intel®

APPLICATION NOTE

AP-447

November 1990

A Memory Subsystem for the i486™ CPU including Second Level Cache

GREGORY A. ROBERTSON
SENIOR APPLICATION ENGINEER

5

Order Number: 240799-001

A MEMORY SUBSYSTEM FOR THE i486™ CPU INCLUDING SECOND LEVEL CACHE

CONTENTS

1.0 INTRODUCTION

The i486™ CPU contains several improvements over its predecessor, the highly successful 386™ CPU. One of the most important of these is the processor's data access rate. The i486 CPU can access instructions and data from its on-chip cache in the same clock cycle. To support the processor's redesigned internal data path, the external bus has also been optimized and can access external memory at twice the rate of the 386 CPU. The internal cache requires rapid access to entire cache lines. Invalidation cycles must be supported to maintain consistency with external memory. All of these functions must be supported by the external memory system. Without them, the full performance potential of the CPU cannot be attained.

The requirements of todays multitasking and multiprocessor operating systems also put increased demand on the external memory system. OS support functions such as paging and context switching can degrade reference locality. Without efficient access to external memory, the performance of these functions is reduced.

Second level caching is a technique used to improve the memory interface. Some applications, such as multiuser office computers, require this feature to meet performance goals. Single-user systems, on the other hand, may not warrant the extra cost. Given the variety of applications incorporating the i486 CPU, memory system architecture will be very diverse.

In this application note, we will work with an example to discuss the details of memory system design. In the example, we have supported as many functions of the CPU as possible. An optional second-level cache is included. A write buffer is also implemented to reduce write latency. The cache supports zero wait state read cycles. The DRAM controller supports the following devices with the wait states shown in Table 2. The DRAM speed given in Table 1 is the RAS access time (tRAC). Table 2 summarizes the bus clocks required for each function.

Table 1

CPU Clock Freq.	DRAM Speed
25 MHz	100 ns
33 MHz	70 ns

Many of the functions and optimizations included here will not be required in every application. The example provides guidelines for the hardware designer but will not necessarily provide the optimal cost/performance solution for many applications. For example, 11 PLDs are required to implement the memory control logic partially due to the implementation of a back-off capability. An address register must also be used to implement this function. If this function is not used, the control logic can be substantially reduced. These and other optimizations will be discussed in the summary.

Table 2

DRAM Function	First Access Burst	Subsequent Burst Accesses	Write Cycles
Page Hit	3	1	2
Page Miss	7	1	5*

NOTE:
*Write miss latencies occur only during cycles subsequent to a write miss cycle.

The discussion assumes a working knowledge of computer system design. Items discussed but not explained include DRAM operation, PLD programming and operation, worst-case timing analysis and i486 CPU bus operation. The complete schematics and PLD equations are in Appendix A.

2.0 THE 485TURBOCACHE SECOND LEVEL CACHE MODULE

Several different types of second level cache architectures are possible candidates for use with the 486 CPU. For single cpu systems the different architectures offer similar performance benefits in most cases. The reason they are so similar is the mechanism which improves performance. The primary benefit of the second level cache is bus cycle latency reduction.

In most systems which incorporate a single i486 CPU, bus traffic from other bus masters is minimal. With any reasonable memory system the CPU uses at most 50% to 70% of the bus. Therefore reduction of bus cycle latency is the only performance benefit external logic can offer.

The second level cache used in this example is an econonmical method of reducing read cycle latency. The 485Turbocache module contains the control circuits, data and tag ram required to implement a 128k byte cache. It is organized as a two way set associative cache. Modules can be cascaded to provide up to 512K bytes of cache memory.

One of the most interesting aspects of this device is it can be a system option. To provide this capability the device is configured as a look-aside cache. It monitors the CPU address and control signals. When a cycle occurs in which the cache can supply data, it intervenes. The cache module then supplies an entire 16-byte line with no wait states.

The performance improvement offered by this cache is substantial in some environments. This performance improvement is particularly obvious when executing multitasking, multiuser operating systems such as

UNIX and OS/2. Some users, however, may not require the performance improvement offered by the cache. In these cases the cache as an option is attractive.

By designing the cache subsystem as an option both user's requirements can be met. A single system design can be manufactured for both customers. The UNIX or OS/2 user can add the cache module. Other users may or may not require the module. They can choose the system configuration which meets their price-performance needs.

When a single or multiple 485Turbocache Module devices are connected to an i486 processor system, the processor's internal cache should map the entire address space including that of the 485Turbocache Module devices to provide the highest performance. This is the most efficient configuration. The i486 CPU can access a line from its internal cache in one clock and the 485Turbocache Module provides the next fastest access in two clocks for the first doubleword and the remaining three doublewords in three clocks.

No matter how many 128-kbyte modules are cascaded, the set and tag addresses are connected to the same pins on the 485Turbocache Module. The processor's address bits A2–A31 are connected to A2- -A31 on the 485Turbocache Module. Internally, address bits A4–A15 are sent to both sets, to select one of 4,096 locations. Because the cache is two-way set associative, each address points to information stored in two banks. On each read or write cycle, the value of A16–A31 is compared to the tags stored at the location addressed by A4–A15. If they are equal, and if the valid bit is set, then a hit occurs. If a read cycle is in progress, then the 485Turbocache Module returns data to the i486 CPU. If the hit cycle is a write cycle, then the new data is updated in the 485Turbocache Module.

When multiple 485Turbocache Modules are used, the chip select starts by decoding A16 onwards. For example, with a 256-kbyte cache A16 and A17 are decoded for generating the CS#. The set and tag addresses of a system with four 485Turbocache Modules is shown in Figure 1.

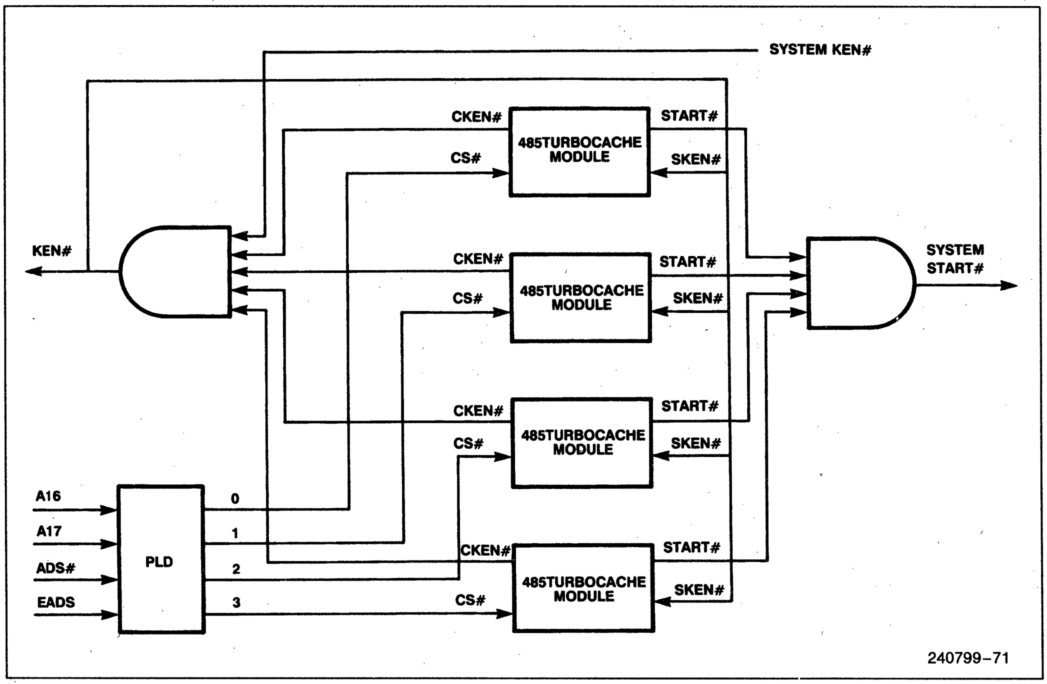


Figure 1. Multiple 485Turbocache Module Configuration

The BRDYO# output and the CBRDY# input must be used in forming of the i486 CPU's BRDY# input. Similarly, the CRDY# input must be used in forming of the i486 CPU's RDY# input. Signals that are common to the i486 CPU and the 485Turbocache Module include BOFF#, BLAST#, EADS#, BEO#-BE3#, and DPO-DP3.

The memory system generates KEN# to the i486 CPU when read data needs to be cached. The 485Turbocache Module receives this signal as the SKEN# input and produces CKEN# when appropriate. The 485Turbocache Module's CKEN# output can be used in the formation of the KEN# input to the i486 CPU. CKEN# can be used in conjunction with other logic that can deassert KEN# to the CPU when the system wants the current line fill to be cached by the 485Turbocache Module and not cached in the i486 CPU. The CKEN# signal is always asserted in T1, but is then deasserted if CS# is inactive.

The 485Turbocache Module connects directly to the i486 CPU's address lines A2-A31. The designer may have to add external buffers to the address outputs, depending upon the loading. Other signals connected to the i486 CPU include the burst control signals, the bus cycle definition signals, the byte enables, the ADS# signal, and the data and parity signals. The 485Turbocache Module and CPU connections are shown in Figure 2. The 485Turbocache Module main memory controller and bus controller interface are shown in Figure 3.

Read Hit Cycles

A read hit cycle occurs when requested data is present in the 485Turbocache Module. The i486 CPU attempts to retrieve the entire line from the 485Turbocache Module without incurring wait states. This may be accomplished by activating the KEN# input at the end of T1 (the clock in which ADS# becomes active). There is very little time to decode the address, generate the KEN# signal to the i486 CPU, and complete a zero wait state read operation. Because KEN# is sampled twice, it is possible to always assert KEN# in T1 and to wait until the end of a line fill to decide whether the data is cacheable. (See Section 3.2.)

