically generates DRAM-refresh cycles at regular intervals. The interval between refresh cycles is programmable, and DRAM refreshing can be disabled in systems that do not require it.

The TMS34010 can be configured to generate one of two types of DRAM-refresh cycle timing:

- $\overline{\text{RAS}}$-only (see Figure 11-7) **or**
- $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ (see Figure 11-8).

During a $\overline{\text{RAS}}$-only refresh cycle, the TMS34010 provides the 8-bit row address needed to refresh a particular row within each of the DRAMs in the memory system. DRAMs that support $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ cycles each contain an on-chip counter which generates the row address needed during the cycle. In other words, these devices do not rely on the TMS34010 to provide the row address during the $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ cycle.

The row address output by the TMS34010 during a DRAM-refresh cycle is the same regardless of whether the TMS34010 is configured for $\overline{\text{RAS}}$-only or $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ refresh timing. Since the TMS34010 outputs a valid row address during a $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ cycle, a system can contain some DRAMs that use $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ refresh timing and others that use $\overline{\text{RAS}}$-only timing. This hybrid approach configures the TMS34010 to perform $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ refresh, and relies on external decode logic to prevent the active-low column address strobe from reaching those DRAMs that require $\overline{\text{RAS}}$-only refreshing. The decode logic detects the fact that $\overline{\text{CAS}}$ falls before $\overline{\text{RAS}}$ during a $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ cycle, and uses this to inhibit transmitting the $\overline{\text{CAS}}$ signal to the $\overline{\text{RAS}}$-only DRAMs.

Several bits in the CONTROL register determine the manner in which the TMS34010 performs DRAM refreshing. The RM bit selects the type of DRAM-refresh cycle:

- RM=0 selects $\overline{\text{RAS}}$-only cycles
- RM=1 selects $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ cycles

The RR bits determine the interval between DRAM-refresh cycles:

- RR=00_2 selects refreshing every 32 local clock periods.
- RR=01_2 selects refreshing every 64 local clock periods.
- RR=10_2 is a reserved code.
- RR=11_2 inhibits DRAM refreshing.

At reset, internal logic forces the RM bit to 0 and the RR field to 00_2. While the $\overline{\text{RESET}}$ signal to the TMS34010 is active, no DRAM-refresh cycles are performed. Following reset, the TMS34010 begins to automatically perform DRAM-refresh cycles at regular intervals.

Both the interval between DRAM-refresh cycles and the addresses output during the cycles are generated within the REFCNT (DRAM-refresh count) register. Bits 2–15 of REFCNT form a continuous binary counter. The RINTVL field occupies bits 2–7, and counts the length of the interval between successive internal requests for DRAM-refresh cycles. The eight MSBs of REFCNT form the ROWADR field, containing the row address output to memory during the DRAM-refresh cycle.

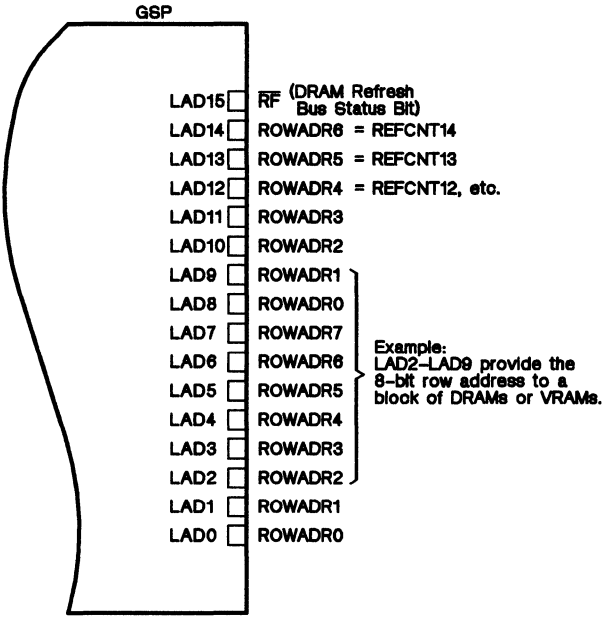


Figure 11-19. Row Address for DRAM-Refresh Cycle

During a DRAM-refresh cycle, the 8-bit row address in the ROWADR field of the REFCNT register is output on the LAD pins during the high-to-low transition of $\overline{RAS}$. As shown in Figure 11-19, the eight bits of ROWADR are output on LAD0–LAD7. The seven LSBs of ROWADR are also output on LAD8–LAD14. LAD15 transmits the $\overline{RF}$ bus status signal, low during the fall of $\overline{RAS}$.

Assume that LAD2–LAD9 are used as the 8-bit row address by a bank of DRAMs, as indicated in Figure 11-19. The address bits output on LAD2–LAD9 are the same eight bits output on LAD0–LAD7, but in a different order. During a series of 256 DRAM-refresh cycles, the row addresses output on LAD0–LAD7 and LAD2–LAD9 contain the same bits. Thus, if the addresses output on LAD0–LAD7 cycle through all 256 row addresses then the addresses output on LAD2–LAD9 also cycle through all 256 row addresses, but in a different order.

11.5.4 An Example - Memory Organization and Decoding

As an example, consider a memory organization based on the address decoding scheme shown in Figure 11-20. Three logical address bits (4, 21, and 26) are used as bank-select bits. Logical address bits 5–12 are used as the 8-bit column address, and bits 13–20 are used as the 8-bit row address. Referring to Figure 11-18, the row and column addresses are multiplexed out on the same eight pins, LAD1–LAD8. The total number of address bits used to address external memory is 19, for a total address reach of one megabyte. The remaining address bits output by the TMS34010 are not used for this example.

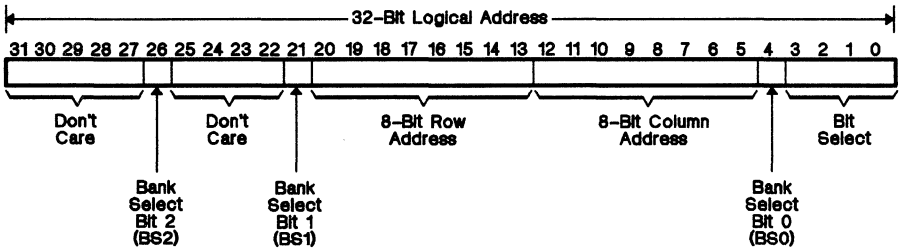


Figure 11-20. Address Decode for Example System

Bank select bit 2 (BS2) in Figure 11-20 selects between the display memory (BS2=0) and the system memory (BS2=1). System memory is a block of conventional DRAM (such as the TMS4256 and TMS4C1024) used for program and data storage. BS2 becomes valid before $\overline{RAS}$ falls, and thus can be used to determine whether the row-address strobe is gated to the display memory or to the system memory. The average power dissipation is reduced because only one or the other (the display memory or the system memory) is enabled during a particular memory read or write cycle.

Figure 11-21 shows the structure of the display memory. Its dimensions are 1024 by 1024 at four bits per pixel. Bank select bit 1 (BS1) selects between the top (BS1=0) and bottom (BS1=1) halves of the display memory. Since BS1 becomes valid before the fall of $\overline{RAS}$, it can be used to gate $\overline{RAS}$ to either the upper or lower half of the display memory during a memory read or write cycle. By transmitting the row address strobe to only half of the display memory, the power dissipation for the cycle is significantly reduced.

Bank select bit 0 (BS0) selects between the even word and odd word of each pair of adjacent words in the display memory. Each word contains four adjacent pixels. Odd and even words are stored in two separate banks of VRAMs, and the decode logic gates the column address strobe to the selected bank only. The row address strobe is gated to both banks (odd and even words). This increases the power dissipation over that required if only one bank were active. A compensating benefit of this organization, however, is that it reduces the rate at which each VRAM must supply serial data to refresh the screen. During screen refresh, the bank containing the even words and the bank containing the odd words alternately provide data to the video monitor. Alternating between the two banks in this fashion reduces the data bandwidth

requirements of each bank to about 10 MHz, which is an eighth of the video bandwidth.

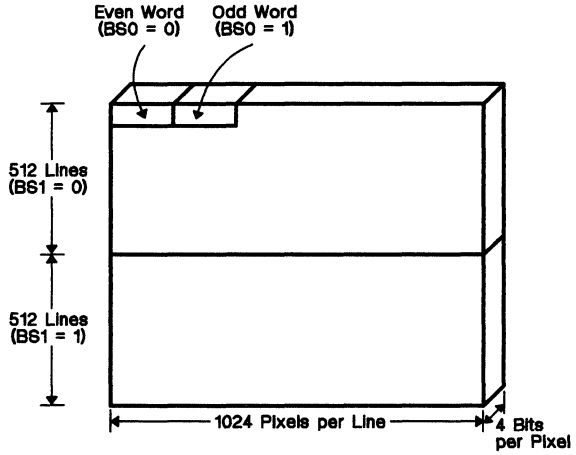


Figure 11-21. Display Memory Dimensions for the Example

The decode logic must be capable of more than just selecting a particular bank of the display memory or system memory during a memory read or write cycle. It must also be capable of enabling all DRAMs and VRAMs during a DRAM-refresh cycle, and enabling all VRAMs during a screen-refresh (memory-to-register) cycle. This means that the decode logic must distinguish DRAM-refresh and screen-refresh cycles from memory access cycles, and during a refresh cycle broadcast the row and column address strobes to all devices that require them. The timing of the $\overline{RF}$ and $\overline{TR}$ bus status bits has been designed to make these signals convenient for the design of the decode logic.

During a read or write cycle, the value of BS2, output with the row address, determines whether $\overline{RAS}$ is gated to the display memory or to system memory. During a DRAM-refresh cycle, the decode logic must broadcast the row-address strobe to all dynamic RAMs (including the VRAMs). The decode logic must be able to determine prior to the fall of the row address strobe whether the cycle that is beginning is a DRAM-refresh cycle, or a memory read or write cycle. This is the reason the TMS34010 outputs the $\overline{RF}$ bus status signal prior to the fall of $\overline{RAS}$.

The decode logic uses the value of BS1 to determine whether the top or bottom half of the display memory receives an active row-address strobe during a memory read or write cycle. The same logic must also be capable of broadcasting $\overline{RAS}$ to all VRAMs during either a DRAM-refresh cycle or a register-transfer cycle. The decode logic therefore monitors the state of the TMS34010's $\overline{TR}/\overline{QE}$ output prior to the fall of $\overline{RAS}$. A low level on $\overline{TR}/\overline{QE}$ indicates that the cycle just beginning is a register-transfer cycle, and that $\overline{RAS}$ should be broadcast.

While the decode logic uses the value of BS0 to determine whether the even or odd word receives a column-address strobe during a read or write cycle involving the display memory, the same logic must be capable of broadcasting $\overline{CAS}$ to all VRAMs during a screen-refresh cycle. Rather than require an external latch to capture the state of the $\overline{TR}/\overline{QE}$ during the fall of $\overline{RAS}$, the TMS34010 outputs the same information a second time in the form of the $\overline{TR}$ bus status signal, which is valid prior to and during the fall of $\overline{CAS}$.

Section 12

TMS34010 Instruction Set

This section contains the TMS34010 instruction set (in alphabetical order). Related subjects, such as addressing modes, are presented first.

12.1 Style and Symbol Conventions

Table 12-1 defines symbols and abbreviations that are used throughout this section; the list following the table describes style conventions used in the instruction set descriptions. Section 12.2 (page 12-4) defines the symbols that indicate various addressing modes.

Table 12-1. Instruction Set Symbol and Abbreviation Definitions

Symbol	Definition	Symbol	Definition
Rs	Source register	Rd	Destination register
Rs**X**	X half of source register	Rs**Y**	Y half of source register
Rd**X**	X half of destination register	Rd**Y**	Y half of destination register
A*n*	Register *n* in register file A	B *n*	Register *n* in register file B
PC	Program counter	PC'	PC prime, specifies the address of the next instruction (current PC + length of the current instruction)
Rp	Pointer register		
ST	Status Register	SP	Stack pointer (A15 or B15)
C	Carry bit	N	Sign bit
V	Overflow bit	Z	Zero bit
IE	Global interrupt enable bit	TOS	Top of stack
SAddress	Source address	DAddress	Destination address
SOffset	Source offset	DOffset	Destination offset
LSB	Least significant bit	MSB	Most significant bit
MSW	Most significant word	LSW	Least significant word
IW	16-bit immediate value	IL	32-bit immediate value
K	5-bit constant	*cc*	Condition code for a jump
F	Optional field select parameter for MOVE instructions, **F=0** selects FS0/FE0, and **F=1** selects FS1/FE1	R	Register file select, indicates which register file (A or B) the operand registers are in. **R=0** specifies register file A, **R=1** specifies register file B

Program listings, coding examples, filenames, and symbol names are shown in a `special font`. Some examples and listings use `a bold version of the special font` for emphasis. Here is a sample program listing:

```
0011   00000210     0001      .field    1, 2
0012   00000212     0003      .field    3, 4
0013   00000215     0006      .field    6, 3
0014   00000220               .even
```

In **syntax descriptions**, the font indicates which parts of the syntax must be entered as shown, and which parts act as place holders indicating the type of information that should be entered. In addition, square brackets identify optional parameters.

- The instruction and any part of the instruction that should be entered as shown are in a **bold face**. Parameters that describe the type of information that should be entered are in *italics*. Here is an example of an instruction syntax:

CVXYL *Rs, Rd*

CVXYL is an instruction that has two parameters, *Rs* and *Rd Rs* and *Rd* are abbreviations for *source register* and *destination register*; when you use CVXYL, these parameters must be real register names (such as A0, B1, etc.). Applying these rules, a valid CVXYL instruction is `CVXYL A0, A3`.

Another example of an instruction syntax is:

PIXBLT B,XY

In this case, **B** and **XY** do not specify values or data; they specify the *type* of PIXBLT instruction, and the instruction should be entered as shown: `PIXBLT B,XY`.

- Square brackets ([and]) identify an optional parameter. Here's an example of an instruction that has an optional parameter:

  CMPI */W, Rd [,* **W***]*

  The **CMPI** instruction has three parameters. The first two parameters, */W* and *Rd*, indicate a 16-bit value and a destination register; these parameters are required. The third parameter, **W**, is optional. As this syntax shows, if you use the optional third parameter, you must precede it with a comma.

Each instruction contains an **instruction execution** field that describes the actions that occur during instruction execution. These descriptions the following symbols and conventions:

- The → symbol means *becomes the contents of*. For example, *Rs* → *PC* means that the contents of the source register become the contents of the PC; that is, the contents of the source register are copied into the PC.

- The | | symbols indicate an absolute value.

- The : symbol indicates concatenation. For example, *Rd:Rd+1* identifies the concatenation of two consecutive registers, such as A0 and A1.

Numeric constants such as hexadecimal, octal, and binary numbers are identified by a letter suffix. Valid suffixes include:

- b or B (binary)
- q or Q (octal)
- h or H (hexadecimal)

Decimal constants have no suffix. Note that all constants must start with a numeral; for example, ABCDh is an illegal constant; 0ABCDh is the legal form.

12.2 Addressing Modes and Operand Formats

The TMS34010 instruction set supports eight addressing modes. Most instructions have register-direct operands or a combination of register-direct and immediate operands; however, the move and graphics instructions use more complex combinations of operands. This section discusses the TMS34010 addressing modes, and defines the symbols used in instruction syntax to indicate an addressing mode.

12.2.1 Immediate Values and Constants

An instruction syntax may use one of these symbols to indicate an immediate *source* operand:

IW is a 16-bit (short) signed immediate value.
IL is a 32-bit (long) signed immediate value.
K is a 5-bit constant.

Instructions that have immediate source operands have register-direct destination operands. Many instructions that have an immediate value can use either a short or a long value.

Figure 12-1 illustrates a MOVI (move immediate) instruction whose first operand is a 32-bit immediate value. The syntax for this MOVI is:

MOVI *IL, Rd [,* **L***]*

The instruction in Figure 12-1 is:

```
MOVI    0FC0h, A2, L
```

Figure 12-1 shows the object code (at address *N*) in memory and the effect of the instruction on the CPU registers. The value 0FC0h is copied into register A2 as a zero-extended 32-bit value. (Note that this is a 2-word instruction; the next instruction to be executed is at address *N*=2.)

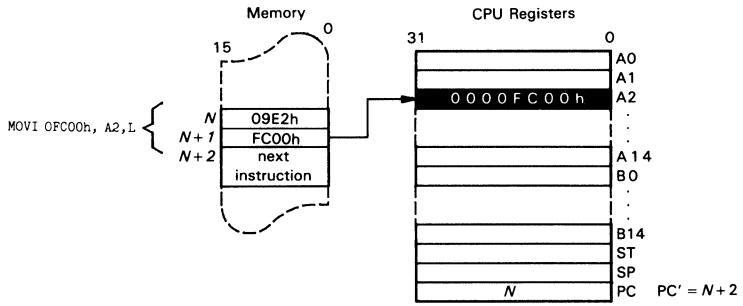


Figure 12-1. An Example of Immediate Addressing

12.2.2 Absolute Addresses

An instruction syntax may use one of these symbols to indicate an absolute operand:

@SAddress is a **source** address that contains the source data.
@DAddress is a **destination** address.

Note that the @ character is entered as part of the operand (this distinguishes it from an immediate operand).

Figure 12-2 illustrates a MOVB (move byte) instruction that has an absolute operand (the first parameter is a 32-bit source address). The syntax for this MOVB is:

MOVB *@SAddress, Rd*

The instruction in Figure 12-2 is:

```
MOVB    @RoutineA, A13
```

Figure 12-2 shows the object code (at address *N*) in memory and the effect of the instruction on the CPU registers. `@RoutineA` is the address of a byte; this MOVB instruction copies the byte at address `RoutineA` into register A13. (Note that this is a 3-word instruction; the next instruction to be executed is at address *N*=3.)

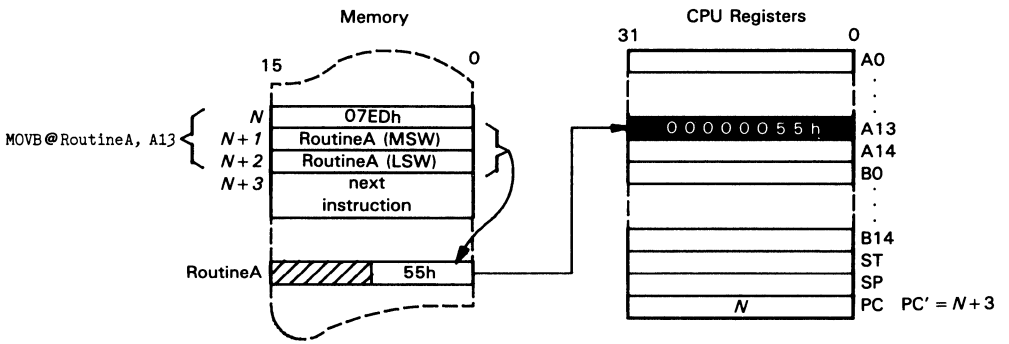


Figure 12-2. An Example of Absolute Addressing

12.2.3 Register-Direct Operands

An instruction syntax may use one of these symbols to indicate a register-direct operand:

Rs is a **source** register that contains the source data.
Rd is a **destination** register that will contain the result.

When both operands of an instruction are register-direct operands, the registers *must be in the same file.* (The MOVE *Rs,Rd* instruction is an exception to this rule.)

Figure 12-3 illustrates a MOVE (move field) instruction that has two register-direct operands. The syntax for this MOVE is:

MOVE *Rs, Rd [, F]*

The example shows this instruction:

```
MOVE    A0, B1
```

Figure 12-3 shows the object code (at address *N*) in memory and the effect of the instruction on the CPU registers. Assume that the field size for the move is 32 bits; the entire contents of register A0 are copied into register B1. (Note that this is a 1-word instruction; the next instruction to be executed is at address *N*=1.)

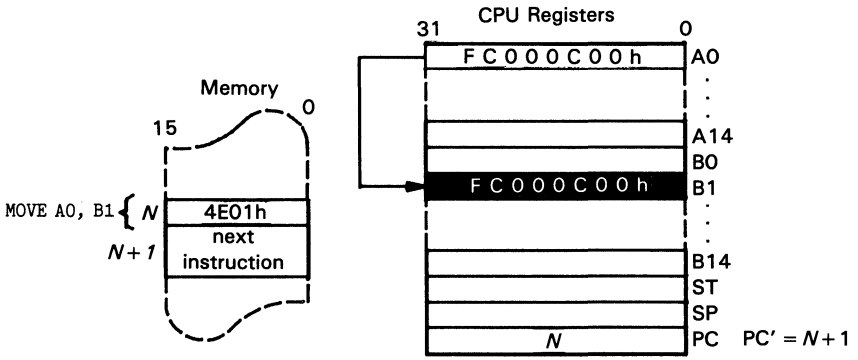


Figure 12-3. An Example of Register-Direct Addressing

12.2.4 Register-Indirect Operands

An instruction syntax may use one of these symbols to indicate a register-indirect operand:

Rs* is a register that contains the address of the **source data.
Rd* is a register that contains the **destination address.

Note that the * character is entered as part of the operand (this distinguishes it from a register-direct operand).

Figure 12-4 illustrates a MOVE (move field) instruction that has two register-indirect operands. The syntax for this MOVE is:

MOVE **Rs, *Rd*

The example shows this instruction:

```
MOVE     *A4, *A3
```

Figure 12-4 shows the object code (at address *N*) in memory and the effect of the instruction on the destination address. The contents of register A4 specify the address of data to be moved; the contents of register A3 specify the destination address. Assume that the field size for the move is 16 bits; the 16 bits of data at *A4 is moved to the location at *A3.. (Note that this is a 1-word instruction; the next instruction to be executed is at address *N*=1.)

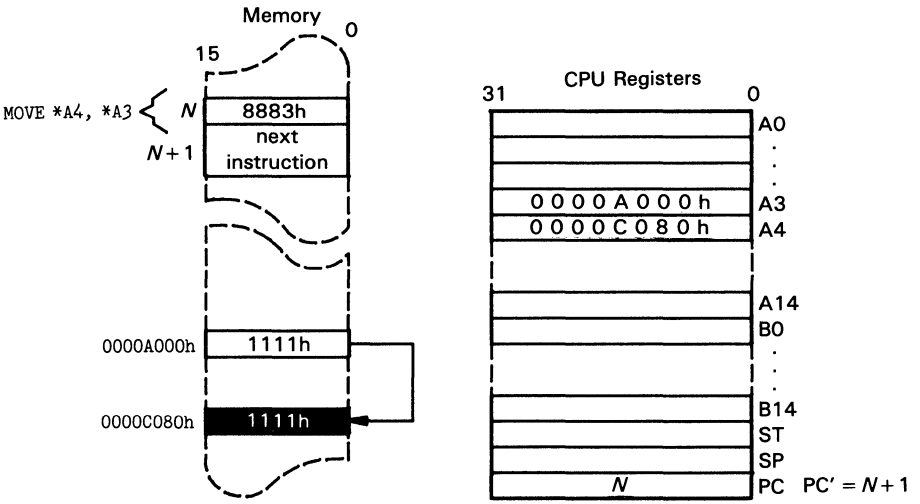


Figure 12-4. An Example of Register-Indirect Addressing

12.2.5 Register-Indirect with Offset

An instruction syntax may use one of these symbols to indicate a register-indirect operand that uses a signed offset:

Rs(offset)* is a **source address formed by adding an offset to the contents of the source register.
Rd(offset)* is a **destination address formed by adding an offset to the contents of the destination register.

The offset is only used to form an address - the contents of the register are not affected. Note that the * character is entered as part of the operand. If both operands use offsets, the syntax may list the operands as **Rs(SOffset)* or **Rd(DOffset)*.

Figure 12-5 illustrates a MOVE (move field) instruction; the first operand of this instruction is a register-direct operand; the second operand is a register-indirect operand with an offset. The syntax for this MOVE is:

MOVE *Rs, *Rd(offset) [, F]*

The example shows this instruction:

```
MOVE     B5, *B7(32)
```

Figure 12-5 shows the object code (at address *N*) in memory and the effect of the instruction on the destination location. The destination address is specified by adding the offset (32 bits, which is equivalent to 2 words) to the contents of register B7; this yields a destination location of 05020h. Assume that the field size for the move is 16 bits; the 16 LSBs in register B5 are copied into the destination location. (Note that this is a 2-word instruction; the next instruction to be executed is at address *N*=2.)

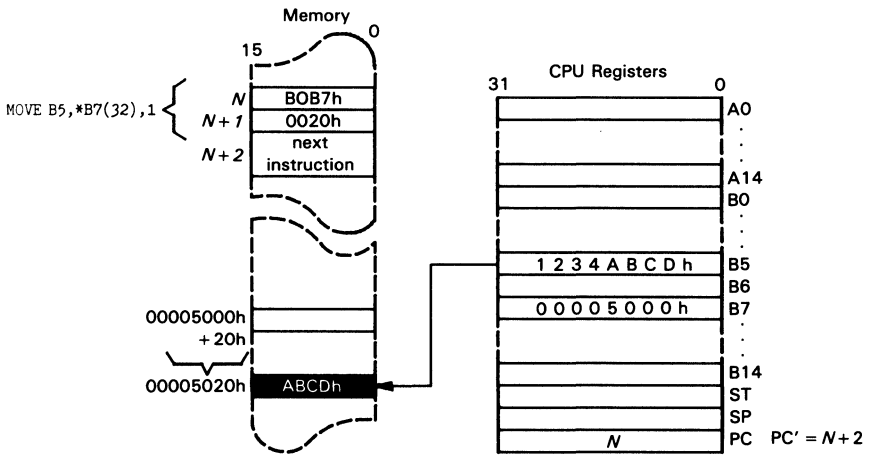


Figure 12-5. An Example of Register-Indirect with Offset Addressing

12.2.6 Register-Indirect with Postincrement

An instruction syntax may use one of these symbols to indicate a register-indirect operand that is postincremented:

Rs+* is a register that contains the address of the **source data.
Rd+* is a register that contains the **destination address.

After the operation is performed, the contents of the specified source or destination register are incremented by the field size used for the operation.

Note that the * and + characters are entered as part of the operand.

Figure 12-6 illustrates a MOVE (move field) instruction; both the source and the destination operands are postincremented register-indirect operands. The syntax for this MOVE is:

MOVE **Rs+, *Rd+ [, F]*

The example shows this instruction:

```
MOVE    *B4+, *B14+
```

Figure 12-6 shows the object code (at address *N*) in memory and the effect of the instruction on the destination location and the CPU registers. The contents of register B4 are the address of the source data; the contents of register B14 specify the destination address. Assume that the field size for the move is 16 bits; the 16 bits of data at the source address are copied into the destination location. After the move, both registers are incremented by 16 bits (1 word). (Note that this is a 1-word instruction; the next instruction to be executed is at address *N*=1.)

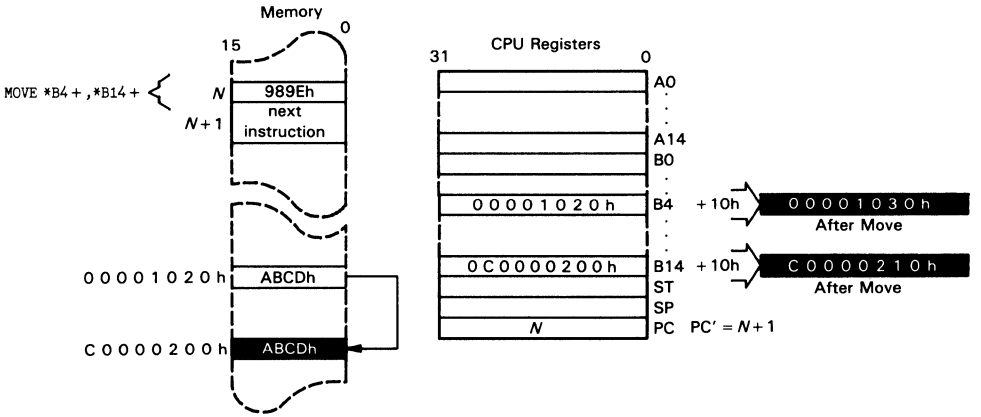


Figure 12-6. An Example of Register-Indirect with Postincrement Addressing

12.2.7 Register-Indirect with Predecrement

An instruction syntax may use one of these symbols to indicate a register-indirect operand that is predecremented.

Before the operation is performed, the contents of the specified source or destination register are decremented by the field size used for the operation.

-Rs* the decremented register contents are the address of the **source data.
-Rd* the decremented register contents specify the **destination address.

Note that the * and - characters are entered as part of the operand.

Figure 12-7 illustrates a MOVE (move field) instruction; the source operand is a register-direct operand the the destination operand is a predecremented register-indirect operand. The syntax for this MOVE is:

MOVE *Rs, *-Rd [, F]*

The example shows this instruction:

```
MOVE    A4, *-A3
```

Figure 12-7 shows the object code (at address *N*) in memory and the effect of the instruction on the destination location and the CPU registers. Assume that the field size for the move is 16 bits. Register A4 contains the source data. The contents of register A3, *minus the field size* (16 bits, or 1 word) form the destination address - 5150h. The 16 LSBs in A4 are copied to address 5150h. (Note that this is a 1-word instruction; the next instruction to be executed is at address *N*=1.)

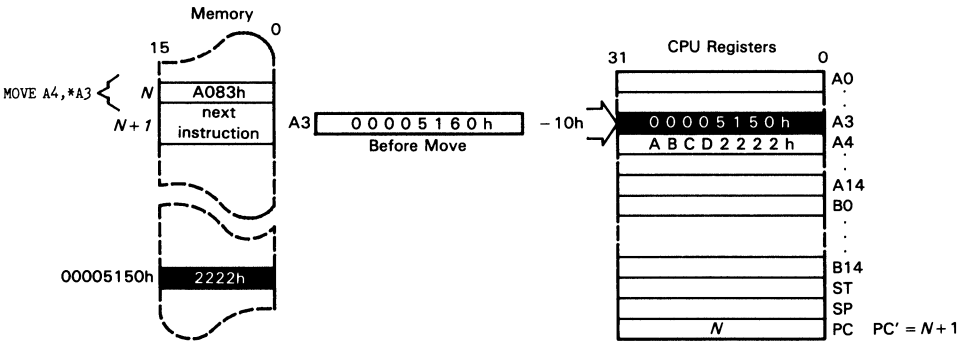


Figure 12-7. An Example of Register-Indirect with Predecrement Addressing

12.2.8 Register-Indirect in XY Mode

An instruction syntax may use one of these symbols to indicate that the a register operands contains an XY address.

Rs*.XY** is a register that contains the XY address of the **source** data.
Rd*.XY** is a register that contains the XY **destination** address.

Note that the * and .**XY** characters are entered as part of the operand. Here's an example that uses an indirect-XY destination operand:

```
PIXT  A0, *A6.XY
```

This instruction moves the contents of register A0 into the XY address specified by the contents of register A6.

12.3 Instruction Set Summary Table

Arithmetic, Logical, and Compare Instructions			
Syntax and Description	**Words**	**Machine States**	**16-Bit Opcode** **MSB LSB**
ABS *Rd* Store absolute value	1	1,4	0000 0011 100R DDDD
ADD *Rs, Rd* Add registers	1	1,4	0100 000S SSSR DDDD
ADDC *Rs, Rd* Add registers with carry	1	1,4	0100 001S SSSR DDDD
ADDI *IW, Rd* Add immediate (16 bits)	2	2,8	0000 1011 000R DDDD
ADDI *IL, Rd* Add immediate (32 bits)	3	3,12	0000 1011 001R DDDD
ADDK *K, Rd* Add constant (5 bits)	1	1,4	0001 00KK KKKR DDDD
ADDXY *Rs, Rd* Add registers in XY mode	1	1,4	1110 000S SSSR DDDD
AND *Rs, Rd* AND registers	1	1,4	0101 000S SSSR DDDD
ANDI *IL, Rd* AND immediate (32 bits)	3	3,12	0000 1011 100R DDDD
ANDN *Rs, Rd* AND register with complement	1	1,4	0101 001S SSSR DDDD
ANDNI *IL, Rd* AND not immediate (32 bits)	3	3,12	0000 1011 100R DDDD
BTST *K, Rd* Test register bit, constant	1	1,4	0001 11KK KKKR DDDD
BTST *Rs, Rd* Test register bit, register	1	2,5	0100 101S SSSR DDDD
CLR *Rd* Clear register	1	1,4	0101 011D DDDR DDDD
CLRC Clear carry	1	1,4	0000 0011 0010 0000
CMP *Rs, Rd* Compare registers	1	1,4	0100 100S SSSR DDDD
CMPI *IW, Rd* Compare immediate (16 bits)	2	2,8	0000 1011 010R DDDD
CMPI *IL, Rd* Compare immediate (32 bits)	3	3,12	0000 1011 011R DDDD
CMPXY *Rs, Rd* Compare X and Y halves of registers	1	3,6	1110 010S SSSR DDDD
DEC *Rd* Decrement register	1	1,4	0001 0100 001R DDDD

Arithmetic, Logical, and Compare Instructions (Continued)				
Syntax and Description	**Words**	**Machine States**	**16-Bit Opcode** **MSB**	**LSB**
DIVS *Rs, Rd* Divide registers signed	1	40,43 Δ 39,42 Δ	0101 100S SSSR DDDD	
DIVU *Rs, Rd* Divide registers unsigned	1	37,40	0101 101S SSSR DDDD	
LMO *Rs, Rd* Leftmost one	1	1,4	0110 101S SSSR DDDD	
MODS *Rs, Rd* Modulus signed	1	40,43	0110 110S SSSR DDDD	
MODU *Rs, Rd* Modulus unsigned	1	35,38	0110 111S SSSR DDDD	
MPYS *Rs, Rd* Multiply registers (signed)	1	20,23	0101 110S SSSR DDDD	
MPYU *Rs, Rd* Multiply registers (unsigned)	1	21,24	0101 111S SSSR DDDD	
NEG *Rd* Negate register	1	1,4	0000 0011 101R DDDD	
NEGB *Rd* Negate register with borrow	1	1,4	0000 0011 110R DDDD	
NOT *Rd* Complement register	1	1,4	0000 0011 111R DDDD	
OR *Rs, Rd* OR registers	1	1,4	0101 010S SSSR DDDD	
ORI *IL, Rd* OR immediate (32 bits)	3	3,12	0000 1011 101R DDDD	
SETC Set carry	1	1,4	0000 1101 1110 0000	
SEXT *Rd, F* Sign extend to long	1	3,6	0000 01F1 000R DDDD	
SUB *Rs, Rd* Subtract registers	1	1,4	0100 010S SSSR DDDD	
SUBB *Rs, Rd* Subtract registers with borrow	1	1,4	0100 011S SSSR DDDD	
SUBI *IW, Rd* Subtract immediate (16 bits)	2	2,8	0000 1011 111R DDDD	
SUBI *IL, Rd* Subtract immediate (32 bits)	3	3,12	0000 1101 000R DDDD	
SUBK *K, Rd* Subtract constant (5 bits)	1	1,4	0001 01KK KKKR DDDD	
SUBXY *Rs, Rd* Subtract registers in XY mode	1	1,4	1110 001S SSSR DDDD	
XOR *Rs, Rd* Exclusive OR registers	1	1,4	0101 011S SSSR DDDD	
XORI *IL, Rd* Exclusive OR immediate value (32 bits)	3	3,12	0000 1011 110D DDDD	
ZEXT *Rd, F* Zero extend to long	1	1,4	0000 01F1 001R DDDD	

Δ Rd even/Rd odd

Move Instructions			
Syntax and Description	**Words**	**Machine States**	**16-Bit Opcode MSB LSB**
MMFM *Rs [, List]* Move multiple registers from memory	2	†	0000 1001 101R DDDD
MMTM *Rs [, List]* Move multiple registers to memory	2	†	0000 1001 100R DDDD
MOVB *Rs, *Rd* Move byte, register to indirect	1	¶	1000 110S SSSR DDDD
MOVB **Rs, Rd* Move byte, indirect to register	1	¶	1000 111S SSSR DDDD
MOVB **Rs, *Rd* Move byte, indirect to indirect	1	¶	1001 110S SSSR DDDD
MOVB *Rs, *Rd(offset)* Move byte, register to indirect with offset	2	¶	1010 110S SSSR DDDD
MOVB **Rs(offset), Rd* Move byte, indirect with offset to register	2	¶	1010 111S SSSR DDDD
MOVB **Rs(SOffset), *Rd(DOffset)* Move byte, indirect with offset to indirect with offset	3	¶	1011 110S SSSR DDDD
MOVB *Rs, @DAddress* Move byte, register to absolute	3	¶	0000 0101 111R SSSS
MOVB *@SAddress, Rd* Move byte, absolute to register	3	¶	0000 0111 111R DDDD
MOVB *@SAddress, @DAddress* Move byte, absolute to absolute	5	¶	0000 0011 0100 0000
MOVE *Rs, Rd* Move register to register	1	1,4	0100 11MS SSSR DDDD
MOVE *Rs, *Rd [, F]* Move field, register to indirect	1	¶	1000 00FS SSSR DDDD
MOVE *Rs, -*Rd [, F]* Move field, register to indirect (predecrement)	1	¶	1010 00FS SSSR DDDD
MOVE *Rs, *Rd+ [, F]* Move field, register to indirect (postincrement)	1	¶	1001 00FS SSSR DDDD
MOVE **Rs, Rd [, F]* Move field, indirect to register	1	¶	1000 01FS SSSR DDDD
MOVE -**Rs, Rd [, F]* Move field, indirect (predecrement) to register	1	¶	1010 01FS SSSR DDDD
MOVE **Rs+, Rd [, F]* Move field, indirect (postincrement) to register	1	¶	1001 01FS SSSR DDDD

† See instruction
¶ See Section 13.2, MOVE and MOVB Instructions Timing

Move Instructions (Continued)			
Syntax and Description	**Words**	**Machine States**	**16-Bit Opcode** MSB LSB
MOVE **Rs, *Rd [, F]* Move field, indirect to indirect	1	¶	1000 10FS SSSR DDDD
MOVE *-*Rs, -*Rd [, F]* Move field, indirect (predecrement) to indirect (predecrement)	1	¶	1010 10FS SSSR DDDD
MOVE **Rs+, *Rd+ [, F]* Move field, indirect (postincrement) to indirect (postincrement)	1	¶	1001 10FS SSSR DDDD
MOVE *Rs, *Rd(offset) [, F]* Move field, register to indirect with offset	2	¶	1011 00FS SSSR DDDD
MOVE **Rs(offset), Rd [, F]* Move field, indirect with offset to register	2	¶	1011 01FS SSSR DDDD
MOVE **Rs(offset), *Rd+ [, F]* Move field, indirect with offset to indirect (postincrement)	2	¶	1101 00FS SSSR DDDD
MOVE **Rs(SOffset), *Rd(DOffset) [, F]* Move field, indirect with offset to indirect with offset	3	¶	1011 10FS SSSR DDDD
MOVE *Rs, @DAddress [, F]* Move field, register to absolute	3	¶	0000 01F1 100R SSSS
MOVE *@SAddress, Rd [, F]* Move field, absolute to register	3	¶	0000 01F1 101R DDDD
MOVE *@SAddress, *Rd+ [, F]* Move field, absolute to indirect (postincrement)	3	¶	1101 01F0 000R DDDD
MOVE *@SAddress, @DAddress [, F]* Move field, absolute to absolute	5	¶	0000 01F1 1100 0000
MOVI *IW, Rd* Move immediate (16 bits)	2	2,8	0000 1001 110R DDDD
MOVI *IL, Rd* Move immediate (32 bits)	3	3,12	0000 1001 111R DDDD
MOVK *K, Rd* Move constant (5 bits)	1	1,4	0001 10KK KKKR DDDD
MOVX *Rs, Rd* Move X half of register	1	1,4	1110 110S SSSR DDDD
MOVY *Rs, Rd* Move Y half of register	1	1,4	1110 111S SSSR DDDD

† See instruction
¶ See Section 13.2, MOVE and MOVB Instructions Timing

Graphics Instructions			
Syntax and Description	**Words**	**Machine States**	**16-Bit Opcode MSB LSB**
CPW *Rs, Rd* Compare point to window	1	1,4	1110 011S SSSR DDDD
CVXYL *Rs, Rd* Convert XY address to linear address	1	3,6	1110 100S SSSR DDDD
DRAV *Rs, Rd* Draw and advance	1	†	1111 011S SSSR DDDD
FILL L Fill array with processed pixels, linear	1	‡	0000 1111 1100 0000
FILL XY Fill array with processed pixels, XY	1	‡	0000 1111 1110 0000
LINE [0, 1] Line draw	1	†	1101 1111 Z001 1010
PIXBLT B, L Pixel block transfer, binary to linear	1	‡‡	0000 1111 1000 0000
PIXBLT B, XY Pixel block transfer and expand, binary to XY	1	‡‡	0000 1111 1010 0000
PIXBLT L, L Pixel block transfer, linear to linear	1	§	0000 1111 0000 0000
PIXBLT L, XY Pixel block transfer, linear to XY	1	§	0000 1111 0010 0000
PIXBLT XY, L Pixel block transfer, XY to linear	1	§	0000 1111 0100 0000
PIXBLT XY, XY Pixel block transfer, XY to XY	1	§	0000 1111 0110 0000
PIXT *Rs, *Rd* Pixel transfer, register to indirect	1	†	1111 100S SSSR DDDD
PIXT *Rs, *Rd.XY* Pixel transfer, register to indirect XY	1	†	1111 000S SSSR DDDD
PIXT **Rs, Rd* Pixel transfer, indirect to register	1	†	1111 101S SSSR DDDD
PIXT **Rs, *Rd* Pixel transfer, indirect to indirect	1	†	1111 110S SSSR DDDD
PIXT **Rs.XY, Rd* Pixel transfer, indirect XY to register	1	†	1111 001S SSSR DDDD
PIXT **Rs.XY, *Rd.XY* Pixel transfer, indirect XY to indirect XY	1	†	1111 010S SSSR DDDD

† See instruction
‡ See Section 13.3, FILL Instructions Timing
‡‡ See Section 13.5, PIXBLT Expand Instructions Timing
§ See Section 13.4, PIXBLT Instructions Timing

Program Control and Context Switching Instructions							
Syntax and Description	**Words**	**Machine States**	**16-Bit Opcode**				
			MSB				**LSB**
CALL *Rs* Call subroutine indirect	1	3+(3),9 3+(9),15Θ	0000	1001	001R	DDDD	
CALLA *Address* Call subroutine address	3	4+(2),15 4+(8),21Θ	0000	1101	0101	1111	
CALLR *Address* Call subroutine relative	2	3+(2),11 3+(8),17Θ	0000	1101	0011	1111	
DINT Disable interrupts	1	3,6	0000	0011	0110	0000	
EINT Enable interrupts	1	3,6	0000	1101	0110	0000	
EMU Initiate emulation	1	6,9	0000	0001	0000	0000	
EXGF *Rd, F* Exchange field size	1	1,4	1101	01F1	000R	DDDD	
EXGPC *Rd* Exchange program counter with register	1	2,5	0000	0001	001R	DDDD	
GETPC *Rd* Get program counter into register	1	1,4	0000	0001	010R	DDDD	
GETST *Rd* Get status register into register	1	1,4	0000	0001	100R	DDDD	
NOP No operation	1	1,4	0000	0011	0000	0000	
POPST Pop status register from stack	1	8,11 10,13Θ	0000	0001	1100	0000	
PUSHST Push status register onto stack	1	2+(3),8 2+(8),13Θ	0000	0001	1110	0000	
PUTST *Rs* Copy register into status	1	3,6	0000	0001	101R	DDDD	
RETI Return from interrupt	1	11,14 15,18Φ	0000	1001	0100	0000	
RETS *[N]* Return from subroutine	1	7,10 9,12Φ	0000	1001	011N	NNNN	
REV *Rd* Find TMS34010 revision level	1	1,4	0000	0000	001R	DDDD	
SETF *FS, FE, F* Set field parameters	1	1,4 2,5 ‡	0000	01F1	01FS	SSSS	
TRAP *N* Software interrupt	1	16,19 30,33Θ	0000	1001	000N	NNNN	

† See instruction
Θ First values for SP aligned, second values for SP nonaligned

Jump Instructions			
Syntax and Description	**Words**	**Machine States**	**16-Bit Opcode** MSB LSB
DSJ *Rd, Address* Decrement register and skip jump	2	3,9 ⊓ 2,8	0000 1101 100R DDDD
DSJEQ *Rd, Address* Conditionally decrement register and skip jump	2	3,9 ⊓ 2,8	0000 1101 101R DDDD
DSJNE *Rd, Address* Conditionally decrement register and skip jump	2	3,9 ⊓ 2,8	0000 1101 110R DDDD
DSJS *Rd, Address* Decrement register and skip jump short	1	2,5 ⊓ 3,6	0011 1Dxx xxxR DDDD
JA*cc Address* Jump absolute conditional	3	3,6 ⊓ 4,7	1100 code 1000 0000
JR*cc Address* Jump relative conditional	2	3,6 ⊓ 1,4	1100 code 0000 0000
JR*cc Address* Jump relative conditional short	1	2,5 ⊓ 2,5	1100 code xxxx xxxx
JUMP *Rs* Jump indirect	1	2,5	0000 0001 011R DDDD
Shift Instructions			
Syntax and Description	**Words**	**Machine States**	**16-Bit Opcode** MSB LSB
RL *K, Rd* Rotate left, constant	1	1,4	0011 00KK KKKR DDDD
RL *Rs, Rd* Rotate left, register	1	1,4	0110 100S SSSR DDDD
SLA *K, Rd* Shift left arithmetic, constant	1	3,6	0010 00KK KKKR DDDD
SLA *Rs, Rd* Shift left arithmetic, register	1	3,6	0110 000S SSSR DDDD
SLL *K, Rd* Shift left logical, constant	1	1,4	0010 01KK KKKR DDDD
SLL *Rs, Rd* Shift left logical, register	1	1,4	0110 001S SSSR DDDD
SRA *K, Rd* Shift right arithmetic, constant	1	1,4	0010 10KK KKKR DDDD
SRA *Rs, Rd* Shift right arithmetic, register	1	1,4	0110 010S SSSR DDDD
SRL *K, Rd* Shift right logical, constant	1	1,4	0010 11KK KKKR DDDD
SRL *Rs, Rd* Shift right logical, register	1	1,4	0110 011S SSSR DDDD

⊓ First values for jump, second values for no jump

12.4 Arithmetic, Logical, and Compare Instructions

The TMS34010 supports a full range of arithmetic, logical, and compare instructions. Most of these instructions use register-direct operands; some use a combination of immediate and register-direct operands. Some instructions have several versions; each uses a different operand format. For example, the *ADD* instruction has several versions:

- The **ADD** instruction uses register-direct operands for both the source and destination operands.
- The **ADDI** instruction uses an immediate source with a destination register.
- The **ADDK** instruction uses a 5-bit constant as the source operand with a destination register.
- The **ADDXY** instruction is similar to the ADD instruction – both operands are register-direct operands – however, the registers contain XY values.

Some instructions that have immediate values as source operands (such as the ADDI instruction) have two forms: a *short form* and a *long form*. In the short form, the source operand is a *16-bit* immediate value and the instruction occupies *two words*. In the long form, the source operand is a *32-bit* immediate value and the instruction occupies *three words*. Each form of the instruction has an optional third operand: **W** for short and **L** for long. If you don't use the **W** or **L** operand, the assembler chooses the short or the long form, depending on the size of the source operand. Using **W** or **L** *forces* the assembler to use the short or long form, respectively. If you use **W** and the source value is greater than 16 bits, the assembler discards all but the 16 LSBs and issues a warning message. If you use **L** and the source value is less than 32 bits, the assembler sign-extends the value to 32 bits.

Some instructions that use immediate operands have only one version. In this case, the operand is long (32-bits).

> **Note:**
>
> When an instruction's source and destination operands are both register-direct operands, the registers *must be in the same file*. (The MOVE Rs, Rd instruction is an exception to this rule.)

12.5 Move Instructions Summary

The TMS34010 supports a variety of move instructions, allowing you to move immediate values into registers, move data between registers, and move data between registers and memory. Table 12-2 summarizes the various types of move instructions.

Table 12-2. Summary of Move Instructions

Move Type	Mnemonic	Description
Register	MOVE	Move register to register
Constant	MOVK	Move constant (5 bits)
	MOVI	Move immediate (16 bits)
	MOVI	Move immediate (32 bits)
XY	MOVX	Move 16 LSBs of register (X half)
	MOVY	Move 16 MSBs of register (Y half)
Multiple register	MMFM	Move multiple registers from memory
	MMTM	Move multiple registers to memory
Byte	MOVB	Move byte (8 bits, 9 addressing modes)
Field	MOVE	Move field to/from memory/register (15 addressing modes)

12.5.1 Register-to-Register Moves

The **MOVE** *Rs,Rd* instruction is a register-to-register move; it moves a full 32 bits of data between any two general-purpose registers. *This is the only MOVE instruction that allows you to move data between register files A and B.*

12.5.2 Value-to-Register Moves

The **MOVI** and **MOVK** instructions move immediate values into registers. MOVK moves a zero-extended value into a register; the value must be in the range of 1 to 32. The MOVI instruction has two forms; it can move a 16-bit or a 32-bit immediate value.

12.5.3 XY Moves

The **MOVX** and **MOVY** instructions move values into the 16 LSBs or 16 MSBs, respectively, of a register.

12.5.4 Multiple-Register Moves

The **MMTM** and **MMFM** instructions use register-direct operands. MMTM allows you to save several register values in memory; MMFM allows you to retrieve register values from memory. Both instructions have two types of operands:

- The *Rp* operand is a *register pointer*. For the MMTM instruction, Rp contains the memory address where MMTM stores the register values. For the MMFM instruction, Rp contains the memory address from which MMFM loads the stored register values.
- The *register list* operand is an optional list of registers. It specifies which registers are stored or retrieved, and also indicates the storing or retrieval order.

Note that Rp and all the registers in the list *must be in the same register file.*

12.5.5 Byte Moves

The **MOVB** instruction is a special form of the MOVE instruction; when you use MOVB, the field size is restricted to 8 bits. MOVB supports nine combinations of operand formats. There are three basic combinations:

- Register to memory (requires a field insertion),
- Memory to register (requires a field extraction), **and**
- Memory to memory (requires both field insertion and extraction).

Note that the MOVB instruction does not move data between registers.

The MOVB instruction allows a byte to begin on any bit boundary in memory. The byte's memory address points to the LSB of the byte. When a byte is moved into a register, the byte's LSB coincides with the register's LSB; the byte is sign-extended into the 24 MSBs of the register.

Table 12-3 lists the valid combinations of operand formats for the MOVB instruction.

Table 12-3. Summary of Operand Formats for the MOVB Instruction

	Destination			
Source	*Rd*	**Rd*	**Rd(DOffset)*	*@DAddress*
Rs		√	√	√
**Rs*	√	√		
**Rs(SOffset)*	√		√	
@SAddress	√			√

Sequences of byte moves are more efficient if the byte addresses are aligned on even 8-bit boundaries. Twice as many memory cycles are required to access a byte that straddles a word boundary.

12.5.6 Field Moves

The **MOVE** instruction supports eighteen combinations of operand formats. There are four basic combinations:

- Register to register,
- Register to memory,
- Memory to register, **and**
- Memory to memory.

The MOVE instruction moves a *field*. A field is a configurable data structure that is identified by its starting address and its length. Field lengths can range from 1 to 32 bits. A field's memory address points to the LSB of the field; the field occupies contiguous bits. A field in a register is right-justified within the register; the field's LSB coincides with the register's LSB.

Note that all forms of the MOVE instruction have an optional **F** parameter. (MOVE *Rs,Rd* is an exception to this; it doesn't have an F parameter because it always moves 32 bits.) **F** selects the field size and field extension for the MOVE:

- If **F=0**, FS0 and FE0 determine the field size and extension.
- If **F=1**, FS1 and FE1 determine the field size and extension.

If you don't specify 0 or 1, 0 is used as the default. The selected field size determines the size of the field that is moved. A moved field is either sign-extended or zero-extended, depending on the value of the appropriate field extension bit. You can use the SETF instruction to set the field size and extension.

Table 12-4 summarizes the valid combinations of operand formats for the MOVE instruction.

Table 12-4. Summary of Operand Formats for the MOVE Instruction

	Destination					
Source	*Rd*	**Rd*	**Rd+*	*-*Rd*	**Rd(DOffset)*	*@DAddress*
Rs	√	√	√	√	√	√
**Rs*	√	√				
**Rs+*	√		√			
*-*Rs*	√			√		
**Rs(SOffset)*	√		√		√	
@SAddress	√		√			√

12.5.6.1 Register-to-Memory Field Moves

Figure 12-8 illustrates the register-to-memory move operation. In this type of move, the source register contains the right-justified field data (width is specified by the field size). The destination location is the bit position pointed to by the destination memory address. The address consists of a portion defining the starting word in which the field is to be written and an offset into that word, the bit address. Depending on the bit address within this word and the field size, the destination location may extend into two or more words.

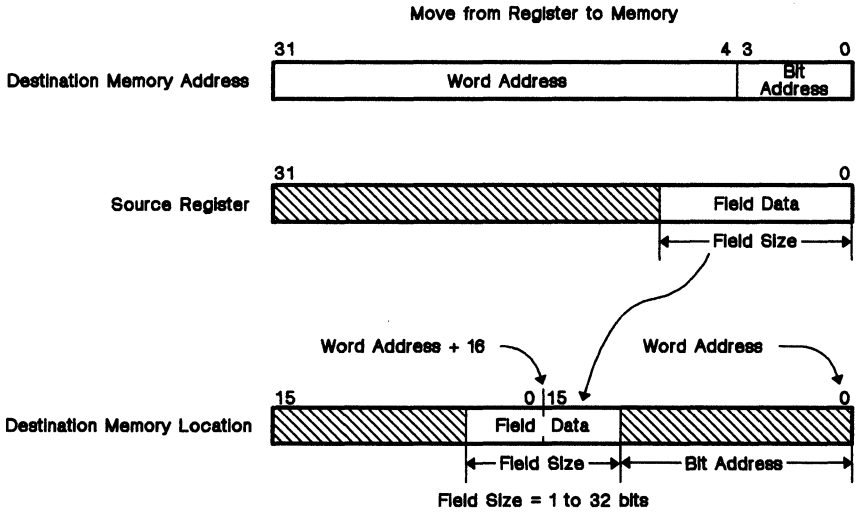


Figure 12-8. Register-to-Memory Moves

12.5.6.2 Memory-to-Register Field Moves

Figure 12-9 shows the memory-to-register move operation. The source memory location is the bit position pointed to by the source address. The address consists of a portion defining the starting word in which the field is to be written and an offset into that word, the bit address. Depending on the bit address within this word and the field size, the source location may extend into two or more words. After the move, the destination register LSBs contain the right-justified field data (width is specified by the field size). The MSBs of the register contain either all 1s or all 0s.

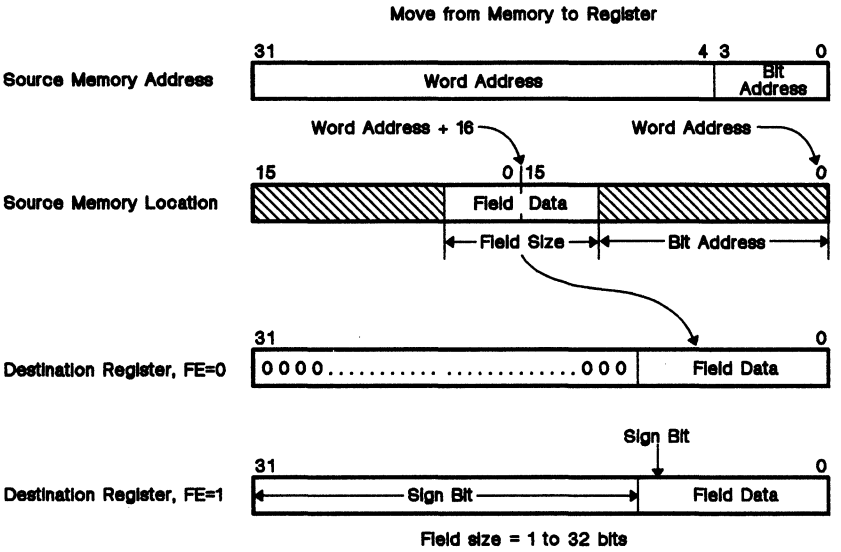


Figure 12-9. Memory-to-Register Moves

12.5.6.3 Memory-to-Memory Field Moves

Figure 12-10 shows a memory-to-memory field move operation. The source memory location is the bit position pointed to by the source address. The destination location is the bit position pointed to by the destination memory address. Depending on the bit addresses within the respective words and the field size, either the source location or destination locations may extend into two or more words. After the move, the destination location contains the field data from the source memory location.

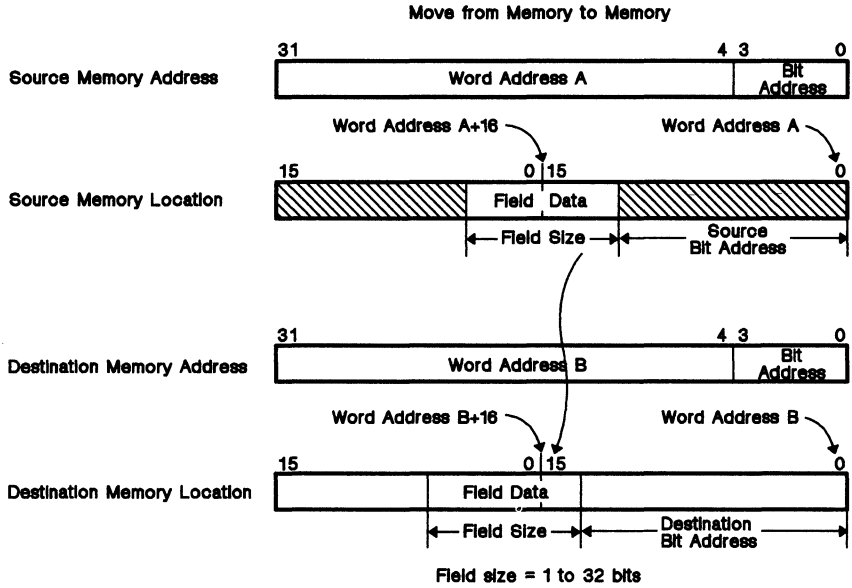


Figure 12-10. Memory-to-Memory Moves

12.6 Graphics Instructions Summary

The TMS34010 instruction set supports several fundamental graphics drawing operations.

12.6.1 Comparing a Point to a Window

The **CPW** instruction compares a point to the window limits defined by the WSTART and WEND registers. The source operand *Rs* contains an XY address. After the compare operation is performed, bits 5–8 contain a code that indicate the point's location with respect to the window limits. The description of the CPW instruction shows these point codes.

12.6.2 Converting an XY Address to a Linear Address

The **CVXYL** instruction converts an XY address to a 32-bit linear address. The source register contains the XY address; the linear address is put in the destination register.

12.6.3 Drawing a Pixel and Advancing to the Next Pixel Address

The **DRAV** instruction draws the pixel value in the COLOR1 register to the XY address specified by the destination register. After the pixel is drawn, the Y half of Rs is added to the Y half of Rd, and the X half of Rs is added to the X half of Rd.

12.6.4 Draw a Line

The **LINE** instruction performs the inner loop of Bresenham's line-drawing algorithm to draw an arbitrarily oriented, straight line. The optional operand may be a 0 or a 1; this selects one of two algorithms. The default for this operand is 0.

12.6.5 Filling a Pixel Block

The **FILL** instruction fills a two-dimensional pixel array with the value in the COLOR1 register. Note that **L** and **XY** *are not actually operands;* they are part of the instruction mnemonic, identifying the form of the FILL instruction. FILL L specifies that the array has a linear starting address; FILL XY specifies that the array has an XY starting address.

12.6.6 Moving a Single Pixel

The **PIXT** instruction transfers a pixel from one location to another. PIXT can transfer a pixel:

- From a register to memory,
- From memory to a register, **or**
- From memory to memory.

Table 12-5 summarizes the valid combinations of operand formats for the PIXT instruction. Note that all addresses are linear unless the operand is suffixed with **.XY**.

Table 12-5. Summary of Operand Formats for the PIXT Instruction

	Destination Pixel		
Source Pixel	*Rd*	**Rd*	**Rd*.**XY**
Rs		√	√
**Rs*	√	√	
Rs*.XY**	√		√

12.6.7 Moving a Two-Dimensional Block of Pixels

The **PIXBLT** instruction moves a two-dimensional block of pixels from one memory location to another. Note that **B**, **L**, and **XY** *are not actually operands*; instead, they identify the source or destination array starting addresses as binary, linear, or XY addresses. The source and destination addresses of the arrays are designated by the SADDR and DADDR registers, respectively.

Table 12-6 summarizes the various combinations of pixel block transfers.

Table 12-6. Summary of Array Types for the PIXBLT Instruction

	Destination Array	
Source Array	**Linear**	**XY**
Binary	√	√
Linear	√	√
XY	√	√

The graphics instructions use the B-file registers and several I/O registers as *implied operands*. These registers must be loaded with appropriate values *before the instruction is executed*. The TMS34010 obtains information from these registers during instruction execution. Table 12-7 summarizes the implied operands that are used by the graphics instructions. The *TMS34010 User's Guide* contains a complete discussion of these registers and describes the types of information they should contain.

12.6.8 Implied Operands

The graphics instructions require additional information that you supply by loading appropriate values into specific B registers and I/O registers. When these registers are used for this purpose, they are called **implied operands**. Section 5 discusses the functions of B registers as implied operands; Section 5 discusses the functions of I/O registers as implied operands.

Note that the LINE instruction uses registers B10–B13 as implied operands; as implied operands, these registers have the following functions:

B10: COUNT register
B11: INC1 register
B12: INC2 register
B13: PATTRN register

Table 12-7 identifies the implied operands that each graphics instruction uses.

Table 12-7. Implied Operands Used by Graphics Instructions

	B File Registers																I/O Registers				
	SADDR	SPTCH	DADDR	DPTCH	OFFSET	WSTART	WEND	DYDX	COLOR0	COLOR1	B10	B11	B12	B13	B14	CONTROL	CONVSP	CONVDP	PSIZE	PMASK
CPW *Rs, Rd*						XY	XY													
CVXYL *Rs, Rd*				L	L													√	√	
DRAV *Rs, Rd*				L	L	XY	XY			P						(1)		√	√	√
FILL L			L	L				XY		P	√	√	√	√	√	(2)			√	√
FILL XY			XY†	L	L	XY	XY	XY†		P	√	√	√	√	√	(1)		√	√	√
LINE [0, 1]	L		XY		L	XY	XY	XY		P	L	XY	XY	pat	√	(1)		√	√	√
PIXBLT B, L	L	L	L	L				XY	P	P	√	√	√	√	√	(2)			√	√
PIXBLT B, XY	L	L	XY†	L	L	XY	XY	XY†	P	P	√	√	√	√	√	(1)	√	√	√	√
PIXBLT L, L	L	L	L	L				XY			√	√	√	√	√	(4)	(5)	(5)	√	√
PIXBLT L, XY	L	L	XY†	L	L	XY	XY	XY†			√	√	√	√	√	(3)	√	√	√	√
PIXBLT XY, L	XY	L	L	L	L			XY			√	√	√	√	√	(4)	√	√	√	√
PIXBLT XY, XY	XY	L	XY†	L	L	XY	XY	XY†			√	√	√	√	√	(3)	√	√	√	√
PIXT *Rs,* ***Rd*																(2)			√	√
PIXT *Rs,* ***Rd*.XY**				L	L	XY	XY									(1)		√	√	√
PIXT ***Rs, Rd*																			√	√
PIXT ***Rs,* ***Rd*																(2)			√	√
PIXT ***Rs*.XY, *Rd*				L	L												√		√	√
PIXT ***Rs*.XY , ****Rd*.XY**		L		L	L	XY	XY									(1)	√	√	√	√

Key:
▒ Changed by instruction execution
√ Used; no particular format
XY Register is in XY format
L Register is in linear format
P Register is in pixel format
pat Register is in pattern format
† Changed as a result of common rectangle function with window hit operation (W=1)
(1) CONTROL bits used: PP, W, T
(2) CONTROL bits used: PP, T
(3) CONTROL bits used: PP, W, T, PBH, PBV
(4) CONTROL bits used: PP, T, PBH, PBV
(5) Used when PBV=1

12.7 Program Control and Context Switching Instructions

The TMS34010 supports a variety of instructions that allow you to control program flow and to save and restore information by letting you:

- Call and return from subroutines,
- Enable or disable interrupts,
- Set software interrupts,
- Set, save, or restore status information, **and**
- Use jump instructions to redirect program flow.

Most of these instructions use register-direct or absolute operands; however, several of them have no operands.

12.7.1 Subroutine Calls and Returns

The TMS34010 allows you to call a subroutine in three ways:

- Indirectly, by loading an address into a register;
- Directly, by using an absolute address; **and**
- Relatively, by specifying an address that is an offset.

These CALL instructions automatically save status information on the stack. The RETS (return from subroutine) instruction pops status information off of the stack and returns control to the program or routine that called the subroutine.

12.7.2 Interrupt Handling

The TMS34010's EINT and DINT instructions allow you to enable or disable hardware interrupts by providing control of the IE (global interrupt enable) status bit. The TMS34010 also supports a TRAP instruction that provides you with control over 32 software interrupts.

12.7.3 Setting, Saving, and Restoring Status Information

Although some instructions automatically save or restore status information, you will often want explicit control over these functions. The TMS34010 supports several instructions that allow you to save and restore PC and ST information. The TMS34010 also supports a SETF instruction that allows you to set field-0/field-1 information in the status register.

12.7.4 Jump Instructions

The TMS34010 supports both conditional and unconditional jumps. The conditional jumps use absolute operands or a combination of register-direct and absolute operands.

- There are four DSJ instructions:
  - **DSJ** and **DSJS** decrement the contents of a register and jump to the specified address if the new contents of Rd do not equal 0. If Rd is decremented to 0, then execution continues with the next instruction.

    DSJ provides a jump range of -32,768 to +32,767 words; DSJS provides a jump range of ±32 words (excluding 0).
  - The operation of **DSJEQ** and **DSJNE** depends on the value of the Z (zero) status bit.

    DSJEQ decrements the contents of Rd when **Z=1** and jumps to the specified address if the new contents of Rd do not equal 0. If Rd is decremented to 0, then execution continues with the next instruction. If **Z=0**, DSJEQ skips the jump and execution continues with the next instruction.

    DSJNE decrements the contents of Rd when **Z=0** and jumps to the specified address if the new contents of Rd do not equal 0. If Rd is decremented to 0, then execution continues with the next instruction. If **Z=0**, DSJNE skips the jump and execution continues with the next instruction.

  The address specified for the DSJ instructions is relative; the assembler uses this address automatically to calculate a displacement, and then it inserts the displacement into the instruction.

- The **JUMP** instruction is unconditional. The source register contains the address for the jump.

- The conditional jump instructions, **JA***cc* and **JR***cc*, use the condition codes listed Table 12-8.

  The JR*cc* instruction has a long and a short form. The short form supports a jump range of ±127 words (excluding 0). The long form supports a jump range of ±32K words (excluding 0).

The 32-bit address specified for the JA*cc* instruction is absolute; the assembler inserts this address into words 2 and 3 of the instruction. The address specified for the JR*cc* instructions is relative; the assembler uses this address automatically to calculate a displacement, and then it inserts the displacement into the instruction. The short form has an 8-bit displacement that is inserted into bits 0–7 of the opcode; the opcode is 1 word long. The long form has 16-bit displacement; the opcode is 2 words long, and the displacement occupies the entire 16 bits of the second word.

Table 12-8 lists the condition codes used with the JR*cc* and JA*cc* instructions. (To use the codes, replace the *cc* with the appropriate mnemonic code; for example, JRUC, JALS, JRYGT, etc.) Before using one of these jump instructions, use the CMP, CMPI, or CMPXY instruction; the compare instructions set the condition codes for the jump by subtracting a source value

from a destination value. The first mnemonics code column in Table 12-8 lists the codes that can be used for a jump following a CMP or CMPI. The second mnemonics code column list codes that can be used for a jump following a CMPXY (codes that are preceded with an *X* can be used with the result of the X comparison and codes that are preceded with a *Y* can be used with the result of the Y comparison).

Table 12-8. Condition Codes for JR*cc* and JA*cc* Instructions

	Mnemonic Code		Result of Compare	Status Bits	Code
Unconditional Compares	UC	–	Unconditional	don't care	0000
Unsigned Compares	LO (C)	–	Dst lower than Src	C	0001
	LS	YLE	Dst lower or same as Src	$C + Z$	0010
	HI	YGT	Dst higher than Src	$\overline{C} \cdot \overline{Z}$	0011
	HS (NC)	–	Dst higher or same as Src	$\overline{C}$	1001
	EQ (Z)	–	Dst = Src	Z	1010
	NE (NZ)	–	Dst ≠ Src	$\overline{Z}$	1011
Signed Compares	LT	XLE	Dst < Src	$(N \cdot \overline{V}) + (\overline{N} \cdot V)$	0100
	LE	–	Dst ≤ Src	$(N \cdot \overline{V} + (\overline{N} \cdot V) + Z$	0110
	GT	–	Dst > Src	$(N \cdot V \cdot \overline{Z}) + (\overline{N} \cdot \overline{V} \cdot \overline{Z})$	0111
	GE	XGT	Dst ≥ Src	$(N \cdot V) + (\overline{N} \cdot \overline{V})$	0101
	EQ (Z)	–	Dst = Src	Z	1010
	NE (NZ)	–	Dst ≠ Src	$\overline{Z}$	1011
Compare to Zero	Z	YZ	Result = zero	Z	0101
	NZ	YNZ	Result ≠ zero	$\overline{Z}$	1011
	P	–	Result is positive	$\overline{N} \cdot \overline{Z}$	0001
	N	XZ	Result is negative	N	1110
	NN	XNZ	Result is nonnegative	$\overline{N}$	1111
General Arithmetic	Z	YZ	Result is zero	Z	1010
	NZ	YNZ	Result is nonzero	$\overline{Z}$	1011
	C	YN	Carry set on result	C	1000
	NC	YNN	No carry on result	$\overline{C}$	1001
	B (C)	–	Borrow set on result	C	1000
	NB (NC)	–	No borrow on result	$\overline{C}$	1001
	V†	XN	Overflow on result	V	1100
	NV†	XNN	No overflow on result	$\overline{V}$	1101

Note: A mnemonic code in parentheses is an alternate code for the preceding code.
† Also used for window clipping
+ Logical OR
· Logical AND
‾ Logical NOT

12.8 Shift Instructions

The TMS3410 supports several instructions that left-rotate, left-shift, or right-shift the contents of the destination register. These instructions use register-direct operands or a combination of register-direct and immediate operands; the shift amount is specified by the value of a 5-bit constant **or** by the value specified in the 5 LSBs of a source register. (Note that the **SRA** *Rs, Rd* and **SRL** *Rs, Rd* use the 2s complement value of the 5 LSBs in Rs.)

- The **RL** instruction left-rotates the contents of the destination register by. (This rotation is a barrel shift.) The bits shifted out of the MSB are shifted into the LSB. The C (carry) bit is set to the final value shifted out of the MSB.

- The **SLA** instruction left shifts the contents of the destination register. 0s are shifted into the LSBs. The MSBs are shifted out through the C (carry) bit so that the C bit is set to the final value shifted out of the MSB. If either the N (sign) bit or any of the bits shifted out differ from the original sign bit, the V (overflow) bit is set.

- The **SLL** instruction left shifts the contents of the destination register. 0s are shifted into the LSBs. The MSBs are shifted out through the C (carry) bit so that the C bit is set to the final value shifted out of the MSB. The main difference between SLL and SLA is that SLL does not check to see if the sign bit changes.

- The **SRA** instruction right shifts the contents of the destination register. The value of the sign bit is shifted into the MSBs; this sign-extends the value and preserves the original value of the sign bit. The LSBs are shifted out through the C (carry) bit so that the C bit is set to the final value shifted out of the LSB.

- The **SRL** instruction right shifts the contents of the destination register. 0s are shifted into the MSBs, beginning with bit 31. The LSBs are shifted out through the C (carry) bit so that the C bit is set to the final value shifted out of the LSB. The main difference between SRL and SRA is that SRL does not preserve the original value of the sign bit.

12.9 XY Instructions

The TMS34010 allows you to use XY addresses. This is useful for specifying pixel addresses on the screen. Many of the graphics instructions use XY addressing; the TMS34010 instruction set also supports several other instructions that allow you to manipulate XY addresses.

An XY address is a 32-bit address that is divided into two parts. The 16 LSBs of the address are the X half of the address or register; the 16 MSBs of the address are the Y half of the address or register. The two parts are treated as completely separate values; for example, using the ADDXY instruction, the X half does not propagate into the Y half.

Table 12-9 summarizes the instructions that use XY addresses.

Table 12-9. Summary of XY Instructions

Instruction	Description	Instruction	Description
ADDXY *Rs, Rd*	Add registers in XY	**PIXBLT B, XY**	Pixel block transfer (binary to XY)
CPW *Rs, Rd*	Compare point to window	**PIXBLT L, XY**	Pixel block transfer (linear to XY)
CMPXY *Rs, Rd*	Compare registers in XY mode	**PIXBLT XY, L**	Pixel block transfer (XY to linear)
CVXYL *Rs, Rd*	Convert XY address to linear address	**PIXBLT XY, XY**	Pixel block transfer (XY to XY)
DRAV *Rs, Rd*	Draw and advance	**PIXT** *Rs, *Rd*.**XY**	Pixel transfer (register to indirect XY)
FILL XY	Fill array with processed pixels	**PIXT** **Rs*.**XY**, *Rd*	Pixel transfer (indirect XY to register)
LINE [**0, 1**]	Line draw with XY addressing	**PIXT** **Rs*.**XY**, **Rd*.**XY**	Pixel transfer (indirect XY to indirect XY)
MOVX *Rs, Rd*	Move X half of Rs to X half of Rd	**SUBXY** *Rs, Rd*	Subtract registers in XY mode
MOVY *Rs, Rd*	Move Y half of Rs to Y half of Rd		

- The **PIXBLT** and **FILL** instructions in Table 12-9 use XY source and/or destination addresses.
- The **PIXT** instructions in Table 12-9 use the contents of registers as XY addresses.
- The **LINE** instruction draws a line along points that are calculated as XY addresses.
- The move instructions in Table 12-9 (**MOVX** and **MOVY**) move the X or Y half of a source register into the X or Y half of a destination register.
- The arithmetic and logical instructions in Table 12-9 (**ADDXY**, **SUBXY**, and **CMPXY**) add, subtract, or compare the X and Y halves of the registers separately.

12.10 Alphabetical Reference of Instructions

The remainder of this section is an alphabetical reference of the TMS34010 assembly language instructions. Each instruction discussion begins on a new page, and contains the following information:

- ***Syntax:*** Shows you how to enter an instruction. (Section 12.1, page 12-2, describes the symbols used in instruction syntaxes.)
- ***Execution:*** Illustrates the effects of instruction execution on CPU registers and memory.
- ***Instruction Words:*** Shows the object code generated by an instruction.
- ***Description:*** Discusses the purpose of the instruction and any other general information related to the instruction.
- ***Machine States:*** Lists the instruction cycle timing. Two timings are listed for each instruction; the first number is the cache-enabled case, the second number is the cache-disabled case.
- ***Status Bits:*** Lists the effects of instruction execution on the status bits (N, C, Z, and V).
- ***Examples:*** Show the effects of the instruction on memory and registers using various sets of data and initial conditions.

Several instructions discuss additional topics; for example, the conditional jump instructions list the conditions codes and mnemonics for various jumps, and the graphics instructions list the implied operands that they use.

Syntax **ABS** *Rd*

Execution |Rd| → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	1	0	0	R	Rd			

Description ABS stores the absolute value of the contents of the destination register back into the destination register. This is accomplished by:

- Subtracting the contents of the destination register data from 0 **and**
- Storing the result back into Rd if status bit N indicates that the result is positive.

If the result of the subtraction is negative, then the original contents of the destination register are retained.

Machine States 1,4

Status Bits

N Set to the sign of the result of 0 - Rd; typically, N=*0* if the original contents of Rd are negative (unless Rd = 80000000h), *1* otherwise
C Unaffected
Z *1* if the original data is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise; an overflow occurs if Rd contains 80000000h (80000000h is returned)

Examples

Code	Before	After	
	A1	**NCZV**	**A1**
ABS A1	7FFFFFFFh	1x00	7FFFFFFFh
ABS A1	FFFFFFFFh	0x00	00000001h
ABS A1	80000000h	1x01	80000000h
ABS A1	80000001h	0x00	7FFFFFFFh
ABS A1	00000001h	1x00	00000001h
ABS A1	00000000h	0x10	00000000h
ABS A1	FFFA0011h	0x00	0005FFEFh

Syntax **ADD** *Rs, Rd*

Execution Rs + Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	Rs				R	Rd			

Description ADD adds the contents of the source register to the contents of the destination register, and stores the result in the destination register.

You can is the ADD instruction with the ADDC instruction to perform multiple-precision arithmetic.

Rs and Rd must be in the same register file.

Machine States 1,4

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a carry, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before		After	
	A1	A0	NCZV	A0
ADD A1,A0	FFFFFFFFh	FFFFFFFFh	1100	FFFFFFFEh
ADD A1,A0	FFFFFFFFh	00000001h	0110	00000000h
ADD A1,A0	FFFFFFFFh	00000002h	0100	00000001h
ADD A1,A0	FFFFFFFFh	80000000h	0101	7FFFFFFFh
ADD A1,A0	FFFFFFFFh	80000001h	1100	80000000h
ADD A1,A0	7FFFFFFFh	80000001h	0110	00000000h
ADD A1,A0	7FFFFFFFh	80000000h	1000	FFFFFFFFh
ADD A1,A0	7FFFFFFFh	00000001h	1001	80000000h
ADD A1,A0	00000002h	00000002h	0000	00000004h

Syntax **ADDC** *Rs, Rd*

Execution Rs + Rd + C → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	Rs				R	Rd			

Description ADDC adds the contents of the source register, the carry bit, and the contents of the destination register, and then stores the result in the destination register. Note that the status bits are set on the final result.

Rs and Rd must be in the same register file.

Machine States 1,4

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a carry, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before			After	
	C	A1	A0	NCZV	A0
ADDC A1,A0	1	FFFFFFFFh	FFFFFFFFh	1100	FFFFFFFFh
ADDC A1,A0	1	FFFFFFFFh	00000001h	0100	00000001h
ADDC A1,A0	1	FFFFFFFFh	00000002h	0100	00000002h
ADDC A1,A0	1	FFFFFFFFh	80000000h	1100	80000000h
ADDC A1,A0	1	FFFFFFFFh	80000001h	1100	80000001h
ADDC A1,A0	1	FFFFFFFFh	80000001h	0100	80000001h
ADDC A1,A0	1	FFFFFFFFh	80000000h	0110	00000000h
ADDC A1,A0	1	7FFFFFFFh	00000001h	1001	80000001h
ADDC A1,A0	1	00000002h	00000002h	0000	00000005h
ADDC A1,A0	0	FFFFFFFFh	FFFFFFFFh	1100	FFFFFFFEh
ADDC A1,A0	0	FFFFFFFFh	00000001h	0110	00000000h
ADDC A1,A0	0	FFFFFFFFh	00000002h	0100	00000001h
ADDC A1,A0	0	FFFFFFFFh	80000000h	0101	7FFFFFFFh
ADDC A1,A0	0	FFFFFFFFh	80000001h	1100	80000000h
ADDC A1,A0	0	7FFFFFFFh	80000001h	0110	00000000h
ADDC A1,A0	0	7FFFFFFFh	80000000h	1000	FFFFFFFFh
ADDC A1,A0	0	7FFFFFFFh	00000001h	1001	80000000h
ADDC A1,A0	0	00000002h	00000002h	0000	00000004h

Syntax **ADDI** *IW, Rd [, W]*

Execution IW + Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	0	0	0	R	Rd			
16-bit value															

Description This ADDI instruction adds a sign-extended, 16-bit immediate value to the contents of the destination register, and stores the result in the destination register. (The symbol *IW* in the syntax above represents a 16-bit, sign-extended immediate value.)

The assembler uses the short (16-bit) add if the immediate value is previously defined and is in the range -32,768 to 32,767. You can force the assembler to use the short form by following the register operand with a **W**:

```
ADDI  IW,Rd,W
```

If you use the **W** parameter and the value is outside the legal range, the assembler discards all but the 16 LSBs and issues an appropriate warning message.

You can use the ADDI instruction with the ADDC instruction to perform multiple-precision arithmetic.

Machine States 2,8

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a carry, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before	After	
	A0	**NCZV**	**A0**
`ADDI 1,A0`	FFFFFFFFh	0110	00000000h
`ADDI 2,A0`	FFFFFFFFh	0100	00000001h
`ADDI 1,A0`	7FFFFFFFh	1001	80000000h
`ADDI 2,A0`	00000002h	0000	00000004h
`ADDI 32767,A0`	00000002h	0000	00008001h
`ADDI 0FFFF0010h,A0,W`	FFFFFFF0h	0110	00000000h

Syntax **ADDI** *IL, Rd [,* **L***]*

Execution IL + Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	0	0	1	R	Rd			
16 LSBs of IL															
16 MSBs of IL															

Description This ADDI instruction adds a 32-bit, signed immediate value to the contents of the destination register, and stores the result in the destination register. (The symbol *IL* in the syntax above represents a 32-bit, signed immediate value.)

The assembler uses the long (32-bit) ADDI if it cannot use the short form. You can force the assembler to use the long form by following the register operand with an **L**:

```
ADDI IL,Rd,L
```

Machine States 3,12

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a carry, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before	After	
	A0	**NCZV**	**A0**
`ADDI 0FFFFFFFFh,A0`	FFFFFFFFh	1100	FFFFFFFEh
`ADDI 80000000h,A0`	FFFFFFFFh	0101	7FFFFFFFh
`ADDI 80000000h,A0`	7FFFFFFFh	1000	FFFFFFFFh
`ADDI 32768,A0`	7FFFFFFFh	1001	80007FFFh
`ADDI 2,A0,L`	FFFFFFFFh	0100	00000001h

Syntax **ADDK** *K, Rd*

Execution K + Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	K					R	Rd			

Description ADDK adds a 5-bit constant to the contents of the destination register and stores the result in the destination register. (The symbol *K* in the syntax above represents a 5-bit constant.)

The constant is treated as an unsigned number in the range 1–32; if the original value of K=32, then K is converted to 0 in the opcode. The assembler issues an error if you try to add 0 to a register.

You can use the ADDK instruction with the ADDC instruction to perform multiple-precision arithmetic.

Machine States 1,4

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a carry, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before	After	
	A0	NCZV	A0
ADDK 1,A0	FFFFFFFFh	0110	00000000h
ADDK 2,A0	FFFFFFFFh	0100	00000001h
ADDK 1,A0	7FFFFFFFh	1001	80000000h
ADDK 1,A0	80000000h	1000	80000001h
ADDK 32,A0	80000000h	1000	80000020h
ADDK 32,A0	00000002h	0000	00000022h

Syntax **ADDXY** *Rs, Rd*

Execution Rs**X** + Rd**X** → Rd**X**
Rs**Y** + Rd**Y** → Rd**Y**

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	Rs				R	Rd			

Description ADDXY adds the signed source X value to the signed destination X value, adds the signed source Y value to the signed destination Y value, and stores the result in the destination register. The source and destination registers are treated as if they contained separate X and Y values. Any carry out from the lower (X) half of the register does not propagate into the upper (Y) half.

If you only want to add the X halves together, then one of the Y values must be 0 (the method for adding the Y halves is similar).

You can use this instruction to manipulate XY addresses in the register file; ADDXY is also useful for incremental figure drawing.

Rs and Rd must be in the same register file.

Machine States 1,4

Status Bits
N *1* if resulting X field is all 0s, *0* otherwise
C The sign bit of the Y half of the result
Z *1* if Y field is all 0s, *0* otherwise
V The sign bit of the X half of the result

Examples

Code	Before		After	
	A1	A0	A0	NCZV
ADDXY A1,A0	00000000h	00000000h	00000000h	1010
ADDXY A1,A0	00000000h	00000001h	00000001h	0010
ADDXY A1,A0	00000000h	00010000h	00010000h	1000
ADDXY A1,A0	00000000h	00010001h	00010001h	0000
ADDXY A1,A0	0000FFFFh	00000001h	00000000h	1010
ADDXY A1,A0	0000FFFFh	00010001h	00010000h	1000
ADDXY A1,A0	0000FFFFh	00000002h	00000001h	0010
ADDXY A1,A0	0000FFFFh	00010002h	00010001h	0000
ADDXY A1,A0	FFFF0000h	00010000h	00000000h	1010
ADDXY A1,A0	FFFF0000h	00010001h	00000001h	0010
ADDXY A1,A0	FFFF0000h	00020000h	00010000h	1000
ADDXY A1,A0	FFFF0000h	00020001h	00010001h	0000
ADDXY A1,A0	FFFFFFFFh	00010001h	00000000h	1010
ADDXY A1,A0	FFFFFFFFh	00010002h	00000001h	0010
ADDXY A1,A0	FFFFFFFFh	00020001h	00010000h	1000
ADDXY A1,A0	FFFFFFFFh	00020002h	00010001h	0000

Syntax **AND** *Rs, Rd*

Execution Rs AND Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	Rs				R	Rd			

Description AND bitwise-ANDs the contents of the source register with the contents of the destination register and stores the result in the destination register. Rs and Rd must be in the same register file.

Machine States 1,4

Status Bits
N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code		Before	After	
	A1	**A0**	**NCZV**	**A0**
`AND A1,A0`	FFFFFFFFh	FFFFFFFFh	xx0x	FFFFFFFFh
`AND A1,A0`	FFFFFFFFh	00000000h	xx1x	00000000h
`AND A1,A0`	00000000h	00000000h	xx1x	00000000h
`AND A1,A0`	AAAAAAAAh	55555555h	xx1x	00000000h
`AND A1,A0`	AAAAAAAAh	AAAAAAAAh	xx0x	AAAAAAAAh
`AND A1,A0`	55555555h	55555555h	xx0x	55555555h
`AND A1,A0`	55555555h	AAAAAAAAh	xx1x	00000000h

Syntax **ANDI** *IL, Rd*

Execution IL AND Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	1	0	0	R	Rd			
1s complement of 16 LSBs of IL															
1s complement of 16 MSBs of IL															

Description ANDI bitwise-ANDs the value of a 32-bit immediate value with the contents of the destination register, and stores the result in the destination register. (The symbol *IL* in the syntax above represents a 32-bit immediate value.)