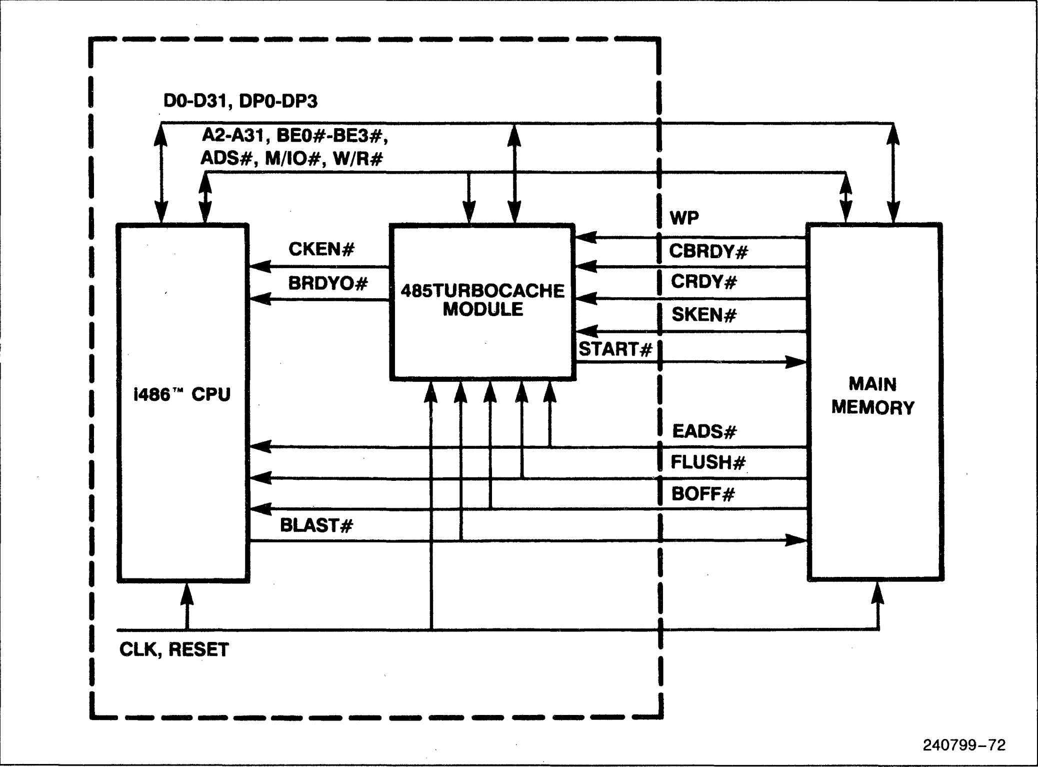


Figure 2. 485Turbocache Module and i486™ CPU Connections

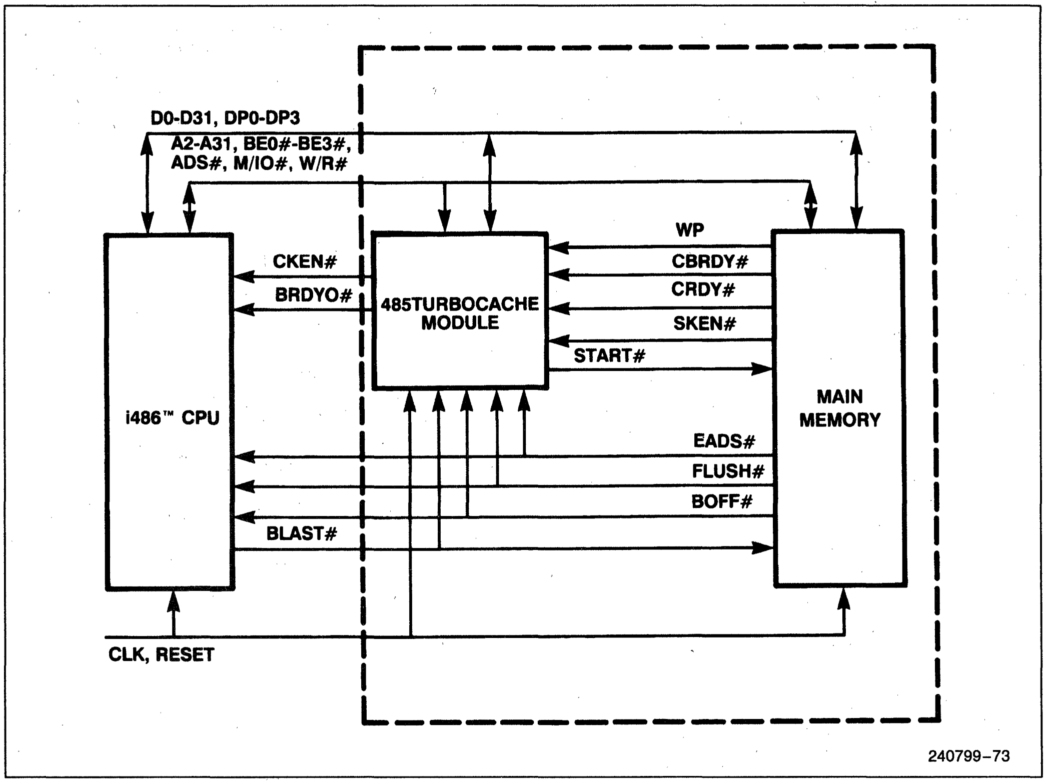


Figure 3. 485Turbocache Module and Main Memory Connections

CKEN# is used in the formation of the KEN# signal to the i486 CPU. Therefore, CKEN# is always activated in T1 (see Figure 4 and Figure 5). If a read hit occurs, data can be sent to the i486 CPU in zero wait states and can still be cached in the processor's on-chip cache. The 485Turbocache Module asserts CKEN# which remains asserted for the duration of the read hit cycle (unless WPSTRP# is low and the line is write protected). This means that the i486 CPU will cache the entire line unless external logic is added to cause the KEN# signal to be sampled high in the clock before the last BRDY0# from the 485Turbocache Module.

If the CKEN# input from the 485Turbocache Module is connected directly to the KEN# input of the i486 CPU, then the CPU will always sample KEN# active at the end of T1. To deassert KEN# to the processor, the system must create another signal that is used in the formation of the i486 CPU's KEN#, and the 485Turbocache Module's SKEN#. Using this technique a non-cacheable, non-burst cycle can be performed.

The BRDY# signal to the i486 CPU can be generated from many sources. Therefore, the various signals should be logically "ORed" to generate the actual i486 BRDY# input.

On a cache read hit, the 485Turbocache Module generates a BRDY0# signal for each of the doublewords it transfers. The 485Turbocache Module asserts BRDY0# in the first T2 cycle, and BRDY0# remains asserted for the duration of the burst. If the i486 CPU either terminates a burst early or fails to generate a burst cycle as defined by BLAST#, the 485Turbocache Module will deassert BRDY0# after the i486 CPU has sampled the required data.

Write Cycles and I/O Cycles

The 485Turbocache Module is a write-through cache, so main memory is updated with every write hit or miss. The 485Turbocache Module is not required to generate a ready signal to the i486 CPU for write cycles. However, it does perform a comparison and updates the cache memory when a write hit occurs (provided the location isn't write protected). The 485Turbocache Module is not updated on write misses. The timings for write operations are shown in Figure 4 and Figure 5.

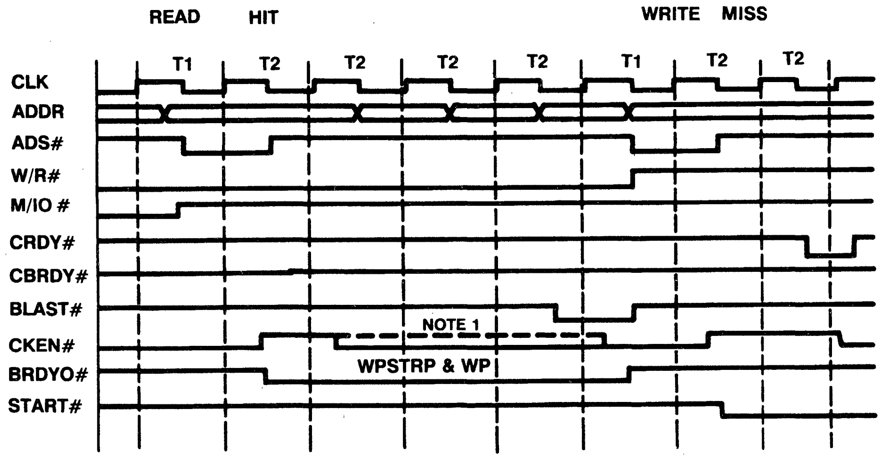


Figure 4. Read Hit-Write

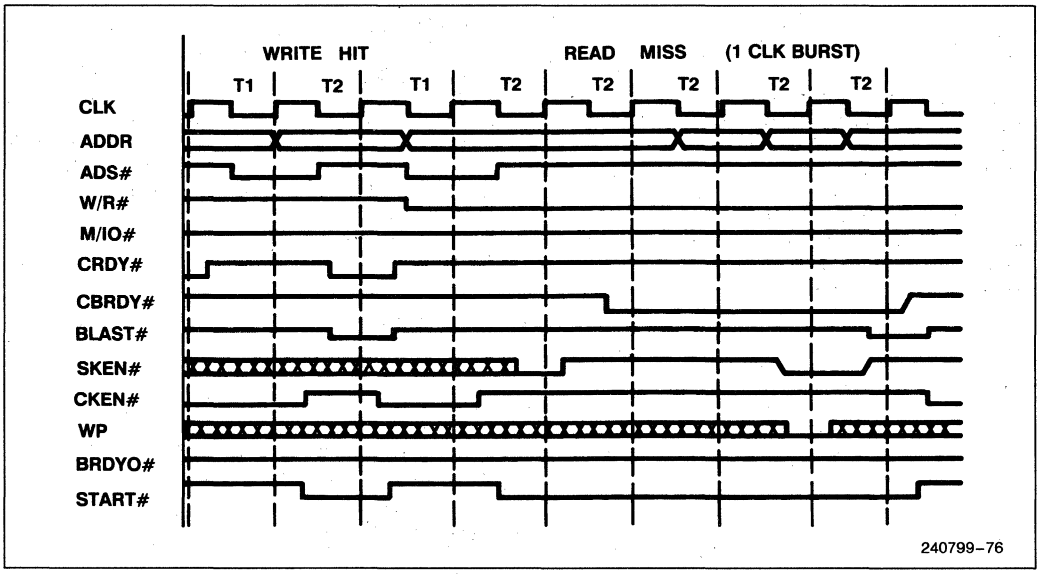


Figure 5. Write-Read Miss

Because the 485Turbocache Module is a write-through cache, writes are immediately forwarded to the system. If a processor write occurs on a valid entry that is not write protected, the new data will be stored into the memory in zero wait states. The 485Turbocache Module will not generate a ready signal. It is the systems's responsibility to update the system memory on all writes and to terminate all cycles with a ready signal. Even after the 485Turbocache Module has completed its internal write update, it remains idle until the system returns a ready to the processor.

A cache location can be write protected by asserting the WP input to the 485Turbocache Module. The WP signal must be valid during the third BRDY0# or RDY# of a cache line fill cycle. It sets a state bit within a particular cache location and remains in effect until the bit is invalidated. Tieing WPSTRP# low will not allow the write protected entry to be cached by the i486 CPU in subsequent accesses. The entry can be invalidated by any of the following: a flush operation, a reset operation, an invalidation cycle, or an LRU replacement.

When an i486 CPU cycle produces a write hit to a write-protected 485Turbocache Module location, data in the cache is not modified. The 485Turbocache Module responds in the same way whether or not a write hit location is write protected by asserting the START# signal. It is the designer's responsibility to prevent inconsistencies between the 485Turbocache Module and main memory when using the WP signal.

The 485Turbocache Module ignores all I/O cycles. When an I/O cycle is executed by the i486 processor, the system responds and terminates the cycle. The 485Turbocache Module does not assert the START# signal for I/O accesses, and the system should monitor the M/IO# signal rather than wait for the assertion of the START# signal.

System Cacheability Indication

The 485Turbocache Module uses the cache enable scheme of the i486 CPU. A cache update to the 485Turbocache Module requires activating the SKEN# signal. The signal is sampled twice, first on the rising clock edge before the first ready signal from BRDY# or RDY#, and again on the rising clock edge before the last ready. If SKEN# was deasserted at either of the specified sample times, then the access is considered non-cacheable. SKEN# is ignored during write cycles.

Typically, the system will use the same logic to generate the i486 CPU's KEN# signal and the 485Turbocache Module requires activating the SKEN# signal. However, it is not necessary for both to be asserted during an access. It is possible to use different cacheing maps for the CPU cache and the 485Turbocache Module cache because the i486 CPU and the 485Turbocache Module maintains their own cache contents via snooping.

Cascadable Cache

The 485Turbocache Module can be cascaded to configure a deeper cache memory for the processor. Up to four can be used to provide as much as 512 kbyte of cache.

System Control Signals and Cascadable Caches

The START# signal used by memory is the logical OR for each individual 485Turbocache Module START# output. If any cache has information that is needed by the processor, then its START# signal is at a high level, and it inhibits the main memory START# signal (as there is no need to access the main memory). If needed data is not present in any of the 485Turbocache Modules, then the START# signals are low, and main memory data is accessed.

The KEN# input to the i486 processor should be a logical OR for each of the 485Turbocache Modules and for a memory controller output. The memory controller output can be asserted high to indicate that the information to the i486 CPU is non-cacheable.

The SKEN# signal is the cache input to the 485Turbocache Module. The memory controllers must assert SKEN# when a transfer to the 485Turbocache Module is cacheable. The SKEN# inputs for all of the 485Turbocache Modules must be tied together. The controller that has its CS# asserted determines which cache will receive the information.

The EADS# signal from the memory controller must be connected to the i486 CPU and to all of the 485Turbocache Modules. In this way, invalidation cycles are executed in all the 485Turbocache Module devices simultaneously.

The entire memory space is covered in a single cache or a cascaded cache configuration. When multiple 485Turbocache Modules are used, only one 485Turbocache Module is selected by asserting the CS# pin.

For example, TAO through TA15 are always connected to A16 to A31. In the configuration with one 485Turbocache Module, the chip select is grounded. In the two 485Turbocache Module configurations, A16 is used to decode between the two caches. In the four 485Turbocache Module configurations, A16 and A17 are used to generate the CS# signals.

3.0 PROCESSOR FEATURE REVIEW

The improvements made to the CPU bus interface obviously impact the memory subsystem design. It is important to understand the impact of these features before attempting to define the system. This section is a review of the bus features which affect the memory interface. The features and their impact on memory system design is discussed.

3.1 The Burst Cycle

The i486 CPU's burst bus cycle feature has more impact on the memory logic than any other feature. It is the most significant departure from previous bus architectures. A large portion of the control logic is dedicated to supporting this feature. The second level cache is also primarily dedicated to supporting burst cycles.

To understand why the logic is designed this way, we must first understand the function of the burst cycle. Burst cycles are generated by the CPU if, and only if, two events occur. First, the CPU must request a cycle which is longer in bytes than the data bus can accomodate. Second, the BRDY# signal must be activated to terminate the cycle. When these two events occur a burst cycle will take place. Note that this cycle will occur regardless of the state of the KEN# input. The KEN# input's function is discussed in the next section.

With this definition we see that several cases are included as "burstable". Some examples of burstable cycles are listed in Table 3. These cycle's length is shown in bytes to clarify the case listed.

Table 3

Burst Bus Cycle	Size (bytes)
All Code Fetches	16
Descriptor Loads	8
Cacheable Reads	16
Floating Point Operand Loads	8
Bus Size 8(16) Writes	4 (max)

The last case shows that write cycles are burstable. In this case a write cycle is transfered on an 8 or 16 bit bus. If BRDY# is returned to terminate this cycle the CPU will generate another without activating ADS#.