This is an alternate mnemonic for ANDNI *IL,Rd*. Note that the assembler stores the 1s complement of IL in the two extension words.

Machine States 3,12

Status Bits

N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before	After	
	A0	**NCZV**	**A0**
`ANDI 0FFFFFFFFh,A0`	FFFFFFFFh	xx0x	FFFFFFFFh
`ANDI 0FFFFFFFFh,A0`	00000000h	xx1x	00000000h
`ANDI 00000000h,A0`	00000000h	xx1x	00000000h
`ANDI 0AAAAAAAAh,A0`	55555555h	xx1x	00000000h
`ANDI 0AAAAAAAAh,A0`	AAAAAAAAh	xx0x	AAAAAAAAh
`ANDI 55555555h,A0`	55555555h	xx0x	55555555h
`ANDI 55555555h,A0`	AAAAAAAAh	xx1x	00000000h

Syntax **ANDN** *Rs, Rd*

Execution (NOT Rs) AND Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	1	Rs				R	Rd			

Description ANDN bitwise-ANDs the 1s complement of the contents of Rs with the contents of Rd, and stores the result in the destination register.

Rs and Rd must be in the same register file. Note that `ANDN Rn,Rn` has the same effect as `CLR Rn`.

Machine States 1,4

Status Bits

N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before		After	
	A1	**A0**	**NC ZV**	**A0**
`ANDN A1,A0`	FFFFFFFFh	FFFFFFFFh	xx 1x	00000000h
`ANDN A1,A0`	FFFFFFFFh	00000000h	xx 1x	00000000h
`ANDN A1,A0`	00000000h	00000000h	xx 1x	00000000h
`ANDN A1,A0`	AAAAAAAAh	55555555h	xx 0x	55555555h
`ANDN A1,A0`	AAAAAAAAh	AAAAAAAAh	xx 1x	00000000h
`ANDN A1,A0`	55555555h	55555555h	xx 1x	00000000h
`ANDN A1,A0`	55555555h	AAAAAAAAh	xx 0x	AAAAAAAAh

Syntax **ANDNI** *IL, Rd*

Execution (NOT IL) AND Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	1	0	0	R	Rd			
16 LSBs of IL															
16 MSBs of IL															

Description ANDNI bitwise-ANDs the 1s complement of a 32-bit immediate value with the contents of the destination register, and stores the result in the destination register. (The symbol *IL* in the syntax above represents a 32-bit immediate value.) ANDI also uses this opcode.

Machine States 3,12

Status Bits

N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before	After	
	A0	**NCZV**	**A0**
`ANDNI 0FFFFFFFFh,A0`	FFFFFFFFh	xx1x	00000000h
`ANDNI 0FFFFFFFFh,A0`	00000000h	xx1x	00000000h
`ANDNI 00000000h,A0`	00000000h	xx1x	00000000h
`ANDNI 0AAAAAAAAh,A0`	55555555h	xx0x	55555555h
`ANDNI 0AAAAAAAAh,A0`	AAAAAAAAh	xx1x	00000000h
`ANDNI 55555555h,A0`	55555555h	xx1x	00000000h
`ANDNI 55555555h,A0`	AAAAAAAAh	xx0x	AAAAAAAAh

Syntax **BTST** *K, Rd*

Execution Set status on value of bit K in Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	~K					R	Rd			

Description BTST tests a bit in the destination register bit and sets status bit Z accordingly. This form of the BTST instruction uses a 5-bit constant to specify the bit in Rd that is tested (the symbol *K* in the syntax above represents a 5-bit constant). The K value must be an absolute expression that evaluates to a number in the range 0 to 31; if the value is greater than 31, the assembler issues a warning and truncates the K operand value to the five LSBs.

Note that the assembler 1s-complements the value of K before inserting it into the opcode.

Machine States 1,4

Status Bits

N Unaffected
C Unaffected
Z *1* if the bit tested is 0, *0* if the bit tested is 1.
V Unaffected

Examples

Code	Before	After
	A0	**NCZV**
`BTST 0,A0`	55555555h	xx0x
`BTST 15,A0`	55555555h	xx1x
`BTST 31,A0`	55555555h	xx1x
`BTST 0,A0`	AAAAAAAAh	xx1x
`BTST 15,A0`	AAAAAAAAh	xx0x
`BTST 31,A0`	AAAAAAAAh	xx0x
`BTST 0,A0`	FFFFFFFFh	xx0x
`BTST 15,A0`	FFFFFFFFh	xx0x
`BTST 31,A0`	FFFFFFFFh	xx0x
`BTST 0,A0`	00000000h	xx1x
`BTST 15,A0`	00000000h	xx1x
`BTST 31,A0`	00000000h	xx1x

Syntax **BTST** *Rs, Rd*

Execution Set status on value of specified bit in Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	Rs				R	Rd			

Description BTST tests a bit in the destination register bit and sets status bit Z accordingly. This form of the BTST instruction uses the 5 LSBs of the source register to specify the bit in Rd that is tested (the symbol *Rs* in the syntax above represents the source register). Note that the 27 MSBs of Rs are ignored.

Rs and Rd must be in the same register file.

Machine States 2,5

Status Bits

N Unaffected
C Unaffected
Z *1* if the bit tested is 0, *0* if the bit tested is 1
V Unaffected

Examples

Code	Before		After
	A1	**A0**	**NCZV**
`BTST A1,A0`	00000000h	55555555h	xx0x
`BTST A1,A0`	0000000Fh	55555555h	xx1x
`BTST A1,A0`	0000001Fh	55555555h	xx1x
`BTST A1,A0`	00000000h	AAAAAAAAh	xx1x
`BTST A1,A0`	0000000Fh	AAAAAAAAh	xx0x
`BTST A1,A0`	0000001Fh	AAAAAAAAh	xx0x
`BTST A1,A0`	FFFFFF8Fh	FFFF7FFFh	xx0x
`BTST A1,A0`	00000000h	FFFFFFFFh	xx0x
`BTST A1,A0`	0000000Fh	FFFFFFFFh	xx0x
`BTST A1,A0`	0000001Fh	FFFFFFFFh	xx0x
`BTST A1,A0`	00000000h	00000000h	xx1x
`BTST A1,A0`	0000000Fh	00000000h	xx1x
`BTST A1,A0`	0000001Fh	00000000h	xx1x

Syntax **CALL** *Rs*

Execution
PC' → TOS
Rs → PC
SP - 32 → SP

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	0	0	1	R	Rs			

Description CALL pushes the address of the next instruction (PC') onto the stack, then jumps to a subroutine whose address is contained in the source register. You can use this instruction for indexed subroutine calls. Note that when Rs is the SP, Rs is decremented **after** being written to the PC (the PC contains the original value of Rs).

The TMS34010 always sets the four LSBs of the program counter to 0, so instructions are always word aligned.

The stack pointer (SP) points to the top of the stack; the stack is located in external memory. The stack grows in the direction of decreasing linear addresses. PC' is pushed onto the stack and the SP is predecremented by 32 before the return address is loaded onto the stack. Stack pointer alignment affects timing as indicated in **Machine States**, below.

Use the RETS instruction to return from a subroutine.

Machine States
3+(3),9 (SP aligned)
3+(9),15 (SP nonaligned)

Status Bits
N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Example

```
CALL A0
```

Before			After	
A0	**PC**	**SP**	**PC**	**SP**
01234560h	04442210h	F000020h	01234560h	F000000h

Memory contains the following values after instruction execution:

Address	Data
F000010h	2220h
F000020h	0444h

Syntax **CALLA** *Address*

Execution
PC' → TOS
Address → PC

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	0	1	0	1	1	1	1	1
16 LSBs of Address															
16 MSBs of Address															

Description
CALLA pushes the address of the next instruction (PC') onto the stack, then jumps to the address contained in the two extension words. The *Address* operand is a 32-bit absolute address. This instruction is used for long (greater than ±32K words) or externally referenced calls.

The lower four bits of the program counter are always set to 0, so instructions are always word-aligned.

The stack pointer (SP) points to the top of the stack; the stack is located in external memory. The stack grows in the direction of decreasing linear address. PC' is pushed onto the stack and the SP is predecremented by 32 before the return address is loaded onto the stack. Stack pointer alignment affects timing as indicated in **Machine States**, below.

Use the RETS instruction to return from a subroutine.

Machine States
4+(2),15 (SP aligned)
4+(8),21 (SP nonaligned)

Status Bits
N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Example

```
CALLA 01234560h
```

Before		After	
PC	**SP**	**PC**	**SP**
04442210h	0F000020h	01234560h	0F000000h

Memory contains the following values after instruction execution:

Address	Data
F000010h	2240h
F000020h	0444h

Syntax **CALLR** *Address*

Execution
PC' → TOS
PC' + (offset×16) → PC

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	0	0	1	1	1	1	1	1
offset															

Description

CALLR pushes the address of the next instruction (PC') onto the stack, then jumps to the subroutine at the address specified by the sum of the next instruction address and the signed word offset. This instruction is used for calls within a specified module or section.

The *Address* operand is a 32-bit address within ±32K words (-32,768 to 32,767) of the PC. The address must be defined within the current section; the assembler does not accept an address value that is externally defined or defined within a different section than PC'. The assembler calculates the offset value for the opcode as (Address - PC')/16.

The lower four bits of the program counter are always set to 0, so instructions are always word aligned.

The stack pointer (SP) points to the top of the stack; the stack is located in external memory. The stack grows in the direction of decreasing linear address. The PC is pushed on to the stack and the SP is predecremented by 32 before the return address is loaded onto the stack. Stack pointer alignment affects timing as indicated in **Machine States**, below.

Use the RETS instruction to return from a subroutine.

Machine States
3+(2),11 (SP aligned)
3+(8),17 (SP nonaligned)

Status Bits
N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before		After	
	PC	**SP**	**PC**	**SP**
`CALLR 0447FFF0h`	04400000h	0F000020h	0447FFF0h	0F000000h
`CALLR 04480000h`	04400000h	0F000020h	04480000h	0F000000h

Memory contains the following values after instruction execution:

Address	**Data**
F000010h	0000h
F000020h	0440h

Syntax **CLR** *Rd*

Execution Rd XOR Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	Rd				R	Rd			

Description CLR clears the destination register by XORing the contents of the register with itself. This is an alternate mnemonic for XOR *Rd,Rd*.

Machine States 1,4

Status Bits
N Unaffected
C Unaffected
Z *1*
V Unaffected

Examples

Code	Before	After	
	A0	**A0**	**NCZV**
CLR A0	FFFFFFFFh	00000000h	xx1x
CLR A0	00000001h	00000000h	xx1x
CLR A0	80000000h	00000000h	xx1x
CLR A0	AAAAAAAAh	00000000h	xx1x

Syntax **CLRC**

Execution 0 → C

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0

Description CLRC sets the C (carry) bit in the status register to 0; the rest of the status register is unaffected. (Note that the SETC instruction *sets* the C bit.)

This instruction is useful for returning a true/false value (in the carry bit) from a subroutine without using a general-purpose register.

Machine States 1,4

Status Bits

N Unaffected
C *0*
Z Unaffected
V Unaffected

Examples

Code	Before		After	
	ST	NCZV	ST	NCZV
CLRC	F0000000h	1111	B0000000h	1011
CLRC	40000010h	0100	00000010h	0000
CLRC	B000001Fh	1011	B000001Fh	1011

Syntax **CMP** *Rs, Rd*

Execution Set status bits on the result of Rd - Rs

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	Rs				R	Rd			

Description CMP sets the status bits on the result of subtracting the contents of Rs from the contents of Rd. This is a nondestructive compare; the contents of the registers are not affected. This instruction is often used in conjunction with the JA*cc* or JR*cc* conditional jump instructions.

Rs and Rd must be in the same register file.

Machine States 1,4

Status Bits
N *1* if the result is negative, *0* otherwise
C *1* if a there is a borrow, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before		After	Jumps Taken
	A1	A0	NCZV	
CMP A1,A0	00000001h	00000001h	0010	UC,NN,NC,Z,NV,LS,GE,LE,HS
CMP A1,A0	00000001h	00000002h	0000	UC,NN,NC,NZ,NV,P,HI,GE,GT,HS
CMP A1,A0	00000001h	FFFFFFFFh	1000	UC,N,NC,NZ,NV,P,HI,LT,LE,HS
CMP A1,A0	00000001h	80000000h	0001	UC,NN,NC,NZ,V,HI,LT,LE,HS
CMP A1,A0	FFFFFFFFh	7FFFFFFFh	1101	UC,N,C,NZ,V,LS,GE,GT,LO
CMP A1,A0	FFFFFFFFh	80000000h	1100	UC,N,C,NZ,NV,LS,LT,LE,LO
CMP A1,A0	80000000h	7FFFFFFFh	1101	UC,N,C,NZ,V,LS,GE,GT,LO

Syntax **CMPI** */W, Rd [, W]*

Execution Set status bits on the result of Rd - IW

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	0	1	0	R	Rd			
1s complement of IW															

Description CMPI sets the status bits on the result of subtracting a 16-bit, sign-extended immediate value from the contents of the destination register. (The symbol */W* in the syntax above represents a 16-bit, signed immediate value.) This is a nondestructive compare; the contents of the destination register are not affected. This instruction is often used in conjunction with the JA*cc* or JR*cc* conditional jump instructions.

Note that the assembler inserts the 1s complement of the 16-bit value into the second instruction word.

The assembler uses the short form of the CMPI instruction if the immediate value is previously defined and is in the range -32,768 to 32,767. You can force the assembler to use the short form by following the register operand with **W**:

```
CMPI  IW,Rd,W
```

The assembler truncates the upper bits and issues an appropriate warning message if the value is greater than 16 bits.

Machine States 2,8

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a borrow, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before	After	Jumps Taken
	A0	NCZV	
CMPI 1,A0	00000002h	0000	UC,NN,NC,NZ,NV,P,HI,GE,GT,HS
CMPI 1,A0	00000001h	0010	UC,NN,NC,Z,NV,LS,GE,LE,HS
CMPI 1,A0	00000000h	1100	UC,N,C,NZ,NV,LS,LT,LE,LO
CMPI 1,A0	FFFFFFFFh	1000	UC,N,NC,NZ,NV,P,HI,LT,LE,HS
CMPI 1,A0	80000000h	0001	UC,NN,NC,NZ,V,HI,LT,LE,HS
CMPI -2,A0	00000000h	0100	UC,NN,C,NZ,NV,P,LS,GE,GT,LO
CMPI -2,A0	FFFFFFFFh	0000	UC,NN,NC,NZ,NV,P,LI,GE,GT,HS
CMPI -2,A0	FFFFFFFEh	0010	UC,NN,NC,Z,NV,LS,GE,LE,HS
CMPI -2,A0	FFFFFFFDh	1100	UC,N,C,NZ,NV,LS,LT,LE,LO
CMPI -1,A0	7FFFFFFFh	1101	UC,N,C,NZ,V,LS,GE,GT,LO

Syntax **CMPI** *IL, Rd [, L]*

Execution Set status bits on the result of Rd - IL

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	0	1	1	R	Rd			
1s complement of 16 LSBs of IL															
1s complement of 16 MSBs of IL															

Description CMPI sets the status bits on the result of subtracting a 32-bit, signed immediate value from the contents of the destination register. (The *IL* symbol in the syntax above represents a 32-bit, signed immediate value.) This is a nondestructive compare; the contents of the destination register are not affected.

Note that the assembler inserts the 1s complement of the 16 LSBs of the value into the second instruction word, and inserts the 1s complement of the 16 MSBs of the value into the third instruction word.

The assembler uses this form of the CMPI instruction if it cannot use the short form. You can force the assembler to use the long form by following the register operand with an **L**:

```
CMPI  IL,Rd,L
```

This instruction is often used in conjunction with the JA*cc* or JR*cc* conditional jump instructions.

Machine States 3,12

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a borrow, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before	After	Jumps Taken
	A0	NCZV	
CMPI 8000h,A0	00008001h	0000	UC,NN,NC,NZ,NV,P,HI,GE,GT,HS
CMPI 8000h,A0	00008000h	0010	UC,NN,NC,Z,NV,LS,GE,LE,HS
CMPI 8000h,A0	00007FFFh	1100	UC,N,C,NZ,NV,LS,LT,LE,LO
CMPI 8000h,A0	FFFFFFFFh	1000	UC,N,NC,NZ,NV,P,HI,LT,LE,HS
CMPI 8000h,A0	80007FFFh	0001	UC,NN,NC,NZ,V,HI,LT,LE,HS
CMPI 0FFFF7FFFh,A0	00000000h	0100	UC,NN,C,NZ,NV,P,LS,GE,GT,LO
CMPI 0FFFF7FFEh,A0	FFFF7FFFh	0000	UC,NN,NC,NZ,NV,P,HI,GE,GT,HS
CMPI 0FFFF7FFEh,A0	FFFF7FFEh	0010	UC,NN,NC,Z,NV,LS,GE,LE,HS
CMPI 0FFFF7FFEh,A0	FFFF7FFDh	1100	UC,N,C,NZ,NV,LS,LT,LE,LO
CMPI 0FFFF7FFFh,A0	7FFF7FFFh	1101	UC,N,C,NZ,V,LS,GE,GT,LO

Syntax **CMPXY** *Rs, Rd*

Execution Set status bits on the results of:

RdX - RsX
RdY - RsY

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	Rs				R	Rd			

Description CMPXY compares the source register to the destination register in XY mode and sets the status bits as if a subtraction had been performed. This is a nondestructive compare; the contents of the register are not affected. The source and destination registers are treated as signed XY registers. Note that no overflow detection is provided.

Rs and Rd must be in the same register file.

Machine States 1,4

Status Bits

N *1* if source X field = destination X field, *0* otherwise
C Sign bit of Y half of the result
Z *1* if source Y field = destination Y field, *0* otherwise
V Sign bit of X half of the result

Examples

Code	Before		After	Jumps Taken
	A1	A0	NCZV	
CMPXY A1,A0	00090009h	00010001h	0101	NN,C,NZ,V,LS,LT
CMPXY A1,A0	00090009h	00090001h	0011	NN,NC,Z,V,LS,LT
CMPXY A1,A0	00090009h	00010009h	1100	N,C,NZ,NV,LS,LT
CMPXY A1,A0	00090009h	00090009h	1010	N,NC,Z,NV,LS,LT
CMPXY A1,A0	00090009h	00000010h	0100	NN,C,NZ,NV,LS,GE
CMPXY A1,A0	00090009h	00090010h	0010	NN,NC,Z,NV,LS,GE
CMPXY A1,A0	00090009h	00100000h	0001	NN,NC,NZ,V,HI,LT
CMPXY A1,A0	00090009h	00100009h	1000	N,NC,NZ,NV,HI,LT
CMPXY A1,A0	00090009h	00100010h	0000	NN,NC,NZ,NV,HI,GE

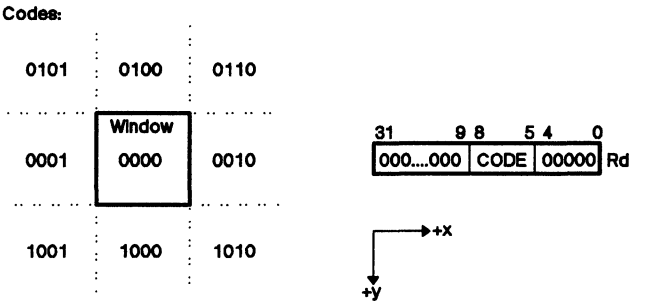
Syntax **CPW** *Rs, Rd*

Execution point code → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	1	Rs				R	Rd			

Description CPW compares a point represented by an XY value in the source register to the window limits in the WSTART and WEND registers. The contents of the source register are treated as an XY address that consists of 16-bit signed X and Y values. WSTART and WEND are also treated as signed XY-format registers. WSTART and WEND must contain positive values; negative values produce unpredictable results. The location of the point with respect to the window is encoded as shown below; the code is loaded into the destination register.



The following list describes the contents of the destination register after CPW execution:

Bit Postion: Contents:

0–4 0s
5 1 if WSTART.X > Rs.X, 0 otherwise
6 1 if Rs.X > WEND.X, 0 otherwise
7 1 if WSTART.Y > Rs.Y, 0 otherwise
8 1 if Rs.Y > WEND.Y, 0 otherwise
9–31 0s

This instruction can also be used to trivially reject lines that do not intersect with a window. The CPW codes for the two points defining the line are ANDed together. If the result is nonzero, then the line must lie completely outside the window (and does not intersect it). A 0 result indicates that the line *may* intersect the window, and a more rigorous test must be applied.

Rs and Rd must be in the same register file.

Implied Operands

B File Registers			
Register	**Name**	**Format**	**Description**
B5	WSTART	XY	Window start. Defines inclusive starting corner of window (lesser value corner).
B6	WEND	XY	Window end. Defines inclusive ending corner of window (greater value corner).

Machine States 1,4

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V *1* if point lies outside window, *0* otherwise

Examples You must select appropriate implied operand values before executing the CPW instruction. In this example, the implied operands are set up as follows, specifying a block of 36 pixels.

WSTART = 5,5
WEND = A,A

```
CPW A1,A0
```

Before		After	
A1	**NCZV**	**A0**	**NCZV**
00040004h	xxx0	000000A0h	xxx1
00040005h	xxx0	00000080h	xxx1
0004000Ah	xxx0	00000080h	xxx1
0004000Bh	xxx1	000000C0h	xxx1
00050004h	xxx1	00000020h	xxx1
00050005h	xxx0	00000000h	xxx0
0005000Ah	xxx0	00000000h	xxx0
0005000Bh	xxx0	00000040h	xxx1
000A0004h	xxx0	00000020h	xxx1
000A0005h	xxx1	00000000h	xxx0
000A000Ah	xxx1	00000000h	xxx0
000A000Bh	xxx0	00000040h	xxx1
000B0004h	xxx0	00000120h	xxx1
000B0005h	xxx0	00000100h	xxx1
000B000Ah	xxx0	00000100h	xxx1
000B000Bh	xxx0	00000140h	xxx1

Syntax **CVXYL** *Rs, Rd*

Execution Rs**XY** → linear address in Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	Rs				R	Rd			

Description CVXYL converts an XY address to a linear address:

- The source register contains an XY address. The signed X value occupies the 16 LSBs of the register and the signed Y value occupies the 16 MSBs. The X value must be positive.
- The XY address is converted into a 32-bit linear address which is stored in the destination register.

The following conversion formula is used:

Address = [(Y × Display Pitch) OR (X × Pixel Size)] + Offset

Since the TMS34010 restricts the screen pitch and pixel size to powers of two (for XY addressing), the multiply operations in this conversion are actually shifts. The offset value is in the OFFSET register. The CONVDP value is used to determine the shift amount for the Y value, while the PSIZE register determines the X shift amount.

Rs and Rd must be in the same register file.

Implied Operands

B File Registers			
Register	**Name**	**Format**	**Description**
B3	DPTCH	Linear	Destination pitch
B4	OFFSET	Linear	Screen origin (location 0,0)
I/O Registers			
Address	**Name**	**Description and Elements (Bits)**	
C0000140h	CONVDP	XY-to-linear conversion (destination pitch)	
C0000150h	PSIZE	Pixel size (1,2,4,8,16)	

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the CVXYL instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an instruction that reads the same location (such as a MOVE SAddress,Rd instruction). For more information, refer to Section 6.2, Latency of Writes to I/O Registers.

Machine States 3,6

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before	After			
	A0	OFFSET	PSIZE	CONVDP	A1
CVXYL A0,A1	00400030h	00000000h	0010h	0014h	00020300h
CVXYL A0,A1	00400030h	00000000h	0008h	0014h	00020180h
CVXYL A0,A1	00400030h	00000000h	0004h	0014h	00020000h
CVXYL A0,A1	00400030h	00008000h	0004h	0014h	00028000h
CVXYL A0,A1	00400030h	0F000000h	0004h	0014h	0F020000h
CVXYL A0,A1	00400030h	00000000h	0002h	0014h	00020060h
CVXYL A0,A1	00400030h	00000000h	0001h	0014h	00020030h
CVXYL A0,A1	00400030h	00000000h	0001h	0013h	00040030h
CVXYL A0,A1	00400030h	00000000h	0001h	0015h	00010000h

CONVDP = 0013h corresponds to DPTCH = 00001000h
CONVDP = 0014h corresponds to DPTCH = 00000800h
CONVDP = 0015h corresponds to DPTCH = 00000400h

Syntax **DEC** *Rd*

Execution Rd - 1 → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	0	0	0	1	R	Rd			

Description DEC subtracts 1 from the contents of the destination register and stores the result in the destination register. This instruction is an alternate mnemonic for `SUBK 1,Rd`.

You can use the DEC instruction with the SUBB instruction to perform multiple-precision arithmetic.

Machine States 1,4

Status Bits
N *1* if the result is negative, *0* otherwise
C *1* if there is a borrow, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before	After	
	A1	A1	NCZV
`DEC A1`	00000010h	0000000Fh	0000
`DEC A1`	00000001h	00000000h	0010
`DEC A1`	00000000h	FFFFFFFFh	1100
`DEC A1`	FFFFFFFFh	FFFFFFFEh	1000
`DEC A1`	80000000h	7FFFFFFFh	0001

Syntax **DINT**

Execution 0 → IE

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	0	1	1	0	0	0	0	0

Description DINT disables interrupts by setting the global interrupt enable bit (IE, status bit 21) to 0. All interrupts except reset and NMI are disabled; the interrupt enable mask in the INTENB register is ignored. The remainder of the status register is unaffected.

The EINT instruction enables interrupts.

Machine States 3,6

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected
IE *0*

Examples

Code	Before	After
	ST	**ST**
DINT	00000010h	00000010h
DINT	00200010h	00000010h

Syntax **DIVS** *Rs, Rd*

Execution Rd Even: Rd:Rd+1/Rs → Rd, remainder → Rd+1
Rd Odd: Rd/Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	Rs				R	Rd			

Description DIVS performs a signed 32-bit or 64-bit divide. The source register contains the 32-bit signed divisor. The destination register contains a 32-bit signed dividend or the most significant half of a 64-bit signed dividend, depending on whether Rd is an odd register (for example, A1 or B3) or an even register (for example, A8 or B2):

Rd Even DIVS performs a signed divide of the 64-bit operand contained in the two consecutive registers, starting at the specified destination register, by the 32-bit contents of the source register. The specified even-numbered destination register, Rd, contains the 32 MSBs of the dividend. The next consecutive register (which is odd-numbered) contains the 32 LSBs of the dividend. The quotient is stored in the destination register, and the remainder is stored in the following register (Rd+1). The remainder is always the same sign as the dividend (in Rd:Rd+1). Avoid using A14 or B14 as the destination register, since this overwrites the SP; the assembler issues a warning in this case.

Rd Odd DIVS performs a signed divide of the 32-bit operand contained in the destination register by the 32-bit value in the source register. The quotient is stored in the destination register; the remainder is not returned.

Rs and Rd must be in the same register file.

Machine States

	Rd Odd	**Rd Even**
Normal	39,42	40,43
Result = 80000000h	41,44	41,44
Rs = 0	7,10	7,10
Rd ≥ Rs	treated as normal	7,10

Status Bits **N** *0* **if:**

- Rs = 0, **or**
- Rd is even and Rd ≥ Rs, **or**
- Quotient is nonnegative.

1 **if:**

- Result = 80000000h **or**
- Quotient is negative.

C Unaffected

Z *0* **if:**

- Rs = 0, **or**
- Rd is even and Rd ≥ Rs, **or**

- Result = 80000000h, **or**
- Quotient ≠ 0.

1 **if:**

- Quotient = 0.

V *1* if quotient overflows (cannot be represented by 32 bits), *0* otherwise
The following conditions cause an overflow and set the overflow flag:

- Divisor (Rs) is 0
- Quotient cannot be contained within 32 bits

Example 1 This example divides the contents of register A0 by the contents of register A2, and stores the result in register A0. Note that the contents of register A2 are not affected by instruction execution.

```
DIVS  A2,A0
```

Before

A0	A1	A2
12345678h	87654321h	87654321h
EDCBA987h	789ABCDFh	87654321h
EDCBA987h	789ABCDFh	789ABCDFh
12345678h	87654321h	789ABCDFh
12345678h	87654321h	00000000h
00000000h	00000000h	00000000h
00000000h	00000000h	87654321h
87654321h	00000000h	87654321h

After

A0	A1	A2	NCZV
D95BC60Ah	15CA1DD7h	87654321h	1x00
26A439F6h	EA35E229h	87654321h	0x00
D95BC60Ah	EA35E229h	789ABCDFh	1x00
26A439F6h	15CA1DD7h	789ABCDFh	0x00
12345678h	87654321h	00000000h	0x01
00000000h	00000000h	00000000h	0x01
00000000h	00000000h	87654321h	0x10
87654321h	00000000h	87654321h	0x01

Example 2 This example divides the contents of register A1 by the contents of of register A2, and stores the result in register A0. Note that the contents of register A2 are not affected by instruction execution.

```
DIVS A2,A1
```

Before

A0	A1	A2
00000000h	87654321h	12345678h
00000000h	87654321h	0EDCBA988h
00000000h	789ABCDFh	0EDCBA988h
00000000h	789ABCDFh	12345678h
00000000h	87654321h	00000000h
00000000h	00000000h	00000000h

After

A0	A1	A2	NCZV
00000000h	FFFFFFFAh	12345678h	1x00
00000000h	00000006h	EDCBA988h	0x00
00000000h	FFFFFFFAh	EDCBA988h	1x00
00000000h	00000006h	12345678h	0x00
00000000h	87654321h	00000000h	0x01
00000000h	00000000h	00000000h	0x01

Syntax **DIVU** *Rs, Rd*

Execution Rd Even: Rd:Rd+1/Rs → Rd, remainder → Rd+1
Rd Odd: Rd/Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	1	Rs				R	Rd			

Description DIVU performs an unsigned 32-bit or 64-bit divide. The source register contains the 32-bit divisor. The destination register contains a 32-bit dividend or the most significant half of a 64-bit dividend, depending on whether Rd is an odd register (for example, A1 or B3) or an even register (for example, A8 or B2):

Rd Even DIVU performs an unsigned divide of the 64-bit operand contained in the two consecutive registers, starting at the destination register, by the 32-bit contents of the source register. The specified even-numbered destination register, Rd, contains the 32 MSBs of the dividend. The next consecutive register (which is odd-numbered) contains the 32 LSBs of the dividend. The quotient is stored in the destination register, and the remainder is stored in the following register (Rd+1). Avoid using A14 or B14 as the destination register, since this overwrites the SP; the assembler issues a warning in this case.

Rd Odd DIVU performs an unsigned divide of the 32-bit operand contained in the destination register by the 32-bit value in the source register. The quotient is stored in the destination register; the remainder is not returned.

Rs and Rd must be in the same register file.

Machine States

	Rd Odd	**Rd Even**
Normal	37,40	37,40
Rs = 0	5,8,	5,8
Rd ≥ Rs	treated as normal	5,8

Status Bits

N Unaffected

C Unaffected

Z *0* **if:**

- Rs = 0, **or**
- Rd is even and Rd ≥ Rs, **or**
- Quotient ≠ 0.

1 **if:**

- Quotient = 0.

V *1* if quotient overflows (cannot be represented by 32 bits), *0* otherwise
The following conditions cause an overflow and set the overflow flag:

- Divisor (Rs) is 0

- Quotient cannot be contained within 32 bits

Example 1 This instruction divides the contents of register A0 by the contents of register A2, and stores the unsigned result in register A0. Note that the contents of register A2 are not affected by instruction execution.

```
DIVU  A2,A0
```

Before

A0	A1	A2
12345678h	87654321h	789ABCDFh
12345678h	87654321h	00000000h
00000000h	00000000h	00000000h
00000000h	00000000h	87654321h
87654321h	00000000h	87654321h

After

A0	A1	A2	NCZV
26A439F6h	15CA1DD7h	789ABCDFh	xx00
12345678h	87654321h	00000000h	xx01
00000000h	00000000h	00000000h	xx01
00000000h	00000000h	87654321h	xx10
87654321h	00000000h	87654321h	xx01

Example 2 This instruction divides the contents of register A1 by the contents of register A2, and stores the unsigned result in register A1. Note that the contents of register A2 are not affected by instruction execution.

```
DIVU  A2,A1
```

Before

A0	A1	A2
00000000h	789ABCDFh	12345678h
00000000h	12345678h	00000000h
00000000h	00000000h	00000000h
00000000h	00000000h	87654321h
00000000h	87654321h	87654321h

After

A0	A1	A2	NCZV
00000000h	00000006h	12345678h	xx00
00000000h	12345678h	00000000h	xx01
00000000h	00000000h	00000000h	xx01
00000000h	00000000h	87654321h	xx10
00000000h	00000001h	87654321h	xx00

Syntax **DRAV** *Rs, Rd*

Execution

COLOR1 pixels → *Rd
Rs**X** + Rd**X** → Rd**X**
Rs**Y** + Rd**Y** → Rd**Y**

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	Rs				R	Rd			

Description

DRAV writes the pixel value in the COLOR1 register to the location pointed to by the XY address in the destination register. Following the write, the XY address in the destination register is incremented by the value in the source register: the X half of Rs is added to the X half of Rd, and the Y half of Rs is added to the Y half of Rd. Any carry out from the lower (X) half of the register does not propagate into the upper (Y) half.

COLOR1 bits 0–15 are output on data bus lines 0–15, respectively. The pixel data used from COLOR1 is that which aligns to the destination location, so 16-bit patterns can be implemented. Rs and Rd must be in the same register file.

Implied Operands

B File Registers			
Register	**Name**	**Format**	**Description**
B3	DPTCH	Linear	Destination pitch
B4	OFFSET	Linear	Screen origin (location 0,0)
B5	WSTART	XY	Window starting corner
B6	WEND	XY	Window ending corner
B9	COLOR1	Pixel	Pixel color
I/O Registers			
Address	**Name**	**Description and Elements (Bits)**	
C00000B0h	CONTROL	PP – Pixel processing operations (22 options) W – Window checking operation T – Transparency operation	
C0000140h	CONVDP	XY-to-linear conversion (destination pitch)	
C0000150h	PSIZE	Pixel size (1,2,4,8,16)	
C0000160h	PMASK	Plane mask – pixel format	

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the DRAV instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an instruction that reads the same location (such as a MOVE SAddress,Rd instruction). For more information, refer to Section 6.2, Latency of Writes to I/O Registers.

Pixel Processing

Set the PPOP field in the CONTROL register to select a pixel processing operation. This operation is applied to the pixel as it is moved to the destination location. At reset, the default pixel processing operation is *replace* (S → D). For more information, see Section 7.7, Pixel Processing, on page 7-15.

Window Checking Select a window checking mode by setting the W bits in the CONTROL register. If you select an active window checking mode (W = 1, 2, or 3), the WSTART and WEND registers define the XY starting and ending corners of a rectangular window. The X and Y values in both WSTART and WEND must be positive.

When the TMS34010 attempts to write a pixel inside or outside a defined value, the following actions may occur:

W=0 No window operation. The pixel is drawn and the WVP and V bits are unaffected.

W=1 Window hit. No pixels are drawn. The V bit is set to 0 if the pixel lies within the window; otherwise, it is set to 1. The WVP bit is set to 1 if the pixel lies within the window; otherwise, it is not affected.

W=2 Window miss. If the pixel lies outside the window, the WVP and V bits are set to 1 and the instruction is aborted (no pixel is drawn). Otherwise, the pixel is drawn, the V bit is set to 0, and the WVP bit is unaffected.

W=3 Window clip. If the pixel lies outside the window, the V bit is set to 1 and the instruction is aborted (no pixels are drawn). Otherwise, the pixel is drawn and the V bit is set to 0.

For more information, see Section 7.10, Window Checking, on page 7-27.

Transparency You can enable transparency for this instruction by setting the T bit in the CONTROL I/O register to 1. The TMS34010 checks for 0 (transparent) source pixels *after* it processes the source data. At reset, the default case for transparency is *off*.

Plane Mask The plane mask is enabled for this instruction.

Shift Register Transfers When this instruction is executed and the SRT bit is set, normal memory read and write operations become SRT reads and writes. Refer to Section 9.10.2, Video Memory Bulk Initialization, on page 9-28 for more information.

Machine States The states consumed depend on the operation selected, as indicated below.

Pixel Processing Operation								Window Violation		
PSIZE	Replace	Boolean	ADD	ADDS	SUB	SUBS	MIN/MAX	W=1	W=2	W=3
1,2,4,8	4+(3),10	6+(3),12	7+(3),13	7+(3),13	7+(3),13	8+(3),14	7+(3),13	5,8	3,6	5,8
16	4+(1),8	6+(1),10	6+(1),10	7+(1),11	7+(1),11	8+(1),12	7+(1),11	5,8	3,6	5,8

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V *1* if a window violation occurs, *0* otherwise; unaffected if window clipping is not used.

Examples These DRAV examples use the following implied operand setup.

Register File B:
DPTCH (B3) = 200h
OFFSET (B4) = 00010000h
WSTART (B5) = 00100000h
WEND (B6) = 003C0040h
COLOR1 (B9) = FFFFFFFFh

I/O Registers:
CONVDP = 0016h

Assume that memory contains the following values before instruction execution:

Address	Data
00018040h	8888h

Code	Before					After		
	A0	A1	PSIZE	PP	W	PMASK	A0	@18040h
DRAV A1,A0	00400040h	00100010h	0001h	00000	00	0000h	00500050h	8889h
DRAV A1,A0	00400020h	00100010h	0002h	00000	00	0000h	00500030h	888Bh
DRAV A1,A0	00400010	00100010h	0004h	00000	00	0000h	00500020h	888Fh
DRAV A1,A0	00400008	00100010h	0008h	00000	00	0000h	00500018h	88FFh
DRAV A1,A0	00400004	00100010h	0010h	00000	00	0000h	00500014h	FFFFh
DRAV A1,A0	00400004	0000FFFFh	0010h	01010	00	0000h	00400003h	0000h
DRAV A1,A0	00400004	FFFF0000h	0010h	10011	00	0000h	003F0004h	0000h
DRAV A1,A0	00400004	00010001h	0010h	00000	11	0000h	00410005h	0000h
DRAV A1,A0	00400004h	00400004h	0010h	00000	00	00FFh	00800008h	FF00h

Syntax **DSJ** *Rd, Address*

Execution
Rd - 1 → Rd
If Rd ≠ 0, then (offset×16) + PC' → PC
If Rd = 0, then go to next instruction

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	1	0	0	R	Rd			
offset															

Description DSJ decrements the contents of the destination register by 1. Depending on the decremented value of Rd, the TMS34010 either jumps or skips the jump:

- **Rd - 1 ≠ 0**

  The TMS34010 jumps. The current PC points to the instruction word that immediately follows the second word of the DSJ instruction. The signed word offset is converted to a bit offset by multiplying by 16. The new PC address is then obtained by adding the resulting signed offset (offset × 16) to the address of the next instruction.

- **Rd - 1 = 0**

  The TMS34010 skips the jump and continues and program execution with the next sequential instruction.

The *Address* operand is a 32-bit address. The assembler calculates the offset as (Address - PC')/16; this results in a jump range of -32,768 to +32,767 words. (The offset is the second instruction word of the opcode.)

The DSJ instruction is useful for large loops involving a counter. For shorter loops, the assembler translates this into a DSJS instruction.

Machine States
3,9 (Jump)
2,8 (No jump)

Status Bits
N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before	After	
	A5	A5	Jump taken?
`DSJ A5,LOOP`	00000009h	00000008h	Yes
`DSJ A5,LOOP`	00000001h	00000000h	No
`DSJ A5,LOOP`	00000000h	FFFFFFFFh	Yes

Syntax **DSJEQ** *Rd, Address*

Execution
If Z = 1, then Rd - 1 → Rd
If Rd ≠ 0, then PC' + (offset×16) → PC
If Rd = 0, then go to next instruction
If Z = 0, then go to next instruction

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	1	0	1	R	Rd			
offset															

Description The DSJEQ instruction evaluates the status Z bit. Depending on the value of that bit, the TMS34010 either skips the jump, or decrements Rd and then makes a decision to jump or skip the jump:

- **Z = 1**

  The TMS34010 decrements the contents of the destination register by 1.

  - **Rd - 1 ≠ 0**

    The TMS34010 jumps relative to the current PC. The current PC points to the instruction word that immediately follows the second word of the DSJEQ instruction. The signed word offset is converted to a bit offset by multiplying by 16. The new PC address is then obtained by adding the resulting signed offset (offset × 16) to the address of the next instruction.

  - **Rd - 1 = 0**

    The TMS34010 skips the jump and continues program execution at the next sequential instruction.

- **Z = 0**

  The TMS34010 skips the jump and continues program execution at the next sequential instruction.

The *Address* operand is a 32-bit address. The assembler calculates the offset as (Address - PC')/16; this results in a jump range of -32,768 to +32,767 words. (The offset is the second instruction word of the opcode.)

You can use this instruction after an explicit or implicit compare to 0. Additional information on these types of compares can be obtained in the CMP and CMPI, and MOVE-to-register instructions, respectively.

Machine States
3,9 (Jump)
2,8 (No jump)

Status Bits
N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before		After	
	A5	NCZV	A5	Jump taken?
DSJEQ A5,LOOP	00000009h	xx1x	00000008h	Yes
DSJEQ A5,LOOP	00000001h	xx1x	00000000h	No
DSJEQ A5,LOOP	00000000h	xx1x	FFFFFFFFh	Yes
DSJEQ A5,LOOP	00000009h	xx0x	00000009h	No
DSJEQ A5,LOOP	00000001h	xx0x	00000001h	No
DSJEQ A5,LOOP	00000000h	xx0x	00000000h	No

Syntax **DSJNE** *Rd, Address*

Execution
If Z = 0, then Rd - 1 → Rd
If Rd ≠ 0, then PC' + (offset x 16) → PC
If Rd = 0, then go to next instruction
If Z = 1, then to to next instruction

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	1	1	0	R	Rd			
offset															

Description The DSJNE instruction evaluates the status Z bit. Depending on the value of that bit, the TMS34010 either skips the jump, or decrements Rd and then makes a decision to jump or skip the jump:

- **Z = 0,**

  The TMS34010 decrements the contents of the destination register by 1.

  - **Rd - 1 ≠ 0**

    The TMS34010 jumps relative to the current PC. The current PC points to the instruction word that immediately follows the second word of the DSJNE instruction. The signed word offset is converted to a bit offset by multiplying by 16. The new PC address is then obtained by adding the resulting signed offset (offset x 16) to the address of the next instruction.

  - **Rd - 1 = 0**

    The TMS34010 skips the jump and continues program execution at the next sequential instruction.

- **Z = 1**

  The TMS34010 skips the jump and continues program execution at the next sequential instruction.

The *Address* operand is a 32-bit address. The assembler calculates the offset as (Address - PC')/16; this results in a jump range of -32,768 to +32,767 words. (The offset is the second instruction word of the opcode.)

You can use this instruction after an explicit or implicit compare to 0. Additional information on these types of compares can be obtained in the CMP, CMPI, and MOVE-to-register instructions.

Machine States
3,9 (Jump)
2,8 (No jump)

Status Bits
N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before		After	
	A5	NCZV	A5	Jump taken?
DSJNE A5,LOOP	00000009h	xx1x	00000009h	No
DSJNE A5,LOOP	00000001h	xx1x	00000001h	No
DSJNE A5,LOOP	00000000h	xx1x	00000000h	No
DSJNE A5,LOOP	00000009h	xx0x	00000008h	Yes
DSJNE A5,LOOP	00000001h	xx0x	00000000h	No
DSJNE A5,LOOP	00000000h	xx0x	FFFFFFFFh	Yes

Syntax **DSJS** *Rd, Address*

Execution
Rd - 1 → Rd
If Rd ≠ 0, then PC' + (offset×16) → PC
If Rd = 0, then go to next instruction

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	D	offset					R	Rd			

Fields
D is a 1-bit direction bit (from PC' to Address):
D=0 - forward jump
D=1 - backward jump

Description DSJS decrements the contents of the destination register by 1. Depending on the result, the TMS34010 either jumps or skips the jump:

- **Rd - 1 ≠ 0**

  The TMS34010 jumps relative to PC'. PC' points to the instruction word that immediately follows the DSJS instruction. Internally, the 5-bit offset is multiplied by 16 to convert it to a bit offset. This allows a jump range of -30 to +32 words from the PC.

  - **If direction bit D = 0**

    The new PC address is obtained by adding the resulting offset to PC'.

  - **If direction bit D = 1**

    The new PC address is obtained by subtracting the resulting offset from PC'.

- **Rd - 1 = 0**

  The TMS34010 skips the jump and continues program execution at the next sequential instruction.

The *Address* operand is a 32-bit address. The assembler calculates the offset as (Address - PC')/16; this results in a jump range of -30 to +32 words from the PC. (The offset is encoded as part of the instruction word.)

This instruction is useful for coding tight loops for cache-resident routines.

Machine States
2,5 (Jump)
3,6 (No jump)

Status Bits
N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before	After	
	A5	**A5**	**Jump taken?**
`DSJS A5,LOOP`	00000009h	00000008h	Yes
`DSJS A5,LOOP`	00000001h	00000000h	No
`DSJS A5,LOOP`	00000000h	FFFFFFFFh	Yes

Syntax **EINT**

Execution 1 → IE

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	0	1	1	0	0	0	0	0

Description EINT sets the global interrupt enable bit (IE) to 1, allowing interrupts to be enabled. When IE=1, individual interrupts are enabled by setting the appropriate bits in the INTENB interrupt mask register. The rest of the status register is unaffected.

The DINT instruction disables interrupts.

Machine States 3,6

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected
IE *1*

Examples

Code	Before	After
	ST	**ST**
EINT	00000010h	00200010h
EINT	00200010h	00200010h

Syntax **EMU**

Execution ST → Rd and conditionally enter emulator mode

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

Description The EMU instruction pulses the $\overline{\text{EMUA}}$ pin and samples the RUN/$\overline{\text{EMU}}$ pin. If the RUN/$\overline{\text{EMU}}$ pin is in the RUN state, the EMU instruction acts as a NOP. If the pin is in the EMU state, emulation mode is entered. This instruction is not intended for general use; refer to the *TMS34010 XDS/22 User's Guide* for more information.

Machine States 8,11 (or more if EMU mode is entered)

Status Bits

N Indeterminate
C Indeterminate
Z Indeterminate
V Indeterminate

Syntax **EXGF** *Rd [, F]*

Execution Rd → FS0, FE0 **or** Rd → FS1, FE1
FS0, FE0 → Rd **or** FS1, FE1 → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	F	1	0	0	0	Rd				

Description EXGF exchanges the six LSBs of the destination register with the selected six bits of field information (field size and field extension). Bit 5 of the 6-bit quantity in Rd is exchanged with the field extension value. The upper 26 bits of Rd are cleared.

Status Register

EXGF's *F* parameter is optional:

F=0 selects FS0, FE0 to be exchanged
F=1 selects FS1, FE1 to be exchanged

If you do not specify an F parameter, the default is 0.

Status Bits
N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before		After	
	A5	ST	A5	ST
`EXGF A5,0`	FFFFFFC0h	F0000FFFh	0000003Fh	F0000FC0h
`EXGF A5,1`	FFFFFFC0h	F0000FFFh	0000003Fh	F000003Fh

Syntax **EXGPC** *Rd*

Execution Rd → PC, PC' → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	0	1	R	Rd			

Description EXGPC exchanges the next program counter value with the destination register contents. After this instruction has been executed, the destination register contains the address of the instruction immediately following the EXGPC instruction.

Note that the TMS34010 sets the four LSBs of the program counter to 0 (word aligned).

This instruction provides a "quick call" capability by saving the return address in a register (rather than on the stack). The return from the call is accomplished by repeating the instruction at the end of the "subroutine." Note that the subroutine address must be reloaded following each call-return operation.

Machine States 2,5

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before		After	
	A1	PC	A1	PC
EXGPC A1	00001C10h	00002080h	00002090h	00001C10h
EXGPC A1	00001C50h	00002080h	00002090h	00001C50h

Syntax **FILL L**

Execution COLOR1 pixels → pixel array (with processing)

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0

Description

FILL processes a set of source pixel values (specified by the COLOR1 register) with a destination pixel array.

This instruction operates on a two-dimensional array of pixels using pixels defined in the COLOR1 register. As the FILL proceeds, the source pixels are combined with destination pixels based on the selected graphics operations.

Note that the **L** parameter in the instruction syntax does not represent a value or a register – the **L** is entered as part of the instruction and identifies the starting address of the pixel array as an L address. That is, the instruction is entered as `FILL L`.

The following set of implied operands govern the operation of the instruction and define both the source pixels and the destination array.

Implied Operands

B File Registers			
Register	**Name**	**Format**	**Description**
B2†	DADDR	Linear	Pixel array starting address
B3	DPTCH	Linear	Pixel array pitch
B7	DYDX	XY	Pixel array dimensions (rows:columns)
B9	COLOR1	Pixel	Fill color or 16-bit pattern
B10–B14†			Reserved registers
I/O Registers			
Address	**Name**	**Description and Operations**	
C00000B0h	CONTROL	PP – Pixel processing operations (22 options) T – Transparency operation	
C0000150h	PSIZE	Pixel size (1,2,4,8,16)	
C0000160h	PMASK	Plane mask – pixel format	

† Changed by FILL during execution.

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the FILL instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an instruction that reads the same location (such as a MOVE SAddress,Rd instruction). For more information, refer to Section 6.2, Latency of Writes to I/O Registers.

Destination Array

The contents of the DADDR, DPTCH, and DYDX registers define the location of the destination pixel array:

- At the outset of the instruction, DADDR contains the **linear** address of the pixel with the lowest address in the array.

During instruction execution, DADDR points to the next pixel (or word of pixels) to be modified in the destination array. When the array transfer is complete, DADDR points to the linear address of the pixel following the last pixel written.

- DPTCH contains the linear difference in the starting addresses of adjacent rows of the destination array. DPTCH must be a multiple of 16, except when a single pixel-width line is drawn (DY=1). In this case, DPTCH may be any value.

- DYDX specifies the dimensions of the destination array in pixels. The DY portion of DYDX contains the number of rows in the array, while the DX portion contains the number of columns.

Pixel Processing Set the PPOP field in the CONTROL register to select a pixel processing operation. This operation is applied to the pixel as it is moved to the destination location. There are 16 Boolean and 6 arithmetic operations; the default operation at reset is *replace* (S → D). Note that the destination data is read through the plane mask and then processed. The 6 arithmetic operations do not operate with pixel sizes of one or two bits per pixel. For more information, see Section 7.7, Pixel Processing, on page 7-15.

Window Checking Window checking **cannot** be used with this instruction. The contents of the WSTART and WEND registers are ignored.

Corner Adjust There is no corner adjust for this instruction. The direction of the FILL is fixed as increasing linear addresses.

Transparency You can enable transparency for this instruction by setting the T bit in the CONTROL I/O register to 1. The TMS34010 checks for 0 (transparent) source pixels *after* it processes the source data. At reset, the default case for transparency is *off*.

Interrupts This instruction can be interrupted at a word or row boundary of the destination array. When the FILL is interrupted, the TMS34010 sets the PBX bit in the status register and then pushes the status register on the stack. At this time, DPTCH and B10–B14 contain intermediate values. DADDR points to the linear address of the next word of pixels to be modified after the interrupt is processed.

Before executing the RETI instruction to return from the interrupt, restore any B-file registers that were modified (also restore the CONTROL register if it was modified). This allows the TMS34010 to resume the FILL correctly.

Plane Mask The plane mask is enabled for this instruction.

Shift Register Transfers If the SRT bit in the DPYCTL register is set, each memory read or write initiated by the FILL generates a shift register transfer read or write cycle at the selected address. This operation can be used for bulk memory clears or transfers. (Not all VRAMs support this capability.) See Section 9.10.2, Video Memory Bulk Initialization, on page 9-28 for more information.

Machine States See Section 13.3, FILL Instructions Timing.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

These FILL examples use the following implied operand setup.

Register File B:		**I/O Registers:**	
DADDR (B2)	= 00002010h	PSIZE	= 0008h
DPTCH (B3)	= 00000080h		
DYDX (B7)	= 0002000Dh		
COLOR1 (B9)	= 30303030h		

Assume that memory contains the following values before instruction execution.

Linear Address	**Data**
02000h	1100h, 3322h, 5544h, 7766h, 9988h, BBAAh,DDCCh, FFEEh
02080h	1100h, 3322h, 5544h, 7766h, 9988h, BBAAh,DDCCh, FFEEh

Example 1

This example uses the pixel processing *replace (S → D)* operation. Before instruction execution, PMASK = 0000h and CONTROL = 0000h (T=0, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	**Data**
02000h	1100h, 3030h, 3030h, 3030h, 3030h, 3030h, 3030h, FF30h
02080h	1100h, 3030h, 3030h, 3030h, 3030h, 3030h, 3030h, FF30h

Example 2

This example uses the *($\overline{S}$ and D) → D* pixel processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 2C00h (T=0, PP=01010).

After instruction execution, memory contains the following values:

Linear Address	**Data**
02000h	1100h, 0302h, 4544h, 4746h, 8988h, 8B8Ah, CDCCh,FFCEh
02080h	1100h, 0302h, 4544h, 4746h, 8988h, 8B8Ah, CDCCh,FFCEh

Example 3

This example uses transparency and the *(S and D) → D* pixel processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 0420h (T=1, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	**Data**
02000h	1100h, 3020h, 1044h, 3020h, 1088h, 3020h, 10CCh, FF20h
02080h	1100h, 3020h, 1044h, 3020h, 1088h, 3020h, 10CCh, FF20h

Example 4 This example uses plane masking – the four MSBs are masked. Before instruction execution, PMASK = 0F0F0h and CONTROL = 0000h (T=0, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	Data
02000h	1100h, 3020h, 5040h, 7060h, 9080h, B0A0h, D0C0h, FFE0h
02080h	1100h, 3020h, 5040h, 7060h, 9080h, B0A0h, D0C0h, FFE0h

Syntax **FILL XY**

Execution COLOR1 pixels → pixel array (with processing)

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0

Description FILL processes a set of source pixel values (specified by the COLOR1 register) with a destination pixel array.

This instruction operates on a two-dimensional array of pixels using pixels defined in the COLOR1 register. As the FILL proceeds, the source pixels are combined with destination pixels based on the selected graphics operations.

Note that the **XY** parameter in the instruction syntax does not represent a value or a register – it is entered as part of the instruction and identifies the starting address of the pixel array as an XY address. That is, the instruction is entered as `FILL L,XY`.

The following set of implied operands govern the operation of the instruction and define both the source pixels and the destination array.

Implied Operands

B File Registers			
Register	**Name**	**Format**	**Description**
B2†‡	DADDR	XY	Pixel array starting address
B3	DPTCH	Linear	Pixel array pitch
B4	OFFSET	Linear	Screen origin (address of 0,0)
B5	WSTART	XY	Window starting corner
B6	WEND	XY	Window ending corner
B7†‡	DYDX	XY	Pixel array dimensions (rows:columns)
B9	COLOR1	Pixel	Fill color or 16-bit pattern
B10–B14†			Reserved registers
I/O Registers			
Address	**Name**	**Description and Elements (Bits)**	
C00000B0h	CONTROL	PP– Pixel processing operations (22 options) W – Window checking operation T – Transparency operation	
C0000140h	CONVDP	XY-to-linear conversion (destination pitch)	
C0000150h	PSIZE	Pixel size (1,2,4,8,16)	
C0000160h	PMASK	Plane mask – pixel format	

† Changed by FILL during execution.
‡ Used for common rectangle function with window hit operation (W=1).

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the FILL instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an instruction that reads the same location (such as a MOVE SAddress,Rd instruction). For more information, refer to Section 6.2, Latency of Writes to I/O Registers.

Destination Array The location of the destination pixel array is defined by the contents of the DADDR, DPTCH, CONVDP, OFFSET, and DYDX registers. At the outset of the instruction, DADDR contains the **XY** address of the pixel with the lowest address in the array. It is used with OFFSET and CONVDP to calculate the linear address of the starting location of the array. DPTCH contains the linear difference in the starting addresses of adjacent rows of the destination array (typically this is the screen pitch). DPTCH must be a power of two (greater than or equal to 16) and CONVDP must be set to correspond to the DPTCH value. CONVDP is computed by operating on the DPTCH register with the LMO instruction; it is used for the XY calculations involved in XY addressing and window clipping. DYDX specifies the dimensions of the destination array in pixels. The DY portion of DYDX contains the number of rows in the array, while the DX portion contains the number of columns. During instruction execution, DADDR points to the next pixel (or word of pixels) to be modified in the destination array. When the array transfer is complete, DADDR points to the linear address of the pixel following the last pixel written. This is that pixel on the **last** row that would have been written had the array transfer been wider in the X dimension.

Pixel Processing Pixel processing can be used with this instruction. The PPOP field of the CONTROL register specifies the pixel processing operation that is applied to pixels as they are processed with the destination array. There are 16 Boolean and 6 arithmetic operations; the default case at reset is the *replace (S → D)* operation. Note that the destination data is read through the plane mask and then processed. The 6 arithmetic operations do not operate with pixel sizes of one or two bits per pixel. For more information, see Section 7.7, Pixel Processing, on page 7-15.

Window Checking The window operations described in Section 7.10, Window Checking, on page 7-27. can be used with this instruction. You can select window pick, violation detect, or preclipping by setting the W bits in the CONTROL register to 1, 2, or 3, respectively. Window pick modifies the DADDR and DYDX registers to correspond to the common rectangle formed by the destination array and the clipping window defined by WSTART and WEND. DADDR is set to the XY address of the pixel with the lowest address in the common rectangle, while DYDX is set to the X and Y dimensions of the rectangle. If no window operations are selected, the WSTART and WEND registers are ignored. At reset, no window operations are enabled.

Corner Adjust There is no corner adjust for this instruction. The direction of the FILL is fixed as increasing linear addresses.

Transparency You can enable transparency for this instruction by setting the T bit in the CONTROL I/O register to 1. The TMS34010 checks for 0 (transparent) source pixels *after* it processes the source data. At reset, the default case for transparency is *off*.

Interrupts This instruction can be interrupted at a word or row boundary of the destination array. When the FILL is interrupted, the TMS34010 sets the PBX bit in the status register and then pushes the status register on At this time, DPTCH and B10–B14 contain intermediate values. DADDR points to the linear address of the next word of pixels to be modified after the interrupt is processed.

Before executing the RETI instruction to return from the interrupt, restore any B-file registers that were modified (also restore the CONTROL register if it was modified). This allows the TMS34010 to resume the FILL correctly.

Plane Mask The plane mask is enabled for this instruction.

Shift Register Transfers If the SRT bit in the DPYCTL register is set, each memory read or write initiated by the FILL generates a shift register transfer read or write cycle at the selected address. This operation can be used for bulk memory clears or transfers. (Not all VRAMs support this capability.) See Section 9.10.2, Video Memory Bulk Initialization, on page 9-28 for more information.

Machine States See Section 13.3, FILL Instructions Timing.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V *1* if a window violation occurs, *0* otherwise; unaffected if window clipping is not enabled

Examples These FILL examples use the following implied operand setup.

Register File B:		**I/O Registers:**	
DADDR (B2)	= 00520007h	CONVDP	= 0017h
DPTCH (B3)	= 00000100h	PSIZE	= 0004h
OFFSET (B4)	= 00010000h	PMASK	= 0000h
WSTART (B5)	= 0030000Ch	CONTROL	= 0000h
WEND (B6)	= 00530014h		(W=00, T=0, PP=00000)
DYDX (B7)	= 00030012h		
COLOR1 (B9)	= FFFFFFFFh		

Assume that memory contains the following values before instruction execution.

Linear Address	**Data**
15200h	3210h, 7654h, BA98h, FEDCh, 3210h, 7654h, BA98h, FEDCh
15300h	3210h, 7654h, BA98h, FEDCh, 3210h, 7654h, BA98h, FEDCh
15400h	3210h, 7654h, BA98h, FEDCh, 3210h, 7654h, BA98h, FEDCh

Example 1 This example uses the *replace* (S → D) pixel processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 0000h (T=0, W=00, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	**Data**
15200h	3210h, F654h, FFFFh, FFFFh, FFFFh, FFFFh, BA9Fh, FEDCh
15300h	3210h, F654h, FFFFh, FFFFh, FFFFh, FFFFh, BA9Fh, FEDCh
15400h	3210h, F654h, FFFFh, FFFFh, FFFFh, FFFFh, BA9Fh, FEDCh

XY Addressing

Y Address	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
	0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F
Address 52	0 1 2 3 4 5 6 F F F F F F F F F F F F F F F F 9 A B C D E F
Address 53	0 1 2 3 4 5 6 F F F F F F F F F F F F F F F F 9 A B C D E F
Address 54	0 1 2 3 4 5 6 F F F F F F F F F F F F F F F F 9 A B C D E F

Example 2 This example uses the *(D XOR S)* → *D* pixel processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 2800h (T=0, W=00, PP=01010).

After instruction execution, memory contains the following values:

Y Address	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
	0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F
Address 52	0 1 2 3 4 5 6 8 7 6 5 4 3 2 1 0 F E D C B A 9 8 7 9 A B C D E F
Address 53	0 1 2 3 4 5 6 8 7 6 5 4 3 2 1 0 F E D C B A 9 8 7 9 A B C D E F
Address 54	0 1 2 3 4 5 6 8 7 6 5 4 3 2 1 0 F E D C B A 9 8 7 9 A B C D E F

Example 3 This example uses transparency, the *(D subs S)* → *D* pixel processing operation. Before instruction execution, COLOR1 = 88888888h, PMASK = 0000h, and CONTROL = 4C20h (T=1, W=00, PP=10011).

After instruction execution, memory contains the following values:

Y Address	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
	0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F
Address 52	0 1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 8 9 A B C D E F
Address 53	0 1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 8 9 A B C D E F
Address 54	0 1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 8 9 A B C D E F

Example 4

This example uses window operation 3 – the destination is clipped. Before instruction execution, PMASK = 0000h and CONTROL = 00C0h (T=0, W=11, PP=00000).

After instruction execution, memory contains the following values:

X Address

Y Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
52	0	1	2	3	4	5	6	7	8	9	A	B	F	F	F	F	F	F	F	F	F	5	6	7	8	9	A	B	C	D	E	F
53	0	1	2	3	4	5	6	7	8	9	A	B	F	F	F	F	F	F	F	F	F	5	6	7	8	9	A	B	C	D	E	F
54	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Example 5

This example uses plane masking – the most significant bit is masked. Before instruction execution, PMASK = 8888h and CONTROL = 0000h (T=0, W=00, PP=00000).

After instruction execution, memory contains the following values:

X Address

Y Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
52	0	1	2	3	4	5	6	7	F	F	F	F	F	F	F	F	7	7	7	7	7	7	7	7	F	9	A	B	C	D	E	F
53	0	1	2	3	4	5	6	7	F	F	F	F	F	F	F	F	7	7	7	7	7	7	7	7	F	9	A	B	C	D	E	F
54	0	1	2	3	4	5	6	7	F	F	F	F	F	F	F	F	7	7	7	7	7	7	7	7	F	9	A	B	C	D	E	F

Syntax **GETPC** *Rd*

Execution PC' → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	1	0	R	Rd			

Description GETPC increments the PC contents by 16 to point past the GETPC instruction, and copies the value into the destination register. Execution continues with the next instruction. You can use GETPC with the EXGPC and JUMP instructions for quick call on jump operations. You can also use GETPC to access relocatable data areas whose position relative to the code area is known at assembly time.

Machine States 1,4

Status Bits
N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before	After
	PC	**A1**
`GETPC A1`	00001BD0h	00001BE0h
`GETPC A1`	00001C10h	00001C20h

Syntax **GETST** *Rd*

Execution ST → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1	0	0	R	Rd			

Description GETST copies the contents of the status register into the destination register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
N	C	Z	V	Res		PBX	Res			IE	Reserved									FE1	FS1					FE0	FS0				

Status Register

Machine States 1,4

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before	After
	PC	**A1**
`GETST A1`	20200010h	20200010h
`GETST A1`	00000010h	00000010h

Syntax **INC** *Rd*

Execution Rd + 1 → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	0	0	1	R	Rd			

Description INC adds 1 to the contents of the destination register and stores the result in the destination register. This instruction is an alternate mnemonic for `ADDK 1,Rd`.

You can accomplish multiple-precision arithmetic by using INC in conjunction with the ADDC instruction.

Machine States 1,4

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a carry, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before	After	
	A1	A1	NCZV
`INC A1`	00000000h	00000001h	0000
`INC A1`	0000000Fh	00000010h	0000
`INC A1`	FFFFFFFFh	00000000h	0110
`INC A1`	FFFFFFFEh	FFFFFFFFh	1000
`INC A1`	7FFFFFFFh	80000000h	1001

Syntax JA*cc* *Address*

Execution If condition *true*, then Address → PC
If condition *false*, then go to next instruction

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	code				1	0	0	0	0	0	0	0
16 LSBs of Address															
16 MSBs of Address															

Fields **code** is a 4-bit digit that identifies the condition for the jump within the opcode. (See the condition codes table below.)

Description The JA*cc* instruction conditionally jumps to an absolute address. The *cc* is part of a mnemonic that represents the condition for the jump; for example, if *cc* is UC, then the instruction is JAUC. (See the condition mnemonics and codes listed below.) If the specified condition is **true**, the TMS34010 jumps to the address and continues execution from that point. If the specified condition is **false**, the TMS34010 skips the jump and continues execution at the next sequential instruction. Note that the lower four bits of the program counter are set to 0 (word aligned).

The *Address* operand in the syntax represents the 32-bit absolute address. Note that the second and third instruction words contain the address for the jump.

The JA*cc* instructions are usually used in conjunction with the CMP and CMPI instructions. The JAV and JANV instructions can also be used to detect window violations or CPW status.

Condition Codes

	Mnemonic		Result of Compare	Status Bits	Code
Unconditional Compares	JAUC	–	Unconditional	don't care	0000
Unsigned Compares	JALO (JAC)	–	Dst lower than Src	C	0001
	JALS	JAYLE	Dst lower or same as Src	$\overline{C} + Z$	0010
	JAHI	JAYGT	Dst higher than Src	$\overline{C} \cdot \overline{Z}$	0011
	JAHS (JANC)	–	Dst higher or same as Src	$\overline{C}$	1001
	JAEQ (JAZ)	–	Dst = Src	Z	1010
	JANE (JANZ)	–	Dst ≠ Src	$\overline{Z}$	1011
Signed Compares	JALT	JAXLE	Dst < Src	$(N \cdot \overline{V}) + (\overline{N} \cdot V)$	0100
	JALE	–	Dst ≤ Src	$(N \cdot \overline{V}) + (\overline{N} \cdot V) + Z$	0110
	JAGT	–	Dst > Src	$(N \cdot V \cdot \overline{Z}) + (\overline{N} \cdot \overline{V} \cdot \overline{Z})$	0111
	JAGE	JAXGT	Dst ≥ Src	$(N \cdot V) + (\overline{N} \cdot \overline{V})$	0101
	JAEQ (JAZ)	–	Dst = Src	Z	1010
	JANE (JANZ)	–	Dst ≠ Src	$\overline{Z}$	1011
Compare to Zero	JAZ	JAYZ	Result = zero	Z	0101
	JANZ	JAYNZ	Result ≠ zero	$\overline{Z}$	1011
	JAP	–	Result is positive	$\overline{N} \cdot \overline{Z}$	0001
	JAN	JAXZ	Result is negative	N	1110
	JANN	JAXNZ	Result is nonnegative	$\overline{N}$	1111

Condition Codes (continued)

	Mnemonic		Result of Compare	Status Bits	Code
General Arithmetic	**JAZ**	**JAYZ**	**Result is zero**	$\underline{Z}$	**1010**
	JANZ	**JAYNZ**	**Result is nonzero**	$\underline{\overline{Z}}$	1011
	JAC	JAYN	Carry set on result	$\underline{C}$	1000
	JANC	JAYNN	No carry on result	$\underline{\overline{C}}$	1001
	JAB (JAC)	–	Borrow set on result	C	1000
	JANB (JANC)	–	No borrow on result	$\overline{C}$	1001
	JAV†	JAXN	Overflow on result	$\underline{V}$	1100
	JANV†	JAXNN	No overflow on result	$\underline{\overline{V}}$	1101

Note: A mnemonic code in parentheses is an alternate code for the preceding code.
† Also used for window clipping
\+ Logical OR
· Logical AND
¯ Logical NOT

Machine States
3,6 (Jump)
4,7 (No jump)

Status Bits
N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code		Flags for Branch			Code		Flags for Branch		
		NCZV	NCZV	NCZV			NCZV	NCZV	NCZV
JAUC	HERE	xxxx			JAV	HERE	xxx1		
JAP	HERE	0x0x			JANZ	HERE	xx0x		
JALS	HERE	xx1x	x1xx		JANN	HERE	0xxx		
JAHI	HERE	x00x			JANV	HERE	xxx0		
JALT	HERE	0xx1	1xx0		JAN	HERE	1xxx		
JAGE	HERE	0xx0	1xx1		JAB	HERE	x1xx		
JALE	HERE	0xx1	1xx0	xx1x	JANB	HERE	x0xx		
JAGT	HERE	0x00	1x01		JALO	HERE	x1xx		
JAC	HERE	x1xx			JAHS	HERE	x00x	xx1x	
JANC	HERE	x0xx			JANE	HERE	xx0x		
JAZ	HERE	xx1x			JAEQ	HERE	xx1x		

Note that the TMS34010 jumps when any one or more of the *Flags for Branch* listed above are set as indicated.

Syntax JR*cc* *Address*

Execution If condition *true*, then offset + PC' → PC
If condition *false*, then go to next instruction

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	code				offset							

Fields **code** is a 4-bit digit that identifies the condition for the jump within the opcode. (See the condition codes table below.)

Description The JR*cc* instruction conditionally jumps to an address that is relative to the current PC. The *cc* is part of a mnemonic that represents the condition for the jump; for example, if *cc* is UC, then the instruction is JAUC. (See the condition mnemonics and codes listed below.) If the specified condition is **true**, the TMS34010 jumps to a new location. The assembler calculates the address of this location by adding the address of the next instruction (PC') to the signed word offset. The TMS34010 then continues execution from this point. If the specified condition is **false**, the TMS34010 skip the jump and continues execution at the next sequential instruction.

The *Address* operand in the syntax represents the 32-bit relative address. The assembler calculates the offset as (Address - PC')/16 and inserts the resulting 8-bit offset into the opcode. The range for this form of the JR*cc* instruction is ±127 words (excluding 0).

If the offset is outside the range of ±127 words, the assembler automatically substitutes the longer form of the JR*cc* instruction. If the offset is 0, the assembler substitutes a NOP instruction. The assembler does not accept an address which is externally defined or an address which is relative to a different section than the PC. Note that the four LSBs of the program counter are always 0 (word aligned).

The JR*cc* instructions are usually used in conjunction with the CMP and CMPI instructions. The JRV and JRNV instructions can also be used to detect window violations or CPW status.

Condition Codes

	Mnemonic		Result of Compare	Status Bits	Code
Unconditional Compares	JRUC	–	Unconditional	don't care	0000
Unsigned Compares	JRLO (JRC)	–	Dst lower than Src	C	0001
	JRLS	JRYLE	Dst lower or same as Src	$\overline{C} + Z$	0010
	JRHI	JRYGT	Dst higher than Src	$\overline{C} \cdot \overline{Z}$	0011
	JRHS (JRNC)	–	Dst higher or same as Src	$\overline{C}$	1001
	JREQ (JRZ)	–	Dst = Src	Z	1010
	JRNE (JRNZ)	–	Dst ≠ Src	$\overline{Z}$	1011

Condition Codes (continued)

	Mnemonic		Result of Compare	Status Bits	Code
Signed Compares	**JRLT**	**JRXLE**	**Dst < Src**	$(N \cdot \overline{V}) + (\overline{N} \cdot V)$	0100
	JRLE	–	Dst ≤ Src	$(N \cdot \overline{V} + (\overline{N} \cdot V) + Z$	0110
	JRGT	–	Dst > Src	$(N \cdot V \cdot \overline{Z}) + (\overline{N} \cdot \overline{V} \cdot \overline{Z})$	0111
	JRGE	JRXGT	Dst ≥ Src	$(N \cdot V) + (\overline{N} \cdot \overline{V})$	0101
	JREQ (JRZ)	–	Dst = Src	Z	1010
	JRNE (JRNZ)	–	Dst ≠ Src	$\overline{Z}$	1011
Compare to Zero	JRZ	JRYZ	Result = zero	Z	0101
	JRNZ	JRYNZ	Result ≠ zero	$\overline{Z}$	1011
	JRP	–	Result is positive	$\overline{N} \cdot \overline{Z}$	0001
	JRN	JRXZ	Result is negative	N	1110
	JRNN	JRXNZ	Result is nonnegative	$\overline{N}$	1111
General Arithmetic	JRZ	JRYZ	Result is zero	Z	1010
	JRNZ	JRYNZ	Result is nonzero	$\overline{Z}$	1011
	JRC	JRYN	Carry set on result	C	1000
	JRNC	JRYNN	No carry on result	$\overline{C}$	1001
	JRB (JRC)	–	Borrow set on result	C	1000
	JRNB (JRNC)	–	No borrow on result	$\overline{C}$	1001
	JRV†	JRXN	Overflow on result	V	1100
	JRNV†	JRXNN	No overflow on result	$\overline{V}$	1101

Note: A mnemonic code in parentheses is an alternate code for the preceding code.
† Also used for window clipping
\+ Logical OR
· Logical AND
¯ Logical NOT

Machine States

2,5 (Jump)
1,4 (No jump)

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Flags for Branch			Code	Flags for Branch		
	NCZV	NCZV	NCZV		NCZV	NCZV	NCZV
JRUC HERE	xxxx			JRC HERE	x1xx		
JRP HERE	0x0x			JRNC HERE	x0xx		
JRLS HERE	xx1x	x1xx		JRZ HERE	xx1x		
JRHI HERE	x00x			JRNZ HERE	xx0x		
JRLT HERE	0xx1	1xx0		JRV HERE	xxx1		
JRGE HERE	0xx0	1xx1		JRNV HERE	xxx0		
JRLE HERE	0xx1	1xx0	xx1x	JRN HERE	1xxx		
JRGT HERE	0x00	1x01		JRNN HERE	0xxx		

Note that the TMS34010 jumps when any one or more of the *Flags for Branch* listed above are set as indicated.

Syntax **JR***cc Address*

Execution If condition *true*, then offset + PC' → PC
If condition *false*, then go to next instruction

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	code				0	0	0	0	0	0	0	0
offset															

Fields **code** is a 4-bit digit that identifies the condition for the jump within the opcode. (See the condition codes table below.)

Description The JR*cc* instruction conditionally jumps to an address that is relative to the current PC. The *cc* is part of a mnemonic that represents the condition for the jump; for example, if *cc* is UC, then the instruction is JAUC. (See the condition mnemonics and codes listed below.) If the specified condition is **true**, the TMS34010 jumps to a new location. The assembler calculates the address of this location by adding the address of the next instruction (PC') to the signed word offset. The TMS34010 then continues execution from this point. If the specified condition is **false**, the TMS34010 skips the jump and continues execution at the next sequential instruction.

The *Address* operand in the syntax represents the 32-bit relative address. The assembler calculates the offset as (Address - PC')/16 and inserts the resulting offset into the second instruction word of the opcode. The range for this form of the JR*cc* instruction is -32,768 to +32,767 words (excluding 0).

If the offset is 0, the assembler substitutes a NOP instruction. If the address is out of range, the assembler uses the JA*cc* instruction instead. The assembler does not accept an address which is externally defined or an address which is relative to a different section than the PC. Note that the four LSBs of the program counter are always 0 (word aligned).

The JR*cc* instructions are usually used in conjunction with the CMP and CMPI instructions. The JRV and JRNV instructions can also be used to detect window violations or CPW status.

Condition Codes

	Mnemonic		Result of Compare	Status Bits	Code
Unconditional Compares	JRUC	–	Unconditional	don't care	0000
Unsigned Compares	JRLO (JRC)	–	Dst lower than Src	C	0001
	JRLS	JRYLE	Dst lower or same as Src	$C + Z$	0010
	JRHI	JRYGT	Dst higher than Src	$\overline{C} \cdot \overline{Z}$	0011
	JRHS (JRNC)	–	Dst higher or same as Src	$\overline{C}$	1001
	JREQ (JRZ)	–	Dst = Src	Z	1010
	JRNE (JRNZ)	–	Dst ≠ Src	$\overline{Z}$	1011

Condition Codes (continued)

	Mnemonic		Result of Compare	Status Bits	Code
Signed Compares	JRLT	JRXLE	Dst < Src	$(N \cdot \overline{V}) + (\overline{N} \cdot V)$	0100
	JRLE	–	Dst ≤ Src	$(N \cdot \overline{V} + (\overline{N} \cdot V) + Z$	0110
	JRGT	–	Dst > Src	$(N \cdot V \cdot \overline{Z}) + (\overline{N} \cdot \overline{V} \cdot \overline{Z})$	0111
	JRGE	JRXGT	Dst ≥ Src	$(N \cdot V) + (\overline{N} \cdot \overline{V})$	0101
	JREQ (JRZ)	–	Dst = Src	Z	1010
	JRNE (JRNZ)	–	Dst ≠ Src	$\overline{Z}$	1011
Compare to Zero	JRZ	JRYZ	Result = zero	Z	0101
	JRNZ	JRYNZ	Result ≠ zero	$\overline{Z}$	1011
	JRP	–	Result is positive	$\overline{N} \cdot \overline{Z}$	0001
	JRN	JRXZ	Result is negative	N	1110
	JRNN	JRXNZ	Result is nonnegative	$\overline{N}$	1111
General Arithmetic	JRZ	JRYZ	Result is zero	Z	1010
	JRNZ	JRYNZ	Result is nonzero	$\overline{Z}$	1011
	JRC	JRYN	Carry set on result	C	1000
	JRNC	JRYNN	No carry on result	$\overline{C}$	1001
	JRB (JRC)	–	Borrow set on result	C	1000
	JRNB (JRNC)	–	No borrow on result	$\overline{C}$	1001
	JRV†	JRXN	Overflow on result	V	1100
	JRNV†	JRXNN	No overflow on result	$\overline{V}$	1101

Note: A mnemonic code in parentheses is an alternate code for the preceding code.
† Also used for window clipping
+ Logical OR
· Logical AND
¯ Logical NOT

Machine States
3,6 (Jump)
4,7 (No jump)

Status Bits
N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code		Flags for Branch			Code		Flags for Branch		
		NCZV	NCZV	NCZV			NCZV	NCZV	NCZV
JRUC	HERE	xxxx			JRV	HERE	xxx1		
JRP	HERE	0x0x			JRNZ	HERE	xx0x		
JRLS	HERE	xx1x	x1xx		JRNN	HERE	0xxx		
JRHI	HERE	x00x			JRNV	HERE	xxx0		
JRLT	HERE	0xx1	1xx0		JRN	HERE	1xxx		
JRGE	HERE	0xx0	1xx1		JRB	HERE	x1xx		
JRLE	HERE	0xx1	1xx0	xx1x	JRNB	HERE	x0xx		
JRGT	HERE	0x00	1x01		JRLO	HERE	x1xx		
JRC	HERE	x1xx			JRHS	HERE	x00x	xx1x	
JRNC	HERE	x0xx			JRNE	HERE	xx0x		
JRZ	HERE	xx1x			JREQ	HERE	xx1x		

Note that the TMS34010 jumps when any one or more of the *Flags for Branch* listed above are set as indicated.

Syntax **JUMP** *Rs*

Execution Rs → PC

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	1	1	R	Rs			

Description JUMP jumps to the address contained in the source register. The TMS34010 sets the four LSBs of the program counter to 0 (word aligned). This instruction can be used in conjunction with the GETPC and/or EXGPC instructions.

Machine States 2,5

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before		After
	A1	PC	PC
`JUMP A1`	00001EE0h	00555550h	00001EE0h
`JUMP A1`	00001EE5h	00555550h	00001EE0h
`JUMP A1`	FFFFFFFFh	00555550h	FFFFFFF0h

Syntax **LINE {0, 1}**

Execution The two execution algorithms for the LINE instruction are explained below. These algorithms are similar, varying only in their treatment of d=0.

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	1	Z	0	0	1	1	0	1	0

Fields The assembler sets bit 7 in the instruction word (the Z bit) to 0 or 1, depending on which LINE algorithm you select:

Z=0 selects algorithm 0
Z=1 selects algorithm 1

Description LINE performs the inner loop of Bresenham's line-drawing algorithm. This type of line draw plots a series of points (x_i,y_i) either diagonally or laterally with respect to the previous point. Movement from pixel to pixel always proceeds in a dominant lateral direction. The algorithm may or may not also increment in the direction with the smaller dimension (this produces a diagonal movement). Two XY-format registers supply the XY increment values for the two possible movements. The LINE instruction maintains a decision variable, d, that acts as an error term, controlling movement in either the dominant or diagonal direction. The algorithm operates in one of two modes, depending on how the condition d=0 is treated.

During LINE execution, some portion of a line [(x_0,y_0)(x_1,y_1)] is drawn. The line is drawn so that the axis with the largest extent has dimension a and the axis with the least extent has dimension b. Thus, a is the larger (in absolute terms) of $y_1 - y_0$ or $x_1 - x_0$ and b is the smaller of the two. This means that $a \geq b \geq 0$.

The following values must be supplied to draw a line from (x_0,y_0) to (x_1,y_1):

1) Set the XY pointer (x_i,y_i) in the DADDR register to the initial value of (x_0,y_0).

2) Use the line endpoints to determine the major and minor dimensions (a and b, respectively) for the line draw; then set the DYDX register to this value (b:a).

3) Place the signed XY increment for a movement in the diagonal (or minor) direction ($d \geq 0$ for Z=0, $d > 0$ for Z=1) in the INC1 register.

4) Place the signed XY increment for a movement in the dominant (or major) direction ($d < 0$ for Z=0, $d \leq 0$ for Z=1) in the INC2 register.

5) Set the initial value of the decision variable in register B0 to $2b - a$.

6) Set the initial count value in the COUNT register to $a + 1$.

7) Set the LINE color in the COLOR1 register.

8) Set the PATTRN register to all 1s.

The LINE instruction may use one of two algorithms, depending on the value of Z:

Algorithm 0 (Z=0):

```
While COUNT > 0
      COUNT = COUNT - 1
      Draw the next pixel
      If d ≥ 0
            d = d + 2b - 2a
            POINTER = POINTER + INC1
      Else d = d + 2b;
            POINTER = POINTER + INC2
```

Algorithm 1 (Z=1):

```
While COUNT > 0
      COUNT = COUNT - 1
      Draw the next pixel
      If d > 0
            d = d + 2b - 2a
            POINTER = POINTER + INC1
      Else d = d + 2b;
            POINTER = POINTER + INC2
```

LINE 1 is commonly used to draw lines with decreasing y values; LINE 0 is used to draw lines with increasing y values. For horizontal lines, use FILL or LINE 0.

Implied Operands

B File Registers			
Register	**Name**	**Format**	**Description**
B0†	SADDR	Integer	Decision variable, *d*
B2†	DADDR	XY	Starting point (y_i:x_i), usually (y_0:x_0)
B4	OFFSET	Linear	Screen origin (0,0)
B5	WSTART	XY	Window starting corner
B6	WEND	XY	Window ending corner
B7	DYDX	XY	*b*:*a* minor:major line dimensions
B9	COLOR1	Pixel	Pixel color to be replicated
B10†	COUNT	Integer	Loop count
B11	INC1	XY	Minor axis (diagonal) increment, INC1
B12	INC2	XY	Major axis (dominant) increment, INC2
B13†	PATTRN	Pattern	Future pattern register, must be set to all 1s
B14	TEMP	–	Temporary register
I/O Registers			
Address	**Name**	**Description and Elements (Bits)**	
C00000B0h	CONTROL	PP – Pixel processing operations W – Window clipping operation T – Transparency operation	
C0000140h	CONVDP	XY-to-linear conversion (destination pitch)	
C0000150h	PSIZE	Pixel size (1,2,4,8,16)	
C0000160h	PMASK	Plane mask – pixel format	

† These registers are changed by instruction execution

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the LINE instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an instruction that reads the same location (such as a MOVE SAddress,Rd instruction). For more information, refer to Section 6.2, Latency of Writes to I/O Registers.

Pixel Processing The PP field in the CONTROL I/O register specifies the operation to be applied to the pixel as it is written. There are 22 operations; the default case at reset is the pixel processing *replace* (S → D) operation. For more information, see Section 7.7, Pixel Processing, on page 7-15.

Window Checking Window clipping or pick is selected by setting the W bits in the CONTROL I/O register to the appropriate value. The WSTART and WEND registers define the window in XY-coordinate space.

Options include:

0 *No window clipping.* LINE draws the entire line. Neither the WVP or V bit are affected. WSTART and WEND are ignored.

1 *Window hit.* The instruction calculates points but no pixels are actually drawn. As soon as the pixel to be drawn lies inside the window, the WVP bit is set, the V bit is cleared, and the instruction is aborted. At this point, registers B0, B2, B10, B13, and B14 are set so as to draw the next pixel in the line; B0 is set to the value for the pixel beyond the next pixel on the line. If the line lies entirely outside the window, then the WVP bit is not affected, the V bit in the status is set, and the instruction completes execution.

2 *Clip and set WVP.* LINE draws pixels until the pixel to be drawn lies outside the window. At this point, the WVP bit is set, the V bit is set, and the instruction is aborted. At this point, registers B0, B2, B10, B13, and B14 are set so as to draw the next pixel in the line; B0 is set to the value for the pixel beyond the next pixel on the line. If the entire line lies within the window, then the WVP bit **is not affected**, the V bit is cleared and the instruction completes execution. The initial value of WVP does not affect instruction execution.

3 *Clip.* LINE calculates all the points, but only draws the points that lie inside the window. The V bit tracks the state of the last pixel. If the pixel was outside the window, V is set to 1; otherwise, it is 0. The instruction traverses the entire line.

The default case at reset is no window clipping. For more information, see Section 7.10, Window Checking, on page 7-25.

Transparency You can enable transparency for this instruction by setting the T bit in the CONTROL I/O register to 1. The TMS34010 checks for 0 (transparent) source pixels *after* it processes the source data. At reset, the default case for transparency is *off*.

Plane Mask The plane mask is enabled for this instruction.

Interrupts LINE may be interrupted after every pixel in the line draw except for the last pixel. If the instruction is interrupted, the PC is decremented by 16 to point back to the LINE instruction (the one being executed) before the PC is

pushed on the stack. Thus, the LINE instruction is resumed upon return from the interrupt. In order for the LINE to be resumed correctly, any B-file registers that are modified by the interrupting routine must be restored, and the RETI instruction must be executed. Note that a LINE instruction that is aborted because of window checking options 1 or 2 does not decrement the PC before pushing it on the stack. In this case, the LINE is not resumed after returning from the interrupt service routine.