Using the burst write feature has debatable performance benefit. Some systems may implement special functions which benefit from the use of burst writes. However, the 486 CPU does not write cache lines. Therefore, all write cycles are 4 bytes long. Also, most of the devices which use dynamic bus sizing are read only. This fact further reduces the utility of burst writes.

Due to these facts, the design example used here does not implement burst write cycles. In fact, the BRDY# input is only asserted during main memory read cycles and cache hit cycles. RDY# is used to terminate all memory write cycles. RDY# is also used for all cycles which are not in the memory subsystem or are not capable of supporting burst cycles. The RDY# input is used, for example, to terminate an EPROM or I/O cycle.

3.2 The KEN# input

The primary purpose of the KEN# input is to determine whether a cycle is to be cached. Only read data and code cycles can be cached. Therefore, these cycles are the only cycles affected by the KEN# input.

Figure 6 shows a typical burst cycle. In this sequence the value of KEN# is important in two different places. First, to begin a cacheable cycle KEN# must be active the clock before BRDY# is returned. Second, KEN# is sampled the clock before BLAST# is active. At this time the CPU determines whether this line will be written to the cache.

The state of KEN# also determines when read cycles can be bursted. Most read cycles are initiated as 4 byte long from the CPU's cache unit. When KEN# is sampled active the clock before BRDY# or RDY# is returned, the cycle is converted to a 16 byte cache line fill by the bus unit. This way, a cycle which would not have been bursted can now be bursted by activating BRDY#.

Some read cycles can be bursted without activating KEN#. The most prevalent example of this type of read cycle is code fetches. All code fetches are generated as 16-byte cycles from the CPU's cache unit. So, regardless of the state of KEN#, code fetches are always burstable. In addition, several types of data read cycles are generated as 8-byte cycles. These cycles, mentioned previously, are descriptor loads and floating point operand loads. These cycles can also be bursted at any time.

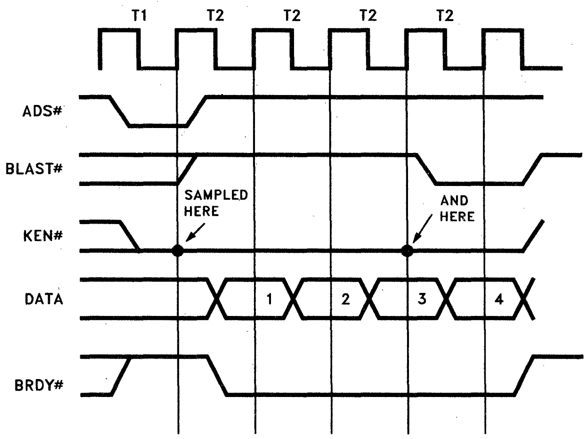


Figure 6. Typical Burst Cycle

It's obvious that the use of the KEN# input affects performance. The design example used here illustrates one way to use this signal effectively.

The primary concern when using KEN# is generating it in time for zero wait state read cycles. Most main memory cycles will be zero wait state if a second level cache is implemented. In this example, the main memory is one wait state during most read cycles. Any Cache access will take place with zero wait states. KEN# must, therefore, be valid during the first T2 of any read cycle.

Once this requirement is established, a problem arises. Decode functions are inherently asynchronous. Therefore, the decoded output which generates KEN# must be synchronized. If not, the setup and hold times of the CPU will be violated and internal metastability will result. With synchronization, the delay required to generate KEN# will be at least three clocks. In this example 4 clocks are required. In either case the KEN# signal will not be valid before BRDY# is returned for zero or one wait state cycles.

This problem is resolved if KEN# is made normally active. Figure 7 illustrates this function. In this diagram KEN# is active during the first two clocks of the burst cycle. If this is a data read cycle, KEN# being active at this time causes it to be converted to a 16 byte length. The decode and synchronization of KEN# takes place during the first two T2 states of the cycle. If the cycle turns out to be non-cacheable, KEN# will be deacitvated in the third T2. Otherwise KEN# will be left active and the data retrieved will be written to the cache.

Some memory devices may be slow enough that 16-byte cycles are undesireable. In this case more than three wait states will exist. The KEN# signal can be deactivated prior to returning RDY# or BRDY# if three or more wait states are present. As a result these slow cycles will not be converted to 16-byte cache line fills.

3.3 Bus Characteristics

The internal cache causes other effects which impact the memory subsystem design. Perhaps the most obvious of these is the effect on bus traffic. The fact that the internal cache uses the write-through policy dramatically increases the number of write bus cycles. Fig. 8 illustrates this effect. The top chart shows the bus cycle mix for an application executed with the 386DX CPU. The bottom chart shows the same application executed with the i486 CPU. The percentage of write bus cycles jumps to 70% from 30% when this application is executed with the i486 CPU.

It seems intuitively obvious that many of these write cycles would be consecutive. In fact, 70% of all write cycles are consecutive. Furthermore, 50% of all write cycles occur three in a row. It is obvious from these statistics that optimizing the memory subsystem for write cycles can improve performance. But it is important to optimize the memory system for consecutive write cycles. Improving individual write cycle latency will not buy much performance if subsequent write cycles suffer.

A technique called write posting proves ideal for this purpose. This technique allows consecutive write cycles to be overlapped. It also allows write cycles to be overlapped with second level cache cycles and reduces overall write miss latency.

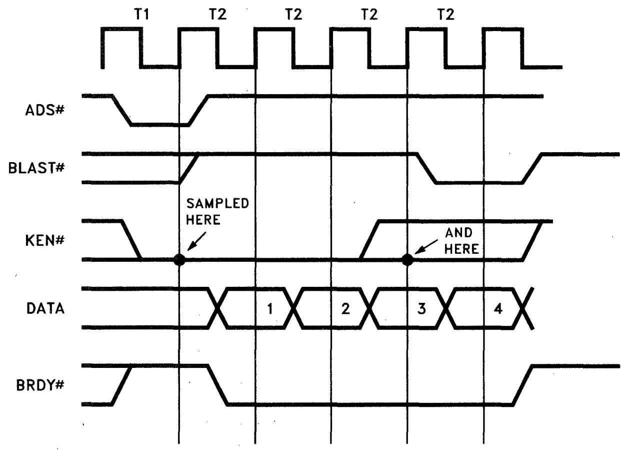
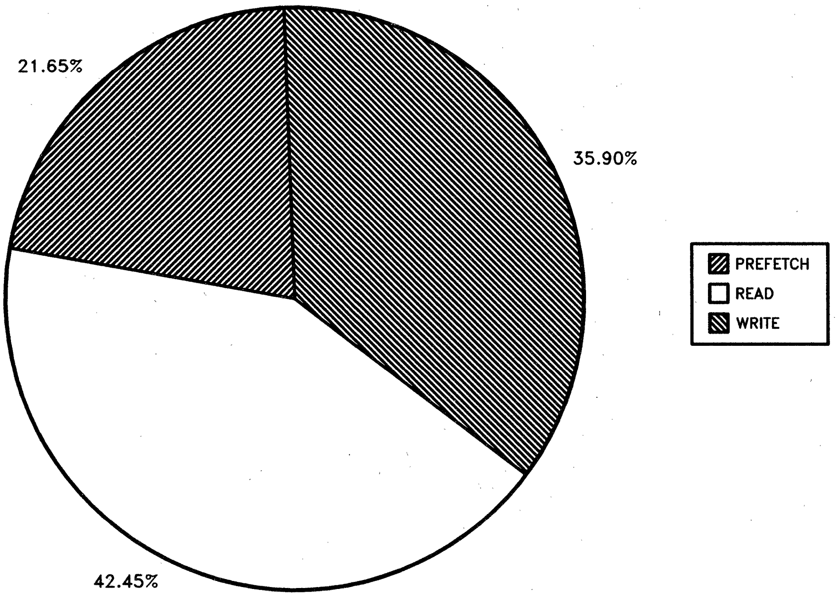
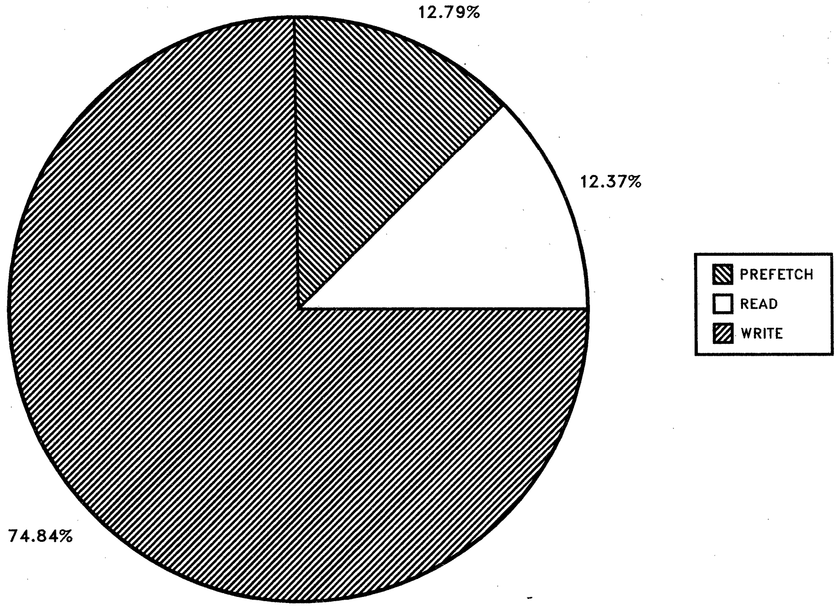


Figure 7. Burst Cycle KEN Normally Active



386DX CPU Bus Cycle Mix



i486 CPU Bus Cycle Mix

Figure 8. CPU Bus Cycle Mix

Using the write posting technique adds complexity to the system logic. It is therefore valid to ask what performance improvement is gained by using this technique. This question is especially pertinent when we consider the logic already implemented in the i486 CPU to improve write performance. The internal i486 write buffers decouple the processor execution unit from the external bus.

Analysis has shown that, in general, 6% degradation in performance can be expected for every additional wait state added to write cycles. This analysis was performed by measuring the CPU clocks required to execute several applications.

The same analysis has shown that write posting reduces average write latency to 2.5 clocks. Without write posting average write latency is 4 clocks. From this data we can conclude that approximately 9% performance improvement can be obtained by using write posting. This improvement may increase due to other affects. These affects, such as overlapping write cycles with cache reads, are discussed in subsequent sections.

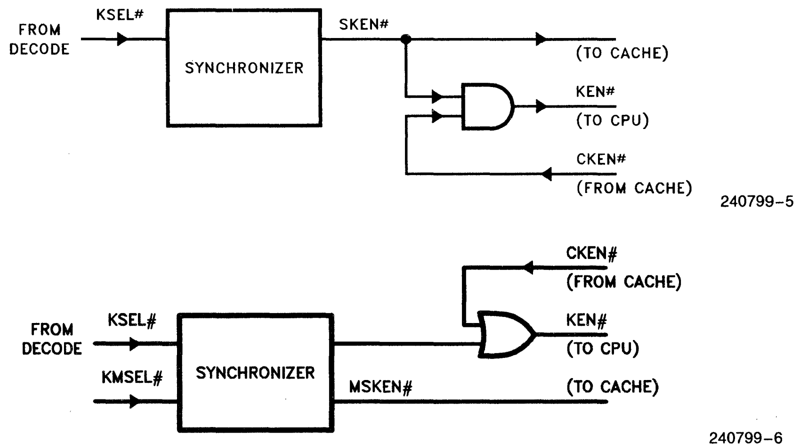


Figure 9. KEN# Logic for Second-Level Cache

4.0 DRAM INTERFACE OVERVIEW

The i486 CPU bus interface unit integrates several functions which improve the memory access rate. These features must be supported by the memory subsystem to provide the intended performance benefit. They are supported by the memory subsystem example. The example also includes logic support for a second-level cache. An overview of the subsystem is presented in this section. Details of the function and logic design of this subsystem are presented in later sections.

This subsystem follows a modular design. Only minor changes to particular logic sections are needed to implement variations. For instance, the PLD which generates the CAS# signal needs only minor changes to support Static Column mode DRAMs. It is also simple to implement a non-interleaved DRAM controller based on this design.

Other possible optimizations will be pointed out throughout the discussion. This first section summarizes the features and functions present in the design example presented in this section.

4.1 Functional Blocks

Two common design techniques are employed in interfacing the i486 CPU to DRAMs. The first, interleaving, is used to support the burst bus feature. The second, write posting, is used to reduce write cycle latency. Both techniques improve performance, and without them, performance is degraded by the access requirements of currently available DRAMs.