Machine States

The total LINE instruction timing is obtained by adding a setup time to a transfer time:

LINE time = LINE setup time + LINE transfer time

- LINE **setup time** is the overhead incurred from initiating the LINE instruction. The setup sequence executes an initialization sequence, performing any necessary setup operations and translations. ***The setup time is always 4 machine states.***

- The **transfer sequence** performs the actual data transfer from the source register to the destination pixels. Table 12-10 shows LINE transfer timing. LINE transfer timing may be influenced by window and pixel processing operations; their affects are discussed in the list that follows Table 12-10.

Table 12-10. LINE Transfer Timing

	Window Option			
Instruction	**W=0 (Off)**	**W=1 Window Hit**	**W=2, Interrupt On Clip**	**W=3 Clipping**
LINE 0	$(3+P)E$	$5q + 5$	$(3+P)E$ †	$(3+P)E + 5q$
LINE 1	$(3+P)E$	$5q + 5$	$(3+P)E$ †	$(3+P)E + 5q$

† Add 5 for a window violation

Key:
E Number of pixels written
q Number of pixels calculated, but not written
P Selected pixel processing operation

Although window operations affect the setup time of most instructions, they are performed *during transfer execution* of the LINE instruction, affecting it on a per-pixel basis. Window operations that affect the LINE instruction include:

- No window checking
- Window clip: V flag set, LINE aborted on first write outside window
- Window hit: WVP flag set, V flag cleared, abort LINE on first write inside window

Pixel processing operations influence the LINE transfer timing. (The effects of other graphics operations, such as plane masking and transparency, are already included.) Pixel processing consumes 2, 4, 5, or 6 machine states per pixel, depending on the operation selected. Table 12-11 shows the effects of pixel processing on LINE timing.

Table 12-11. Per-Word Timing Values for Pixel Processing (*P*)

Replace	Other Booleans or ADD	ADDS,SUBS MAX or MIN	SUBS
2	4	5	6

Figure 12-11 illustrates timing for a **LINE 0**, drawing a line from (3h,52h) to (19h,55h).

```
*************************************************
* Implied operand setup for LINE example (assume *
* that B register and I/O register names are     *
* equated with the proper registers)             *
*************************************************
       MOVI  0FFFFFFF0h, B0   ; Decision variable d=2b-a=-16
       MOVI  00520003h, B2    ; DADDR
       MOVI  00000800h, B3    ; DPTCH (CONVDP=14)
       MOVI  00000100h, B4    ; OFFSET
       MOVI  00300003h, B5    ; WSTART
       MOVI  00550025h, B6    ; WEND
       MOVI  00030016h, B7    ; b:a; b=3 and a=22
       MOVI  44444444h, B9    ; COLOR1 (color of the line)
       MOVI  00000017h, B10   ; COUNT (a+1)
       MOVI  00010001h, B11   ; Diagonal increment (+1,+1)
       MOVI  00000001h, B12   ; Nondiagonal increment (0,+1)
       MOVI  0FFFFFFFFh, B13  ; PATTRN (all 1s)
       MOVI  00C0h, A0
       MOVE  A0, @CONTROL     ; W=3, T=0, PP=0,
       CLR   A0
       MOVE  A0, @PMASK       ; No plane masking
```

Figure 12-11. Implied Operand Setup for LINE Timing Example

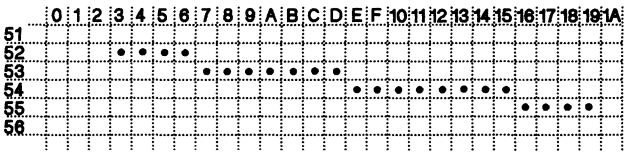


Figure 12-12. LINE Timing Example

Follow these steps to determine the number of machine states consumed by this LINE example:

1) The **setup time** for a LINE instruction is always 4 machine states.

2) Determine the **transfer time**. Transfer time comprehends windowing, the number of pixels drawn, and graphics operations.

   a) *Windowing*: is on for this LINE 0 instruction; as Table 12-10 shows, the transfer timing is $(3+P)E + 5Q$.
   b) *Graphics operations*: The pixel processing *replace* operation has been selected; according to Table 12-11, ***P*=2**.

c) *Number of pixels drawn*: Register B10 indicates the total number of pixels in the line (23). Since the line fits within the window, all pixels calculated are drawn; thus, $E = 23$ and $Q=0$.

The total machine states required for this instruction are:

LINE time	= LINE setup time	+	LINE transfer time
	= 4	+	$(3+P)E + 5Q$
	= 4	+	$(3+2) \times 23 + 0$
	= **119 states**		

119 states are needed to draw these 23 pixels.

The LINE instruction may be interrupted on any pixel boundary during the transfer portion of the algorithm. The context of the LINE is saved in reserved registers; the PC is decremented before it is pushed on the stack, so that execution returns to the LINE opcode. This operation takes 20 machine states for the interrupt to be recognized. The time for the context switch must be added; see the TRAP instruction for context switch timing.

Status Bits

N Undefined
C Undefined
Z Undefined
V Set depending upon window operation.

Linedraw Code

The following code segment shows setup and execution of the LINE instruction.

```
*************************************************************************
* Draw a line from point (xs,ys) to point (xe,ye) using Bresenham's     *
* algorithm. When _draw_line is called, xs is in the 16 LSBs of B2,     *
* ys is in the 16 MSBs of B2, xe is in the 16 LSBs of B0, and ye is     *
* in the 16 MSBs of B0.                                                 *
*************************************************************************

          .global  _draw_line
_draw_line:
          SUBXY    B2, B0       ; Calculate a and b

* Now set up B7 = (a,b) and B11 = (dx_diag,dy_diag). Assume that
* a < 0 and b < 0; if a >= 0 or b >= 0, make corrections later.
* Register B11 (INC1) contains dy_diag::dx_diag
* Register B12 (INC2) contains dy_nondiag::dx_nondiag

          MOVI     -1, B11      ; dx_diag = dy_diag - 1
          MOVK     1, B12       ; Constant = 1
          CLR      B7
          SUBXY    B0, B7       ; B7 = (-a,-b)
          JRNC     L1           ; Jump if b < 0
* Handle case where b >= 0:
          MOVY     B0, B7       ; Make a in B7 positive
          SRL      15, B11      ; Change dy_diag to +1
L1:
          JRNV     L2           ; Jump if a < 0

* Handle case where a >= 0:
          MOVX     B0, B7       ; Take absolute value of a
          MOVX     B12, B11     ; Change dx_diag to +1
L2:
          MOVX     B11, B12     ; dx_nondiag=dx_diag, dy_nondiag=0

* Compare magnitudes of a and b:
          MOVE     B7, B0       ; Copy a and b
          SRL      16, B0       ; Move b into 16 LSBs
          CMPXY    B0, B7       ; Compare a and b
          JRNV     L3           ; Jump if a >= b

* Handle case where a < b; must swap a and b so that a >= b:
          MOVX     B7, B0       ; Copy b into B0
          RL       16, B7       ; Swap a and b halves of B7
          CLR      B12
          MOVY     B11, B12     ; dx_nondiag=0, dy_nondiag=dy_diag

* Calculate initial values of decision variable (d) and
* loop counter:
L3:       ADD      B0, B0       ; B0 = 2 x b
          MOVX     B7, B10      ; B10 = a
          SUB      B10, B0      ; B0 = d (2 x b - a)
          ADDK     1, B10       ; Loop count = a + 1 (in B10)

* Draw line and return to caller:
          LINE     0            ; Inner loop of line algorithm
          RETS     0            ; Return to caller
```

Example 1 This example draws a line from (3,52) to (19,55). Window checking is off, transparency and the pixel processing replace operation are selected, and plane masking is disabled. Assume the following registers have been loaded with these values:

B0	= FFFFFFF1h	Decision variable $d = 2b - a = -15$
B2	= 00520003h	DADDR
B3	= 00000800h	DPTCH (CONVDP=13)
B4	= 00000100h	OFFSET
B5	= 00300003h	WSTART
B6	= 00550025h	WEND
B7	= 00030016h	*b*:*a*; *b*=3 and *a*=22
B9	= 44444444h	COLOR1 (color of the line)
B10	= 00000017h	COUNT (*a*+1)
B11	= 00010001h	Diagonal increment (+1,+1)
B12	= 00000001h	Nondiagonal increment (0,+1)
B13	= FFFFFFFFh	PATTRN (all 1s)

This line is shown in Figure 12-13, represented by ●s.

Before LINE execution, DADDR contains the first pixel to be drawn. During LINE execution, DADDR is updated so that it always points to the next pixel to be drawn. After this example is completed, DADDR equals 0055001Ah. Register B7 contains the X and Y dimensions of the line. Register B10 indicates the number of pixels that are drawn; if you want the endpoint to be drawn (in this case, (19,55)), B10 should equal *a*+1.

B11 contains the XY increment for diagonal moves. You can see the line progressing in a diagonal direction when it moves from (6,52) to (7,53); it is incremented by 1 in both the X and the Y dimensions. B12 contains the XY increment for nondiagonal moves. You can see the line progressing in a nondiagonal direction when it moves from (3,52) to (4,52); it is incremented by 1 in the X dimension.

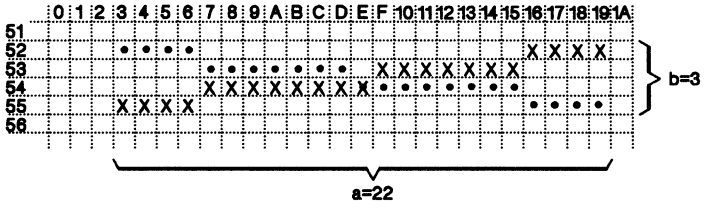


Figure 12-13. LINE Examples

Example 2 This example draws a line from (19,52) to (3,55). Window checking is off, transparency and the pixel processing replace operation are selected, and plane masking is disabled. Assume the following registers have been loaded with these values:

B0	= FFFFFFF1h	Decision variable $d = 2b - a = -15$
B2	= 00520019h	DADDR
B3	= 00000800h	DPTCH (CONVDP=13)
B4	= 00000100h	OFFSET
B5	= 00300003h	WSTART
B6	= 00550025h	WEND
B7	= 00030016h	*b*:*a*; b=3 and a=22
B9	= 22222222h	COLOR1 (color of the line)
B10	= 00000017h	COUNT (a+1)
B11	= 0001FFFFh	Diagonal increment (+1,-1)
B12	= 0000FFFFh	Nondiagonal increment (0,-1)
B13	= FFFFFFFFh	PATTRN (all 1s)

This line is shown in Figure 12-13, represented by Xs.

Before LINE execution, DADDR contains the first pixel to be drawn. During LINE execution, DADDR is updated so that it always points to the next pixel to be drawn. After this example is completed, DADDR equals 00550002h. Register B7 contains the X and Y dimensions of the line. Register B10 indicates the number of pixels that are drawn; if you want the endpoint to be drawn (in this case, (3,55)), B10 should equal a+1.

B11 contains the XY increment for diagonal moves. You can see the line progressing in a diagonal direction when it moves from (F,53) to (E,54); it is decremented by 1 in the X dimension and incremented by 1 in the Y dimension. B12 contains the XY increment for nondiagonal moves. You can see the line progressing in a nondiagonal direction when it moves from (14,53) to (13,53); it is decremented by 1 in the X dimension.

Syntax **LMO** *Rs, Rd*

Execution 31 - (bit number of leftmost 1 in Rs) → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	1	Rs				R	Rd			

Description LMO locates the leftmost (most significant) 1 in the source register. It then loads the 1s complement of the **bit number** of the leftmost-1 bit into the five LSBs of the destination register. The 27 MSBs of the destination register are loaded with 0s. Bit 31 of Rs is the MSB (leftmost) and bit 0 is the LSB. If the source register contains all 0s, then the destination register is loaded with all 0s and status bit Z is set.

You can normalize the contents of the source register by following the LMO instruction with an RL *Rs,Rd* instruction, where Rs is the destination register of the LMO instruction and Rd is the source register.

Rs and Rd must be in the same register file.

Machine States 1,4

Status Bits
N Unaffected
C Unaffected
Z *1* if the source register contents are 0, *0* otherwise
V Unaffected

Examples

Code	Before	After	
	A0	**NCZV**	**A1**
`LMO A0,A1`	00000000h	xx1x	00000000h
`LMO A0,A1`	00000001h	xx0x	0000001Fh
`LMO A0,A1`	00000010h	xx0x	0000001Bh
`LMO A0,A1`	08000000h	xx0x	00000004h
`LMO A0,A1`	80000000h	xx0x	00000000h

Syntax **MMFM** *Rp, register list*

Execution For each register R*n* in the register list,
32 bits of data at the address specified in Rp → R*n*
Rp + 32 → Rp

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	1	0	1	R	Rs			
binary representation of the register list															

Description MMFM loads the contents of a specified list of *either* A or B file registers (not both) from a block of memory.

- The *Rp* operand is a register that points to the first location in the block of memory.
- The *register list* is a list of registers separated by commas (such as A0, A1, A9). These are the registers that MMFM loads new values into.

The MMTM and MMFM instructions are "stack" instructions for storing multiple registers in memory and then retrieving their values. Both instructions use Rp as a "stack pointer" that contains the bit address of the top of the stack. The stack grows toward lower addresses so that the bottom of the stack is the highest address in the stack. MMTM stores the registers in memory. MMFM reverses the action of the MMTM instruction by "popping" register values from memory. At the outset of the MMFM instruction, Rp must contain the address of the 16 LSBs of the highest order register in the list. The LSW is moved into the register, and then the contents of the next consecutive word are moved into the MSW of the register. After a register is "popped", the contents of Rp are incremented by 32 to point to the address of the LSW of the next register to be restored.

Rp and the registers in the list must all be in the same register file. The registers in the list can be specified in any order; the highest order register is always restored first (that is, the value at the top of the stack – the lowest address in the stack – is loaded into the highest order register). Don't include Rp as one of the registers in the register list, because this produces unpredictable results. The original contents of Rp should be aligned on a word-boundary; the alignment of Rp affects the instruction timing as indicated in **Machine States**, below.

The second word of the MMFM instruction is a binary-mask representation of the registers in the list. The R bit (bit 4) in the first word indicates which register file is affected; the bits that are set to 1 in the mask indicate which registers are restored. The bit assignments in the mask are:

	15(MSB)															0(LSB)
	SP	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
or	SP	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0

Machine States

	Cache Enabled	Cache Disabled
Rp Aligned:	3 + 4n + (2)	11 + 4n
Rp Nonaligned:	3 + 6n + (4)	13 + 6n

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

This example restores several B-file registers:

```
MMFM  B0,B1,B2,B3,B7,B12,B13,B14,SP
```

This instruction uses register B0 as the stack pointer. Assume that B0 = 00010000h; this is the address of the top of the stack. MMFM moves the data at this location into the LSW of the SP (which is the highest order register listed in this example). Assume that memory contains the following values before instruction execution:

Address	Data	Address	Data
000100F0h	1111h	00010070h	CCCCh
000100E0h	01B1h	00010060h	BCBCh
000100D0h	2222h	00010050h	DDDDh
000100C0h	0B2B2h	00010040h	BDBDh
000100B0h	3333h	00010030h	EEEEh
000100A0h	03B3h	00010020h	BEBEh
00010090h	7777h	00010010h	FFFFh
00010080h	B7B7h	00010000h	BFBFh

After the MMFM instruction is executed, the registers in the list have the following values:

B0 = 00010100h
B1 = 1111B1B1h
B2 = 2222B2B2h
B4 = 3333B3B3h
B8 = 7777B7B7h
B12 = CCCCBCBCh
B13 = DDDDBDBDh
B14 = EEEEBEBEh
SP = FFFFBFBFh

The other B-file registers (which weren't specified in the register list) are not affected by this instruction. Note that B0 now contains the value 10100h; the last part of the data that was restored was for B1, and B0 points to the word past that data.

Syntax **MMTM** *Rp, register list*

Execution

For each register R*n*in the register list,
Rp - 32 → Rp
32 bits of data at the address specified in R*n* → Rp

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	1	0	0	R	Rd			
binary representation of the register list															

Description

MMTM stores the contents of a specified list of *either* A or B file registers (not both) in memory.

- The *Rp* operand is a register that points to the first location in a block of memory.
- The *register list* is a list of registers that are separated by commas (such as A0, A1, A9). These are the registers that MMTM stores in memory.

The MMTM and MMFM instructions are "stack" instructions for storing multiple registers in memory and then retrieving their values. Both instructions use Rp as a "stack pointer" that contains the bit address of the top of the stack. The stack grows toward lower addresses so that the bottom of the stack is the highest address in the stack. MMTM stores the registers in memory. Before a register's contents are "pushed" onto the stack, the Rp is decremented by 32 bits; the register is then pushed, LSW first. Thus, at the outset of the MMTM instruction, Rp must contain an incremented value. This value is the address where you want to store the LSW of the lowest-order register, *plus 32 bits*; this assures that Rp is predecremented to point to the correct location in memory.

When MMTM execution is complete, the contents of the lowest-order register in the list reside at the highest address in the memory "stack," and Rp points to the address of the highest-order register in the list.

Rp and the registers in the list must all be in the same register file. The registers in the list can be specified in any order; the lowest order register is always saved first. Don't include Rp as one of the registers in the register list, because this produces unpredictable results. The original contents of Rp should be aligned on a word boundary; the alignment of Rp affects the instruction timing as shown in **Machine States**, below.

The second word of the MMFM instruction is a binary-mask representation of the registers in the list. The R bit (bit 4) in the first word indicates which register file is affected; the bits that are set to 1 in the mask indicate which registers are restored. The bit assignments in the mask are:

	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	SP
or	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	SP

15(MSB) **0(LSB)**

Machine States

	Cache Enabled	Cache Disabled
Rp Aligned:	2 + 4n + (2)	8 + 4n + 2
Rp Nonaligned:	2 + 10n + (8)	10(n + 1)

Status Bits

N Set to the sign of the result of 0 - Rp. (This value is typically 1 if the original contents of Rp are positive; otherwise, it is 0. The only exceptions to this are when Rp=80000000h, N is set to 0, and when Rp=0, N is set to 1.)
C Unaffected
Z Unaffected
V Unaffected

Examples

This example saves the values of several A-file registers in memory:

```
MMTM    A1,A0,A2,A4,A8,A12,A13,A14,SP
```

This instruction uses register A1 as the stack pointer. Assume that A1 = 100000h before instruction execution; this value is decremented by 32 to point to the address where the contents of A0 (the lowest order register in the list) are stored. Assume that the registers in the list contain the following values before instruction execution:

A0 = 0000A0A0h	A12 = CCCCACACh
A2 = 2220A2A2h	A13 = DDDDADADh
A4 = 4444A4A4h	A14 = EEEEAEAEh
A8 = 8888A8A8h	SP = FFFFAFAFh

MMTM saves these register values in memory as shown below:

Address	Data	Address	Data
000FFF00h	AFAFh	000FFF80h	A8A8h
000FFF10h	FFFFh	000FFF90h	8888h
000FFF20h	AEAEh	000FFFA0h	A4A4h
000FFF30h	EEEEh	000FFFB0h	4444h
000FFF40h	ADADh	000FFFC0h	A2A2h
000FFF50h	DDDDh	000FFFD0h	2222h
000FFF60h	ACACh	000FFFE0h	A0A0h
000FFF70h	CCCCh	000FFFF0h	0000h

After instruction execution, register A1 = 000FFF00h. Note that A1 now contains the value 0FFF00h; this is the address of the last portion of register data that is saved.

Syntax **MODS** *Rs, Rd*

Execution Rd mod Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	0	Rs				R	Rd			

Description MODS performs a 32-bit signed divide of the 32-bit dividend in the destination register by the 32-bit value in the source register, and returns a 32-bit remainder in the destination register. The remainder is the same sign as the dividend. The original contents of the destination register are always overwritten.

Rs and Rd must be in the same register file.

Machine States

40,43 (normal case)
41,44 if result = 80000000
3,6 if Rs = 0

Status Bits

N Unaffected
C Unaffected
Z Unaffected if RS=0, *1* if quotient is 0, *0* otherwise
V *1* if the quotient overflows (cannot be represented by 32 bits), *0* otherwise

The following conditions set the overflow flag:

- The divisor is 0
- The quotient cannot be contained within 32 bits

Examples

Code	Before		After	
	A0	**A1**	**NCZV**	**A0**
MODS A0,A1	00000000h	00000000h	0x01	00000000h
MODS A0,A1	00000000h	00000007h	0x01	00000007h
MODS A0,A1	00000000h	FFFFFFF9h	0x01	FFFFFFF9h
MODS A0,A1	00000004h	00000008h	0x10	00000000h
MODS A0,A1	00000004h	00000007h	0x00	00000003h
MODS A0,A1	00000004h	00000000h	0x10	00000000h
MODS A0,A1	00000004h	FFFFFFF9h	1x00	FFFFFFFDh
MODS A0,A1	00000004h	FFFFFFF8h	0x10	00000000h
MODS A0,A1	FFFFFFFCh	00000008h	0x10	00000000h
MODS A0,A1	FFFFFFFCh	00000007h	0x00	00000003h
MODS A0,A1	FFFFFFFCh	00000000h	0x10	00000000h
MODS A0,A1	FFFFFFFCh	FFFFFFF9h	1x00	FFFFFFFDh
MODS A0,A1	FFFFFFFCh	FFFFFFF8h	0x10	00000000h

Syntax **MODU** *Rs, Rd*

Execution Rd mod Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	Rs				R	Rd			

Description MODU performs a 32-bit unsigned divide of the 32-bit dividend in the destination register by the 32-bit value in the source register, and returns a 32-bit remainder in the destination register. The original contents of the destination register are always overwritten.

Rs and Rd must be in the same register file.

Machine States

35,38
3,6 if Rs = 0

Status Bits

N Unaffected
C Unaffected
Z Unaffected if RS=0, *1* if quotient is 0, *0* otherwise
V *1* if divisor Rs equals 0, *0* otherwise

Examples

Code	Before		After	
	A0	**A1**	**NCZV**	**A1**
`MODU A0,A1`	00000000h	00000000h	xx01	00000000h
`MODU A0,A1`	00000000h	00000007h	xx01	00000007h
`MODU A0,A1`	00000000h	FFFFFFF9h	xx01	FFFFFFF9h
`MODU A0,A1`	00000004h	00000008h	xx10	00000000h
`MODU A0,A1`	00000004h	00000007h	xx00	00000003h
`MODU A0,A1`	00000004h	00000000h	xx10	00000000h
`MODU A0,A1`	00000004h	FFFFFFF9h	xx00	00000001h

Syntax **MOVB** *Rs, *Rd*

Execution 8 LSBs of Rs → *Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	1	0	Rs				R	Rd			

Description MOVB moves a byte from the source register to the memory address contained in the destination register. The source operand byte is right justified in the source register; only the 8 LSBs of the register are moved. The memory address is a bit address and the field size for the move is 8 bits.

Rs and Rd must be in the same register file.

Machine States 1+(3),7 (when the destination address is aligned on a byte boundary)

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples Assume that memory contains the following values before instruction execution:

Address	**Data**
5000h	0000h
5010h	0000h

Code	**Before**		**After**	
	A0	**A1**	**@5000h**	**@5010h**
`MOVB A0,*A1`	89ABCDEFh	00005000h	00EFh	0000h
`MOVB A0,*A1`	89ABCDEFh	00005001h	01DEh	0000h
`MOVB A0,*A1`	89ABCDEFh	00005009h	0DE00h	0001h
`MOVB A0,*A1`	89ABCDEFh	0000500Ch	F000h	000Eh

Syntax **MOVB** *Rs, *Rd(offset)*

Execution 8 LSBs of Rs → *Rd + offset

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	1	0	Rs				R	Rd			
offset															

Description MOVB moves a byte from the source register to the destination memory address. The source operand byte is right justified in the source register; only the 8 LSBs of the register are moved. The destination memory address is a bit address and is formed by adding the contents of the specified register to the signed 16-bit offset. This is a field move, and the field size for the move is 8 bits.

Rs and Rd must be in the same register file.

Machine States 3+(3),9 (when the destination address is aligned on a byte boundary)

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples Assume that memory contains the following values before instruction execution:

Address	**Data**
10000h	0000h
10010h	0000h

Code	**Before**	**After**		
	A0	**A1**	**@10000h**	**@10010h**
`MOVB A0,*A1(0)`	89ABCDEFh	00010000h	00EFh	0000h
`MOVB A0,*A1(1)`	89ABCDEFh	00010000h	01DEh	0000h
`MOVB A0,*A1(9)`	89ABCDEFh	00010000h	DE00h	0001h
`MOVB A0,*A1(12)`	89ABCDEFh	00010000h	F000h	000Eh
`MOVB A0,*A1(32767)`	89ABCDEFh	00008001h	00EFh	0000h
`MOVB A0,*A1(-32768)`	89ABCDEFh	00018000h	00EFh	0000h

Syntax **MOVB** *Rs, *DAddress*

Execution 8 LSBs of Rs → DAddress

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	1	1	1	R	Rs			
16 LSBs of destination address															
16 MSBs of destination address															

Description MOVB moves a byte from the source register to the destination memory address. The source operand byte is right justified in the source register; only the 8 LSBs of the register are moved. The specified destination memory address is a bit address and the field size for the move is 8 bits.

Rs and Rd must be in the same register file.

Machine States 1+(3),7 (when the destination address is aligned on a byte boundary)

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples Assume that memory contains the following values before instruction execution:

Address	**Data**
5000h	0000h
5010h	0000h

Code	**Before**	**After**	
	A0	**@5000h**	**@5010h**
`MOVB A0,@5000h`	89ABCDEFh	00EFh	0000h
`MOVB A0,@5001h`	89ABCDEFh	01DEh	0000h
`MOVB A0,@5009h`	89ABCDEFh	DE00h	0001h
`MOVB A0,@500Ch`	89ABCDEFh	F000h	000Eh

Syntax **MOVB** **Rs, Rd*

Execution byte at *Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	1	1	Rs				R	Rd			

Description MOVB moves a byte from the memory address contained in the source register to the destination register. The source memory address is a bit address and the field size for the move is 8 bits. When the byte is moved into the destination register, it is right justified and sign extended to 32 bits. This instruction also performs an implicit compare to 0 of the field data.

Rs and Rd must be in the same register file.

Machine States 3,6 (when the source data is aligned on a byte boundary)

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N *1* if the sign-extended data moved into register is negative, *0* otherwise
C Unaffected
Z *1* if the sign-extended data moved into register is 0, *0* otherwise
V *0*

Examples Assume that memory contains the following values before instruction execution:

Address	**Data**
5000h	00EFh
5010h	89ABh

Code	**Before**	**After**	
	A0	**A1**	**NCZV**
`MOVB *A0,A1`	00005000h	FFFFFFEFh	1x00
`MOVB *A0,A1`	00005001h	00000077h	0x00
`MOVB *A0,A1`	00005008h	00000000h	0x10
`MOVB *A0,A1`	0000500Ch	FFFFFFB0h	1x00

Syntax **MOVB** **Rs, *Rd*

Execution byte at *Rs → *Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	1	0	Rs				R	Rd			

Description MOVB moves a byte from the source memory address to the destination memory address. The source address is specified by the contents of Rs, and the destination address is specified by the contents of Rd. Both memory addresses are bit addresses and the field size for the move is 8 bits.

Rs and Rd must be in the same register file.

Machine States 3+(3),7 (when the source data and destination address are aligned on byte boundaries)

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples Assume that memory contains the following values before instruction execution:

Address	**Data**
5000h	CDEF
5010h	89AB
6000h	0000
6010h	0000

Code	**Before**	**After**		
	A0	**A1**	**@6000h**	**@6010h**
`MOVB *A0,*A1`	00005000h	00006000h	00EFh	0000h
`MOVB *A0,*A1`	00005000h	00006001h	01DEh	0000h
`MOVB *A0,*A1`	00005000h	00006009h	DE00h	0001h
`MOVB *A0,*A1`	00005000h	0000600Ch	F000h	000Eh
`MOVB *A0,*A1`	00005001h	00006000h	00F7h	0000h
`MOVB *A0,*A1`	00005001h	00006001h	01EEh	0000h
`MOVB *A0,*A1`	00005001h	00006009h	EE00h	0001h
`MOVB *A0,*A1`	00005001h	0000600Ch	7000h	000Fh
`MOVB *A0,*A1`	00005009h	00006000h	00E6h	0000h
`MOVB *A0,*A1`	00005009h	00006001h	01CCh	0000h
`MOVB *A0,*A1`	00005009h	00006009h	CC00h	0001h
`MOVB *A0,*A1`	00005009h	0000600Ch	6000h	000Eh
`MOVB *A0,*A1`	0000500Ch	00006000h	00BCh	0000h
`MOVB *A0,*A1`	0000500Ch	00006001h	0178h	0000h
`MOVB *A0,*A1`	0000500Ch	00006009h	7800h	0001h
`MOVB *A0,*A1`	0000500Ch	0000600Ch	C000h	000Bh

Syntax **MOVB** **Rs(offset), Rd*

Execution byte at (*Rs + offset) → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	1	1	Rs				R	Rd			
offset															

Description MOVB moves a byte from the source memory address to the destination register. The source memory address is a bit address and is formed by adding the contents of the specified register to the signed 16-bit offset. The field size is 8 bits. When the byte is moved into the destination register, it is right justified and sign extended to 32 bits. This instruction also performs an implicit compare to 0 of the field data.

Rs and Rd must be in the same register file.

Machine States 5,11 (when the source data is aligned on a byte boundary)

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N *1* if the sign-extended data moved into register is negative, *0* otherwise
C Unaffected
Z *1* if the sign-extended data moved into register is 0, *0* otherwise
V *0*

Examples Assume that memory contains the following values before instruction execution:

Address	**Data**
10000h	00EFh
10010h	89ABh

Code	**Before**	**After**	
	A0	**A1**	**NCZV**
`MOVB *A0(0),A1`	00010000h	FFFFFFEFh	1x00
`MOVB *A0(1),A1`	00010000h	00000077h	0x00
`MOVB *A0(8),A1`	00010000h	00000000h	0x10
`MOVB *A0(12),A1`	00010000h	FFFFFFB0h	1x00
`MOVB *A0(32767),A1`	00008001h	FFFFFFEFh	1x00
`MOVB *A0(-32768),A1`	00018000h	FFFFFFEFh	1x00

Syntax **MOVB** **Rs(SOffset), *Rd(DOffset)*

Execution byte at (*Rs + SOffset) → (*Rd + DOffset)

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	0	Rs				R	Rd			
source offset															
destination offset															

Description MOVB moves a byte from the source memory address to the destination memory address. Both the source and destination memory addresses are bit addresses and are formed by adding the contents of the specified register to its respective signed 16-bit offset. The field size is 8 bits.

Rs and Rd must be in the same register file.

Machine States 5+(3),9 (when the source data and destination address are aligned on byte boundaries)

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples Assume that memory contains the following values before instruction execution:

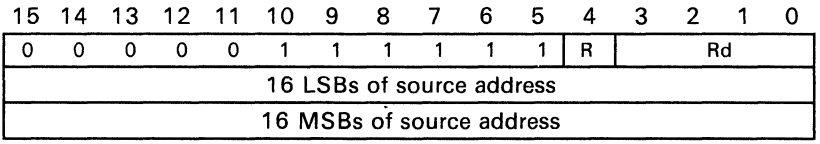
Address	Data
10000h	CDEFh
10010h	89ABh
11000h	0000h
11010h	0000h

Code	Before		After	
	A0	A1	@11000h	@11010h
MOVB *A0(0),*A1(0)	00010000h	00011000h	00EFh	0000h
MOVB *A0(0),*A1(1)	00010000h	00011000h	01DEh	0000h
MOVB *A0(0),*A1(9)	00010000h	00011000h	DE00h	0001h
MOVB *A0(0),*A1(12)	00010000h	00011000h	F000h	000Eh
MOVB *A0(0),*A1(32767)	00010000h	00009001h	00EFh	0000h
MOVB *A0(0),*A1(-32768)	00010000h	00019000h	00EFh	0000h
MOVB *A0(12),*A1(0)	00010000h	00011000h	00BCh	0000h
MOVB *A0(12),*A1(1)	00010000h	00011000h	0178h	0000h
MOVB *A0(12),*A1(9)	00010000h	00011000h	7800h	0001h
MOVB *A0(12),*A1(12)	00010000h	00011000h	C000h	000Bh
MOVB *A0(12),*A1(32767)	00010000h	00009001h	00BCh	0000h
MOVB *A0(12),*A1(-32768)	00010000h	00019000h	00BCh	0000h
MOVB *A0(32767),*A1(0)	00008001h	00011000h	00EFh	0000h
MOVB *A0(32767),*A1(1)	00008001h	00011000h	01DEh	0000h
MOVB *A0(32767),*A1(9)	00008001h	00011000h	DE00h	0001h
MOVB *A0(32767),*A1(12)	00008001h	00011000h	F000h	000Eh
MOVB *A0(32767),*A1(32767)	00008001h	00009001h	00EFh	0000h
MOVB *A0(32767),*A1(-32678)	00008001h	00019000h	00EFh	0000h
MOVB *A0(-32768),*A1(0)	00018000h	00011000h	00EFh	0000h
MOVB *A0(-32768),*A1(1)	00018000h	00011000h	01DEh	0000h
MOVB *A0(-32768),*A1(9)	00018000h	00011000h	DE00h	0001h
MOVB *A0(-32768),*A1(12)	00018000h	00011000h	F000h	000Eh
MOVB *A0(-32768),*A1(32767)	00018000h	00009001h	00EFh	0000h
MOVB *A0(-32768),*A1(-32678)	00018000h	00019000h	00EFh	0000h

Syntax **MOVB** *@SAddress, Rd*

Execution byte at SAddress → Rd

Instruction Words



Description MOVB moves a byte from the source memory address to the destination register. The specified source memory address is a bit address and the field size for the move is 8 bits. When the byte is moved into the destination register, it is right justified and sign extended to 32 bits. This instruction also performs an implicit compare to 0 of the field data.

Rs and Rd must be in the same register file.

Machine States 5,14 (when the source data is aligned on a byte boundary)

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N *1* if the sign-extended data moved into register is negative, *0* otherwise
C Unaffected
Z *1* if the sign-extended data moved into register is 0, *0* otherwise
V *0*

Examples Assume that memory contains the following values before instruction execution:

Address	**Data**
10000h	00EFh
10010h	89ABh

Code	**After**	
	A1	**NCZV**
`MOVB @10000h,A1`	FFFFFFEFh	1x00
`MOVB @10001h,A1`	00000077h	0x00
`MOVB @10008h,A1`	00000000h	0x10
`MOVB @1000Ch,A1`	FFFFFFB0h	1x00

Syntax **MOVB** *@SAddress, @DAddress*

Execution byte at SAddress → DAddress

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0
16 LSBs of source address															
16 MSBs of source address															
16 LSBs of destination address															
16 MSBs of destination address															

Description MOVB moves a byte from the source memory address to the destination memory address. Both the source and destination addresses are interpreted as bit addresses and the field size for the move is 8 bits.

Machine States 7+(3),25 (when the source data and destination address are aligned on byte boundaries)

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples Assume that memory contains the following values before instruction execution:

Address	**Data**
10000h	CDEFh
10010h	89ABh
11000h	0000h
11010h	0000h

Code	After	
	@11000h	@11010h
MOVB @10000h,@11000h	00EFh	0000h
MOVB @10000h,@11001h	01DEh	0000h
MOVB @10000h,@11009h	DE00h	0001h
MOVB @10000h,@1100Ch	F000h	000Eh
MOVB @10001h,@11000h	00F7h	0000h
MOVB @10001h,@11001h	01EEh	0000h
MOVB @10001h,@11009h	EE00h	0001h
MOVB @10001h,@1100Ch	7000h	000Fh
MOVB @10009h,@11000h	00E6h	0000h
MOVB @10009h,@11001h	01CCh	0000h
MOVB @10009h,@11009h	CC00h	0001h
MOVB @10009h,@1100Ch	6000h	000Eh
MOVB @1000Ch,@11000h	00BCh	0000h
MOVB @1000Ch,@11001h	0178h	0000h
MOVB @1000Ch,@11009h	7800h	0001h
MOVB @1000Ch,@1100Ch	C000h	000Bh

Syntax **MOVE** *Rs, Rd*

Execution Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	M	Rs				R	Rd			

Description MOVE moves the 32 bits of data from the source register to the destination register. Note that this is not a field move; therefore, the field size has no effect. The source and destination registers can be any of the 31 locations in the on-chip register file. Note that this is the only MOVE instruction that allows the source and destination registers to be in different files. This instruction also performs an implicit compare to 0 of the register data.

Fields The assembler sets bit 9 (the M bit) in the instruction word to specify whether the move is within a register file or if it crosses the register files. The assembler sets M to 0 if the source and destination registers are in the same file; it sets M to 1 if the registers are in different files.

The assembler sets bit 4 (the R bit) in the instruction word to specify which file the registers are in. The assembler sets R to 0 if the registers are in file A; it sets R to 1 if the registers are in file B.

Note that when M=0, R specifies the register file for *both* registers; if M=1, R specifies the register file for the *source register*

Machine States 1, 4

Status Bits
N *1* if the 32-bit data moved is negative, *0* otherwise
C Unaffected
Z *1* if the 32-bit data moved is 0, *0* otherwise
V *0*

Examples

Code	Before A0	After A1	B1	NCZV
MOVE A0,A1	0000FFFFh	0000FFFFh	xxxxxxxxh	0x00
MOVE A0,A1	00000000h	00000000h	xxxxxxxxh	0x10
MOVE A0,A1	FFFFFFFFh	FFFFFFFFh	xxxxxxxxh	1x00
MOVE A0,B1	0000FFFFh	xxxxxxxxh	0000FFFFh	0x00
MOVE A0,B1	00000000h	xxxxxxxxh	00000000h	0x10
MOVE A0,B1	FFFFFFFFh	xxxxxxxxh	FFFFFFFFh	1x00

Syntax **MOVE** *Rs, *Rd [, F]*

Execution field in Rs → *Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	F	Rs				R	Rd			

Description MOVE moves a field from the source register to the memory address contained in the destination register. This memory address is a bit address. The field size for the move is 1–32 bits, depending on the selected field size; the field is right justified within the source register. The optional F parameter determines the field size and extension for the move:

F=0 selects FS0
F=1 selects FS1

The SETF instruction sets the field size and extension. Rs and Rd must be in the same register file.

Machine States The following typical cases assume that the destination address is aligned on a 16-bit boundary:

16-Bit Field	**32-Bit Field**
1+(1),5	1+(3),7

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples Assume that memory contains the following values before instruction execution:

Address	**Data**
15500h	0000h
15510h	0000h
15520h	0000h

Register A0 = FFFFFFFFh

Code	**Before**		**After**		
	A1	**FS0/1**	**@15500h**	**@15510h**	**@15520h**
`MOVE A0,*A1,0`	00015500h	5/x	001Fh	0000h	0000h
`MOVE A0,*A1,1`	00015503h	x/8	07F8h	0000h	0000h
`MOVE A0,*A1,0`	00015508h	13/x	FF00h	001Fh	0000h
`MOVE A0,*A1,1`	0001550Bh	x/16	F800h	07FFh	0000h
`MOVE A0,*A1,0`	0001550Dh	19/x	E000h	FFFFh	0000h
`MOVE A0,*A1,1`	0001550Ch	x/24	F000h	FFFFh	000Fh
`MOVE A0,*A1,0`	00015512h	27/x	0000h	FFFCh	1FFFh
`MOVE A0,*A1,1`	00015510h	x/32	0000h	FFFFh	FFFFh

Syntax **MOVE** *Rs, *Rd+ [, F]*

Execution field in Rs → *Rd
Rd + field size → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	F	Rs				R	Rd			

Description MOVE moves a field from the source register to the memory address contained in the destination register. This memory address is a bit address. After the move, the contents of the destination register are postincremented by the selected field size. The field size for the move is 1–32 bits, depending on the selected field size; the field is right justified within the source register. The optional F parameter determines the field size and extension for the move:

F=0 selects FS0
F=1 selects FS1

The SETF instruction sets the field size and extension. Rs and Rd must be in the same register file.

Machine States The following typical cases assume that the destination address is aligned on a 16-bit boundary:

16-Bit Field	**32-Bit Field**
1+(1),5	1+(3),7

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples Assume that memory contains the following values before instruction execution:

Address	**Data**
15500h	0000h
15510h	0000h
15520h	0000h

Register A0 = FFFFFFFFh

Code	Before		After			
	A1	FS0/1	A1	@15500h	@15510h	@15520h
MOVE A0,*A1+,0	00015528h	5/x	0001552Dh	0000h	0000h	1F00h
MOVE A0,*A1+,1	00015525h	x/8	0001552Dh	0000h	0000h	1FE0h
MOVE A0,*A1+,0	00015520h	13/x	0001552Dh	0000h	0000h	1FFFh
MOVE A0,*A1+,1	0001551Dh	x/16	0001552Dh	0000h	E000h	1FFFh
MOVE A0,*A1+,0	00015516h	19/x	00015529h	0000h	FFC0h	01FFh
MOVE A0,*A1+,1	00015507h	x/24	0001551Fh	FF80h	7FFFh	0000h
MOVE A0,*A1+,0	00015507h	27/x	0001551Fh	FF80h	FFFFh	0003h
MOVE A0,*A1+,1	00015500h	x/32	00015520h	FFFFh	FFFFh	0000h

Syntax **MOVE** *Rs, -*Rd [, F]*

Execution
Rd - field size → Rd
field in Rs → *Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	F	Rs				R	Rd			

Description MOVE moves a field from the source register to the memory address contained in the destination register; the destination address is predecremented by the field size. The memory address is a bit address. Before the move, the field size is subtracted from the contents of the destination register to determine the location that the field is moved to. (This value is also the final value for the register.)

The field size for the move is 1–32 bits, depending on the selected field size; the field is right justified within the source register. The optional F parameter determines the field size and extension for the move:

F=0 selects FS0
F=1 selects FS1

The SETF instruction sets the field size and extension. Rs and Rd must be in the same register file.

Machine States The following typical cases assume that the destination address is aligned on a 16-bit boundary:

16-Bit Field	**32-Bit Field**
2+(1),6	2+(3),8

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits
N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples Assume that memory contains the following values before instruction execution:

Address	**Data**
15500h	0000h
15510h	0000h
15520h	0000h

Register A0 = FFFFFFFFh

Code	Before		After			
	A1	FS0/1	A1	@15500h	@15510h	@15520h
MOVE A0,-*A1,0	0001530h	5/x	000152Bh	0000h	0000h	F800h
MOVE A0,-*A1,1	000152Dh	x/8	0001525h	0000h	0000h	1FE0h
MOVE A0,-*A1,0	0001528h	13/x	000151Bh	0000h	F800h	00FFh
MOVE A0,-*A1,1	0001528h	x/16	0001518h	0000h	FF00h	00FFh
MOVE A0,-*A1,0	0001523h	19/x	0001510h	0000h	FFFFh	0007h
MOVE A0,-*A1,1	0001520h	x/24	0001508h	FF00h	FFFFh	0000h
MOVE A0,-*A1,0	0001524h	27/x	0001509h	FE00h	FFFFh	000Fh
MOVE A0,-*A1,1	0001520h	x/32	0001500h	FFFFh	FFFFh	0000h

Syntax **MOVE** *Rs, *Rd(offset) [, F]*

Execution field in Rs → *(Rd + offset)

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	F	Rs				R	Rd			
offset															

Description MOVE moves a field from the source register to the destination memory memory address. The destination memory address is a bit address and is formed by adding the contents of the destination register to the signed 16-bit offset. The field size for the move is 1–32 bits, depending on the selected field size; the field is right justified within the source register. The optional F parameter determines the field size and extension for the move:

F=0 selects FS0
F=1 selects FS1

The SETF instruction sets the field size and extension. Rs and Rd must be in the same register file.

Machine States The following typical cases assume that the destination address is aligned on a 16-bit boundary:

16-Bit Field	**32-Bit Field**
3+(1),7	3+(3),9

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples Assume that memory contains the following values before instruction execution:

Address	Data
15530h	0000h
15540h	0000h
15550h	0000h

Register A0 = FFFFFFFFh

Code	Before		After		
	A1	FS0/1	@15530h	@15540h	@15550h
`MOVE A0,*A1(0000h),1`	00015530h	x/1	0001h	0000h	0000h
`MOVE A0,*A1(0001h),0`	0001552Fh	5/x	001Fh	0000h	0000h
`MOVE A0,*A1(000Fh),0`	0001552Dh	8/x	F000h	000Fh	0000h
`MOVE A0,*A1(0020h),1`	0001551Ch	x/13	F000h	01FFh	0000h
`MOVE A0,*A1(00FFh),0`	00015435h	16/x	FFF0h	000Fh	0000h
`MOVE A0,*A1(0FFFh),0`	00014531h	19/x	FFFFh	0007h	0000h
`MOVE A0,*A1(7FFFh),1`	0000D531h	x/22	FFFFh	003Fh	0000h
`MOVE A0,*A1(0FFF2h),1`	00015540h	x/25	FFFCh	07FFh	0000h
`MOVE A0,*A1(8000h),0`	0001D530h	27/x	FFFFh	07FFh	0000h
`MOVE A0,*A1(0FFF0h),0`	00015540h	31/x	FFFFh	7FFFh	0000h
`MOVE A0,*A1(0FFECh),1`	00015548h	x/31	FFF0h	FFFFh	0007h
`MOVE A0,*A1(0FFECh),0`	0001554Dh	32/x	FE00h	FFFFh	01FFh
`MOVE A0,*A1(001Dh),0`	00015520h	32/x	E000h	FFFFh	1FFFh
`MOVE A0,*A1(0020h),1`	00015520h	x/32	0000h	FFFFh	FFFFh

Syntax **MOVE** *Rs, @DAddress [, F]*

Execution field in Rs → DAddress

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	F	1	1	0	0	R	Rs			
16 LSBs of destination address															
16 MSBs of destination address															

Description MOVE moves a field from the source register to the destination memory address. The specified destination memory address is a linear bit address. The field size for the move is 1–32 bits, depending on the selected field size; the field is right justified within the source register. The optional F parameter determines the field size and extension for the move:

F=0 selects FS0
F=1 selects FS1

SETF sets the field size and extension.

Machine States The following typical cases assume that the destination address is aligned on a 16-bit boundary:

16-Bit Field	**32-Bit Field**
3+(1),7	3+(3),9

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples Assume that memory contains these values before instruction execution:

Address	**Data**
15500h	0000h
15510h	0000h
15520h	0000h

Register A0 = FFFFFFFFh

Code	**Before**	**After**		
	FS0/1	**@15500h**	**@15510h**	**@15520h**
`MOVE A0,@15500h,0`	5/x	001Fh	0000h	0000h
`MOVE A0,@15503h,1`	x/8	07F8h	0000h	0000h
`MOVE A0,@15508h,0`	13/x	FF00h	001Fh	0000h
`MOVE A0,@1550Bh,1`	x/16	F800h	07FFh	0000h
`MOVE A0,@1550Dh,0`	19/x	E000h	FFFFh	0000h
`MOVE A0,@15510h,1`	x/24	0000h	FFFFh	00FFh
`MOVE A0,@15512h,0`	27/x	0000h	FFFCh	1FFFh
`MOVE A0,@1550Ch,1`	x/32	F000h	FFFFh	0FFFh

Syntax **MOVE** **Rs, Rd [, F]*

Execution field at *Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	F	Rs				R	Rd			

Description MOVE moves a field from the source memory address to the destination register. The contents of the source register specify the address of the field. When the field is moved into the destination register, it is right justified and sign extended or zero extended to 32 bits, according to the value of FE. This instruction also performs an implicit compare to 0 of the field data.

The field size for the move is 1–32 bits, depending on the selected field size; the optional F parameter determines the field size and extension for the move:

F=0 selects FS0
F=1 selects FS1

The SETF instruction sets the field size and extension. Rs and Rd must be in the same register file.

Machine States The following typical cases assume that the source data is aligned on a 16-bit boundary:

16-Bit Field	**32-Bit Field**
3,6	5,8

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N *1* if the field-extended data moved to register is negative, *0* otherwise
C Unaffected
Z *1* if the field-extended data moved to register is 0, *0* otherwise
V *0*

Examples Assume that memory contains the following values before instruction execution:

Address	Data
15500h	7770h
15510h	7777h

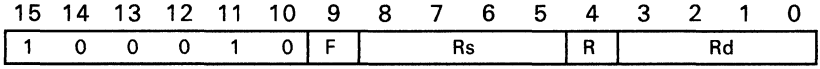
Register A0 = 00015500h

Code	Before		After	
	FS0/1	FE0/1	A1	NCZV
`MOVE *A0,A1,1`	x/1	x/1	00000000h	0x10
`MOVE *A0,A1,0`	5/x	0/x	00000010h	0x00
`MOVE *A0,A1,1`	x/5	x/1	FFFFFFF0h	1x00
`MOVE *A0,A1,0`	12/x	1/x	00000770h	0x00
`MOVE *A0,A1,1`	x/12	x/0	00000770h	0x00
`MOVE *A0,A1,0`	18/x	0/x	00037770h	0x00
`MOVE *A0,A1,1`	x/18	x/1	FFFF7770h	1x00
`MOVE *A0,A1,0`	27/x	1/x	FF777770h	1x00
`MOVE *A0,A1,1`	x/27	x/0	07777770h	0x00
`MOVE *A0,A1,0`	31/x	0/x	77777770h	0x00
`MOVE *A0,A1,1`	x/31	x/1	F7777770h	1x00
`MOVE *A0,A1,0`	32/x	x/x	77777770h	0x00

Syntax **MOVE** **Rs, *Rd [, F]*

Execution field at *Rs → *Rd

Instruction Words



Description MOVE moves a field from a source address to a destination address. Both memory addresses are bit addresses; the source register contains the address of the field and the destination register specifies the address that the field is moved to. The field size for the move is 1–32 bits, depending on the selected field size; the optional F parameter determines the field size and extension for the move:

F=0 selects FS0
F=1 selects FS1

SETF sets the field size and extension. Rs and Rd must be in the same register file.

Machine States The following typical cases assume that the source data and the destination address are aligned on 16-bit boundaries:

16-Bit Field	**32-Bit Field**
3+(1),7	5+(3),11

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples Assume that memory contains the following values before instruction execution:

Address	Data	Address	Data
15500h	FFFFh	15530h	0000h
15510h	FFFFh	15540h	0000h
15520h	FFFFh	15550h	0000h

Code	Before			After		
	A0	A1	FS0/1	@15530h	@15540h	@15550h
MOVE *A0,*A1,1	00015500h	00015530h	x/1	0001h	0000h	0000h
MOVE *A0,*A1,0	00015500h	00015534h	5/x	01F0h	0000h	0000h
MOVE *A0,*A1,1	00015500h	0001553Ah	x/10	FC00h	000Fh	0000h
MOVE *A0,*A1,0	00015500h	0001553Fh	19/x	8000h	FFFFh	0003h
MOVE *A0,*A1,1	00015504h	00015530h	x/7	007Fh	0000h	0000h
MOVE *A0,*A1,0	0001550Ah	00015530h	13/x	1FFFh	0000h	0000h
MOVE *A0,*A1,1	0001550Dh	00015534h	x/8	0FF0h	0000h	0000h
MOVE *A0,*A1,0	0001550Dh	00015530h	28/x	FFFFh	0FFFh	0000h
MOVE *A0,*A1,1	00015505h	00015535h	x/23	FFE0h	0FFFh	0000h
MOVE *A0,*A1,0	00015508h	00015536h	31/x	FFC0h	FFFFh	001Fh
MOVE *A0,*A1,1	00015508h	00015531h	x/31	FFFEh	FFFFh	0000h
MOVE *A0,*A1,0	0001550Ah	00015530h	32/x	FFFFh	FFFFh	0000h
MOVE *A0,*A1,0	00015500h	0001553Ah	x/32	FC00h	FFFFh	03FFh

Syntax **MOVE** **Rs+, Rd [, F]*

Execution
field at *Rs → Rd
Rs + field size → Rs

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	1	F	Rs				R	Rd			

Description MOVE moves a field from memory to the destination register. The source register contains the address of the field; after the move, the contents of the source register are incremented by the field size. When the field is moved into the destination register, it is right justified and sign extended or zero extended, as specified by the selected field extension. This instruction also performs an implicit compare to 0 of the field data.

The field size for the move is 1–32 bits, depending on the selected field size; the optional F parameter determines the field size and extension for the move:

F=0 selects FS0
F=1 selects FS1

The SETF instruction sets the field size and extension. Rs and Rd must be in the same register file.

Machine States The following typical cases assume that the source data is aligned on a 16-bit boundary:

16-Bit Field	**32-Bit Field**
3,6	5,8

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits
N *1* if the field-extended data moved to register is negative, *0* otherwise
C Unaffected
Z *1* if the field-extended data moved to register is 0, *0* otherwise
V *0*

Examples Assume that memory contains the following values before instruction execution:

Address	Data
15500h	7770h
15510h	7777h

Register A0 = 00015500h

Code	Before		After		
	FS0/1	FE0/1	A0	A1	NCZV
`MOVE *A0+,A1,1`	x/1	x/0	00015501h	00000000h	0x10
`MOVE *A0+,A1,1`	x/5	x/0	00015505h	00000010h	0x00
`MOVE *A0+,A1,0`	5/x	1/x	00015505h	FFFFFFF0h	1x00
`MOVE *A0+,A1,0`	12/x	0/x	0001550Ch	00000770h	0x00
`MOVE *A0+,A1,1`	x/12	x/1	0001550Ch	00000770h	0x00
`MOVE *A0+,A1,0`	18/x	1/x	00015512h	FFFF7770h	1x00
`MOVE *A0+,A1,1`	x/18	x/0	00015512h	00037770h	0x00
`MOVE *A0+,A1,0`	27/x	0/x	0001551Bh	07777770h	0x00
`MOVE *A0+,A1,1`	x/27	x/1	0001551Bh	FF777770h	1x00
`MOVE *A0+,A1,0`	31/x	1/x	0001551Fh	F7777770h	1x00
`MOVE *A0+,A1,1`	x/31	x/0	0001551Fh	77777770h	0x00
`MOVE *A0+,A1,0`	32/x	x/x	00015520h	77777770h	0x00

Syntax **MOVE** **Rs+, *Rd+ [, F]*

Execution
field at *Rs → *Rd
Rs + field size → Rs
Rd + field size → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	F	Rs				R	Rd			

Description MOVE moves a field from one memory address to another. The source register contains the bit address of the field; the destination register contains the bit address of field's destination. After the move, the contents of both instructions are incremented by the field size.

The field size for the move is 1–32 bits, depending on the selected field size; the optional F parameter determines the field size and extension for the move:

F=0 selects FS0
F=1 selects FS1

The SETF instruction sets the field size and extension.

If Rs and Rd specify the same register, the data read from the location pointed to by the original contents of Rs is written to the location pointed to by the incremented value of Rs(Rd). Rs and Rd must be in the same register file.

Machine States The following typical cases assume that the source data and the destination address are aligned on 16-bit boundaries:

16-Bit Field	**32-Bit Field**
4,7	6+(2),11

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits
N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples Assume that memory contains the following values before instruction execution:

Address	Data	Address	Data
15500h	FFFFh	15530h	0000h
15510h	FFFFh	15540h	0000h
15520h	FFFFh	15550h	0000h

```
MOVE *A0+,*A1+,F
```

Before

F	A0	A1	FS0/1
1	00015500h	0001553Dh	x/1
0	00015505h	00015538h	5/x
1	0001550Ah	0001553Fh	x/10
0	0001550Dh	00015530h	19/x
1	00015510h	00015532h	x/7
0	00015511h	0001553Ah	13/x
1	00015513h	0001553Fh	x/8
0	00015510h	0001553Ah	28/x
1	00015518h	00015534h	x/23
0	00015510h	00015530h	31/x
1	00015511h	0001553Dh	x/31
0	00015510h	0001553Fh	32/x
1	00015500h	00015530h	x/32

After

A0	A1	@15530h	@15540h	@15550h
00015501h	0001553Eh	2000h	0000h	0000h
0001550Ah	0001553Dh	1F00h	0000h	0000h
00015514h	00015549h	8000h	01FFh	0000h
00015520h	00015543h	FFFFh	0007h	0000h
00015517h	00015539h	01FCh	0000h	0000h
0001551Eh	00015547h	FC00h	007Fh	0000h
0001551Bh	00015547h	8000h	007Fh	0000h
0001552Ch	00015556h	FC00h	FFFFh	003Fh
0001552Fh	0001554Bh	FFF0h	07FFh	0000h
0001552Fh	0001554Fh	FFFFh	7FFFh	0000h
00015530h	0001555Ch	E000h	FFFFh	0FFFh
00015530h	0001555Fh	8000h	FFFFh	7FFFh
00015520h	00015550h	FFFFh	FFFFh	0000h

Syntax **MOVE** *-*Rs, Rd [, F]*

Execution Rs - field size → Rs
field at *Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	F	Rs				R	Rd			

Description MOVE moves a field from memory to the destination register. The source register contains a bit address; before the move, the contents of the source register are decremented by the field size to form the address of the field. (This value is also the final value for the register.)

The field size for the move is 1–32 bits, depending on the selected field size; the optional F parameter determines the field size and extension for the move:

F=0 selects FS0
F=1 selects FS1

The SETF instruction sets the field size and extension.

When the field is moved into the destination register, it is right justified and sign extended or zero extended to 32 bits according to the value of FE for the particular F bit selected. This instruction also performs an implicit compare to 0 of the field data.

Rs and Rd must be in the same register file. If Rs and Rd are the same register, the pointer information is overwritten by the data fetched.

Machine States The following typical cases assume that the source data is aligned on a 16-bit boundary:

16-Bit Field	**32-Bit Field**
4,7	6,9

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N *1* if the field-extended data moved to register is negative, *0* otherwise
C Unaffected
Z *1* if the field-extended data moved to register is 0, *0* otherwise
V *0*

Examples Assume that memory contains the following values before instruction execution:

Address	Data
15500h	7770h
15510h	7777h

Register A0 = 00015520h

Code	Before		After		
	FS0/1	FE0/1	A0	A1	NCZV
MOVE -*A0,A1,1	x/1	x/0	0001551Fh	00000000h	0x10
MOVE -*A0,A1,0	5/x	1/x	0001551Bh	0000000Eh	0x00
MOVE -*A0,A1,1	x/5	x/0	0001551Bh	0000000Eh	0x00
MOVE -*A0,A1,0	12/x	0/x	00015514h	00000777h	0x00
MOVE -*A0,A1,1	x/12	x/1	00015514h	00000777h	0x00
MOVE -*A0,A1,0	18/x	1/x	0001550Eh	0001DDDDh	0x00
MOVE -*A0,A1,1	x/18	x/0	0001550Eh	0001DDDDh	0x00
MOVE -*A0,A1,0	27/x	0/x	00015505h	03BBBBBBh	0x00
MOVE -*A0,A1,1	x/27	x/1	00015505h	03BBBBBBh	0x00
MOVE -*A0,A1,0	31/x	1/x	00015501h	3BBBBBB8h	0x00
MOVE -*A0,A1,1	x/31	x/0	00015501h	3BBBBBB8h	0x00
MOVE -*A0,A1,0	32/x	x/x	00015500h	77777770h	0x00

Syntax **MOVE** *-*Rs, -*Rd [, F]*

Execution
Rs - field size → Rs
Rd - field size → Rd
(field)*Rs → (field)*Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	0	F	Rs				R	Rd			

Description MOVE moves a field from one memory address to another. Both registers contain bit addresses; before the move, the contents of both registers are decremented by the field size. The source register then contains the address of the field, and the destination register specifies the destination address for the move.

The field size for the move is 1–32 bits, depending on the selected field size; the optional F parameter determines the field size and extension for the move:

F=0 selects FS0
F=1 selects FS1

The SETF instruction sets the field size and extension.

Rs and Rd must be in the same register file. If Rs and Rd are the same register, then the final contents of the register are its original contents decremented by twice the field size.

Machine States The following typical cases assume that the source data and the destination address are aligned on 16-bit boundaries:

16-Bit Field	**32-Bit Field**
4+(1),8	6+(3),12

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits
N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples Assume that memory contains the following values before instruction execution:

Address	Data	Address	Data
15500h	FFFFh	15530h	0000h
15510h	FFFFh	15540h	0000h
15520h	FFFFh	15550h	0000h

```
MOVE -*A0,-*A1,F
```

Before				After				
F	**A0**	**A1**	**FS0/1**	**A0**	**A1**	**@15530h**	**@15540h**	**@15550h**
1	00015501h	00015531h	x/1	00015500h	00015530h	0001h	0000h	0000h
0	00015505h	00015539h	5/x	00015500h	00015534h	01F0h	0000h	0000h
1	0001550Ah	00015544h	x/10	0001550h0	0001553Ah	FC00h	000Fh	0000h
0	00015513h	00015552h	19/x	00015500h	0001553Fh	8000h	FFFFh	0003h
1	0001550Bh	00015537h	x/7	00015504h	00015530h	007Fh	0000h	0000h
0	00015517h	0001553Dh	13/x	0001550Ah	00015530h	1FFFh	0000h	0000h
1	00015515h	0001553Ch	x/8	0001550Dh	00015534h	0FF0h	0000h	0000h
0	00015529h	0001554Ch	28/x	0001550Dh	00015530h	FFFFh	0FFFh	0000h
1	0001551Ch	0001554Ch	x/23	00015505h	00015535h	FFE0h	0FFFh	0000h
0	00015527h	00015555h	31/x	00015508h	00015536h	FFC0h	FFFFh	001Fh
1	00015527h	00015550h	x/31	00015508h	00015531h	FFFEh	FFFFh	0000h
0	0001552Ah	00015550h	32/x	0001550Ah	00015530h	FFFFh	FFFFh	0000h
1	00015520h	0001555Ah	x/32	00015500h	0001553Ah	FC00h	FFFFh	03FFh

Syntax **MOVE** **Rs(offset), Rd [, F]*

Execution field at (*Rs + offset) → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	F	Rs				R	Rd			
offset															

Description This MOVE instruction moves a field from a memory address to the destination register. The address of the source data is formed by adding a signed, 16-bit offset to the contents of Rs. When the field is moved into the destination register, it is right justified and sign extended or zero extended to 32 bits, according to the value of the current FE bit. This instruction also performs an implicit compare to 0 of the field data.

The field size for the move is 1–32 bits, depending on the selected field size; the field is right justified within the source register. The optional F parameter determines the field size and extension for the move:

F=0 selects FS0
F=1 selects FS1

The SETF instruction sets the field size and extension.

Rs and Rd must be in the same register file.

Machine States The following typical cases assume that the source data is aligned on a 16-bit boundary:

16-Bit Field	**32-Bit Field**
5,11	7,13

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N *1* if the field-extended data moved to register is negative, *0* otherwise
C Unaffected
Z *1* if the field-extended data moved to register is 0, *0* otherwise
V *0*

Examples Assume that memory contains the following values before instruction execution:

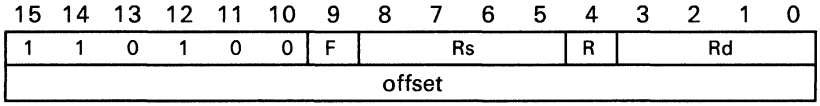
Address	Data
15530h	3333h
15540h	4444h
15550h	5555h

Code	Before			After	
	A0	FS0/1	FE0/1	A1	NCZV
`MOVE *A0(0000h),A1,1`	00015530h	x/1	x/1	FFFFFFFFh	1x00
`MOVE *A0(0003h),A1,1`	0001552Fh	x/2	x/0	00000000h	0x10
`MOVE *A0(0001h),A1,0`	0001552Fh	5/x	0/x	00000013h	0x00
`MOVE *A0(000Fh),A1,0`	0001552Dh	8/x	1/x	00000043h	0x00
`MOVE *A0(0020h),A1,1`	0001551Ch	x/13	x/0	00000443h	0x00
`MOVE *A0(00FFh),A1,0`	00015435h	16/x	1/x	00004333h	0x00
`MOVE *A0(0FFFh),A1,0`	00014531h	19/x	1/x	FFFC3333h	1x00
`MOVE *A0(7FFFh),A1,1`	0000D531h	x/22	x/1	00043333h	0x00
`MOVE *A0(0FFF2h),A1,1`	00015540h	x/25	x/0	01110CCCh	0x00
`MOVE *A0(8000h),A1,0`	0001D530h	27/x	1/x	FC443333h	1x00
`MOVE *A0(0FFF0h),A1,0`	00015540h	31/x	0/x	44443333h	0x00
`MOVE *A0(0FFE0h),A1,1`	00015558h	x/31	x/1	D5444433h	1x00
`MOVE *A0(0FFECh),A1,0`	0001554Dh	32/x	0/x	AAA22219h	1x00
`MOVE *A0(001Dh),A1,0`	00015520h	32/x	1/x	AAAA2221h	1x00
`MOVE *A0(0020h),A1,1`	00015520h	x/32	x/0	55554444h	0x00

Syntax **MOVE** **Rs(offset), *Rd+ [, F]*

Execution field at (*Rs + offset) → *Rd
Rd + field size → Rd

Instruction Words



Description MOVE moves a field from the one memory location to another. Both the source and destination registers contain bit addresses. The source memory address is formed by adding the contents of the source register to the signed 16-bit offset. The destination register contains the address of the field's destination; after the move, the contents of Rd are incremented by the selected field size.

The field size for the move is 1–32 bits, depending on the selected field size; the field is right justified within the source register. The optional F parameter determines the field size and extension for the move:

F=0 selects FS0
F=1 selects FS1

The SETF instruction sets the field size and extension.

Rs and Rd must be in the same register file.

Machine States The following typical cases assume that the source data and the destination address are aligned on 16-bit boundaries:

16-Bit Field	**32-Bit Field**
5+(1),12	7+(3),16

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples Assume that memory contains the following values before instruction execution:

Address	Data	Address	Data
15500h	0000h	15530h	3333h
15510h	0000h	15540h	4444h
15520h	0000h	15550h	5555h

Code	Before			After			
					@15500h		@15520h
	A0	A1	FS0/1	A1		@15510h	
MOVE *A0(0000h),*A1+,1	00015530h	0015500h	x/1	00015501h	0001h	0000h	0000h
MOVE *A0(0001h),*A1+,1	0001552Fh	00015504h	5/x	00015509h	0130h	0000h	0000h
MOVE *A0(000Fh),*A1+,1	0001552Dh	0001550Ch	8/x	00015514h	3000h	0004h	0000h
MOVE *A0(0020h),*A1+,1	0001551Ch	00015[illegible]Dh	x/13	0001551Ah	6000h	0088h	0000h
MOVE *A0(00FFh),*A1+,1	00015535h	0001550Ch	16/x	0001551Ch	3000h	0433h	0000h
MOVE *A0(0FFFh),*A1+,1	00015531h	00015510h	19/x	00015523h	0000h	3333h	0004h
MOVE *A0(7FFFh),*A1+,1	0000D531h	00015508h	x/22	0001551Eh	3300h	0433h	0000h
MOVE *A0(0FFF2h),*A1+,1	00015540h	0015500h	x/25	00015519h	0CCCh	0111h	0000h
MOVE *A0(8000h),*A1+,1	0001D530h	00015503h	27/x	0001551Eh	9998h	2221h	0000h
MOVE *A0(0FFF0h),*A1+,1	00015540h	00015501h	31/x	0001552Ah	6666h	8888h	0000h
MOVE *A0(0FFE0h),*A1+,1	00015558h	00015508h	x/31	00015527h	3300h	4444h	0055h
MOVE *A0(0FFECh),A1+,1	0001554Dh	0001550Ah	32/x	00015528h	3200h	4444h	0155h
MOVE *A0(001Dh),A1+,1	00015520h	00015510h	32/x	00015530h	0000h	2221h	AAAAh
MOVE *A0(0020h),A1+,1	00015520h	00015510h	x/32	00015530h	0000h	4444h	5555h

Syntax **MOVE** **Rs(SOffset), *Rd(DOffset) [, F]*

Execution field at (*Rs + SOffset) → (*Rd + DOffset)

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	F	Rs				R	Rd			
source offset															
destination offset															

Description This MOVE instruction moves a field from one memory location to another. Both the source and destination registers contain bit addresses. The address of the source address is formed by adding a signed 16-bit offset to the contents of the source register. The address of the destination location is formed by adding a signed 16-bit offset to the contents of the destination register.

The field size for the move is 1–32 bits, depending on the selected field size; the field is right justified within the source register. The optional F parameter determines the field size and extension for the move:

F=0 selects FS0
F=1 selects FS1

The SETF instruction sets the field size and extension.

Rs and Rd must be in the same register file.

Machine States The following typical cases assume that the source data and the destination address are aligned on 16-bit boundaries:

16-Bit Field	**32-Bit Field**
5+(1),15	7+(3),19

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples Assume that memory contains the following values before instruction execution:

Address	Data	Address	Data
15500h	0000h	15530h	3333h
15510h	0000h	15540h	4444h
15520h	0000h	15550h	5555h

Code	Before			After		
				@15500h		@15520h
	A0	A1	FS0/1	@15510h		
MOVE *A0(0000h),*A1(0000h),1	00015530h	0015500h	x/1	0001h	0000h	0000h
MOVE *A0(0001h),*A1(0000h),0	0001552Fh	00015504h	5/x	0130h	0000h	0000h
MOVE *A0(000Fh),*A1(000Fh),0	0001552Dh	000154FDh	8/x	3000h	0004h	0000h
MOVE *A0(0020h),*A1(001Dh),1	0001551Ch	000154F0h	x/13	5000h	0088h	0000h
MOVE *A0(00FFh),*A1(0FFF8h),0	00015435h	00015514h	16/x	3000h	0433h	0000h
MOVE *A0(0FFFh),*A1(0FFFh),0	00014531h	00014511h	19/x	0000h	3333h	0004h
MOVE *A0(7FFFh),*A1(8000h),1	0000D531h	0001D508h	x/22	3300h	0433h	0000h
MOVE *A0(0FFF2h),*A1(7FFFh),1	00015540h	0000D501h	x/25	0CCCh	0111h	0000h
MOVE *A0(8000h),*A1(0020h),0	0001D530h	000154E3h	27/x	9998h	2221h	0000h
MOVE *A0(0FFF0h),*A1(0010h),0	00015540h	000154F1h	31/x	6666h	8888h	0000h
MOVE *A0(0FFE0h),*A1(0FFE0h),1	00015558h	00015528h	x/31	3300h	4444h	0055h
MOVE *A0(0FFECh),*A1(0FFECh),0	0001554Dh	0001551Dh	32/x	3200h	4444h	0155h
MOVE *A0(001Dh),*A1(0020h),0	00015520h	000154F0h	32/x	0000h	2221h	AAAAh
MOVE *A0(0020h),*A1(0020h),1	00015520h	000154F0h	x/32	0000h	4444h	5555h

Syntax **MOVE** *@SAddress, Rd [, F]*

Execution field at SAddress → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	F	1	1	0	1	R	Rd			
16 LSBs of source address															
16 MSBs of source address															

Description This MOVE instruction moves a field from memory to the destination register. The field data for the move is contained at a source-memory bit address. When the field is moved into the destination register, it is right justified and sign extended or zero extended to 32 bits according to the selected value of FE. This instruction also performs an implicit compare to 0 of the field data.

The field size for the move is 1–32 bits, depending on the selected field size; the field is right justified within the source register. The optional F parameter determines the field size and extension for the move:

F=0 selects FS0
F=1 selects FS1

The SETF instruction sets the field size and extension.

Machine States The following typical cases assume that the source data is aligned on a 16-bit boundary:

16-Bit Field	**32-Bit Field**
5,15	7,13

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N *1* if the field-extended data moved to register is negative, *0* otherwise
C Unaffected
Z *1* if the field-extended data moved to register is 0, *0* otherwise
V *0*

Examples Assume that memory contains the following values before instruction execution:

Address	Data
15500h	7770h
15510h	7777h

Code	Before		After	
	FE0/1	FS0/1	A1	NCZV
MOVE @15500h,A1,1	x/0	x/1	00000000h	0x10
MOVE @15500h,A1,0	0/x	5/x	00000010h	0x00
MOVE @15503h,A1,1	x/1	x/5	0000000Eh	0x00
MOVE @15500h,A1,0	0/x	12/x	00000770h	0x00
MOVE @1550Dh,A1,1	x/1	x/12	FFFFFBBBh	1x00
MOVE @15504h,A1,0	1/x	18/x	FFFF7777h	1x00
MOVE @15500h,A1,1	x/0	x/18	00037770h	0x00
MOVE @15500h,A1,0	0/x	27/x	07777770h	0x00
MOVE @15500h,A1,1	x/1	x/27	FF777770h	1x00
MOVE @15501h,A1,0	0/x	30/x	3BBBBBB8h	0x00
MOVE @15501h,A1,1	x/1	x/30	FBBBBBB8h	1x00
MOVE @15500h,A1,0	x/x	32/x	77777770h	0x00

Syntax **MOVE** *@SAddress, *Rd+ [, F]*

Execution field at SAddress → *Rd
Rd + field size → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	F	0	0	0	0	R	Rd			
16 LSBs of source address															
16 MSBs of source address															

Description This MOVE instruction moves a field from one location in memory to another. The source address is a 32-bit address; the destination address is specified by the contents of Rd. After the move, the contents of the destination register are incremented by the field size.

The field size for the move is 1–32 bits, depending on the selected field size; the field is right justified within the source register. The optional F parameter determines the field size and extension for the move:

F=0 selects FS0
F=1 selects FS1

The SETF instruction sets the field size and extension.

Rs and Rd must be in the same register file.

Machine States The following typical cases assume that the source data and the destination address are aligned on 16-bit boundaries:

16-Bit Field	**32-Bit Field**
5+(1),15	7+(3),19

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples Assume that memory contains the following values before instruction execution:

Address	Data	Address	Data
15500h	FFFFh	15530h	0000h
15510h	FFFFh	15540h	0000h
15520h	FFFFh	15550h	0000h

Code	Before			After			
					@15500h	@15510h	@15520h
	A0	A1	FE0/1	A1			
MOVE @15500,*A1+,1	00015530h	00015531h	x/1	00015531h	0001h	0000h	0000h
MOVE @15500,*A1+,0	00015534h	00015539h	5/x	00015539h	01F0h	0000h	0000h
MOVE @15500,*A1+,1	0001553Ah	00015544h	x/10	00015544h	FC00h	000Fh	0000h
MOVE @15500,*A1+,0	0001553Fh	00015552h	19/x	00015552h	8000h	FFFFh	0003h
MOVE @15504,*A1+,1	00015530h	00015537h	x/7	00015537h	007Fh	0000h	0000h
MOVE @1550A,*A1+,0	00015530h	0001553Dh	13/x	0001553Dh	1FFFh	0000h	0000h
MOVE @1550D,*A1+,1	00015534h	00015536h	x/8	00015536h	0FF0h	0000h	0000h
MOVE @1550D,*A1+,0	00015530h	0001554Ch	28/x	0001554Ch	FFFFh	0FFFh	0000h
MOVE @15505,*A1+,1	00015535h	0001554Dh	x/23	0001554Dh	FFE0h	0FFFh	0000h
MOVE @15508,*A1+,0	00015536h	00015555h	31/x	00015555h	FFC0h	FFFFh	001Fh
MOVE @15508,*A1+,1	00015531h	00015548h	x/31	00015548h	FFFEh	FFFFh	0000h
MOVE @1550A,*A1+,0	00015530h	00015550h	32/x	00015550h	FFFFh	FFFFh	0000h
MOVE @15500,*A1+,1	0001553Ah	0001555Ah	x/32	0001555Ah	FC00h	FFFFh	03FFh

Syntax **MOVE** *@SAddress, @DAddress [, F]*

Execution field at SAddress → DAddress

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	F	1	1	1	0	0	0	0	0	0
16 LSBs of source address															
16 MSBs of source address															
16 LSBs of destination address															
16 MSBs of destination address															

Description This MOVE instruction moves a field from one location in memory to another. Both memory addresses are 32-bit addresses.

The field size for the move is 1–32 bits, depending on the selected field size; the field is right justified within the source register. The optional F parameter determines the field size and extension for the move:

F=0 selects FS0
F=1 selects FS1

The SETF instruction sets the field size and extension.

Machine States The following typical cases assume that the source data and the destination address are aligned on 16-bit boundaries:

16-Bit Field	**32-Bit Field**
7+(1),12	9+(3),27

For other cases, see MOVE and MOVB Instructions Timing, Section 13.2.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples Assume that memory contains the following values before instruction execution:

Address	Data
15500h	FFFFh
15510h	FFFFh
15520h	FFFFh
15530h	0000h
15540h	0000h
15550h	0000h

Code	Before	After		
	FS0/1	@15530h	@15540h	@15550h
MOVE @15500h,@15530h,1	x/1	0001h	0000h	0000h
MOVE @15500h,@15534h,0	5/x	01F0h	0000h	0000h
MOVE @15500h,@1553Ah,1	x/10	FC00h	000Fh	0000h
MOVE @15500h,@1553Fh,0	19/x	8000h	FFFFh	0003h
MOVE @15504h,@15530h,1	x/7	007Fh	0000h	0000h
MOVE @1550Ah,@15530h,0	13/x	1FFFh	0000h	0000h
MOVE @1550Dh,@15534h,1	x/8	0FF0h	0000h	0000h
MOVE @1550Dh,@15530h,0	28/x	FFFFh	0FFFh	0000h
MOVE @15505h,@15535h,1	x/23	FFE0h	0FFFh	0000h
MOVE @15508h,@15536h,0	31/x	FFC0h	FFFFh	001Fh
MOVE @15508h,@15531h,1	x/31	FFFEh	FFFFh	0000h
MOVE @1550Ah,@15530h,0	32/x	FFFFh	FFFFh	0000h
MOVE @15500h,@1553Ah,0	x/32	FC00h	FFFFh	03FFh

Syntax **MOVI** */W, Rd [,* **W***]*

Execution IW → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	1	1	0	R	Rd			
16-bit value															

Description MOVI stores a 16-bit, sign-extended immediate value in the destination register. (*IW* in the instruction syntax represents the 16-bit value.)

The assembler uses the short form if the immediate value has been previously defined and is in the range -32,768 through 32,767. You can force the assembler to use the short form by following the register operand with **,W**:

```
MOVI IW,Rd,W
```

The assembler truncates the upper bits and issue an appropriate warning message.

Machine States 2,8

Status Bits
N *1* if the data being moved is negative, *0* otherwise
C Unaffected
Z *1* if the data being moved is 0, *0* otherwise
V *0*

Examples

Code	After A0	NCZV
`MOVI 32767,A0`	00007FFFh	0x00
`MOVI 1,A0`	00000001h	0x00
`MOVI 0,A0`	00000000h	0x10
`MOVI -1,A0`	FFFFFFFFh	1x00
`MOVI -32768,A0`	FFFF8000h	1x00
`MOVI 0000h,A0`	00000000h	0x10
`MOVI 7FFFh,A0`	00007FFFh	0x00

Syntax **MOVI** *IL, Rd [, L]*

Execution IL → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	1	1	1	R	Rd			
16 LSBs of IL															
16 MSBs of IL															

Description MOVI stores a 32-bit immediate value in the destination register. (*IL* in the instruction syntax represents the 32-bit value.)

The assembler uses this opcode if it cannot use the `MOVI IW,Rd` opcode, or if the long opcode is forced by following the register operand with **,L**:

```
MOVI IL,Rd,L
```

Machine States 3,12

Status Bits
N *1* if the data being moved is negative, *0* otherwise
C Unaffected
Z *1* if the data being moved is 0, *0* otherwise
V *0*

Examples

Code	After A0	After NCZV
`MOVI 2147483647,A0`	7FFFFFFFh	0x00
`MOVI 32768,A0`	00008000h	0x00
`MOVI -32769,A0`	FFFF7FFFh	1x00
`MOVI -2147483648,A0`	80000000h	1x00
`MOVI 8000h,A0`	00008000h	0x00
`MOVI 0FFFFFFFFh,A0`	FFFFFFFFh	1x00
`MOVI 0FFFFh,A0,L`	FFFFFFFFh	1x00

Syntax **MOVK** *K, Rd*

Execution K → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	K					R	Rd			

Description MOVK stores a 5-bit constant in the destination register. (*K* in the instruction syntax represents the constant.) The constant is treated as an unsigned number in the range 1–32, where K = 0 in the opcode corresponds to a value of 32. The resulting constant value is zero extended to 32 bits.

Note that you cannot set a register to 0 with this instruction. You can clear a register by XORing the register with itself; use `CLR Rd` (an alternate mnemonic for XOR) to accomplish this. Both these methods alter the Z bit (set it to 1).

Machine States 1,4

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	After
	A0
`MOVK 1,A0`	00000001h
`MOVK 8,A0`	00000008h
`MOVK 16,A0`	00000010h
`MOVK 32,A0`	00000020h

Syntax **MOVX** *Rs, Rd*

Execution Rs**X** → Rd**X**

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	Rs				R	Rd			

Description MOVX moves the X half of the source register (16 LSBs) to the X half of the destination register. The Y halves of both registers are unaffected.

You an also use the MOVX and MOVY instructions for handling packed 16-bit quantities and XY addresses. You can use the RL instruction to swap the contents of X and Y.

Rs and Rd must be in the same register file.

Machine States 1,4

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before		After
	A0	**A1**	**A1**
`MOVX A0,A1`	00000000h	FFFFFFFFh	FFFF0000h
`MOVX A0,A1`	12345678h	00000000h	00005678h
`MOVX A0,A1`	FFFFFFFFh	00000000h	0000FFFFh

Syntax **MOVY** *Rs, Rd*

Execution Rs**Y** → Rd**Y**

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	Rs				R	Rd			

Description MOVY moves the Y half of the source register (16 MSBs) to the Y half of the destination register. The X halves of both registers are unaffected.

You an also use the MOVX and MOVY instructions for handling packed 16-bit quantities and XY addresses. You can use the RL instruction to swap the contents of X and Y.

Rs and Rd must be in the same register file.

Machine States 1,4

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before		After
	A0	**A1**	**A1**
`MOVY A0,A1`	00000000h	FFFFFFFFh	0000FFFFh
`MOVY A0,A1`	12345678h	00000000h	12340000h
`MOVY A0,A1`	FFFFFFFFh	00000000h	FFFF0000h

Syntax **MPYS** *Rs, Rd*

Execution
Rd Even: Rs × Rd → Rd:Rd+1
Rd Odd: Rs × Rd → Rd

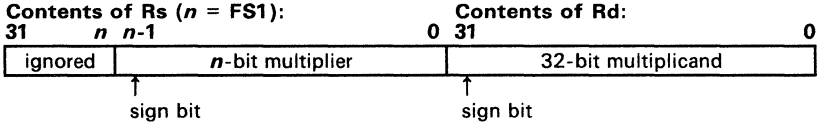
Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	0	Rs				R	Rd			

Description

MPYS performs a signed multiply of a variably-sized field in the source register by the 32 bits in the destination register. This produces a 32-bit to a 64-bit result, depending on the register and field definitions. Note that Rs and Rd must be in the same register file.

The value of field size 1 (FS1) defines the size of the multiplier in Rs. FS1 may have any **even** value *n* from 2 to 32 (that is, *n* = 2, 4, 6 ... 30, 32). The instruction executes a 32-bit-by-*n*-bit multiply – multiplying the 32 bits in Rd by the *n* bits in Rs. All values are signed. The MSB of the source field (bit *n* - 1 in Rs) defines the sign of the field; the bits to the left of bit *n* are ignored. The MSB of Rd defines the sign of the multiplicand.

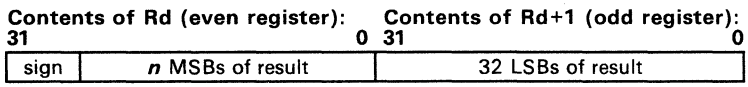


MPYS has two modes, depending on whether Rd is even or odd:

- **Rd Even:**

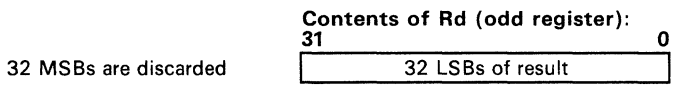
  MPYS multiplies the contents of Rd by the *n*-bit field in Rs, and stores the result in two consecutive registers, Rd and Rd+1. (For example, if Rd=B4, the result is stored in registers B4 and B5.) The result is sign extended and right justified; the 32 MSBs are stored in Rd and the 32 LSBs are stored in Rd+1. Note that all 32 bits of both registers are used, regardless of the field size of the multiply.

  Do not use A14 or B14 as the destination register, because Rd+1 (A15 or B15) is the stack pointer register (SP). It is not desirable to write over the contents of the SP.



- **Rd Odd:**

  MPYS multiplies the contents of Rd by the *n*-bit field in Rs, and stores the 32 LSBs of the result in Rd; Rs is not changed. If the result is greater than 32 bits, the extra MSBs are discarded, regardless of the field size.