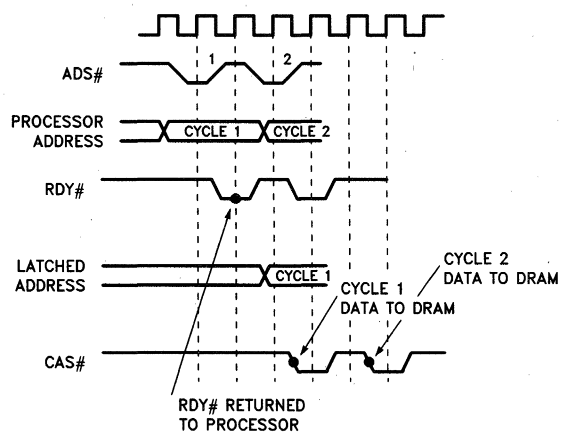
Interleaving can be implemented in several ways. Here, alternate 32-bit DRAM banks are accessed.The bank accessed is determined by the value of A2. In this way,even DWORDs (A2 = 0) are stored in one bank while odd DWORDs (A2 = 1) are stored in the other. When data is retrieved from memory during a cache line fill, cycles are overlapped to allow single clock DWORD accesses. Timing of this operation is detailed in the next section.

A multiplexor alternates data flow between the DRAM banks and the appropriate data path is selected according to the value of A2. The multiplexor prevents bus contention.

With write posting, bus cycles are again overlapped to reduce latency. Figure 10 illustrates how this technique is applied within the write cycle. The RDY# signal terminates the cycle in the clock after ADS# becomes active. This creates a zero- waitstate write cycle, the fastest possible.

When the cycle terminates, however, data must still be written to memory. The delay allows additional DRAM access time. Figure 10 shows that data is actually written to memory two clocks after RDY# is returned to the CPU. The CAS# signal completes the write cycle four clocks after it is started by the CPU.

Write data and address registers support the posted write function by holding write data and address after RDY# is returned to the CPU. These registers are required to allow the CPU to start another cycle immediately following the first (see Figure 10). ADS# is activated in the clock after RDY# is returned to the CPU. This cycle starts before the first is complete, and the cycles overlap by two clocks.



240799-7

Figure 10. Write Posting

In effect the write cycle completes in two clocks. Write cycles can be overlapped in this manner indefinitely. The timing and logic required to support this function is described in Section 5.3.

Address registers also support invalidation with the AHOLD signal. They are required if AHOLD is activated when bus are cycles in progress to hold the current address while the bus cycle completes.

The efficient CPU interface and invalidation support make this DRAM subsystem well-suited for use with an optional cache. The memory system includes specific functions designed to support the optional 486 Turbocache module. The subsystem supports 256K × 4 and 1Mbyte × 1 DRAM configurations.The minimum memory configuration is 2 Mbytes with 256K × 4 devices; the maximum is 16 Mbytes with 1Mbyte × 1 devices. Additional banks can be added to increase the memory capacity.

The control logic for this example is implemented with EPLDs.The modular approach allows quick modification so that the example can be tailored for specific implementation requirements.

The control state machine is distributed among the various EPLDs, and each functional block receives control input from other blocks. In addition most of the functional blocks are implemented as state machines.

Figure 11a is a top level block diagram of the memory system. This diagram depicts the sections of logic that will be described subsequently. We will first discuss the address path logic.

4.2 Address Path Logic

Unlike processors without on-chip caches, the address bus of the i486 processor is bidirectional. The address pins serve as inputs whenever external memory is changed by DMA or another CPU. The address is driven into the CPU to invalidate the corresponding cache entry if present.

Invalidation of the 486 CPU's internal cache can be performed in several different ways. This example supports invalidation cycles during a memory access.

As described in the previous section, AHOLD is used to perform the invalidation function. AHOLD tristates the 486 address bus. Address registers must be used to hold the address to allow the current bus cycle to be completed. These registers hold the current address when AHOLD is activated.

The registers shown in Figure 11b hold the entire row and column address, as well as the current byte enables and control definition. These signals are latched at the rising clock edge of the first T2 of a bus cycle. They must be held from this edge to allow zero wait state write cycles.

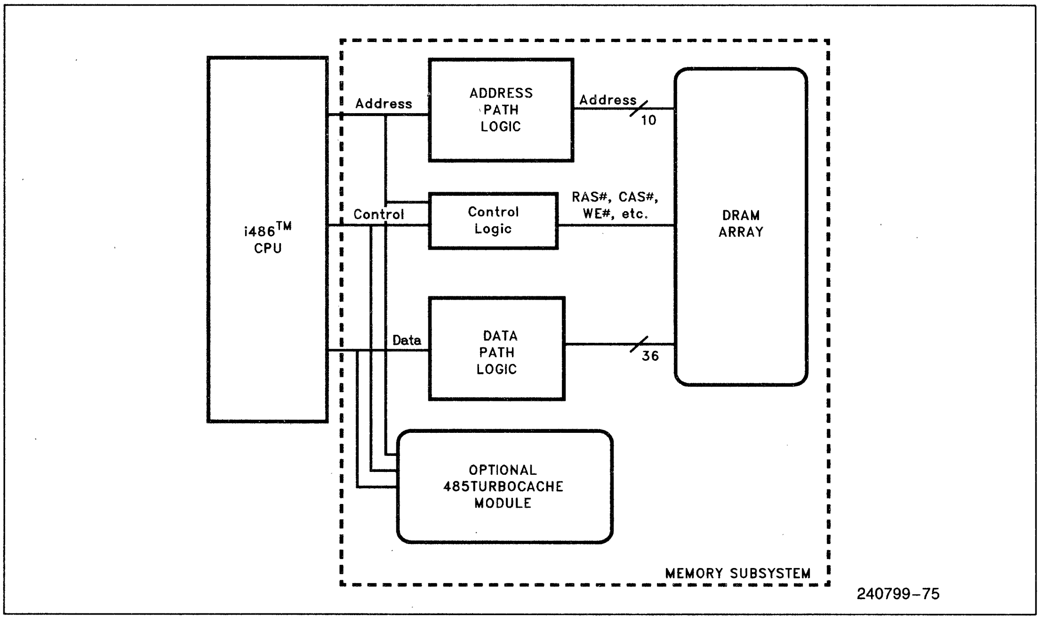


Figure 11a. Memory Subsystem Block Diagram

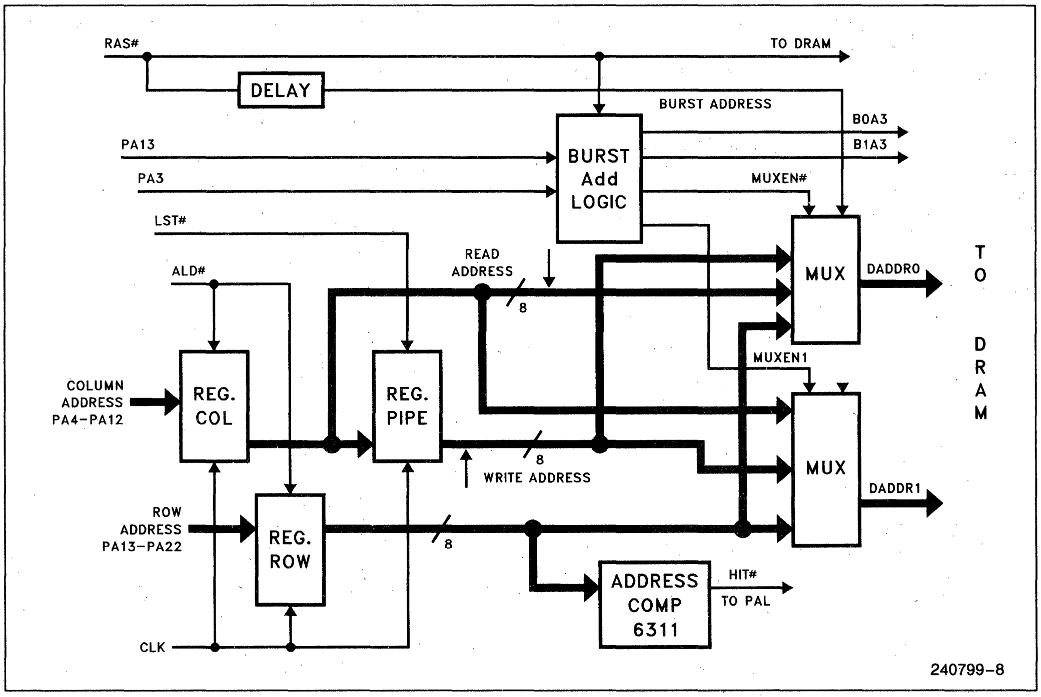


Figure 11b. Address Path Logic

Registers with enable inputs are needed. The enable input can select the CLK edge appropriate for latching the address and control state. The control logic generates the enable signal ALD which disables the CLK input of the registers during a bus cycle. When ALD is active (High) the current row and column addresses are held in the registers. 74AS823 registers have enable inputs and are used in this example.

An additional address register is required for posted write cycles. This register holds the write column address. The address is latched only on write cycles and is held until the write cycle completes at the DRAM.

Separate write and read address paths are implemented with a 3 to 1 address multiplexor. The read address path is required to meet the timing of a three CLK read cycle. In this case the read address must propagate through the address mux one CLK sooner than the write address. If the initial read access is 4 CLKs long the read and write address paths can be combined. See section 5.1 for a complete description of read cycle timing. The third address path is for the row address.

A delay line is used to meet the row address DRAM hold time requirement(tRAH). The RAS# signal is delayed 20ns to create the DRAS# signal. This signal is used as the multiplexor path select input. When DRAS# is inactive (high) the multiplexor always selects the row address path. When DRAS# is active (low) the mux enable signal (MEN0# or MEN1#) controls whether the read path or the write path is selected.

The comparator and register combination is connected to the row address path to generate the HIT# signal. This signal indicates that the current cycles address is in the same DRAM row as that of the previous cycle and also determines whether RAS# will be deactivated.

In this example a standard component designed specifically for this purpose is used. This component contains a register and a comparator. The register in this component holds the previous row address. When a bus cycle occurs to a new DRAM row, the new row address is latched. The RALE signal enables the row address latch.

The timing of this component meets the requirements of a 33 MHz CPU clock. Discrete registers and comparators can be used to improve the timing of the HIT# signal, if desired.

The last important address logic component is the burst address generator. This state machine generate A3 and A2 during burst accesses and is needed to achieve zero wait state performance during burst cycles. It predicts the value of A2 and A3. Section 5.6 contains a complete description of the burst cycle timing.

Note that because interleaving is used, A3 is the lowest order DRAM address. Two A3 equivalent signals are generated. One for Bank 0 (B0A0) and one for Bank B1A0. These signals are connected directly to the DRAM devices to meet critical timing requirements. The signals must also reflect the lowest order row address during miss cycles. As a result A13 is, therefore, an input to this logic. It is the lowest order row address when 1MBx1 DRAMs are used.

4.3 Data Path

A2 must also be predicted during burst read accesses. For this purpose, the burst address logic creates the DATASEL signal. DATASEL reflects the value of A2 for each access of a burst cycle and is used to control the data multiplexor as shown in Figure 12.

During burst cycles, the data multiplexor alternates between the bank 0 and bank 1 data paths. A2 must alternate states each clock for interleaving to function properly. The i486 CPU's burst address sequence is defined such that A2 changes state on every access.

A2 also selects the bank to which data is written. Data path logic is not involved in steering data during writes. Figure 12 shows separate data registers for each bank. Separate registers are only required to divide the data paths. These registers hold the same write data on every write cycle. The CAS# and WE# (write enable) signals control doubleword and byte steering.

Because of write data timing, the data registers must have the enable function. This function, can be used to select the clock upon which data is latched. The processor clock can be used as the register clock input to guarantee proper data setup and hold times.

As Figure 12 indicates, the MRDY# signal enables the write data registers and terminates memory write cycles. Data is therefore latched during the last clock of any write cycle.

MRDY# is restricted to write cycles while the MBRDY# signal is used for read cycles. The need for these signals illustrates the convenience of the CPU's dual-ready inputs. The MBRDY# signal enables the output of the data path multiplexor to prevent bus contention.

These ready signals are combined with similar system logic signals to form the processor RDY# and BRDY# inputs. I/O, peripheral and other non-burst devices can use the RDY# input. Burst devices, such as a second level cache controller must also use the BRDY# input. The MBRDY# and MRDY# signals are, therefore, used only with the DRAM control logic. They are isolated from the rest of the system by combinatorial logic.

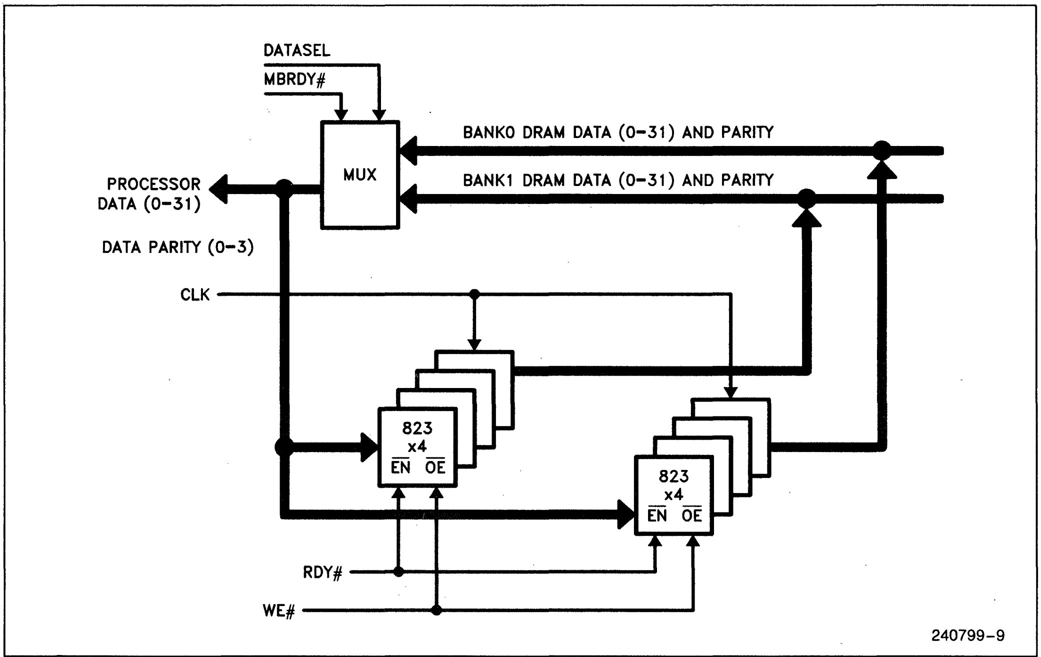


Figure 12. Data Path Logic

4.4 Second Level Cache Support

Second level cache strategies for the i486 CPU are diverse and application dependent. The example described illustrates a second level cache strategy that is ideal for single CPU systems.

The 485Turbocache second level cache used in this example is optional and is used to complement the i486 internal cache to improve the performance when running complex applications and operating systems. Some users will not require the extra performance. Since the cache is optional, O.E.M.'s or end-users can decide whether it should be included. System board design and manufacturing costs are thus eased since one system board supports multiple performance requirements.

The 485Turbocache is a completely self contained cache module. Optionality is accomplished by including control logic, tag ram and data ram in one package. A socket is added to the system board in much the same manner as a math coprocessor socket. In systems which, for example, run UNIX, the cache module is simply plugged in.

This option must; of course, be supported by the system logic. Specifically, the memory control logic is directly interfaced to the cache module. The DRAM controller example described here is particularly well- suited for this cache configurations.

The support included in the 485Turbocache module's memory control logic for the 485Turbocache module is illustrated in Figure 13. Since the 485Turbocache is a write-through cache, provision must be made for read cycles. When read data is found in the second level cache, the cycle is called a cache hit. At the time this cycle is determined to be a cache hit, it has already been started in the DRAM controller. This cycle must be aborted by the DRAM controller.

The BRDYO# signal from the 485Turbocache module provides a convenient cache hit indication. This signal is included in the decoder function. When a cache hit occurs, the DRAM controller aborts the cycle. The memory chip select signal is not activated and the first level control logic is reset aborting the cycle. The control logic then waits for another cycle to start. This function is very similar to the back-off function.

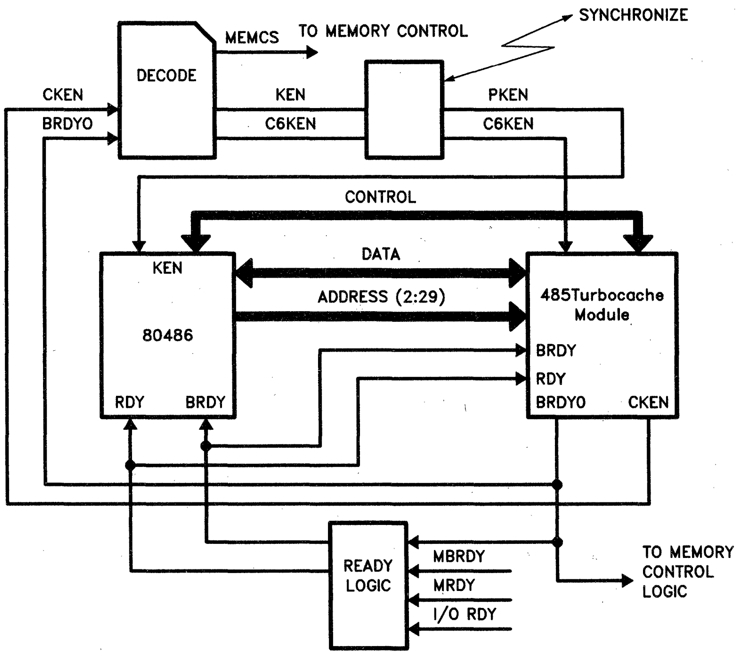


Figure 13. Logic Required for Optional 485Turbocache Module

Like the i486 internal cache, the 485Turbocache module supports non-cacheable memory by decoding. The SKEN# input is analogous to the i486 CPU's KEN# input. This function is also supported by the decode logic. Note that, as with the KEN# signal, SKEN# must be synchronized to the CPU clock.

Separate cache enable inputs also allow areas of memory to be noncacheable in the i486 CPU internal cache yet cachable in the second level cache. This feature is convienient for BIOS.

4.5 Control Logic

Memory control logic generates the signals that control the memory devices, multiplexors, and registers described earlier. These control signals can be generated in a variety of ways. This example employs a distributed state machine.

Since this example is a prototype, PLDs were the logical choice for the controller implementation. Because the number of terms in a PLD is limited, the state machine implementation must be distributed. Function distribution was determined based on this constraint. Figure 14 shows a block diagram of the controller, with each block made up of one or two PLDs.

There are two levels of logic in the controller shown in Figure 14. The first is made up of two PLDs, one which tracks bus cycles and another which generates the MRDY# signal. The first level signals to PLDs in the second level that a cycle has started. The second level is made up of several PLDs which generate the actual control signals such as RAS# and CAS#.

Implementing the controller in this manner has two important advantages. First, more decode time is allowed. The cycle start signal, CIP#, is used by the second level logic to sample the decode output. CIP# is valid in the first T2 of any bus cycle. As a result, decode does not need to be valid until the end of this T2 bus state. Without this function, the decode output must be valid at the end of every T1 bus state. In this case, the time allowed for decode at 33 MHz is very short. With 7-ns PLDs, the time allowed for decode would be 7ns. With 5-ns PLDs, this time is still only 9ns. The advantage of the extra clock period is clear.

The second advantage of the two level approach is similarly clear. The AQ0 signal indicates the start of a bus cycle to all second-level PLDs. Without this signal ADS# would have to be connected to these devices, and the resulting load on ADS# would be prohibitive.

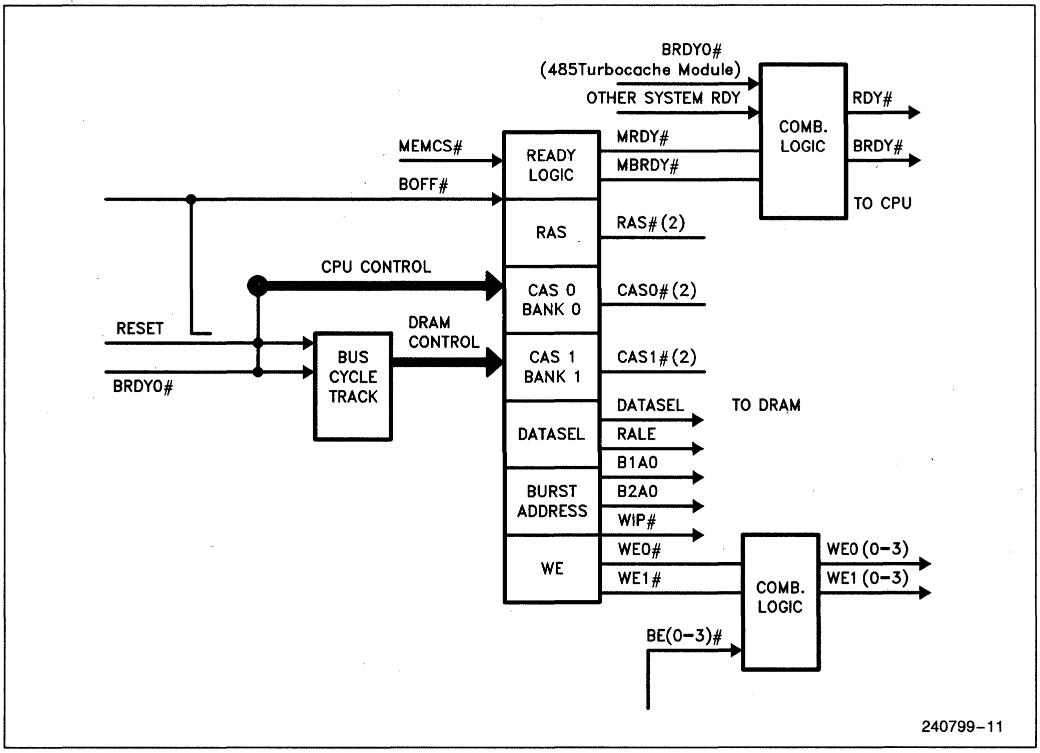


Figure 14. Control Logic Overview

Invalidation within bus cycles is another case that makes decode design difficult. The AHOLD signal must be used to implement this function. As its name implies, AHOLD can be active in any clock. If AHOLD is active in the first clock (T1) of a bus cycle, the CPU address lines are tristated in T2. Unless decode is latched at the begining of T2, it will not be valid for the DRAM cycle.

The two-level approach allows decode to be a transparent function. The decode circuit is shown in Figure 15. The 85C508 shown here includes a flow-through latch function. Using this function, the decode outputs can be latched. The DALE signal is generated at the beggining of the first T2 of any bus cycle. This signal activates the latch input of the 85C508. In this manner, decode is held during T2. If AHOLD is active in T1, the decode outputs may not be valid in T2. In this case, the cycle must not be started until the CPU address is redriven. Cycle-tracking PLD handles this function. By delaying the cycle start signal, the DRAM cycle is delayed. When AHOLD is deasserted, the CPU redrives the address again. At that time, CIP# is activated and the cycle begins. If AHOLD is active in any other clock, the bus cycle can continue normally.

The first level of interface with the memory subsystem, the cycle tracking PLD handles many other functions, most of which relate to synchronization. Refresh synchronization is one example, as is determining the RAS# precharge duration. AQ0# is not the only signal which supports the AHOLD function. Address registers, controlled by the PLD, generate the ALD signal to disable the registers during bus cycles. These and other functions of the control logic are described completely in Section 5.11.

The PLDs in the next level of logic perform more specific functions. RAS# and CAS# are generated at this level, and the PLDs that generate these signals are devoted solely to this function. The RAS# PLD generates four RAS# signals, RAS0#–RAS3#. These signals are identical but drive different DRAM modules to reduce the load on the RAS# signal.

The RAS# function is designed to support page or static column mode memory devices. To support these devices, RAS# must be left active between accesses to the same row. The RAS# state machine is designed so that RAS is deactivated only for a refresh or page miss cycle. This module generates RAS# for both DRAM banks.

For the CAS# function, the PLD's are responsible for implementing burst accesses. During write cycles, the CAS# signals determine which DRAM bank is written to. All even doublewords (A2 = 0) are stored in bank 0 while odd doublewords (A2 = 1) are stored in bank 1. When data is retrieved from memory, cycles can be overlapped. to allows zero wait state burst accesses.