Machine States 5 + FS1/2, 8 + FS1/2

Status Bits
N *1* if the result is negative, *0* otherwise
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Example 1

```
MPYS A1, A0
```

Before

A0	A1	FS1
00000000h	00000000h	32
7FFFFFFFh	7FFFFFFFh	32
7FFFFFFFh	FFFFFFFFh	32
FFFFFFFFh	7FFFFFFFh	32
FFFFFFFFh	FFFFFFFFh	32
80000000h	7FFFFFFFh	32
80000000h	80000000h	32
80000001h	80000000h	32
8040156Fh	7FF3B074h	32
8040156Fh	7FF3B074h	24
8040156Fh	7FF3B074h	20
8040156Fh	7FF3B074h	16
8040156Fh	7FF3B074h	14
8040156Fh	7FF3B074h	8
8040156Fh	7FF3B074h	6
8040156Fh	7FF3B074h	4
8040156Fh	7FF3B074h	2

After

A0	A1	NCZV
00000000h	00000000h	0x1x
3FFFFFFFh	00000001h	0x0x
FFFFFFFFh	80000001h	1x0x
FFFFFFFFh	80000001h	1x0x
00000000h	00000001h	0x0x
C0000000h	80000000h	1x0x
40000000h	00000000h	0x0x
3FFFFFFFh	80000000h	0x0x
C0262CDCh	53E486F8h	1x0x
000624B1h	53E486F8h	0x0x
FFFE28B2h	594486F8h	1x0x
000027B2h	17EC86F8h	0x0x
000007C2h	1C0206F8h	0x0x
FFFFFFC6h	1D0766F8h	1x0x
00000005h	FCFF3BF8h	0x0x
FFFFFFFEh	01004158h	1x0x
00000000h	00000000h	0x1x

Example 2

```
MPYS A0,A1
```

Before

A0	A1	FS1
00000000h	00000000h	32
7FFFFFFFh	7FFFFFFFh	32
7FFFFFFFh	7FFFFFFFh	32
FFFFFFFFh	7FFFFFFFh	32
FFFFFFFFh	FFFFFFFFh	32
80000000h	7FFFFFFFh	32
80000000h	80000000h	32
80000001h	80000000h	32
7FF3B074h	80401056h	32
7FF3B074h	80401056h	24
7FF3B074h	80401056h	20
7FF3B074h	80401056h	16
7FF3B074h	80401056h	14
7FF3B074h	80401056h	8
7FF3B074h	80401056h	6
7FF3B074h	80401056h	4
7FF3B074h	80401056h	2

After

A0	A1	NCZV
unchanged	00000000h	0x1x
unchanged	00000001h	0x0x
unchanged	80000001h	1x0x
unchanged	80000001h	1x0x
unchanged	00000001h	0x0x
unchanged	80000000h	1x0x
unchanged	00000000h	0x0x
unchanged	80000000h	0x0x
unchanged	53E486F8h	1x0x
unchanged	53E486F8h	0x0x
unchanged	594486F8h	1x0x
unchanged	17EC86F8h	0x0x
unchanged	1C0206F8h	0x0x
unchanged	1D0766F8h	1x0x
unchanged	FCFF3BF8h	0x0x
unchanged	01004158h	1x0x
unchanged	00000000h	0x1x

Syntax **MPYU** *Rs, Rd*

Execution
Rd Even: Rs × Rd → Rd:Rd+1
Rd Odd: Rs × Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	Rs				R	Rd			

Description MPYU performs an unsigned multiply of a variably-sized field in the source register by the 32 bits in the destination register. This produces a 32-bit to a 64-bit result, depending on the register and field definitions. Note that Rs and Rd must be in the same register file.

The value of field size 1 (FS1) defines the size of the multiplier in Rs. FS1 may have any **even** value *n* from 2 to 32 (that is, *n* = 2, 4, 6 ... 30, 32). The instruction executes a 32-bit-by-*n*-bit multiply – multiplying the 32 bits in Rd by the *n* bits in Rs. All values are unsigned.

Contents of Rs (*n* = FS1): 31	*n*	*n*-1	0	**Contents of Rd:** 31	0
ignored		***n***-bit multiplier		32-bit multiplicand	

MPYS has two modes, depending on whether Rd is even or odd:

- **Rd Even:**

  MPYU multiplies the contents of Rd by the *n*-bit field in Rs, and stores the result in two consecutive registers, Rd and Rd+1. (For example, if Rd=B4, the result is stored in registers B4 and B5.) The result is zero extended and right justified; the 32 MSBs are stored in Rd and the 32 LSBs are stored in Rd+1. Note that all 32 bits of both registers are used, regardless of the field size of the multiply.

  Do not use A14 or B14 as the destination register, because Rd+1 (A15 or B15) is the stack pointer register (SP). It is not desirable to write over the contents of the SP.

Contents of Rd (even register): 31	0	**Contents of Rd+1 (odd register):** 31	0
0s	***n*** MSBs of result	32 LSBs of result	

- **Rd Odd:**

  MPYU multiplies the contents of Rd by the *n*-bit field in Rs, and stores the 32 LSBs of the result in Rd; Rs is not changed. If the result is greater than 32 bits, the extra MSBs are discarded, regardless of the field size.

	Contents of Rd (odd register): 31	0
32 MSBs are discarded	32 LSBs of result	

Machine States

Rs nonnegative: 5 + FS1/2, 8 + FS1/2
Rs negative: 6 + FS1/2, 9 + FS1/2

Status Bits

N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Example 1

```
MPYU A1,A0
```

Before

A0	A1	FS1
FFFF0000h	10000000h	32
FFFF0000h	10001010h	32
FFFF0000h	10001010h	16
FFFF0000h	10001010h	8
FFFF0000h	10001010h	4
08001056h	0003B074h	32
08001056h	0003B074h	16
08001056h	0003B074h	14
08001056h	0003B074h	8
08001056h	0003B074h	6
08001056h	0003B074h	4
08001056h	0003B074h	2

After

A0	A1	NCZV
0FFFF000h	00000000h	xx0x
1000000Fh	EFF00000h	xx0x
0000100Fh	EFF00000h	xx0x
0000000Fh	FFF00000h	xx0x
00000000h	00000000h	xx1x
00001D83h	DC4486F8h	xx0x
00000583h	AB4286F8h	xx0x
00000183h	A31786F8h	xx0x
00000003h	A00766F8h	xx0x
00000001h	A0035178h	xx0x
00000000h	20004158h	xx0x
00000000h	00000000h	xx1x

Example 2

```
MPYU A0,A1
```

Before

A0	A1	FS1
10000000h	FFFF0000h	32
10001010h	FFFF0000h	32
10001010h	FFFF0000h	16
10001010h	FFFF0000h	8
10001010h	FFFF0000h	4
0003B074h	08001056h	32
0003B074h	08001056h	16
0003B074h	08001056h	14
0003B074h	08001056h	8
0003B074h	08001056h	6
0003B074h	08001056h	4
0003B074h	08001056h	2

After

A0	A1	NCZV
unchanged	00000000h	xx0x
unchanged	EFF00000h	xx0x
unchanged	EFF00000h	xx0x
unchanged	FFF00000h	xx0x
unchanged	00000000h	xx1x
unchanged	DC4486F8h	xx0x
unchanged	AB4286F8h	xx0x
unchanged	A31786F8h	xx0x
unchanged	A00766F8h	xx0x
unchanged	A0035178h	xx0x
unchanged	20004158h	xx0x
unchanged	00000000h	xx1x

Syntax **NEG** *Rd*

Execution 2s complement of Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	1	0	1	R	Rd			

Description NEG stores the 2s complement of the contents of the destination register back into the destination register.

Machine States 1,4

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a borrow (Rd ≠ 0), *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow (Rd = 80000000h), *0* otherwise

Examples

Code	Before	After	
	A0	NCZV	A0
NEG A0	00000000h	0010	00000000h
NEG A0	55555555h	1100	AAAAAAABh
NEG A0	7FFFFFFFh	1100	80000001h
NEG A0	80000000h	1101	80000000h
NEG A0	80000001h	0100	7FFFFFFFh
NEG A0	FFFFFFFFh	0100	00000001h

Syntax **NEGB** *Rd*

Execution (2s complement of Rd) - C → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	1	1	0	R	Rd			

Description NEGB takes the 2s complement of the destination register's contents and decrements by 1 if the borrow bit (C) is set; the result is stored in the destination register. This instruction can be used in sequence with itself and with the NEG instruction for negating multiple-register quantities.

Machine States 1,4

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a borrow, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before		After	
	A0	C	NCZV	A0
NEGB A0	00000000h	0	0010	00000000h
NEGB A0	00000000h	1	1100	FFFFFFFFh
NEGB A0	55555555h	0	1100	AAAAAAABh
NEGB A0	55555555h	1	1100	AAAAAAAAh
NEGB A0	7FFFFFFFh	0	1100	80000001h
NEGB A0	7FFFFFFFh	1	1100	80000000h
NEGB A0	80000000h	0	1101	80000000h
NEGB A0	80000000h	1	0100	7FFFFFFFh
NEGB A0	80000001h	0	0100	7FFFFFFFh
NEGB A0	80000001h	1	0100	7FFFFFFEh
NEGB A0	FFFFFFFFh	0	0100	00000001h
NEGB A0	FFFFFFFFh	1	0110	00000000h

Syntax **NOP**

Execution No operation

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0

Description The program counter is incremented to point to the next instruction. The processor status is otherwise unaffected.

You an use the NOP instruction to pad loops and perform other timing functions.

Machine States 1,4

Status Bits
N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Example

Code	Before	After
	PC	**PC**
NOP	00020000h	00020010h

Syntax **NOT** *Rd*

Execution NOT Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	1	1	1	R	Rd			

Description NOT stores the 1s complement of the destination register's contents back into the destination register.

Machine States 1,4

Status Bits

N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before	After	
	A0	**NCZV**	**A0**
`NOT A0`	00000000h	xx0x	FFFFFFFFh
`NOT A0`	55555555h	xx0x	AAAAAAAAh
`NOT A0`	FFFFFFFFh	xx1x	00000000h
`NOT A0`	80000000h	xx0x	7FFFFFFFh

Syntax **OR** *Rs, Rd*

Execution Rs OR Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	Rs				R	Rd			

Description This instruction bitwise-ORs the contents of the source register with the contents of the destination register; the result is stored in the destination register.

Rs and Rd must be in the same register file.

Machine States 1,4

Status Bits
N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before		After	
	A0	**A1**	**A1**	**NCZV**
`OR A0,A1`	FFFFFFFFh	00000000h	FFFFFFFFh	xx0x
`OR A0,A1`	00000000h	FFFFFFFFh	FFFFFFFFh	xx0x
`OR A0,A1`	55555555h	AAAAAAAAh	FFFFFFFFh	xx0x
`OR A0,A1`	00000000h	00000000h	00000000h	xx1x

Syntax **ORI** *IL, Rd*

Execution IL OR Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	1	0	1	R	Rd			
16 LSBs of IL															
16 MSBs of IL															

Description This instruction bitwise-ORs a 32-bit immediate value with the contents of the destination register and stores the result in the destination register. (*IL* in the syntax represents the 32-bit value.)

Machine States 3,12

Status Bits

N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before	After	
	A0	**A0**	**NCZV**
`ORI 0FFFFFFFFh,A0`	00000000h	FFFFFFFFh	xx0x
`ORI 00000000h,A0`	FFFFFFFFh	FFFFFFFFh	xx0x
`ORI 0AAAAAAAAh,A0`	55555555h	FFFFFFFFh	xx0x
`ORI 00000000h,A0`	00000000h	00000000h	xx1x

Syntax **PIXBLT B, L**

Execution binary pixel array → linear pixel array (with processing)

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0

Description This PIXBLT instruction expands, transfers, and processes a binary source pixel array with a destination pixel array.

This instruction operates on two-dimensional arrays of pixels using linear starting addresses for both the source and the destination. The source pixel array is treated as a one bit per pixel array. As the PixBlt proceeds, the source pixels are expanded and then combined with the corresponding destination pixels based on the selected graphics operations.

Note that the parameters are entered exactly as shown in the syntax; that is, the instruction is entered as `PIXBLT B,L`. The first parameter, **B**, indicates that the starting address of the source array is a linear address but the source array is a binary array. The second parameter, **L**, indicates that the starting address of the destination array is a linear address. The following set of implied operands govern the operation of the instruction and define the source and destination arrays.

Implied Operands

B File Registers			
Register	**Name**	**Format**	**Description**
B0†	SADDR	Linear	Source pixel array starting address
B1	SPTCH	Linear	Source pixel array pitch
B2†	DADDR	Linear	Destination pixel array starting address
B3	DPTCH	Linear	Destination pixel array pitch
B7	DYDX	XY	Pixel array dimensions (rows:columns)
B8	COLOR0	Pixel	Background expansion color
B9	COLOR1	Pixel	Foreground expansion color
B10–B14†			Reserved registers
I/O Registers			
Address	**Name**	**Description and Elements (Bits)**	
C000000B0h	CONTROL	PP–Pixel processing operations (22 options) T –Transparency operation	
C0000150h	PSIZE	Pixel size (1,2,4,8,16)	
C0000160h	PMASK	Plane mask – pixel format	

† These registers are changed by PIXBLT execution.

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the PIXBLT instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an instruction that reads the same location (such as a MOVE SAddress,Rd instruction). For more information, refer to Section 6.2, Latency of Writes to I/O Registers.

Source Array The source pixel array for the expand operation is defined by the contents of the SADDR, SPTCH, and DYDX registers:

- At the outset of the instruction, SADDR contains the **linear** address of the pixel with the lowest address in the array. During instruction execution, SADDR points to the address of the next set of 32 pixels to be read from the source array. When the transfer is complete, SADDR points to the linear address of the first pixel on the **next** row of pixels that would have been moved if the block transfer continued.

- SPTCH contains the linear difference in the starting addresses of adjacent rows of the source array. For this PIXBLT instruction, SPTCH can be any value.

- DYDX specifies the dimensions, in pixels, of both the source and destination arrays. The DY portion of DYDX contains the number of rows in the array; the DX portion contains the number of pixels per row.

Source Expansion The actual values of the source pixels are determined by the interaction of the source array with the contents of the COLOR1 and COLOR0 registers. In the expansion operation, a **1** bit in the source array selects a pixel from the COLOR1 register for operation on the destination array. A **0** bit in the source array selects a COLOR0 pixel for this purpose. The pixels selected from the COLOR1 and COLOR0 registers are those that align directly with their intended position in the destination array word.

Destination Array The location of the destination pixel block is defined by the contents of the DADDR, DPTCH, and DYDX registers:

- At the outset of the instruction, DADDR contains the **linear** address of the pixel with the lowest address in the array. During instruction execution, DADDR points to the next pixel (or word of pixels) to be modified in the destination array. When the block transfer is complete, DADDR points to the linear address of the first pixel on the **next** row of pixels that would have been moved if the block transfer continued.

- DPTCH contains the linear difference in the starting addresses of adjacent rows of the destination array (typically this is the screen pitch). DPTCH **must** be a multiple of 16.

- DYDX specifies the dimensions, in pixels, of both the source and destination arrays. The DY portion of DYDX contains the number of rows in the array; the DX portion contains the number of pixels per row.

Corner Adjust No corner adjust is performed for this instruction
PBH and PBV are' ignored. The pixel transfer simply proceeds in the order of increasing linear addresses.

Window Checking You **cannot** use window checking with this PixBlt instruction. The contents of the WSTART and WEND registers are ignored.

Pixel Processing You can select a pixel processing option for this instruction by setting the PPOP bits in the CONTROL register. The pixel processing operation is applied to *expanded pixels* as they are processed with the destination array; that is, the data is *first expanded* and *then processed*. There are 16 Boolean and 6 arithmetic operations; the default case at reset is the *replace* (S → D) operation. The 6 arithmetic operations do not operate with pixel sizes of one or two bits per pixel. For more information, see Section 7.7, Pixel Processing, on page 7-15.

Transparency You can enable transparency for this instruction by setting the T bit in the CONTROL I/O register to 1. The TMS34010 checks for 0 (transparent) pixels *after* it expands and processes the source data. At reset, the default case for transparency is *off*.

Plane Mask The plane mask is enabled for this instruction.

Interrupts This instruction can be interrupted at a word or row boundary of the destination array. When the PixBlt is interrupted, the TMS34010 sets the PBX bit in the status register and then pushes the status register on the stack. At this time, DPTCH, SPTCH, and B10–B14 contain intermediate values. DADDR points to the linear address of the next word of pixels to be modified after the interrupt is processed. SADDR points to the address of the next 32 pixels to be read from the source array after the interrupt is processed.

The PIXBLT instruction uses several I/O and B-file registers as implied operands. If an interrupt service routine modifies a register that the PIXBLT uses as an implied operand, you must restore that register to the value it had when the routine began, before returning from the routine. (You can use the MMFM and MMTM instructions to save and restore the B-file registers.) In order to maintain compatibility with future TMS340 devices, use only the RETI instruction to return from an interrupt routine.

Shift Register Transfers If the SRT bit in the DPYCTL I/O register is set, each memory read or write initiated by the PixBlt generates a shift register transfer read or write cycle at the selected address. This operation can be used for bulk memory clears or transfers. (Not all VRAMs support this capability.)

Machine States See PIXBLT Expand Instructions Timing, Section 13.5.

Status Bits

N Undefined
C Undefined
Z Undefined
V Undefined

Examples Before executing the PIXBLT instruction, load the implied operand registers with appropriate values. These PIXBLT examples use the following implied operand setup:

Register File B:		**I/O Registers:**	
SADDR (B0)	= 00002030h	PSIZE	= 0010h
SPTCH (B1)	= 00000100h		
DADDR (B2)	= 00033000h		

DPTCH (B3) = 00001000h
DYDX (B7) = 00020010h
COLOR0 (B8) = FEDCFEDCh
COLOR1 (B9) = BA98BA98h

Additional implied operand values are listed with each example.

For this example, assume that memory contains the following data before instruction execution.

Linear Address	Data
02000h	xxxxh, xxxxh, xxxxh, 1234h, xxxxh, xxxxh, xxxxh, xxxxh
02080h	xxxxh, xxxxh, xxxxh, xxxxh, xxxxh, xxxxh, xxxxh, xxxxh
02100h	xxxxh, xxxxh, xxxxh, 5678h, xxxxh, xxxxh, xxxxh, xxxxh
02180h	xxxxh, xxxxh, xxxxh, xxxxh, xxxxh, xxxxh, xxxxh, xxxxh
33000h	FFFFh, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh
33080h	FFFFh, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh
34000h	FFFFh, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh
34080h	FFFFh, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh

Example 1 This example uses the *replace (S → D)* pixel processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 0000h (T=0, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	Data
33000h	FEDCh, FEDCh, BA98h, FEDCh, BA98h, BA98h, FEDCh, FEDCh
33080h	FEDCh, BA98h, FEDCh, FEDCh, BA98h, FEDCh, FEDCh, FEDCh
34000h	FEDCh, FEDCh, FEDCh, BA98h, BA98h, BA98h, BA98h, FEDCh
34080h	FEDCh, BA98h, BA98h, FEDCh, BA98h, FEDCh, BA98h, FEDCh

Example 2 This example uses the *(D - S) → D* pixel processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 4800h (T=0, PP=10010).

After instruction execution, memory contains the following values:

Linear Address	Data
33000h	0123h, 0123h, 4567h, 0123h, 4567h, 4567h, 0123h, 0123h
33080h	0123h, 4567h, 0123h, 0123h, 4567h, 0123h, 0123h, 0123h
34000h	0123h, 0123h, 0123h, 4567h, 4567h, 4567h, 4567h, 0123h
34080h	0123h, 4567h, 4567h, 0123h, 4567h, 0123h, 4567h, 0123h

Example 3 This example uses transparency with COLOR0 = 00000000h. Before instruction execution, PMASK = 0000h and CONTROL = 0020h (T=1, W=00, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	Data
33000h	FFFFh, FFFFh, BA98h, FFFFh, BA98h, BA98h, FFFFh, FFFFh
33080h	FFFFh, BA98h, FFFFh, FFFFh, BA98h, FFFFh, FFFFh, FFFFh
34000h	FFFFh, FFFFh, FFFFh, BA98h, BA98h, BA98h, BA98h, FFFFh
34080h	FFFFh, BA98h, BA98h, FFFFh, BA98h, FFFFh, BA98h, FFFFh

Example 4 This example uses plane masking
the four LSBs are masked. Before instruction execution, PMASK = 000Fh and CONTROL = 0000h (T=0, W=00, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	Data
33000h	FEDFh, FEDFh, BA9Fh, FEDFh, BA9Fh, BA9Fh, FEDFh, FEDFh
33080h	FEDFh, BA9Fh, FEDFh, FEDFh, BA9Fh, FEDFh, FEDFh, FEDFh
34000h	FEDFh, FEDFh, FEDFh, BA9Fh, BA9Fh, BA9Fh, BA9Fh, FEDFh
34080h	FEDFh, BA9Fh, BA9Fh, FEDFh, BA9Fh, FEDFh, BA9Fh, FEDFh

Syntax **PIXBLT B, XY**

Execution binary pixel array → XY pixel array (with processing)

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	1	0	1	0	0	0	0	0

Description This PIXBLT instruction expands, transfers, and processes a binary source pixel array with a destination pixel array.

This instruction operates on two-dimensional arrays of pixels using a linear starting address for the source and an XY address for the destination. The source pixel array is treated as a one bit per pixel array. As the PixBlt proceeds, the source pixels are expanded and then combined with the corresponding destination pixels based on the selected graphics operations.

Note that the parameters are entered exactly as shown in the syntax; that is, the instruction is entered as `PIXBLT B,XY`. The first parameter, **B**, indicates that the starting address of the source array is a linear address but the source array is a binary array. The second parameter, **XY**, indicates that the starting address of the destination array is an XY address.

The following set of implied operands govern the operation of the instruction and define the source and destination arrays.

Implied Operands

B File Registers			
Register	**Name**	**Format**	**Description**
B0†	SADDR	Linear	Source pixel array starting address
B1	SPTCH	Linear	Source pixel array pitch
B2†‡	DADDR	XY	Destination pixel array starting address
B3	DPTCH	Linear	Destination pixel array pitch
B4	OFFSET	Linear	Screen origin (0,0)
B5	WSTART	XY	Window starting corner
B6	WEND	XY	Window ending corner
B7‡	DYDX	XY	Pixel array dimensions (rows:columns)
B8	COLOR0	Pixel	Background expansion color
B9	COLOR1	Pixel	Foreground expansion color
B10–B14†			Reserved registers
I/O Registers			
Address	**Name**	**Description and Elements (Bits)**	
C000000B0h	CONTROL	PP – Pixel processing operations (22 options) W – Window clipping or pick operation T – Transparency operation	
C0000130h	CONVSP	XY-to-linear conversion (source pitch) Used for source preclipping.	
C0000140h	CONVDP	XY-to-linear conversion (destination pitch)	
C0000150h	PSIZE	Pixel size (1,2,4,6,8,16)	
C0000160h	PMASK	Plane mask – pixel format	

† These registers are changed by PIXBLT execution.
‡ Used for common rectangle function with window hit operation (W=1).

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the PIXBLT instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an instruction that reads the same location (such as a MOVE SAddress,Rd instruction). For more information, refer to Section 6.2, Latency of Writes to I/O Registers.

Source Array The source pixel array for the expand operation is defined by the contents of the SADDR, SPTCH, DYDX, and (possibly) CONVSP registers:

- At the outset of the instruction, SADDR contains the **linear** address of the pixel with the lowest address in the array. During instruction execution, SADDR points to the address of the next set of 32 pixels to be read from the source array. When the block transfer is complete, SADDR points to the linear address of the first pixel on the **next** row of pixels that would have been moved if the block transfer continued.

- SPTCH contains the linear difference in the starting addresses of adjacent rows of the source array. SPTCH can be any value for this PIXBLT. For window clipping, SPTCH must be a power of two, and CONVSP must be set to correspond to the SPTCH value.

- CONVSP is calculated by taking the LMO of SPTCH; this value is used for the XY calculations involved in XY addressing and window clipping.

- DYDX specifies the dimensions, in pixels, of both the source and destination arrays in pixels. The DY portion of DYDX contains the number of rows in the array; the the DX portion contains the number of pixels per row.

Source Expansion The actual values of the source pixels are determined by the interaction of the source array with contents of the COLOR1 and COLOR0 registers. In the expansion operation, a **1** bit in the source array selects a pixel from the COLOR1 register for operation on the destination array. A **0** bit in the source array selects a COLOR0 pixel for this purpose. The pixels selected from the COLOR1 and COLOR0 registers are those that align directly with their intended position in the destination array word.

Destination Array The location of the destination pixel block is defined by the contents of the DADDR, DPTCH, CONVDP, OFFSET, and DYDX registers:

- At the outset of the instruction, DADDR contains the **XY** address of the pixel with the lowest address in the array; it is used with OFFSET and CONVDP to calculate the linear address of the array. During instruction execution, DADDR points to the **linear address** of next pixel (or word of pixels) to be modified in the destination array. When the block transfer is complete, DADDR points to the **linear address** of the first pixel on the **next** row of pixels that would have been moved if the block transfer continued.

- DPTCH contains the linear difference in the starting addresses of adjacent rows of the destination array (typically this is the screen pitch). DPTCH **must** be a power of two (greater than or equal to 16) and CONVDP must be set to correspond to the DPTCH value.

- CONVDP is determined by taking the LMO of the DPTCH register; this value is used for the XY calculations involved in XY addressing and window clipping.
- DYDX specifies the dimensions, in pixels, of both the source and destination arrays. The DY portion of DYDX contains the number of rows in the array; the DX portion contains the number of pixels per row.

Corner Adjust No corner adjust is performed for this instruction. The transfer executes in the order of increasing linear addresses. PBH and PBV are ignored.

Window Checking You can use window checking with this instruction by setting the W bits in the CONTROL register to the desired value. If you select window checking mode 1, 2, or 3, the WSTART and WEND registers define the XY starting and ending corners of a rectangular window.

0 *No windowing.* The entire pixel array is drawn and the WVP and V bits are unaffected.

1 *Window hit.* No pixels are drawn. The V bit is set to 0 if any portion of the destination array lies within the window. Otherwise, the V bit is set to 1.

If the V bit is set to 0, the DADDR and DYDX registers are modified to correspond to the common rectangle formed by the intersection of the destination array with the rectangular window. DADDR is set to the XY address of the pixel in the starting corner of the common rectangle. DYDX is set to the X and Y dimensions of the common rectangle.

If the V bit is set to 1, the array lies entirely outside the window, and the values of DADDR and DYDX are indeterminate.

2 *Window miss.* If the array lies **entirely** within the active window, it is drawn and the V bit is set to 0. Otherwise, no pixels are drawn, the V and WVP bits are set to 1, and the instruction is aborted.

3 *Window clip.* The source and destination arrays are preclipped to the window dimensions. Only those pixels that lie within the common rectangle (corresponding to the intersection of the specified array and the window) are drawn. If any preclipping is required, the V bit is set to 1.

Pixel Processing You can select a pixel processing option for this instruction by setting the PPOP bits in the CONTROL register. The pixel processing operation is applied to *expanded pixels* as they are processed with the destination array; that is, the data is *first expanded* and *then processed.* There are 16 Boolean and 6 arithmetic operations; the default case at reset is the S → D operation. The 6 arithmetic operations do not operate with pixel sizes of one or two bits per pixel. For more information, see Section 7.7, Pixel Processing, on page 7-15.

Transparency You can enable transparency for this instruction by setting the T bit in the CONTROL I/O register to 1. The TMS34010 checks for 0 (transparent) pixels *after* it expands and processes the source data. At reset, the default case for transparency is *off.*

Plane Mask The plane mask is enabled for this instruction.

Interrupts This instruction can be interrupted at a word or row boundary of the destination array. When the PixBlt is interrupted, the TMS34010 sets the PBX bit in the status register and then pushes the status register on the stack. At this time, DPTCH, SPTCH, and B10–B14 contain intermediate values. DADDR points to the linear address of the next word of pixels to be modified after the interrupt is processed. SADDR points to the address of the next 32 pixels to be read from the source array after the interrupt is processed.

The PIXBLT instruction uses several I/O and B-file registers as implied operands. If an interrupt service routine modifies a register that the PIXBLT uses as an implied operand, you must restore that register to the value it had when the routine began, before returning from the routine. (You can use the MMFM and MMTM instructions to save and restore the B-file registers.) In order to maintain compatibility with future TMS340 devices, use only the RETI instruction to return from an interrupt routine.

Shift Register Transfers If the SRT bit in the DPYCTL I/O register is set, each memory read or write initiated by the PixBlt generates a shift register transfer read or write cycle at the selected address. This operation can be used for bulk memory clears or transfers. (Not all VRAMs support this capability.)

Machine States See PIXBLT Expand Instructions Timing, Section 13.5.

Status Bits

N Undefined
C Undefined
Z Undefined
V *1* if a window violation occurs, *0* otherwise; undefined if window checking is not enabled (W=00)

Examples Before executing a PIXBLT instruction, load the implied operand registers with appropriate values. These PIXBLT examples use the following implied operand setup.

Register File B:

SADDR (B0)	= 00002010h
SPTCH (B1)	= 00000010h
DADDR (B2)	= 00300022h
DPTCH (B3)	= 00001000h
OFFSET (B4)	= 00010000h
WSTART (B5)	= 00000026h
WEND (B6)	= 00400050h
DYDX (B7)	= 00040010h
COLOR0 (B8)	= 00000000h
COLOR1 (B9)	= 7C7C7C7Ch

I/O Registers:

PSIZE	= 0008h
CONVSP	= 001Bh
CONVDP	= 0013h

Additional implied operand values are listed with each example.

For this example, assume that memory contains the following data before instruction execution.

Linear Address	Data
2000h	xxxxh, 0123h, 4567h, 89ABh, CDEFh, xxxxh, xxxxh, xxxxh
40000h to 43200h	FFFFh

Example 1 This example uses the *replace (S → D)* pixel processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 0000h (T=0, W=00, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	Data
40100h	FFFFh, 7C7Ch, 0000h, 7C00h, 0000h, 007Ch, 0000h, 0000h
40180h	0000h, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh
41100h	FFFFh, 7C7Ch, 007Ch, 7C00h, 007Ch, 007Ch, 007Ch, 0000h
41180h	007Ch, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh
42100h	FFFFh, 7C7Ch, 7C00h, 7C00h, 7C00h, 007Ch, 7C00h, 0000h
42180h	7C00h, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh
43100h	FFFFh, 7C7Ch, 7C7Ch, 7C00h, 7C7Ch, 007Ch, 7C7Ch, 0000h
43180h	7C7Ch, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh, FFFFh

XY Addressing

Y Address	X Address																			
	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	30	31	32	34
30	FF	FF	7C	7C	00	00	00	7C	00	00	7C	00	00	00	00	00	00	00	FF	FF	FF
31	FF	FF	7C	7C	7C	00	00	7C	7C	00	7C	00	7C	00	00	00	7C	00	FF	FF	FF
32	FF	FF	7C	7C	00	7C	00	7C	00	7C	7C	00	00	7C	00	00	00	7C	FF	FF	FF
33	FF	FF	7C	7C	7C	7C	00	7C	7C	7C	7C	00	7C	7C	00	00	7C	7C	FF	FF	FF

Example 2 This example uses the XOR pixel processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 2800h (T=0, W=00, PP=01010).

After instruction execution, memory contains the following values:

Y Address	X Address																			
	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	30	31	32	34
30	FF	FF	83	83	FF	FF	FF	83	FF	FF	83	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
31	FF	FF	83	83	83	FF	FF	83	83	FF	83	FF	83	FF	FF	FF	83	FF	FF	FF	FF
32	FF	FF	83	83	FF	83	FF	83	FF	83	83	FF	FF	83	FF	FF	FF	83	FF	FF	FF
33	FF	FF	83	83	83	83	FF	83	83	83	83	FF	83	83	FF	FF	83	83	FF	FF	FF

Example 3 This example uses transparency. Before instruction execution, PMASK = 0000h and CONTROL = 0020h (T=1, W=00, PP=00000).

After instruction execution, memory contains the following values:

Y Address	X Address																			
	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	30	31	32	34
30	FF	FF	7C	7C	FF	FF	FF	7C	FF	FF	7C	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
31	FF	FF	7C	7C	7C	FF	FF	7C	7C	FF	7C	FF	7C	FF	FF	FF	7C	FF	FF	FF	FF
32	FF	FF	7C	7C	FF	7C	FF	7C	FF	7C	7C	FF	FF	7C	FF	FF	FF	7C	FF	FF	FF
33	FF	FF	7C	7C	7C	7C	FF	7C	7C	7C	7C	FF	7C	7C	FF	FF	7C	7C	FF	FF	FF

Example 4 This example uses window operation 3 (clipped destination). Before instruction execution, PMASK = 0000h and CONTROL = 00C0h (T=0, W=11, PP=00000).

After instruction execution, memory contains the following values:

		X Address																		
Y Address		20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	30	31	32	33	34
	30	FF	FF	FF	FF	FF	FF	00	7C	00	00	7C	00	00	00	00	00	00	00	FF	FF	FF
	31	FF	FF	FF	FF	FF	FF	00	7C	7C	00	7C	00	7C	00	00	00	7C	00	FF	FF	FF
	32	FF	FF	FF	FF	FF	FF	00	7C	00	7C	7C	00	00	7C	00	00	00	7C	FF	FF	FF
	33	FF	FF	FF	FF	FF	FF	00	7C	7C	7C	7C	00	7C	7C	00	00	7C	7C	FF	FF	FF

Example 5 This example uses plane masking; the four LSBs of each pixel are masked. Before instruction execution, PMASK = 0F0Fh and CONTROL = 0020h (T=1, W=00, PP=00000).

After instruction execution, memory contains the following values:

		X Address																		
Y Address		20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	30	31	32	33	34
	30	FF	FF	7F	7F	FF	FF	FF	7F	FF	FF	7F	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
	31	FF	FF	7F	7F	7F	FF	FF	7F	7F	FF	7F	FF	7F	FF	FF	FF	7F	FF	FF	FF	FF
	32	FF	FF	7F	7F	FF	7F	FF	7F	FF	7F	7F	FF	FF	7F	FF	FF	FF	7F	FF	FF	FF
	33	FF	FF	7F	7F	7F	7F	FF	7F	7F	7F	7F	FF	7F	7F	FF	FF	7F	7F	FF	FF	FF

Example 6 This example shows how to use the PIXBLT B,XY instruction's window preclipping capabilitied when the source pitch is not a power of 2.

```
*------------------------------------------------------------------------
* Assume that the registers have been loaded as follows:
*   B0 = linear start address of source bitmap
*   B1 = SPTCH (no restrictions)
*   B2 = start y coord. ytop in 16 MSBs, start x coord.
*        xleft in 16 LSBs
*   B3 = DPTCH (must be power of 2)
*   B4 = OFFSET
*   B5 = WSTART
*   B6 = WEND
*   B7 = DY::DX (array height in 16 MSBs, array width in 16 LSBs)
*   B8 = COLOR0
*   B9 = COLOR1
*   FS1 >= 16 (assume multiplier for MPYS below is less than 16 bits)
*   CONVSP will not be used.
*   Implied operands in other I/O registers (incl. CONVDP) are valid.
*   Window option = 3
*------------------------------------------------------------------------

_color_expand:

        MOVY    B2,B10          ;copy ytop
        SUBXY   B5,B10          ;window y overlap = ytop - ystart
        JRYNN   INWINDOW        ;jump if ytop below top of window
* Need to clip destination array to top edge of clipping window
        MOVY    B5,B2           ;clip ytop to top of window
        SRA     16,B10          ;shift y overlap to 16 LSBs
        MOVE    B1,B11          ;copy SPTCH
        MPYS    B10,B11         ;(y overlap) * SPTCH
        SUB     B11,B0          ;clip SADDR to top of window
        SLL     16,B10          ;shift y overlap to 16 MSBs
        ADDXY   B10,B7          ;clip DY to top of window
        JRLS    DONE            ;done if DY<=0 (completely
                                ;above window)
* PIXBLT instruction will do any additional clipping required
INWINDOW:
        PIXBLT  B,XY            ;color expand bitmap to screen
* Restore registers and return
DONE:
        RETS    0               ;done
        .end
```

Syntax **PIXBLT L, L**

Execution linear pixel array → linear pixel array (with processing)

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0

Description The PIXBLT instruction transfers and processes a source pixel array with a destination pixel array.

This instruction operates on two-dimensional arrays of pixels using linear starting addresses for both the source and the destination. As the PixBlt proceeds, the source pixels are combined with the corresponding destination pixels based on the selected graphics operations.

Note that the parameters are entered exactly as shown in the syntax; that is, the instruction is entered as `PIXBLT L,L`. The first parameter, **L**, indicates that the starting address of the source array is a linear address; the second parameter, **L**, indicates that the starting address of the destination array is also a linear address.

The following set of implied operands govern the operation of the instruction and define the source and destination arrays.

Implied Operands

B File Registers			
Register	**Name**	**Format**	**Description**
B0†‡	SADDR	Linear	Source pixel array starting address
B1†	SPTCH	Linear	Source pixel array pitch
B2†‡	DADDR	Linear	Destination pixel array starting address
B3	DPTCH	Linear	Destination pixel array pitch
B7	DYDX	XY	Pixel array dimensions (rows:columns)
B10–B14†			Reserved registers
I/O Registers			
Address	**Name**	**Description and Elements (Bits)**	
C000000B0h	CONTROL	PP– Pixel processing operations (22 options) T – Transparency operation PBH– PixBlt horizontal direction PBV– PixBlt vertical direction	
C0000150h	PSIZE	Pixel size (1,2,4,8,16)	
C0000160h	PMASK	Plane mask – pixel format	

† These registers are changed by PIXBLT execution.
‡ You must adjust SADDR and DADDR to correspond to the corner selected by the values of PBH and PBV. See **Corner Adjust** below for additional information.

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the PIXBLT instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an instruction that reads the same location (such as a MOVE SAddress,Rd instruction). For more information, refer to Section 6.2, Latency of Writes to I/O Registers.

Source Array The source pixel array for the processing operation is defined by the contents of the SADDR, SPTCH, and DYDX registers:

- At the outset of the instruction, SADDR contains the **linear** address of the pixel at the appropriate starting corner of the array as determined by the PBH and PBV bits in the CONTROL I/O register. (See **Corner Adjust** below.) During instruction execution, SADDR points to the next pixel (or word of pixels) to be read from the source array. When the block transfer is complete, SADDR points to the starting address of the next set of 32 pixels that would have been moved had the block transfer continued.
- SPTCH contains the linear difference in the starting addresses of adjacent rows of the source array. SPTCH must be a multiple of 16.
- DYDX specifies the dimensions, in pixels, of both the source and destination arrays. The DY portion of DYDX contains the number of rows in the array; the DX portion contains the number of pixels per row.

Destination Array The location of the destination pixel array is defined by the contents of the DADDR, DPTCH, and DYDX registers:

- At the outset of the instruction, DADDR contains the **linear** address of the pixel at the appropriate starting corner of the array as determined by the PBH and PBV bits in the CONTROL I/O register. (See **Corner Adjust** below.) During instruction execution, DADDR points to the next pixel (or word of pixels) to be modified in the destination array. When the block transfer is complete, DADDR points to the linear address of the first pixel on the **next** row of pixels that would have been moved had the block transfer continued.
- DPTCH contains the linear difference in the starting addresses of adjacent rows of the destination array. DPTCH must be a multiple of 16.
- DYDX specifies the dimensions, in pixels, of both the source and destination arrays in pixels. The DY portion of DYDX contains the number of rows in the array, while the DX portion contains the number of columns.

Corner Adjust The PBH and PBV bits in the CONTROL I/O register govern the direction of the PixBlt. If the source and destination arrays overlap, then PBH and PBV should be set to prevent any portion of the source array from being overwritten before it is moved.

Note that this PIXBLT's corner adjustment is unique. The PBH and PBV bits control the direction of the PIXBLT; however, the adjustment of SADDR and DADDR to point to the appropriate starting corner is not automatic. You must explicitly set these two registers to point to the selected starting corners of the source and destination arrays, respectively, as indicated by PBH and PBV. This facility allows you to use corner adjust for screen definitions that do not lend themselves to XY addressing (those not binary powers of two). In effect, you supply your own corner adjust operation in software and the PixBlt instruction provides directional control.

- For **PBH = 0** and **PBV = 0**, set SADDR and DADDR as they are normally set for linear PixBlts. Set both registers to correspond to the linear address of the **first** pixel on the **first** line of the array (that is, the pixel with the lowest address).

- For **PBH = 0** and **PBV = 1**, set SADDR and DADDR to correspond to the linear address of the **first** pixel on the **last** line of the array. In other words,

  SADDR = (linear address of 1st source pixel) + [(DY-1) × SPTCH)]

  and

  DADDR = (linear address of 1st destination pixel) + [(DY-1) × DPTCH)]

- For **PBH = 1** and **PBV = 0**, set SADDR and DADDR to correspond to the linear address of the *pixel following* the **last** pixel on the **first** line of the array. In other words,

  SADDR = (linear address of 1st source pixel) + (DX × PSIZE)

  and

  DADDR = (linear address of 1st destination pixel) + (DX × PSIZE)

- For **PBH = 1** and **PBV = 1**, set SADDR and DADDR to correspond to the linear address of the *pixel following* the **last** pixel on the **last** line of the array. In other words,

  SADDR = (linear address of 1st source pixel) + [(DY-1) × SPTCH)] + (DX × PSIZE)

  and

  DADDR = (linear address of 1st destination pixel) + [(DY-1) × DPTCH)]+ (DX × PSIZE)

Window Checking Window operations are not enabled for this instruction. The contents of the WSTART and WEND registers are ignored.

Pixel Processing You can select a pixel processing option for this instruction by setting the PPOP bits in the CONTROL register. The pixel processing option is applied to pixels as they are processed with the destination array. Note that the data is read through the plane mask and then processed. There are 16 Boolean and 6 arithmetic operations; the default case at reset is the *replace* (S → D) operation. The 6 arithmetic operations do not operate with pixel sizes of 1 or 2 bits per pixel. For more information, see Section 7.7, Pixel Processing, on page 7-15.

Transparency You can enable transparency for this instruction by setting the T bit in the CONTROL I/O register to 1. The TMS34010 checks for 0 (transparent) pixels *after* it expands and processes the source data. At reset, the default case for transparency is *off*.

Plane Mask The plane mask is enabled for this instruction.

Interrupts This instruction can be interrupted at a word or row boundary of the destination array. When the PixBlt is interrupted, the TMS34010 sets the PBX bit in the status register and then pushes the status register on the stack. At this time, DPTCH, SPTCH, and B10–B14 contain intermediate values. DADDR points to the linear address of the next word of pixels to be modified after the interrupt is processed. SADDR points to the address of the next 32 pixels to be read from the source array after the interrupt is processed.

The PIXBLT instruction uses several I/O and B-file registers as implied operands. If an interrupt service routine modifies a register that the PIXBLT uses as an implied operand, you must restore that register to the value it had when the routine began, before returning from the routine. (You can use the MMFM and MMTM instructions to save and restore the B-file registers.) In order to maintain compatibility with future TMS340 devices, use only the RETI instruction to return from an interrupt routine.

Shift Register Transfers If the SRT bit in the DPYCTL I/O register is set, each memory read or write initiated by the PixBlt generates a shift register transfer read or write cycle at the selected address. This operation can be used for bulk memory clears or transfers. (Not all VRAMs support this capability.)

Machine States See Section 13.4, PIXBLT Instructions Timing.

Status Bits

N Undefined
C Undefined
Z Undefined
V Undefined

Examples Before executing a PIXBLT instruction, load the implied operand registers with appropriate values. These PIXBLT examples use the following implied operand setup.

Register File B:		**I/O Registers:**	
SADDR (B0)	= 00002004h	PSIZE	= 0004h
SPTCH (B1)	= 00000080h		
DADDR (B2)	= 00002228h		
DPTCH (B3)	= 00000080h		
OFFSET (B4)	= 00000000h		
DYDX (B7)	= 0002000Dh		

Additional implied operand values are listed with each example.

For these examples, assume that memory contains the following data before instruction execution.

Linear Address	**Data**							
02000h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02080h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02100h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02180h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02200h	xxxxh,	xxxxh,	FFxxh,	FFFFh,	FFFFh,	xFFFh,	xxxxh,	xxxxh
02280h	xxxxh,	xxxxh,	FFxxh,	FFFFh,	FFFFh,	xFFFh,	xxxxh,	xxxxh
02300h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh

Example 1 This example uses the *replace (S → D)* pixel processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 0000h (T=0, W=00, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	Data						
02000h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh, xxxxh
02080h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh, xxxxh
02100h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh, xxxxh
02180h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh, xxxxh
02200h	xxxxh,	xxxxh,	00xxh,	1110h,	2221h,	x332h,	xxxxh, xxxxh
02280h	xxxxh,	xxxxh,	00xxh,	1110h,	2221h,	x332h,	xxxxh, xxxxh
02300h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh, xxxxh

Example 2 This example uses the *(D - S) → D* pixel processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 4800h (T=0, W=00, PP=10010).

After instruction execution, memory contains the following values:

Linear Address	Data						
02000h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh, xxxxh
02080h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh, xxxxh
02100h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh, xxxxh
02180h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh, xxxxh
02200h	xxxxh,	xxxxh,	FFxxh,	EEEFh,	DDDEh,	xCCDh,	xxxxh, xxxxh
02280h	xxxxh,	xxxxh,	FFxxh,	EEEFh,	DDDEh,	xCCDh,	xxxxh, xxxxh
02300h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh, xxxxh

Example 3 This example uses transparency. Before instruction execution, PMASK = 0000h and CONTROL = 0020h (T=1, W=00, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	Data						
02000h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh, xxxxh
02080h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh, xxxxh
02100h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh, xxxxh
02180h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh, xxxxh
02200h	xxxxh,	xxxxh,	0FFxxh,	111Fh,	2221h,	x332h,	xxxxh, xxxxh
02280h	xxxxh,	xxxxh,	0FFxxh,	111Fh,	2221h,	x332h,	xxxxh, xxxxh
02300h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh, xxxxh

Example 4 This example uses plane masking (the MSB of each pixel is masked). Before instruction execution, PMASK = 8888h and CONTROL = 0000h (T=0, W=00, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	Data							
02000h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02080h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02100h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02180h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02200h	xxxxh,	xxxxh,	88xxh,	9998h,	AAA9h,	xBBAh,	xxxxh,	xxxxh
02280h	xxxxh,	xxxxh,	88xxh,	9998h,	AAA9h,	xBBAh,	xxxxh,	xxxxh
02300h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh

Syntax **PIXBLT L, XY**

Execution linear pixel array → XY pixel array (with processing)

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	0	1	0	0	0	0	0

Description

The PIXBLT instruction transfers and processes a source pixel array with a destination pixel array.

This instruction operates on two-dimensional arrays of pixels using a linear starting address for the source array and an XY address for the destination array. As the PixBlt proceeds, the source pixels are combined with the corresponding destination pixels based on the selected graphics operations.

Note that the parameters are entered exactly as shown in the syntax; that is, the instruction is entered as `PIXBLT L,XY`. The first parameter, **L**, indicates that the starting address of the source array is a linear address; the second parameter, **XY**, indicates that the starting address of the destination array is an XY address.

The following set of implied operands govern the operation of the instruction and define the source and destination arrays.

Implied Operands

B File Registers			
Register	**Name**	**Format**	**Description**
B0†	SADDR	Linear	Source pixel array starting address
B1	SPTCH	Linear	Source pixel array pitch
B2†‡	DADDR	XY	Destination pixel array starting address
B3	DPTCH	Linear	Destination pixel array pitch
B4	OFFSET	Linear	Screen origin (0,0)
B5	WSTART	XY	Window starting corner
B6	WEND	XY	Window ending corner
B7‡	DYDX	XY	Pixel array dimensions (rows:columns)
B10–B14†			Reserved registers
I/O Registers			
Address	**Name**	**Description and Elements (Bits)**	
C000000B0h	CONTROL	PP – Pixel processing operations (22 options) W – Window operations T – Transparency operation PBH – PixBlt horizontal direction PBV – PixBlt vertical direction	
C0000130h	CONVSP	XY-to-linear conversion (source pitch) Used for preclipping and corner adjust	
C0000140h	CONVDP	XY-to-linear conversion (destination pitch)	
C0000150h	PSIZE	Pixel size (1,2,4,8,16)	
C0000160h	PMASK	Plane mask – pixel format	

† These registers are changed by PIXBLT execution.
‡ Used for common rectangle function with window pick.

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the

PIXBLT instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an instruction that reads the same location (such as a MOVE SAddress,Rd instruction). For more information, refer to Section 6.2, Latency of Writes to I/O Registers.

Source Array The source pixel array for the processing operation is defined by the contents of the SADDR, SPTCH, DYDX, and (possibly) CONVSP registers:

- At the outset of the instruction, SADDR contains the **linear** address of the pixel with the lowest address in the array. During instruction execution, SADDR points to the next pixel (or word of pixels) to be accessed in the source array. When the block transfer is complete, SADDR points to the linear address of the first pixel on the **next** row of pixels that would have been moved had the block transfer continued.

- SPTCH contains the linear difference in the starting addresses of adjacent rows of the source array. SPTCH must be a multiple of 16. For window clipping or corner adjust, SPTCH must be a power of two and CONVSP must be set to correspond to the SPTCH value.

- CONVSP is determined by taking the LMO of the SPTCH register; this value is used for the XY calculations involved in window clipping and corner adjust.

- DYDX specifies the dimensions, in pixels, of both the source and destination arrays. The DY portion of DYDX contains the number of rows in the array; the DX portion contains the number of pixels per row.

Destination Array The location of the destination pixel array is defined by the contents of the DADDR, DPTCH, CONVDP, OFFSET, and DYDX registers:

- At the outset of the instruction, DADDR contains the **XY** address of the pixel with the lowest address in the array; it is used with OFFSET and CONVDP to calculate the linear address of the starting location of the array. During instruction execution, DADDR points to the **linear address** of next pixel (or word of pixels) to be accessed in the destination array. When the block transfer is complete, DADDR points to the **linear address** of the first pixel on the **next** row of pixels that would have been moved had the block transfer continued.

- DPTCH contains the linear difference in the starting addresses of adjacent rows of the destination array (typically this is the screen pitch). DPTCH must be a power of two (greater than or equal to 16).

- CONVDP must be set to correspond to the DPTCH value. CONVDP is determined by taking The LMO of the DPTCH register; this value is used for the XY calculations involved in XY addressing, window clipping and corner adjust.

- DYDX specifies the dimensions of both the source and destination arrays in pixels. The DY portion of DYDX contains the number of rows in the array, while the DX portion contains the number of columns.

Corner Adjust The PBH and PBV bits in the CONTROL I/O register govern the direction of the PixBlt. If the source and destination arrays overlap, then PBH and PBV should be set to prevent any portion of the source array from being overwritten before it is moved. This PixBlt performs the corner adjust function automatically under the control of the PBH and PBV bits. If PBV=1, SPTCH must be a power of two and CONVSP should be valid. The SADDR and DADDR registers should be set to correspond to the appropriate format address of the **first** pixel on the **first** line of the source (linear) and destination (XY) arrays, respectively.

Window Checking You can use window checking with this instruction by setting the W bits in the CONTROL register to the desired value. If you select window checking mode 1, 2, or 3, the WSTART and WEND registers define the XY starting and ending corners of a rectangular window.

0 *No windowing.* The entire pixel array is drawn and the WVP and V bits are unaffected.

1 *Window hit.* No pixels are drawn. The V bit is set to 0 if any portion of the destination array lies within the window. Otherwise, the V bit is set to 1.

If the V bit is set to 0, the DADDR and DYDX registers are modified to correspond to the common rectangle formed by the intersection of the destination array with the rectangular window. DADDR is set to the XY address of the pixel in the starting corner of the common rectangle. DYDX is set to the X and Y dimensions of the common rectangle.

If the V bit is set to 1, the array lies entirely outside the window, and the values of DADDR and DYDX are indeterminate.

2 *Window miss.* If the array lies **entirely** within the active window, it is drawn and the V bit is set to 0. Otherwise, no pixels are drawn, the V and WVP bits are set to 1, and the instruction is aborted.

3 *Window clip.* The source and destination arrays are preclipped to the window dimensions. Only those pixels that lie within the common rectangle (corresponding to the intersection of the specified array and the window) are drawn. If any preclipping is required, the V bit is set to 1.

Pixel Processing You can select a pixel processing option to use with this instruction by setting the PPOP bits in the CONTROL register. The pixel processing operation is applied to pixels as they are processed with the destination array. Note that the data is read through the plane mask and then processed. There are 16 Boolean and 6 arithmetic operations; the default case at reset is the *replace* (S → D) operation. The 6 arithmetic operations do not operate with pixel sizes of 1 or 2 bits per pixel. For more information, see Section 7.7, Pixel Processing, on page 7-15.

Transparency You can enable transparency for this instruction by setting the T bit in the CONTROL I/O register to 1. The TMS34010 checks for 0 (transparent) pixels *after* it expands and processes the source data. At reset, the default case for transparency is *off*.

Plane Mask The plane mask is enabled for this instruction.

Interrupts This instruction can be interrupted at a word or row boundary of the destination array. When the PixBlt is interrupted, the TMS34010 sets the PBX bit in the status register and then pushes the status register on the stack. At this time, DPTCH, SPTCH, and B10–B14 contain intermediate values. DADDR points to the linear address of the next word of pixels to be modified after the interrupt is processed. SADDR points to the address of the next 32 pixels to be read from the source array after the interrupt is processed.

The PIXBLT instruction uses several I/O and B-file registers as implied operands. If an interrupt service routine modifies a register that the PIXBLT uses as an implied operand, you must restore that register to the value it had when the routine began, before returning from the routine. (You can use the MMFM and MMTM instructions to save and restore the B-file registers.) In order to maintain compatibility with future TMS340 devices, use only the RETI instruction to return from an interrupt routine.

Shift Register Transfers If the SRT bit in the DPYCTL I/O register is set, each memory read or write initiated by the PixBlt generates a shift register transfer read or write cycle at the selected address. This operation can be used for bulk memory clears or transfers (the TMS4461 VRAM supports this capability).

Machine States See PIXBLT Instructions Timing, Section 13.4.

Status Bits

N Undefined
C Undefined
Z Undefined
V If window clipping is enabled – 1 if a window violation occurs, *0* otherwise; undefined if window clipping not enabled ($W=00_2$)

Examples Before executing a PIXBLT instruction, load the implied operand registers with appropriate values. These PIXBLT examples use the following implied operand setup.

Register File B:		**I/O Registers:**	
SADDR (B0)	= 00002004h	CONVDP	= 0017h
SPTCH (B1)	= 00000080h	PSIZE	= 0004h
DADDR (B2)	= 00520007h	PMASK	= 0000h
DPTCH (B3)	= 00000100h	CONTROL	= 0000h
OFFSET (B4)	= 00010000h		(W=00, T=0, PP=00000)
WSTART (B5)	= 0030000Ch		
WEND (B6)	= 00530014h		
DYDX (B7)	= 00030016h		

Additional implied operand values are listed with each example.

For this example, assume that memory contains the following data before instruction execution.

Linear Address	**Data**
02000h	3210h, 7654h, BA98h, FEDCh, 3210h, 7654h, BA98h, FEDCh
02080h	3210h, 7654h, BA98h, FEDCh, 3210h, 7654h, BA98h, FEDCh
02100h	3210h, 7654h, BA98h, FEDCh, 3210h, 7654h, BA98h, FEDCh
15200h to 15480h	8888h

Example 1 This example uses the *replace (S → D)* pixel processing operation. Before instruction execution, PMASK = 7777h and CONTROL = 0000h (T=0, W=00, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	**Data**
15200h	8888h, 1888h, 5432h, 9876h, DCBAh,10FEh, 5432h, 8886h
15300h	8888h, 1888h, 5432h, 9876h, DCBAh,10FEh, 5432h, 8886h
15400h	8888h, 1888h, 5432h, 9876h, DCBAh,10FEh, 5432h, 8886h

XY Addressing

		X Address																															
Y		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
A																																	
d	**52**	8	8	8	8	8	8	8	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	8	8	8	8
d																																	
r	**53**	8	8	8	8	8	8	8	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	8	8	8	8
e																																	
s	**54**	8	8	8	8	8	8	8	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	8	8	8	8
s																																	

Example 2 This example uses the *(D subs S)* → *D* pixel processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 4C00h (T=0, W=00, PP=10011).

After instruction execution, memory contains the following values:

		X Address
Y		0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
		0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F
A d d r e s s	52	8 8 8 8 8 8 8 7 6 5 4 3 2 1 0 0 0 0 0 0 0 0 8 7 6 5 4 3 2 8 8 8
	53	8 8 8 8 8 8 8 7 6 5 4 3 2 1 0 0 0 0 0 0 0 0 8 7 6 5 4 3 2 8 8 8
	54	8 8 8 8 8 8 8 7 6 5 4 3 2 1 0 0 0 0 0 0 0 0 8 7 6 5 4 3 2 8 8 8

Example 3 This example uses transparency with the *(D subs S)* → *D* pixel processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 4C20h (T=1, W=00, PP=10011).

After instruction execution, memory contains the following values:

		X Address
Y		0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
		0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F
A d d r e s s	52	8 8 8 8 8 8 8 7 6 5 4 3 2 1 8 8 8 8 8 8 8 8 8 7 6 5 4 3 2 8 8 8
	53	8 8 8 8 8 8 8 7 6 5 4 3 2 1 8 8 8 8 8 8 8 8 8 7 6 5 4 3 2 8 8 8
	54	8 8 8 8 8 8 8 7 6 5 4 3 2 1 8 8 8 8 8 8 8 8 8 7 6 5 4 3 2 8 8 8

Example 4 This example uses window operation 3 (the destination is clipped). Before instruction execution, PMASK = 0000h and CONTROL = 00C0h (T=0, W=11, PP=00000).

After instruction execution, memory contains the following values:

		X Address
Y		0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
		0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F
A d d r e s s	52	8 8 8 8 8 8 8 8 8 8 8 8 8 6 7 8 9 A B C D E 8 8 8 8 8 8 8 8 8 8
	53	8 8 8 8 8 8 8 8 8 8 8 8 8 6 7 8 9 A B C D E 8 8 8 8 8 8 8 8 8 8
	54	8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8

Example 5 This example uses plane masking (the most significant bit is masked). Before instruction execution, PMASK = 8888h and CONTROL = 0000h (T=0, W=00, PP=00000).

After instruction execution, memory contains the following values:

		X Address																													
Y		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
A																																	
d	52	8	8	8	8	8	8	8	9	A	B	C	D	E	F	8	9	A	B	C	D	E	F	8	9	A	B	C	D	E	8	8	8
d																																	
r	53	8	8	8	8	8	8	8	9	A	B	C	D	E	F	8	9	A	B	C	D	E	F	8	9	A	B	C	D	E	8	8	8
e																																	
s	54	8	8	8	8	8	8	8	9	A	B	C	D	E	F	8	9	A	B	C	D	E	F	8	9	A	B	C	D	E	8	8	8
s																																	

Syntax **PIXBLT XY, L**

Execution XY pixel array → linear pixel array (with processing)

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	0	0	0	0	0

Description The PIXBLT instruction transfers and processes a source pixel array with a destination pixel array.

This instruction operates on two-dimensional arrays of pixels using an XY starting address for the source pixel array and a linear address for the destination array. As the PixBlt proceeds, the source pixels are combined with the corresponding destination pixels based on the selected graphics operations.

Note that the parameters are entered exactly as shown in the syntax; that is, the instruction is entered as `PIXBLT XY,L`. The first parameter, **XY**, indicates that the starting address of the source array is an XY address; the second parameter, **L**, indicates that the starting address of the destination array is a linear address.

The following set of implied operands govern the operation of the instruction and define the source and destination arrays.

Implied Operands

B File Registers			
Register	**Name**	**Format**	**Description**
B0†	SADDR	XY	Source pixel array starting address
B1	SPTCH	Linear	Source pixel array pitch
B2†	DADDR	Linear	Destination pixel array starting address
B3	DPTCH	Linear	Destination pixel array pitch
B4	OFFSET	Linear	Screen origin (0,0)
B7	DYDX	XY	Pixel array dimensions (rows:columns)
B10–B14†			Reserved registers
I/O Registers			
Address	**Name**	**Description and Elements (Bits)**	
C000000B0h	CONTROL	PP – Pixel processing operations (22 options) T – Transparency operation PBH – PixBlt horizontal direction PBV – PixBlt vertical direction	
C0000130h	CONVSP	XY-to-linear conversion (source pitch) Used for XY operations	
C0000140h	CONVDP	XY-to-linear conversion (destination pitch) Used for XY operations	
C0000150h	PSIZE	Pixel size (1,2,4,8,16)	
C0000160h	PMASK	Plane mask – pixel format	

† These registers are changed by PIXBLT execution.

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the PIXBLT instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an instruction that reads the same location (such as a MOVE SAddress,Rd instruction). For more information, refer to Section 6.2, Latency of Writes to I/O Registers.

Source Array The source pixel array for the processing operation is defined by the contents of the SADDR, SPTCH, CONVSP, OFFSET, and DYDX registers:

- At the outset of the instruction, SADDR contains the **XY** address of the pixel with the lowest address in the array; it is used with OFFSET and CONVSP to calculate the linear address of the starting location of the array. During instruction execution, SADDR points to the next pixel (or word of pixels) to be accessed from the source array. When the block transfer is complete, SADDR points to the **linear address** of the first pixel on the **next** row of pixels that would have been moved had the block transfer continued.

- SPTCH contains the linear difference in the starting addresses of adjacent rows of the source array (typically this is the screen pitch). SPTCH must be a power of two (greater than or equal to 16) unless only one line is moved and CONVSP must be set to correspond to the SPTCH value.

- CONVSP is determined by taking the LMO of the SPTCH register; this value is used for the XY calculations involved in XY addressing, window clipping and corner adjust.

- DYDX specifies the dimensions, in pixels, of both the source and destination arrays. The DY portion of DYDX contains the number of rows in the array, while the DX portion contains the number of columns.

Destination Array The location of the destination pixel array is defined by the contents of the DADDR, DPTCH, DYDX, and (potentially) CONVDP registers:

- At the outset of the instruction, DADDR contains the **linear** address of the pixel with the lowest address in the array. During instruction execution, DADDR points to the next pixel (or word of pixels) to be modified in the destination array. When the block transfer is complete, DADDR points to the linear address of the first pixel on the **next** row of pixels that would have been moved had the block transfer continued.

- DPTCH contains the linear difference in the starting addresses of adjacent rows of the destination array. DPTCH must be a multiple of 16. For window clipping or corner adjust, DPTCH must be a power of two and CONVDP must be set to correspond to the DPTCH value.

- CONVDP is determined by taking the LMO of the DPTCH register; this value is used for the XY calculations involved in window clipping and corner adjust.
- DYDX specifies the dimensions of both the source and destination arrays in pixels. The DY portion of DYDX contains the number of rows in the array; the DX portion contains the number of columns.

Corner Adjust The PBH and PBV bits in the CONTROL I/O register govern the direction of the PixBlt. If the source and destination arrays overlap, then PBH and PBV should be set to prevent any portion of the source array from being overwritten before it is moved. This PixBlt performs the corner adjust function automatically under the control of the PBH and PBV bits. If PBV=1, DPTCH must be a power of two and CONVDP must be valid. The SADDR and DADDR registers should be set to correspond to the appropriate format address of the **first** pixel on the **first** line of the source (XY) and destination (linear) arrays, respectively.

Window Checking Window operations are not enabled for this instruction. The contents of the WSTART and WEND registers are ignored.

Pixel Processing You can select a pixel processing option to use with this instruction by setting the PPOP bits in the CONTROL register. The pixel processing operation is applied to pixels as they are processed with the destination array. Note that the data is read through the plane mask and then processed. There are 16 Boolean and 6 arithmetic operations; the default case at reset is the S → D operation. The 6 arithmetic operations do not operate with pixel sizes of one or two bits per pixel. For more information, see Section 7.7, Pixel Processing, on page 7-15.

Transparency You can enable transparency for this instruction by setting the T bit in the CONTROL I/O register to 1. The TMS34010 checks for 0 (transparent) pixels *after* it expands and processes the source data. At reset, the default case for transparency is *off*.

Plane Mask The plane mask is enabled for this instruction.

Interrupts This instruction can be interrupted at a word or row boundary of the destination array. When the PixBlt is interrupted, the TMS34010 sets the PBX bit in the status register and then pushes the status register on the stack. At this time, DPTCH, SPTCH, and B10–B14 contain intermediate values. DADDR points to the linear address of the next word of pixels to be modified after the interrupt is processed. SADDR points to the address of the next 32 pixels to be read from the source array after the interrupt is processed.

The PIXBLT instruction uses several I/O and B-file registers as implied operands. If an interrupt service routine modifies a register that the PIXBLT uses as an implied operand, you must restore that register to the value it had when the routine began, before returning from the routine. (You can use the MMFM and MMTM instructions to save and restore the B-file regis-

ters.) In order to maintain compatibility with future TMS340 devices, use only the RETI instruction to return from an interrupt routine.

Shift Register Transfers If the SRT bit in the DPYCTL I/O register is set, each memory read or write initiated by the PixBlt generates a shift register transfer read or write cycle at the selected address. This operation can be used for bulk memory clears or transfers (not all VRAMs support this capability).

Machine States See PIXBLT Instructions Timing, Section 13.4.

Status Bits

N Undefined
C Undefined
Z Undefined
V Undefined

Examples Before executing a PIXBLT instruction, load the implied operand registers with appropriate values. These PIXBLT examples use the following implied operand setup.

Register File B:		**I/O Registers:**	
SADDR (B0)	= 00400001h	CONVSP	= 0018h
SPTCH (B1)	= 00000080h	PSIZE	= 0004h
DADDR (B2)	= 00002228h		
DPTCH (B3)	= 00000080h		
OFFSET (B4)	= 00000000h		
DYDX (B7)	= 0002000Dh		

Additional implied operand values are listed with each example.

For this example, assume that memory contains the following data before instruction execution.

Linear Address	**Data**							
02000h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02080h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02100h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02180h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02200h	xxxxh,	xxxxh,	FFxxh,	FFFFh,	FFFFh,	xFFFh,	xxxxh,	xxxxh
02280h	xxxxh,	xxxxh,	FFxxh,	FFFFh,	FFFFh,	xFFFh,	xxxxh,	xxxxh
02300h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh

Example 1 This example uses the *replace (S → D)* pixel processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 0000h (T=0, W=00, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	Data							
02000h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02080h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02100h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02180h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02200h	xxxxh,	xxxxh,	00xxh,	1110h,	2221h,	x332h,	xxxxh,	xxxxh
02280h	xxxxh,	xxxxh,	00xxh,	1110h,	2221h,	x332h,	xxxxh,	xxxxh
02300h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh

Example 2 This example uses the *0s → D* pixel processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 0C00h (T=0, W=00, PP=00011).

After instruction execution, memory contains the following values:

Linear Address	Data							
02000h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02080h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02100h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02180h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02200h	xxxxh,	xxxxh,	00xxh,	0000h,	0000h,	x000h,	xxxxh,	xxxxh
02280h	xxxxh,	xxxxh,	00xxh,	0000h,	0000h,	x000h,	xxxxh,	xxxxh
02300h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh

Example 3 This example uses transparency. Before instruction execution, PMASK = 0000h and CONTROL = 0200h (T=1, W=00, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	Data							
02000h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02080h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02100h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02180h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02200h	xxxxh,	xxxxh,	FFxxh,	111Fh,	2221h,	x332h,	xxxxh,	xxxxh
02280h	xxxxh,	xxxxh,	FFxxh,	111Fh,	2221h,	x332h,	xxxxh,	xxxxh
02300h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh

Example 4 This example uses plane masking the two MSBs of each pixel are masked. Before instruction execution, PMASK = CCCCh and CONTROL = 0000h (T=0, W=00, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	Data							
02000h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02080h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02100h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02180h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02200h	xxxxh,	xxxxh,	CCxxh,	DDDCh,	EEEDh,	xFFEh,	xxxxh,	xxxxh
02280h	xxxxh,	xxxxh,	CCxxh,	DDDCh,	EEEDh,	xFFEh,	xxxxh,	xxxxh
02300h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh

Syntax **PIXBLT XY, XY**

Execution XY pixel array → XY pixel array (with processing)

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	1	0	0	0	0	0

Description The PIXBLT instruction transfers and processes a source pixel array with a destination pixel array.

This instruction operates on two-dimensional arrays of pixels using XY starting addresses for both the source and destination pixel arrays. As the PixBlt proceeds, the source pixels are combined with the corresponding destination pixels based on the selected graphics operations.

Note that the parameters are entered exactly as shown in the syntax; that is, the instruction is entered as `PIXBLT XY,XY`. The first **XY** indicates that the starting address of the source array is an XY address; the second **XY** indicates that the starting address of the destination array is also an XY address.

The following set of implied operands govern the operation of the instruction and define the source and destination arrays.

Implied Operands

B File Registers			
Register	**Name**	**Format**	**Description**
B0†	SADDR	XY	Source pixel array starting address
B1	SPTCH	Linear	Source pixel array pitch
B2†‡	DADDR	XY	Destination pixel array starting address
B3	DPTCH	Linear	Destination pixel array pitch
B4	OFFSET	Linear	Screen origin (0,0)
B5	WSTART	XY	Window starting corner
B6	WEND	XY	Window ending corner
B7‡	DYDX	XY	Pixel array dimensions (rows:columns)
B10–B14†			Reserved registers
I/O Registers			
Address	**Name**	**Description and Elements (Bits)**	
C00000B0h	CONTROL	PP – Pixel processing operations (22 options) W – Window clipping or pick operation T – Transparency operation PBH– PixBlt horizontal direction PBV– PixBlt vertical direction	
C0000130h	CONVSP	XY-to-linear conversion (source pitch)	
C0000140h	CONVDP	XY-to-linear conversion (destination pitch)	
C0000150h	PSIZE	Pixel size (1,2,4,8,16)	
C0000160h	PMASK	Plane mask – pixel format	

† These registers are changed by PIXBLT execution.
‡ Used for common rectangle function with window pick.

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the PIXBLT instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an instruction that reads the same location (such as a MOVE SAddress,Rd instruction). For more information, refer to Section 6.2, Latency of Writes to I/O Registers.

Source Array The source pixel array for the processing operation is defined by the contents of the SADDR, SPTCH, CONVSP, OFFSET, and DYDX registers:

- At the outset of the instruction, SADDR contains the **XY** address of the pixel with the lowest address in the array; it is used with OFFSET and CONVSP to calculate the linear address of the starting location of the array. During instruction execution, SADDR points to the next pixel (or word of pixels) to be read from the source array. When the block transfer is complete, SADDR points to the **linear address** of the first pixel on the **next** row of pixels that would have been moved had the block transfer continued.

- SPTCH contains the linear difference in the starting addresses of adjacent rows of the source array (typically this is the screen pitch). SPTCH must be a power of two (greater than or equal to 16) and CONVSP must be set to correspond to the SPTCH value.

- CONVSP is determined by taking the LMO of the SPTCH register; this value is used for the XY calculations involved in XY addressing, window clipping and corner adjust.

- DYDX specifies the dimensions of both the source and destination arrays in pixels. The DY portion of DYDX contains the number of rows in the array, while the DX portion contains the number of columns.

Destination Array The location of the destination pixel array is defined by the contents of the DADDR, DPTCH, CONVDP, OFFSET, and DYDX registers:

- At the outset of the instruction, DADDR contains the **XY** address of the pixel with the lowest address in the array; it is used with OFFSET and CONVDP to calculate the linear address of the starting location of the array. During instruction execution, DADDR points to the next pixel (or word of pixels) to be read from the destination array. When the block transfer is complete, DADDR points to the **linear address** of the first pixel on the **next** row of pixels that would have been moved had the block transfer continued.

- DPTCH contains the linear difference in the starting addresses of adjacent rows of the destination array (typically this is the screen pitch). DPTCH must be a power of two (greater than or equal to 16) and CONVDP must be set to correspond to the DPTCH value.

- CONVDP is determined by taking the LMO of the DPTCH register; this value is used for the XY calculations involved in XY addressing, window clipping and corner adjust.

- DYDX specifies the dimensions of both the source and destination arrays in pixels. The DY portion of DYDX contains the number of rows in the array, while the DX portion contains the number of columns.

Window Checking

You can use window checking with this instruction by setting the W bits in the CONTROL register to the desired value. If you select window checking mode 1, 2, or 3, the WSTART and WEND registers define the XY starting and ending corners of a rectangular window.

0 *No windowing.* The entire pixel array is drawn and the WVP and V bits are unaffected.

1 *Window hit.* No pixels are drawn. The V bit is set to 0 if any portion of the destination array lies within the window; otherwise, the V bit is set to 1.

If the V bit is set to 0, the DADDR and DYDX registers are modified to correspond to the common rectangle formed by the intersection of the destination array with the rectangular window. DADDR is set to the XY address of the pixel in the starting corner of the common rectangle. DYDX is set to the X and Y dimensions of the common rectangle.

If the V bit is set to 1, the array lies entirely outside the window, and the values of DADDR and DYDX are indeterminate.

2 *Window miss.* If the array lies **entirely** within the active window, it is drawn and the V bit is set to 0; otherwise, no pixels are drawn, the V and WVP bits are set to 1, and the instruction is aborted.

3 *Window clip.* The source and destination arrays are preclipped to the window dimensions. Only those pixels that lie within the common rectangle (corresponding to the intersection of the specified array and the window) are drawn. If any preclipping is required, the V bit is set to 1.

Pixel Processing

You can select a pixel processing option to use with this instruction by setting the PPOP bits in the CONTROL register. The pixel processing operation is applied to pixels as they are processed with the destination array. Note that the data is read through the plane mask and then processed. There are 16 Boolean and 6 arithmetic operations; the default case at reset is the *replace* (S → D) operation. The 6 arithmetic operations do not operate with pixel sizes of one or two bits per pixel. For more information, see Section 7.7, Pixel Processing, on page 7-15.

Corner Adjust The PBH and PBV bits in the CONTROL I/O register govern the direction of the PixBlt. If the source and destination arrays overlap, then PBH and PBV should be set to prevent any portion of the source array from being overwritten before it is moved. This PixBlt performs the corner adjust

function automatically under the control of the PBH and PBV bits. The SADDR and DADDR registers should be set to correspond to the appropriate format address of the **first** pixel on the **first** line of the source (XY) and destination (XY) arrays, respectively.

Transparency You can enable transparency for this instruction by setting the T bit in the CONTROL I/O register to 1. The TMS34010 checks for 0 (transparent) pixels *after* it expands and processes the source data. At reset, the default case for transparency is *off*.

Plane Mask The plane mask is enabled for this instruction.

Interrupts This instruction can be interrupted at a word or row boundary of the destination array. When the PixBlt is interrupted, the TMS34010 sets the PBX bit in the status register and then pushes the status register on the stack. At this time, DPTCH, SPTCH, and B10–B14 contain intermediate values. DADDR points to the linear address of the next word of pixels to be modified after the interrupt is processed. SADDR points to the address of the next 32 pixels to be read from the source array after the interrupt is processed.

The PIXBLT instruction uses several I/O and B-file registers as implied operands. If an interrupt service routine modifies a register that the PIXBLT uses as an implied operand, you must restore that register to the value it had when the routine began, before returning from the routine. (You can use the MMFM and MMTM instructions to save and restore the B-file registers.) In order to maintain compatibility with future TMS340 devices, use only the RETI instruction to return from an interrupt routine.

Shift Register Transfers If the SRT bit in the DPYCTL I/O register is set, each memory read or write initiated by the PixBlt generates a shift register transfer read or write cycle at the selected address. This operation can be used for bulk memory clears or transfers. (Not all VRAMs support this capability.)

Machine States See Section 13.4, PIXBLT Instructions Timing.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V If window clipping is enabled – 1 if a window violation occurs, *0* otherwise; unaffected if window clipping not enabled

Examples Before executing a PIXBLT instruction, load the implied operand registers with appropriate values. These PIXBLT examples use the following implied operand setup.

Register File B:		**I/O Registers:**	
SADDR (B0)	= 00200004h	CONVSP	= 0016h
SPTCH (B1)	= 00000200h	CONVDP	= 0016h
DADDR (B2)	= 00410004h	PSIZE	= 0004h
DPTCH (B3)	= 00000200h	PMASK	= 0000h
OFFSET(B4)	= 00010000h	CONTROL	= 0000h
WSTART(B5)	= 00300009h		(W=00, T=0, PP=00000)
WEND (B6)	= 00420012h		
DYDX (B7)	= 00030016h		

Additional implied operand values are listed with each example. For this example, assume that memory contains the following data before instruction execution.

Linear Address	**Data**
14000h	3210h, 7654h, 0BA98h,FEDCh, 3210h, 7654h, 0BA98h,FEDCh
14200h	3210h, 7654h, 0BA98h,FEDCh, 3210h, 7654h, 0BA98h,FEDCh
14400h	3210h, 7654h, 0BA98h,FEDCh, 3210h, 7654h, 0BA98h,FEDCh
18200h to 18680h	3333h

Example 1 This example uses the *replace (S →D)* pixel processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 0000h (T=0, W=00, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	**Data**
18200h	3333h, 7654h, BA98h, FEDCh, 3210h, 7654h, 3398h, 3333h
18400h	3333h, 7654h, BA98h, FEDCh, 3210h, 7654h, 3398h, 3333h
18600h	3333h, 7654h, BA98h, FEDCh, 3210h, 7654h, 3398h, 3333h

XY Addressing

		X Address																													
Y		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
A																																	
d	**41**	3	3	3	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	3	3	3	3	3	3
d																																	
r	**42**	3	3	3	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	3	3	3	3	3	3
e																																	
s	**43**	3	3	3	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	3	3	3	3	3	3
s																																	

Example 2 This example uses the *(D adds S)* → *D* pixel processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 4400h (T=0, W=00, PP=10001).

After instruction execution, memory contains the following values:

		X Address																														
Y		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
A d d r e s s	41	3	3	3	3	7	8	9	A	B	C	D	E	F	F	F	F	3	4	5	6	7	8	9	A	B	C	3	3	3	3	3	3
	42	3	3	3	3	7	8	9	A	B	C	D	E	F	F	F	F	3	4	5	6	7	8	9	A	B	C	3	3	3	3	3	3
	43	3	3	3	3	7	8	9	A	B	C	D	E	F	F	F	F	3	4	5	6	7	8	9	A	B	C	3	3	3	3	3	3

Example 3 This example uses transparency and the *(D SUBS S)* → *D* pixel processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 4C20h (T=1, W=00, PP=10011).

After instruction execution, memory contains the following values:

		X Address																														
Y		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
A d d r e s s	41	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	1	3	3	3	3	3	3	3	3	3	3	3	3	3	3
	42	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	1	3	3	3	3	3	3	3	3	3	3	3	3	3	3
	43	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	1	3	3	3	3	3	3	3	3	3	3	3	3	3	3

Example 4 This example uses window operation 3 (the destination is clipped). Before instruction execution, PMASK = 0000h and CONTROL = 00C0h (T=0, W=11, PP=00000).

After instruction execution, memory contains the following values:

		X Address																														
Y		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
A d d r e s s	41	3	3	3	3	3	3	3	3	3	9	A	B	C	D	E	F	0	1	2	3	3	3	3	3	3	3	3	3	3	3	3	3
	42	3	3	3	3	3	3	3	3	3	9	A	B	C	D	E	F	0	1	2	3	3	3	3	3	3	3	3	3	3	3	3	3
	43	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

Example 5 This example uses plane masking the third least significant bit is masked. Before instruction execution, PMASK = 5555h and CONTROL = 0000h (T=0, W=00, PP=00000).

After instruction execution, memory contains the following values:

		X Address																													
Y		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Address	41	3	3	3	3	1	1	3	3	9	9	B	B	9	9	B	B	1	1	3	3	1	1	3	3	9	9	3	3	3	3	3	3
	42	3	3	3	3	1	1	3	3	9	9	B	B	9	9	B	B	1	1	3	3	1	1	3	3	9	9	3	3	3	3	3	3
	43	3	3	3	3	1	1	3	3	9	9	B	B	9	9	B	B	1	1	3	3	1	1	3	3	9	9	3	3	3	3	3	3

Syntax **PIXT** *Rs, *Rd*

Execution pixel in Rs → *Rd

(Note that Rd contains a **linear** address.)

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	Rs				R	Rd			

Description This PIXT instruction transfers a pixel from a register to memory. The source pixel is the 1, 2, 4, 8, or 16 LSBs of the source register, depending on the pixel size specified in the PSIZE I/O register. The destination register contains a linear address; the source pixel is transferred to this memory location.

Rs and Rd must be in the same register file.

Implied Operands

I/O Registers		
Address	**Name**	**Description and Elements (Bits)**
C000000B0h	CONTROL	PP – Pixel processing operations (22 options) T – Transparency operation
C0000150h	PSIZE	Pixel size (1,2,4,6,8,16)
C0000160h	PMASK	Plane mask – pixel format

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the PIXT instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an instruction that reads the same location (such as a MOVE SAddress,Rd instruction). For more information, refer to Section 6.2, Latency of Writes to I/O Registers.

Pixel Processing You can select a pixel processing option to use with this instruction by setting the PPOP bits in the CONTROL register. The pixel processing operation is applied to the pixel as it is transferred to the destination location. The default case at reset is the *replace* option. For more information, see Section 7.7, Pixel Processing, on page 7-15.

Window Checking Window checking **cannot** be used with this instruction. The W bits are ignored.

Transparency You can enable transparency for this instruction by setting the T bit in the CONTROL I/O register to 1. The TMS34010 checks for 0 (transparent) source pixels *after* it processes the source data. At reset, the default case for transparency is *off*.

Plane Mask The plane mask is enabled for this instruction.

Machine States

Pixel Processing Operation							
PSIZE	**Replace**	**Boolean**	**ADD**	**ADDS**	**SUB**	**SUBS**	**MIN/MAX**
1,2,4,8 16	2+(3),8 2+(1),6	4+(3),10 4+(1),8	4+(3),11 4+(1),8	5+(3),11 5+(1),9	5+(3),12 5+(1),9	6+(3),11 6+(1),10	5+(3),10 5+(1),9

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

```
PIXT A0,*A1
```

Before **After**

	A0	A1	@20500h	F SIZE	PP	T	PMASK	@20500h
1)	0000FFFFh	00020500h	0000h	0001h	00000	0	0000h	0001h
1)	0000FFFFh	00020500h	0000h	0002h	00000	0	0000h	0003h
1)	0000FFFFh	00020500h	0000h	0004h	00000	0	0000h	000Fh
1)	0000FFFFh	00020500h	0000h	0008h	00000	0	0000h	00FFh
1)	0000FFFFh	00020500h	0000h	0010h	00000	0	0000h	FFFFh
1)	00000006h	00020508h	0000h	0004h	00000	0	0000h	0600h
2)	00000006h	00020508h	0300h	0004h	01010	0	0000h	0500h
3)	00000006h	00020508h	0100h	0004h	00001	0	0000h	0000h
4)	00000006h	00020508h	0100h	0004h	00001	1	0000h	0100h
5)	00000006h	00020508h	0000h	0004h	00000	0	AAAAh	0400h

Notes:

1) S replaces D
2) (S XOR D) replaces D
3) (S AND D) = 0, transparency is off, D is replaced
4) (S + D) = 0, transparency is on, D is not replaced
5) S replaces unmasked bits of D

Syntax **PIXT** *Rs, *Rd*.**XY**

(Note that Rd contains an **XY** address.)

Execution pixel in Rs → *Rd.XY

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	Rs				R	Rd			

Description This PIXT instruction transfers a pixel from a register to memory. The source pixel is the 1, 2, 4, 8, or 16 LSBs of the source register, depending on the pixel size specified in the PSIZE I/O register. The destination register contains an XY address; the X value occupies the 16 LSBs of the register and the Y value occupies the 16 MSBs. The source pixel is moved to the XY address specified in Rd.

Rs and Rd must be in the same register file.

Implied Operands

B File Registers			
Register	**Name**	**Format**	**Description**
B3	DPTCH	Linear	Destination pitch
B4	OFFSET	Linear	Screen origin (0,0)
B5	WSTART	XY	Window starting corner
B6	WEND	XY	Window ending corner
I/O Registers			
Address	**Name**	**Description and Elements (Bits)**	
C000000B0h	CONTROL	PP– Pixel processing operations (22 options) W – Window clipping or pick operation T – Transparency operation	
C0000140h	CONVDP	XY-to-linear conversion (destination pitch)	
C0000150h	PSIZE	Pixel size (1,2,4,8,16)	
C0000160h	PMASK	Plane mask – pixel format	

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the PIXT instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an instruction that reads the same location (such as a MOVE SAddress,Rd instruction). For more information, refer to Section 6.2, Latency of Writes to I/O Registers.

Window Checking You can use window checking with this instruction by setting the W bits in the CONTROL register to the desired value. If you select window checking mode 1, 2, or 3, the WSTART and WEND registers define the XY starting and ending corners of a rectangular window. When an attempt is made to write a pixel inside or outside a window, the results depend on the selected window checking mode:

0 *No window checking.* The pixel is drawn and the WVP and V bits are unaffected.

1 *Window hit.* No pixels are drawn. The V bit is set to 0 if the pixel lies within the window; otherwise, it is set to 1.

2 *Window miss.* If the pixel lies outside the window, the V and WVP bits are set to 1 and the instruction is aborted (no pixels are drawn). Otherwise, the pixel is drawn and the V bit is set to 0.

3 *Window clip.* If the pixel lies outside the window, the V bit is set to 1 and the instruction is aborted (no pixels are drawn). Otherwise, the pixel is drawn and the V bit is set to 0.

For more information, see Section 7.10, Window Checking, on page 7-26.

Pixel Processing You can select a pixel processing option to use with this instruction by setting the PPOP bits in the CONTROL register. The pixel processing operation is applied to the pixel as it is transferred to the destination location. The default case at reset is the *replace* option. For more information, see Section 7.7, Pixel Processing, on page 7-15.

Transparency You can enable transparency for this instruction by setting the T bit in the CONTROL I/O register to 1. The TMS34010 checks for 0 (transparent) source pixels *after* it processes the source data. At reset, the default case for transparency is *off*.

Plane Mask The plane mask is enabled for this instruction.

Machine States

Pixel Processing Operation								Window Violation		
PSIZE	Replace	Boolean	ADD	ADDS	SUB	SUBS	MIN/MAX	W=1	W=2	W=3
1,2,4,8	4+(3),10	6+(3),12	6+(3),12	7+(3),13	7+(3),13	8+(3),14	7+(3),13	6,9	6,9	4,7
16	4+(1),8	6+(1),10	6+(1),10	7+(1),11	7+(1),11	8+(1),12	7+(1),11	6,9	6,9	4,7

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V *1* if the pixel lies outside the window and W=1, W=2, or W=3, *0* otherwise. Unaffected if W=0.

Examples Before executing a PIXT instruction, load the implied operand registers with appropriate values. These PIXT examples use the following implied operand setup.