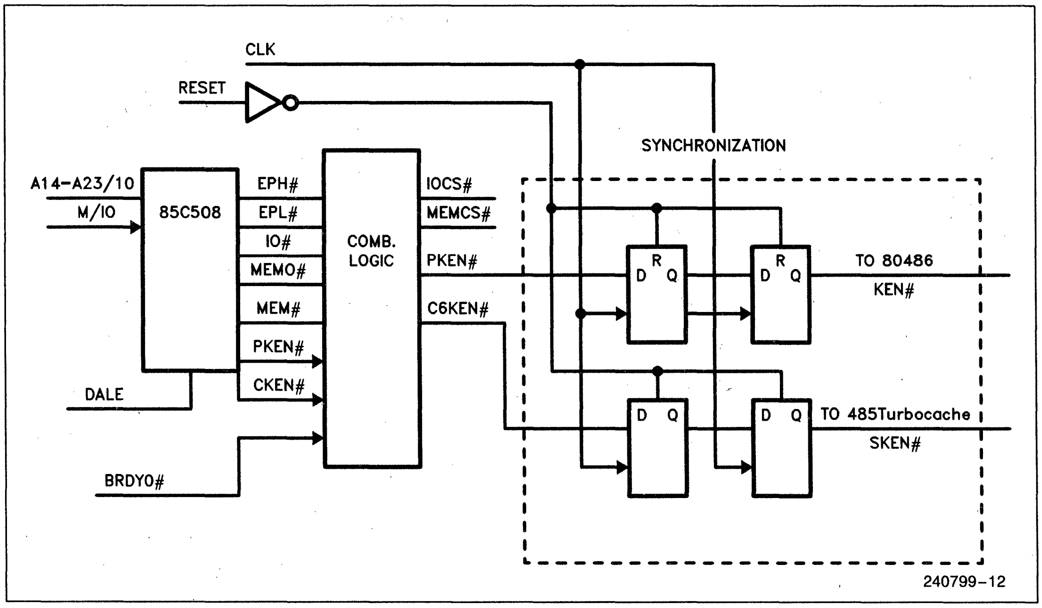


Figure 15. Decode Logic

Address generation is another important consideration in burst accesses. The address for the last three access of a burst must be generated by logic because the CPU cannot generate these addresses in time to allow zero-wait state accesses. The burst address logic shown in Figure 14 is actually two PLDs which generate the burst address for bank 0 and bank 1, respectively. The burst address consists of two signals- -the lowest order DRAM addresses from each PLD.

Because of timing constraints, these signals are connected directly to the DRAM devices. The burst address PLD must generate the burst address, provide the multiplexer function for row and column addresses and generate the write address. The burst address signals must, therefore, reflect the value of A13 during miss cycles. These reflect during burst read and write cycles. These signals reflect A3.

B00MA0 and B01MA0 are the burst address signals for bank 0. Two identical signals are used to divide loading. B10MA0 and B11MA0 are the burst address signals for bank 1. A detailed description of the burst address function is given in Sections 5.6 and 5.16.

The DSEL PLD main function is to generate the data select signal. As described above, this signal is used during a burst to switch the data path multiplexer. It reflects the value of A2 during burst read cycles only and is one component of the burst address. The DSEL PLD also generates the RALE signal to control the row address register described above.

BRDY# terminates all read cycles. MBRDY# is generated by the MRDY PLD and is separated from the RDY# signal to facilitate posted writes by preventing data bus contention. When a write cycle is immediately followed by a read, the read cycle must be delayed. This delay is implemented by delaying MBRDY# until the previous write cycle is complete. MBRDY# is combined with other burst ready inputs using combinatorial logic.

WIP# (write in progress) indicates to the MRDY PLD that a write is taking place, and MBRDY# is not generated unless this signal is inactive. WIP# tracks the state of the CAS# state machines.

The WE PLD generates WIP# and other signals associated with the write function. The MUXEN# signals control the address multiplexors and activate the write address path during write cycles. The WE# signals are used to create the DRAM W inputs and to implement byte steering. They are combined with latched CPU byte enables using combinatorial logic. In this way, DRAM W inputs are not active for unselected bytes. Data bus contention on unselected bytes is prevented by controling the write data register output enables.

By implementing byte steering in this way the CAS# logic is simplified. The CAS# timing path is critical during burst read cycles, and by placing the byte steering logic in the write enable path, CAS# timing restrictions are eased.

The MRDY# signal terminates all write cycles. The logic used to generate this signal is unusual because it uses the ADS# input and is therefore at the first level. This configuration is needed to implement zero wait state write cycles.

MRDY# must be active by the end of the first T2 to terminate a write cycle and maintain zero wait-state performance. To meet this restriction, it must be active during any write cycle, or before decode is available because the CPU RDY# signal must not be activated during non-memory write cycles, MRDY# is inhibited by the decode output, MEMCS#, in combinatorial logic.

5.0 MEMORY SUBSYSTEM FUNCTION

In this section we will explore the function of the memory subsystem in detail. Each of the signals will be described, and bus cycles will be illustrated to show the memory logic function.

The bus cycle description in this section is specific to this example. Signals such as KEN# and RDY#, for example, are shown as they are driven by this particular control logic. The signals are not restricted to the timing shown here.

A list of the memory control signals follows.

Memory Interface Signals

5.1 CPU Interface Signals

KEN#	KEN# is an input to the processor, indicating whether the next bus cycle is cacheable or not. This signal is a logical AND of SKEN# and CKEN# signals.
PBRDY#	PBRDY# is the burst ready input to the processor. This is a logical AND of the BRDY# signal from the system and the BRDYO# from the second level cache.

5.2 Data Path Control

DATASEL	DATASEL reflects the value of A2 during burst accesses. It is used to control the data multiplexor for bank 0 and bank 1 data paths.
MRDY#	MRDY# enables the write data registers that are used to support write posting and terminates memory write cycles.
MBRDY#	MBRDY# is used for read cycles and enables the output of the data path multiplexor.
WE0#/WE1#	WE0# and WE1# signals enable the outputs of data write registers used for write posting. Both the signals are active during a write and CAS# determines the correct bank to which the data is written.
WBE00#-WBE03#	WBE00#-WBE03# are a combination of write enable and byte enable signals. They control which byte is written into bank 0 during a write cycle.
WBE10#-WBE13#	WBE10#-WBE13# control which byte is written into bank 1 during write cycles.

5.3 Address Path Control

ALD	ALD disables the clock input to the registers that hold the row and column addresses corresponding to the current bus cycle.
MUXEN0#,1#	MUXEN0#, MUXEN1# control signals are inputs to the address multiplexors and are used in selecting the read or write paths to the respective banks.
RALE#	RALE# enables the row address latch, allowing a new row address to be latched for successive bus cycles.
DALE#	DALE# activates the latch inputs of the decode logic in the first T2 of a bus cycle and holds the decode during the bus cycle.
B00MA0/B01MA0	B00MA0 and B01MA0 are the burst address signals for bank 0. They correspond to the value of A3 during burst read cycles.
B10MA0/B11MA0	B10MA0 and B11MA0 are the burst address signals for bank 1. They correspond to the value of A3 during burst read cycles.

5.4 DRAM Interface

HIT#	HIT# is active if the row address for the current memory cycle is the same as the previous memory cycle.
WIP#	WIP# indicates that a write cycle is in progress and a read to the DRAM needs to be delayed till WIP# becomes inactive.
CIP#	CIP# indicates a memory cycle is in progress. If the current cycle is not to DRAM, CIP# is deactivated else it remains active till the end of the bus cycle.
RAS0–3#	RAS0–3# go active for a valid row address. It remains active between accesses to the same row and is de-activated only for page miss and refresh cycles.
DRAS#	DRAS# is the delayed RAS# signal to accomodate the RAS# hold time requirements.
RFRQ	RFRQ indicates that a refresh of the DRAM is required. This signal is activated every 15.6 us.
RFACK	RFACK is asserted as a response to RFRQ and indicates that the DRAM controller is ready to perform the refresh cycle. It is active during idle cycles or after the current cycle is complete.
PCHG	PCHG determines the timing of refresh cycles and RAS# precharge count.
CAS0#/CAS1#	CAS0# and CAS1# signals are active when a valid column address is present on the bus and control the bank to which the data is written into.
MEMCS#	MEMCS# is active when a read or a write is performed to the DRAM. It is the synchronized output of the address decoder.

5.5 Controller Signals

CT	CT indicates that a new cycle had started while a cycle was in progress or the refresh cycle was taking place. It is de-activated when the pending cycle is recognized.
SKEN#	SKEN# indicates if any of the caches is enabled. It is an input to the second level cache and is similar to the KEN# signal input to the processor.
CKEN#	CKEN# is the output of the second level cache. It is activated twice for a valid line fill - first to enable a 485Turbocache cache line fill and the second time to validate it.
LA2, LA313	LA2 and LA313 are latched versions of address lines A2 and A13. LA313 is the lowest order DRAM address line. The multiplexor output reflects A3 when RAS# is loand A13 when RAS# is high.
M#	M# indicates the occurrance of a write miss.
BRDYO#	BRDYO# is a burst ready signal driven by the second level cache. It is activated when a read hit occurs in this cache.

5.6 Read Cycles

Timing Diagram 16 shows a burst read cycle. At the start of the bus cycle, RAS# is inactive. This case is a rare occurence because RAS# is normally active. Unless a cycle is the first bus cycle after a reset or refresh cycle, RAS# will be active in T1.

It is useful to examine this case because it demonstrates a complete DRAM cycle. The basic function of most of the control logic is illustrated.

The cycle begins with the activation of ADS#. The controller samples this signal and activates both ALD and CIP#. The CPU address registers are disabled by ALD. Therefore, the previously latched address is held throughout the bus cycle. The latched address is valid in the first T2 of the bus cycle.

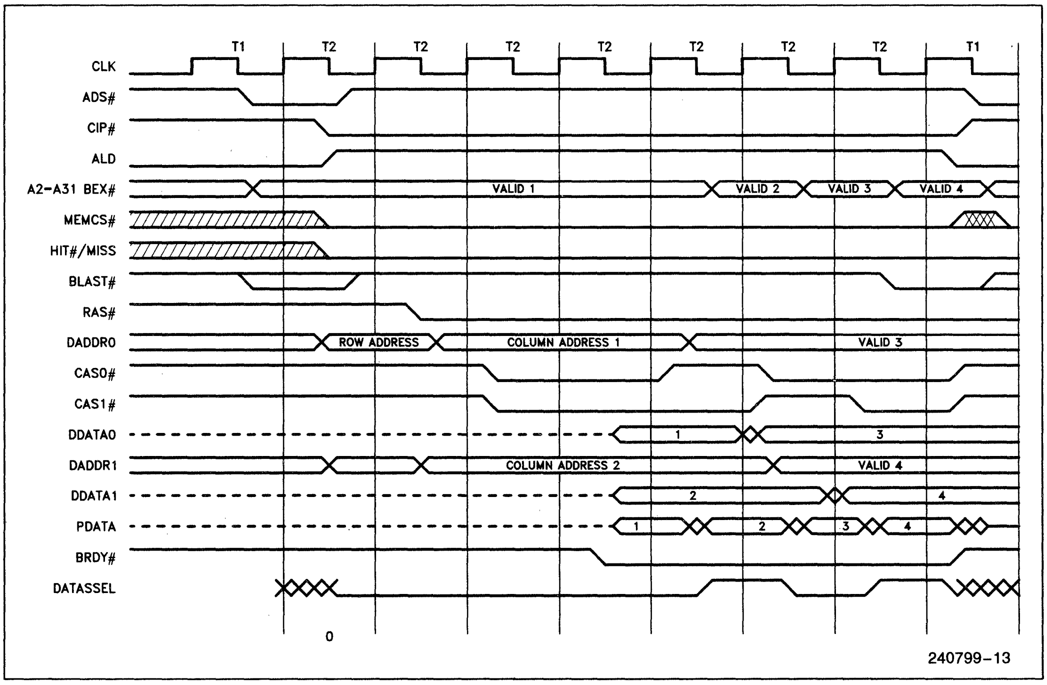


Figure 16. Burst Read Cycle

The row address comparison is made with this address. As a result, the HIT# signal is not valid until the rising edge of the second T2. At this rising clock edge, the CIP#, MEMCS# and HIT# signals are sampled. If MEMCS# is sampled active, the RAS# signal is activated.

The delay line holds the DRAS# signal high for 20 ns after RAS# is activated. In this way the row address is maintained to meet tRAH, the row address hold time. When DRAS# is activated, the address multiplexers switch to the column address path. The MUXEN# signals are not active, and the read path is selected.

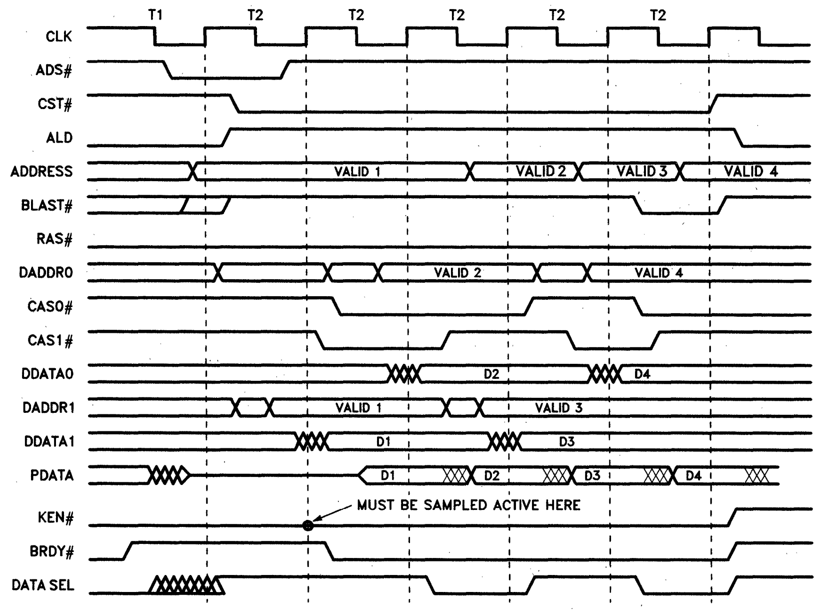
In the third T2 of the bus cycle CAS# is asserted. This cycle begins with A2 low and the first access is to bank 0. Due to the access time of the DRAM two clocks are required to retrieve data from memory. MBRDY# is asserted in the fourth T2 of the bus cycle, and this action completes the first access of the burst read. The access is completed in five clocks. The minimum time for this access is two clocks indicating that three wait-states were added to the first cycle.

The timing diagram reveals two important points about burst cycle implementation. First DRAM access requires two clocks. Second, the burst address from the CPU is not available until the clock after MBRDY# is sampled active. These circumstances make implementing zero-wait-state burst cycles difficult. The DRAM bank interleaving alleviates this difficulty.

The first advantage of interleaving is revealed in the second and third T2 states. Access to both the first and second memory doublewords can be made simultaneously. This function requires that the burst address be predicted. As mentioned above, the burst address from the CPU is not available until several clocks later. The burst address for both the first and second accesses is generated in the second T2. Therefore, CAS# for both banks can be asserted in the next T2 state.

The second advantage of interleaving is seen in fifth T2 of the burst cycles in which DATASEL switches the data multiplexer. The second doubleword is driven on the CPU data bus. In this CLK, the burst address for the third access of the cycle is generated. CAS00# and CAS01# are also deasserted to begin the third access. Note that this access is started before the second access is completed. The cycle overlap shown allows new data to be driven on the CPU data bus every clock. This way zero-wait-state access is achieved.

Timing is even more critical during page hit cycles. Fig. 17 shows the timing of this cycle. Because of the function of RAS#, this cycle is more common than the cycle discussed above. The row address is the same as in the previous cycle. Therefore, the RAS# signal is left active.



240799–14

Figure 17. Burst Read DRAM Page Hit

When a burst read starts with RAS# active, fewer clock, are required to complete the first access. This reduction improves performance. As a result, however, some timings become more critical. One of these is the time allowed to generate the burst address.

The CAS# signals are asserted in the second T2 of the bus cycle. MBRDY# is also asserted at this time. To meet the address access time of the DRAMS, the burst address must be generated in the second T2. The rest of the read column address must also be available at this time. Two logic functions are needed to meet this timing requirement. First, read and write address paths must be separate to allow the read address to be available in the first T2. Second, the burst address path logic must latch the CPU A3 signal directly. In this way, the logic can generate the necessary address in time. The burst address state machine must track the state of A3 at the begining of every cycle. The state machine function is described in Section 5.11.

The timing of KEN# must also be considered in this example. KEN# must be valid at the begining of the second T2 of the cycle. If it is not, the cycle will not be cached, and a 16-byte access can not be generated. If KEN# is active, a 16-byte burst access will be generated, and the cycle will be cached as long as KEN# is active in the second to last T2.

At first glance this timing may not appear critical. KEN# is a decode function, and decode is valid at the clock edge called for. The KEN# input to the CPU must be synchronized to clock, however. Since decode is not synchronous, a two-clock synchronizer delay is required, and this delay is the reason that KEN# is normally active in this example.

From the time CAS# is activated, this cycle is exactly the same as in the previously described burst cycle. It is terminated when BLAST# is asserted, and MBRDY# is deasserted when BLAST# is sampled active.

5.7 Write Cycles

As described in Section 4.1, a posted or delayed write function is employed in this example to reduce write cycle latency. Latency is reduced since write cyles are overlapped with other cycles including other Write cycles or reads from the second level cache. Write cycles normally make up 70 percent of all cycles, and overlapping can increase performance accordingly.

Figure 18 illustrates the posted write implementation. In this example cycles begin when RAS# is inactive. As with read cycles, this case is rare in practice.

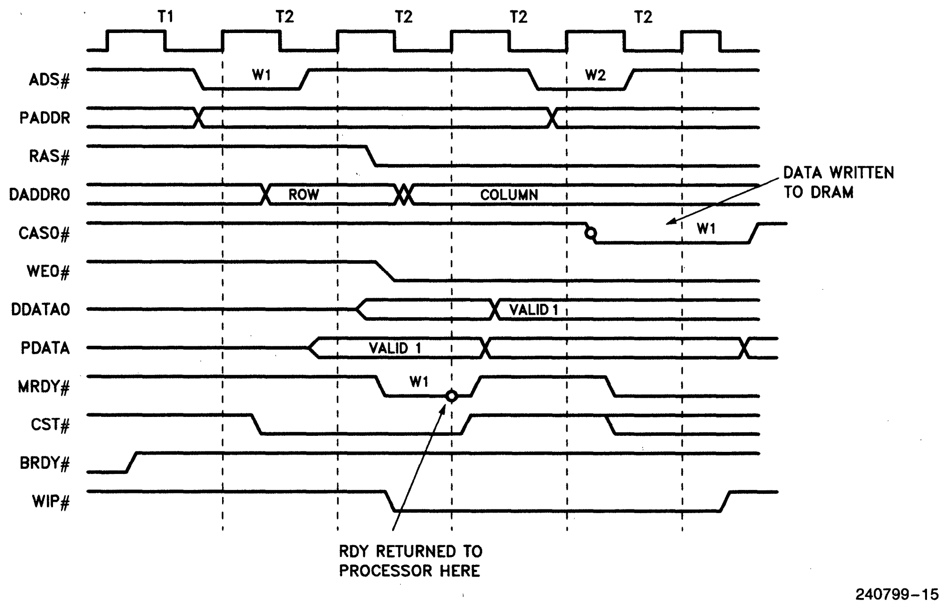


Figure 18. Basic Write Cycle

The cycle begins like a read. The CPU drives ADS# active, and the decode is sampled. RAS# is activated if the cycle is in DRAM space. In the second T2 of the cycle, however, the latched version of W/R# (LW/R#) is sampled active at the rising edge of the second T2. In response, the control logic begins several write cycle functions at this clock edge.

The CAS# state machine for the appropriate bank enters the write sequence. The MUXEN# and WE# signals are asserted. MRDY# is also asserted, terminating the cycle at the CPU. The MUXEN# signals activate the write address path. This address is not present at the multiplexor outputs, however, until the next clock at which the write pipeline register latches the write address.

The write data is latched at the same clock edge. The write data registers are enabled by MRDY# which simultaneously terminates the CPU cycle. Note that data is latched in both the bank 0 and bank 1 registers.

The WE0# and WE1# signals are also both active. The CAS# signals determine which bank is written to. These signals are asserted within two clocks after MRDY#. This action completes the write cycle. Note that, while five clocks are required clocks are required to complete the cycle, the CPU cycle is terminated in three CLKs. The wait state is only required if RAS# is inactive at the start of the cycle.

In Figure 18 the next bus cycle starts immediately after RDY# is sampled. In this case, CAS# is activated during the second clock of the next bus cycle. This overlap of cycles is similar to the pipelining feature used by many processors except that the i486 processor bus is not involved in the posting function. All logic for this function is implemented in the memory controller.

Figure 19 is a more typical i486 processor bus sequence which clearly illustrates the advantages of the posting technique. Four write cycles have occurred together without idle bus clocks occurring between cycles. Since all writes access the same DRAM row, RAS# is active throughout the sequence.

Without the extra clock to activate RAS#, MRDY# can be asserted in the clock after ADS# is asserted. These cycles, therefore, have no wait-states. As before, the write cycle is not complete when MRDY# is asserted but instead when CAS# is asserted two clocks after MRDY# to terminate the CPU bus cycle.

At zero wait-states, each write cycle still requires four clock cycles. The last two clocks of each write cycle overlap with the next cycle. The net effect on the CPU bus is the same as a string of two-clock write cycles, as illustrated in Figure 19.

The first write in this figure is to bank 0. The falling edge of CAS0# clocks the data into the bank 0 DRAM. This edge is denoted by W1 in the diagram.

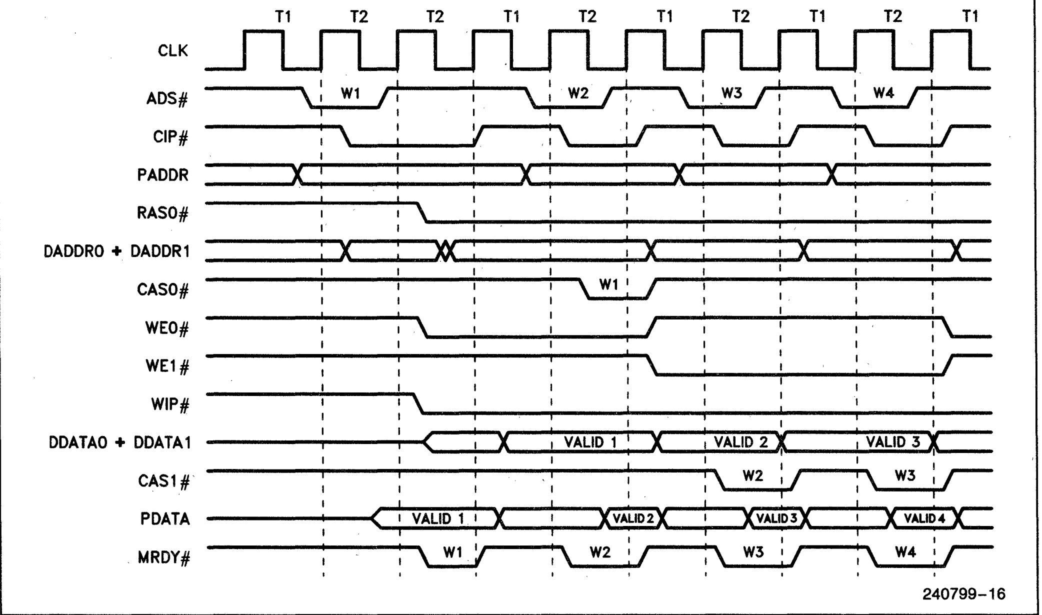


Figure 19. Back to Back Write Cycles

CAS0# is asserted in the same clock that MRDY# terminates the second write (W2), which accesses bank 1. CAS1# is activated in the same clock as MRDY# for the third write (W3).

The second and third writes happen to be to the same DRAM bank. As we see, no timing modification is required in this case. Write cycles can be completed with zero wait states in either case. This is important since writes often occur in sequence on the i486 bus, but not necessarily to sequential addresses. Write posting supports zero wait-state write cycles to sequential and non-sequential addresses.

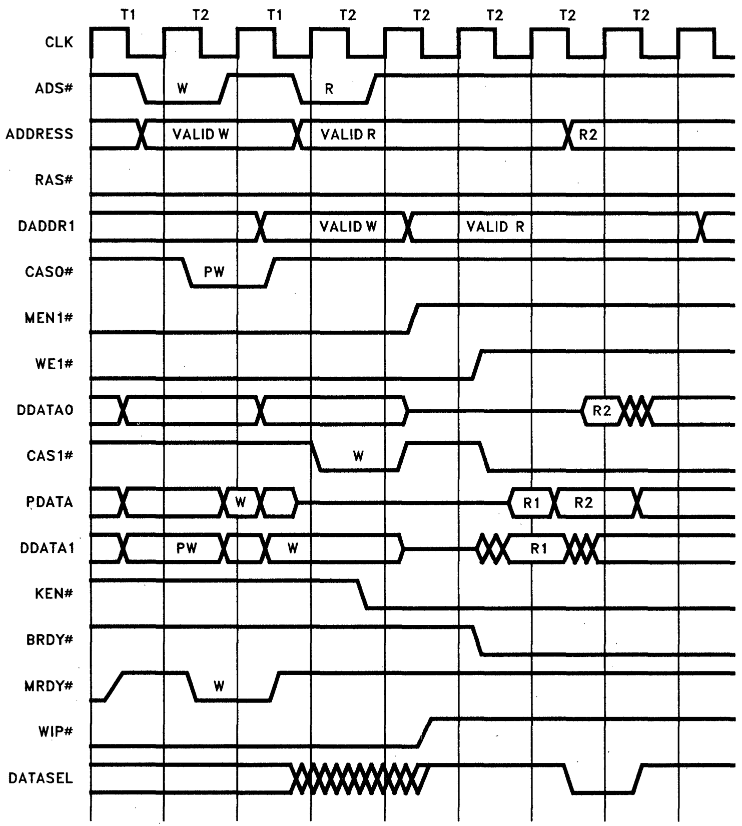
This fact is also important if the design is to be modified. For example while, interleaved DRAMs may not be required in systems with a permanent second level-cache, the write posting technique may still be used in the system. The benefits of this technique still apply since write cycles may still be overlapped as described.