Register File B:
DPTCH (B3) = 00000800h
OFFSET (B4) = 00000000h
WSTART (B5) = 00300020h
WEND (B6) = 00500142h

I/O Registers:
CONVDP = 0014h

```
PIXT A0,*A1.XY
```

Before **After**

	A0	A1	@20500h	PSIZE	PP	W	T	PMASK	@20500h
1)	0000FFFFh	00400500h	0000h	0001h	00000	00	0	0000h	0001h
1)	0000FFFFh	00400280h	0000h	0002h	00000	00	0	0000h	0003h
1)	0000FFFFh	00400140h	0000h	0004h	00000	00	0	0000h	000Fh
1)	0000FFFFh	004000A0h	0000h	0008h	00000	00	0	0000h	00FFh
1)	0000FFFFh	00400050h	0000h	0010h	00000	00	0	0000h	FFFFh
1)	00000006h	00400142h	0000h	0004h	00000	00	0	0000h	0600h
2)	00000006h	00400142h	0300h	0004h	01010	00	0	0000h	0500h
3)	00000006h	00400142h	0100h	0004h	00001	00	0	0000h	0000h
4)	00000006h	00400142h	0100h	0004h	00001	00	1	0000h	0100h
5)	00000006h	00400142h	0000h	0004h	00000	00	0	AAAAh	0400h
6)	00000006h	00400142h	0000h	0004h	00000	11	0	0000h	0600h
7)	00000006h	00400143h	0000h	0004h	00000	11	0	0000h	0000h
8)	00000006h	00400143h	0000h	0004h	00000	10	0	0000h	0000h

XY Address in A1 = Linear Address 20500h

Notes:

1) S replaces D
2) (S XOR D) replaces D
3) (S AND D) = 0, transparency is off, D is replaced
4) (S + D) = 0, transparency is on, D not replaced
5) S replaces unmasked bits of D
6) Window Option = 3, D inside window, S replaces D
7) Window Option = 3, D outside window, D not replaced, V bit set in status register
8) Window Option = 2, D outside window, D not replaced, WV interrupt generated, V bit set in status register

Syntax **PIXT** **Rs, Rd*

(Note that Rs contains a **linear** address.)

Execution pixel at *Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	Rs				R	Rd			

Description This PIXT instruction transfers a pixel from memory to a register. The source register contains a linear address; the pixel at this address is transferred into the destination register. When the pixel is moved into Rd, it is right justified and zero extended to 32 bits according to the pixel size specified in the PSIZE I/O register.

Rs and Rd must be in the same register file.

Implied Operands

I/O Registers		
Address	**Name**	**Description and Elements (Bits)**
C0000150h	PSIZE	Pixel size (1,2,4,6,8,16)
C0000160h	PMASK	Plane mask – pixel format

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the PIXT instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an instruction that reads the same location (such as a MOVE SAddress,Rd instruction). For more information, refer to Section 6.2, Latency of Writes to I/O Registers.

Window Checking Window checking **cannot** be used with this instruction. The W bits are ignored.

Pixel Processing Pixel processing **cannot** be used with this instruction.

Transparency Transparency **cannot** be used with this instruction.

Plane Mask The plane mask is enabled for this instruction.

Machine States 4,7

Status Bits

N Undefined
C Undefined
Z Undefined
V Set to 1 if the pixel is 1, set to 0 if the pixel is 0.

Examples Assume that memory contains the following values:

Address	Data
@20500h	0FFFFh
@20510h	3333h

```
PIXT *A0,A1
```

Before			After
A0	**PSIZE**	**PMASK**	**A1**
00020500h	0001h	0000h	00000001h
00020500h	0001h	FFFFh	00000000h
00020500h	0002h	0000h	00000003h
00020500h	0002h	5555h	00000002h
00020500h	0004h	0000h	0000000Fh
00020510h	0004h	9999h	00000002h
00020500h	0008h	0000h	000000FFh
00020510h	0008h	5454h	00000023h
00020500h	0010h	0000h	0000FFFFh
00020500h	0010h	BA98h	00004567h
00020510h	0010h	BA98h	00000123h

Syntax **PIXT** **Rs, *Rd*

(Note that Rs and Rd contain **linear** addresses.)

Execution pixel at *Rs → *Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	Rs				R	Rd			

Description This PIXT instruction transfers a pixel from one memory location to another. The source and destination registers both contain linear addresses. The address in Rs is the address of the source pixel; the pixel is moved into the address in Rd.

Rs and Rd must be in the same register file.

Implied Operands

I/O Registers		
Address	**Name**	**Description and Elements (Bits)**
C000000B0h	CONTROL	PP–Pixel processing operations (22 options) T –Transparency operation
C0000150h	PSIZE	Pixel size (1,2,4,6,8,16)
C0000160h	PMASK	Plane mask – pixel format

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the PIXT instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an instruction that reads the same location (such as a MOVE SAddress,Rd instruction). For more information, refer to Section 6.2, Latency of Writes to I/O Registers.

Pixel Processing You can select a pixel processing option to use with this instruction by setting the PPOP bits in the CONTROL register. The pixel processing operation is applied to the pixel as it is transferred to the destination location. The default case at reset is the *replace* option. For more information, see Section 7.7, Pixel Processing, on page 7-15.

Window Checking Window checking **cannot** be used with this instruction. The W bits are ignored.

Transparency You can enable transparency for this instruction by setting the T bit in the CONTROL I/O register to 1. The TMS34010 checks for 0 (transparent) source pixels *after* it processes the source data. At reset, the default case for transparency is *off*.

Plane Mask The plane mask is enabled for this instruction.

Machine States

Pixel Processing Operation								Window Violation		
PSIZE	Replace	Boolean	ADD	ADDS	SUB	SUBS	MIN/MAX	W=1	W=2	W=3
1,2,4,8	4+(3),10	6+(3),12	6+(3),12	7+(3),13	7+(3),13	8+(3),14	7+(3),13	–	–	–
16	4+(1),8	6+(1),10	6+(1),10	7+(1),11	7+(1),11	8+(1),12	7+(1),11	–	–	–

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

```
PIXT *A0,*A1
```

	Before							After	
	A0	A1	@20500h	PSIZE	PP	T	PMASK	@20500h	@20510h
1)	00020500h	00020508h	000Fh	0001h	00000	0	0000h	010Fh	xxxx
1)	00020500h	00020508h	000Fh	0002h	00000	0	0000h	030Fh	xxxx
1)	00020500h	00020508h	000Fh	0004h	00000	0	0000h	0F0Fh	xxxx
1)	00020500h	00020508h	00EFh	0008h	00000	0	0000h	EFEFh	xxxx
1)	00020500h	00020508h	1234h	0010h	00000	0	0000h	3434h	xx12h
2)	00020500h	00020508h	030Fh	0004h	01010	0	0000h	0C0Fh	xxxx
3)	00020500h	00020508h	010Eh	0004h	00001	0	0000h	000Eh	xxxx
4)	00020500h	00020508h	020Eh	0004h	00001	1	0000h	020Eh	xxxx
5)	00020500h	00020508h	000Fh	0004h	00000	0	AAAAh	050Fh	xxxx

Notes:

1) S replaces D
2) (S XOR D) replaces D
3) (S AND D) = 0, transparency is off, D is replaced
4) (S + D) = 0, transparency is on, D not replaced
5) S replaces unmasked bits of D

Syntax **PIXT** **Rs*.**XY**, *Rd*

(Note that Rs contains an **XY** address.)

Execution pixel at *Rs.XY → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	Rs				R	Rd			

Description This PIXT instruction transfers a pixel from a memory location to a register. The source register contains an XY address; the X value occupies the 16 LSBs of the register and the Y value occupies the 16 MSBs. The address in Rs is the address of the source pixel; this pixel is moved into the destination register. When the pixel is moved into Rd, it is right justified and zero extended to 32 bits according to the pixel size specified in the PSIZE I/O register.

Rs and Rd must be in the same register file.

Implied Operands

B File Registers			
Register	**Name**	**Format**	**Description**
B1	SPTCH	Linear	Source pitch
B4	OFFSET	Linear	Screen origin (0,0)
I/O Registers			
Address	**Name**	**Description and Elements (Bits)**	
C0000130h	CONVSP	XY-to-linear conversion (source pitch)	
C0000150h	PSIZE	Pixel size (1,2,4,8,16)	
C0000160h	PMASK	Plane mask – pixel format	

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the PIXT instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an instruction that reads the same location (such as a MOVE SAddress,Rd instruction). For more information, refer to Section 6.2, Latency of Writes to I/O Registers.

Window Checking Window checking **cannot** be used with this instruction. The W bits are ignored.

Pixel Processing Pixel processing **cannot** be used with this instruction.

Transparency Transparency **cannot** be used with this instruction.

Plane Mask The plane mask is enabled for this instruction.

Machine States 6,9

Status Bits

N Undefined
C Undefined
Z Undefined
V Set to 1 if the pixel is 1, set to 0 if the pixel is 0.

Examples

These PIXT examples use the following implied operand setup.

Register File B:
DPTCH (B3) = 800h
OFFSET (B4) = 00000000h

I/O Registers:
CONVSP = 0014h

Assume that memory address @20500h contains CF3Fh before instruction execution.

```
PIXT *A0.XY,A1
```

Before			After
A0	**PSIZE**	**PMASK**	**A1**
00400500h	0001h	0000h	00000001h
00400500h	0001h	FFFFh	00000000h
00400280h	0002h	0000h	00000003h
00400280h	0002h	AAAAh	00000001h
00400140h	0004h	0000h	0000000Fh
00400140h	0004h	9999h	00000006h
004000A0h	0008h	0000h	0000003Fh
004000A0h	0008h	8989h	00000036h
00400050h	0010h	0000h	0000CF3Fh
00400050h	0010h	7310h	00008C2Fh

Note:

The XY addresses stored in register A1 in these examples translate to the linear memory address 20500h. The pitch of the source was not changed for any of these examples.

Syntax **PIXT** ***Rs*.XY, ***Rd*.XY**

(Note that Rs and Rd contain **XY** addresses.)

Execution pixel at *Rs.XY → *Rd.XY

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	Rs				R	Rd			

Description This PIXT instruction transfers a pixel from one memory location to another. The source and destination registers both contain XY addresses; the X value occupies the 16 LSBs of the register and the Y value occupies the 16 MSBs. Rs contains the address of the source pixel; Rd contains the address where the pixel is moved.

Rs and Rd must be in the same register file.

Implied Operands

B File Registers			
Register	**Name**	**Format**	**Description**
B1	SPTCH	Linear	Source pitch
B3	DPTCH	Linear	Destination pitch
B4	OFFSET	Linear	Screen origin (0,0)
B5	WSTART	XY	Window starting corner
B6	WEND	XY	Window ending corner

I/O Registers		
Address	**Name**	**Description and Elements (Bits)**
C000000B0h	CONTROL	PP – Pixel processing operations (22 options) W – Window clipping or pick operation T – Transparency operation
C0000130h	CONVSP	XY-to-linear conversion (source pitch)
C0000140h	CONVDP	XY-to-linear conversion (destination pitch)
C0000150h	PSIZE	Pixel size (1,2,4,8,16)
C0000160h	PMASK	Plane mask – pixel format

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the PIXT instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an instruction that reads the same location (such as a MOVE SAddress,Rd instruction). For more information, refer to Section 6.2, Latency of Writes to I/O Registers.

Window Checking You can use window checking with this instruction by setting the W bits in the CONTROL register to the desired value. If you select window checking mode 1, 2, or 3, the WSTART and WEND registers define the XY starting and ending corners of a rectangular window. When an attempt is made to write a pixel inside or outside a window, the results depend on the selected window checking mode:

0 *No window checking.* The pixel is drawn and the WVP and V bits are unaffected.

1 *Window hit.* No pixels are drawn. The V bit is set to 0 if the pixel lies within the window; otherwise, it is set to 1.

2 *Window miss.* If the pixel lies outside the window, the V and WVP bits are set to 1 and the instruction is aborted (no pixels are drawn). Otherwise, the pixel is drawn and the V bit is set to 0.

3 *Window clip.* If the pixel lies outside the window, the V bit is set to 1 and the instruction is aborted (no pixels are drawn). Otherwise, the pixel is drawn and the V bit is set to 0.

For more information, see Section 7.10, Window Checking, on page 7-25.

Pixel Processing You can select a pixel processing option to use with this instruction by setting the PPOP bits in the CONTROL register. The pixel processing operation is applied to the pixel as it is transferred to the destination location. The default case at reset is the *replace* option. For more information, see Section 7.7, Pixel Processing, on page 7-15.

Transparency You can enable transparency for this instruction by setting the T bit in the CONTROL I/O register to 1. The TMS34010 checks for 0 (transparent) source pixels *after* it processes the source data. At reset, the default case for transparency is *off*.

Plane Mask The plane mask is enabled for this instruction.

Machine States

Pixel Processing Operation								Window Violation		
PSIZE	Replace	Boolean	ADD	ADDS	SUB	SUBS	MIN/MAX	W=1	W=2	W=3
1,2,4,8	7+(3),13	9+(3),15	9+(3),15	10+(3),16	10+(3),16	11+(3),17	10+(3),16	–	8,11	6,9
16	7+(1),11	9+(1),13	9+(1),13	10+(1),14	10+(1),14	11+(1),15	10+(1),14	–	8,11	6,9

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V *1* if the pixel lies outside the window and W=1, W=2, or W=3, *0* otherwise. Unaffected if W=0.

Examples These PIXT examples use the following implied operand setup.

Register File B:
SPTCH (B1) = 800h
DPTCH (B3) = 800h
OFFSET (B4) = 00000000h
WSTART (B5) = 00300020h
WEND (B6) = 00500142h

I/O Registers:
CONVSP = 0014h
CONVDP = 0014h

```
PIXT *A0.XY,*A1.XY
```

	Before								After	
	A0	**A1**	**@20500h**	**PSIZE**	**PP**	**W**	**T**	**PMASK**	**@20500h**	**@20510h**
1)	00400500h	00400508h	000Fh	0001h	00000	00	0	0000h	010Fh	xxxx
1)	00400280h	00400284h	000Fh	0002h	00000	00	0	0000h	030Fh	xxxx
1)	00400140h	00400142h	000Fh	0004h	00000	00	0	0000h	0F0Fh	xxxx
1)	004000A0h	004000A1h	00EFh	0008h	00000	00	0	0000h	EFEFh	xxxx
1)	00400050h	00400051h	CDEFh	0010h	00000	00	0	0000h	CDEFh	CDEFh
2)	00400140h	00400142h	0306h	0004h	01010	00	0	0000h	0506h	xxxx
3)	00400140h	00400142h	0106h	0004h	00001	00	0	0000h	0006h	xxxx
4)	00400140h	00400142h	0106h	0004h	10001	00	1	0000h	0106h	xxxx
5)	00400140h	00400142h	0006h	0004h	00000	00	0	AAAAh	0406h	xxxx
6)	00400140h	00400142h	0006h	0004h	00000	11	0	0000h	0606h	xxxx
7)	00400140h	00400143h	0006h	0004h	00000	11	0	0000h	0006h	xxxx
8)	00400140h	00400143h	0006h	0004h	00000	10	0	0000h	0006h	xxxx

XY Address in A0 = Linear Address 20500h

Notes:

1) S replaces D
2) (S XOR D) replaces D
3) (S AND D) = 0, transparency is off, D is replaced
4) (S + D) = 0, transparency is on, D not replaced
5) S replaces unmasked bits of D
6) Window Option = 3, D inside window, S replaces D
7) Window Option = 3, D outside window, D not replaced, V bit set in status register
8) Window Option = 2, D outside window, D not replaced, WV interrupt generated, V bit set in status register

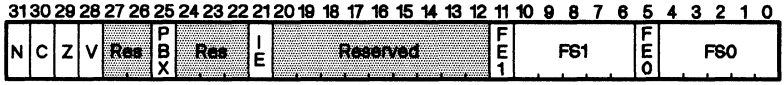
Syntax **POPST**

Execution *SP+ → ST

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0

Description POPST pops the status register from the stack and increments the SP by 32 after the status register is removed from the stack.



Status Register

Machine States 8,11 (SP aligned)
10,13 (SP nonaligned)

Status Bits

N Set from bit 31 of stack status.
C Set from bit 30 of stack status.
Z Set from bit 29 of stack status.
V Set from bit 28 of stack status.
IE Set from bit 21 of stack status.

Examples Assume that memory contains the following values before instruction execution:

Address	**Data**
0FF00000h	0010h
0FF00010h	C000h

Code	**Before**	**After**	
	SP	**ST**	**SP**
POPST	0FF00000h	C0000010h	0FF00020h

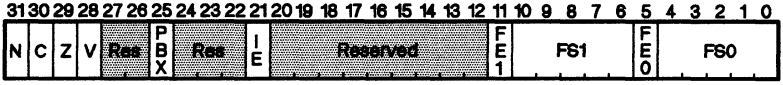
Syntax **PUSHST**

Execution ST → -*SP

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0

Description PUSHST writes the status register contents to the address contained in the SP-32.



Status Register

Machine States

2+(3),8 (SP aligned)
2+(8),13 (SP nonaligned)

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Example

Code	Before		After
	SP	**ST**	**SP**
PUSHST	0FF00020h	C0000010h	0FF00000h

Memory contains the following values after instruction execution:

Address	Data
0FF00010h	0010h
0FF00020h	C000h

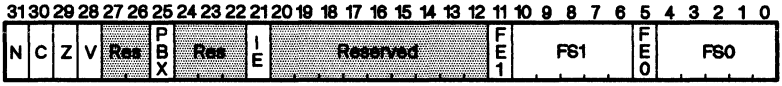
Syntax **PUTST** *Rs*

Execution Rs → ST

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1	0	1	R	Rs			

Description PUTST copies the contents of the specified register into the status register.



Status Register

Machine States 3,6

Status Bits

N Set to value of bit 31 in source register
C Set to value of bit 30 in source register
Z Set to value of bit 29 in source register
V Set to value of bit 28 in source register
IE Set to value of bit 21 in source register

Example

Code	Before		After
	A0	ST	ST
PUTST A0	C0000010h	xxxxxxxxh	C0000010h

Syntax **RETI**

Execution
*SP+ → ST
*SP+ → PC

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	0	1	0	0	0	0	0	0

Description

RETI returns to an interrupted routine from an interrupt service routine. The instruction restores the ST and PC to their original values that were stored on the system stack.

The stack is located in external memory and the top is indicated by the stack pointer (SP). The stack grows in the direction of decreasing linear address. The ST and PC are popped from the stack and the SP is incremented by 32 after each register is removed from the stack.

> **Note:**
>
> If the PBX status bit is set in the restored ST value, then the bit is cleared and a PIXBLT or FILL is resumed, depending on the values stored in the B-file registers.
>
> The CONTROL register and any B-file registers modified by an interrupt routine should be restored before RETI is executed. Otherwise, interrupted PIXBLT and FILL instructions may not resume execution correctly.

Machine States

11,14 (aligned stack)
15,18 (nonaligned stack)

Status Bits

N Copy of corresponding bit in stack location
C Copy of corresponding bit in stack location
Z Copy of corresponding bit in stack location
V Copy of corresponding bit in stack location
IE Copy of corresponding bit in stack location

Interrupts

If the IE bit in the restored ST is a 1, interrupts are enabled by the time the RETI instruction finishes executing. If an interrupt request is active during the last state of the RETI instruction, and the interrupt is enabled in the INTENB register, the interrupt will be taken immediately following the RETI. Since interrupts are level-triggered, the interrupt service routine should write to the interrupting device to clear the interrupt request before executing an RETI. The following example shows a typical interrupt service routine; in this example, the symbol `DEVICE` is the symbolic address of the interrupting device.

```
        .
        .
        .
CLR     A0
MOVE    A0, @DEVICE
        .
        .
        .
RETI
```

In this example, the interrupt request is cleared by the MOVE instruction, which writes a 0 to the device address.

Examples Assume that memory contains the following values before instruction execution:

Address	Data
CCC0000h	0010h
CCC0010h	C000h
CCC0020h	FFF0h
CCC0030h	0044h

Code	Before	After		
	SP	ST	PC	SP
RETI	CCC0000h	C0000010h	0044FFF0h	CCC0040h

Syntax **RETS** *[N]*

Execution *SP → PC (*N* defaults to 0)
SP + 32+ 16*N* → SP

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	0	1	1	N				

Description RETS returns from a subroutine by popping the program counter from the stack and optionally incrementing the stack pointer.

The *N* parameter is optional; it can be a value between 0 and 32 that indicates a number of words that are added to the stack pointer. If *N* is specified, the stack pointer is incremented by 32 + 16*N*. If *N* is not specified, the stack is incremented by 32. Execution then continues according to the PC value loaded.

Machine States 7,10 (Aligned stack)
9,12 (Unaligned stack)

Status Bits
N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples Assume that memory contains the following values before instruction execution:

Address	**Data**
0FF00000h	0FFF0h
0FF00010h	0001h

Code	**Before**	**After**	
	SP	**PC**	**SP**
RETS	0FF00000h	0001FFF0h	0FF00020h
RETS 1	0FF00000h	0001FFF0h	0FF00030h
RETS 2	0FF00000h	0001FFF0h	0FF00040h
RETS 16	0FF00000h	0001FFF0h	0FF00120h
RETS 31	0FF00000h	0001FFF0h	0FF00210h

Syntax **REV** *Rd*

Execution revision number → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	1	R	Rd			

Description REV stores the revision number of the TMS340 family device in the destination register. The revision number information is stored in the following format:

31	16	15	3	2	0
undefined		0 0 0 0 0 0 0 0 0 0 0 0 0 1		undefined	

Machine States 1,4

Status Bits
N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before	After
	A1	**A1**
REV A1	0FFFFFFFFh	00000008h

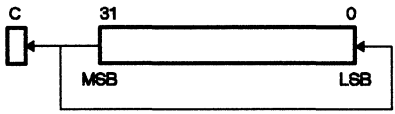
Syntax **RL** *K, Rd*

Execution left-rotate Rd by K → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	K					R	Rd			

Description RL left-rotates the contents of the destination register by a specified number of bits. (This rotation is a barrel shift.) The rotate count is specified by a 5-bit immediate value, or constant; this produces a rotation amount of 0 to 31 bits. (*K* in the syntax represents the 5-bit constant.) This is a circular rotate so that bits shifted out the MSB are shifted into the LSB.



The assembler only accepts absolute expressions for the rotate count. If the specified rotation value is greater than 31, the assembler issues a warning and set the K field in the opcode to the 5 LSBs of *K*.

The carry bit is set to the value of the last bit that is shifted out of the MSB (this value is the same as the final value of the LSB). You can use a rotate count of 0 to clear the carry and test a register for 0 simultaneously.

Machine States 1,4

Status Bits

N Unaffected
C Set to value of last bit rotated out, *0* for rotate count of 0
Z *1* if result is 0, *0* otherwise
V Unaffected

Examples

Code	Before	After	
	A1	**NCZV**	**A1**
RL 0,A1	0000000Fh	x00x	0000000Fh
RL 1,A1	F0000000h	x10x	E0000001h
RL 4,A1	F0000000h	x10x	0000000Fh
RL 5,A1	F0000000h	x00x	0000001Eh
RL 30,A1	F0000000h	x10x	3C000000h
RL 5,A1	00000000h	x01x	00000000h

Syntax **RL** *Rs, Rd*

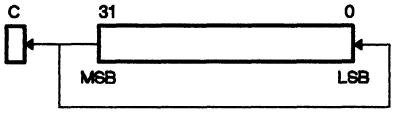
Execution left-rotate Rd by 5 LSBs of Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	Rs				R	Rd			

Operands **Rs** The five LSBs of the source register specify the left rotate count (a value from 0 to 31). The 27 MSBs are ignored.

Description RL left-rotates the contents of the destination register by a specified number of bits. (This rotation is a barrel shift.) The rotate count is specified by the 5 LSBs of Rs (the 27 MSBs are ignored); this produces a rotation amount between 0 and 31 bits. This is a circular rotate; the bits that are shifted out of the MSB of Rd are shifted into the LSB.



Note that the you must designate Rs with a keyword or symbol which has been defined to be a register, for example, A9; otherwise, the assembler uses the RL K,Rd instruction.

The carry bit is set to the value of the last bit that is shifted out of the MSB (this value is the same as the final value of the LSB).

Rs and Rd must be in the same register file.

Machine States 1,4

Status Bits
N Unaffected
C Set to value of last bit rotated out, *0* for rotate count of 0
Z *1* if result is 0, *0* otherwise
V Unaffected

Examples

Code	Before		After	
	5 LSBs A0	A1	NCZV	A1
RL A0,A1	00000	0000000Fh	x00x	0000000Fh
RL A0,A1	00100	F0000000h	x10x	0000000Fh
RL A0,A1	00101	F0000000h	x00x	0000001Eh
RL A0,A1	11111	F0000000h	x00x	78000000h
RL A0,A1	xxxxx	00000000h	x01x	00000000h

Syntax **SETC**

Execution 1 → C

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	1	1	1	0	0	0	0	0

Description SETC sets the carry bit (C) in the status register to 1. The rest of the status register is unaffected.

This instruction is useful for returning a true/false value (in the carry bit) from a subroutine without using a general-purpose register.

Machine States 1,4

Status Bits

N Unaffected
C *1*
Z Unaffected
V Unaffected

Examples

Code	Before		After	
	ST	**NCZV**	**ST**	**NCZV**
SETC	00000000h	0000	40000000h	0100
SETC	B0000010h	1011	F0000010h	1111
SETC	4000001Fh	0100	4000001Fh	0100

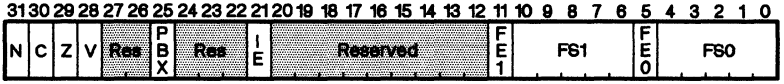
Syntax **SETF** *FS, FE [, F]*

Execution FS, FE → ST

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	F	1	0	1	FE	FS				

Description SETF loads specified field size (FS) and field extension (FE) values into the status register; depending on the value of the *F* parameter, this information sets the field size and extension for either field 0 or field 1. (The remainder of the status register is not affected.)



Status Register

- The *FS* parameter is a value between 1 and 32; it selects the field size. (Note that an FS value of 0 in the opcode corresponds to an actual selected field size of 32.)
- The *FE* parameter is a value of 0 or 1:
  FE=0 selects zero extension for a field.
  FE=1 selects sign extension for a field.
- The *F* parameter is optional; the default value for *F* is 0. The *F* value determines whether the SETF instruction sets the field size and extension for field 0 or for field 1.
  F=0 selects FS0, FE0 to be altered.
  F=1 selects FS1, FE1 to be altered.

Each MOVE instruction also has an *F* parameter that selects the field size and extension of either field 0 or field 1 for the individual move. You can use the SETF instruction to prepare for MOVE instructions.

Machine States 1,4 for F=0
2,5 for F=1

Status Bits **N** Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before	After
	ST	ST
SETF 32,0,0	xxxxx000h	xxxxx000h
SETF 32,1,0	xxxxx000h	xxxxx020h
SETF 31,1,0	xxxxx000h	xxxxx03Fh
SETF 16,0,0	xxxxx000h	xxxxx010h
SETF 32,0,1	xxxxx000h	xxxxx000h
SETF 32,1,1	xxxxx000h	xxxxx800h
SETF 31,1,1	xxxxx000h	xxxxxFC0h
SETF 16,0,1	xxxxx000h	xxxxx400h

Syntax **SEXT** *Rd [, F]*

Execution field in Rd → sign-extended field Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	F	1	0	0	0	R	Rd			

Description SEXT sign extends the right-justified field contained in the destination register by copying the MSB of the field data into all the nonfield bits of the destination register. The size of the field is determined by the current field size. The optional *F* parameter, which must be specified as a 0 or a 1, selects the field size:

F=0 selects FS0 for the field size.
F=1 selects FS1 for the field size.

The default value for *F* is 0.

Machine States 3,6

Status Bits
N *1* if the result is negative, *0* otherwise
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before		After	
	FS0/1	A0	NCZV	A0
SEXT A0,0	17/x	00008000h	0x0x	00008000h
SEXT A0,0	16/x	00008000h	1x0x	FFFF8000h
SEXT A0,0	15/x	00008000h	0x1x	00000000h
SEXT A0,1	x/17	00008000h	0x0x	00008000h
SEXT A0,1	x/16	00008000h	1x0x	FFFF8000h
SEXT A0,1	x/15	00008000h	0x1x	00000000h

Syntax **SLA** *K, Rd*

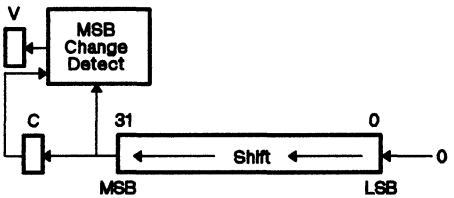
Execution left-shift Rd by K → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	K					R	Rd			

Description SLA left-shifts the contents of the destination register by a specified number of bits. The shift count is specified by a 5-bit constant (*K* in the syntax); this is a value between 0 and 31.

As shown in the diagram, 0s are shifted into the least significant bits. The last bit shifted out of the destination register (the original value of bit 32-*K*) is shifted into the carry bit. If either the new sign bit (N) or any of the bits shifted out of the register differ from the original sign bit, the overflow bit (V) is set.



The assembler accepts only absolute expressions for the shift count. If the shift count is greater than 31, the assembler issues a warning and sets the K field in the opcode to the 5 LSBs of *K*.

Note that SLA executes slower than SLL because it provides overflow detection.

Machine States 3,6

Status Bits
N *1* if the result is negative, *0* otherwise
C Set to the value of last bit shifted out, *0* for shift count of 0
Z *1* if a 0 result generated, *0* otherwise
V *1* if the MSB changes during shift operation, *0* otherwise

Examples

Code	Before	After	
	A1	A1	NCZV
SLA 0,A1	33333333h	33333333h	0000
SLA 0,A1	CCCCCCCCh	CCCCCCCCh	1000
SLA 1,A1	CCCCCCCCh	99999998h	1100
SLA 2,A1	33333333h	CCCCCCCCh	1001
SLA 2,A1	CCCCCCCCh	33333330h	0101
SLA 3,A1	CCCCCCCCh	66666660h	0001
SLA 5,A1	CCCCCCCCh	99999980h	1101
SLA 30,A1	CCCCCCCCh	00000000h	0111
SLA 31,A1	CCCCCCCCh	00000000h	0011
SLA 31,A1	00000000h	00000000h	0010

Syntax **SLA** *Rs, Rd*

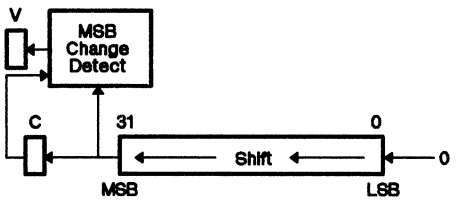
Execution left-shift Rd by 5 LSBs of Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	Rs				R	Rd			

Description SLA left-shifts the contents of the destination register by a specified number of bits. The shift count is specified by the 5 LSBs of Rs (the 27 MSBs are ignored); this produces a shift count from 0 to 31.

The last bit shifted out of the destination register (the original value of bit 32-*K*) is shifted into the carry bit. If either the new sign bit (N) or any of the bits shifted out of the register differ from the original sign bit, the overflow bit (V) is set.



Note that you must designate Rs with a keyword or symbol which has been defined to be a register, for example, A9; otherwise, the assembler uses uses the SLA K,Rd instruction. Rs and Rd must be in the same register file.

Note that SLA executes slower than SLL because it provides overflow detection.

Machine States 3,6

Status Bits

N *1* if the result is negative, *0* otherwise
C Set to value of last bit shifted out, *0* for shift count of 0
Z *1* if the result is 0, *0* otherwise
V *1* if the MSB changes during shift operation, *0* otherwise

Examples

Code	Before		After	
	5 LSBs A0	A1	A1	NCZV
SLA A0,A1	00000	33333333h	33333333h	0000
SLA A0,A1	00000	CCCCCCCCh	CCCCCCCCh	1000
SLA A0,A1	00001	CCCCCCCCh	99999998h	1100
SLA A0,A1	00010	33333333h	CCCCCCCCh	1001
SLA A0,A1	00010	CCCCCCCCh	33333330h	0101
SLA A0,A1	00011	CCCCCCCCh	66666660h	0001
SLA A0,A1	00101	CCCCCCCCh	99999980h	1101
SLA A0,A1	11110	CCCCCCCCh	00000000h	0111
SLA A0,A1	11111	CCCCCCCCh	00000000h	0011
SLA A0,A1	11111	00000000h	00000000h	0010

Syntax **SLL** *K, Rd*

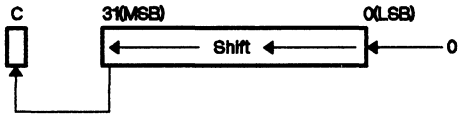
Execution left-shift Rd by K → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	K					R	Rd			

Description SLL left-shifts the contents of the destination register by a specified number of bits. The shift count is specified by a 5-bit constant (*K* in the syntax.), which is a value between 0 and 31.

The last bit shifted out of the destination register (the original value of bit 32-*K*) is shifted into the carry bit. 0s are shifted into the least significant bits. This instruction differs from the SLA instruction only in its effect on the overflow (V) bit.



The assembler only accepts absolute expressions for the shift count. If the specified shift count is greater than 31, the assembler issues a warning and sets the K field in the opcode to the 5 LSBs of *K*.

Machine States 1,4

Status Bits

N Unaffected
C *1* to the value of last bit shifted out, *0* for shift count of 0
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before	After	
	A1	A1	NCZV
SLL 0,A1	00000000h	00000000h	x01x
SLL 0,A1	88888888h	88888888h	x00x
SLL 1,A1	88888888h	11111110h	x10x
SLL 4,A1	88888888h	88888880h	x00x
SLL 30,A1	FFFFFFFCh	00000000h	x11x
SLL 31,A1	FFFFFFFCh	00000000h	x01x

Syntax **SLL** *Rs, Rd*

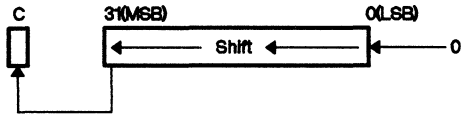
Execution left-shift Rd by 5 LSBs of Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	1	Rs				R	Rd			

Description SLL left-shifts the contents of the destination register by a specified number of bits. The shift count is specified by the 5 LSBs of Rs (the 27 MSBs are ignored); this produces a shift count between 0 and 31.

The last bit shifted out of the destination register (the original value of bit 32-*K*) is shifted into the carry bit. 0s are shifted into the least significant bits. This instruction differs from the SLA instruction only in its effect on the overflow (V) bit.



Note that you must designate Rs with a keyword or symbol which has been defined to be a register, for example, A9; otherwise, the assembler uses the SLA K,Rd instruction.

Rs and Rd must be in the same register file.

Machine States 1,4

Status Bits

N Unaffected
C Set to the value of last bit shifted out, *0* for shift value of 0
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before		After	
	5 LSBs A0	A1	A1	NCZV
SLL A0,A1	00000	00000000h	00000000h	x01x
SLL A0,A1	00000	88888888h	88888888h	x00x
SLL A0,A1	00001	88888888h	11111110h	x10x
SLL A0,A1	00100	88888888h	88888880h	x00x
SLL A0,A1	11110	FFFFFFFCh	00000000h	x11x
SLL A0,A1	11111	FFFFFFFCh	00000000h	x01x

Syntax **SRA** *K, Rd*

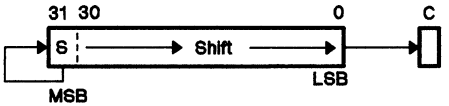
Execution right-shift Rd by K → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0	2s complement of K					R	Rd			

Description SRA right-shifts the contents of the destination register by a specified number of bits. The shift count is specified by the 2s complement of a 5-bit immediate value, or constant; this produces a shift count of 0 to 31. (*K* in the syntax represents this 5-bit constant).

The last bit shifted out of the destination register (the original value of bit *K*-1) is shifted into the carry bit. The sign bit (MSB) is extended into the most significant bits.



The assembler only accepts absolute expressions for the shift count. If the specified shift count is greater than 31, the assembler issues a warning and sets the K field in the opcode to the 2s complement of the 5 LSBs of *K*.

Machine States 1,4

Status Bits

N *1* if the result is negative, *0* otherwise
C Set to the value of last bit shifted out, *0* for shift count of 0
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before	After	
	A1	A1	NCZV
SRA 0,A1	00000000h	00000000h	001x
SRA 0,A1	FFFF0000h	FFFF0000h	100x
SRA 8,A1	7FFF0000h	007FFF00h	000x
SRA 8,A1	FFFF0000h	FFFFFF00h	100x
SRA 30,A1	7FFF0000h	00000001h	010x
SRA 31,A1	7FFF0000h	00000000h	011x
SRA 31,A1	FFFF0000h	FFFFFFFFh	110x

Syntax **SRA** *Rs, Rd*

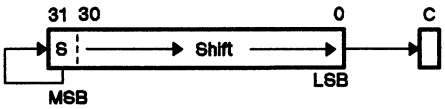
Execution right-shift Rd by 2s complement of 5 LSBs in Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	Rs				R	Rd			

Description SRA right-shifts the contents of the destination register by a specified number of bits. The shift amount is specified by the 2s complement of the 5 LSBs of Rs (the 27 MSBs of Rs are ignored); this produces a shift count between 0 and 31.

The last bit shifted out of the destination register (the original value of bit *K*-1) is shifted into the carry bit. The sign bit (MSB) is extended into the most significant bits.



You must specify Rs with a keyword or a symbol which has been defined to be a register, for example, A9; otherwise, the assembler uses the SRA K,Rd instruction. Rs and Rd must be in the same register file.

Machine States 1,4

Status Bits

N *1* if the result is negative, *0* otherwise
C Set to the value of last bit shifted out, *0* for shift count of 0
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before		After	
	5 LSBs A0	A1	A1	NCZV
SRA A0,A1	00000	00000000h	00000000h	001x
SRA A0,A1	00000	FFFF0000h	FFFF0000h	100x
SRA A0,A1	11111	7FFF0000h	3FFF8000h	000x
SRA A0,A1	11111	FFFF0000h	FFFF8000h	100x
SRA A0,A1	11000	7FFF0000h	007FFF00h	000x
SRA A0,A1	11000	FFFF0000h	FFFFFF00h	100x
SRA A0,A1	00010	7FFF0000h	00000001h	010x
SRA A0,A1	00001	7FFF0000h	00000000h	011x
SRA A0,A1	00001	FFFF0000h	FFFFFFFFh	110x

Syntax **SRL** *K, Rd*

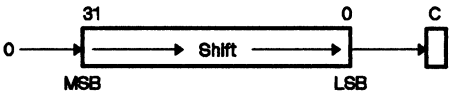
Execution right-shift Rd by 2s complement of K → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	2s complement of K					R	Rd			

Description SRL right-shifts the contents of the destination register by a specified number of bits. The shift amount is specified by the 2s complement of a 5-bit immediate value; this produces a shift count between 0 and 31. (*K* in the syntax represents the immediate value, or constant.)

The last bit shifted out of the destination register (the original value of bit *K*-1) is shifted into the carry bit. 0s are shifted into the most significant bits.



The assembler accepts only absolute expressions for the shift count. If the specified shift amount is greater than 31, the assembler issues a warning and set the K field in the opcode to the 2s complement of the 5 LSBs of K.

Machine States 1,4

Status Bits

N Unaffected
C Set to the value of last bit shifted out, *0* for shift count of 0
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before	After	
	A1	A1	NCZV
SRL 0,A1	00000000h	00000000h	x01x
SRL 0,A1	7FFFFFFFh	7FFFFFFFh	x00x
SRL 1,A1	7FFFFFFFh	3FFFFFFFh	x10x
SRL 8,A1	7FFF0000h	007FFF00h	x00x
SRL 30,A1	7FFF0000h	00000001h	x10x
SRL 31,A1	7FFF0000h	00000000h	x11x
SRL 31,A1	3FFF0000h	00000000h	x01x

Syntax **SRL** *Rs, Rd*

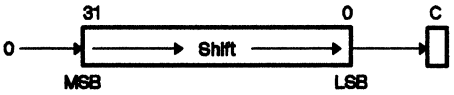
Execution right-shift Rd by 2s complement of 5 LSBs in Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	1	Rs				R	Rd			

Description SRL right-shifts the contents of the destination register by a specified number of bits. The shift amount is specified by the 2s complement of the 5 LSBs of Rs (the 27 MSBs of Rs are ignored); this produces a shift value of 0 to 31.

The last bit shifted out of the destination register (the original value of bit *K*-1) is shifted into the carry bit. 0s are shifted into the most significant bits.



You must specify Rs with a keyword or symbol which has been defined to be a register, for example, A9; otherwise, the assembler uses the SRL K,Rd instruction. Rs and Rd must be in the same register file.

Machine States 1,4

Status Bits

N Unaffected
C Set to the value of last bit shifted out, *0* for shift count of 0
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before		After	
	5 LSBs A0	A1	A1	NCZV
SRL A0,A1	00000	00000000h	00000000h	x01x
SRL A0,A1	00000	7FFFFFFFh	7FFFFFFFh	x00x
SRL A0,A1	11111	7FFFFFFFh	3FFFFFFFh	x10x
SRL A0,A1	11000	7FFF0000h	007FFF00h	x00x
SRL A0,A1	00010	7FFF0000h	00000001h	x10x
SRL A0,A1	00001	7FFF0000h	00000000h	x11x
SRL A0,A1	00001	3FFF0000h	00000000h	x01x

Syntax **SUB** *Rs, Rd*

Execution Rd - Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	Rs				R	Rd			

Description SUB subtracts the contents of the source register from the contents of the destination register and stores the result in the destination register.

You can accomplish multiple-precision arithmetic by using SUB in conjunction with the SUBB instruction.

Rs and Rd must be in the same register file.

Machine States 1,4

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a borrow, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before		After	
	A0	A1	NCZV	A0
SUB A1,A0	7FFFFFF2h	7FFFFFF1h	0000	00000001h
SUB A1,A0	7FFFFFF2h	7FFFFFF2h	0010	00000000h
SUB A1,A0	7FFFFFF1h	7FFFFFF2h	1100	FFFFFFFFh
SUB A1,A0	7FFFFFF1h	FFFFFFFFh	0100	7FFFFFF2h
SUB A1,A0	7FFFFFFFh	FFFFFFFFh	1101	80000000h
SUB A1,A0	FFFFFFFDh	FFFFFFFFh	1100	FFFFFFFEh
SUB A1,A0	FFFFFFFDh	FFFFFFFDh	0010	00000000h
SUB A1,A0	FFFFFFFEh	FFFFFFFDh	0000	00000001h
SUB A1,A0	FFFFFFFFh	00000001h	1000	FFFFFFFEh
SUB A1,A0	80000000h	00000001h	0001	7FFFFFFFh

Syntax **SUBB** *Rs, Rd*

Execution Rd - Rs - C → Rd (the carry bit acts as a borrow)

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	Rs				R	Rd			

Description SUBB subtracts both the contents of the source register and the carry bit from the contents of the destination register, and stores the result in the destination register.

You can use this instruction with the SUB, SUBK, and SUBI instructions for extended-precision arithmetic.

Rs and Rd must be in the same register file.

Machine States 1,4

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a borrow, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

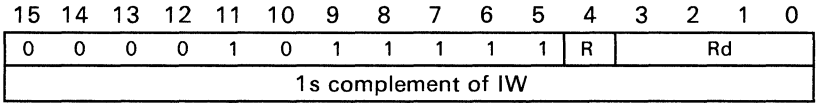
Examples

Code	Before			After	
	C	A0	A1	NCZV	A0
SUBB A1,A0	0	00000002h	00000001h	0000	00000001h
SUBB A1,A0	1	00000002h	00000001h	0010	00000000h
SUBB A1,A0	0	00000002h	00000002h	0010	00000000h
SUBB A1,A0	1	00000002h	00000002h	1100	FFFFFFFFh
SUBB A1,A0	0	00000002h	00000003h	1100	FFFFFFFFh
SUBB A1,A0	0	7FFFFFFEh	FFFFFFFFh	0100	7FFFFFFFh
SUBB A1,A0	0	7FFFFFFEh	FFFFFFFEh	1101	80000000h
SUBB A1,A0	1	7FFFFFFEh	FFFFFFFEh	0100	7FFFFFFFh
SUBB A1,A0	0	FFFFFFFEh	FFFFFFFFh	1100	FFFFFFFFh
SUBB A1,A0	0	FFFFFFFEh	FFFFFFFEh	0010	00000000h
SUBB A1,A0	1	FFFFFFFEh	FFFFFFFEh	1100	FFFFFFFFh
SUBB A1,A0	0	FFFFFFFEh	FFFFFFFDh	0000	00000001h
SUBB A1,A0	1	FFFFFFFEh	FFFFFFFDh	0010	00000000h
SUBB A1,A0	0	80000001h	00000001h	1000	80000000h
SUBB A1,A0	1	80000001h	00000001h	0001	7FFFFFFFh
SUBB A1,A0	0	80000001h	00000002h	0001	7FFFFFFFh

Syntax **SUBI** *IW, Rd [, W]*

Execution Rd - IW → Rd

Instruction Words



Description SUBI subtracts a sign-extended, 16-bit immediate value from the contents of the destination register, and stores the result in the destination register. (The *IW* in the syntax represents a sign-extended, 16-bit immediate value.)

The assembler uses this form of the SUBI instruction if the immediate value was previously defined and is in the range -32,768 to 32,767. You can force the assembler to use the short form by by following the register operand with **,W**:

```
SUBI  IW,Rd,W
```

The assembler truncates any upper bits and issues an appropriate warning message. You can accomplish multiple-precision arithmetic by using SUBI in conjunction with the SUBB instruction.

Machine States 2,8

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if a borrow is generated, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before	After	
	A0	A0	NCZV
SUBI 32765,A0	00007FFEh	00000001h	0000
SUBI 32766,A0	00007FFEh	00000000h	0010
SUBI 32767,A0	00007FFEh	FFFFFFFFh	1100
SUBI 32766,A0	80007FFEh	80000000h	1000
SUBI 32767,A0	80007FFEh	7FFFFFFFh	0001
SUBI -32766,A0	FFFF8001h	FFFFFFFFh	1100
SUBI -32767,A0	FFFF8001h	00000000h	0010
SUBI -32768,A0	FFFF8001h	00000001h	0000
SUBI -32767,A0	FFFF8000h	7FFFFFFFh	0100
SUBI -32768,A0	7FFF8000h	80000000h	1101

Syntax **SUBI** */L, Rd [, L]*

Execution Rd - IL → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	0	0	0	R	Rd			
1s complement of 16 LSBs of IL															
1s complement of 16 MSBs of IL															

Description SUBI subtracts a signed 32-bit immediate value from the contents of the destination register, and stores the result in the destination register. (The */L* in the syntax represents a signed 32-bit immediate value.)

The assembler uses this version of the SUBI instruction if it cannot use the SUBI IW,Rd opcode, or if you request the long opcode by following the register operand with **,L**:

```
SUBI  IL,Rd,L
```

You can accomplish multiple-precision arithmetic by using SUBI in conjunction with the SUBB instruction.

Machine States 3,12

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a borrow, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before	After	
	A0	A0	NCZV
SUBI 2147483647,A0	7FFFFFFFh	00000000h	0010
SUBI 32768,A0	00008001h	00000001h	0000
SUBI 32769,A0	00008001h	00000000h	0010
SUBI 32770,A0	00008001h	FFFFFFFFh	1100
SUBI 32768,A0	80008000h	80000000h	1000
SUBI 32769,A0	80008000h	7FFFFFFFh	0001
SUBI -2147483648,A0	80000000h	00000000h	0010
SUBI -32769,A0	FFFF7FFEh	FFFFFFFFh	1100
SUBI -32770,A0	FFFF7FFEh	00000000h	0010
SUBI -32771,A0	FFFF7FFEh	00000001h	0000
SUBI -32770,A0	7FFF7FFDh	7FFFFFFFh	0100
SUBI -32771,A0	7FFF7FFDh	80000000h	1101

Syntax **SUBK** *K, Rd*

Execution Rd - K → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	K					R	Rd			

Description SUBK subtracts the 5-bit constant from the contents of the destination register; the result is stored in the destination register. The *K* in the syntax represents a constant that is treated as an unsigned number in the range 1–32. Note that K=0 in the opcode corresponds to the value 32; the assembler converts the value 32 to 0. The assembler issues an error if you try to subtract 0 from a register.

You can accomplish multiple-precision arithmetic by using SUBK in conjunction with the SUBB instruction.

Machine States 1,4

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a borrow, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before	After	
	A0	A0	NCVZ
SUBK 5,A0	00000009h	00000004h	0000
SUBK 9,A0	00000009h	00000000h	0010
SUBK 32,A0	00000009h	FFFFFFE9h	1100
SUBK 1,A0	80000000h	7FFFFFFFh	0001

Syntax **SUBXY** *Rs, Rd*

Execution RdX - RsX → RdX
RdY - RsY → RdY

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	1	Rs				R	Rd			

Description SUBXY subtracts the source X and Y values individually from the destination X and Y values; the result is stored in the destination register.

You can use this instruction for manipulating XY addresses; it is particularly useful for incremental figure drawing. These addresses are stored as XY pairs in the register file.

Rs and Rd must be in the same register file.

Machine States 1,4

Status Bits
N *1* if source X field = destination X field, *0* otherwise
C *1* if source Y field > destination Y field, *0* otherwise
Z *1* if source Y field = destination Y field, *0* otherwise
V *1* if source X field > destination X field, *0* otherwise

Examples

Code	Before		After	
	A0	A1	A0	NCZV
SUBXY A1,A0	00090009h	00010001h	00080008h	0000
SUBXY A1,A0	00090009h	00090001h	00000008h	0010
SUBXY A1,A0	00090009h	00010009h	00080000h	1000
SUBXY A1,A0	00090009h	00090009h	00000000h	1010
SUBXY A1,A0	00090009h	00000010h	0009FFF9h	0001
SUBXY A1,A0	00090009h	00090010h	0000FFF9h	0011
SUBXY A1,A0	00090009h	00100000h	FFF90009h	0100
SUBXY A1,A0	00090009h	00100009h	FFF90000h	1100
SUBXY A1,A0	00090009h	00100010h	FFF9FFF9h	0101

Syntax **TRAP** *N*

Execution
PC → -*SP
ST → -*SP
trap vector *N* → PC

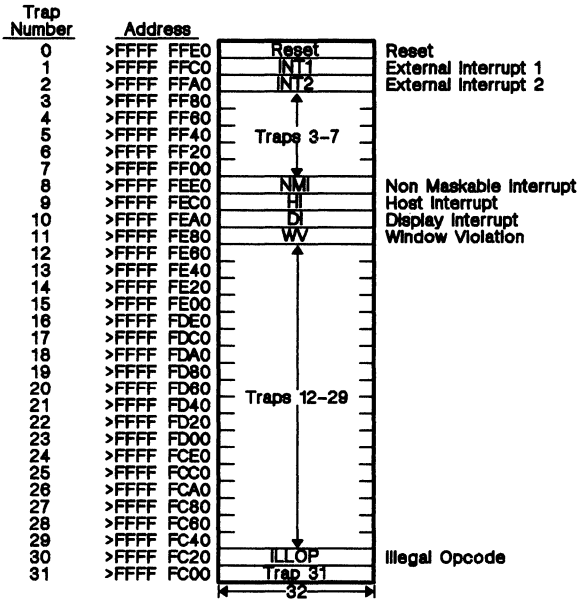
Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	0	0	0	*N*				

Description TRAP executes a software interrupt. The *N* parameter is a trap number from 0 to 31 that selects the trap to be executed. During a software interrupt,

- The return address (the address of next instruction) is pushed on the stack.
- The status register is pushed on the stack.
- The IE (interrupt enable) bit in ST is set to 0, disabling maskable interrupts, and ST is set to 00000010h.
- Finally, the trap vector is loaded into the PC.

The TMS34010 generates the trap vector addresses as shown below:



The stack, which is located in external memory, grows toward lower addresses. The PC and ST are pushed on the stack MSW first, and the SP is predecremented before each word is loaded onto the stack.

Notes:

1. The level 0 trap differs from all other traps; it does not save the old status register or program counter. This may be useful in cases where the stack pointer is corrupted or uninitialized; such a situation could cause an erroneous write.
2. The NMI bit does not affect the operation of TRAP 8.

For more information, refer to Section 8 (Interrupts, Traps, and Reset).

Machine States

16,19 (SP aligned)
30,33 (SP nonaligned)

Status Bits

N *0*
C *0*
Z *0*
V *0*

Examples

Code	Before		After		
	PC	SP	PC	SP	ST
TRAP 0	xxxxxxxxh	80000000h	FFFFFFE0h	80000000h	00000010h
TRAP 1	xxxxxxxxh	80000000h	FFFFFFC0h	7FFFFFC0h	00000010h
.	.	.	.	.	.
.	.	.	.	.	.
TRAP 30	xxxxxxxxh	80000000h	FFFFFC20h	7FFFFFC0h	00000010h
TRAP 31	xxxxxxxxh	80000000h	FFFFFC00h	7FFFFFC0h	00000010h

Syntax **XOR** *Rs, Rd*

Execution Rs XOR Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	Rs				R	Rd			

Description XOR bitwise-exclusive-ORs the contents of the source register with the contents of the destination register, and stores the result in the destination register.

You can use this instruction to clear registers (for example, `XOR B0,B0`); the CLR instruction also supports this function.

Rs and Rd must be in the same register file.

Machine States 1,4

Status Bits

N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before	After		
	A0	**A1**	**NCZV**	**A1**
`XOR A0,A1`	FFFFFFFFh	00000000h	xx0x	FFFFFFFFh
`XOR A0,A1`	FFFFFFFFh	AAAAAAAAh	xx0x	55555555h
`XOR A0,A1`	FFFFFFFFh	FFFFFFFFh	xx1x	00000000h

Syntax **XORI** *IL, Rd*

Execution IL XOR Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	1	1	0	R	Rd			
16 LSBs of IL															
16 MSBs of IL															

Description XORI bitwise exclusive ORs a 32-bit immediate data with the contents of the destination register and stores the result in the destination register. (The *IL* parameter in the syntax above represents a 32-bit immediate value.)

Machine States 3,12

Status Bits

N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before	After	
	A0	**NCZV**	**A0**
`XORI 0FFFFFFFFh,A0`	00000000h	xx0x	FFFFFFFFh
`XORI 0FFFFFFFFh,A0`	AAAAAAAAh	xx0x	55555555h
`XORI 0FFFFFFFFh,A0`	FFFFFFFFh	xx1x	00000000h
`XORI 00000000h,A0`	00000000h	xx1x	00000000h
`XORI 00000000h,A0`	FFFFFFFFh	xx0x	FFFFFFFFh

Syntax **ZEXT** *Rd [, F]*

Execution field in Rd → zero-extended field Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	F	1	0	0	1	R	Rd			

Description ZEXT zero extends a right-justified field in the destination register by zeroing all the nonfield bits in Rd. The size of the field is determined by the current field size. The optional *F* parameter, which must be specified as a 0 or a 1, selects the field size:

F=0 selects FS0 for the field size.
F=1 selects FS1 for the field size.

The default value for *F* is 0.

Machine States 1,4

Status Bits

N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before			After	
	FS0	FS1	A0	NCZV	A0
ZEXT A0,0	32	x	FFFFFFFFh	xx0x	FFFFFFFFh
ZEXT A0,0	31	x	FFFFFFFFh	xx0x	7FFFFFFFh
ZEXT A0,0	1	x	FFFFFFFFh	xx0x	00000001h
ZEXT A0,0	16	x	FFFF0000h	xx1x	00000000h
ZEXT A0,1	x	16	FFFF0000h	xx1x	00000000h

Section 13

Instruction Timings

Section 12, The TMS34010 Instruction Set, describes each TMS34010 assembly language instruction, including instruction cycle timings. This section provides details pertaining to instruction timings for the following groups of instructions:

13.1 General Instructions

> **Note:**
>
> General instructions include all TMS34010 instructions *except* MOVEs, MOVBs, FILLs, PIXBLTs, and LINE.

Each instruction description in Section 12 contains a **Machine States** field that describes the instruction execution time in terms of the machine state. A machine state is the fundamental time unit of the processor. Logically, it is the time required to decode, interpret, and execute a single microinstruction internal to the CPU. Physically, a TMS34010 machine state is equal to a single local clock period (the time from one LCLK1 low-to-high transition to the next). For example, this value is 160 nanoseconds for a TMS34010 clocked at 50 MHz, and 200 nanoseconds for a 40-MHz TMS34010.

The descriptions in the instruction discussions appear as:

Machine States *cache hit case, cache disabled case*

These two values represent the number of CPU states required to execute the instruction for each of two cases:

- The **cache hit case** gives the number of execution states if the instruction and its extension words reside entirely in cache. Thus, only actual execution states (using the CPU) and external memory cycles for data transfer are counted with the instruction.

- The **cache disabled case** gives the number of execution states if the cache is disabled when the instruction is executed. In this case, external memory cycles for fetching the instruction word and any extension words are counted with the instruction in addition to states through the CPU and memory states for data transfer. Cache is usually only disabled during debugging.

Cache disabled timing is not necessarily worst case timing. It may sometimes be exceeded when the cache is enabled but the instruction is not in the cache (this is known as a *cache miss*).

13.1.1 Best Case Timing - Considering Hidden States

Best case timing occurs when an instruction is executed entirely in parallel with the end of a previous instruction. According to some microprocessor conventions, many TMS34010 instructions would have a best case timing of 0 states. Since this is unrealistic, the convention used here assigns a finite (nonzero) timing value but allows for instruction overlap by using the concept of *hidden states*.

Hidden states are memory write cycles that occur at the end of a given instruction. Parallelism is achieved when the CPU is executing instructions at the same time the memory controller is writing to memory. The machine states consumed by the instructions that the CPU is executing hide the machine

states consumed by the write cycles. These hidden machine states are not counted against the instruction that incurs them, but are counted against subsequent instructions. If an instruction uses the local bus before all of the hidden cycles have been overlapped by subsequent instructions, that instruction must wait for the hidden cycles to complete. Up to nine machine states may be hidden by write cycles incurred by a single instruction.

In the timing charts in this section and in the **Machine States** portions of the instruction descriptions, hidden states are indicated by parentheses as shown below:

Machine States *cache hit case+(hidden states),cache disabled case*

13.1.2 Other Effects on Instruction Timing

Instruction timing varies, depending on:

- Whether the cache is enabled.
- Whether the instruction and extension words are in cache or not.
- The field size and the word alignment of memory data manipulated by the instruction.

The timing for some instructions (particularly the MOVE, MOVEB, LINE, FILL, and PIXBLT instructions) is affected by the values of implied operands and on the alignment and field sizes of any associated memory accesses.

In addition, several system-dependent factors that are not included in timing values may further influence the instruction timings:

- Wait states on the local memory bus
- Host accesses via the host port
- Display refresh operations
- DRAM refresh operations
- HOLD/HLDA accesses

13.2 MOVE and MOVB Instructions

Timings for MOVE and MOVB instructions are in the following tables:

MOVE and MOVB instructions are field operations, so their timings are affected by factors such as memory address, field size, and field extensions. These factors define the field alignment, which in turn defines the number of memory states required to insert or extract the field from memory. Figure 13-1 illustrates seven cases of alignment, labelled A–G, that are used in the MOVE and MOVB timing tables.

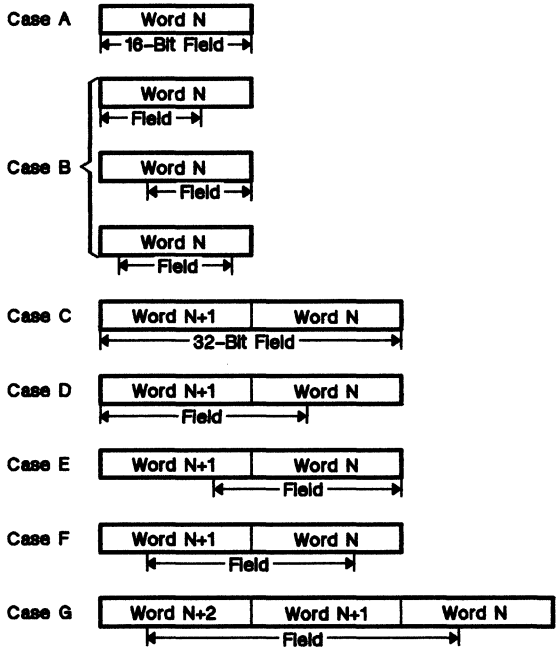


Figure 13-1. Field Alignments in Memory

Case A A 16-bit field is aligned on word boundaries.

Cases B1–B3
The field length is less than 16 bits.

- In **Case B1**, the field starting address is not aligned to a word boundary, although the end of the field coincides with the end of the word.

- In **Case B2**, the field starting address is aligned to a word boundary, but the end of the field does not coincide with the end of the word.
- In **Case B3**, the field length is 14 bits or less, and neither the start nor the end of the field is aligned to a word boundary.

Case C A 32-bit field is aligned on word boundaries.

Case D The field size is greater than 16 bits. The field starting address is not aligned to a word boundary, although the end of the field coincides with the end of a word.

Case E The field size is greater than 16 bits. The field starting address is aligned to a word boundary, but the end of the field does not coincide with the end of a word.

Case F The field straddles the boundary between two words. Neither the start nor the end of the field is aligned to a word boundary.

Case G The field size ranges from 18 to 32 bits, and the field straddles two word boundaries. Neither the start nor the end of the field is aligned to a word boundary.

13.2.1 Moves Between Registers and Memory

Table 13-1 lists the timing for memory-to-register moves for each case of the destination alignment in Figure 13-1. Table 13-2 lists the timing for register-to-memory moves. Note that there are no hidden states for memory-to-register moves.

Table 13-1. MOVE and MOVB Memory-to-Register Timings

Instruction	*Field Alignment Type*		
	A or B	C, D, E, F	G
MOVB **Rs, Rd*	3,6	5,8	-
MOVB **Rs(offset), Rd*	5,11	7,13	-
MOVB *@Address, Rd*	5,14	7,16	-
MOVE **Rs, Rd*	3,6	5,8	7,10
MOVE **Rs+, Rd*	3,6	5,8	7,10
MOVE *-*Rs, Rd*	4,7	6,9	8,11
MOVE **Rs(offset), Rd*	5,11	7,13	9,15
MOVE *@Address, Rd*	5,14	7,16	9,18

Notes: 1. Add 1 state to MOVES for sign extension.
2. The first number specifies the number of cycles required when the entire instruction is contained within cache (cache hit case). The second number specifies the number of cycles required when the cache is disabled (cache disabled case).

Table 13-2. MOVE and MOVB Register-to-Memory Timings

Instruction	*Field Alignment Type*				
	A	B or C	D or E	F	G
MOVB *Rs, *Rd*	-	1+(3),7	-	1+(7),11	-
MOVB *Rs, *Rd(offset)*	-	3+(3),7	-	3+(7),13	-
MOVB *Rs, @Address*	-	1+(3),7	-	3+(7),13	-
MOVE *Rs, *Rd*	1+(1),5	1+(3),7	1+(5),9	1+(7),11	1+(9),13
MOVE *Rs, *Rd+*	1+(1),5	1+(3),7	1+(5),9	1+(7),11	1+(9),13
MOVE *Rs, -*Rd*	2+(1),6	2+(3),8	2+(5),10	2+(7),12	2+(9),14
MOVE *Rs, *Rd(offset)*	3+(1),9	3+(3),12	3+(5),14	3+(7),13	3+(9),18
MOVE *Rs, @Address*	3+(1),13	3+(3),15	3+(5),17	3+(7),19	3+(9),21

Note: The first number specifies the number of cycles required when the entire instruction is contained within cache (cache hit case). The second number specifies the number of cycles required when the cache is disabled (cache disabled case). Hidden states are indicated by parentheses.

13.2.2 Memory-to-Memory Moves

Table 13-4 lists memory-to-memory move timings for each combination of source and destination alignment. Table 13-3 lists numeric indices which are used in Table 13-4. The indices are associated with each source and destination alignment pair (the alignments are shown in Figure 13-1 on page 13-4). To use these tables:

1) Determine the source and destination alignment,
2) Locate the alignment and its index in Table 13-3, **and**
3) Use the index to select the correct column for a particular MOVE addressing mode in Table 13-4.

Table 13-3. Alignment Indices for Memory-to-Memory Moves

Source Field Alignment	*Destination Field Alignment*						
	A	B	C	D	E	F	G
A	1	-	-	-	-	3	-
B	-	2	-	-	-	3	-
C	-	-	6	-	-	-	9
D	-	-	-	7	7	8	9
E	-	-	-	7	7	8	9
F	4	5	-	7	7	8	9
G	-	-	10	11	11	12	13

Table 13-4. MOVE Memory-to-Memory Timings

Instruction	Memory-to-Memory Index - Source to Destination						
	1	2	3	4	5	6	7
MOVB *Rs, *Rd	-	3+(3),7	3+(7),13	-	5+(3),11	-	-
MOVB *Rs(offset), *Rd(offset)	-	5+(3),7	5+(7),21	-	6+(3),13	-	-
MOVB @SAddr, @DAddr	-	7+(3),7	7+(7),29	-	6+(3),12	-	-
MOVE *Rs, *Rd	3+(1),7	3+(3),9	3+(7),13	5+(1),9	5+(3),11	5+(3),11	5+(5),13
MOVE *Rs+, *Rd+	4,7	4+(2),9	4+(6),13	6,9	6+(2),11	6+(2),11	6+(4),13
MOVE -*Rs, -*Rd	4+(1),8	4+(3),10	4+(7),14	6+(1),10	6+(3),12	6+(3),12	6+(5),14
MOVE *Rs(offset), *Rd+	5+(1),12	5+(3),14	5+(7),18	7+(1),14	7+(3),16	7+(3),13	7+(5),15
MOVE *Rs(offset), *Rd(offset)	5+(1),15	5+(3),17	5+(7),21	7+(1),17	7+(3),19	7+(3),16	7+(5),18
MOVE @SAddr, *Rd+	5+(1),15	5+(3),17	5+(7),21	7+(1),17	7+(3),19	7+(3),16	7+(5),18
MOVE @SAddr, @DAddr	7+(1),23	7+(3),25	7+(7),29	9+(1),25	9+(3),27	9+(3),24	9+(5),26

Instruction	Memory-to-Memory Index - Source to Destination					
	8	9	10	11	12	13
MOVB *Rs, *Rd	5+(7),15	-	-	-	-	-
MOVB *Rs(offset), *Rd(offset)	7+(7),19	-	-	-	-	-
MOVB @SAddr, @DAddr	9+(7),27	-	-	-	-	-
MOVE *Rs, *Rd	5+(7),15	5+(9),17	7+(3),13	7+(5),15	5+(7),17	9+(9),21
MOVE *Rs+, *Rd+	6+(6),15	6+(8),17	8+(2),13	8+(4),15	6+(6),17	10+(8),21
MOVE -*Rs, -*Rd	6+(7),15	6+(9),18	8+(3),14	8+(5),16	6+(7),18	10+(9),22
MOVE *Rs(offset), *Rd+	7+(7),16	7+(9),19	9+(3),18	9+(5),20	7+(7),22	11+(9),26
MOVE *Rs(offset), *Rd(offset)	7+(7),19	7+(9),22	9+(3),21	9+(5),23	7+(7),25	11+(9),29
MOVE @SAddr, *Rd+	7+(7),19	7+(9),22	9+(3),21	9+(5),23	7+(7),25	11+(9),29
MOVE @SAddr, @DAddr	9+(7),27	9+(9),30	11+(3),29	11+(5),31	9+(7),33	13+(9),37

Note: The number on the left specifies the number of cycles required when the entire instruction is contained within cache (cache hit case). The number on the right specifies the number of cycles required when the cache is disabled (cache disabled case). Hidden states are indicated by parentheses.

13.2.3 MOVE Timing Example

This example illustrates the timing for the following MOVE instruction:

```
**********************************************
*  Example of a MOVE @SADDR,@DADDR instruction: *
*  Source address = 0E5h                       *
*  Destination address = 161h                  *
*  Size of field 0 = 31 bits (FE0 = don't care) *
**********************************************
        SETF   31, 0          ; Set FS0 field in ST
        MOVE   @0E5h, @161h, 0
```

This example moves 31 bits of data from one memory location to another memory location (a *memory-to-memory* move). We know that the field size is 31 bits because we FS0 to 31 and then used field 0 for the move. To determine the timing for this MOVE instruction, follow these steps:

1) *Determine the field alignment of the source data.*

The 31 bits of source data begin at address 0E5h and span three words. Figure 13-2 below illustrates the alignment of the source data in memory; if you look at Figure 13-1 on page 13-4, you'll see that this is alignment G.

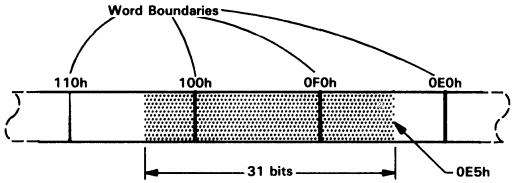


Figure 13-2. Source Data, Alignment G

2) *Determine the field alignment of the destination location.*

The destination location begins at address 161h and spans two words. Figure 13-3 illustrates alignment of the destination location; according to Figure 13-1 (page 13-4), this is alignment E.

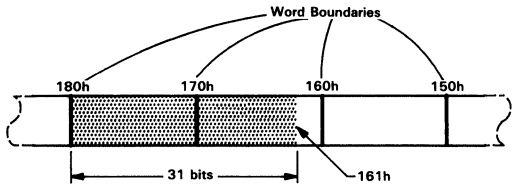


Figure 13-3. Destination Location, Alignment E

3) *Find the alignment index for the combination of the source alignment and the destination alignment.*

   Table 13-3 (page 13-6) shows the source-to-destination alignment indices. The correct index for the combination of source alignment G with destination alignment E is index **11**.

4) *Find the index for this instruction in Table 13-4 (page 13-7).*

   The example instruction, `MOVE @0E5h, @161h`, corresponds to MOVE @SAddr,@Daddr in Table 13-4. Follow this row in the table across to the entry beneath column 11. The timing listed in this entry, **11+(5),31**, is the timing for the example instruction.

Thus, this MOVE example consumes 11 machine states (plus 5 hidden states) if this code resides in cache. If the instruction cache is not enabled, this example consumes 31 machine states. The memory accesses at the end of the MOVE consume 5 machine states, which may be hidden by subsequent cache-resident instructions.

This example is for a *memory-to-memory* move. If you want to determine the timing for a *memory-to-register* or a *register-to-memory* move, use Table 13-2 or Table 13-1.

13.3 FILL Instructions

The total time for the FILL instruction is determined by adding a setup time to a transfer time:

FILL time = FILL setup time + FILL transfer time

- The **setup sequence** executes an initialization sequence, performing any necessary setup operations and translations. (This may include XY-to-linear conversions and window preclipping.) The result of the setup includes the dimensions of the array that is to be moved.

- The **transfer sequence** performs the actual data transfer from the source register to the destination array.

FILL setup and transfer timings are in the following tables:

13.3.1 FILL Setup Time

FILL setup time is the overhead incurred by the FILL instructions from performing initialization, XY conversions, and window operations. Window operations are performed before the FILL transfer begins. Window options that affect FILL setup timing include:

- No window clipping (*W=0*)
- A window clip that requires no change (*array fits*)
- A window clip that affects the starting pointer (*start adjust*)
- A window clip that affects the array transfer dimensions (*dimension adjust*)
- A window clip that affects both the starting and the ending pointers (*adjust both*)
- A window *miss* requesting an interrupt
- A window *hit*

Table 13-5 illustrates the effects of windowing operations on FILL setup timing. Corner adjust operations have no effect on FILL setup timing.

Table 13-5. FILL Setup Time

	Window Operation							*Corner Adjust*		
Instruction	W=0	Array Fits	Start Adjust	Dimens Adjust	Adjust Both	Miss	Hit	PBH=1 PBV=0	PBH=0 PBV=1	PBH=1 PBV=1
FILL L	4	–	–	–	–	–	–	–	–	–
FILL XY	6	9	16	12	20	–	–	–	–	–

Note: These timings are for the cache hit case; add 3 machine states for cache disabled timing.

For example, a FILL XY with preclipping that requires both the starting and ending array corners to be adjusted would consume 20 states of setup time.

13.3.2 FILL Transfer Timing

Table 13-6 lists FILL transfer timings. Transfer timing is the time required (in addition to the setup time) to execute the actual data transfer to memory. Transfer timing is based on several parameters such as the number of rows in the adjusted array (*L*), the number of words affected per row (*N*), graphics operations (*G*), and four possible destination array alignments (A, B, C, and D). These factors are described in the list that follows the table.

Table 13-6. FILL Transfer Timing†

Line Length	*Array Alignments*			
	A	**B**	**C**	**D**
Short (*N*=1)	(1+*G*)*L* + 2	(2+*G*)*L* + 2	(2+*G*)*L* + 1	(2+*G*)*L* + 1
Medium (*N*=2)	(2+2*G*)*L* + 2	(3+2*G*)*L* + 2	(3+2*G*)*L* + 2	(4+2*G*)*L* + 1
Long (*N*≥3)	(1+*NG*)*L* + 2	(2+*NG*)*L* + 5	(3+*NG*)*L* + 2	(4+*NG*)*L* + 1

† Subtract any alignment/graphics adjustment from these values

Key:
L Number of rows (see page 13-11)
N Number of words per row (see page 13-12)
G Value derived from selected graphics operation (see Table 13-7 on page 13-13)

13.3.2.1 Number of Rows in the Adjusted Array (L)

The working dimensions (*L* rows × *M* pixels) for the fill are determined by the originally supplied destination pointer (DADDR) and dimensions (DYDX) in conjunction with window preclipping.

13.3.2.2 Alignment of Leading and Trailing Words in Rows

After clipping, the data transfer portion of the FILL treats the array as a series of *L* rows of *M* pixels. These *M* pixels are spread across *N* words in each row of the destination array. Figure 13-4 illustrates a single row of a destination array in memory. The FILL algorithm resolves rows into three portions:

1) The leading edge at the beginning of the row
2) The center *N*-2 words of the row
3) The trailing edge at the end of the row

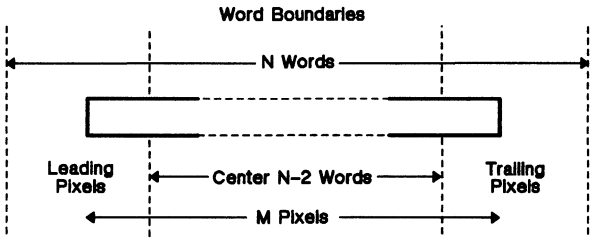


Figure 13-4. Pixel Block Alignment in X

As Figure 13-4 shows, a row of *N* words includes one word each for the leading and trailing parts of the transfer and *N*-2 words for the center portion. The FILL always transfers the center portion of the row as a series of 16-bit words. Thus, the alignment of the leading and trailing words in the row characterize the alignment type of the array. Figure 13-5 illustrates the four possible alignments (A, B, C, and D) of destination array rows within pixel blocks in memory.

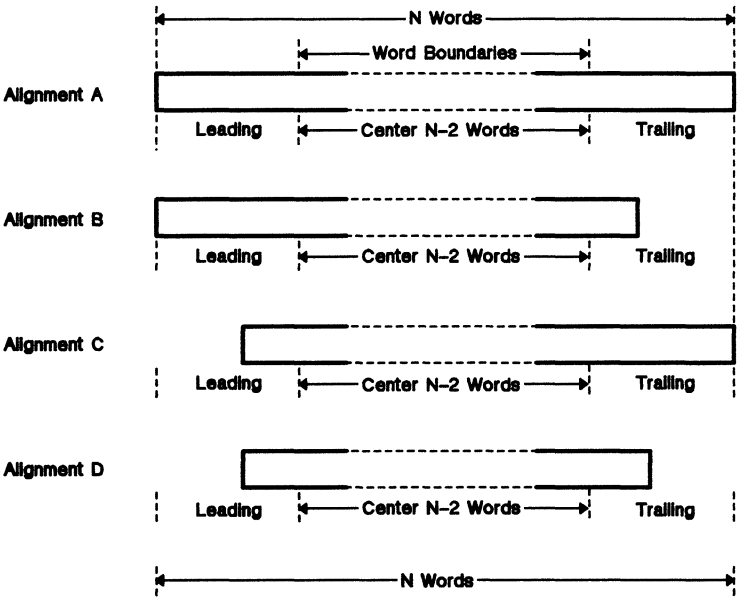


Figure 13-5. Pixel Block Alignments

Word alignment is constant from row to row because DPTCH is constrained to be a multiple of 16 for most FILLs. If a FILL is only one pixel wide, and all the rows are contained in single words in memory, DPTCH may be any value. If DPTCH is not a multiple of 16, word alignment may vary between cases B, C, and D. Average timing for this situation may be derived using alignment C. Worst case timing for this situation may be derived using alignment D.

13.3.2.3 Row Length (Number of Words N per Row)

Row length is determined by a combination of the computed array pointer value in DADDR, the clipped DX dimension, and the pixel size stored in the PSIZE register. The data transfer algorithm breaks down into one of three cases, short, medium, or long, according to the number of words *N* in a row. These three cases include:

- **Short case**. The destination array row occupies only one word in memory (*N*=1). In this case, only one write (or read-modify-write) operation is required to place the row into the destination array. Alignment

for the short case is either type A for exactly aligned arrays or type B, C, or D for nonaligned arrays (which require a read-modify-write).

- **Medium case**. The destination row occupies two words in memory (N=2). In this case, the row has no center portion and the array alignment is determined by the alignments of the first and last words in the row.

- **Long case**. The destination row occupies all or part of at least three words ($N \geq 3$). This is the general case for array alignment discussions.

13.3.2.4 Transfer Direction in X

Transfer direction does not apply to FILLs. FILL transfers proceed a single word of pixels at a time in the order of increasing X and increasing Y. This corresponds to a transfer from left-to-right and top-to-bottom for the default screen orientation.

13.3.2.5 Selected Graphics Operations (G)

Graphics operations such as plane masking, transparency, and pixel processing influence FILL transfer timing because the destination pixels must be read before they are replaced. However, the effects of these operations vary because they are performed by different portions of the TMS34010 hardware. For instance, plane masking, transparency, and field insertion are all performed by the TMS34010 memory controller; any combination of these operations uses 2 machine states for each word written. Pixel processing, on the other hand, is performed by the TMS34010 CPU, and requires 2, 4, 5, or 6 states per word (independently of other operations). *The minimum cycle time for any graphics operation*, then, is **2 machine states** (one memory cycle) using the pixel processing *replace* operation, with plane masking and transparency disabled. Table 13-7 shows these values.

Table 13-7. Timing Values per Word for Graphics Operations (*G*)

	Pixel Processing Operation			
Graphics Operation	**Replace**	**Other Booleans or ADD**	**ADDS,SUB MAX, or MIN**	**SUBS**
No plane masking or transparency	2	4	5	6
Read-modify-write, plane masking, or transparency	4	6	7	8

13.3.2.6 Alignment/Graphics Adjustment

An additional adjustment may be necessary when plane masking or transparency are enabled and the alignment type is B, C, or D. As the second line of Table 13-7 shows, if a particular word in a destination row has already been read as part of a read-modify-write operation, no **additional** states are required to perform *plane masking* or *transparency* for that word. Since the alignment types with misaligned edges (B, C, and D) already assume a RMW (read-modify-write) on their respective edges, the effect of plane masking or transparency can be ignored for these edges. That is, after you have calculated the timing using the proper value for the graphics operation, you can **subtract** 2 states (cases B and C) or 4 states (case D) per row from the transfer timings for the respective alignment cases. Case A requires no adjustment.

13.3.3 FILL Timing Examples

To determine the timing for a FILL instruction, add the FILL setup value to the FILL transfer value and subtract the alignment adjustment:

FILL time = FILL setup time + FILL transfer time - alignment adjustment

FILL setup timings, transfer timings, and the effects of graphics operations are listed in the following tables:

The following three examples illustrate timing for a **FILL XY**. The setup and transfer timings for these examples are the same, except each uses a different graphics operation. Figure 13-7 illustrates the destination array and window used in these examples, as defined by the implied operands in Figure 13-6. Note that the shaded portion is the area of intersection.

```
*************************************************
* Implied operand setup for FILL examples (assume  *
* that the B register names and I/O register names *
* are equated with the proper registers)           *
*************************************************
    MOVI  004400E4h, DADDR    ; X=228, Y=68
    MOVI  800h, DPTCH         ; X extent = 512 pixels
                              ; (at 4 bits per pixel)
    CLR   OFFSET
    MOVI  004900EBh, WSTART   ; X=235, Y=73
    MOVI  005F0140h, WEND     ; X=320, Y=95
    MOVI  0014003Ch, DYDX     ; DX=60, DY=20
    MOVI  4h, A0
    MOVE  A0, @PSIZE          ; Pixel size = 4 bits
    MOVI  14h, A0
    MOVE  A0, @CONVDP         ; (LMO DPTCH)
    MOVI  0Ch, A0
    MOVE  A0, @CONTROL        ; W=3, T=0, PPOP=0
    CLR   A0
    MOVE  A0, @PMASK          ; Disable plane masking
```

Figure 13-6. Implied Operand Setup for FILL Example

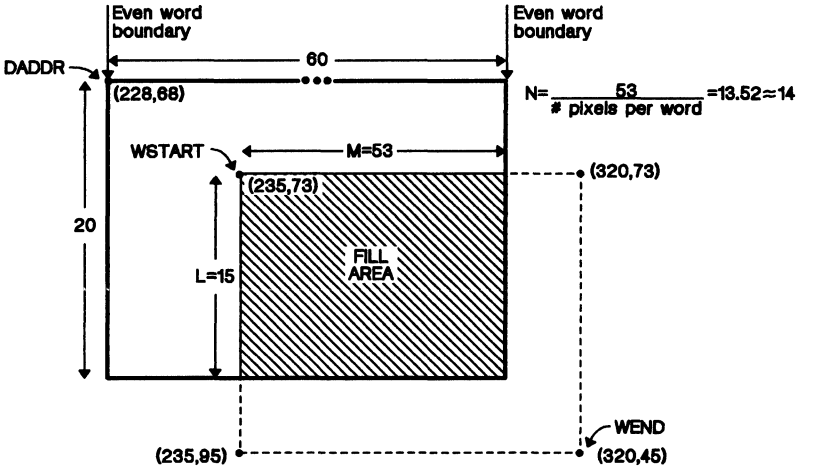


Figure 13-7. FILL XY Timing Example

Follow these steps to determine the number of machine states consumed by this example:

1) Determine the **setup time**; refer to Table 13-5 on page 13-10.

   Setup time comprehends the time required for windowing operations. These examples use window preclipping (W bits=3); this option requires the starting corner to be adjusted. As Table 13-5 (page 13-10) shows, the setup time for a FILL XY with a starting corner adjust is **16** machine states.

2) Determine the **transfer time**; refer to Table 13-6 on page 13-11. Transfer time is affected by the number of words per row, line length, and graphics operations.

   a) *Number of words per row:* As Figure 13-7 shows, adjusting the array by clipping it to the window dimensions produces a new Y dimension, so *L* (the number of rows in the adjusted array) equals 15.

   b) *Line length:* Adjusting the array to fit the window also produces a new X dimension of 53 pixels. The number of pixels divided by the pixel size yields the number of words *N* per row; 53 divided by 4 produces 13.25, so *N*=14. Since *N* is greater than 3, this example conforms to the long case. The trailing edge is word aligned but the leading edge is not, so the alignment type is C.

   As Table 13-6 shows, the transfer time for a FILL XY with these characteristics is $(3+NG)L + 2$. The only variable in the following three examples is *G*, which represents the selected graphics operations.

Example 13-1. Replace, No Transparency, No Plane Masking

The implied operand setup in Figure 13-6 selects the following graphics options:

- Pixel processing *replace* operation (PPOP=0),
- No transparency, **and**
- No plane masking.

According to Table 13-7 (page 13-13), variable ***G*** = **2**. The FILL timing for this instruction is determined as follows:

FILL time = FILL setup time + FILL transfer time
= Adjust pointer + $[3+(N \times G)]L + 2$
= 16 + $[(3 + (14 \times 2)]15 + 2$
= **483 states**

The FILL writes 795 pixels in these 483 states. (The portion of the array lying within the window contains 795 pixels.)

Example 13-2. MAX, No Transparency, No Plane Masking

Select the pixel processing MAX option (be sure to retain the values of the W bits and the T bit, which are also in the CONTROL register):

```
MOVI  50C0h, A0
MOVE  A0, @CONTROL    ; MAX, W=3, T=0
```

These instructions, in combination with the implied operand setup in Figure 13-6, select the following graphics options:

- Pixel processing *MAX* operation (PPOP=14h),
- No transparency, **and**
- No plane masking.

According to Table 13-7, variable ***G*** = **5**. The FILL timing is now calculated as:

FILL time = FILL setup time + FILL transfer time
= Adjust pointer + $[3+(N \times G)]L + 2$
= 16 + $[3 + (5 \times 14)]15 + 2$
= **1,113 states**

This FILL example consumes 1,113 machine states.

Example 13-3. XNOR with Transparency and Plane Masking

Select the pixel processing XNOR operation and enable transparency and plane masking:

```
MOVI   14E0h, A0
MOVE   A0, @CONTROL   ; XNOR, W=3, T=1
MOVI   1111h, A0
MOVE   A0, @PMASK     ; Use a plane mask
```

These instructions, in combination with the implied operand setup in Figure 13-6, select the following graphics options:

- Pixel processing *XNOR* operation (PPOP=05h),
- No transparency, **and**
- No plane masking.

According to Table 13-7, variable G = **6**.

If plane masking or transparency is enabled, you must consider the array alignment in the timing. This example conforms to alignment type C (as shown in Figure 13-5 on page 13-12), which incurs a read-modify-write at the leading edge of each row. The extra read in the RMW can be used by the plane masking or transparency hardware, so an alignment/graphics adjustment is necessary. The adjustment negates the effect of the extra read cycles in each row that are attributed to the graphics operations. For this example, the amount subtracted is $2L$ (the number of machine states for a read cycle times the number of rows). The FILL timing is now calculated as:

FILL time	= FILL setup time	+	FILL transfer time	-	adjustment
	= Adjust pointer	+	$[3+(N \times G)]L + 2$	-	$2L$
	= 16	+	$[3 + (6 \times 14)]15 + 2$	-	(2×15)
	= **1,293 states**				

This FILL example consumes 1,293 machine states.

13.3.4 Interrupt Effects on FILL Timing

A FILL instruction can be interrupted on a word boundary during the transfer portion of the FILL algorithm. It can also be interrupted at the end of each row. The context of the FILL is saved in reserved registers, and the PBX bit is set in the copy of the status register that is pushed onto the stack. The worst case latency caused by an interrupt is 20 machine states for the interrupt to be recognized. The time for the context switch must be added to this. See Section 8.5.1, Interrupt Latency (page 8-6) for context switch information.

13.4 PIXBLT Instructions

PIXBLT instructions covered in this section include:

- PIXBLT L,L
- PIXBLT XY,L
- PIXBLT L,XY
- PIXBLT XY,XY

(PIXBLT B,L and PIXBLT B,XY are discussed in Section 13.5.)

The total PIXBLT instruction timing is obtained by adding a setup time to a transfer time:

PIXBLT time = PIXBLT setup time + PIXBLT transfer time

- The **setup sequence** executes an initialization sequence, performing any necessary setup operations and translations. (This includes XY-to-linear conversion and window preclipping.) The result of the setup includes the dimensions of the source array.

- The **transfer sequence** performs the actual data transfer from the source array to the destination array.

PIXBLT setup and transfer timings are in the following tables:

13.4.1 PIXBLT Setup Time

Table 13-8 lists PIXBLT setup times. Setup time is the overhead incurred by the PIXBLT instructions in performing initialization, XY conversions, window options, and corner adjust. Setup time is affected by both the window and corner adjust operations. The effects of these operations are described in the list that follows Table 13-8.

Table 13-8. PIXBLT Setup Time

	Window Operation							*Corner Adjust*		
Instruction	W=0	Array Fits	Start Adjust	Dimens Adjust	Adjust Both	Miss	Hit	PBH=1 PBV=0	PBH=0 PBV=1	PBH=1 PBV=1
PIXBLT L, L	7	–	–	–	–	–	–	–	–	–
PIXBLT XY, L	9	–	–	–	–	–	–	+1	+2	+4
PIXBLT L, XY	9	12	19	15	23	–	–	+1	+2	+4
PIXBLT XY, XY	12	15	22	18	26	–	–	+1	+2	+4

For example, consider a PIXBLT XY,XY instruction with preclipping that requires both the starting and ending array corners to be adjusted (PBH=1 and PBV=0). The setup timing for this example would be 26+1=27 states.

13.4.1.1 Window Operations

Window operations are performed before the PIXBLT transfer begins. Window options that affect PIXBLT setup timing include:

- No window checking (*W=0*)
- A window clip that requires no change (*array fits*)
- A window clip that affects the starting pointer (*start adjust*)
- A window clip that affects the array transfer dimensions (*dimension adjust*)
- A window clip that affects both the starting and ending pointers (*adjust both*)
- A window *miss* that requests an interrupt
- A window *hit*

13.4.1.2 Corner Adjust (PBH and PBV)

The TMS34010 may need to adjust the starting corner of the source and destination arrays for the PIXBLT L,XY, PIXBLT XY,L, and PIXBLT XY,XY instructions. The default starting corner is the upper left corner of the array. This can be altered by changing the values of the PBH and PBV (PIXBLT horizontal and vertical) bits. Possible corner adjustments (with default origin ORG=0) include:

- No corner adjust (PBH=0, PBV=0)
- Adjust to upper right corner (PBH=1, PBV=0)
- Adjust to lower left corner (PBH=0, PBV=1)
- Adjust to lower right corner (PBH=1, PBV=1)

The TMS34010 adjusts corners before PIXBLT execution begins. For each combination of PBH and PBV, the TMS34010 adjusts the source and destination starting address pointers to point to the appropriate corner of the arrays. This assures that the same pixel block is moved, despite the difference in X and Y transfer directions.

The original source and destination pointers must be supplied through software. The pointers should indicate the least significant pixel in the array, except for PIXBLT L,L. For this instruction, the PBH and PBV bits affect only the *direction* of the move; the TMS34010 does not adjust the starting corner.

13.4.2 PIXBLT Transfer Timing

Table 13-9 lists PIXBLT transfer timings. Transfer timing is the time required (in addition to the setup time) to execute the actual data transfer to memory. Transfer timing is affected by several factors, including the number of rows in the adjusted array (L), the number of words affected per row (N), graphics operations (G), and four possible destination array alignments (A, B, C, and D). These factors are described in the list that follows the table.

Table 13-9. PIXBLT Transfer Timing†

PBH = 0				
Row Lengths and Alignment	***Destination Array Alignment***			
	A	**B**	**C**	**D**
Short (N=1)				
D ≥ S	$(G+4)L + 5$	$(G+6)L + 3$	$(G+6)L + 3$	$(G+6)L + 3$
D < S	$(G+4)L + 5$	$(G+6)L + 3$	$(G+6)L + 3$	$(G+6)L + 3$
Medium (N=2)				
D ≥ S	$[2+(4+2G)]L + 5$	$[4+(4+2G)]L + 3$	$[4+(4+2G)]L + 5$	$[6+(4+2G)]L + 3$
D < S	$[4+(4+2G)]L + 4$	$[6+(4+2G)]L + 2$	$[6+(4+2G)]L + 4$	$[8+(4+2G)]L + 2$
Long (N≥3)				
D ≥ S	$[(2+G)N]L + 5$	$[2+(2+G)N]L + 3$	$[2+(2+G)N]L + 5$	$[2+(4+G)N]L + 3$
D < S	$[2+(2G)N]L + 4$	$[4+(2G)N]L + 2$	$[4+(2G)N]L + 4$	$[6+(2G)N]L + 2$
PBH = 1				
Row Lengths and Alignment	***Destination Array Alignment***			
	A	**B**	**C**	**D**
Short (N=1)				
D ≥ S	$(G+3)L + 8$	$(G+4)L + 7$	$(G+4)L + 7$	$(G+4)L + 7$
D < S	$(G+3)L + 8$	$(G+4)L + 7$	$(G+4)L + 7$	$(G+4)L + 7$
Medium (N=2)				
D ≥ S	$[2+(4+2G)]L + 4$	$[4+(4+2G)]L + 3$	$[4+(4+2G)]L + 4$	$[6+(4+2G)]L + 3$
D < S	$[4+(4+2G)]L + 5$	$[5+(4+2G)]L + 4$	$[6+(4+2G)]L + 5$	$[7+(4+2G)]L + 4$
Long (N≥3)				
D ≥ S	$[1+(2+G)N]L + 4$	$[3+(2+G)N]L + 3$	$[3+(2+G)N]L + 4$	$[5+(2+G)N]L + 3$
D < S	$[3+(2+G)N]L + 5$	$[4+(2+G)N]L + 4$	$[5+(2+G)N]L + 5$	$[6+(2+G)N]L + 4$

† Subtract any alignment/graphics adjustment from these values

Key:

L Number of rows in the array (see page 13-20)
N Number of destination words per row (see page 13-22)
G Value dependent on selected graphics operation (see Table 13-10 on page 13-24)
D≥S First destination to source alignment case (see page 13-22)
D<S Second destination to source alignment case (see page 13-22)

13.4.2.1 Number of Rows in the Array (L)

The working dimensions (*L* rows by *N* words) for the block transfer are determined by the original destination pointer (DADDR) and dimensions (DYDX) in conjunction with window preclipping. *L* represents the number of rows in the clipped array.

13.4.2.2 Alignment of Leading and Trailing Words in Rows

After clipping, the data transfer portion of the PIXBLT treats the array as a series of *L* rows of *M* pixels. These *M* pixels are spread across *N* words in each row of the destination array. *N* and *L* affect the transfer timing. Alignment does not vary from row to row because DPTCH is constrained to be a power of two.

Figure 13-8 illustrates a single row of a destination array in memory. The PIXBLT algorithm resolves rows into three portions:

1) The leading edge at the beginning of a row
2) The center *N*-2 words of the row
3) The trailing edge at the end of the row

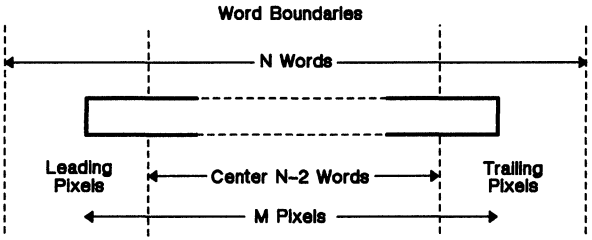


Figure 13-8. Pixel Block Alignment in X

As Figure 13-8 shows, a row of *N* words includes one word each for the leading and trailing parts of the transfer and *N*-2 words for the center portion. The PIXBLT always transfers the center portion of the row as a series of 16-bit words. Thus, the alignment of the leading and trailing portions characterize the alignment type of the array. Figure 13-9 illustrates the four possible alignments (A, B, C, and D) of a destination array.

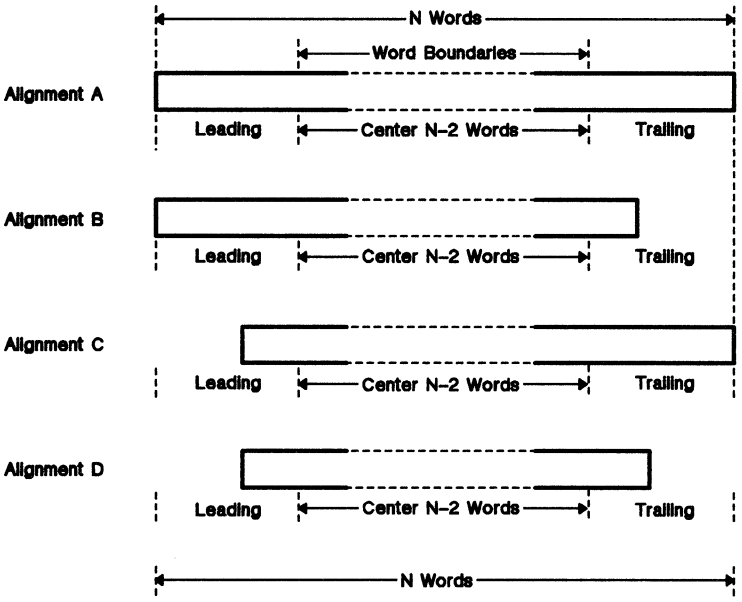


Figure 13-9. Pixel Block Alignments

13.4.2.3 Row Length (Number of Words N per Row)

Row length is determined by a combination of the computed array pointer value in DADDR, the clipped DX dimension, and the pixel size stored in the PSIZE register. The data transfer algorithm breaks down into one of three cases, short, medium, or long, according to the number of words *N* in a row. These three cases include:

- **Short case**. The destination array row occupies only one word in memory ($N=1$). In this case, only one write (or read-modify-write) operation is required to place the row into the destination array. Alignment for the short case is either type A for exactly aligned arrays or type B, C, or D for nonaligned arrays (which require a read-modify-write).

- **Medium case**. The destination row occupies two words in memory ($N=2$). In this case, there is no center portion to the row and the array alignment is determined by the alignments of the first and last words in the row.

- **Long case**. The destination row occupies all or part of at least three words ($N\geq 3$). This is the general case for array alignment discussions.

13.4.2.4 Relative Alignment of Source Rows to Destination Rows

The alignment of the leading pixels in a source row with respect to a destination row influences PIXBLT transfer timing. This alignment determines whether one or two words are required from the source array to fully write the first word of the destination array. This initial condition can be divided into two cases:

D≥S The four LSBs of the destination address are greater than the four LSBs of the source address. This implies that the amount of data available from the first word of the source array exceeds the amount needed to write to the first word of the destination array. The write to the destination array can proceed immediately.

D<S The four LSBs of the destination address are less than the four LSBs of the source address. This implies that the amount of data to be written to the first word of the destination array exceeds the amount available from the first word of the source array. Another word must be read from the source array.

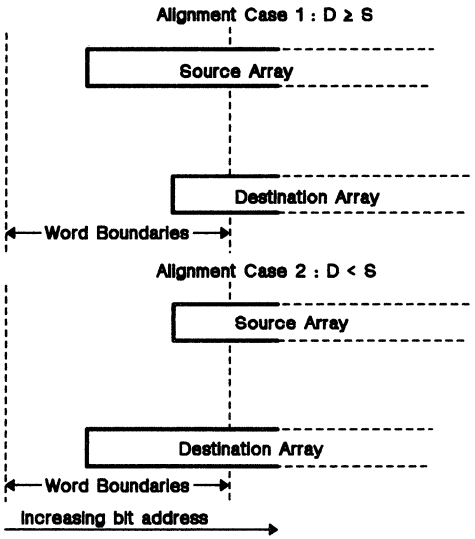


Figure 13-10. Source to Destination Alignments

13.4.2.5 Transfer Direction in X (PBH)

PIXBLT transfers proceed a word of data at a time in a consistent direction in X and Y. The default direction is from the smallest word address to the largest, corresponding to left-to-right and top-to-bottom for the default screen orientation. The values of the PBH and PBV bits determine the transfer direction in X and Y.

For the four regular PIXBLTs (without expand), PBH determines the order in which ***words*** are written on each row of the destination array:

PBH=0: Words within rows are written in the order of increasing addresses.

PBH=1: Words are written in the order of decreasing addresses. The value of PBH influences the per-row transfer timings of these PIXBLTs.

The sense of the PBV bit determines the order in which ***rows*** are transferred to the destination array.

PBV=0: Rows are transferred in the order of increasing addresses.

PBV=1: Rows are transferred in the order of decreasing addresses.

This value affects the setup timing, but not the transfer timing.

13.4.2.6 Selected Graphics Operations (G)

Graphics operations such as plane masking, transparency, and pixel processing influence PIXBLT transfer timing because the destination pixels must be read before they are replaced. However, the effects of these operations vary because they are performed by different portions of the TMS34010 hardware. For instance, plane masking, transparency, and field insertion are all performed by the TMS34010 memory controller hardware; any combination of these operations uses 2 machine states for each word written. Pixel processing, on the other hand, is performed by the TMS34010 CPU, and requires 2, 4, 5, or 6 states per word independent, of other operations. *The minimum time for any graphics operation*, then, is **2 machine states**(one memory cycle) using the *replace* operation with plane masking and transparency disabled. These values are shown in Table 13-10.

Table 13-10. Timing Values per Word for Graphics Operations (*G*)

	Pixel Processing Operation			
Graphics Operation	**Replace**	**Other Booleans or ADD**	**ADDS,SUB MAX or MIN**	**SUBS**
No plane masking or transparency	2	4	5	6
Read-modify-write, plane masking, or transparency	4	6	7	8

13.4.2.7 Alignment/Graphics Adjustment

An additional adjustment may be necessary when plane masking or transparency are enabled and the alignment type is B, C, or D. As the second line of Table 13-10 shows, if a particular word in a destination row has already been read as part of a read-modify-write operation, no **additional** states are required to perform plane masking or transparency for that word. Since the alignment types with misaligned edges (B, C, and D) already assume a RMW (read-modify-write) on their respective edges, the effect of plane masking or transparency can be ignored for these edges. That is, after you have computed the timing using the proper value for the graphics operation, you can **subtract** 2 states (case B and C) or 4 states (case D) per row from the row timings for the respective alignment cases. Case A requires no adjustment.

13.4.3 PIXBLT Timing Examples

To determine PIXBLT timing, add the PIXBLT setup value to the PIXBLT transfer value and subtract the alignment adjustment:

PIXBLT time = PIXBLT setup time + PIXBLT transfer time - alignment adjustment

PIXBLT setup timings, transfer timings, and the effects of graphics operations are in the following tables:

The following three examples illustrate timing for a **PIXBLT XY,L**. The setup and transfer timings for these examples are the same, except each uses a different graphics operation. Figure 13-12 illustrates the destination array and window used in these examples, as defined by the implied operands in Figure 13-11. The shaded portion of Figure 13-12 is the destination array.

```
*************************************************
* Implied operand setup for PIXBLT XY, L examples *
* (assume that the B register and I/O register    *
* names are equated with the proper registers)    *
*************************************************
       MOVI  003A00E6h, SADDR    ; X=230, Y= 58
       MOVI  800h, SPTCH         ; X extent = 512 pixels
                                 ; (at 4 bits per pixel)
       MOVI  000030E8, DADDR     ; linear address
       MOVI  800h, DPTCH         ; X extent = 512 pixels
       MOVI  00040000, OFFSET
       CLR   WSTART              ; ignored
       CLR   WEND                ; ignored
       MOVI  000F0036, DYDX      ; DY=15, DX=54
       MOVI  4h, A0
       MOVE  A0, @PSIZE          ; Pixel size = 4
       MOVI  14h, A0
       MOVE  A0, CONVSP
       MOVE  A0, CONVDP          ; ignored
       CLR   A0
       MOVI  A0, PMASK           ; Disable plane masking
       MOVI  0300h, A0
       MOVE  A0, @CONTROL        ; PBH=1, PBV=1
```

Figure 13-11. Implied Operand Setup for PIXBLT Timing Examples

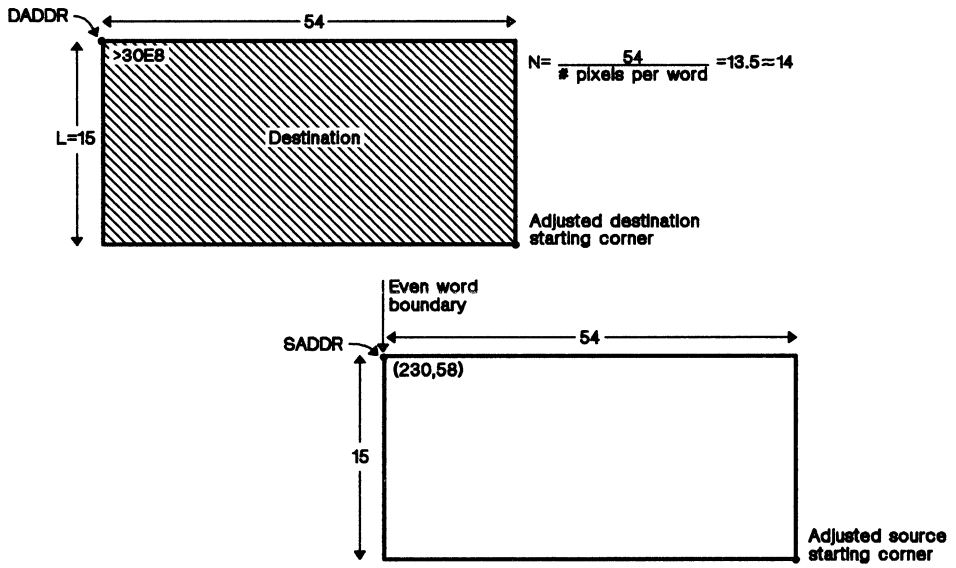


Figure 13-12. PIXBLT XY,L Timing Example

To calculate the number of machine steps consumed by these PIXBLT examples, follow these steps:

1) Determine the **setup time**; refer to Table 13-8 (page 13-18). Setup timing comprehends windowing and corner-adjust operations.

   a) *Windowing*: Windowing is not enabled for this example (W=0, **9** states).
   b) *Corner adjust*: PBH=1 and PBV=1, so the starting corner must be adjusted in both the X and Y dimensions (+**4** states).

   As Table 13-8 shows, the setup time for a PIXBLT XY,L with these operations is 9 + 4 machine states.

2) Determine the **transfer time**; refer to Table 13-9 (page 13-20). Transfer time comprehends the direction of the move, array and row alignments, and line lengths.

   - PBH=1.
   - *Number of rows in the array*: The Y dimension is 15, so *L*=15.
   - *Number of words in a row*: The X dimension is 54 pixels, and the pixel size is four; 54 divided by 4 produces 13.5, so the number of words per row, *N*, = 14.
   - *Row length and alignment*: *N* is greater than 3, so this example conforms to the long case.

- The four LSBs of DADDR are greater than the four LSBs of SADDR (D≥S).
- *Destination array alignment*: The trailing edge is word aligned but the leading edge is not, so the alignment type is C.

As Table 13-9 shows, the transfer time for this PIXBLT instruction is **[5+(2+*G*)*N*]*L* + 4**. The only variable in the following three examples is *G*, which represents the selected graphics operations.

Example 13-4. Replace, No Transparency, No Plane Masking

The implied operand setup in Figure 13-11 selects the following graphics options:

- Pixel processing *replace* operation (PPOP=0),
- No transparency, **and**
- No plane masking.

According to Table 13-10 on page 13-24, variable ***G* = 2**. The total machine states required for this instruction are:

PIXBLT time	= PIXBLT setup time	+	PIXBLT transfer time
	= 9 + 4	+	[5+(2+*G*)*N*]*L* + 4
	= 13	+	(5 + 4×14)×15 + 4
	= **932 states**		

The instruction in this example consumes 932 machine states.

Example 13-5. MAX Option, No Transparency, No Plane Masking

Select the pixel processing MAX option (be sure to retain the values of the W bits and the T bit, which are also in the CONTROL register):

```
MOVI   50C0h, A0
MOVE   A0, @CONTROL    ; MAX, W=3, T=0
```

These instructions, in combination with the implied operand setup in Figure 13-11, select the following graphics options:

- Pixel processing *MAX* operation (PPOP=14h),
- No transparency, **and**
- No plane masking.

According to Table 13-10, variable ***G* = 5**. Thus, the timing equation becomes:

PIXBLT time	= PIXBLT setup time	+	PIXBLT transfer time
	= 9 + 4	+	$[5+(2+G)N]L + 4$
	= 13	+	$(5 + 7 \times 14) \times 15 + 4$
	= **1562 states**		

The instruction in this example consumes 1,562 machine states.

Example 13-6. XNOR with Transparency and Plane Masking

Select the pixel processing XNOR operation and enable transparency and plane masking:

```
MOVI  14E0h, A0
MOVE  A0, @CONTROL   ; XNOR, W=3, T=1
MOVI  1111h, A0
MOVE  A0, @PMASK     ; Use a plane mask
```

These instructions, in combination with the implied operand setup in Figure 13-11, select the following graphics options:

- Pixel processing *XNOR* operation (PPOP=05h),
- No transparency, **and**
- No plane masking.

According to Table 13-10, variable ***G*** = **6**.

If plane masking or transparency is enabled, you must consider the array alignment in the timing. Alignment type C incurs a read-modify-write at the leading edge of each row. The extra read included in the RMW can be used by the plane masking or transparency hardware, so an alignment/graphics adjustment is necessary. The adjustment negates the effect of the extra read cycles in each row that are attributed to the graphics operations. For this example, the amount subtracted is $2L$ (the number of machine states for a read cycle times the number of rows). The timing is now calculated as:

PIXBLT time = PIXBLT setup time + PIXBLT transfer time - adjustment
= 9 + 4 + $[5+(2+G)N]L$ + 4 - $2L$
= 13 + $(5 + 8\times14)\times15$ + 4 - (2×15)
= **1772 states**

The instruction in this example consumes 1,772 machine states.

13.4.4 The Effect of Interrupts on PIXBLT Instructions

The PIXBLT instruction may be interrupted on a destination word boundary during the transfer portion of the algorithm. It may also be interrupted at the end of any row in the array. The context of the PIXBLT is saved in reserved registers. The PBX bit is set in the copy of the ST register that is pushed to the stack. The worst case latency caused by an interrupt is 20 machine states for the interrupt to be recognized. The time for the context switch must be added to this; see Section 8.5.1, Interrupt Latency (page 8-6) for context switch timing.

13.5 PIXBLT Expand Instructions

PIXBLT expand instructions include:

- PIXBLT B,L
- PIXBLT B,XY

To determine PIXBLT expand instruction timing, add a setup time to a transfer time:

PIXBLT time = PIXBLT setup time + PIXBLT transfer time

- The **setup sequence** executes an initialization sequence, performing any necessary setup operations and translations. (This includes XY-to-linear conversion and window preclipping.) The result of the setup includes the dimensions of the source array.

- The **transfer sequence** performs the actual data transfer from the source array to the destination array.

PIXBLT setup and transfer timings are in the following tables:

13.5.1 PIXBLT Setup Time

PIXBLT setup time is the overhead incurred by the PIXBLT instructions from performing initialization, XY conversions, and window operations.

Window operations are performed before the PIXBLT transfer begins. Window options that affect PIXBLT setup timing include:

- No window checking (*W=0*)
- A window clip that requires no change (*array fits*)
- A window clip that affects the starting pointer (*adjust start*)
- A window clip that affects the array transfer dimensions (*dimension adjust*)
- A window clip that affects both the starting and ending pointers (*adjust both*)
- A window *miss* that requests an interrupt
- A window *hit*

Table 13-11 shows the effect of these options on the PIXBLT setup time. Corner adjust operations have no effect on PIXBLT setup timing.

Table 13-11. PIXBLT Expand Setup Time

	Window Operation							*Corner Adjust*		
Instruction	W=0	Array Fits	Start Adjust	Dimens Adjust	Adjust Both	Miss	Hit	PBH=1 PBV=0	PBH=0 PBV=1	PBH=1 PBV=1
PIXBLT B, L	4	–	–	–	–	–	–	–	–	–
PIXBLT B, XY	6	9	17	12	21	–	–	–	–	–

For example, a PIXBLT B,XY with the preclipping option requiring an adjustment to the end corner of the array requires 12 states of setup time.

13.5.2 PIXBLT Transfer Timing

Table 13-12 shows transfer timing for PIXBLT expand instructions. Transfer timing is the time required (in addition to the setup time) to execute the actual data transfer to memory. Transfer timing is affected by several factors, including the number of rows in the adjusted array (*L*), the number of words affected per row (*N*), graphics operations (*G*), the four possible destination array alignments (A, B, C, and D), and the arrangement of words in source rows. These factors are described in the list that follows the table.

Table 13-12. PIXBLT Expand Transfer Timing†

Destination Alignment	Transfer Timing
Short case	$(3+2R+G)L + 3$
Medium case	
Alignment A or C	$(3+2R+NG)L + 3$
Alignment B or D	$(5+2R+NG)L + 3$
Long case	
Alignment A	$[(3+2R+2GP)S + 2V + NG]L + 3$
Alignment D	$[(8+2R+2GP)S + 2V + YG + 8]L + 3$

† Subtract any alignment/graphics adjustment from these values

Key:

L Number of rows in the array (below)
N Number of destination words per row (see page 13-33)
R Number of source words involved in set (see page 13-33)
S Number of 32-bit sets in long source rows (DX/32), except for the case of an even number of sets; in this case, *S* is the number of 32-bit sets minus 1 (DX/32 - 1) (see page 13-35)
V Number of source words involved in reading source pixels at end of row after all the complete 32-bit sets have been transferred *P* Current pixel size
G Value dependent on selected graphics operations (see Table 13-13)
Y Number of remaining destination words affected in a given row after *S* 32-bit sets are written

13.5.2.1 Number of Rows in the Array (L)

The working dimensions (*L* rows × *N* words) for the block transfer are determined by the original destination pointer (DADDR) and dimensions (DYDX) in conjunction with window preclipping. The symbol *L* is used to represent the number of rows in the clipped destination array.

13.5.2.2 Alignment of Leading and Trailing words in Rows

After clipping, the data transfer portion of the PIXBLT treats the array as a series of *L* rows of *M* pixels. These *R* pixels are spread across *N* words in each row of the destination array. *N* and *L* affect the transfer timing. This alignment does not vary from row to row because DPTCH is constrained to be a multiple of 16 for binary PIXBLTs.

Figure 13-13 illustrates a single row of a destination array in memory. The PIXBLT algorithm resolves rows into three portions:

1) The leading edge at the beginning of the row
2) The center *N*-2 words of the row
3) The trailing edge at the end of the row

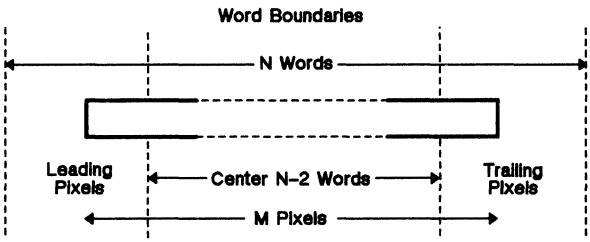


Figure 13-13. Pixel Block Alignment in X

As Figure 13-13 shows, a row of *N* words includes one word each for the leading and trailing parts of the transfer and *N*-2 words for the center portion. PIXBLT expand instructions always transfer the center portion of the row as a series of 16 bit words, and are not affected by the alignment of the leading word. Thus, the alignment of the trailing words in the row characterize the alignment type for the row. Figure 13-14 illustrates the four possible alignments (A, B, C, and D) of a row in the destination array.

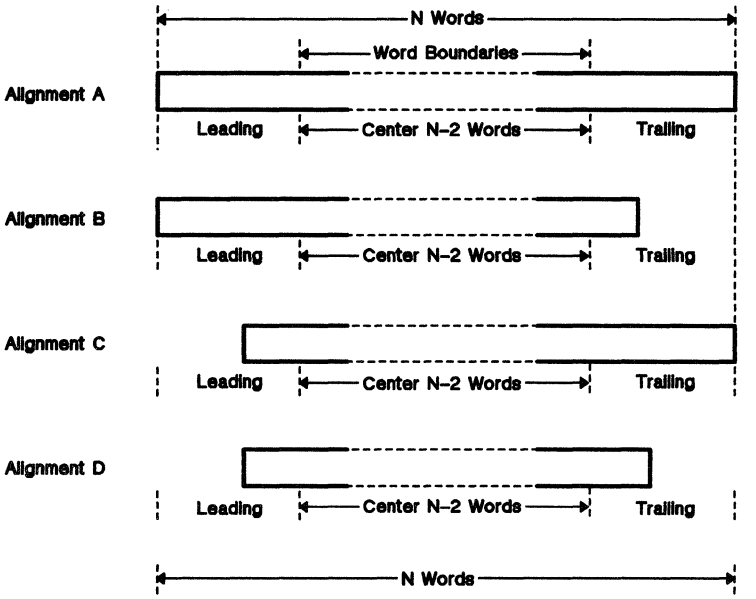


Figure 13-14. Pixel Block Row Alignments

13.5.2.3 Row Length (N Words per Row)

Row length is determined by a combination of the computed array pointer value in DADDR, the clipped DX dimension, and the pixel size stored in the PSIZE register. The data transfer algorithm breaks down into one of three cases, short, medium, or long, according to the number of words *N* in a row. These three cases include:

Short case. A row of source array pixels is contained in 16 bits or less and the expanded data involves only one word of the destination array per row ($N=1$). Alignment does not affect the short case.

Medium case. A row of source array pixels is contained in 32 bits or less but the expanded data involves more than one word of the destination array per row ($N>1$). In this case, the array alignment is determined by the alignments of the last word in the row. Thus, alignments A,C and B,D have equal transfer timings.

Long case. A row of source array pixels is contained in more than 32 bits. The expanded data involves multiple words in the destination array row. In this case, the array alignment is determined by the alignments of the last word in the row. Thus, alignments A and B and alignments C and D have equal transfer timings.

Note that the timings for the short and medium row lengths are not affected by the alignment of the first word on each row of the destination array. That is, the destination array row transfer can start with either a write or a read-modify-write. The long case is treated as a series of 32-pixel medium cases followed by a short case (if necessary) at the end of each row. Each 32-pixel set is expanded and written to the destination in a serial fashion, without optimizing for beginning and ending alignments. Thus, the timing for the long case becomes a product of the number of 32-pixel sets (*S*) and the timing for each set, plus the timing for expanding any remaining segment of the source array (less than or equal to 32 bits) that is left in the row. Note that the remaining segment of the source array may have an alignment type (B or C) that is different from the preceding 32-bit sets.

13.5.2.4 Arrangement of Source Rows

As discussed in the *Row Length* section, the number of bits in a row of the source array affects the time required to perform the PIXBLT transfer algorithm. The short and medium cases have explicit timings based on the number of words read from the source row, *R*. Note that the timings for the short and medium row lengths are not affected by the alignment of the last word on each row of the destination array. That is, the destination array row transfer can either end with a write or a read-modify-write.

The long case is treated as a series of 32-pixel segments. Each 32-pixel set is expanded and written to the destination in a serial fashion without optimizing for beginning and ending alignments for the source or destination. The final portion of the transfer may be up to a 32-pixel "partial" segment. Thus, the timing for the long case becomes a product of the timing for each set and the number of 32-pixel sets (*S*), plus the timing for expanding any remaining segment of the source array (up to 32 bits). Note that the alignment of the remaining segment of the source array is determined by the original (clipped) source array alignment.

The PIXBLT does not attempt to optimize read operations from the source array; therefore, depending on the alignment of the source array, either two or three words may need to be read in order to obtain a 32-bit set of source pixels for expansion. This value, *R*, is the number of source words involved in a 32-bit set of source pixels and may be either two or three. The timings are affected by *R* as wells as the number of such complete 32-bit sets *S* in a source row.

The bits remaining after all of the complete 32-bit sets on a row have been moved are transferred. Depending on the number of remaining bits and the alignment of the source array, either one, two, or three words may need to be read in order to obtain the remaining set of source pixels for expansion. This value, *V*, is the number of of source words read to obtain the final segment while *Y* is is the number of destination words involved for this fragment.

13.5.2.5 Transfer Direction in X (PBH Bit)

These PIXBLT instructions proceed a single word of pixels at a time in the direction of increasing X and increasing Y. This corresponds to left-to-right and top-to-bottom for the default screen orientation. Setting the PBH and PBV bits has no effect.

13.5.2.6 Selected Graphics Options (G)

Graphics operations such as plane masking, transparency, and pixel processing influence PIXBLT transfer timing because the destination pixels must be read before they are replaced. However, the effects of these operations are performed by different parts of the TMS34010 hardware. For instance, plane masking, transparency, and field insertion are all performed by the TMS34010 memory controller hardware; any combination of these operations uses 2 machine states for each word written. Pixel processing, on the other hand, is performed by the TMS34010 CPU, and requires 2, 4, 5, or 6 states per word independent of other operations. *The minimum time for any graphics operation*, then, is **2 machine states** (one memory cycle) using the replace operation with plane masking and transparency disabled. These values are shown in Table 13-13.

Table 13-13. Timing Values per Word for Graphics Operations (*G*)

	Pixel Processing Operation			
Graphics Operation	**Replace**	**Other Booleans or ADD**	**ADDS,SUB MAX or MIN**	**SUBS**
No plane masking or transparency	2	4	5	6
Read-modify-write, plane masking, or transparency	4	6	7	8

13.5.2.7 Alignment/Graphics Adjustment

An additional adjustment may be necessary when plane masking or transparency are enabled and the alignment type is B, C, or D. As the second line of Table 13-13 shows, if a particular word in a destination row has already been read as part of a read-modify-write operation, no **additional** states are required to perform plane masking or transparency for that word. Since the alignment types with misaligned edges (B, C, and D) already assume a RMW (read-modify-write) on their respective edges, the effect of plane masking or transparency can be ignored for these edges. That is, after you have calculated the timing using the proper value for the graphics operation, you can **subtract** 2 states (cases B and C) or 4 states (case D) per row from the row timings for the respective alignment cases. Case A requires no adjustment.

13.5.3 PIXBLT Timing Examples

PIXBLT timing is calculated by adding the PIXBLT setup value to the PIXBLT transfer value:

PIXBLT time = PIXBLT setup time + PIXBLT transfer time - alignment adjustment

PIXBLT setup timings, transfer timings, and the effects of graphics operations are listed in the following tables:

The following three examples illustrate timing for a **PIXBLT B,XY** that expands a 10-by-10 font (*L*=10) into eight bits per pixel with color. The setup and transfer timings for these examples are the same, except each uses a different graphics operation. Figure 13-16 illustrates the destination array and window used in these examples, as defined by the implied operands in Figure 13-15. The shaded portion in Figure 13-16 is the destination array.

```
*************************************************
* Implied operand setup for PIXBLT B, XY examples *
* (assume that B register and I/O regiser names   *
* are equated to with proper registers)           *
*************************************************
       MOVI  0003E2E8h, SADDR    ; linear address
       MOVI  00AD0h, SPTCH       ; X extent = 2768 pixels
       MOVI  0032010Bh, DADDR    ; X=267, Y= 50
       MOVI  800h, DPTCH         ; X extent = 512 pixels
       MOVI  00040000h, OFFSET
       CLR   WSTART              ; ignored
       MOVI  01000100h, WEND     ; ignored
       MOVI  000A000Ah, DYDX     ; DX=10, DY=10
       MOVI  8h, A0
       MOVE  A0, @PSIZE          ; Pixel size = 8 bits
       MOVI  14h, A0
       MOVE  A0, @CONVDP
       MOVE  A0, @CONVSP         ; ignored
       CLR   A0
       MOVE  A0, @PMASK
       MOVI  0300h, A0
       MOVE  A0, @CONTROL        ; W=0,T=0,PP=0,PBH=1,PBV=1
```

Figure 13-15. Implied Operand Setup for PIXBLT-Expand Examples

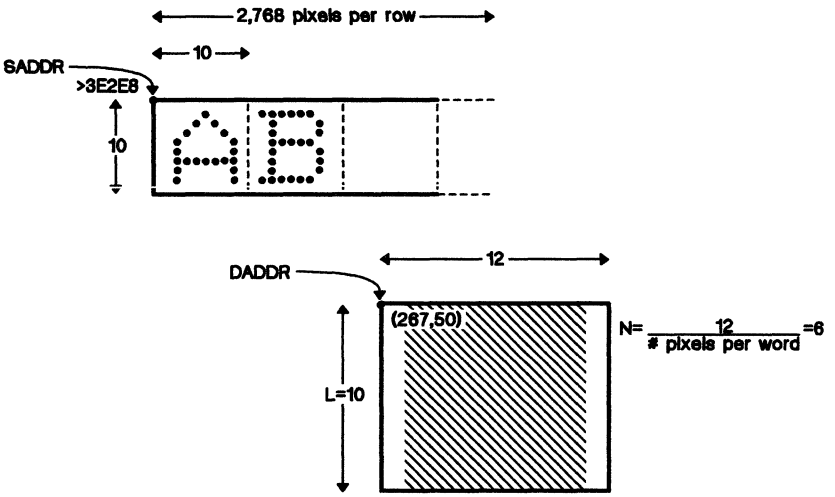


Figure 13-16. PIXBLT B,XY Timing Example

Follow these steps to determine the number of machine states consumed in these PIXBLT examples:

1) Determine the **setup time**; refer to Table 13-11 (page 13-32). Setup time comprehends the machine states consumed by windowing and corner adjust operations;

   a) *Windowing*: Is not enabled for this example.
   b) *Corner adjust*: PBH and PBV are ignored.

   As Table 13-11 shows, the setup time for this PIXBLT is **6** machine states.

2) Determine the **transfer time**; refer to Table 13-12 (page 13-32). Transfer time comprehends the number and alignment of rows in the array, row length, the direction of the move, and the graphics operations.

   a) *Number of words per row*: The source is part of a packed font. The source array starts in the middle of a word and extends into the next word, so two words are read for each row of the font (R=2).
   b) *Number of rows in the array*: The Y dimension is 10 (L=10).
   c) Neither the leading nor the trailing edges are word aligned, so the alignment type is D.
   d) *Array alignment*: The X dimension is 10 pixels wide, but with alignment type D, an extra word is involved for both the leading and trailing pixels; the pixel size is eight, so 12 divided by 2 (two pixels per word) produces N=6. Since the width is less than 32 pixels (10), but more than one word of the destination is affected, this example is a medium case.

As Table 13-12 shows, the transfer timing is $(5+2R+2GN)L + 3$. The only variable in the timing for these three examples is the selected graphics operations.

Example 13-7. Replace, No Transparency, No Plane Masking

The implied operand setup in Figure 13-15 selects the following graphics options:

- Pixel processing *replace* operation (PPOP=0),
- No transparency, **and**
- No plane masking.

According to Table 13-13, variable ***G* = 2**. The total machine states required for this instruction are:

PIXBLT time	= PIXBLT setup time	+	PIXBLT transfer time
	= 6	+	$(5+2R+NG)L + 3$
	= 6	+	$(5 + 2\times2 + 6\times2)\times10 + 3$
	= **219 states**		

This examples consumes 219 machine states as it reads, expands, and writes these 100 pixels.

Example 13-8. MAX, No Transparency, No Plane Masking

Select the pixel processing MAX option (be sure to retain the values of the W bits and the T bit, which are also in the CONTROL register):

```
MOVI  50C0h, A0
MOVE  A0, @CONTROL    ; MAX, W=3, T=0
```

These instructions, in combination with the implied operand setup in Figure 13-15, select the following graphics options:

- Pixel processing *MAX* operation (PPOP=14h),
- No transparency, **and**
- No plane masking.

According to Table 13-13, variable ***G*=5**; the timing equation becomes:

PIXBLT time	= PIXBLT setup time	+	PIXBLT transfer time
	= 6	+	$(5+2R+NG)L + 3$
	= 6	+	$(5 + 2\times2 + 6\times5)\times10 + 3$
	= **399 states**		

The instruction in this example consumes 399 machine states.

Example 13-9. XNOR with Transparency and Plane Masking

Select the pixel processing XNOR operation and enable transparency and plane masking:

```
MOVI   14E0h, A0
MOVE   A0, @CONTROL   ; XNOR, W=3, T=1
MOVI   1111h, A0
MOVE   A0, @PMASK     ; Use a plane mask
```

These instructions, in combination with the implied operand setup in Figure 13-15, select the following graphics options:

- Pixel processing *XNOR* operation (PPOP=05h),
- No transparency, **and**
- No plane masking.

According to Table 13-13, variable ***G*** = **6**.

If plane masking or transparency is enabled, you must consider the array alignment in the timing. Alignment type D incurs a read-modify-write at the leading and trailing edges of each row. The extra read included in the RMW can be used by the plane masking or transparency hardware, so an alignment/graphics adjustment is necessary. The adjustment negates the effect of the extra read cycles in each row that are attributed to the graphics operations. For this example, the amount subtracted is $4L$ (the number of machine states for a read cycle times 2 times the number of rows). The timing is now calculated as:

PIXBLT time = PIXBLT setup time + PIXBLT transfer time - adjustment

$= 6 + (5+2R+NG)L + 3 - 4L$

$= 6 + (5 + 2\times2 + 6\times6)\times10+3 - (4\times10)$

$=$ **419 states**

The instruction in this example consumes 419 machine states.

13.5.4 The Effect of Interrupts

The PIXBLT instruction may be interrupted on a destination word boundary during the transfer portion of the algorithm. It may also be interrupted at the end of any row in the array. The context of the PIXBLT is saved in reserved registers. The PBX bit is set in the copy of the ST register that is pushed to the stack. The worst case latency caused by an interrupt is 20 machine states for the interrupt to be recognized. The time for the context switch must be added to this; see Section 8.5.1, Interrupt Latency (page 8-6) for context switch timings.

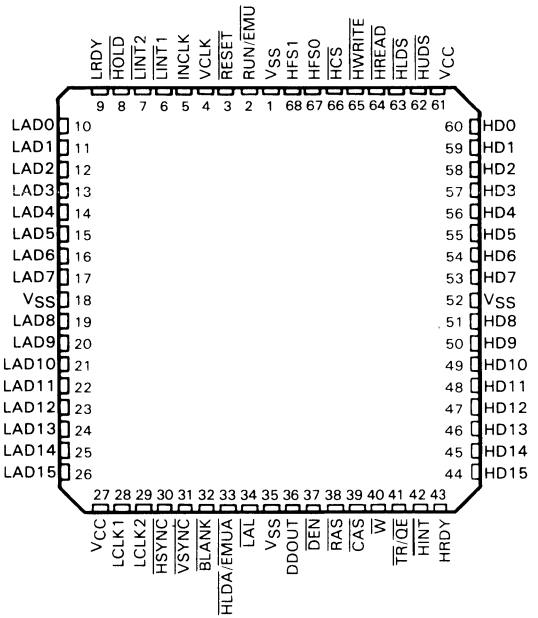
Appendix A

TMS34010 Data Sheet

JANUARY 1986 – REVISED JUNE 1988

- **Instruction Cycle Time:**
  - **132 ns . . . (TMS34010-60)**
  - **160 ns . . . (TMS34010-50)**
  - **200 ns . . . (TMS34010-40)**
- **Fully Programmable 32-Bit General-Purpose Processor with 128-Megabyte Address Range**
- **Pixel Processing, XY Addressing, and Window Checking Built into the Instruction Set**
- **Programmable 1, 2, 4, 8, or 16-Bit Pixel Size with 16 Boolean and 6 Arithmetic Pixel Processing Options (Raster-Ops)**
- **30 General-Purpose 32-bit Registers and 32-bit Stack Pointer**
- **256-Byte LRU On-Chip Instruction Cache**
- **Direct Interfacing to Both Conventional DRAM and Multiport Video RAM**
- **Dedicated 8/16-Bit Host Processor Interface and HOLD/HLDA Interface**
- **Programmable CRT Control ($\overline{\text{HSYNC}}$, $\overline{\text{VSYNC}}$, $\overline{\text{BLANK}}$)**
- **High-Level Language Support**
- **Full Line of Hardware and Software Development Tools Including a "C" Compiler**
- **68-Leaded Packaging (PLCC)**
- **5-Volt CMOS Technology**

FN PACKAGE
(TOP VIEW)



description

The TMS34010 Graphics System Processor (GSP) is an advanced high-performance CMOS 32-bit microprocessor optimized for graphics display systems. With a built-in instruction cache, the ability to simultaneously access memory and registers, and an instruction set designed specifically for raster graphics operation, the TMS34010 provides user-programmable control of the CRT interface as well as the memory interface (both standard DRAM and multiport video RAM). The 1-gigabit address space is completely bit-addressable on bit boundaries using variable width data fields (1 to 32 bits). Additional graphics addressing modes support 1, 2, 4, 8, and 16-bit wide pixels. The TMS34010 is exceptionally well-supported by graphics software interface standards such as CGI/VDI, DGIS, and MS-Windows, as well as a full line of hardware and software support tools. Current support is highlighted in the TMS34010 Third Party Reference Guide (literature number SPVB066A).

architecture

The TMS34010 is a CMOS 32-bit processor with hardware support for graphics operations such as PixBlts (raster ops) and curve-drawing algorithms. Also included is a complete set of general-purpose instructions with addressing tuned to support high-level languages. In addition to its ability to address a large external memory range, the TMS34010 contains 30 general-purpose 32-bit registers, a hardware stack pointer

and a 256-byte instruction cache. On-chip functions include 28 programmable I/O registers that contain CRT control, input/output control, and instruction parameters. The TMS34010 directly interfaces to dynamic RAMs and video RAMs and generates video monitor control signals. It also accommodates a conventional HOLD/HLDA shared access as well as a separate, generalized interface for communicating with any standard host processor.

pin descriptions

PIN NAME	PIN NUMBER	I/O	DESCRIPTION
			Host Interface Bus Pins
$\overline{HCS}$	66	I	Host chip select
HD0-HD15	44-51, 53-60	I/O	Host bidirectional data bus
HFS0, HFS1	67, 68	I	Host function select
$\overline{HINT}$	42	O	Host interrupt request
$\overline{HLDS}$	63	I	Host lower data select
$\overline{HUDS}$	62	I	Host upper data select
HRDY	43	O	Host ready
$\overline{HREAD}$	64	I	Host read strobe
$\overline{HWRITE}$	65	I	Host write strobe
			Local Bus Interface Pins
$\overline{RAS}$	38	O	Local row-address strobe
$\overline{CAS}$	39	O	Local column-address strobe
DDOUT	36	O	Local data direction out
$\overline{DEN}$	37	O	Local data enable
LAD0-LAD15	10-17, 19-26	I/O	Local address/data bus
$\overline{LAL}$	34	O	Local address latched
LCLK1, LCLK2	28, 29	O	Local output clocks
$\overline{LINT1}$, $\overline{LINT2}$	6, 7	I	Local interrupt request pins
LRDY	9	I	Local ready
$\overline{TR}/\overline{QE}$	41	O	Local shift register transfer or output enable
$\overline{W}$	40	O	Local write strobe
INCLK	5	I	Input clock
			Hold and Emulation
$\overline{HOLD}$	8	I	Hold request
RUN/$\overline{EMU}$	2	I	Run/Not emulate
$\overline{HLDA}/\overline{EMUA}$	33	O	Hold acknowledge or emulate acknowledge
			Video Timing Signals
$\overline{BLANK}$	32	O	Blanking
$\overline{HSYNC}$	30	I/O	Horizontal sync
VCLK	4	I	Video clock
$\overline{VSYNC}$	31	I/O	Vertical sync
			Miscellaneous
$\overline{RESET}$	3	I	Reset
V_{CC}	27, 61	I	Nominal 5-volt power supply
V_{SS}	1, 18, 35, 52	I	Ground

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

system block diagram

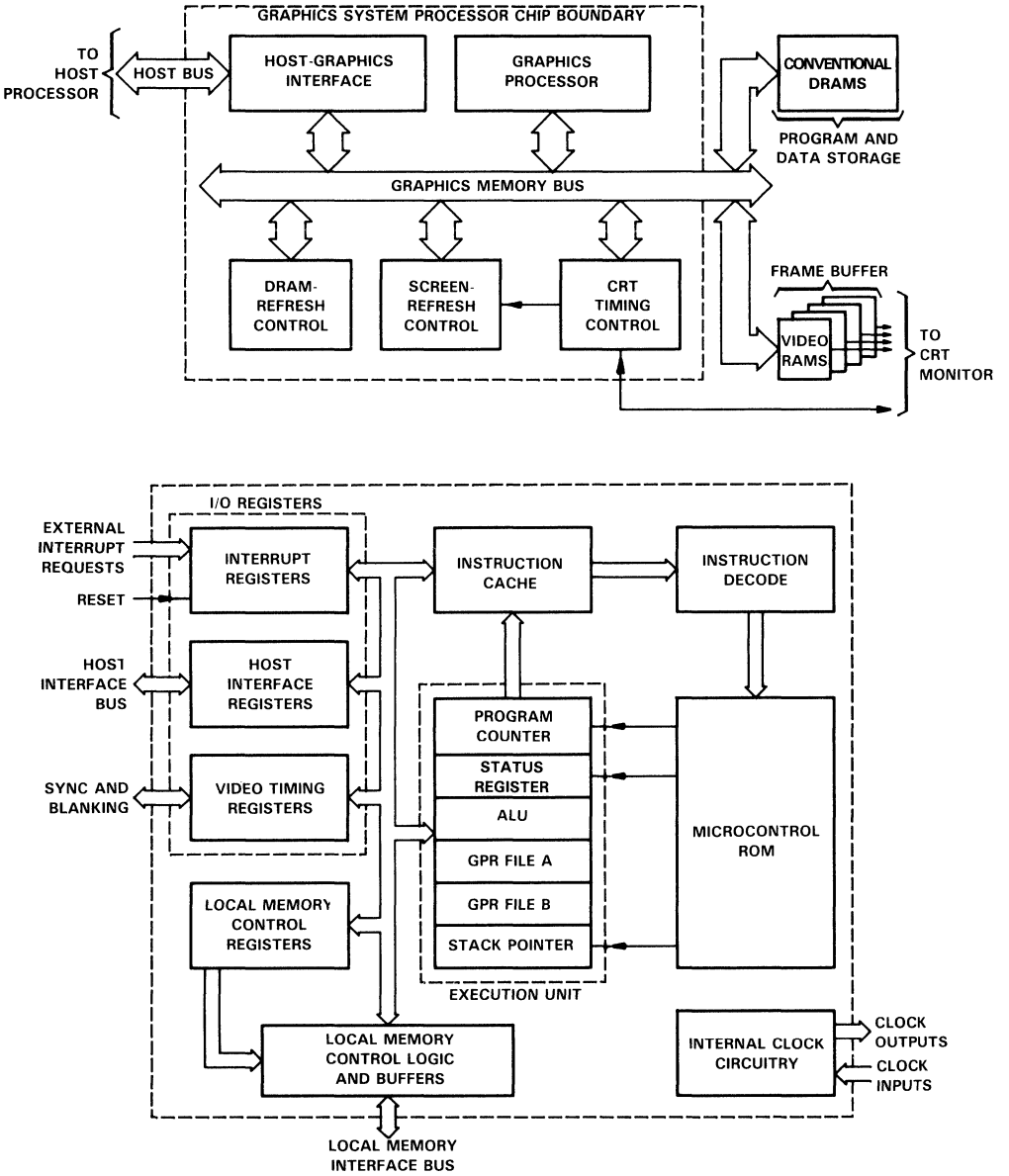


FIGURE 1. TMS34010 INTERNAL ARCHITECTURE

The TMS34010 provides single-cycle execution of most common integer arithmetic and Boolean operations from its instruction cache. Additionally, the TMS34010 incorporates a hardware barrel shifter that provides a single-state bidirectional shift and rotate function for 1 to 32 bits.

A microcoded local memory controller supports pipelined memory write operations of variable-size fields that can be performed in parallel with subsequent instruction execution.

TMS34010 graphics processing hardware supports pixel and pixel-array processing capabilities for both monochrome and color systems that have a variety of pixel sizes. The hardware incorporates two-operand raster operations with Boolean and arithmetic operations, XY addressing, window clipping, window checking operations, 1 to n bits per pixel transforms, transparency, and plane masking. The architecture further supports operations on single pixels (PIXT instructions) or on two-dimensional pixel arrays of arbitrary size (PixBlts).

The TMS34010's flexible graphics processing capabilities allow software-based graphics algorithms without sacrificing performance. These algorithms include: arbitrary window size, custom incremental curve drawing, and two-operand raster operations.

register files

Boolean, arithmetic, byte, and field move instructions operate on data within the TMS34010's general-purpose register files. The TMS34010 contains thirty-one 32-bit registers, including a system stack pointer (SP). The SP is accessible to both Register File A and B as the sixteenth register. Transfers between registers and memory are facilitated via a complete set of field MOVE instructions with selectable field sizes. Transfers between registers are facilitated via the MOVE instruction.

The fifteen general-purpose registers in Register File A are used for high-level language support and assembly language programming. The fifteen registers in Register File B are dedicated to special functions during PixBlts and other pixel operations, but can be used as general-purpose registers at other times.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

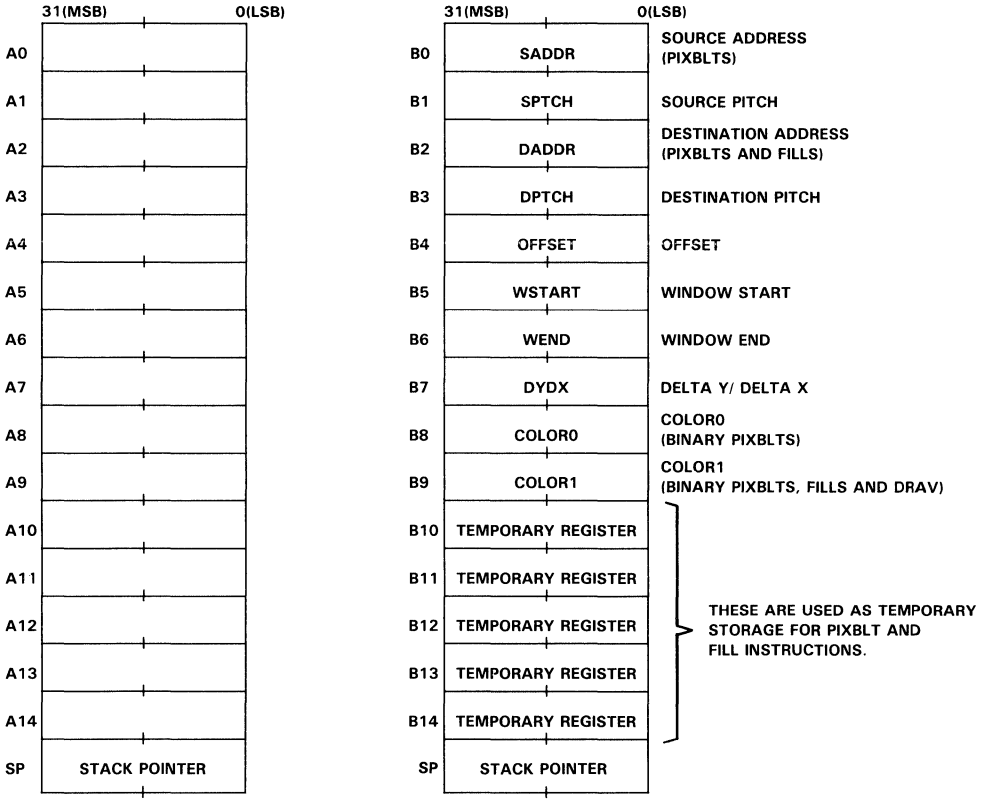


FIGURE 2. REGISTER FILES A AND B

program counter (PC)

The TMS34010's 32-bit program counter register points to the next instruction-stream word to be fetched. Since instruction words are aligned to 16-bit boundaries, the four LSBs of the PC are always zero.

instruction cache

An on-chip instruction cache contains 256 bytes of RAM and provides fast access to instructions. It operates automatically and is transparent to software. The cache is divided into four 64-byte segments. Associated with each segment is a 23-bit segment address register to identify the addresses in memory corresponding to the current contents of the cache segment. Each cache segment is further partitioned into eight subsegments of four words each. Each subsegment has associated with it a present (P) flag to indicate whether the subsegment contains valid data.

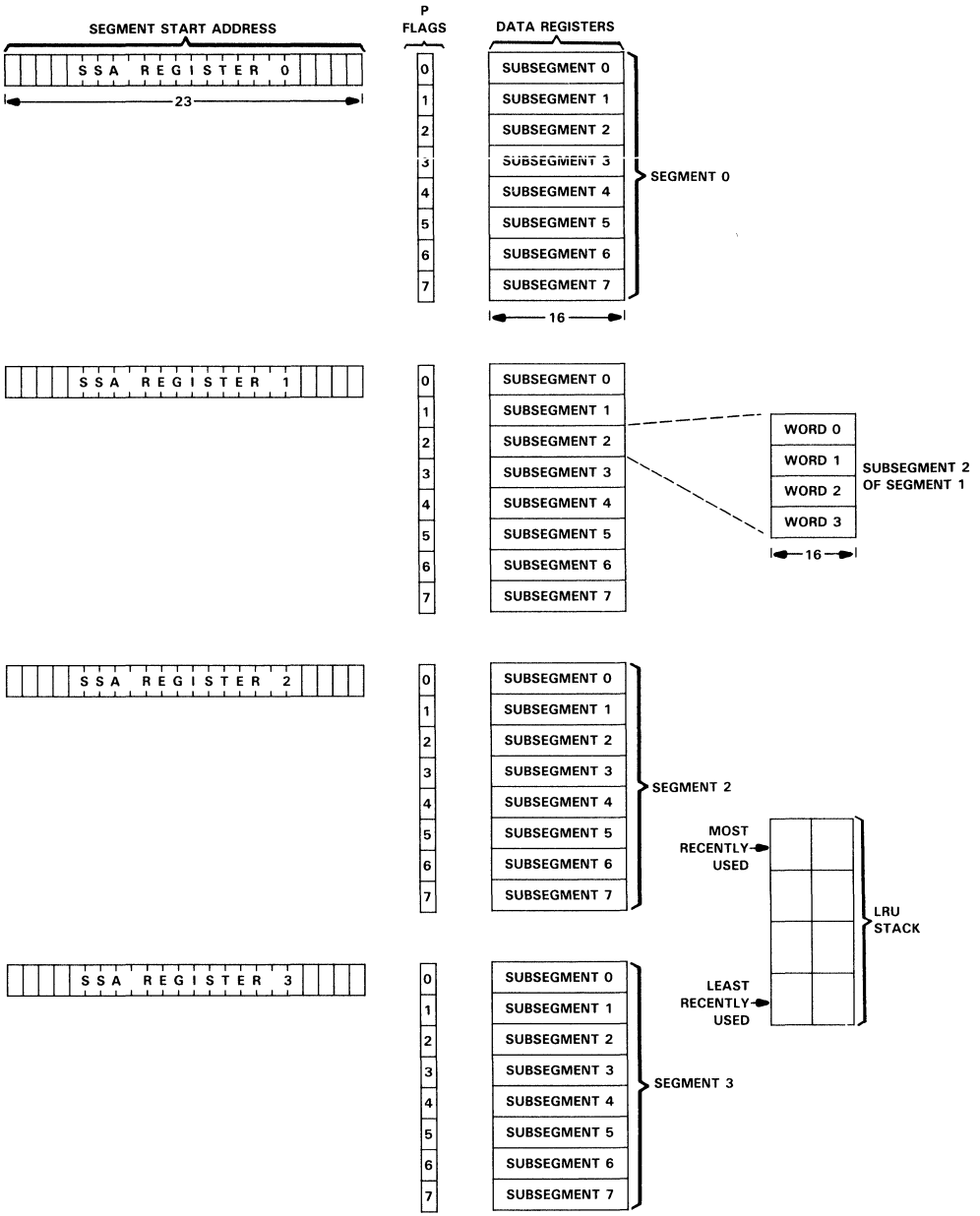


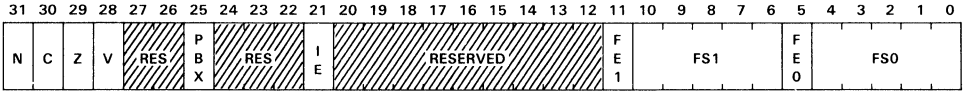
FIGURE 3. INSTRUCTION CACHE

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

The cache is loaded only when an instruction requested by the TMS34010 is not already contained within the cache. A least-recently-used (LRU) algorithm is used to determine which of the four segments of the cache is overwritten with the new data. For this purpose, an internal four-by-two LRU stack is used to keep track of cache usage.

status register

The status register (ST) is a special-purpose 32-bit register dedicated to status codes set by the results of implicit and explicit compare operations and parameters used to specify the length and behavior of fields 0 and 1.



N - Sign bit
C - Carry bit
Z - Zero bit
V - Overflow bit
PBX - PixBlt executing
IE - Interrupt enable bit
FE1 - Field extension bit 1
FS1 - Field size bit 1
FE0 - Field extension bit 0
FS0 - Field size bit 0

FIGURE 4. STATUS REGISTER

fields, bytes, pixels, and pixel arrays

A 26-bit address output by the TMS34010 selects a 16-bit word of physical memory; logically, however, the TMS34010 views memory data as fields addressable at the bit level. Primitive data types supported by the TMS34010 include: bytes, pixels, two 1- to 32-bit fields, and user-defined pixel arrays.

Fields 0 and 1 are specified independently to be from 1 to 32 bits in length. Bytes are special 8-bit cases of the field data type, while pixels are 1, 2, 4, 8 or 16 bits in length. In general, fields (including bytes) may start and terminate on arbitrary bit boundaries; pixels must pack evenly into 16-bit words.

pixel operations

Pixel arrays are two-dimensional data types of user-defined width, height, pixel depth (number of bits per pixel), and pitch (distance between rows). A pixel or pixel array may be accessed by means of either its memory address or its XY coordinates. Transfers of individual pixels or pixel blocks are influenced by the pixel processing, transparency, window checking, plane masking, or corner adjust operations selected.

I/O registers

The GSP contains an on-chip block of twenty-eight 16-bit I/O registers mapped into the TMS34010's memory address space. They can be accessed either by the TMS34010's CPU or by the host processor via the host interface. The I/O registers contain control parameters necessary to configure the operation of the following interfaces: interface to host processor (5 I/O registers), interface to local memory (6 registers), video timing and screen refresh functions (15 registers), and externally and internally generated interrupts (2 registers). The I/O registers also furnish status information on these interfaces.

ADDRESS	REGISTER	
0C00001F0h	REFCNT	DRAM REFRESH COUNT
0C00001E0h	DPYADR	DISPLAY ADDRESS
0C00001D0h	VCOUNT	VERTICAL COUNT
0C00001C0h	HCOUNT	HORIZONTAL COUNT
0C00001B0h	DPYTAP	DISPLAY TAP POINT
0C00001A0h		
0C0000190h	RESERVED	
0C0000180h		
0C0000170h		
0C0000160h	PMASK	PLANE MASK
0C0000150h	PSIZE	PIXEL SIZE
0C0000140h	CONVDP	CONVERSION (DESTINATION PITCH)
0C0000130h	CONVSP	CONVERSION (SOURCE PITCH)
0C0000120h	INTPEND	INTERRUPT PENDING
0C0000110h	INTENB	INTERRUPT ENABLE
0C0000100h	HSTCTLH	HOST CONTROL (8 MSB'S)
0C00000F0h	HSTCTLL	HOST CONTROL (8 LSB'S)
0C00000E0h	HSTADRH	HOST ADDRESS (16 MSB'S)
0C00000D0h	HSTADRL	HOST ADDRESS (16 LSB'S)
0C00000C0h	HSTDATA	HOST DATA
0C00000B0h	CONTROL	CONTROL
0C00000A0h	DPYINT	DISPLAY INTERRUPT
0C0000090h	DPYSTRT	DISPLAY START
0C0000080h	DPYCTL	DISPLAY CONTROL
0C0000070h	VTOTAL	VIDEO TOTAL
0C0000060h	VSBLNK	VERTICAL START BLANK
0C0000050h	VEBLNK	VERTICAL END BLANK
0C0000040h	VESYNC	VERTICAL END SYNC
0C0000030h	HTOTAL	HORIZONTAL TOTAL
0C0000020h	HSBLNK	HORIZONTAL START BLANK
0C0000010h	HEBLNK	HORIZONTAL END BLANK
0C0000000h	HESYNC	HORIZONTAL END SYNC

FIGURE 5. I/O REGISTERS

host interface registers

The host interface registers are provided for communications between the TMS34010 and the host processor. The registers are mapped into five of the I/O register locations accessible to the TMS34010. These same registers are mapped into four locations in the GSP interface to the host.

One of the registers is devoted to host interface control functions such as the passing of interrupt requests and 3-bit status codes from host to TMS34010 and from TMS34010 to host. Other control functions available to the host processor include flushing the instruction cache, halting the TMS34010, and transmitting a non-maskable interrupt request to the TMS34010.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

The remaining host registers are used for block transfers between the TMS34010 and host processor. The host uses these registers to indirectly access blocks within the TMS34010's local memory. Two of the 16-bit registers contain the 32-bit address of the current word location in memory. Another 16-bit register buffers data transferred to and from the memory by the host processor. The host interface can be programmed to automatically increment the pointer address following each transfer to provide the host with rapid access to a block of sequential addresses.

memory interface control registers

Six of the I/O registers are dedicated to various local memory interface functions including:

- Frequency and type of DRAM refresh cycles
- Pixel size
- Masking (write protection) of individual color planes
- Various pixel access control parameters
  - Window checking mode
  - Boolean or arithmetic pixel processing operation
  - Transparency
  - PixBlt direction control

video timing and screen refresh

Fourteen I/O registers are dedicated to video timing and screen refresh functions. The TMS34010 generates the horizontal sync ($\overline{\text{HSYNC}}$), vertical sync ($\overline{\text{VSYNC}}$), and blanking ($\overline{\text{BLANK}}$) signals used to drive a video monitor in a graphics system. These signals are controlled by means of a set of programmable video timing I/O registers and are based on the input video clock, VCLK. VCLK does not have to be synchronous with respect to INCLK, the TMS34010's CPU input clock.

The TMS34010 directly supports multiport video RAMs (VRAMs) by generating the memory-to-register load cycles necessary to refresh the display being shown on the video monitor. The memory locations from which display information is taken, as well as the number of horizontal scan lines displayed between memory-to-register load cycles, are programmable. VRAM tap point addresses are also fully programmable to support horizontal panning.

The TMS34010 supports various screen resolutions and either interlaced or noninterlaced video. The TMS34010 can optionally be programmed to synchronize to externally generated sync signals so that graphics images created by the TMS34010 can be superimposed upon images created externally. The external sync mode can also be used to synchronize the video signals generated by two or more TMS34010 chips in a multiple-TMS34010 graphics system.

interrupt interface registers

Two dedicated I/O registers monitor and mask interrupt requests to the TMS34010, including two externally generated interrupts and three internally generated interrupts. An internal interrupt request can be generated on one of the following conditions:

- Window violation: an attempt has been made to write a pixel to a location inside or outside a specified window boundary.
- Host interrupt: the host processor has set the interrupt request bit in the host control register.
- Display interrupt: a specified line number in the frame has been displayed on the screen.

A nonmaskable interrupt occurs when the host processor sets a particular control bit in the host interface registers. The TMS34010 reset function is controlled by a dedicated pin.

memory controller/local memory interface

The memory controller manages the TMS34010's interface to the local memory and automatically performs the bit alignment and masking necessary to access data located at arbitrary bit boundaries within memory. The memory controller operates autonomously with respect to the CPU. It has a "write queue" one field (1 to 32 bits) deep that permits it to complete the memory cycles necessary to insert the field into memory without delaying the execution of subsequent instructions. Only when a second memory operation is required before the memory controller has completed the first operation is the TMS34010 forced to defer instruction execution.

The TMS34010 directly interfaces to all standard dynamic RAMs and, in particular to JEDEC standard 64K and 256K video RAMs such as the TMS4161 and TMS4461 Multiport VRAMs. The TMS34010 memory interface consists of a triple-multiplexed address/data bus plus the associated control signals. Row address, column address, and data are multiplexed over the same address/data lines. DRAM refresh is supported with a variety of modes including $\overline{CAS}$-before-$\overline{RAS}$ refresh.

TMS34010 memory map

From the programmer's point of view, the TMS34010 treats data and instructions as residing in the same memory space.

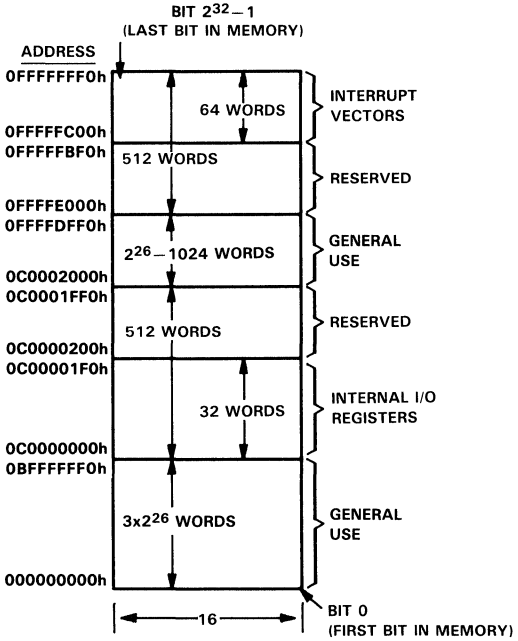


FIGURE 6. MEMORY MAP

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

instruction set

The TMS34010 instructions fall into three categories. The *graphics instructions* manipulate pixel data, accessed via memory addresses or XY coordinates. They provide support for graphics operations such as array and raster ops, pixel processing, windowing, plane masking, and transparency. The *move instructions* comprehend bit addressing and field operations; they manipulate fields of data using linear addressing for transfer to and from memory and the register file. The TMS34010 *general-purpose instructions* provide a complete set of arithmetic and Boolean operations on the register file as well as general program control and data processing. Partial timing information is provided in the table below. The two values given for jump instructions in the Minimum Cycles column indicate the jump and no-jump conditions, respectively. Full timing information can be obtained in the *TMS34010 User's Guide* (number SPVU001A).

The following abbreviations are used below in the opcodes: S (source register), D (destination register), R (register file select), F (field select), K (constant), M (cross A/B file boundary), Z (draw option), code (jump select code), X (don't care), N (trap select and stack adjust), RS (source register), RD (destination register), xxxx (address displacement), IL (32-bit immediate operand), and IW (16-bit immediate operand).

GRAPHICS INSTRUCTIONS

SYNTAX	DESCRIPTION	NO. WORDS	MINIMUM CYCLES	16-BIT OPCODE MSB			LSB	STATUS BITS			
ADDXY Rs, Rd	Add Registers in XY Mode	1	1	1110	000S	SSSR	DDDD	N	C	Z	V
CMPXY Rs, Rd	Compare X and Y Halves of Registers	1	3	1110	010S	SSSR	DDDD	N	C	Z	V
CPW Rs, Rd	Compare Point to Window	1	1	1110	011S	SSSR	DDDD	–	–	–	V
CVXYL Rs, Rd	Convert XY Address to Linear Address	1	3	1110	100S	SSSR	DDDD	–	–	–	–
DRAV Rs, Rd	Draw and Advance	1	†	1111	011S	SSSR	DDDD	–	–	–	V
FILL L	Fill Array with Processed Pixels: Linear	1	†	0000	1111	1100	0000	–	–	–	–
FILL XY	Fill Array with Processed Pixels: XY	1	†	0000	1111	1110	0000	–	–	–	V
LINE Z	Line Draw	1	†	1101	1111	Z001	1010	–	–	–	V
MOVX Rs, Rd	Move X Half of Register	1	1	1110	110S	SSSR	DDDD	–	–	–	–
MOVY Rs, Rd	Move Y Half of Register	1	1	1110	111S	SSSR	DDDD	–	–	–	–
PIXBLT B,L	Pixel Block Transfer: Binary to Linear	1	†	0000	1111	1000	0000	–	–	–	–
PIXBLT B,XY	Pixel Block Transfer and Expand: Binary to XY	1	†	0000	1111	1010	0000	–	–	–	V
PIXBLT L,L	Pixel Block Transfer: Linear to Linear	1	†	0000	1111	0000	0000	–	–	–	–
PIXBLT L,XY	Pixel Block Transfer: Linear to XY	1	†	0000	1111	0010	0000	–	–	–	V
PIXBLT XY, L	Pixel Block Transfer: XY to Linear	1	†	0000	1111	0100	0000	–	–	–	–
PIXBLT XY,XY	Pixel Block Transfer: XY to XY	1	†	0000	1111	0110	0000	–	–	–	V
PIXT Rs,*Rd	Pixel Transfer: Register to Indirect	1	†	1111	100S	SSSR	DDDD	–	–	–	–
PIXT Rs,*Rd.XY	Pixel Transfer: Register to Indirect XY	1	†	1111	000S	SSSR	DDDD	–	–	–	V
PIXT *Rs, Rd	Pixel Transfer: Indirect to Register	1	4	1111	101S	SSSR	DDDD	–	–	–	–
PIXT *Rs,*Rd	Pixel Transfer: Indirect to Indirect	1	†	1111	110S	SSSR	DDDD	–	–	–	–
PIXT *Rs.XY, Rd	Pixel Transfer: Indirect XY to Register	1	6	1111	001S	SSSR	DDDD	–	–	–	–
PIXT *Rs.XY,*Rd.XY	Pixel Transfer: Indirect XY to Indirect XY	1	†	1111	010S	SSSR	DDDD	–	–	–	V
SUBXY Rs,Rd	Subtract Registers in XY Mode	1	1	1110	001S	SSSR	DDDD	N	C	Z	V

†Number of cycles depends on pixel size and/or pixel array size and graphics option selected. See TMS34010 User's Guide (SPVU001A).

MOVE INSTRUCTIONS

SYNTAX	DESCRIPTION	NO. WORDS	MINIMUM CYCLES	16-BIT OPCODE MSB LSB	STATUS BITS
MOVB Rs,*Rd	Move Byte: Register to Indirect	1	†	1000 110S SSSR DDDD	– – – –
MOVB *Rs.Rd	Move Byte: Indirect to Register	1	†	1000 111S SSSR DDDD	N – Z 0
MOVB *Rs,*Rd	Move Byte: Indirect to Indirect	1	†	1001 110S SSSR DDDD	– – – –
MOVB Rs,*Rd(offset)	Move Byte: Register to Indirect with offset.	2	†	1010 110S SSSR DDDD	– – – –
MOVB *Rs(offset),Rd	Move Byte: Indirect with offset. to Register	2	†	1010 111S SSSR DDDD	N – Z 0
MOVB *Rs(offset),*Rd(offset)	Move Byte: Ind. with offset. to Ind. with offset.	3	†	1011 110S SSSR DDDD	– – – –
MOVB Rs,@Daddress	Move Byte: Register to Absolute	3	†	0000 0101 111R SSSS	– – – –
MOVB @Saddress,Rd	Move Byte: Absolute to Register	3	†	0000 0111 111R DDDD	N – Z 0
MOVB @Saddress,@Daddress	Move Byte: Absolute to Absolute	5	†	0000 0011 0100 0000	– – – –
MOVE Rs,Rd	Move Register to Register	1	1	0100 11MS SSSR DDDD	N – Z 0
MOVE Rs,*Rd,F	Move Field: Register to Indirect	1	†	1000 00FS SSSR DDDD	– – – –
MOVE Rs,–*Rd,F	Move Field: Register to Indirect (pre-dec)	1	†	1010 00FS SSSR DDDD	– – – –
MOVE Rs,*Rd+,F	Move Field: Register to Indirect (post-inc)	1	†	1001 00FS SSSR DDDD	– – – –
MOVE *Rs,Rd,F	Move Field: Indirect to Register	1	†	1000 01FS SSSR DDDD	N – Z 0
MOVE –*Rs,Rd,F	Move Field: Indirect (pre-dec) to Register	1	†	1010 01FS SSSR DDDD	N – Z 0
MOVE *Rs+,Rd,F	Move Field: Indirect (post-inc) to Register	1	†	1001 01FS SSSR DDDD	N – Z 0
MOVE *Rs,*Rd,F	Move Field: Indirect to Indirect	1	†	1000 10FS SSSR DDDD	– – – –
MOVE –*Rs,–*Rd,F	Move Field: Ind. (pre-dec) to Ind. (pre-dec)	1	†	1010 10FS SSSR DDDD	– – – –
MOVE *Rs+,*Rd+,F	Move Field: Ind. (post-inc) to Ind. (post-inc)	1	†	1001 10FS SSSR DDDD	– – – –
MOVE Rs,*Rd(offset),F	Move Field: Register to Indirect with offset.	2	†	1011 00FS SSSR DDDD	– – – –
MOVE *Rs(offset),Rd,F	Move Field: Indirect with offset. to Register	2	†	1011 01FS SSSR DDDD	N – Z 0
MOVE *Rs(offset),*Rd+,F	Move Field: Ind. with offset. to Ind. (post-inc)	2	†	1101 00FS SSSR DDDD	– – – –
MOVE *Rs(offset),*Rd(offset),F	Move Field: Ind. with offset. to Ind. with offset.	3	†	1011 10FS SSSR DDDD	– – – –
MOVE Rs,@Daddress,F	Move Field: Register to Absolute	3	†	0000 01F1 100R SSSS	– – – –
MOVE @Saddress,Rd,F	Move Field: Absolute to Register	3	†	0000 01F1 101R DDDD	N – Z 0
MOVE @Saddress,*Rd+,F	Move Field: Absolute to Indirect (post-inc)	3	†	1101 01F0 000R DDDD	– – – –
MOVE @Saddress,@Daddress,F	Move Field: Absolute to Absolute	5	†	0000 01F1 1100 0000	– – – –

†Number of cycles depends on field size and alignment. See TMS34010 User's Guide (SPVU001A).

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

GENERAL INSTRUCTIONS

SYNTAX	DESCRIPTION	NO. WORDS	MINIMUM CYCLES	16-BIT OPCODE MSB LSB	STATUS BITS
ABS Rd	Store Absolute Value	1	1	0000 0011 100R DDDD	N – Z 0
ADD Rs,Rd	Add Registers	1	1	0100 000S SSSR DDDD	N C Z V
ADDC Rs,Rd	Add Register with Carry	1	1	0100 001S SSSR DDDD	N C Z V
ADDI IW,Rd	Add Immediate (16 Bits)	2	2	0000 1011 000R DDDD	N C Z V
ADDI IL,Rd	Add Immediate (32 Bits)	3	3	0000 1011 001R DDDD	N C Z V
ADDK K,Rd	Add Constant (5 Bits)	1	1	0001 00KK KKKR DDDD	N C Z V
AND Rs,Rd	AND Registers	1	1	0101 000S SSSR DDDD	– – Z –
ANDI IL,Rd	AND Immediate (32 Bits)	3	3	0000 1011 100R DDDD	– – Z –
ANDN Rs,Rd	AND Register with Complement	1	1	0101 001S SSSR DDDD	– – Z –
ANDNI IL,Rd	AND Not Immediate (32 Bits)	3	3	0000 1011 100R DDDD	– – Z –
BTST K,Rd	Test Register Bit - Constant	1	1	0001 11KK KKKR DDDD	– – Z –
BTST Rs,Rd	Test Register Bit - Register	1	2	0100 101S SSSR DDDD	– – Z –
CLR Rd	Clear Register	1	1	0101 011D DDDR DDDD	– – – –
CLRC	Clear Carry	1	1	0000 0011 0010 0000	– 0 – –
CMP Rs,Rd	Compare Registers	1	1	0100 100S SSSR DDDD	N C Z V
CMPI IW,Rd	Compare Immediate (16 Bits)	2	2	0000 1011 010R DDDD	N C Z V
CMPI IL,Rd	Compare Immediate (32 Bits)	3	3	0000 1011 011R DDDD	N C Z V
DEC Rd	Decrement Register	1	1	0001 0100 001R DDDD	– – – –
DINT	Disable Interrupts	1	3	0000 0011 0110 0000	– – – –
DIVS Rs,Rd	Divide Registers Signed	1	40	0101 100S SSSR DDDD	N – Z V
DIVU Rs,Rd	Divide Registers Unsigned	1	37	0101 101S SSSR DDDD	– – Z V
EINT	Enable Interrupts	1	3	0000 1101 0110 0000	– – – –
EXGF Rd,F	Exchange Field Size	1	1	1101 01F1 000R DDDD	– – – –
LMO Rs,Rd	Leftmost One	1	1	0110 101S SSSR DDDD	– – Z –
MMFM Rs,Register List	Move Multiple Registers from Memory	2	†	0000 1001 101R DDDD	– – – –
MMTM Rd,Register List	Move Multiple Registers to Memory	2	†	0000 1001 100R DDDD	– – – –
MODS Rs,Rd	Modulus Signed	1	40	0110 110S SSSR DDDD	N – Z V
MODU Rs,Rd	Modulus Unsigned	1	35	0110 111S SSSR DDDD	– – Z V
MOVI IW,Rd	Move Immediate (16 Bits)	2	2	0000 1001 110R DDDD	N – Z 0
MOVI IL,Rd	Move Immediate (32 Bits)	3	3	0000 1001 111R DDDD	N – Z 0
MOVK K,Rd	Move Constant (5 Bits)	1	1	0001 10KK KKKR DDDD	– – – –
MPYS Rs,Rd	Multiply Registers (Signed)	1	$5+\frac{FS1}{2}$	0101 110S SSSR DDDD	N – Z –
MPYU Rs,Rd	Multiply Registers (Unsigned)	1	$5+\frac{FS1}{2}$	0101 111S SSSR DDDD	– – Z –
NEG Rd	Negate Register	1	1	0000 0011 101R DDDD	N C Z V
NEGB Rd	Negate Register with Borrow	1	1	0000 0011 110R DDDD	N C Z V
NOP	No operation	1	1	0000 0011 0000 0000	– – – –
NOT Rd	Complement Register	1	1	0000 0011 111R DDDD	– – Z –
OR Rs,Rd	OR Registers	1	1	0101 010S SSSR DDDD	– – Z –
ORI IL,Rd	OR Immediate (32 bits)	3	3	0000 1011 101R DDDD	– – Z –
RL K,Rd	Rotate Left - Constant	1	1	0011 00KK KKKR DDDD	– C Z –
RL Rs,Rd	Rotate Left - Register	1	1	0110 100S SSSR DDDD	– C Z –
SETC	Set Carry	1	1	0000 1101 1110 0000	– 1 – –
SETF FS,FE,F	Set Field Parameters	1	1,2	0000 01F1 01FS SSSS	– – – –
SEXT Rd,F	Sign Extend to Long	1	3	0000 01F1 000R DDDD	N – Z –

†Number of cycles depends on number of registers in list and stack alignment. See TMS34010 User's Guide (SPVU001A).

SYNTAX	DESCRIPTION	NO. WORDS	MINIMUM CYCLES	16-BIT OPCODE MSB			LSB	STATUS BITS			
SLA K,Rd	Shift Left Arithmetic - Constant	1	3	0010	00KK	KKKR	DDDD	N	C	Z	V
SLA Rs,Rd	Shift Left Arithmetic - Register	1	3	0110	000S	SSSR	DDDD	N	C	Z	V
SLL K,Rd	Shift Left Logical - Constant	1	1	0010	01KK	KKKR	DDDD	–	C	Z	–
SLL Rs,Rd	Shift Left Logical - Register	1	1	0110	001S	SSSR	DDDD	–	C	Z	–
SRA K,Rd	Shift Right Arithmetic - Constant	1	1	0010	10KK	KKKR	DDDD	N	C	Z	–
SRA Rs,Rd	Shift Right Arithmetic - Register	1	1	0110	010S	SSSR	DDDD	N	C	Z	–
SRL K,Rd	Shift Right Logical - Constant	1	1	0010	11KK	KKKR	DDDD	–	C	Z	–
SRL Rs,Rd	Shift Right Logical - Register	1	1	0110	011S	SSSR	DDDD	–	C	Z	–
SUB Rs,Rd	Subtract Registers	1	1	0100	010S	SSSR	DDDD	N	C	Z	V
SUBB Rs,Rd	Subtract Registers with Borrow	1	1	0100	011S	SSSR	DDDD	N	C	Z	V
SUBI IW,Rd	Subtract Immediate (16 Bits)	2	2	0000	1011	111R	DDDD	N	C	Z	V
SUBI IL,Rd	Subtract Immediate (32 Bits)	3	3	0000	1101	000R	DDDD	N	C	Z	V
SUBK K,Rd	Subtract Immediate (5 Bits)	1	1	0001	01KK	KKKR	DDDD	N	C	Z	V
XOR Rs,Rd	Exclusively OR Registers	1	1	0101	011S	SSSR	DDDD	–	–	Z	–
XORI IL,Rd	Exclusively OR Immediate Value (32 Bits)	3	3	0000	1011	110D	DDDD	–	–	Z	–
ZEXT Rd,F	Zero Extend to Long	1	1	0000	01F1	001R	DDDD	–	–	Z	–

PROGRAM CONTROL AND CONTEXT SWITCHING

SYNTAX	DESCRIPTION	NO. WORDS	MINIMUM CYCLES†	16-BIT OPCODE MSB			LSB	STATUS BITS			
CALL Rs	Call Subroutine Indirect	1	6	0000	1001	001R	DDDD	–	–	–	–
CALLA Address	Call Subroutine Absolute	3	6	0000	1101	0101	1111	–	–	–	–
CALLR Address	Call Subroutine Relative	2	5	0000	1101	0011	1111	–	–	–	–
DSJ Rd,Address	Decrement Register and Skip Jump	2	3,2	0000	1101	100R	DDDD	–	–	–	–
DSJEQ Rd,Address	Conditionally Decrement Register and Skip Jump	2	3,2	0000	1101	101R	DDDD	–	–	–	–
DSJNE Rd,Address	Conditionally Decrement Register and Skip Jump	2	3,2	0000	1101	110R	DDDD	–	–	–	–
DSJS Rd,Address	Decrement Register and Skip Jump - Short	1	2,3	0011	1Dxx	xxxR	DDDD	–	–	–	–
EMU	Initiate Emulation	1	6	0000	0001	0000	0000	–	–	–	–
EXGPC Rd	Exchange Program Counter with Register	1	2	0000	0001	001R	DDDD	–	–	–	–
GETPC Rd	Get Program Counter into Register	1	1	0000	0001	010R	DDDD	–	–	–	–
GETST Rd	Get Status Register into Register	1	1	0000	0001	100R	DDDD	–	–	–	–
JAcc Address	Jump Absolute Conditional	3	3,4	1100	code	1000	0000	–	–	–	–
JRcc Address	Jump Relative Conditional	2	3,2	1100	code	0000	0000	–	–	–	–
JRcc Address	Jump Relative Conditional - Short	1	2,1	1100	code	xxxx	xxxx	–	–	–	–
JUMP Rs	Jump Indirect	1	2	0000	0001	011R	DDDD	–	–	–	–
POPST	Pop Status Register from Stack	1	8	0000	0001	1100	0000	–	–	–	–
PUSHST	Push Status Register onto Stack	1	2	0000	0001	1110	0000	–	–	–	–
PUTST Rs	Copy Register into Status	1	3	0000	0001	101R	DDDD	N	C	Z	V
RETI	Return from Interrupt	1	11	0000	1001	0100	0000	N	C	Z	V
RETS (N)	Return from Subroutine	1	7	0000	1001	011N	NNNN	–	–	–	–
REV Rd	Get Revision Number	1	1	0000	0000	001R	DDDD	–	–	–	–
TRAP N	Software Interrupt	1	16	0000	1001	000N	NNNN	0	0	0	0

†Where two numbers appear, the first number assumes that the jump is taken, and the second assumes that the jump is not taken.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

development systems and software support

Texas Instruments, together with third party suppliers, offers a full range of hardware and software development tools for the TMS34010. The support environment is aimed at four areas of support with the key tools based on the IBM PC, DEC VAX, SUN, MAC II, APOLLO and TI Professional computers:

DESIGNER	TOOLS
Hardware	XDS-22 Real Time Emulator (with PC-based Debugger Interface)
	PC Software Development Board (with Debugger Interface)
Software	Assembly Language Package, including:
	Assembler, Linker, Archiver, ROM Object Format Converter , Software Simulator (PC only)
	Graphics/Math Function Library
	Bit-Mapped Font Library
	CCITT Data Compression/Decompression Function Library
	8514A Emulation Function Library
Languages	C Compiler Package including:
	TMS34010 C Compiler
	Runtime Support
Systems	Window Management Support
	Image Processing Support
	Graphics Interfaces and Standards
	Debugger Adaptation Software

Further support is provided through a network of Regional Technology Centers (RTCs).

TMS340 FAMILY HARDWARE AND SOFTWARE SUPPORT

SILICON	PART NUMBER
Graphics System Processor 68-Pin PLCC	TMS34010FNL
Video System Controller 68-Pin PLCC	TMS34061FNL
Color Palette 22-Pin DIP	TMS34070NL
64Kx1 Multiport Memory 22-Lead PLCC (120 and 150 ns)	TMS4161FML
64Kx1 Multiport Memory 22-Pin DIP (120 and 150 ns)	TMS4161NL
64Kx4 Multiport Memory 24-Pin DIP (120 and 150 ns)	TMS4461NL

SOFTWARE TOOLS	COMPUTER	OPERATING SYSTEM	PART NUMBER
TMS34010 Assembler Package:			
Assembler, Linker, Archiver,	IBM/TI PC	MS-DOS 2.11+	TMDS3440808002
Object Format Converter, Simulator			
TMS34010 Assembler Package:			
Assembler, Linker, Archiver,	VAX	VMS	TMDS3440200059
Object Format Converter	VAX	ULTRIX	TMDS3440200069
	VAX	System V	TMDS3440200089
	HP	System V	TMDS3440500089
	Sun	System V	TMDS3440550086
	Mac 11	MPW	TMDS3440560021
	Apollo	System V	TMDS3440570088
TMS34010 C Compiler Package	IBM/TI PC	MS-DOS 2.11+	TMDS3440805002
	VAX	VMS	TMDS3440205059
	VAX	ULTRIX	TMDS3440205069
	VAX	System V	TMDS3440205089
	HP	System V	TMDS3440505089
	Sun	System V	TMDS3440555086
	Mac II	MPW	TMDS3440565021
	Apollo	System V	TMDS3440575088
Combination packages:			
Assembler, Linker, Archiver,	IBM/TI PC	MS-DOS 2.11+	TMDS3440804003
Object Format Converter, Simulator,			
C Compiler with runtime support			
TMS34010 Graphics/Math			
Function Library	IBM/TI PC - Source		TMDS3440802202
	VAX - Source		TMDS3440802208
TMS34010 Bit-Mapped Font Library	IBM/TI PC	MS-DOS 2.11+	TMDS3440802302
	VAX	ALL	TMDS3440202308
TMS34010 CCITT Function Library	IBM/TI PC	MS-DOS 2.11+	TMDS3440802102
	VAX	All	TMDS3440202108
TMS34010	8514/A Emulation Library	TMDS-3440 8020 02 IBM/TIPC,	MS-DOS 2.11+
TMS34010 PC Debugger Development Package (For Internal Use)	IBM/TI PC	MS-DOS 2.11+	TMDS3440806002
TMS34010 PC Debuffer Development Package (For Resale)	IBM/TI PC	MS-DOS 2.11+	TMDS3440806003

HARDWARE TOOLS	COMPUTER	VERSION	PART NUMBER
TMS34010 XDS-22 Real-Time Emulator with BT&T		U.S.	TMDS3469910000
		Europe	TMDS3469981000
Color Graphics Controller Board (TMS34061, TMS34070)	IBM/TI PC		TMDS3471804000
TMS34010 Software Development Board	IBM/TI PC		TMDS3411804420

DESIGN KITS	PART NUMBER
TMS340 Graphics Design Kit, including TMS34061, TMS34070, TMS4161s	TMS340GDK
TMS34010 Graphics Design Kit, including TMS34010, TMS34070, TMS4461s, PC Assembler	TMS34010GDK

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

reset

Reset puts the TMS34010 into a known initial state. It is entered when the input signal at the $\overline{\text{RESET}}$ pin is asserted low. $\overline{\text{RESET}}$ must remain active low for a minimum of 40 local clock (LCLK1 and LCLK2) periods to ensure that the TMS34010 has sufficient time to establish its initial internal state.

While $\overline{\text{RESET}}$ remains asserted, all outputs are in a known state, no DRAM-refresh cycles take place, and no screen-refresh cycles are performed.

At the low-to-high transition of the $\overline{\text{RESET}}$ signal, the state of the $\overline{\text{HCS}}$ input determines whether the TMS34010 will be halted or begin executing instructions. The TMS34010 may be in one of two modes, host-present or self-bootstrap mode.

1. Host-Present Mode

   If $\overline{\text{HCS}}$ is high at the end of reset, TMS34010 instruction execution is halted and remains halted until the host clears the HLT (halt) bit in HSTCTL (host control register). Following reset, the eight $\overline{\text{RAS}}$-only refresh cycles required to initialize the dynamic RAMs are performed automatically by the TMS34010 memory control logic. As soon as the eight $\overline{\text{RAS}}$-only cycles are completed, the host is allowed access to TMS34010 memory. At this time, the TMS34010 begins to automatically perform DRAM refresh cycles at regular intervals. The TMS34010 remains halted until the host clears the HLT bit. Only then does the GSP fetch the level-0 vector address from location 0FFFFFFE0h and begin executing its reset service routine.

2. Self-Bootstrap Mode

   If $\overline{\text{HCS}}$ is low at the end of reset, the TMS34010 first performs the eight $\overline{\text{RAS}}$-only refresh cycles required to initialize the DRAMs. Immediately following the eight $\overline{\text{RAS}}$-only cycles, the TMS34010 fetches the level-0 vector address from location 0FFFFFFE0h, and begins executing its reset service routine.

Unlike other interrupts and software traps, reset does not save previous ST or PC values. This is because the value of the stack pointer just before a reset is generally not valid, and saving its value on the stack is unnecessary. A TRAP 0 instruction, which uses the same vector address as reset, similarly does not save the ST or PC values.

asserting reset

A reset is initiated by asserting the $\overline{\text{RESET}}$ input pin at its active-low level. To reset the TMS34010 at power up, $\overline{\text{RESET}}$ must remain active low for a minimum of 40 local clock periods after power levels have become stable. At times other than power up, the TMS34010 is also reset by holding $\overline{\text{RESET}}$ low for a minimum of 40 clock periods. The 40-clock interval is required to bring TMS34010 internal circuitry to a known initial state. While $\overline{\text{RESET}}$ remains asserted, the output and bidirectional signals are driven to a known state.

The TMS34010 drives its $\overline{\text{RAS}}$ signal inactive high as long as $\overline{\text{RESET}}$ remains low. The specifications for certain DRAM and VRAM devices, including the TMS4161, TMS4164 and TMS4464 devices, require that the $\overline{\text{RAS}}$ signal be driven inactive-high for 100 microseconds during system reset. Holding $\overline{\text{RESET}}$ low for 150 microseconds will cause the $\overline{\text{RAS}}$ signal to remain high for the 100 microseconds required to bring the memory devices to their initial states. DRAMs such as the TMS4256 specify an initial $\overline{\text{RAS}}$ high time of 200 microseconds, requiring that $\overline{\text{RESET}}$ be held low for 250 microseconds. In general, holding $\overline{\text{RESET}}$ low for t microseconds ensures that $\overline{\text{RAS}}$ remains high initially for $t - 50$ microseconds.

suspension of DRAM-refresh cycles during reset

An active-low level at the $\overline{\text{RESET}}$ pin is considered to be a power-up condition, and DRAM refresh is not performed until $\overline{\text{RESET}}$ goes inactive high. Consequently, the previous contents of the local memory may not be valid after a reset.

initial state following reset

While the $\overline{\text{RESET}}$ pin is asserted low, the TMS34010's output and bidirectional pins are forced to the states listed below.

INITIAL STATE OF PINS FOLLOWING A RESET

OUTPUTS DRIVEN TO HIGH LEVEL	OUTPUTS DRIVEN TO LOW LEVEL	BIDIRECTIONAL PINS DRIVEN TO HIGH IMPEDANCE
DDOUT	$\overline{\text{BLANK}}$	$\overline{\text{HSYNC}}$
HRDY		$\overline{\text{VSYNC}}$
$\overline{\text{DEN}}$		HD0-HD15
$\overline{\text{LAL}}$		LAD0-LAD15
$\overline{\text{TR/QE}}$		
$\overline{\text{RAS}}$		
$\overline{\text{CAS}}$		
$\overline{\text{W}}$		
$\overline{\text{HINT}}$		
$\overline{\text{HLDA/EMUA}}$		

Immediately following reset, all I/O registers are cleared (set to 0h), with the possible exception of the HLT bit in the HSTCTL register. The HLT bit is set to 1 if $\overline{\text{HCS}}$ is high just prior to the low-to-high transition of $\overline{\text{RESET}}$.

Just prior to execution of the first instruction in the reset routine, the TMS34010's internal registers are in the following state:

- General-purpose register files A and B are uninitialized.
- The ST is set to 00000010h.
- The PC contains the 32-bit vector fetched from memory address 0FFFFFFE0h.

TMS34010 local memory interface

The TMS34010 local memory interface consists of a triple-multiplexed address/data bus on which row addresses, column addresses, and data are transmitted. The associated memory control signals support direct interfacing to both DRAMs and VRAMs. At the beginning of a typical memory cycle, the address is output in multiplexed fashion as a row address followed by a column address. The remainder of the cycle is used to transfer data between the TMS34010 and memory.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

TMS34010 local memory interface

The TMS34010 local memory interface consists of a triple-multiplexed address/data bus on which row addresses, column addresses, and data are transmitted. The associated memory control signals support direct interfacing to both DRAMs and VRAMs. At the beginning of a typical memory cycle, the address is output in multiplexed fashion as a row address followed by a column address. The remainder of the cycle is used to transfer data between the TMS34010 and memory.

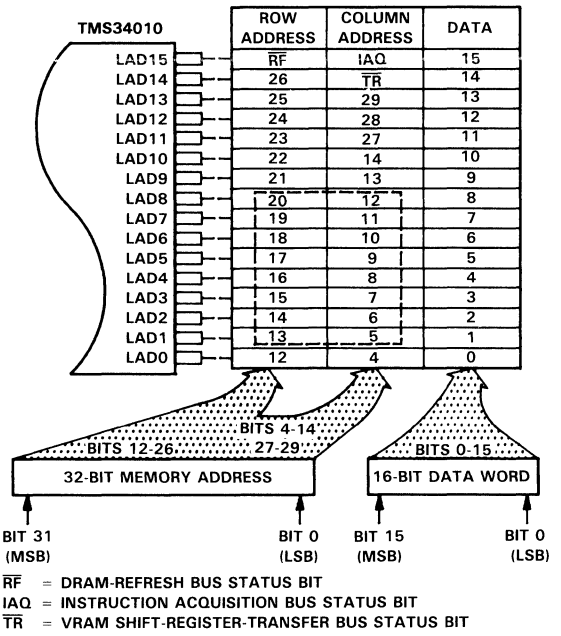


FIGURE 7. TRIPLE MULTIPLEXING OF ADDRESSES AND DATA

The following types of memory cycles are supported: read, write, VRAM memory-to-shift-register, VRAM shift-register-to-memory, $\overline{RAS}$-only DRAM refresh and $\overline{CAS}$-before-$\overline{RAS}$ DRAM refresh. The functional timing for these cycles is shown in the next six figures. Each memory cycle is a minimum of two machine states (a state is one local clock period) in duration. The seventh figure indicates the timing signals output during an internal cycle, i.e., a cycle during which no memory access takes place. An internal cycle is one state in duration.

During a memory cycle, the row address, column address, and data are transmitted over the same physical bus lines. The manner in which logical addresses are output at the memory interface makes external multiplexing hardware unnecessary, while supporting a wide variety of memory configurations. For example, in Figure 7, 16 consecutive address bits (5 through 20) are output on LAD1-LAD8 during the row and column address times. Output along with the address are bus status signals that indicate when DRAM refresh cycles, screen refresh (VRAM memory-to-shift-register) cycles, and instruction fetch cycles are occurring.

The following remarks apply to memory timing in general. A row address is output on LAD0-LAD15 at the start of the cycle, and is valid before and after the fall of $\overline{RAS}$. Next a column address is output on LAD0-LAD15. The column address is valid briefly before and after the falling edge of $\overline{LAL}$, but is not valid at the falling edge of $\overline{CAS}$. The column address is clocked into an external transparent latch (e.g., a 74AS373 octal latch) on the falling edge of $\overline{LAL}$ to provide the hold time on the column address required for dynamic RAMs and video RAMs. A transparent latch is required in order that the row address be available at the outputs of the latch during the start of the cycle.

Very large memory configurations may require external buffering of data lines. The $\overline{DEN}$ signal serves as the drive-enable signal to external bidirectional buffers, e.g., 74AS245 octal buffers. The DDOUT signal serves as the direction control for the buffers.

When an I/O register is addressed by the TMS34010, a special memory read or write cycle is performed. During this cycle, the external $\overline{RAS}$ signal falls, but the external $\overline{CAS}$ remains inactive-high for the duration of the cycle.

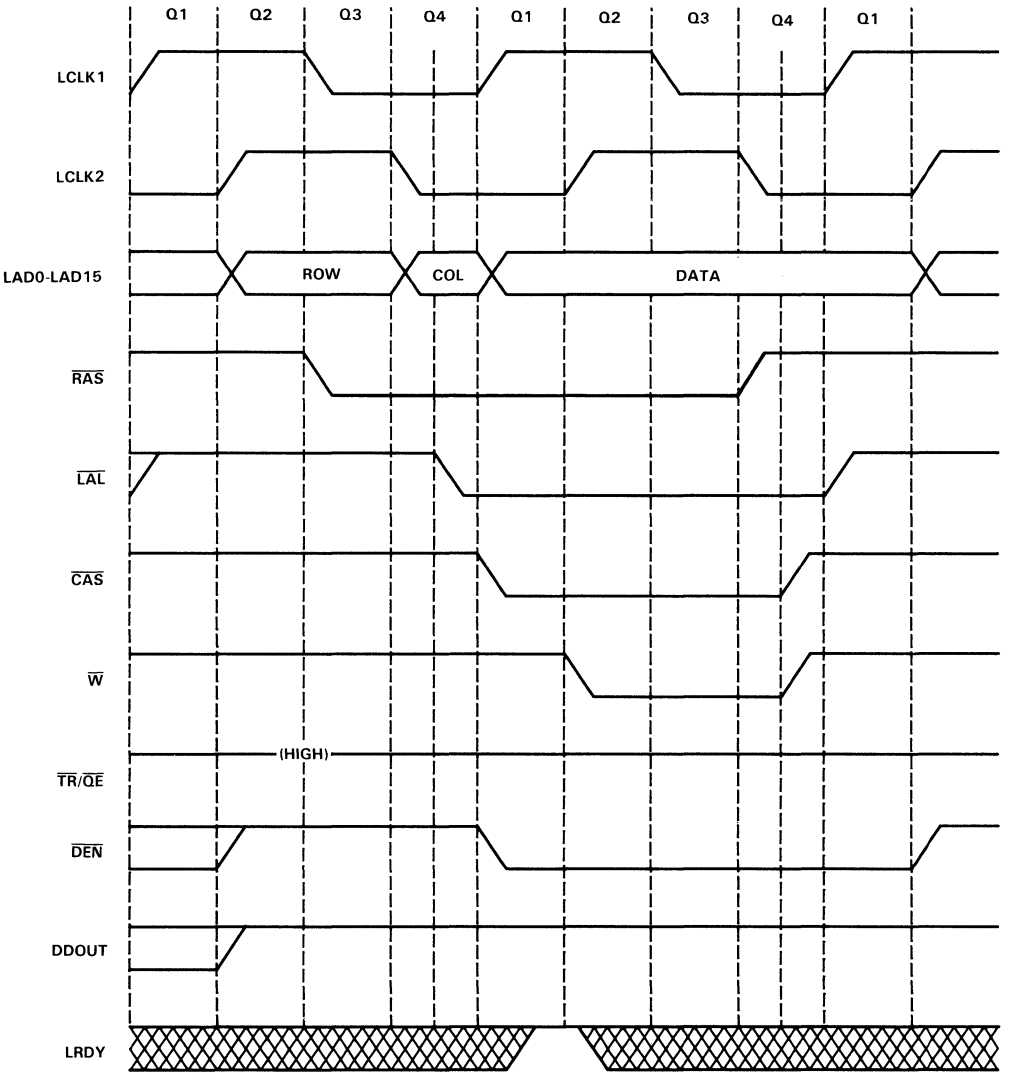
The timing shown in the first six functional timing diagrams assumes that the LRDY input remains high during the cycle. The LRDY pin is pulled low by slower memories requiring a longer cycle time. The TMS34010 samples the LRDY input at the end of Q1, as indicated in the figures. If LRDY is low, the TMS34010 inserts an additional state, called a "wait" state, into the cycle. Wait states continue to be inserted until LRDY is sampled at a high level. The cycle then completes in the manner indicated in the functional timing diagrams. A wait state is one local clock period in duration. Three additional timing diagrams provide examples of cycles extended by wait states.

The LRDY input is ignored by the TMS34010 during internal cycles.

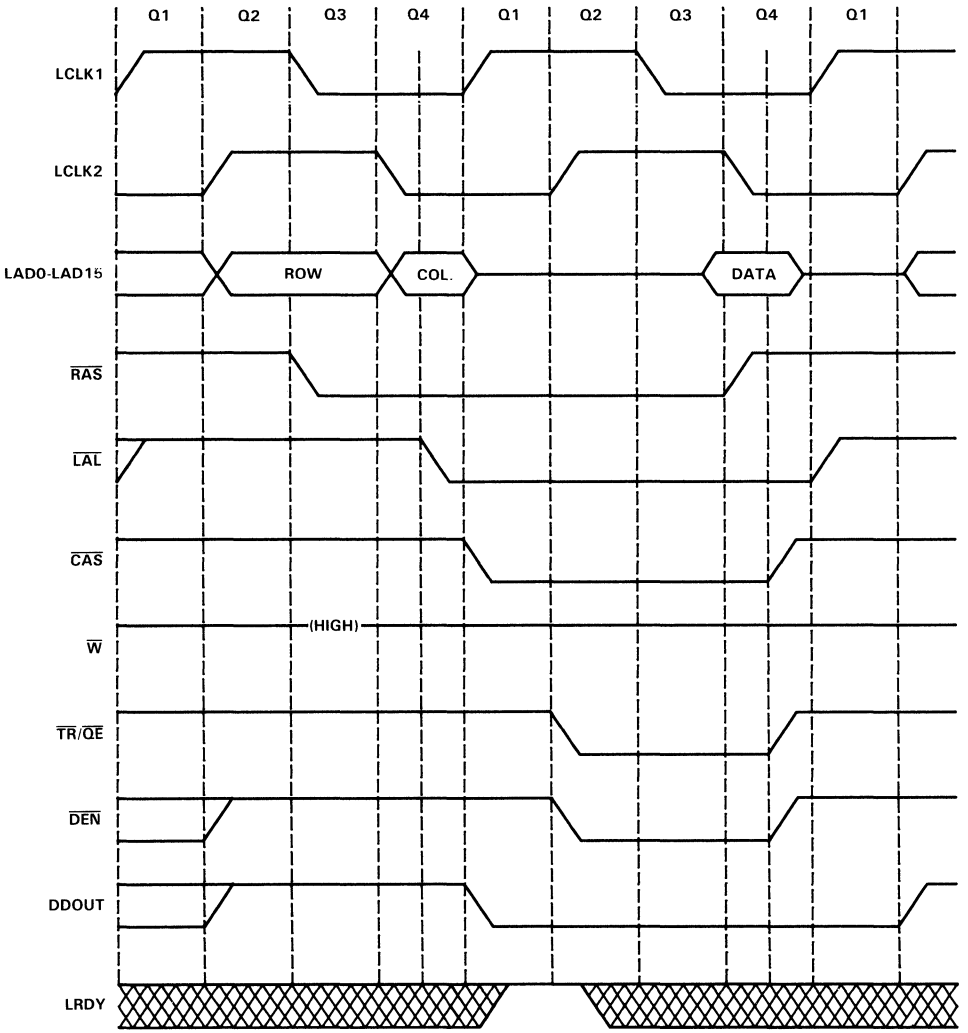
A hold/hold acknowledge capability is also built into the local memory interface to allow external devices to request control of the bus. After acknowledging a hold request, the TMS34010 releases the bus by driving its address/data bus and control outputs into high impedance.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

write cycle timing

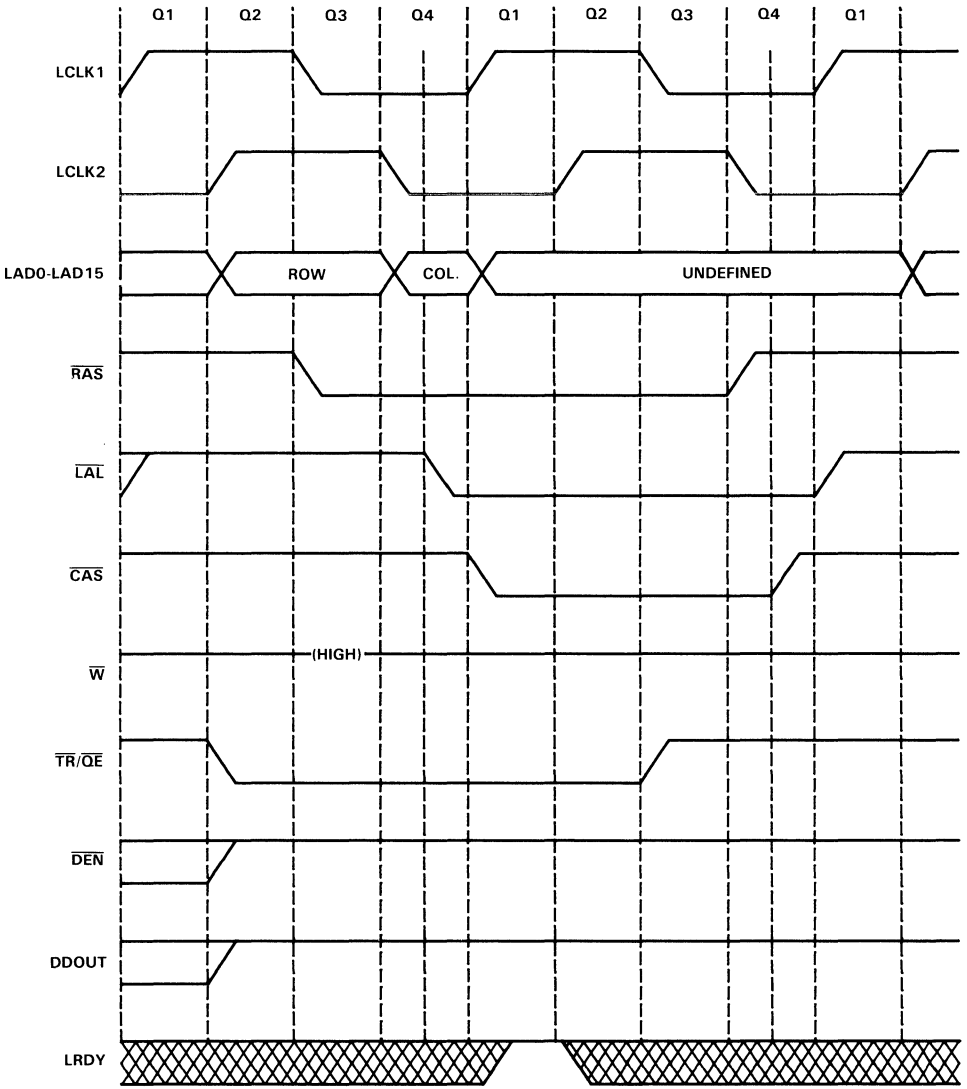


read cycle timing

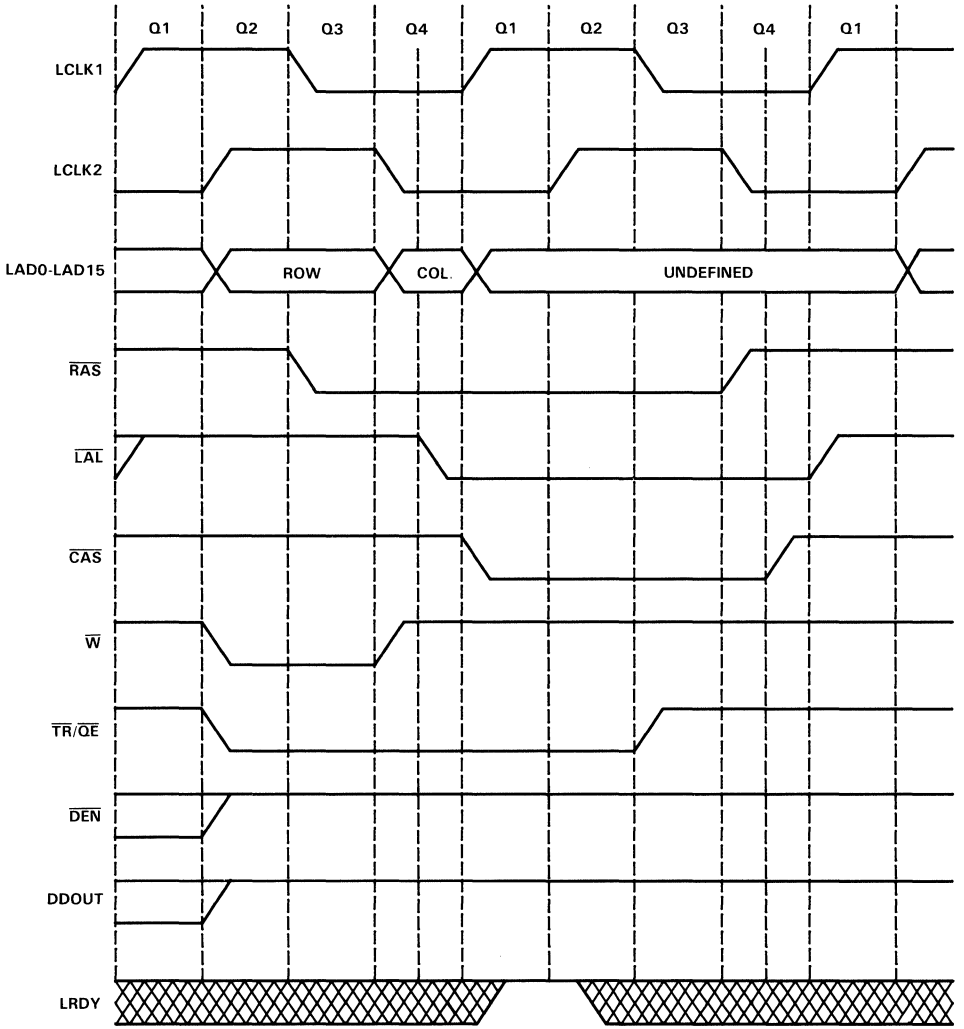


TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

memory-to-register cycle timing

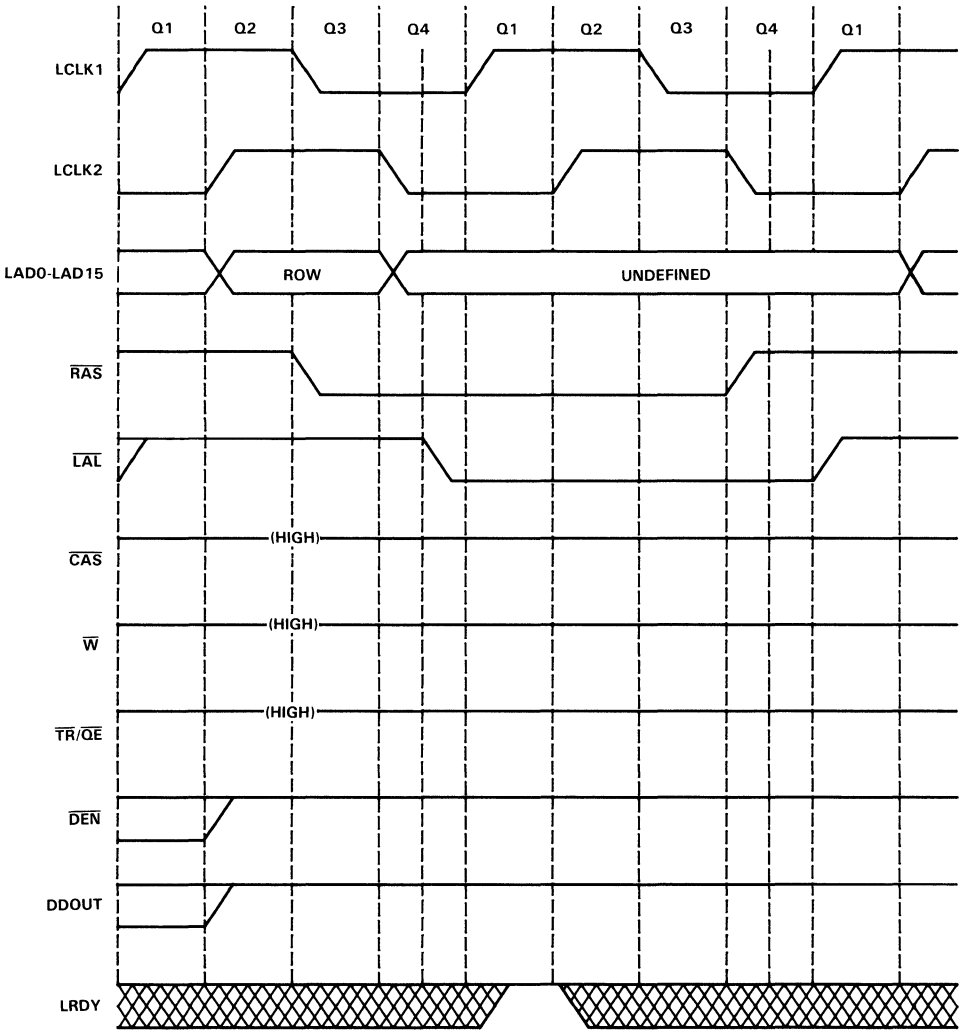


register-to-memory cycle timing

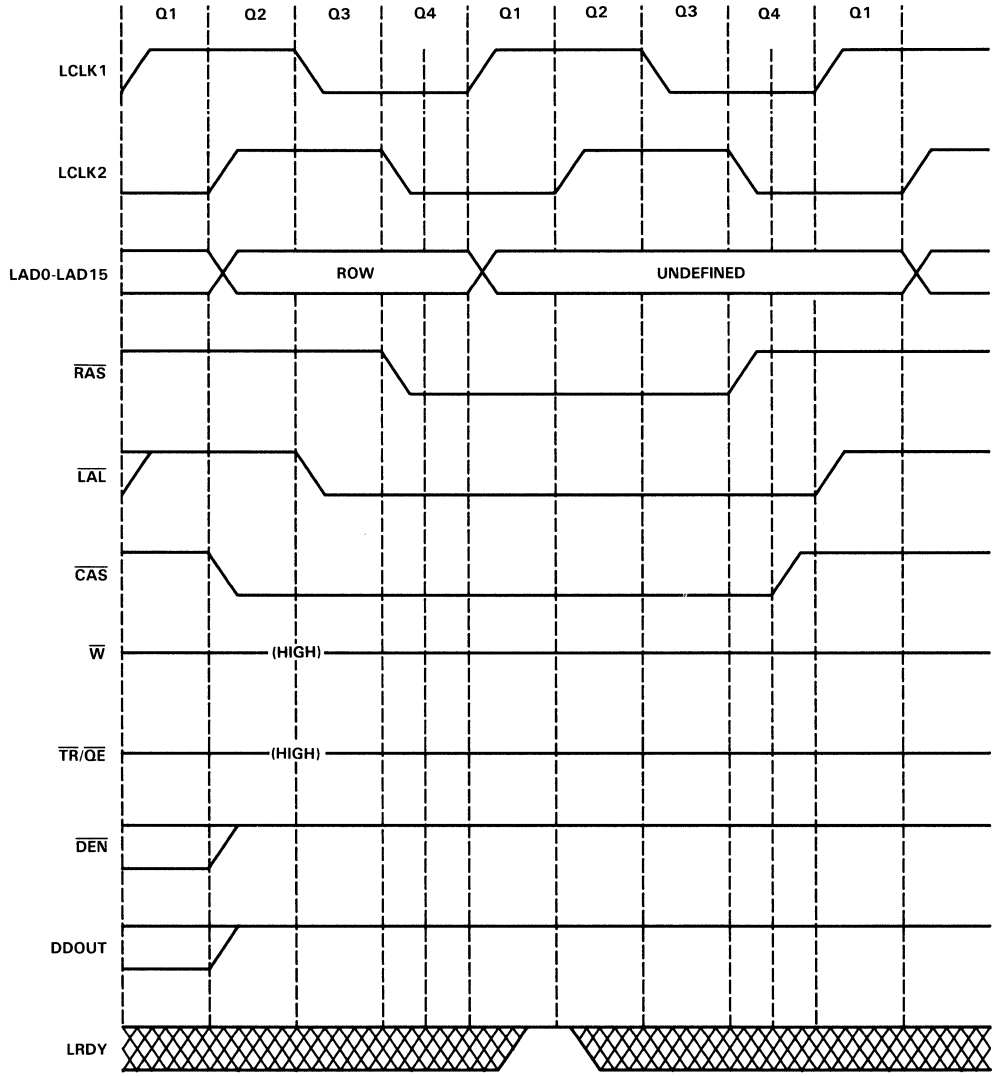


TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

$\overline{\text{RAS}}$-only DRAM refresh cycle timing

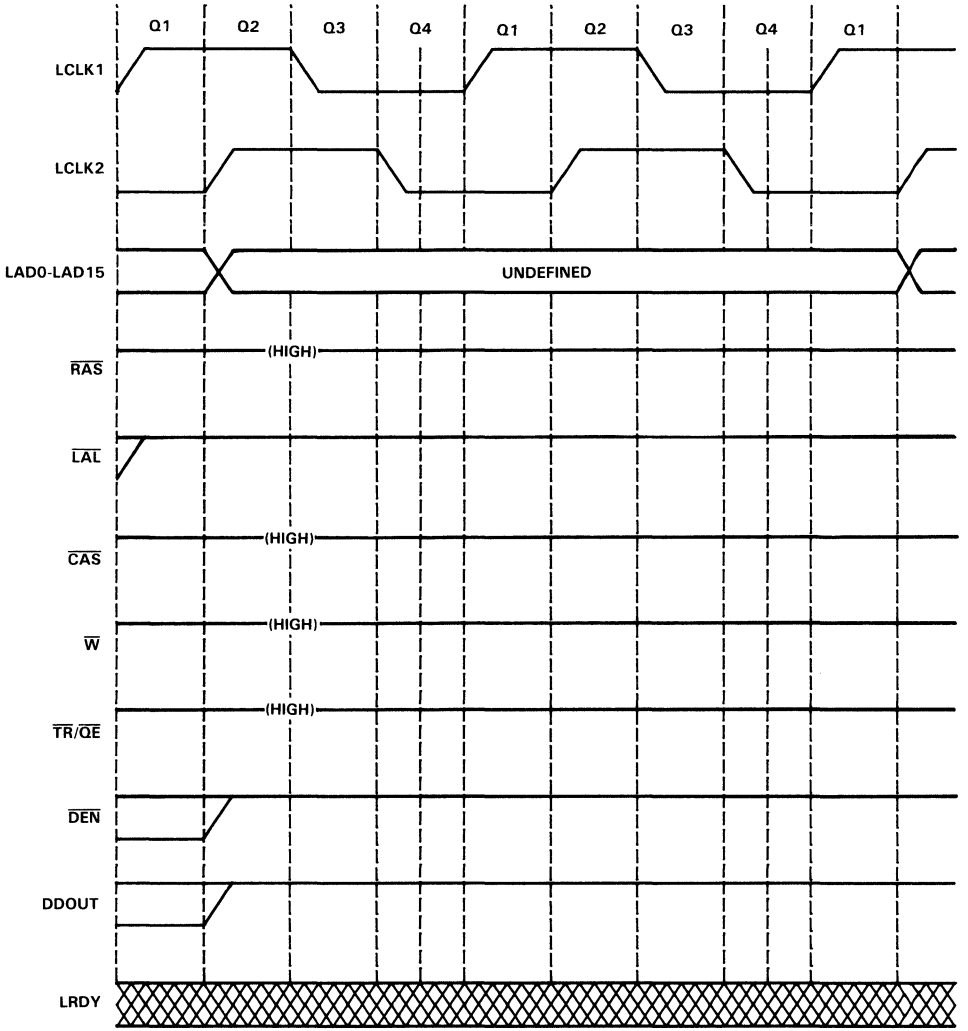


$\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ refresh cycle timing

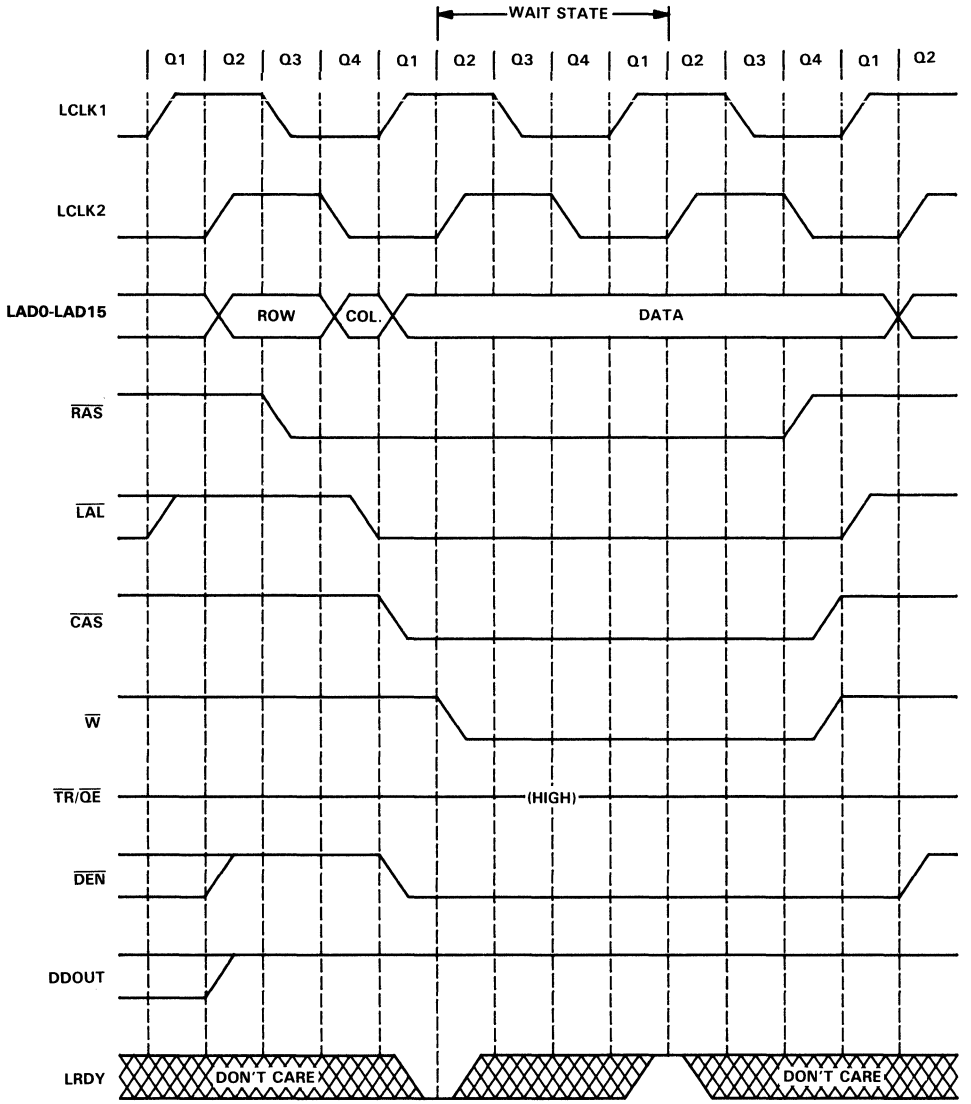


TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

internal cycles back to back

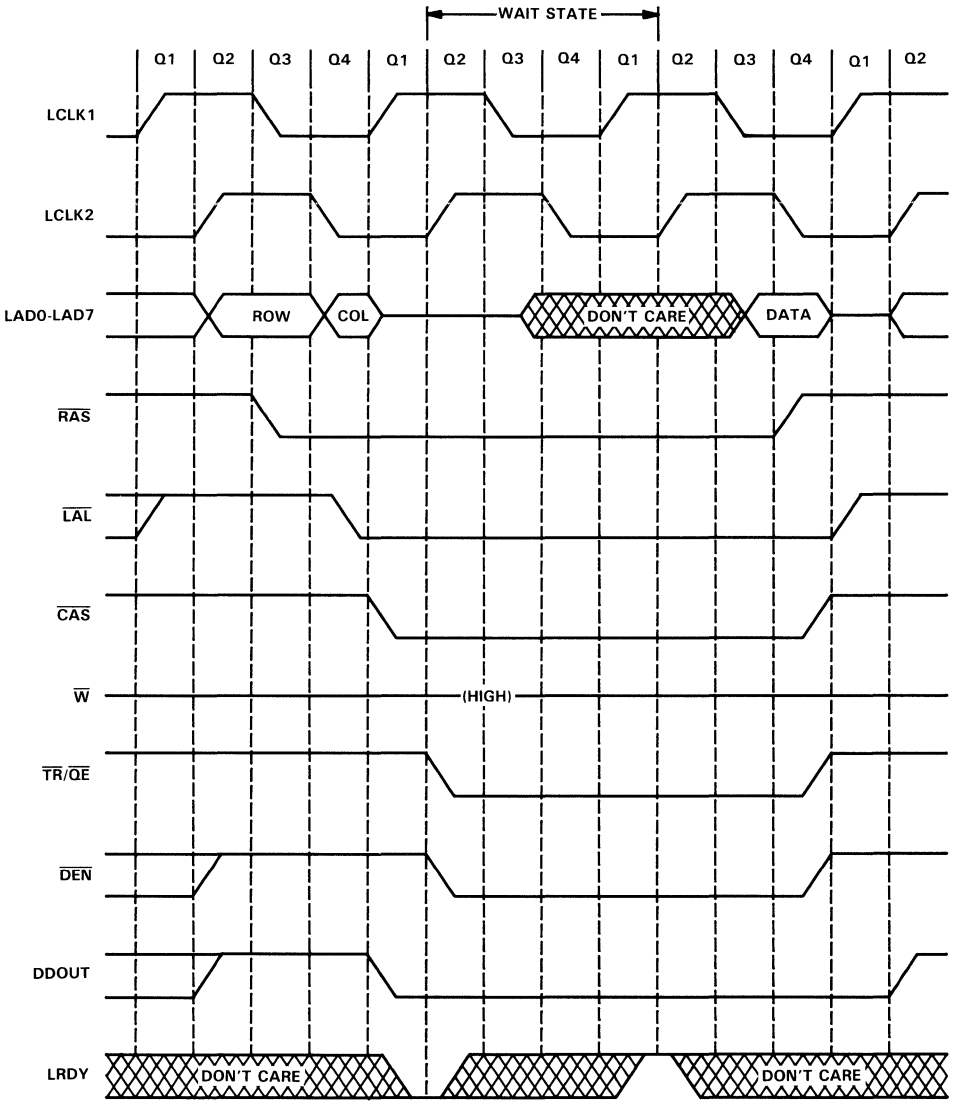


write cycle with one wait state

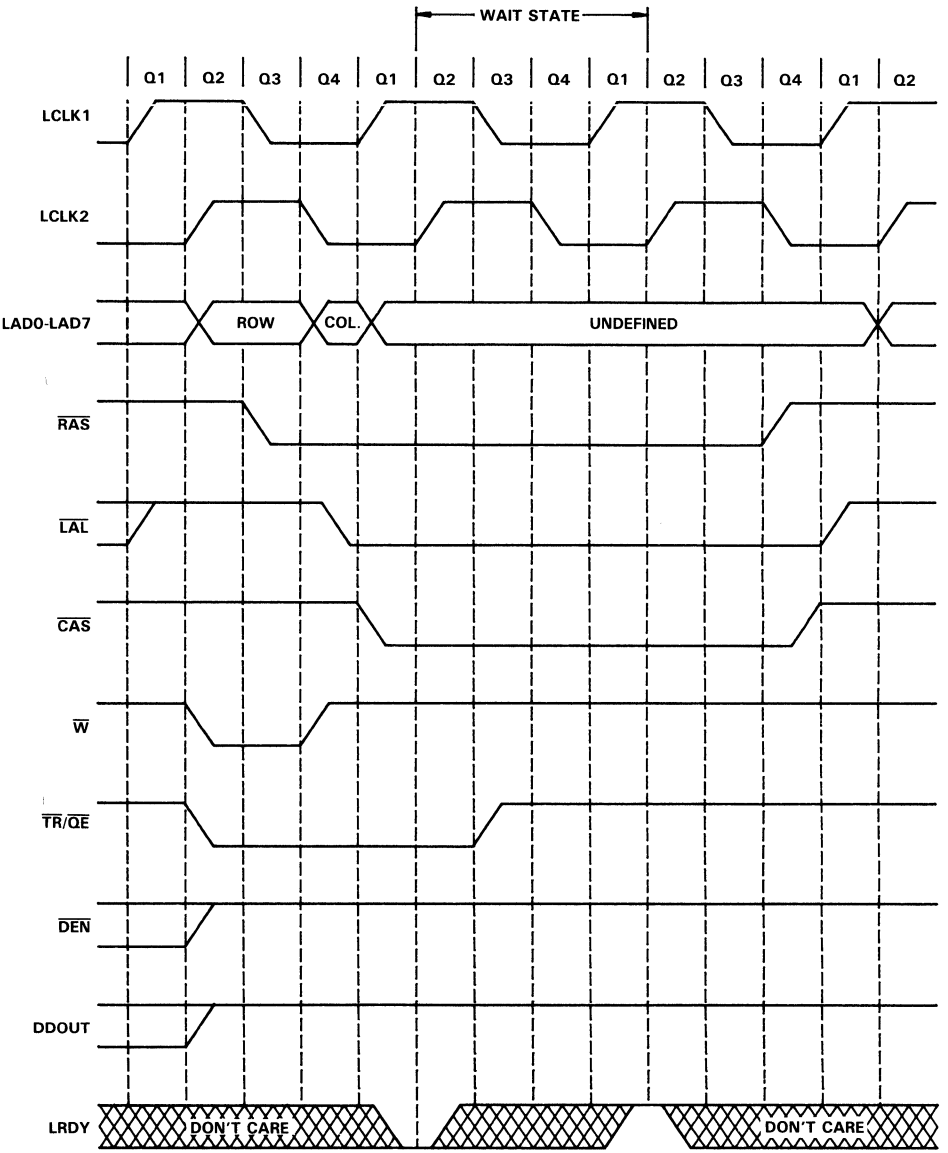


TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

read cycle with one wait state



register-to-memory cycle with one wait state



TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

absolute maximum ratings over operating free-air temperature range†

Supply voltage, V_{CC} . . . . . 7 V
Input voltage range . . . . . −0.3 V to 20 V
Off-state output voltage range . . . . . −2 V to 7 V
Operating free-air temperature range . . . . . 0°C to 70°C
Storage temperature range . . . . . −10°C to 150°C

†Stresses beyond those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only. Functional operation of the device at these or any other conditions beyond those indicated in the "Recommended Operating Conditions" section of this specification is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability. Voltage values are with respect to the V_{SS} pins of the chip.

recommended operating conditions

		MIN	NOM	MAX	UNIT
V_{CC}	Supply voltage	4.75	5.0	5.25	V
V_{SS}	Supply voltage†	0	0	0	V
I_{OH}	High-level output current			−400	μA
I_{OL}	Low-level output current			2.0‡	mA
T_A	Operating free-air temperature	0		70	°C

†Care should be taken by card designers to provide a minimum inductance path between the V_{SS} pins and system ground in order to minimize V_{SS} noise.
‡Output current of 2.0 mA is sufficient to drive five low-power Schottky TTL loads or 10 advanced low-power Schottky TTL loads (worst case).

DC electrical characteristics

PARAMETER			TEST CONDITIONS		MIN†	TYP‡	MAX†	UNIT
V_{IH}§	High-level input voltage, TTL-level signal	All inputs except INCLK	V_{CC} = 5.0 V		2.2		V_{CC}+0.3	V
		INCLK			3.0		V_{CC}+0.3	
V_{IL}	Low-level input voltage, TTL-level signal	All inputs except INCLK			−0.3		0.8	V
		INCLK			−0.3		0.8	
V_{OH}	High-level output voltage, TTL-level signal		V_{CC} = min, I_{OH} = max,		2.6			V
V_{OL}	Low-level output voltage, TTL-level signal		V_{CC} = max, I_{OL} = max,				0.6	V
I_O	High-impedance leakage current, bidirectional pins		V_{CC}=max	V_O=2.8 V			20	μA
				V_O=0.6 V			−20	
I_I	Input current	All inputs except RUN/$\overline{EMU}$§	V_I=V_{SS} to V_{CC}				±20	μA
I_{CC}	Supply current		V_{CC} = max, 40 MHz				125	mA
			V_{CC} = max, 50 MHz				150	
			V_{CC} = max, 60 MHz				175	
C_I	Input capacitance					10		pF
C_O	Output capacitance (except address/data lines)					10		pF

†For conditions shown as "min" or "max," use the appropriate value specified under "Recommended Operating Conditions."
‡All typical values are at V_{CC}=5 V, T_A=25°C.
§RUN/$\overline{EMU}$ will be no-connected in a typical configuration. The nominal pull-up current will be 250 μA.

ADVANCE INFORMATION

NOTE

Advance information notices apply only to the TMS34010-60.

signal transition levels

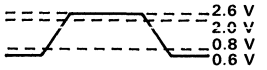


FIGURE 8. TTL-LEVEL OUTPUTS

TTL-level outputs are driven to a minimum logic-high level of 2.6 volts and to a maximum logic-low level of 0.6 volts. Output transition times are specified as follows.

For a high-to-low transition on a TTL-compatible output signal, the level at which the output is said to be "no longer high" is 2.0 volts, and the level at which the output is said to be "low" is 0.8 volts. For a low-to-high transition, the level at which the output is said to be "no longer low" is 0.8 volts, and the level at which the output is said to be "high" is 2.0 volts.

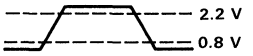
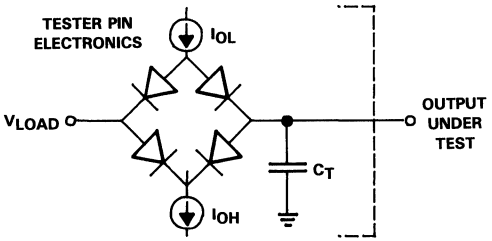


FIGURE 9. TTL-LEVEL INPUTS

Transition times for TTL-compatible inputs are specified as follows. For a high-to-low transition on an input signal, the level at which the input is said to be "no longer high" is 2.2 volts, and the level at which the input is said to be "low" is 0.8 volts. For a low-to-high transition on an input signal, the level at which the input is said to be "no longer low" is 0.8 volts, and the level at which the input is said to be "high" is 2.2 volts.

test measurement

The test load circuit shown in Figure 10 represents the programmable load of the tester pin electronics, which are used to verify timing parameters of TMS34010 output signals.



Where: I_{OL} = 2.0 mA DC level verification (all outputs)
I_{OH} = 400 μA (all outputs)
V_{LOAD} = 1.5 V DC level verification
0.7 V Timing verification
C_T = 65 pF typical load circuit capacitance

FIGURE 10. TEST LOAD CIRCUIT

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

timing parameter symbology

Timing parameter symbols have been created in accordance with JEDEC Standard 100. In order to shorten the symbols, some of the pin names and other related terminology have been abbreviated as follows:

AL	$\overline{\text{LAL}}$	HS	$\overline{\text{HSYNC}}$ or $\overline{\text{VSYNC}}$
C	$\overline{\text{CAS}}$	ICK	INCLK
CA	Column address	LR	$\overline{\text{LRDY}}$
CK	LCLK1 and LCLK2	QE	$\overline{\text{TR}}/\overline{\text{QE}}$, when used as output enable
CK1	LCLK1	R	$\overline{\text{RAS}}$
CK2	LCLK2	RA	Row address
CS	$\overline{\text{HCS}}$	RS	$\overline{\text{HREAD}}$
D	Data	RY	$\overline{\text{HRDY}}$
DD	DDOUT	S	$\overline{\text{HREAD}}$ or $\overline{\text{HWRITE}}$
EN	$\overline{\text{DEN}}$	TR	$\overline{\text{TR}}/\overline{\text{QE}}$, when used as shift register enable
F	$\overline{\text{HFS0}}$, $\overline{\text{HFS1}}$	VCK	VCLK
HK	$\overline{\text{HLDA}}/\overline{\text{EMUA}}$	W	$\overline{\text{W}}$
HR	$\overline{\text{HOLD}}$	WS	$\overline{\text{HWRITE}}$

Lowercase subscripts and their meaning are:

a	access time
c	cycle time (period)
d	delay time
h	hold time
su	setup time
t	transition time
w	pulse duration (width)

The following additional letters and symbols and their meaning are:

H	High
L	Low
V	Valid
Z	High impedance
↑	No longer low
↓	No longer high

host interface timing parameters

The timing parameters for host interface signals are shown in the next four figures. The purpose of these figures and the accompanying table is to quantify the timing relationships among the various signals. The explanation of the logical relationships among signals will be found in the *TMS34010 User's Guide* (number SPVU001A).

The *write strobe* referred to in the following table is the enabling signal during a write to one of the host interface registers (see comment 2 on the next page). Similarly, the *read strobe* is the enabling signal during a read.

Quarter clock time t_Q, which appears in the following table, is one quarter of a local output clock period, or twice the input clock period, $t_{c(ICK)}$.

NO.		PARAMETER	TMS34010-40		TMS34010-50 TMS34010-60		UNIT
			MIN	MAX	MIN	MAX	
1	$t_{su(FV-SL)}$	Setup time of $\overline{HWRITE}/\overline{HREAD}$ high or HFS0, HFS1 valid to read or write strobe↓	10		10		ns
2	$t_{d(WSL-DV)}$	Delay from write strobe ↓ to data in valid, write cycle		$2t_Q$		$2t_Q$	ns
3	$t_{d(SL-SL)}$	Delay from read or write strobe low to next read or write strobe↓	$7t_Q+10$		$7t_Q+10$		ns
4	$t_{w(SL)}$	Duration of read or write strobe low	80		80		ns
5	$t_{d(SH-SL)}$	Delay from read or write strobe high to next read or write strobe↓	60		60		ns
6	$t_{h(WSH-DV)}$	Hold time of data in valid after write strobe high, write cycle	10		10		ns
7	$t_{h(SH-FV)}$	Hold time of $\overline{HWRITE}/\overline{HREAD}$ high or HFS0, HFS1 valid after read or write strobe high	10		10		ns
8	$t_{h(RSL-DZ)}$	Hold time of data high impedance after read strobe↓, read cycle	0§		0§		ns
9	$t_{d(RSL-DV)}$	Delay from read strobe low to data out valid, read cycle with no wait		90		90	ns
10	$t_{h(RSH-DV)}$	Hold time of data out valid after read strobe↑, read cycle	0		0		ns
11	$t_{d(RSH-DZ)}$	Delay from read strobe high to data out high impedance, read cycle		30§		30§	ns
12	$t_{h(CSL-RYH)}$	Hold time of HRDY high after $\overline{HCS}$↓, cycle with wait	0		0		ns
13	$t_{d(CSL-RYL)}$	Delay from $\overline{HCS}$ low to HRDY low, cycle with wait		40		40	ns
14	$t_{w(RYL)}$	Pulse duration of HRDY low, cycle with wait		†		†	ns
15	$t_{d(RYL-RYH)}$	Delay from HRDY↓ to HRDY high, cycle with wait	0‡		0‡		ns
16	$t_{h(RYH-WSL)}$	Hold time of write strobe low after HRDY↑, write cycle with wait	40		40		ns
17	$t_{d(RYH-DV)}$	Delay from HRDY↑ to data out valid, read cycle with wait		40		40	ns
18	$t_{h(RYH-RSL)}$	Hold time of read strobe low after HRDY↑, read cycle with wait	40		40		ns

NOTE: Advance information notices apply only to the TMS34010-60.
†Parameter 14 is a function of local bus memory contention. This parameter is not tested. Refer to the *TMS34010 User's Guide* for details.
‡Parameter 15 is specified as minimum 0 ns to indicate that a low-going pulse on HRDY can be arbitrarily narrow.
§These values are derived from characterization and are not tested.

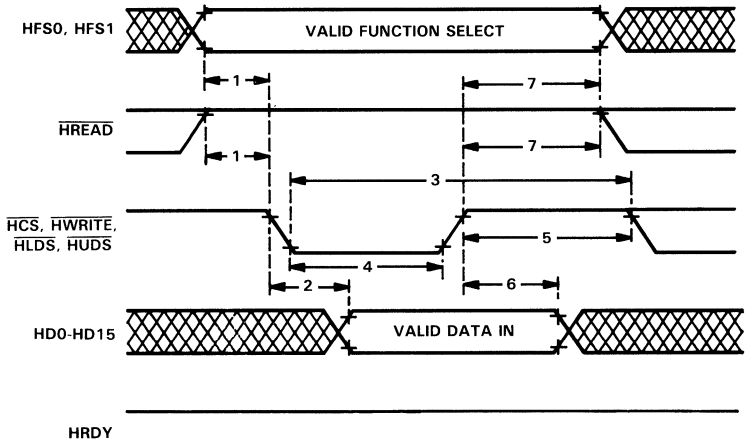
TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

general comments on host interface timing

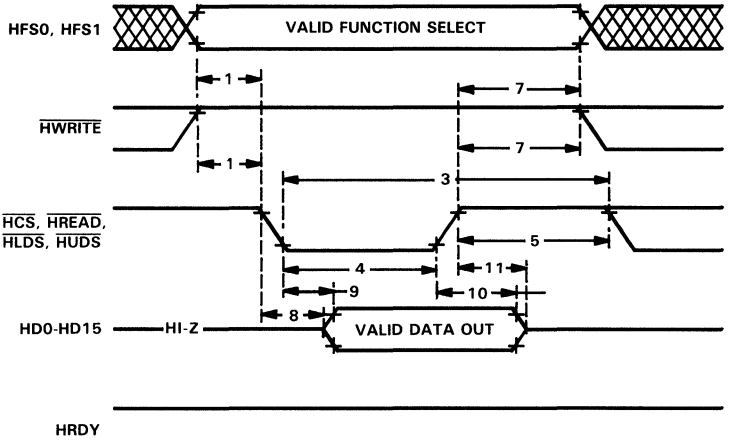
The following general comments apply to host interface timing:

1. The HRDY signal is enabled by an active-low level on the $\overline{\text{HCS}}$ input. When $\overline{\text{HCS}}$ is inactive-high, HRDY is forced high regardless of the internal state of the device. Low-going transient pulses on $\overline{\text{HCS}}$ may result in low-going transient pulses on HRDY, but otherwise have no effect unless accompanied by active levels on other control signals.
2. A host interface write cycle occurs when $\overline{\text{HCS}}$, $\overline{\text{HWRITE}}$, and $\overline{\text{HLDS}}$ are low, or when $\overline{\text{HCS}}$, $\overline{\text{HWRITE}}$, and $\overline{\text{HUDS}}$ are low. The combination of these signals defines a *write strobe*. In either case, the last of the three signals to make the high-to-low transition is the strobe (write strobe) that begins the cycle. The first of the three signals to make the low-to-high transition ends the cycle. Similarly, a host interface read cycle occurs when $\overline{\text{HCS}}$, $\overline{\text{HREAD}}$, and $\overline{\text{HLDS}}$ are low, or when $\overline{\text{HCS}}$, $\overline{\text{HREAD}}$, and $\overline{\text{HUDS}}$ are low. The combination at these signals define a *read strobe*. In either case, the last of the three signals to make the high-to-low transition is the strobe (read strobe) that begins the cycle. The first of the three signals to make the low-to-high transition ends the cycle. All access times are specified with respect to the strobing edges that begin and end the cycle.
3. During a host interface read or write, $\overline{\text{HWRITE}}$ and $\overline{\text{HREAD}}$ must not be active-low simultaneously.
4. Host interface input signals $\overline{\text{HCS}}$, $\overline{\text{HUDS}}$, $\overline{\text{HLDS}}$, HFS0, HFS1, $\overline{\text{HREAD}}$, and $\overline{\text{HWRITE}}$ are assumed to be asynchronous with respect to the output clocks LCLK1 and LCLK2.

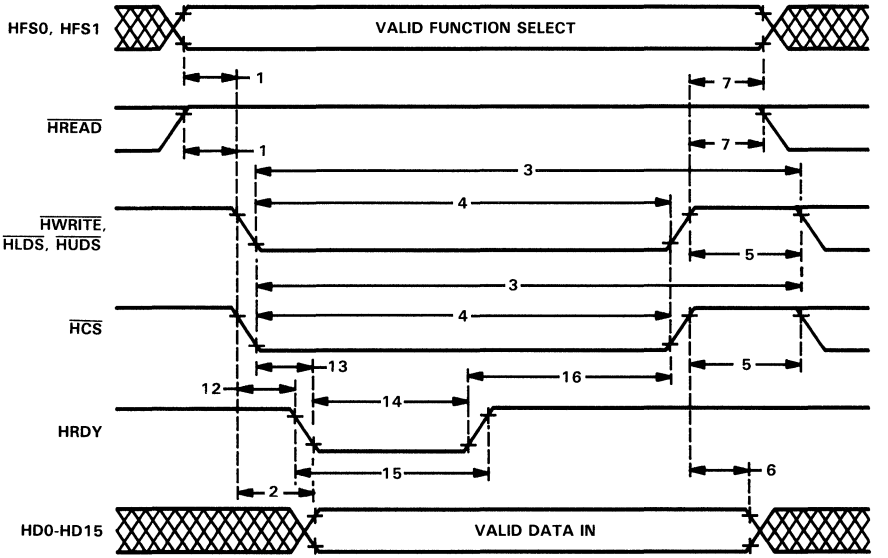
host interface timing: write cycle with no wait



host interface timing: read cycle with no wait

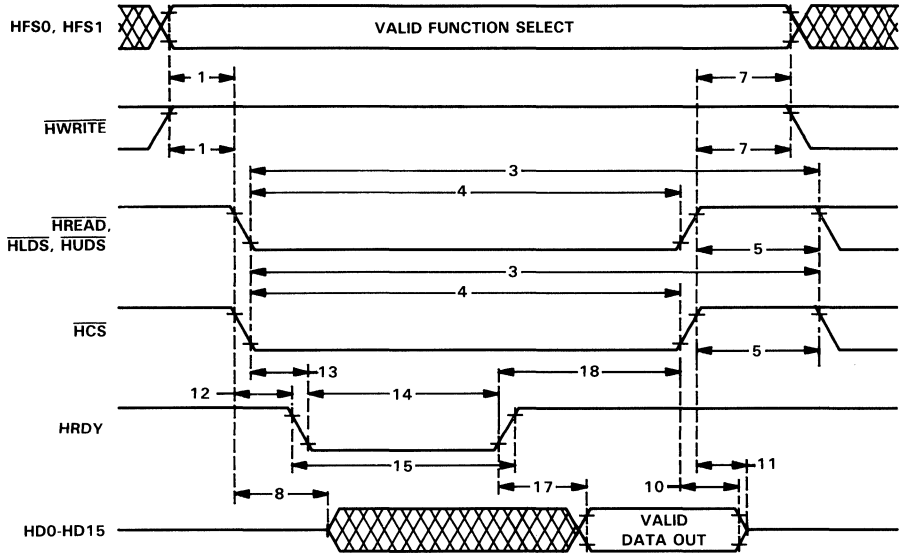


host interface timing: write cycle with wait



TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

host interface timing: read cycle with wait



reset timing

The timing parameters for device reset are shown in the next two figures. The purpose of these figures is to quantify the timing relationships among the $\overline{\text{RESET}}$, $\overline{\text{HCS}}$, and LCLK1 signals. $\overline{\text{RESET}}$ and $\overline{\text{HCS}}$ are asynchronous inputs that are internally synchronized by latches internal to the TMS34010. The timing relationships specified for these signals relative to LCLK1 need be met only to guarantee recognition of a transition of one of these signals at a particular clock edge. The explanation of the logical relationships among signals will be found in the *TMS34010 User's Guide*.

Quarter clock time t_Q which appears in the following table, is one quarter of a local output clock period, or twice the input clock period, $t_{c(ICK)}$.

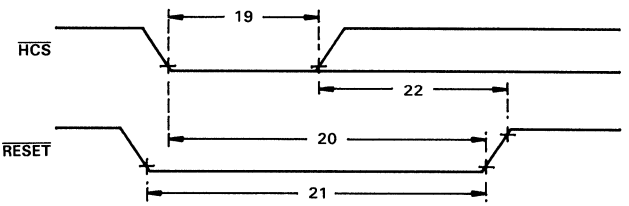
NO.	PARAMETER		TMS34010-40		TMS34010-50 TMS34010-60		UNIT
			MIN	MAX	MIN	MAX	
19	$t_{w(CSL)}$	Duration of $\overline{\text{HCS}}$ low to configure GSP to run in self-bootstrap mode	$4t_Q+55$		$4t_Q+55$		ns
20	$t_{su(CSL-REH)}$	Setup time of $\overline{\text{HCS}}$ low to $\overline{\text{RESET}}\uparrow$ to configure the GSP to run in self-bootstrap mode	$8t_Q+55$		$8t_Q+55$		ns
21	$t_{w(REL)}$	Duration of $\overline{\text{RESET}}$ low to ensure that GSP is properly reset	$160t_Q-40$		$160t_Q-40$		ns
22	$t_{d(CSH-REH)}$	Delay from $\overline{\text{HCS}}\uparrow$ to $\overline{\text{RESET}}$ high, end of reset, to configure GSP to run in self-bootstrap mode		$4t_Q-50$†		$4t_Q-50$†	ns
23	$t_{su(REV-CK1L)}$	Setup time of $\overline{\text{RESET}}$ valid to LCLK1↓ to guarantee recognition at a particular clock edge	40‡		40‡		ns
24	$t_{h(CK1L-REV)}$	Hold time of $\overline{\text{RESET}}$ valid after LCLK1 low to guarantee recognition at a particular clock edge	10‡		10‡		ns
25	$t_{su(CSV-CK1L)}$	Setup time of $\overline{\text{HCS}}$ valid to LCLK1↓ to guarantee recognition at a particular clock edge	40‡		40‡		ns
26	$t_{h(CK1L-CSV)}$	Hold time of $\overline{\text{HCS}}$ valid after LCLK1 low to guarantee recognition at a particular clock edge	10‡		10‡		ns

NOTE: Advance information notices apply only to the TMS34010-60.

†Parameter 22 is the maximum amount by which the $\overline{\text{RESET}}$ low-to-high transition can be delayed after the $\overline{\text{HCS}}$ low-to-high transition and still guarantee that the GSP is configured to run in self-bootstrap mode (HLT bit = 0) following the end of reset. $\overline{\text{HCS}}$ may be held low for some time past the low-to-high $\overline{\text{RESET}}$ transition, and will be ignored by the GSP for 17 local clock periods following the clock edge at which the low-to-high $\overline{\text{RESET}}$ transition is detected. Following completion of the eight $\overline{\text{RAS}}$-only cycles that automatically follow reset, however, a low $\overline{\text{HCS}}$ level will be interpreted as a chip select.

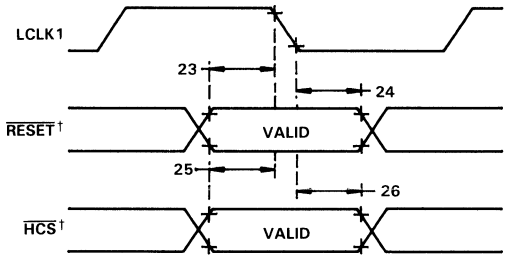
‡$\overline{\text{RESET}}$ and $\overline{\text{HCS}}$ are asynchronous inputs. The specified setup and hold times of these signals with respect to the high-to-low transition of LCLK1 need be met only to guarantee that a transition of $\overline{\text{RESET}}$ or $\overline{\text{HCS}}$ is detected by the device at a particular clock edge.

reset: asynchronous timing relationships



TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

reset: synchronous timing relationships



†$\overline{\text{RESET}}$ and $\overline{\text{HCS}}$ are asynchronous inputs. The specified setup and hold times of $\overline{\text{RESET}}$ or $\overline{\text{HCS}}$ with respect to the high-to-low LCLK1 transition must be met only to guarantee that a $\overline{\text{RESET}}$ or $\overline{\text{HCS}}$ transition is detected by the device at a particular clock edge.

local bus timing parameters

The following six figures show the timing parameters for the signals of the local memory interface bus, often simply referred to as the local bus. The purpose of these figures and the accompanying tables is to quantify the timing relationships among the various signals. The explanation of the logical relationships among signals will be found in the *TMS34010 User's Guide* (number SPVU001).

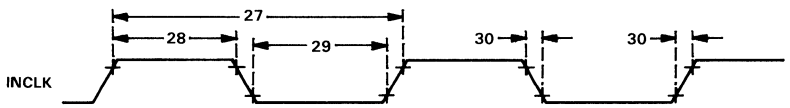
A number of parameter values are expressed in terms of quarter clock time t_Q, which is one quarter of a local clock period, or twice the input clock period, $t_{c(ICK)}$.

Input clock INCLK is divided internally by 8 to produce output clocks LCLK1 and LCLK2. Transitions of the other local interface output signals are also generated as delays from INCLK transitions. The divide-down logic that converts INCLK to the internal clocks used to generate LCLK1 and LCLK2 introduces significant propagation delays from the transitions of INCLK to the corresponding transitions of LCLK1 and LCLK2. While the frequency of INCLK is precisely eight times the frequency of LCLK1 or LCLK2, no timing relationship other than the frequency is specified between transitions of input clock INCLK and transitions of the output clocks LCLK1 and LCLK2.

NO.	PARAMETER		TMS34010-40		TMS34010-50		TMS34010-60		UNIT
			MIN	MAX	MIN	MAX	MIN	MAX	
27	$t_{c(ICK)}$	Period of INCLK	25	62.5	20	62.5	16.5	62.5	ns
28	$t_{w(ICKH)}$	Pulse duration of INCLK high	8‡		8‡		6.5‡		ns
29	$t_{w(ICKL)}$	Pulse duration of INCLK low	8‡		8‡		6.5‡		ns
30	$t_{t(ICK)}$	Transition time (rise and fall) of INCLK	2†	8†	2†	8†	2†	8†	ns

NOTE: Advance information notices apply only to the TMS34010-60.
†These values are based on computer simulation and are not tested.
‡This pulse width is tested at 1.4 volts.

local bus timing: input clock



local bus timing parameters (continued)

Quarter clock time t_Q, which appears in the following table, is one quarter of a local output clock period, or $2t_{c(ICK)}$.

NO.	PARAMETER		TMS34010-40		TMS34010-50 TMS34010-60		UNIT
			MIN	MAX	MIN	MAX	
31	$t_{c(CK)}$	Period of local clocks LCLK1 and LCLK2	$8t_{c(ICK)}$†		$8t_{c(ICK)}$†		ns
32	$t_{w(CKH)}$	Pulse duration of local clock high	$2t_Q-10$		$2t_Q-10$		ns
33	$t_{w(CKL)}$	Pulse duration of local clock low	$2t_Q-10$		$2t_Q-10$		ns
34	$t_{h(CK1H-CK2L)}$	Hold time of LCLK2 low after LCLK1 high	t_Q-10		t_Q-10		ns
35	$t_{h(CK1L-CK2H)}$	Hold time of LCLK2 high after LCLK1 low	t_Q-10		t_Q-10		ns
36	$t_{h(CK2H-CK1H)}$	Hold time of LCLK1 high after LCLK2 high	t_Q-10		t_Q-10		ns
37	$t_{h(CK2L-CK1L)}$	Hold time of LCLK1 low after LCLK2 low	t_Q-10		t_Q-10		ns
38	$t_{h(CK1H-CK2H)}$	Hold time of LCLK2 high after LCLK1 high	$3t_Q-10$		$3t_Q-10$		ns
39	$t_{h(CK1L-CK2L)}$	Hold time of LCLK2 low after LCLK1 low	$3t_Q-10$		$3t_Q-10$		ns
40	$t_{h(CK2H-CK1L)}$	Hold time of LCLK1 low after LCLK2 high	$3t_Q-10$		$3t_Q-10$		ns
41	$t_{h(CK2L-CK1H)}$	Hold time of LCLK1 high after LCLK2 low	$3t_Q-10$		$3t_Q-10$		ns
42	t_t	Transition time (rise and fall) of LCLK1 or LCLK2		10		10	ns
43	$t_{su(RAV-CK2H)}$	Setup time of row address valid to LCLK2↑	$4t_Q-25$		$4t_Q-15$		ns
44	$t_{su(CAV-CK2H)}$	Setup time of column address valid to LCLK2↑	$2t_Q-25$		$2t_Q-15$		ns
45	$t_{su(LRV-CK2H)}$	Setup time of LRDY valid to LCLK2↑	30‡		30‡		ns
46	$t_{h(CK2H-LRV)}$	Hold time of LRDY valid after LCLK2 high	0‡		0‡		ns
47	$t_{su(RAV-CK1L)}$	Setup time of row address valid to LCLK1↓	t_Q-25		t_Q-15		ns
48	$t_{su(CAV-CK1H)}$	Setup time of column address valid to LCLK1↑	t_Q-25		t_Q-15		ns
49	$t_{su(ALH-CK1L)}$	Setup time of $\overline{LAL}$ high to LCLK1↓	$2t_Q-20$		$2t_Q-10$		ns

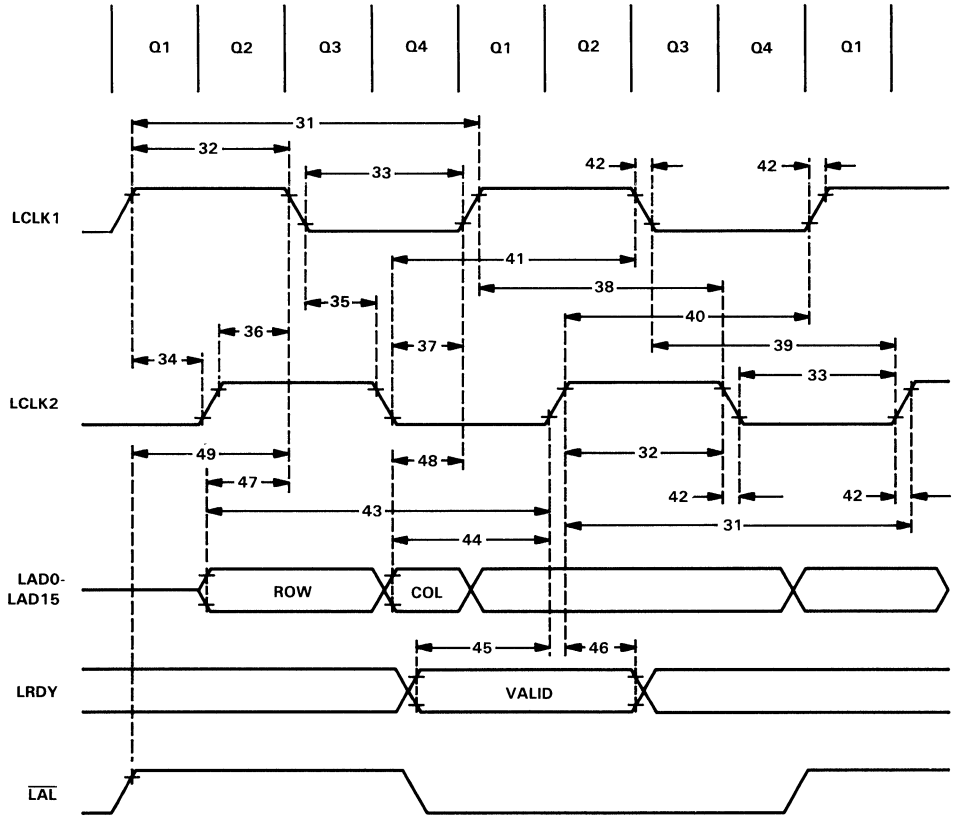
NOTE: Advance information notices apply only to the TMS34010-60.

†This is a functional minimum and is not tested. This parameter can also be specified as $4t_Q$.

‡LRDY is a synchronous input sampled during the low-to-high transition of LCLK2. The specified setup and hold times must be met for the device to operate properly.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

local bus timing: output clock and LRDY signal



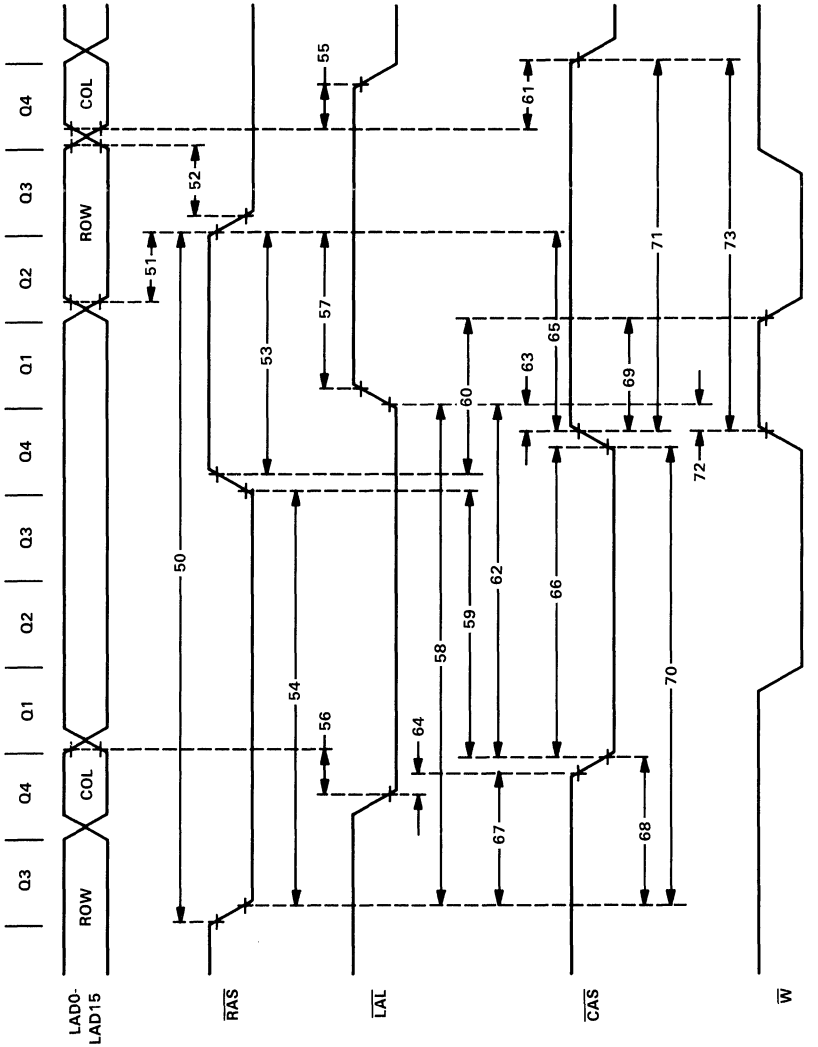
NO.	PARAMETER		TMS34010-40		TMS34010-50		TMS34010-60		UNIT
			MIN	MAX	MIN	MAX	MIN	MAX	
50	$t_{d(RL-RL)}$	Delay from $\overline{RAS}\downarrow$ to $\overline{RAS}\downarrow$	$8t_Q$†		$8t_Q$†		$8t_Q$†		ns
51	$t_{su(RAV-RL)}$	Setup time of row address valid to $\overline{RAS}\downarrow$	t_Q-20		t_Q-15		t_Q-15		ns
52	$t_{h(RL-RAV)}$	Hold time of row address valid after $\overline{RAS}$ low	t_Q-20		t_Q-10		t_Q-5		ns
53	$t_{w(RH)}$	Pulse duration, $\overline{RAS}$ high	$3t_Q-20$		$3t_Q-10$		$3t_Q-5$		ns
54	$t_{w(RL)}$	Pulse duration, $\overline{RAS}$ low	$5t_Q-20$		$5t_Q-10$		$5t_Q-10$		ns
55	$t_{su(CAV-ALL)}$	Setup time of column address valid to $\overline{LAL}\downarrow$	$0.5t_Q-20$		$0.5t_Q-10$		$0.5t_Q-10$		ns
56	$t_{h(ALL-CAV)}$	Hold time of column address valid after $\overline{LAL}$ low	$0.5t_Q-15$		$0.5t_Q-10$		$0.5t_Q-10$		ns
57	$t_{h(ALH-RH)}$	Hold time of $\overline{RAS}$ high after $\overline{LAL}$ high	$2t_Q-20$		$2t_Q-10$		$2t_Q-10$		ns
58	$t_{h(RL-ALL)}$	Hold time of $\overline{LAL}$ low after $\overline{RAS}$ low	$6t_Q-20$		$6t_Q-10$		$6t_Q-10$		ns
59	$t_{h(CL-RL)}$	Hold time of $\overline{RAS}$ low after $\overline{CAS}$ low	$3t_Q-20$		$3t_Q-10$		$3t_Q-10$		ns
60	$t_{h(RH-WH)}$	Hold time of $\overline{W}$ high after after $\overline{RAS}$ high, shift register transfer follows read	$2t_Q-20$		$2t_Q-10$		$2t_Q-10$		ns
61	$t_{su(CAV-CL)}$	Setup time of column address valid to $\overline{CAS}\downarrow$	t_Q-20		t_Q-10		t_Q-10		ns
62	$t_{h(CL-ALL)}$	Hold time of $\overline{LAL}$ low after $\overline{CAS}$ low	$4t_Q-20$		$4t_Q-10$		$4t_Q-10$		ns
63	$t_{h(CH-ALL)}$	Hold time of $\overline{LAL}$ low after $\overline{CAS}$ high, write cycle	$0.5t_Q-15$		$0.5t_Q-10$		$0.5t_Q-10$		ns
64	$t_{h(ALL-CH)}$	Hold time of $\overline{CAS}$ high after $\overline{LAL}$ low	$0.5t_Q-15$		$0.5t_Q-10$		$0.5t_Q-10$		ns
65	$t_{h(CH-RH)}$	Hold time of $\overline{RAS}$ high after $\overline{CAS}$ high	$2.5t_Q-15$		$2.5t_Q-10$		$2.5t_Q-10$		ns
66	$t_{w(CL)}$	Pulse duration, $\overline{CAS}$ low	$3.5t_Q-25$		$3.5t_Q-10$		$3.5t_Q-10$		ns
67	$t_{h(RL-CH)}$	Hold time of $\overline{CAS}$ high after $\overline{RAS}$ low	$2t_Q-20$		$2t_Q-10$		$2t_Q-10$		ns
68	$t_{d(RL-CL)}$	Delay time from $\overline{RAS}$ low to $\overline{CAS}$ low	$2t_Q+20$		$2t_Q+10$		$2t_Q+10$		ns
69	$t_{h(CH-WH)}$	Hold time of $\overline{W}$ high after $\overline{CAS}$ high, shift register transfer follows read	$1.5t_Q-15$		$1.5t_Q-10$		$1.5t_Q-10$		ns
70	$t_{h(RL-CL)}$	Hold time of $\overline{CAS}$ low after $\overline{RAS}$ low	$5.5t_Q-25$		$5.5t_Q-10$		$5.5t_Q-10$		ns
71	$t_{w(CH)}$	Pulse duration, $\overline{CAS}$ high	$4.5t_Q-15$		$4.5t_Q-10$		$4.5t_Q-10$		ns
72	$t_{h(WH-ALL)}$	Hold time of $\overline{LAL}$ low after $\overline{W}$ high, write cycle	$0.5t_Q-15$		$0.5t_Q-10$		$0.5t_Q-10$		ns
73	$t_{su(WH-CL)}$	Setup time of $\overline{W}$ high to $\overline{CAS}\downarrow$, end of write	$4.5t_Q-15$		$4.5t_Q-10$		$4.5t_Q-10$		ns

NOTE: Advance information notices apply only to the TMS34010-60.

†This is a functional minimum and is not tested.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

local bus timing: the $\overline{RAS}$, $\overline{CAS}$, and $\overline{LAL}$ outputs



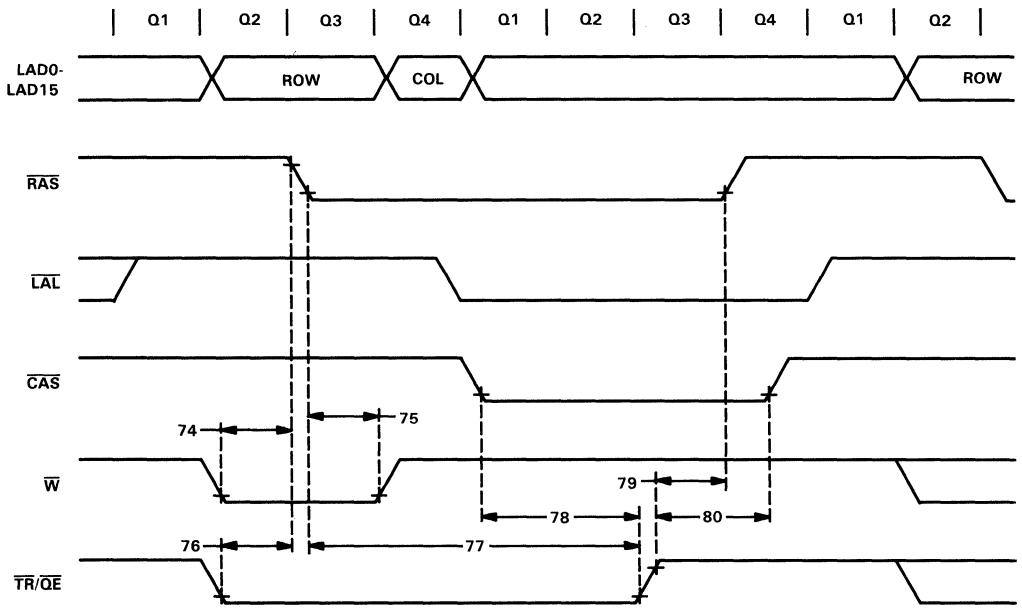
local bus timing parameters (continued)

Quarter clock time t_Q, which appears in the following table, is one quarter of a local output clock period, or $2t_{c(ICK)}$.

NO.	PARAMETER		TMS34010-40		TMS34010-50 TMS34010-60		UNIT
			MIN	MAX	MIN	MAX	
74	$t_{su(WL-RL)}$	Setup time of $\overline{W}$ low to $\overline{RAS}\downarrow$, shift register transfer cycle	t_Q-20		t_Q-10		ns
75	$t_{h(RL-WL)}$	Hold time of $\overline{W}$ low after $\overline{RAS}$ low, shift register transfer cycle	t_Q-20		t_Q-10		ns
76	$t_{su(TRL-RL)}$	Setup time of $\overline{TR/QE}$ low to $\overline{RAS}\downarrow$, shift register transfer cycle	t_Q-20		t_Q-10		ns
77	$t_{h(RL-TRL)}$	Hold time of $\overline{TR/QE}$ low after $\overline{RAS}$ low, shift register transfer cycle	$4t_Q-20$		$4t_Q-10$		ns
78	$t_{h(CL-TRL)}$	Hold time of $\overline{TR/QE}$ low after $\overline{CAS}$ low, shift register transfer cycle	$2t_Q-20$		$2t_Q-10$		ns
79	$t_{su(TRH-RH)}$	Setup time of $\overline{TR/QE}$ high to $\overline{RAS}\uparrow$, shift register transfer cycle	t_Q-20		t_Q-10		ns
80	$t_{su(TRH-CH)}$	Setup time of $\overline{TR/QE}$ high to $\overline{CAS}\uparrow$, shift register transfer cycle	$1.5t_Q-25$		$1.5t_Q-10$		ns

NOTES: 1. Advance information notices apply only to the TMS34010-60.
2. Parameters 81 and 82 intentionally omitted.

local bus timing parameters: shift register transfer cycle



TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

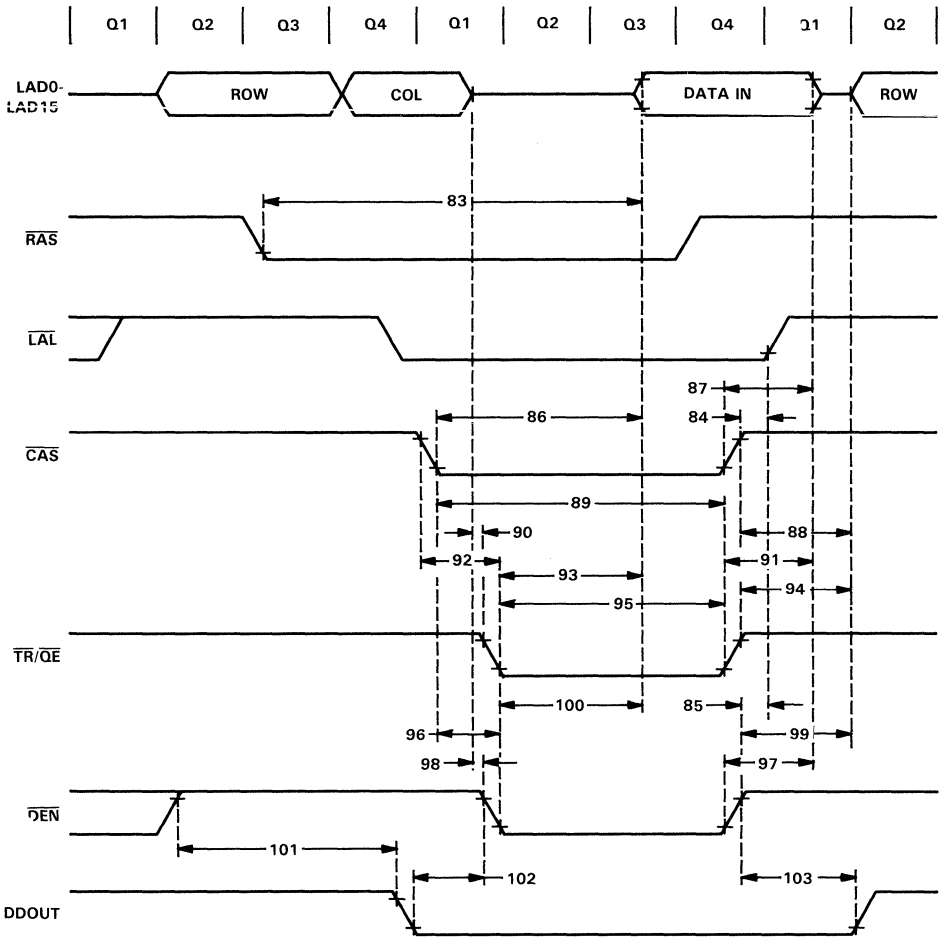
local bus timing parameters (continued)

Quarter clock time t_Q, which appears in the following table, is one quarter of a local output clock period, or $2t_{c(ICK)}$.

NO.	PARAMETER		TMS34010-40		TMS34010-50 TMS34010-60		UNIT
			MIN	MAX	MIN	MAX	
83	$t_{a(RL-DV)}$	Access time from $\overline{RAS}$ low to data in valid, read cycle		$5.5t_Q-40^\dagger$		$5.5t_Q-25^\dagger$	ns
84	$t_{su(CH-ALH)}$	Setup time of $\overline{CAS}$ high to $\overline{LAL}\uparrow$	$0.5t_Q-15$		$0.5t_Q-10$		ns
85	$t_{su(ENH-ALH)}$	Setup time of $\overline{DEN}$ high to $\overline{LAL}\uparrow$	$0.5t_Q-15$		$0.5t_Q-10$		ns
86	$t_{a(CL-DV)}$	Access time from $\overline{CAS}$ low to data in valid, read cycle		$3.5t_Q-40^\dagger$		$3.5t_Q-25^\dagger$	ns
87	$t_{h(CH-DV)}$	Hold time of data in valid after $\overline{CAS}\uparrow$, read cycle	0		0		ns
88	$t_{h(CH-RAZ)}$	Hold time of row address high impedance after $\overline{CAS}$ high, end of read cycle	$1.5t_Q-10^\ddagger$		$1.5t_Q-10^\ddagger$		ns
89	$t_{h(CL-QEL)}$	Hold time of $\overline{TR}/\overline{QE}$ low after $\overline{CAS}$ low, read cycle	$3.5t_Q-25$		$3.5t_Q-10$		ns
90	$t_{su(CAZ-QEL)}$	Setup time of column address high impedance to $\overline{TR}/\overline{QE}\downarrow$, read cycle	$t_Q-10^\ddagger$		$t_Q-10^\ddagger$		ns
91	$t_{h(QEH-DV)}$	Hold time of data in valid after $\overline{TR}/\overline{QE}\uparrow$, read cycle	0		0		ns
92	$t_{d(CL-QEL)}$	Delay time from $\overline{CAS}\downarrow$ to $\overline{TR}/\overline{QE}$ low, read cycle		t_Q+20		t_Q+10	ns
93	$t_{a(QEL-DV)}$	Access time from $\overline{TR}/\overline{QE}$ low to data in valid, read cycle		$2.5t_Q-40^\dagger$		$2.5t_Q-25^\dagger$	ns
94	$t_{h(QEH-RAZ)}$	Hold time of row address high impedance after $\overline{TR}/\overline{QE}$ high, end of read cycle	$1.5t_Q-10^\ddagger$		$1.5t_Q-10^\ddagger$		ns
95	$t_{w(QEL)}$	Pulse duration, $\overline{TR}/\overline{QE}$ low, read cycle	$2.5t_Q-25$		$2.5t_Q-10$		ns
96	$t_{d(CL-ENL)}$	Delay time from $\overline{CAS}$ low to $\overline{DEN}$ low, read cycle		t_Q+20		t_Q+10	ns
97	$t_{h(ENH-DV)}$	Hold time of data in valid after $\overline{DEN}\uparrow$, read cycle	0		0		ns
98	$t_{su(CAZ-ENL)}$	Setup time of column address high impedance to $\overline{DEN}\downarrow$, read cycle	$t_Q-10^\ddagger$		$t_Q-10^\ddagger$		ns
99	$t_{h(ENH-RAZ)}$	Hold time of next row address high impedance after $\overline{DEN}$ high, end of read cycle	$1.5t_Q-10^\ddagger$		$1.5t_Q-10^\ddagger$		ns
100	$t_{a(ENL-DV)}$	Access time from $\overline{DEN}$ low to data in valid, read cycle		$2.5t_Q-40^\dagger$		$2.5t_Q-25^\dagger$	ns
101	$t_{h(ENH-DDH)}$	Hold time of DDOUT high after $\overline{DEN}$ high, read follows write cycle	$3t_Q-20$		$3t_Q-10$		ns
102	$t_{su(DDL-ENL)}$	Setup time of DDOUT low to $\overline{DEN}\downarrow$, read cycle	t_Q-20		t_Q-10		ns

NOTE: Advance information notices apply only to the TMS34010-60.
$^\dagger 4t_Q$ is added to these values for each wait state inserted.
‡These values are derived from characterization and are not tested.

local bus timing: read cycle



TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

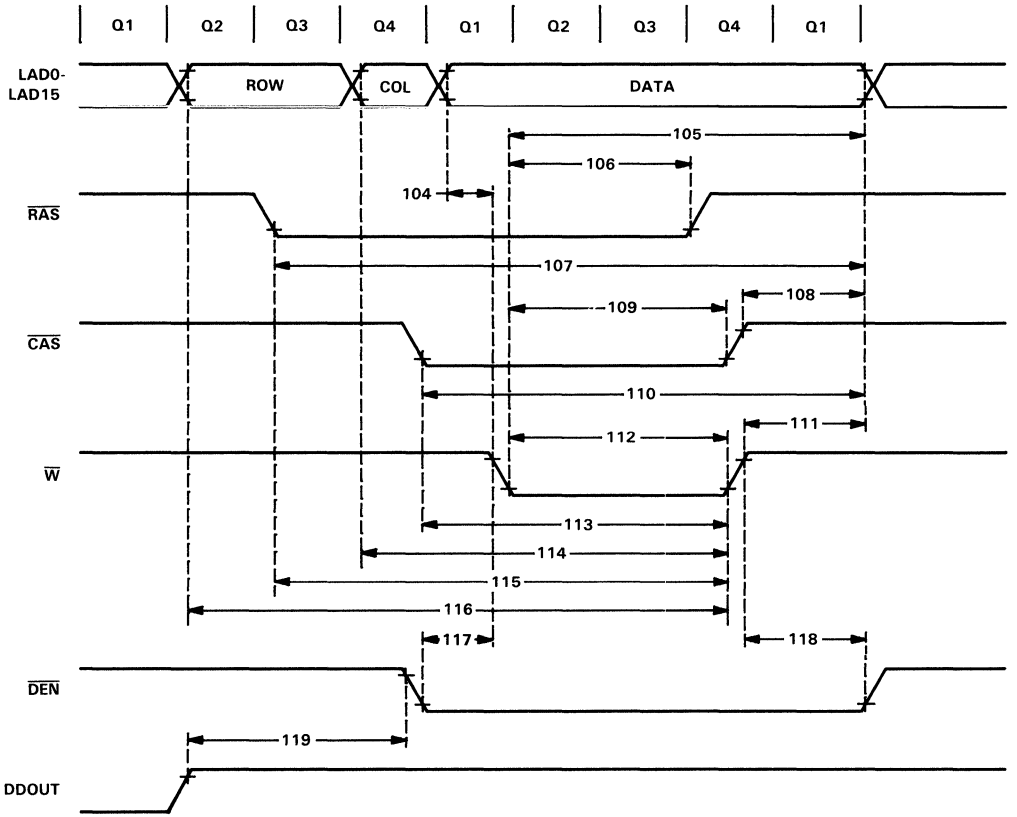
local bus timing parameters (continued)

Quarter clock time t_Q, which appears in the following table, is one quarter of a local output clock period, or $2t_{c(ICK)}$.

NO.	PARAMETER		TMS34010-40		TMS34010-50 TMS34010-60		UNIT
			MIN	MAX	MIN	MAX	
103	$t_{h(ENH-DDL)}$	Hold time of DDOUT low after $\overline{DEN}$ high, read cycle	$1.5t_Q-15$		$1.5t_Q-10$		ns
104	$t_{su(DV-WL)}$	Setup time of data out valid to $\overline{W}\downarrow$, write cycle	t_Q-20		t_Q-15		ns
105	$t_{h(WL-DV)}$	Hold time of data out valid after $\overline{W}$ low, write cycle	$4t_Q-20$		$4t_Q-10$		ns
106	$t_{su(WL-RH)}$	Setup time of $\overline{W}$ low to $\overline{RAS}\uparrow$, write cycle	$2t_Q-20$		$2t_Q-10$		ns
107	$t_{h(RL-DV)}$	Hold time of data out valid after $\overline{RAS}$ low, write cycle	$7t_Q-20$		$7t_Q-10$		ns
108	$t_{h(CH-DV)}$	Hold time of data out valid after $\overline{CAS}$ high, write cycle	$1.5t_Q-15$		$1.5t_Q-10$		ns
109	$t_{su(WL-CH)}$	Setup time of $\overline{W}$ low to $\overline{CAS}\uparrow$, write cycle	$2.5t_Q-25$		$2.5t_Q-10$		ns
110	$t_{h(CL-DV)}$	Hold time of data out valid after $\overline{CAS}$ low, write cycle	$5t_Q-20$		$5t_Q-10$		ns
111	$t_{h(WH-DV)}$	Hold time of data out valid after $\overline{W}$ high, write cycle	$1.5t_Q-15$		$1.5t_Q-10$		ns
112	$t_{w(WL)}$	Pulse duration, $\overline{W}$ low	$2.5t_Q-25$		$2.5t_Q-10$		ns
113	$t_{h(CL-WL)}$	Hold time of $\overline{W}$ low after $\overline{CAS}$ low, write cycle	$3.5t_Q-25$		$3.5t_Q-10$		ns
114	$t_{su(CAV-WH)}$	Setup time of column address valid to $\overline{W}\uparrow$, write cycle	$4.5t_Q-30$		$4.5t_Q-15$		ns
115	$t_{h(RL-WL)}$	Hold time of $\overline{W}$ low after $\overline{RAS}$ low, write cycle	$5.5t_Q-25$		$5.5t_Q-10$		ns
116	$t_{su(RAV-WH)}$	Setup time of row address valid to $\overline{W}\uparrow$, write cycle	$6.5t_Q-35$		$6.5t_Q-15$		ns
117	$t_{su(ENL-WH)}$	Setup time of $\overline{DEN}$ low to $\overline{W}\downarrow$, write cycle	t_Q-20		t_Q-10		ns
118	$t_{h(WH-ENL)}$	Hold time of $\overline{DEN}$ low after $\overline{W}$ high, write cycle	$1.5t_Q-15$		$1.5t_Q-10$		ns
119	$t_{su(DDH-ENL)}$	Setup time of DDOUT high to $\overline{DEN}\downarrow$, write follows read	$3t_Q-20$		$3t_Q-10$		ns

NOTE: Advance information notices apply only to the TMS34010-60.

local bus timing: write cycle



TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

local bus timing parameters (continued)

Quarter clock time t_Q, which appears in the following table, is one quarter of a local output clock period, or $2t_{c(ICK)}$.

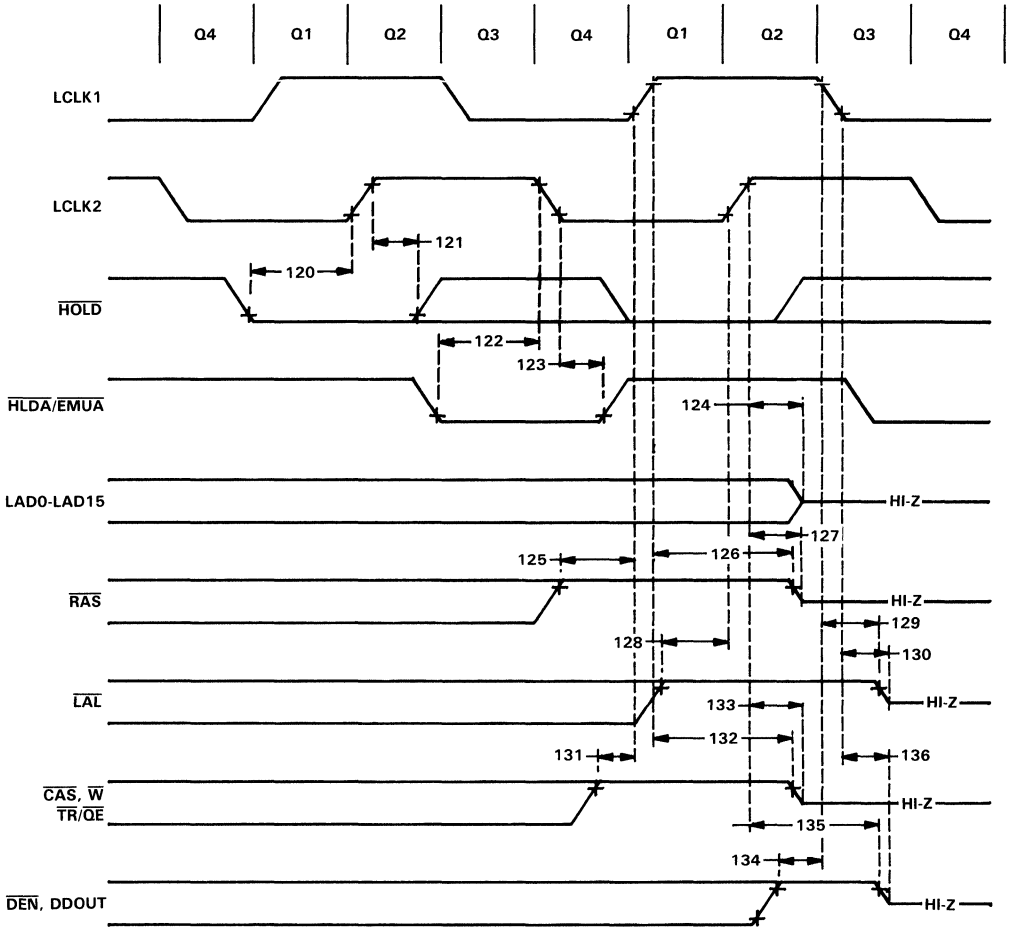
NO.	PARAMETER		TMS34010-40		TMS34010-50 TMS34010-60		UNIT
			MIN	MAX	MIN	MAX	
120	$t_{su(HRV-CK2H)}$	Setup time of $\overline{HOLD}$ valid to LCLK2↑	50†		40†		ns
121	$t_{h(CK2H-HRV)}$	Hold time of $\overline{HOLD}$ valid after LCLK2 high	0†		0†		ns
122	$t_{su(HKV-CK2L)}$	Setup time of $\overline{HLDA}/\overline{EMUA}$ output valid before LCLK2↓	t_Q-20		t_Q-10		ns
123	$t_{h(CK2L-HKL)}$	Hold time of $\overline{HLDA}/\overline{EMUA}$ low, after LCLK2 low	t_Q-15		t_Q-15		ns
124	$t_{d(CK2H-DZ)}$	Delay from LCLK2 high to LAD pins high impedance, bus release		30‡		30‡	ns
125	$t_{su(RH-CK1H)}$	Setup time of $\overline{RAS}$ high to LCLK1↑	t_Q-20		t_Q-10		ns
126	$t_{h(CK1H-RH)}$	Hold time of $\overline{RAS}$ driven high after LCLK1 high, bus release	t_Q-10‡		t_Q-10‡		ns
127	$t_{d(CK2H-RZ)}$	Delay from LCLK2 high to $\overline{RAS}$ high impedance, bus release		30‡		30‡	ns
128	$t_{su(ALH-CK2H)}$	Setup time of $\overline{LAL}$ high to LCLK2↑	t_Q-20		t_Q-10		ns
129	$t_{h(CK1L-ALH)}$	Hold time of $\overline{LAL}$ driven high after LCLK1↓, bus release	−5‡		−5‡		ns
130	$t_{d(CK1L-ALZ)}$	Delay from LCLK1 low to $\overline{LAL}$ high impedance, bus release		30‡		30‡	ns
131	$t_{su(CH-CK1H)}$	Setup time of $\overline{CAS}$, $\overline{W}$, and $\overline{TR}/\overline{QE}$ high to LCLK1↑	$0.5t_Q-15$		$0.5t_Q-10$		ns
132	$t_{h(CK1H-CH)}$	Hold time of $\overline{CAS}$, $\overline{W}$, and $\overline{TR}/\overline{QE}$ high after LCLK1 high, bus release	t_Q-10‡		t_Q-10‡		ns
133	$t_{d(CK2H-CZ)}$	Delay from LCLK2 high to $\overline{CAS}$, $\overline{W}$, and $\overline{TR}/\overline{QE}$ high impedance, bus release		30‡		30‡	ns
134	$t_{su(ENH-CK2H)}$	Setup time of $\overline{DEN}$ or DDOUT high to LCLK1↓	t_Q-20		t_Q-10		ns
135	$t_{h(CK2H-ENH)}$	Hold time of $\overline{DEN}$ and DDOUT high after LCLK1↓, bus release	t_Q-10‡		t_Q-10‡		ns
136	$t_{d(CK1L-ENZ)}$	Delay from LCLK1 low to $\overline{DEN}$ and DDOUT high impedance, bus release		30‡		30‡	ns
137	$t_{h(CK2H-DZ)}$	Hold time of LAD bus high impedance after LCLK2↑	−5‡		−5‡		ns
138	$t_{h(CK2H-RZ)}$	Hold time of $\overline{RAS}$, $\overline{CAS}$, $\overline{W}$, $\overline{LAL}$, and $\overline{TR}/\overline{QE}$ high impedance after LCLK1↑	−5‡		−5‡		ns
139	$t_{d(CK1H-RH)}$	Delay from LCLK1 high to RAS, $\overline{CAS}$, $\overline{W}$, $\overline{LAL}$, and $\overline{TR}/\overline{QE}$ driven high, resume bus control		30		30	ns
140	$t_{h(CK2H-RH)}$	Hold time of $\overline{RAS}$ high after LCLK2 high, resumes bus control	t_Q-15		t_Q-10		ns
141	$t_{h(CK2H-CH)}$	Hold time of $\overline{CAS}$, $\overline{W}$, and $\overline{TR}/\overline{QE}$ high after LCLK2 high, resume bus control	−5‡		−5‡		ns

NOTE: Advance information notices apply only to the TMS34010-60.

†$\overline{HOLD}$ is a synchronous input sampled during the low-to-high transition of LCLK2. The specified setup and hold times must be met for the device to operation properly.

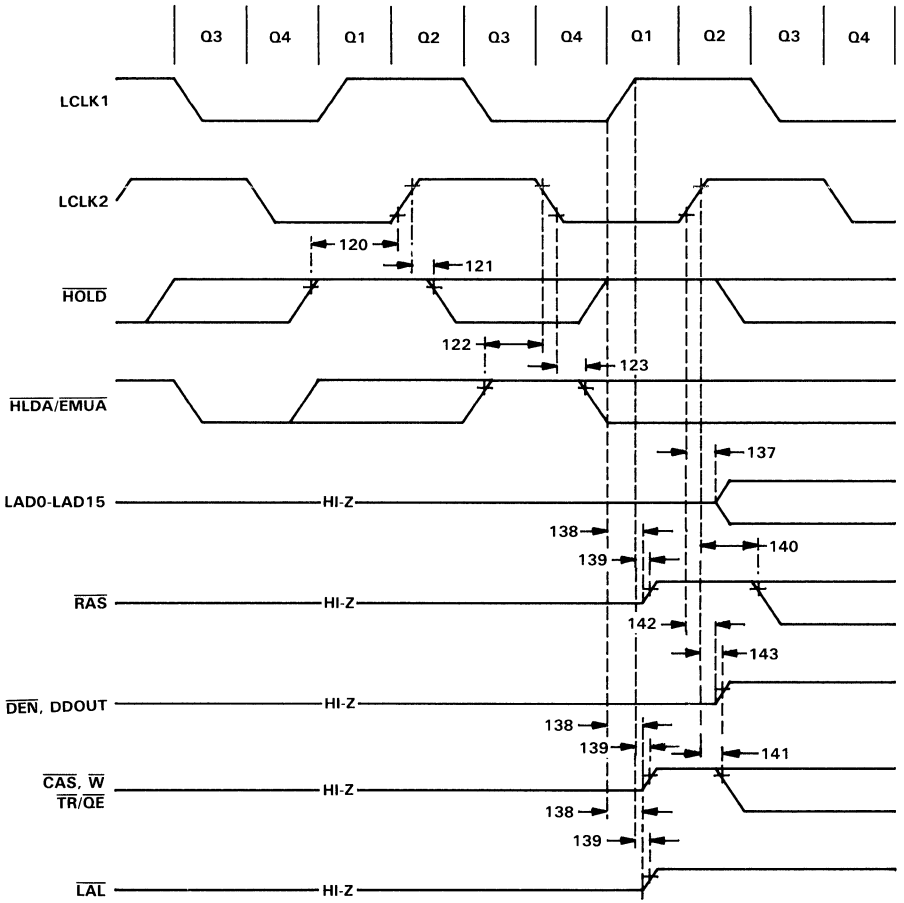
‡These values are derived from characterization and are not tested.

GSP releases control of local bus



TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

GSP resumes control of local bus



local bus timing parameters (continued)

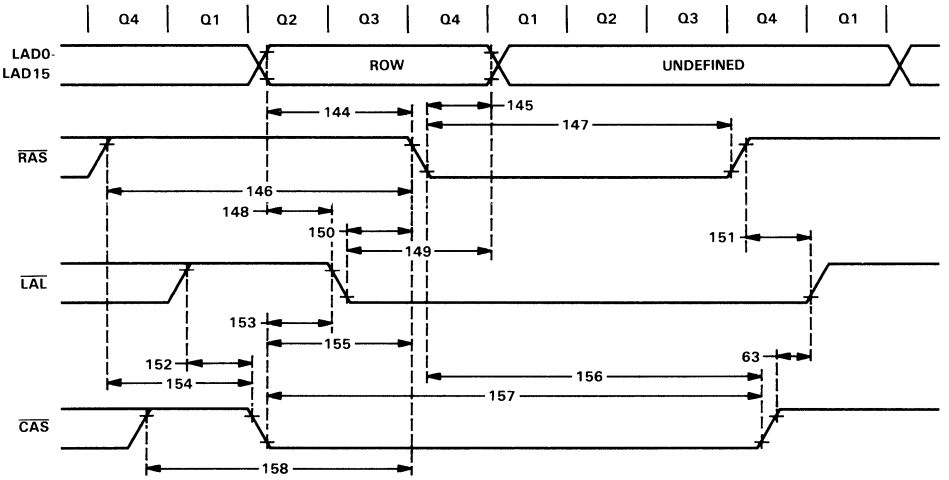
Quarter clock time t_Q, which appears in the following table, is one quarter of a local output clock cycle, or $2t_{c(ICK)}$.

NO.	PARAMETER		TMS34010-40		TMS34010-50 TMS34010-60		UNIT
			MIN	MAX	MIN	MAX	
142	$t_{h(CK2H-ENZ)}$	Hold time of $\overline{DEN}$, DDOUT high impedance after LCLK2 high, resume bus control	−5†		−5†		ns
143	$t_{d(CK2H-ENH)}$	Delay from LCLK2 high to $\overline{DEN}$, and DDOUT driven high, resume bus control		30		30	ns
144	$t_{su(RAV-RL)}$	Setup time of row address valid to $\overline{RAS}$↓, $\overline{CAS}$-before-$\overline{RAS}$ refresh	$2t_Q-25$		$2t_Q-15$		ns
145	$t_{h(RL-RAV)}$	Hold time of row address valid after $\overline{RAS}$ low, $\overline{CAS}$-before-$\overline{RAS}$ refresh	t_Q-20		t_Q-10		ns
146	$t_{w(RH)}$	Pulse duration, $\overline{RAS}$ high, start of $\overline{CAS}$-before-$\overline{RAS}$ refresh	$4t_Q-20$		$4t_Q-10$		ns
147	$t_{w(RL)}$	Pulse duration, $\overline{RAS}$ low, $\overline{CAS}$-before-$\overline{RAS}$ refresh	$4t_Q-20$		$4t_Q-10$		ns
148	$t_{su(RAV-ALL)}$	Setup time of row address valid to LAL↓, $\overline{CAS}$-before-$\overline{RAS}$ refresh	t_Q-20		t_Q-15		ns
149	$t_{h(ALL-RAV)}$	Hold time of row address valid after $\overline{LAL}$ low, $\overline{CAS}$-before-$\overline{RAS}$ refresh	$2t_Q-20$		$2t_Q-10$		ns
150	$t_{h(ALL-RH)}$	Hold time of $\overline{RAS}$ high after $\overline{LAL}$ low, $\overline{CAS}$-before-$\overline{RAS}$ refresh	t_Q-20		t_Q-10		ns
151	$t_{su(RH-ALH)}$	Setup time of $\overline{RAS}$ high to $\overline{LAL}$↑, $\overline{CAS}$-before-$\overline{RAS}$ refresh	t_Q-20		t_Q-10		ns
152	$t_{su(ALH-CL)}$	Setup time of $\overline{LAL}$ high to $\overline{CAS}$↓, $\overline{CAS}$-before-$\overline{RAS}$ refresh	t_Q-20		t_Q-10		ns
153	$t_{su(CL-ALL)}$	Setup time of $\overline{CAS}$ low to $\overline{LAL}$↓, $\overline{CAS}$-before-$\overline{RAS}$ refresh	t_Q-20		t_Q-10		ns
154	$t_{su(RH-CL)}$	Setup time of $\overline{RAS}$ high to $\overline{CAS}$↓, $\overline{CAS}$-before-$\overline{RAS}$ refresh	$2t_Q-20$		$2t_Q-10$		ns
155	$t_{su(CL-RL)}$	Setup time of $\overline{CAS}$ low to $\overline{RAS}$↓, $\overline{CAS}$-before-$\overline{RAS}$ refresh	$2t_Q-20$		$2t_Q-10$		ns
156	$t_{h(RL-CL)}$	Hold time of $\overline{CAS}$ low after $\overline{RAS}$ low, $\overline{CAS}$-before-$\overline{RAS}$ refresh	$4.5t_Q-25$		$4.5t_Q-10$		ns
157	$t_{w(CL)}$	Pulse duration, $\overline{CAS}$ low, $\overline{CAS}$-before-$\overline{RAS}$ refresh	$6.5t_Q-25$		$6.5t_Q-10$		ns
158	$t_{su(CH-RL)}$	Setup time of $\overline{CAS}$ high to $\overline{RAS}$↓, $\overline{CAS}$-before-$\overline{RAS}$ refresh	$3.5t_Q-15$		$3.5t_Q-10$		ns

NOTE: Advance information notices apply only to the TMS34010-60.
†These values are derived from characterization and are not tested.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

$\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ DRAM refresh cycle timing



local bus timing parameters (continued)

Quarter clock time t_Q, which appears in the following table, is one quarter of a local output clock cycle, or $2t_{c(ICK)}$.

NO.	PARAMETER		TMS34010-40		TMS34010-50 TMS34010-60		UNIT
			MIN	MAX	MIN	MAX	
159	$t_{h(CK2H-RH)}$	Hold time of $\overline{RAS}$ high after LCLK2 high, all cycles except internal and $\overline{CAS}$-before-$\overline{RAS}$ refresh	t_Q-15		t_Q-10		ns
160	$t_{su(RL-CK2L)}$	Setup time of $\overline{RAS}$ low to LCLK2↓, all cycles except internal and $\overline{CAS}$-before-$\overline{RAS}$ refresh	t_Q-20		t_Q-10		ns
161	$t_{h(CK1L-RH)}$	Hold time of $\overline{RAS}$ high after LCLK1 low, $\overline{CAS}$-before-$\overline{RAS}$ refresh	t_Q-15		t_Q-10		ns
162	$t_{su(RL-CK1H)}$	Setup time of $\overline{RAS}$ low to LCLK1↑, $\overline{CAS}$-before-$\overline{RAS}$ refresh	t_Q-20		t_Q-10		ns
163	$t_{h(CK1L-RL)}$	Hold time of $\overline{RAS}$ low after LCLK1 low, all cycles except internal	t_Q-15		t_Q-10		ns
164	$t_{su(RH-CK1H)}$	Setup time of $\overline{RAS}$ high to LCLK1↑, all cycles except internal	t_Q-20		t_Q-10		ns
165	$t_{h(CK2L-ALH)}$	Hold time of $\overline{LAL}$ high after LCLK2 low, all cycles except internal	$0.5t_Q-15$		$0.5t_Q-10$		ns
166	$t_{su(ALL-CK1H)}$	Setup time of $\overline{LAL}$ low to LCLK1↑, all cycles except internal	$0.5t_Q-15$		$0.5t_Q-10$		ns
167	$t_{h(CK2L-ALL)}$	Hold time of $\overline{LAL}$ low after LCLK2 low, all cycles except internal	t_Q-15		t_Q-10		ns
168	$t_{su(ALH-CK2H)}$	Setup time of $\overline{LAL}$ high after LCLK2↑, all cycles except internal	t_Q-20		t_Q-10		ns
169	$t_{h(CK1H-CH)}$	Hold time of $\overline{CAS}$ high after LCLK1 high, $\overline{CAS}$-before-$\overline{RAS}$ refresh	t_Q-15		t_Q-10		ns
170	$t_{su(CL-CK1L)}$	Setup time of $\overline{CAS}$ low to LCLK↓, $\overline{CAS}$-before-$\overline{RAS}$ refresh	t_Q-20		t_Q-10		ns
171	$t_{h(CK2L-CH)}$	Hold time of $\overline{CAS}$ high after LCLK2 low, cycles except internal, DRAM refresh and $\overline{CAS}$-before-$\overline{RAS}$ refresh	t_Q-15		t_Q-10		ns
172	$t_{su(CL-CK2H)}$	Setup time of $\overline{CAS}$ low to LCLK2↑, all cycles except internal, DRAM refresh, and $\overline{CAS}$-before-$\overline{RAS}$ refresh	t_Q-20		t_Q-10		ns
173	$t_{h(CK2L-CL)}$	Hold time of $\overline{CAS}$ low after LCLK2 low, all cycles except internal and DRAM refresh	$0.5t_Q-15$		$0.5t_Q-10$		ns
174	$t_{su(CH-CK1H)}$	Setup time of $\overline{CAS}$ high to LCLK1↑, all cycles except internal and DRAM refresh	$0.5t_Q-15$		$0.5t_Q-10$		ns
175	$t_{h(CK1H-WH)TR}$	Hold time of $\overline{W}$ high after LCLK1 high, shift register transfer	t_Q-15		t_Q-10		ns
176	$t_{su(WL-CK1L)TR}$	Setup time of $\overline{W}$ low to LCLK1↓, shift register transfer	t_Q-20		t_Q-10		ns

NOTE: Advance information notices apply only to the TMS34010-60.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

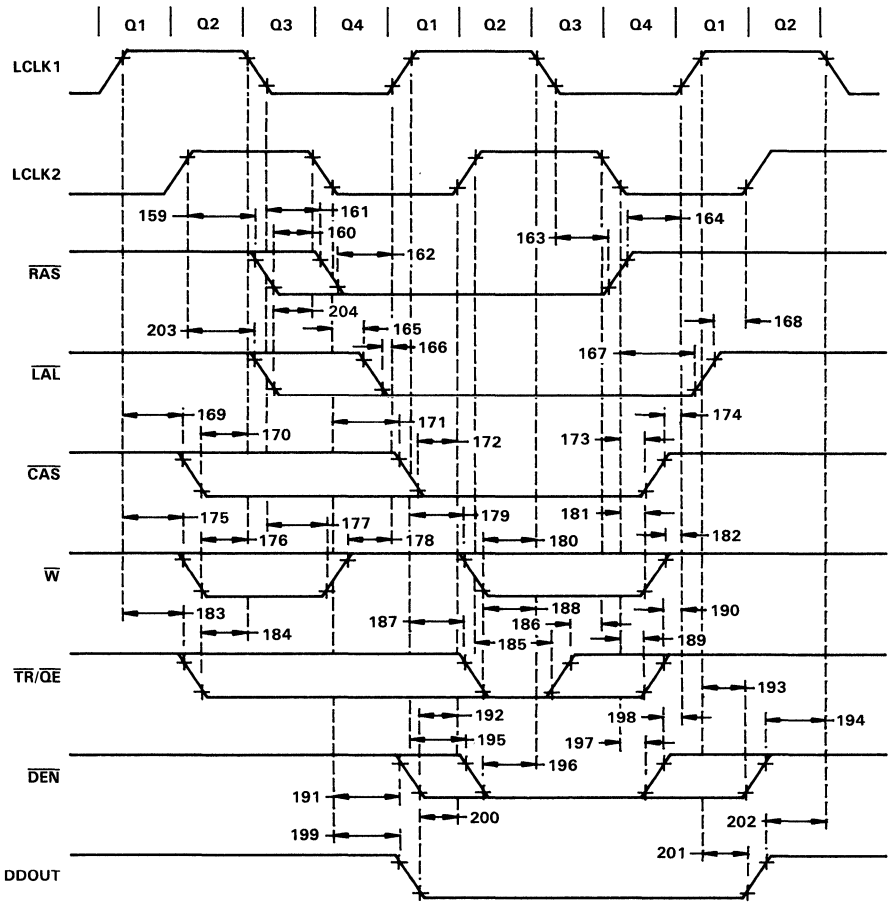
local bus timing parameters (concluded)

Quarter clock time t_Q, which appears in the following table, is one quarter of a local output clock cycle, or $2t_{c(ICK)}$.

NO.	PARAMETER		TMS34010-40		TMS34010-50 TMS34010-60		UNIT
			MIN	MAX	MIN	MAX	
177	$t_{h(CK1L-WL)}$	Hold time of $\overline{W}$ low after LCLK1 low, shift register transfer	t_Q-15		t_Q-10		ns
178	$t_{su(WH-CK1H)}$	Setup time of $\overline{W}$ high to LCLK1↑, shift register transfer	t_Q-20		t_Q-10		ns
179	$t_{h(CK1H-WH)}$	Hold time of $\overline{W}$ high after LCLK1 high, write	t_Q-15		t_Q-10		ns
180	$t_{su(WL-CK1L)}$	Setup time of $\overline{W}$ low to LCLK1↓, write	t_Q-20		t_Q-10		ns
181	$t_{h(CK2L-WL)}$	Hold time of $\overline{W}$ low after LCLK2 low, write	$0.5t_Q-15$		$0.5t_Q-10$		ns
182	$t_{su(WH-CK1H)}$	Setup time of $\overline{W}$ high to LCLK1↑, write	$0.5t_Q-15$		$0.5t_Q-10$		ns
183	$t_{h(CK1L-TRH)}$	Hold time of $\overline{TR}/\overline{QE}$ high after LCLK1 high, shift register transfer	t_Q-15		t_Q-10		ns
184	$t_{su(TRL-CK1H)}$	Setup time of $\overline{TR}/\overline{QE}$ low to LCLK1↓, shift register transfer	t_Q-20		t_Q-10		ns
185	$t_{h(CK2H-TRL)}$	Hold time of $\overline{TR}/\overline{QE}$ low after LCLK2 high, shift register transfer	t_Q-15		t_Q-10		ns
186	$t_{su(TRH-CK2L)}$	Setup time of $\overline{TR}/\overline{QE}$ high to LCLK2↓, shift register transfer	t_Q-20		t_Q-10		ns
187	$t_{h(CK1H-QEH)}$	Hold time of $\overline{TR}/\overline{QE}$ high after LCLK1 high, read	t_Q-15		t_Q-10		ns
188	$t_{su(QEL-CK1L)}$	Setup time of $\overline{TR}/\overline{QE}$ low to LCLK1↓, read	t_Q-20		t_Q-10		ns
189	$t_{h(CK2L-QEL)}$	Hold time of $\overline{TR}/\overline{QE}$ low after LCLK2 low, read	$0.5t_Q-15$		$0.5t_Q-10$		ns
190	$t_{su(QEH-CK1H)}$	Setup time of $\overline{TR}/\overline{QE}$ high to LCLK1↑, read	$0.5t_Q-15$		$0.5t_Q-10$		ns
191	$t_{h(CK2L-ENH)}$	Hold time of $\overline{DEN}$ high after LCLK2 low, write	t_Q-15		t_Q-10		ns
192	$t_{su(ENL-CK2H)}$	Setup time of $\overline{DEN}$ low to LCLK2↑, read	t_Q-20		t_Q-10		ns
193	$t_{h(CK1H-ENL)}$	Hold time of $\overline{DEN}$ low after LCLK1 high, write	t_Q-15		t_Q-10		ns
194	$t_{su(ENH-CK1L)}$	Setup time of $\overline{DEN}$ high to LCLK1↓, write	t_Q-20		t_Q-10		ns
195	$t_{h(CK1H-ENH)}$	Hold time of $\overline{DEN}$ high after LCLK1 high, read	t_Q-15		t_Q-10		ns
196	$t_{su(ENL-CK1L)}$	Setup time of $\overline{DEN}$ low to LCLK1↓, read	t_Q-20		t_Q-10		ns
197	$t_{h(CK2L-ENL)}$	Hold time of $\overline{DEN}$ low after LCLK2 low, read	$0.5t_Q-15$		$0.5t_Q-10$		ns
198	$t_{su(ENH-CK1H)}$	Setup time of $\overline{DEN}$ high to LCLK1↑, read	$0.5t_Q-15$		$0.5t_Q-10$		ns
199	$t_{h(CK2L-DDH)}$	Hold time of DDOUT high after LCLK2 low, read	t_Q-15		t_Q-10		ns
200	$t_{su(DDL-CK2H)}$	Setup time of DDOUT low to LCLK2↑, read	t_Q-20		t_Q-10		ns
201	$t_{h(CK1H-DDL)}$	Hold time of DDOUT low after LCLK1 high, read	t_Q-15		t_Q-10		ns
202	$t_{su(DDH-CK1L)}$	Setup time of DDOUT high to LCLK1↓, read	t_Q-20		t_Q-10		ns
203	$t_{h(CK2H-ALH)}$	Hold time of $\overline{LAL}$ high after LCLK2 high, $\overline{CAS}$-before-$\overline{RAS}$ refresh	t_Q-15		t_Q-10		ns
204	$t_{su(ALL-CK2L)}$	Setup time of $\overline{LAL}$ low to LCLK2↓, $\overline{CAS}$-before-$\overline{RAS}$ refresh	t_Q-20		t_Q-10		ns

NOTE: Advance information notices apply only to the TMS34010-60.

local bus timing: relationship of control signals to clocks



TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

video interface timing parameters

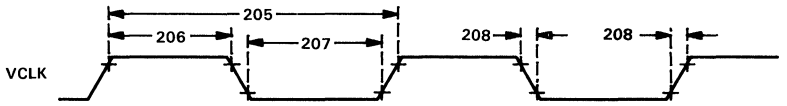
The timing parameters for TMS34010 video interface signals are shown in the next three tables and diagrams. The video interface includes the following TMS34010 pins: VCLK (video input clock), $\overline{\text{BLANK}}$ (blanking), $\overline{\text{HSYNC}}$ (horizontal sync, bidirectional), and $\overline{\text{VSYNC}}$ (vertical sync, bidirectional). $\overline{\text{HSYNC}}$ and $\overline{\text{VSYNC}}$ are inputs if external sync mode is enabled; otherwise they are outputs.

video input clock timing parameters

NO.	PARAMETER		TMS34010-40		TMS34010-50 TMS34010-60		UNIT
			MIN	MAX	MIN	MAX	
205	$t_{c(VCK)}$	Period of video input clock VCLK	100		80		ns
206	$t_{w(VCKH)}$	Pulse duration of VCLK high	40		30		ns
207	$t_{w(VCKL)}$	Pulse duration of VCLK low	40		30		ns
208	$t_{t(VCK)}$	Transition time (rise and fall) of VCLK		5†		5†	ns

NOTE: Advance information notices apply only to the TMS34010-60.
†This value is determined through computer simulation and is not tested.

video input clock timing

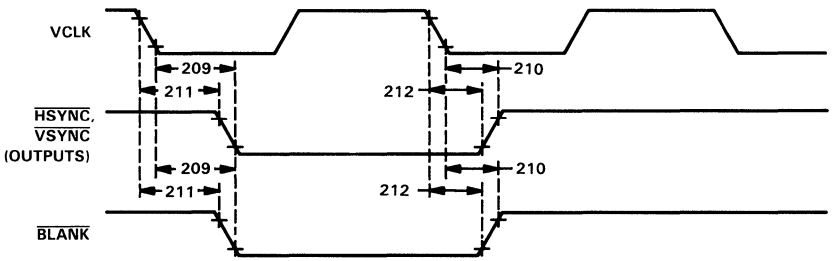


video interface timing parameters: outputs

NO.	PARAMETER		TMS34010-40		TMS34010-50 TMS34010-60		UNIT
			MIN	MAX	MIN	MAX	
209	$t_{d(VCKL-HSL)}$	Delay from VCLK low to $\overline{\text{HSYNC}}$, $\overline{\text{VSYNC}}$, or $\overline{\text{BLANK}}$ low		30		30	ns
210	$t_{d(VCKL-HSH)}$	Delay from VCLK low to $\overline{\text{HSYNC}}$, $\overline{\text{VSYNC}}$, or $\overline{\text{BLANK}}$ high		30		30	ns
211	$t_{h(VCKL-HSH)}$	Hold time of $\overline{\text{HSYNC}}$, $\overline{\text{VSYNC}}$, or $\overline{\text{BLANK}}$ high after VCLK↓	0		0		ns
212	$t_{h(VCKL-HSL)}$	Hold time of $\overline{\text{HSYNC}}$, $\overline{\text{VSYNC}}$, or $\overline{\text{BLANK}}$ low after VCLK↓	0		0		ns

NOTE: Advance information notices apply only to the TMS34010-60.

video output timing

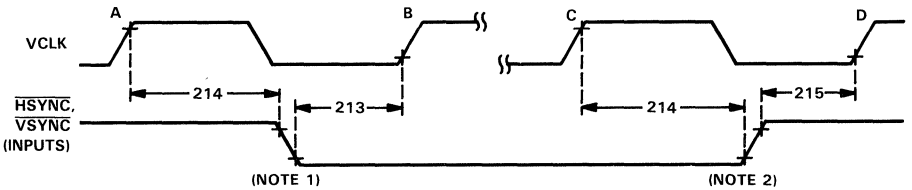


ADVANCE INFORMATION

video interface timing: external sync inputs

NO.	PARAMETER		TMS34010-40		TMS34010-50 TMS34010-60		UNIT
			MIN	MAX	MIN	MAX	
213	$t_{su(HSV-VCKH)}$	Setup time of $\overline{HSYNC}$, $\overline{VSYNC}$ valid to VCLK↑	20†		20†		ns
214	$t_{h(VCKH-HSV)}$	Hold time of $\overline{HSYNC}$, $\overline{VSYNC}$ valid after VCLK high	20†		20†		ns
215	$t_{su(HSH-VCKH)}$	Setup time of $\overline{HSYNC}$, $\overline{VSYNC}$ high to VCLK↑	20‡		20‡		ns

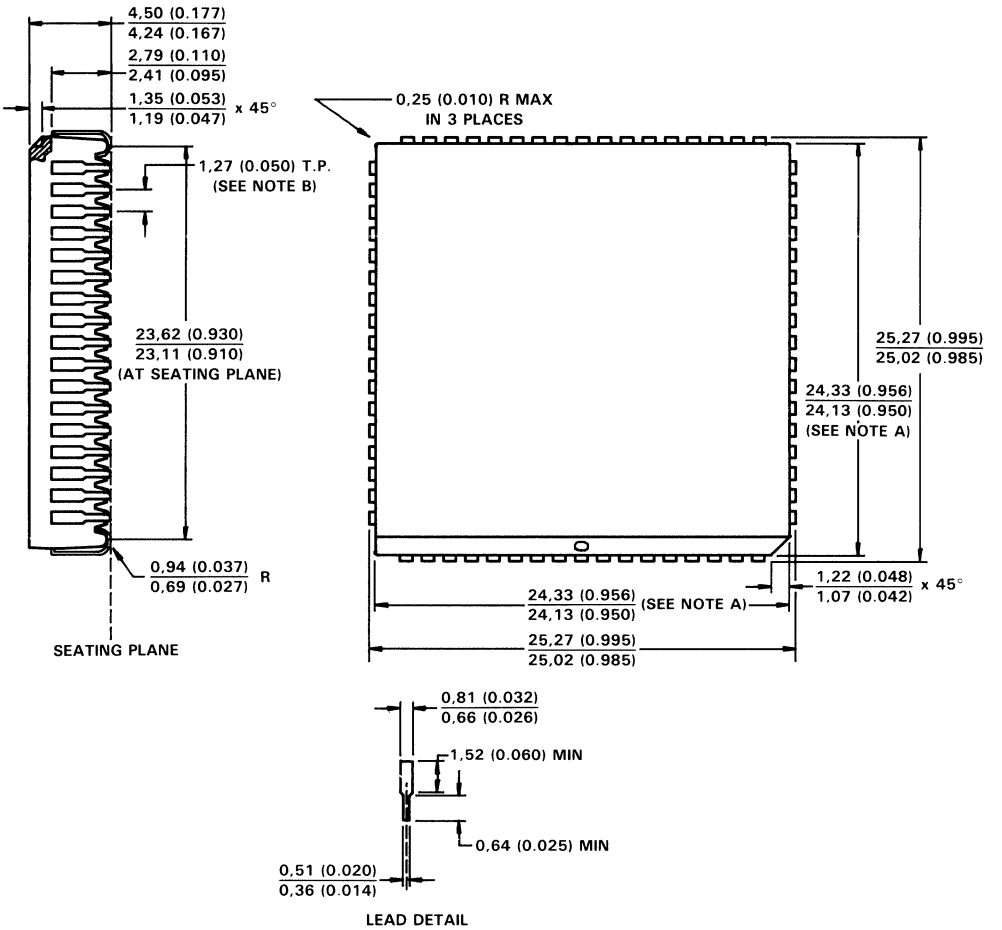
NOTE: Advance information notices apply only to the TMS34010-60.
†Specified setup and hold times on asynchronous inputs are required only to guarantee recognition at indicated clock edge.
‡This value is determined through computer simulation.



NOTES: 1. If the falling edge of the sync signal occures more than $t_{h(SV-VCH)}$ past VCLK edge A, and at least $t_{su(SV-VCH)}$ before edge B, the transition will be detected at edge B instead of edge A.
2. If the rising edge of the sync signal occurs more than $t_{h(SV-VCH)}$ past VCLK edge C, and at least $t_{su(SV-VCH)}$ before edge D, the transition will be detected at edge D instead of edge C.

TEXAS INSTRUMENTS
POST OFFICE BOX 1443 • HOUSTON, TEXAS 77001

MECHANICAL DATA



NOTES: A. Centerline of center pin each side is within 0,10 (0.004) of package centerline as determined by this dimension.
B. Location of each pin is within 0,127 (0.005) of true position with respect to center pin on each side.

ALL LINEAR DIMENSIONS ARE IN MILLIMETERS AND PARENTHETICALLY IN INCHES.

Printed in U.S.A.

SPVS002B

Appendix B

System Design Considerations

Please read these emulation guidelines before starting a system design that requires use of an XDS emulator. In-circuit emulators place added constraints on the system hardware and software design; the XDS TMS34010 emulator minimizes these constraints as much as possible. Many of the emulator signals come directly from the device itself keeping the delays to a minimum. The information provided in this appendix will allow your design to be compatible with the TMS34010 XDS emulator.

Topics in this section include:

B.1 Pin Loading

The loading provided by the emulator differs from the device loading; in some cases, this additional loading can cause a system to fail or pass. For instance, within the emulator the VCLK and INCLK clocks are buffered by an AS157 device. If the target clocks are driven by an AS driver over a long distance, the system may operate satisfactorily with the emulator's loading. When the device is used, the clocks can ring, causing the system to fail.

To minimize ringing caused by the emulator, all the outputs and I/Os are terminated with 22-ohm series resistors. Most of the signals are connected directly to the target connector to minimize the added delays. Table B-1 shows the device loads that are placed on the emulator pins.

In Table B-1, **pin** refers to the emulator target pin. An arrow (→) indicates that the specified device drives the device(s) it the arrow points to. Devices in parallel are separated by commas; devices in series are in brackets. As an example, consider LAD0; the pin connects to a 22-ohm series resistor, and the other side of the resistor is connected to an ALS245, AS573, and the TMS34010.

Table B-1. Loading

Pin	Signal	I/O	Loading
1	V_{SS}		
2	RUN/EMU		pin, 4.7KPU, ALS08
3	$\overline{\text{RESET}}$	I	pin, 4.7KPU, TIBPAL-12, TMS9901, ALS574, AS257 → [ALS08 + AS04 + ALS08 + AS874 + AS04] → TMS34010
4	VCLK	I	pin, 330PU, 510PD, A5157 → TMS34010
5	INCLK	I	pin → [AS157] → TMS34010
6	$\overline{\text{LINT1}}$	I	pin, 4.7KPU → [AS257] → TMS34010
7	$\overline{\text{LINT2}}$	I	pin, 4.7KPU → [AS257] → TMS34010
8	$\overline{\text{HOLD}}$	I	pin, 4.7KPU, AS874, TMS34010
9	LRDY	O	pin, 4.7KPU, [TIBPAL-12] → TMS34010
10	LAD0	I/O	pin → [22S] → ALS245, AS573, TMS34010
11	LAD1	I/O	pin → [22S] → ALS245, AS573, TMS34010
12	LAD2	I/O	pin → [22S] → ALS245, AS573, TMS34010
13	LAD3	I/O	pin → [22S] → ALS245, AS573, TMS34010
14	LAD4	I/O	pin → [22S] → ALS245, AS573, TMS34010
15	LAD5	I/O	pin → [22S] → ALS245, AS573, TMS34010
16	LAD6	I/O	pin → [22S] → ALS245, AS573, TMS34010
17	LAD7	I/O	pin → [22S] → ALS245, AS573, TMS34010
18	$\overline{\text{VSS}}$		
19	LAD8	I/O	pin → [22S] → ALS245, AS573, TMS34010

PD – Pull down
PU – Pull up
S – Series resistor
[ ] – Devices in series
pin – Emulator target cable pin

Table B-1. Loading (Continued)

Pin	Signal	I/O	Loading
20	LAD9	I/O	pin → [22S] → ALS245, AS573, TMS34010
21	LAD10	I/O	pin → [22S] → ALS245, AS573, TMS34010
22	LAD11	I/O	pin → [22S] → ALS245, AS573, TMS34010
23	LAD12	I/O	pin → [22S] → ALS245, AS573, TMS34010
24	LAD13	I/O	pin → [22S] → ALS245, AS573, TMS34010
25	LAD14	I/O	pin → [22S] → ALS245, AS573, TMS34010
26	LAD15	I/O	pin → [22S] → ALS245, AS573, TMS34010
27	V_{CC}		
28	LCLK1	O	TMS34010, AS244, AS04 → [22S] → pin
29	LCLK2	O	TMS34010, AS244 → [22S] → pin
30	HSYNC	I/O	TMS34010, ALS573 → [22S] → pin
31	VSYNC	I/O	TMS34010, ALS573 → [22S] → pin
32	BLANK	O	TMS34010, ALS573 → [22S] → pin
33	HLDA/EMU	O	TMS34010 → [AS08 + 10S] → pin
34	LAL	O	TMS34010, 100KPU, AS11, AS04 → [22S] → pin
35	V_{SS}		
36	DDOUT	O	TMS34010 → [TIBPAL-12 + 22S] → pin
37	$\overline{DEN}$	O	TMS34010 → [TIBPAL-12 + 22S] → pin
38	$\overline{RAS}$	O	TMS34010 AS157, 100KPU → [22S] → pin
39	$\overline{CAS}$	0	TMS34010 → [TIBPAL-12 + 22S] → pin
40	$\overline{WRITE}$	0	TMS34010 → [TIBPAL-12 + 22S] → pin
41	TR/QE	O	TMS34010 → [TIBPAL-12 + 22S] → AL244, pin
42	HINT	O	TMS34010 ALS244 → [22S] → pin
43	HRDY	O	TMS34010 → [AS08 + 10S] → pin
44	HAD15	I/O	TMS34010 → [22S] → pin
45	HAD14	I/O	TMS34010 → [22S] → pin
46	HAD13	I/O	TMS34010 → [22S] → pin
47	HAD12	I/O	TMS34010 → [22S] → pin
48	HAD11	I/O	TMS34010 → [22S] → pin
49	HAD10	I/O	TMS34010 → [22S] → pin
50	HAD9	I/O	TMS34010 → [22S] → pin
51	HAD8	I/O	TMS34010 → [22S] → pin
52	V_{SS}		

PD – Pull down
PU – Pull up
S – Series resistor
[ ] – Devices in series
pin – Emulator target cable pin

Table B-1. Loading (Concluded)

Pin	Signal	I/O	Loading
53	HAD7	I/O	TMS34010 → [22S] → pin
54	HAD6	I/O	TMS34010 → [22S] → pin
55	HAD5	I/O	TMS34010 → [22S] → pin
56	HAD4	I/O	TMS34010 → [22S] → pin
57	HAD3	I/O	TMS34010 → [22S] → pin
58	HAD2	I/O	TMS34010 → [22S] → pin
59	HAD1	I/O	TMS34010 → [22S] → pin
60	HAD0	I/O	TMS34010 → [22S] → pin
61	V_{CC}		
62	$\overline{HUDS}$	I	pin, 100KPU, TMS34010
63	$\overline{HLDS}$	I	pin, 100KPU, TMS34010
64	$\overline{HREAD}$	I	pin, 100KPU, ALS573, TMS34010
65	$\overline{HWRITE}$	I	pin, 100KPU, AS573, TMS34010
66	$\overline{HCS}$	I	pin, 4.7KPU → [AS232 + AS08 + TMS34010] → ALS00, ALS74, TIBPAL-12, LS02
67	HFS0	I	pin, 100KPU, ALS573, TMS34010
68	HFS1	I	pin, 100KPU, ALS573, TMS34010

PD – Pull down
PU – Pull up
S – Series resistor
[ ] – Devices in series
pin – Emulator target cable pin

B.2 Signal Timing and Delay

The target cable delays all signal timings by approximately four nanoseconds; the following signals impose an additional delay:

Signal	Delay	Signal	Delay
VCLK	6ns	$\overline{CAS}$	12ns
INCLK	6ns	$\overline{WRITE}$	12ns
LINT1	6ns	$\overline{TR/OE}$	12ns
LINT2	6ns	$\overline{LRDY}$	12ns
DDOUT	12ns	HRDY	6ns
$\overline{DEN}$	12ns	$\overline{HCS}$	12ns

Remember these additional delays when you are calculating system timings. The cable delays should cancel out when comparing the signal to the clock; for instance, the clock is delayed by four nanoseconds and so is the address. The problem comes from setup times required by the TMS34010. For instance, $\overline{CAS}$ out is delayed by 16 nanoseconds and data-in is delayed by four nanoseconds. This 20 nanoseconds must be added to the memory access time. For this reason, it is important not to use $\overline{CAS}$ to control LRDY. Both

LRDY and $\overline{CAS}$ are delayed by 12 nanoseconds; combined, this adds 24 nanoseconds to the LRDY setup, which violates the device requirements.

B.3 Transmission Line Phenomena

Since the XDS target cable is approximately 20 inches long, use of advanced CMOS or fast/advanced Schottky-TTL may cause line reflections (ringing above input thresholds) on input lines to the XDS. Series termination resistors (22 to 68 ohms) can help to eliminate this problem. In some cases where significant additional signal length is added to XDS outputs, the series resistors on the XDS may not be sufficient to control reflections. In this case, additional corrective actions may be necessary.

B.4 Host Port Operation

The emulator host port supports two modes of operation:

- The first mode blocks the host port while the emulator is in control mode (that is, not running) and when the XDS emulator requires its internal TMS34010 as a resource. Systems that access the host port when in control mode have HRDY inactive when they start an access and remain inactive until the emulator is put into run mode. This is a problem for PCs because they use DMA cycles to perform DRAM refresh and are prevented from performing any further memory cycles.

- The second mode of operation allows host accesses while the emulator is in control mode. When the emulator is halted, a snapshot is taken of the I/O registers before transferring control to the user. While halted, the host has access to the TMS34010's I/O registers and memory space. Data read from the I/O and memory space may not represent the actual data as the host can be changing the data through the host port. Data written through the XDS user interface to any I/O registers or memory locations used by the host port can cause unpredictable results. This mode is the typical dual-allocation problem that is prevalent in multiport memory systems.

B.5 Reset Buffering

The $\overline{RESET}$ input from the target system is buffered so that reset cannot abort a memory cycle that is in progress. This is necessary to prevent corruption of the substitution memory. Reset is ANDed with $\overline{RAS}$ and $\overline{CAS}$ and clocked on the rising edge of LCLK1. If reset is active and $\overline{RAS}$ and $\overline{CAS}$ are inactive, then reset is applied to the processor. While the $\overline{RESET}$ input is active, internal emulator logic provides $\overline{CAS}$-before-$\overline{RAS}$ refreshes for emulator memory, but not for your system memory; target memory is not refreshed during reset. Therefore, you shouldn't perform a target reset following a download of object code into target memory, because this may destroy the contents of target memory.

If the target system also generates a reset during the emulator-TMS34010 reset, and $\overline{HCS}$ is high (which, under normal circumstances, would halt the TMS34010), the emulator reset takes precedence and does not halt the TMS34010.

B.6 Local Ready Timing

The ready logic requires special attention; the target system should not depend on having $\overline{CAS}$ to clear ready. This can cause a deadlock situation if the memory is write protected. In this case, the emulator blocks the $\overline{CAS}$ output to the target and the target system is locked waiting for $\overline{CAS}$ to release LRDY. Thus, if you are working with the emulator and write-protecting memory, your ready logic should use a combination of $\overline{RAS}$ and clock delays instead of $\overline{CAS}$.

B.7 Memory Substitution

The 256K bytes of substitution memory is implemented with two banks of 64Kx4 DRAMs. This memory can be mapped on 2K word boundaries. The memory can be selected as write-only using the memory protect feature described below. When a memory access is made that is mapped into substitution memory the signals are modified as follows:

- $\overline{DEN}$, $\overline{CAS}$, $\overline{WR}$, and $\overline{TR}/\overline{QE}$ are blocked from going active.
- Ready is accepted from the target to allow adding wait states to the substitution memory.

Ready should not be controlled by $\overline{DEN}$, $\overline{CAS}$, $\overline{WR}$ or $\overline{TR}/\overline{QE}$ because these signals are blocked and a deadlock condition would take place.

All the other signals operate as though a standard memory cycle were taking place. Be aware that when you're using substitution memory, it can only be accessed by the processor and host port and cannot be accessed when the TMS34010 is put into hold. The target system should not drive the LAD bus unless $\overline{CAS}$ and $\overline{DEN}$ are active.

B.8 Write Protecting Memory

The emulator allows memory to be write-protected in blocks of 2048 words. The memory can be external memory as well as the substitution memory. The memory write operations are inhibited by blocking the $\overline{CAS}$ output. As with the substitution memory, ready should not be controlled by $\overline{CAS}$ because a deadlock condition will take place. External system memory is only write protected from CPU and host accesses and not DMA accesses initiated with a HOLD/HOLDA sequence.

B.9 Tracing and Cache

Please note that the breakpoint trace and timing capabilities are used to monitor bus activity. The TMS34010's pipelined-cache-based architecture fetches the current instruction and the three associated instructions in the cache subsegment when a cache miss occurs. This is indicated as four fetches even though only one instruction may be executed. Also, when the cache is enabled, code already located in cache does not generate any instruction-fetch activity on the memory bus when it is executed. If you want to create a complete trace history, run the TMS34010 with cache disabled. By disabling cache, all instructions executed are moved over the external memory bus every time they are executed, allowing them to be captured in the trace buffer.

Appendix C

Software Compatibility with Future GSPs

This appendix provides guidelines for writing TMS34010 programs that will be compatible with future versions of TMS340x0 devices. In some cases, following these guidelines may not produce the fastest TMS34010 code; however, your code should run without modification on future GSPs.

These guidelines cover several areas:

C.1 General Guidelines

- Future GSPs may have different instruction execution times than the TMS34010 has; therefore, timing loops based on TMS34010 instruction execution time may not be compatible with future GSPs. Even if future devices are generally faster than the TMS34010, specific cases may run slower. To avoid this, timing could be based on DPYINT (the display interrupt) or on an external time source (via $\overline{\text{LINT1}}$ or $\overline{\text{LINT2}}$). Note that if you use DPYINT, you must consider different display resolutions and refresh rates.

- For optimum performance on future GSPs, align data on 32-bit boundaries (instead of 16-bit boundaries). This could reduce the number of memory cycles for future GSPs, and in most cases will have little impact on the TMS34010. In particular, keeping the stack pointer (SP) aligned to 32-bit boundaries will speed up subroutine calls and interrupts.

- Future GSPs may use the reserved bits in the status register and in the I/O registers. During context switches, the values of reserved bits should be saved and restored as if they were valid; do not assume that these bits have known values. If you don't follow this guideline, your code may inadvertently enable/disable new options or features.

  Unless otherwise noted in this user's guide, when reserved bits have a value of 0, they will cause future GSPs to behave like the TMS34010. However, you should not set these bits to 0, because this may incorrectly reset a bit.

- Use the REV instruction to determine which version of the GSP you are using. You can use this instruction to decide whether to enable or disable version-dependent code.

- Instruction cache statistics characteristics (including cache size, loading order, number of bytes loaded per cache miss, and time per cache fetch) may differ between versions of GSPs. Code should not depend on the state of any of these characteristics.

C.2 Graphics Compatibility

- Extend the values in the COLOR0 and COLOR1 registers to 32 bits. The TMS34010 uses only the 16 LSBs of these registers; future GSPs may use all 32 bits.

- Treat the PMASK register as a 32-bit register. The TMS34010 uses only the 16 bits at address 0C0000160h; however, future GSPs may also use the 16 bits at address 0C0000170h. Whenever you save/restore the value at 0C0000160h, you should also save/restore the value at 0C0000170h.

- When you save/restore the graphics context (this includes all graphics operations control registers), you should also save/restore the I/O reg-

isters that are reserved in the TMS34010 I/O register map (addresses 0C0000130h–0C00001A0h).

- At initialization, load register B13 with all 1s. Future GSPs may use B13 as a pattern register; if you don't set B13 to the suggested value, future devices may draw a patterned line where the TMS34010 would draw a solid line.

- If an instruction uses the CONVSP register, then SPTCH (B1) must agree with CONVSP (the 5 LSBs of CONVSP must equal the 1s complement of $\log_2$ SPTCH, which is given by the LMO of SPTCH). Future GSPs may have instructions that use SPTCH to determine the pitch values instead of using CONVSP; that is, the instruction may perform the $\log_2$ conversion automatically.

  Set the 11 MSBs of CONVSP to 0; the TMS34010 ignores the values of these bits, but future GSPs may use these bits.

- If an instruction uses the CONVDP register, then DPTCH (B3) must agree with CONVDP (the 5 LSBs of CONVDP must equal the 1s complement of $\log_2$ DPTCH, which is given by the LMO of DPTCH). Future GSPs may have instructions that use DPTCH to determine the pitch values instead of using CONVDP; that is, the instruction may perform the $\log_2$ conversion automatically.

  Set the 11 MSBs of CONVDP to 0; the TMS34010 ignores the values of these bits, but future GSPs may use these bits.

C.3 Memory Map Compatibility

- The 32 16-bit words following the TMS34010 I/O registers (addresses 0C0000200h–0C00003F0h) are currently reserved. Future GSPs may use these addresses for additional I/O registers, so do not write code that uses these addresses.

- Do not use any reserved addresses in the TMS34010 memory map; future GSPs may use these locations. Specifically, address 0FFFFE000h, which is currently reserved, may be used for system configuration information.

C.4 I/O Register and Video Timing Compatibility

- Future GSPs may use different I/O registers to control video timing and VRAM shift register control. The vertical and horizontal counters may still be accessible at their current locations. DPYADR, DPYCTL, DPYSTRT, and DPYTAP may have new functions and/or addresses. HESYNC, HEBLNK, HSBLNK, HTOTAL, VESYNC, VEBLNK, VSBLNK, and VTOTAL may have similar functions but different addresses. These video control functions may be redefined so that future GSPs can take advantage of new advances in video RAM technology.

- Code that accesses video timing registers should be separated from other code so that you can easily replace it.

- Future GSPs may use different DRAM refresh methods; the TMS34010 provides control every 32 or 64 CPU cycles, and it may be necessary to have more control.

C.5 Interrupts Compatibility

- Interrupt service routines should not make assumptions about the state of the stack (except for the location of the ST and the PC). Future GSPs may push additional parameters or status information on the stack before pushing the PC and ST.

> **Note:**
>
> You **must** use RETI to return from an interrupt service routine. This ensures that any additional parameters that future GSPs may push on the stack will also be popped from the stack, and also ensures that the correct internal registers will be restored.

- PIXBLT interruption may behave differently on future GSPs. An interrupted PIXBLT may store status information on the stack instead of in registers, and different information may be stored.

> **Note:**
>
> Do not modify values stored in the register file by an interrupted PIXBLT. Future GSPs may not use this information or these locations.

- Opcodes that the TMS34010 flags as illegal may be valid opcodes for future GSPs. Therefore, if you want to use a software trap, use the TRAP instruction instead of an illegal opcode.

- Traps 3–7 and 12–15 are reserved for future interrupts.

C.6 Host Interface Compatibility

- Certain features of the TMS34010 host interface may need to be implemented in external hardware for future GSPs. However, the host interface registers and their functions will remain the same so that TMS34010 code that uses these registers will be compatible with future GSPs.

- Code written for a host processor that accesses the GSP host interface may have to be modified to comprehend a modified host interface.

Appendix D

Glossary

aliasing: A stairstep effect on a raster display of a line or arc segment.

antialiasing: A method for reducing the severity of aliasing effects seen in lines and edges drawn on a bit-mapped display device. This method adjusts the intensity of a pixel used to represent a portion of a line or edge according to the pixel's distance from the line or edge. Antialiasing requires that the display device be capable of producing one or more intermediate intensity levels between bright and off.

asynchronous communications: A method of transmitting data in which the timing of character placement of connecting transmitting lines is not critical. The transmitted characters are preceded by a start and followed by a stop bit, thus permitting the interval between characters to vary.

aspect ratio: The ratio of width to height. For the rectangular picture transmitted by a television station, the aspect ratio is 4:3.

back porch: The portion of a horizontal blanking pulse that follows the trailing edge of the horizontal synchronizing pulse.

background illumination: The average brightness of a screen.

bandwidth: The number of bits per second that can be transferred by a device.

binary array: Alternate name for a two-dimensional bit map in which each pixel is represented as single bit.

BitBlt: Bit aligned block transfer. Transfer of a rectangular array of pixel information from one location in a bitmap to another with potential of applying 1 of 16 boolean operators during the transfer.

bit map: 1. The digital representation of an image in which bits are mapped to pixels. 2. A block of memory used to hold raster images in a device-specific format.

bit plane: Hardware used as a storage medium for a bit map.

black level: The amplitude of the composite signal at which the beam of the picture tube is extinguished (becomes black) to blank retrace of the beam. This level is established at 75% of the signal amplitude.

blanking signal: Pulses used to extinguish the scanning beam during horizontal and vertical retrace periods.

breakpoint: A place in a routine specified by an instruction, instruction digit, or other condition, where the routine may be interrupted by external intervention or by a monitor routine.

clipping: Removing parts of display elements that lie outside a given boundary, usually a window or a viewport.

composite video: The color-picture signal plus all blanking and synchronizing signals. The signal includes luminance and chrominance signals, vertical- and horizontal-sync pulses, vertical- and horizontal-sync pulses, vertical-and horizontal-blanking pulses, and the color-burst signal.

DAC: Digital-to-analog converter. A device that converts a digital input code to an analog output voltage or current. The analog output level represents the value of the digital input code.

direct access: Pertaining to the process of obtaining data from, or placing data into, storage where the time required for such access is independent of the location of the data most recently obtained or placed in storage.

display area: The rectangular part of the physical display screen in which information coded in conformance with a video encoding standard is visibly displayed. The display area does not include the border area.

display element: A basic graphic element that can be used to construct a display image.

display memory: The area of memory which is used to hold the graphics image output to the video monitor.

display pitch: The difference in memory addresses between two pixels that appear in vertically adjacent positions (one directly above the other) on the screen.

display unit: A device which provides a visual representation of data.

dot clock: The dot clock cycles the rate at which video data is output to a CRT monitor.

DRAM refresh: The operation of maintaining data stored in dynamic RAMs. Data are stored in dynamic RAms as electrical charges across a grid of capacitive cells. The charge stored in a cell will leak off over time.

execution unit: The portion of a central processing unit that actually executes the data operations specified by program instructions.

field: 1. A group of contiguous bits in a register or memory dedicated to a particular function or representing a single entity. 2. A software-configurable data type in the TMS34010 whose length can be programmed to be any value in the range 1 to 32 bits.

fill: Solid coloring or shading of a display surface, often achieved as a pattern of horizontal segments.

frame: 1. The time required to refresh an entire screen. 2. The screen image output during a single vertical sweep.

frame buffer: A portion of memory used to buffer rasterized data to be output to a CRT display monitor. The contents of the frame buffer are often referred to as the bit map of the display and contain the logical pixels corresponding to the points on the monitor screen.

front porch: The portion of a horizontal blanking pulse that precedes the leading edge of the horizontal sync pulse.

GKS: Graphical Kernel System. An application programmer's standard interface to a graphics display.

glue logic: The small- and medium-scale-integrated devices necessary to complete the interface between two or more large or very-large-scale integrated devices.

gray scale: A scale of light intensities from black to white.

GSP: Graphics System Processor. A single-chip device embodying all the processing power and control capabilities necessary to manage a high-performance bit-mapped graphics system. The TMS34010 is the first such device.

high-impedance: The third state of a three-state output driver, in which the output is driven neither high or low but behaves as an open connection.

hold signal: A signal from a device capable of controlling a processor bus (for example, a processor or a DMA controller) which the device sends to a bus arbiter to request control of the bus. Typically, the arbiter signals the granting of the request by sending a hold-acknowledgement signal to the requesting device.

hold time: The minimum amount of time that valid data must be present at an input after the device is clocked to ensure proper data acceptance.

horizontal blanking interval: The time during which the display is blanked to cover the horizontal retracing of the electron beam.

horizontal sync: The synchronization signal that enables horizontal retrace of the electron beam of a CRT display.

icon: A graphic symbol representing a menu item.

interlaced scanning: A system of TV-picture scanning. Odd-numbered scanning lines, which make up an odd field, are interlaced with the even-numbered lines of an even field. The two interlaced fields constitute one frame. In effect, the number of transmitted pictures is doubled, thus reducing flicker.

lookup table: A table used during scan conversion of the digital image that converts color-map addresses into the actual color values displayed.

LRU: Least-recently-used cache-replacement algorithm. When a cache miss occurs, a cache-replacement algorithm selects which cache segment will be overwritten, based on the likelihood that the data in the discarded segment will not be needed again for some time. The LRU algorithm selects the segment which was used least recently.

mask: A pattern of characters that is used to control the retention or elimination of portions of another pattern of characters.

memory map: A map of memory space partitioned into functional blocks.

monotonicity: The quality of proceeding in a uniform manner. For example, the analog level output from a DAC should increase with each increase in the value of the digital input code.

multiplexing: Refers to a process of transmitting more than one set of signals at a time over a single wire or communications link.

NABTS: North American Broadcast Teletext Specification

NAPLPS: Abbreviation for the North American Presentation Level Protocol Syntax, which is a proposed standard for Videotex services.

nonmaskable interrupt: An interrupt request that cannot be disabled.

NMI: Nonmaskable interrupt. The NMI is an interrupt that is permanently enabled; it cannot be disabled.

NTSC: Abbreviation for the National Television System Committee, a group representing a wide range of interests in the television broadcast and video industry. The NTSC is instrumental in developing standards.

operand: That which is operated upon. An operand is usually identified by an address part of an instruction.

origin: The zero intersection of X and Y axes from which all points are calculated.

overlay: The plane of a graphics display that can be superimposed on another plane.

pack: To compress data in a storage medium by eliminating redundant information in such a way that the original data can later be recovered.

palette: A digital lookup table used in a computer graphics display for translating data from the bit map into the pixel values to be shown on the display.

pan: Apparent horizontal or vertical movement of a computer graphics screen (or window) over an image contained in a frame buffer that is too large to be completely displayed in a single static picture.

phase: The time interval for each clock period in a system is divided into two phases. One phase corresponds to the time the clock signal is high, and the other phase corresponds to the time the clock signal is low.

PHIGS: The programmer's Hierarchical Interactive Graphics Standard

pipelining: A design technique for reducing the effective propagation delay per operation by partitioning the operation into a series of stages, each of which performs a portion of the operation. A series of data is typically clocked through the pipeline in sequential fashion, advancing one stage per clock period.

pitch: The difference in starting addresses of two adjacent rows of pixels in a two-dimensional pixel array.

pixel: Picture element. 1. The smallest controllable point of light on a CRT display screen. 2. In a bit-mapped display, the logical data structure that contains the attributes to be shown at the corresponding physical pixel position on the CRT display screen.

pixel processing operation: A specified Boolean or arithemetic operation used to combine two pixel values (source and destination).

PixBlt: (Abbreviation of Pixel Block transfer) Operations on arrays of pixels in which each pixel is represented by one or more bits. PixBlt operations are a superset of BitBlt operations, and include not only the commonly-used boolean functions, but also integer arithmetic and other multi-bit operations.

plane: (Also bit plane or color plane.) A plane is a bit-map layer in a display device with multiple bits per pixel. If the pixel size is n bits, and the bits in each pixel are numbered 0 to n-1, plane 0 is made up of bits numbered 0 from all the pixels, and the plane n-1 is made up of bits numbered n-1 from all the pixels. A layered graphics display allows planes or groups of planes to be manipulated independently of the other planes.

primary colors: A set of three colors from which all other colors may be regarded as derived; hence, any of a set of visual stimuli from which all colors may be produced by mixture. Each primary color must be different from the others, and a combination of two primaries must be capable of producing a third. In color television, the three primary colors are red, green and blue.

propagation delay: The time required for a change in logic level at an input to a circuit to be translated into a resulting change at an output.

protocol: A set of rules, formats, and procedures governing the exchange of information between peer processes at the same level.

pulse width: Pulse width, T_W. The time interval between specified reference points on the leading and trailing edges of the pulse waveform.

Random Access Memory (RAM): A memory from which all information can be obtained at the output with approximately the same time delay by choosing an address randomly and without first searching through a vast amount of irrelevant data.

raster: A rectangular grid of picture elements whose intensity levels are manipulated to represent images. In a bit-mapped display, the bits within a portion of the memory referred to as the frame buffer are mapped to the raster pattern of a CRT monitor.

raster display: A CRT display generated by an electron beam that illuminates the CRT by sweeping the beam horizontally across the phosphor surface in a predetermined pattern, providing substantially uniform coverage of the display area.

raster graphics: Computer graphics in which a display image is composed of an array of pixels arranged in rows and columns.

Raster-Op: The arithmetic or logical combination operation that takes place during the transfer of pixel arrays from one location to another.

raster scan: The grid pattern traced by the electron beam on the face of the CRT in a television or similar raster-scan display device.

ready signal: A signal from a memory or a memory-mapped peripheral that informs the processor when it is ready to complete a memory cycle. Slower memories or memory-mapped peripherals must extend the length of the memory cycle by negating the ready signal (in other words, by sending the processor a "not ready" signal until such time as the cycle can be completed.

resolution: The number of visible distinguishable units in the device coordinate space.

refresh: Method which restores charge on capacitance which deteriorates because of leakage.

reset: To restore to normal action.

resolution: The number of visible distinguishable units in the device coordinate space.

retrace: The line traced by the scanning beam or beams of a picture tube as it travels from the end of one horizontal line or field to the beginning of the next line or field.

RGB monitor: Red-Green-Blue Monitor. An RGB monitor is a CRT monitor capable of displaying colors and having separate inputs for the three signals used to drive the red, green and blue guns of the CRT.

relative coordinates: Location of a point relative to another data point.

rotate: To transform a display or display item by revolving it around a specified axis or center point.

scale: A size change made by multiplying or dividing the coordinate dimensions by a constant value.

scale factor: The value by which you divide or multiply the display dimensions in a scaling operation.

scaling: Enlarging or reducing all or part of a display image by multiplying the coordinates of display elements by a constant value.

scan line: A horizontal line traced across a CRT by the electron beam in a television or similar raster-scan device.

screen refresh: The operation of dumping the contents of the frame buffer to a CRT monitor in synchronization with the movement of the electron beam.

scrolling: Moving text strings or graphics vertically or horizontally.

segment: A collection of display elements that can be manipulated as a unit.

sequencing: Control method used to cause a set of steps to occur in a particular order.

setup time: The minimum amount of time that valid data must be present at an input before the device is clocked to ensure proper data acceptance.

shift register transfer: A transfer between the RAM storage and internal shift register in a video RAM.

sprite: A graphic object of a specified pattern appearing on its plane in a position determined by a single coordinate pair, specifying the sprite's location on the screen in the horizontal and vertical axis.

stairstepping: A visual effect seen in bit-mapped display devices which produce images by brightening or dimming individual picture elements (or pixels) contained in a two-dimensional grid of such elements. Stairstepping (also called aliasing) is the rough or jagged appearance of lines and edges which are not perfectly horizontal or vertical, resulting from transitions of the line or edge from one row or column of elements to another.

superimposed: Refers to the process that moves data from one location to another, superimposing bits or characters on the contents of specified locations.

tap point: The column address provided to a VRAM during a memory-to-shift-register cycle. The column address specifies the point at which the shift register is to be "tapped;" in other words, which cell of the shift register is to be connected to the serial output of the VRAM.

trace: A line of the graphics display.

transformation: Geometric alteration of a graphics display, such as scaling, translation, or rotation.

transparency: When a pixel with the attribute of transparency is written to the screen, it is effectively invisible, and does not alter that portion of the screen it is written to. For example, in a pixel array containing the pattern for the letter *A*, all pixels surrounding the *A* pattern could be given a special value indicating that they are transparent. When the array is written to the screen, the *A* pattern, but not the pixels in the rectangle containing it, would be invisible.

VDI: Virtual Device Interface. The standard interface between the device-independent and the device-dependent levels of a graphics system.

VDM: Virtual Device Metafile. A standard mechanism for retaining and and transporting graphics data and control information at the level of the Virtual Device Interface.

vertical blanking interval: The time during which the display is blanked to cover the vertical retracing of the electron beam.

vertical blanking pulse: A positive or negative pulse developed during vertical retrace and appearing at the end of each field. It is used to blank out scanning lines during the vertical retrace interval.

vertical sync: The synchronization signal that enables vertical retrace of the electron beam of a CRT display.

video display processor: A microprocessor device dedicated to the tasks of display memory management (storage, retrieval, and refresh) and gener-

ation of all required video, control, and synchronization signals required by a TV display or CRT monitor.

video overlay: The mixing of one video signal with another such that parts of the image carried by the first signal replace the corresponding parts of the image carried by the second signal.

video RAM, VRAM: Video Random-Access Memory. A dual-ported memory device for computer graphics applications, containing two interfaces; one interface to allow a processor to read or write data from an internal memory array; a second interface to provide a serial stream of screen refresh data to a CRT display device.

viewport: The specified window on the display surface that marks the limits of a display.

virtual coordinate system: A coordinate system created by mapping a portion of the world coordinate system to the space available on your device.

virtual space: Space referenced with the coordinates defined by the application.

wait state: A clock period inserted into a memory cycle in order to permit accesses of slower memories and slower memory-mapped peripherals.

window: A specified rectangular area of a virtual space shown on the display.

window clipping: Allowing text and graphics drawing to occur only within a specified rectangular window on the screen.

wire frame: A three-dimensional image displayed as a series of line segments outlining its surface.

zoom: To scale a display or display item so it is magnified or reduced on the screen.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

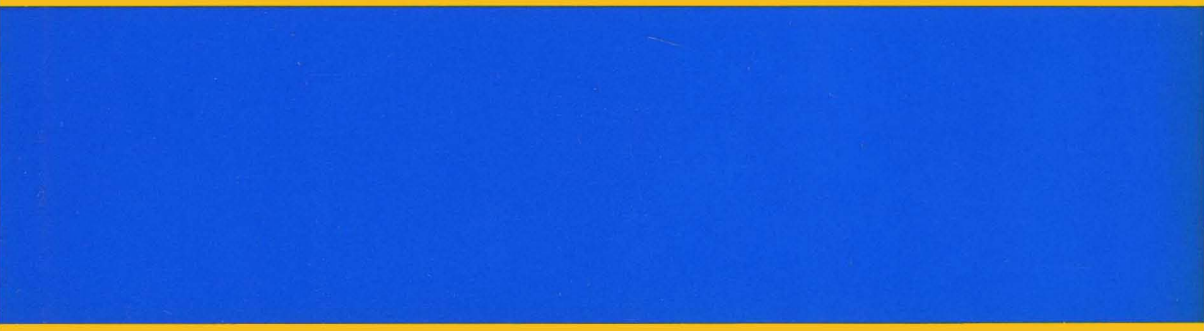
T

V

W

X

Z



TEXAS INSTRUMENTS

Printed in U.S.A., August 1988
1604899-9703

SPVU001A