5.8 Consecutive Bus Cycles

The DRAM control logic is optimized for write cycles, as warranted by the i486 processor's bus characteristics. Over 70 percent of all cycles are writes. By employing the posted write technique, system performance is increased.

The posted write technique poses some special problems, however. Page miss, refresh and consecutive write-read cycles require special consideration. We will begin by discussing the consecutive write-read case. Page miss and refresh cycles will be discussed in sections 5.9 and 5.10.

When a read cycle immediately follows a write, the read cycle must be delayed as illustrated in Figure 20. The read cycle is delayed to allow the write to complete. Only read cycles to DRAM, i.e. (cache misses) need be delayed. Cache hits and write cycles overlap easily because the cache is on the CPU side of the DRAM controller.



240799-17

Figure 20. Consecutive Write-Read Cycle

Write cycles cannot overlap DRAM read cycles, however, primarily because of data bus contention. The DRAMs used here have common data I/O pins. In this case read and write data paths cannot be active at the same time.

To prevent data bus contention, the first data access of the read is delayed. In Figure 20 the first read access is to the same bank as the write. In addition, the read cycle accesses the same DRAM row. Two functions are required to ensure that the write is completed. First, the write address must be held until CAS# is asserted. Second, the data mux outputs must not be enabled until the CPU tristates the bus.

The first function is accomplished by the MUXEN# signals. The MUXEN# state machine tracks the CAS# function for the appropriate bank. When the write for that bank is complete, MUXEN# is deactivated. In this way, the read address path is not enabled until the CLK after CAS# becomes active. Normally, the read address would be valid in the first T2 of the read cycle; however it must be delayed one clock to allow the write complete. Note that if one or more idle CLKs intervenes between these cycles, no delay occurs.

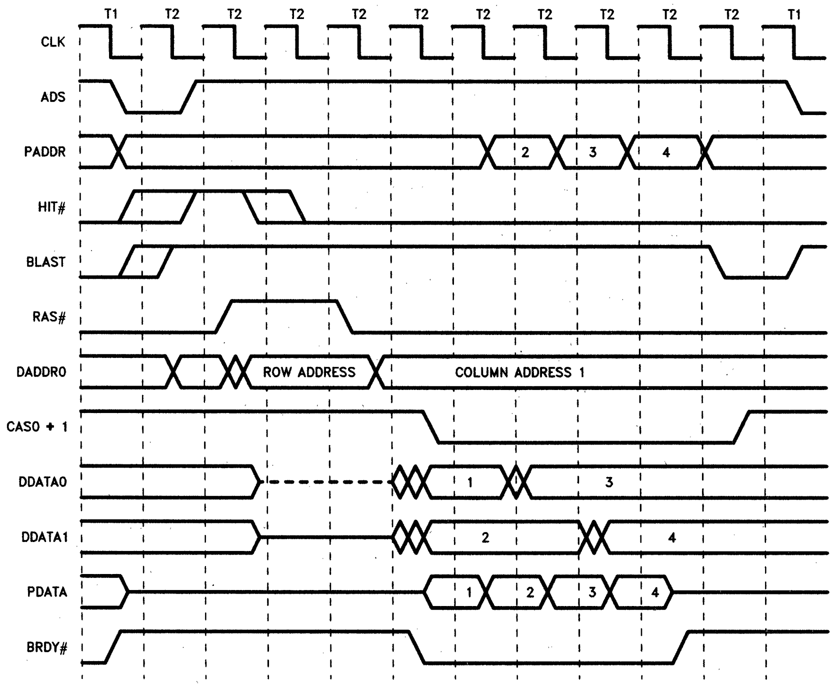
The second function is accomplished with the WIP# signal which is active until all write cycles are complete. A read cycle to either bank will be delayed if it immediately follows a write. The first access of the read is delayed by MBRDY#, which is not asserted until the WIP# signal is deasserted.

WIP# is deasserted once all pending writes are complete. In Figure 20 the read cycle is delayed 3 CLKs by this signal; in other words, three additional wait-states are added. If a read does occur immediately after a write, the number of wait-states added will decrease by the number of idle CLKs between cycles. For example, if ADS# for the read is asserted three clocks after MRDY# for the write, MBRDY# will not be delayed.

5.9 Page Miss Cycles

As described previously, page miss cycles occur when the CPU generates a cycle which changes the DRAM row address. The RAS# signal must be deasserted to change the ROW address in the DRAMS. Any time RAS# is deasserted, it must remain high for the precharge time (tRP). A delay is added to every page miss cycle to satisfy this requirement.

For read cycles this function simply requires extra wait states as illustrated in Figure 21.



240799-18

Figure 21. DRAM Page Miss-Read Cycle

The bus cycle starts with RAS# low or active. The row address generated by the CPU is different than in the previous cycle, and the row address comparator deasserts HIT#. This signal is valid in the first T2. HIT# is sampled at the RAS# PLD at the rising edge of the second T2. In response, RAS# is immediately deasserted and held inactive for two clocks. This time satisfies the RAS# precharge requirement.

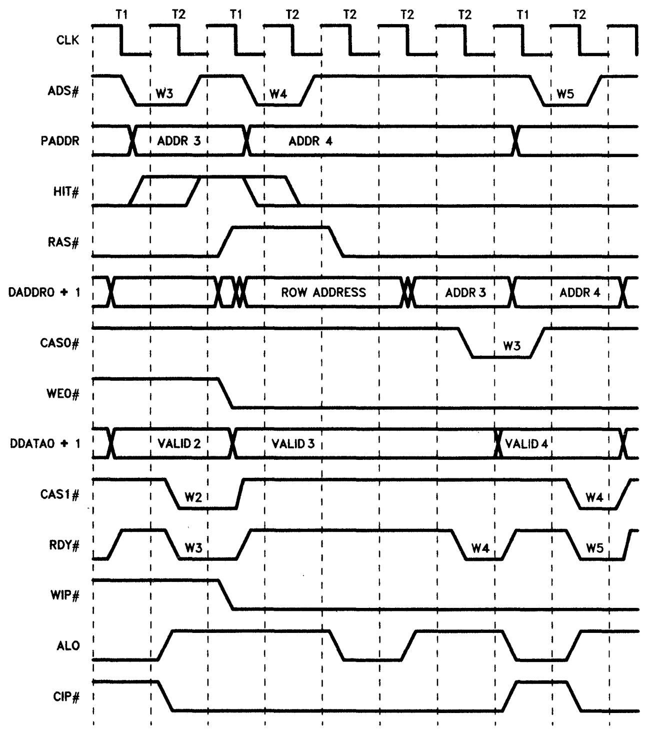
Four wait states are added to process the miss cycle. These clocks are added to every read cycle which accesses a new DRAM row. The delay is accomplished, again, with the MBRDY# signal. MBRDY# will not be asserted when RAS# is inactive. Once RAS# is sampled active, MBRDY# is asserted. From here, the cycle proceeds as described in section 5.7.

Write miss cycles are more complex than read miss cycles, due mainly to the write posting technique. The added complexity results in lower latency than in a non-posted memory system, however. Figure 22 illustrates how this improvement is achieved.

The write cycle in Figure 22 also begins with RAS# active. The HIT# signal is deasserted in the first T2 at the same time that MRDY# is asserted. MRDY# could be inhibited at this point to prevent write cycle termination. The wait-states added to meet RAS# precharge time would then be added to this cycle. Five wait states are required to meet the precharge time.

The average number of write cycle clocks can be reduced, however, if another method is used. MRDY# can be allowed to terminate the cycle. In this case, any necesary wait-states will be added to the next cycle.

This method improves the average in two ways. First, some write miss cycles will not require wait- states. This is the case when the next cycle occurs four or more clocks after a write miss. In addition, wait states will be reduced when the next cycle occurs in two or three clocks. Second, three wait-states are required to complete the next cycle when it follows immediately as illustrated in Figure 22.



240799–19

Figure 22. DRAM Page Miss-Write Cycle

The first cycle in this figure is a page miss. It is terminated at the CPU without wait-states. Because HIT# is not active in the first T2, RAS# is deasserted. At this point, additional clocks are added to perform the miss function. Part of the time required for RAS# precharge is overlapped with the next cycle. The two clock overlap reduces the number of wait-states required in the next cycle. Therefore, the average write cycle latency is reduced.

5.10 Refresh Cycles

The CAS# before RAS# refresh function is used in this example. This function uses internal counters in the DRAM devices to generate the refresh address. When the CAS# input is activated prior to RAS#, the internal counter is incremented. The output of the counter is then used as the address of the row to be refreshed.

Each refresh cycle refreshes one row of the DRAM array. The refresh cycles are distributed such that one occurs every 15.6 μs, with every row being refreshed in 8 ms. Refresh cycles are initiated by the RFRQ signal. This signal is activated every 15.6 μs by a counter.

RFACK is asserted in response to RFRQ. This signal indicates that the DRAM controller is ready to perform the refresh cycle. It also signals the counter circuit that RFRQ can be deasserted.

The function of RFRQ and RFACK is very similar to that of the CPU's HOLD and HLDA signals. RFRQ is sampled at the end of each cycle and during idle cycles. RFACK is activated in the clock after RFRQ is sampled, except immediately after write cycles.

Again, the posted write function must complete before the refresh cycle begins. If WIP# is active when RFRQ is sampled, RFACK will not be immediately asserted. RFACK will be asserted after WIP# is deactivated as illustrated in Figure 23.

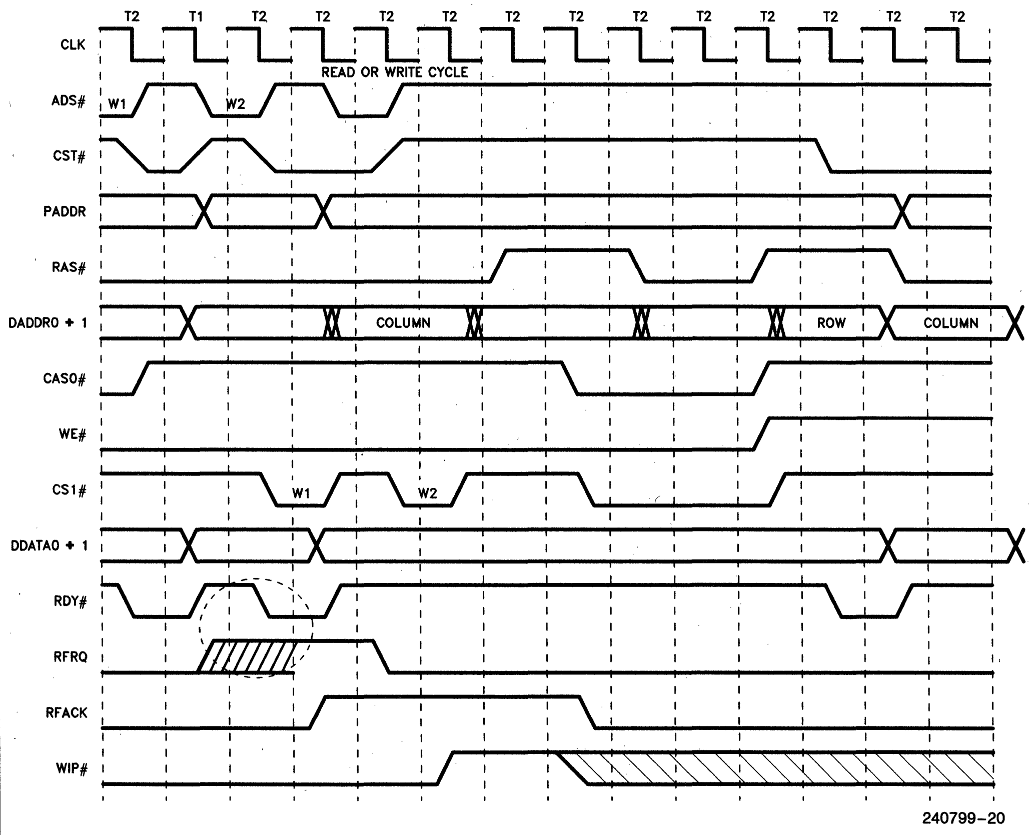


Figure 23. Refresh Timing Concurrent with Write

Another cycle can start between RFRQ and RFACK. The cycle start PLD tracks this case. GP# ~ will not be asserted for any cycle that starts during this interval. Once the refresh cycle is complete, this cycle can be started.

6.0 CONTROLLER IMPLEMENTATION

The functions described in the previous section are generated by the control logic. The controller, as outlined in Section 4.0, is made up of several PLDs. These devices enerate the control signals described in Section 5.0. The function of the logic is determined by the state machine definition. These state machines are distributed in the different PLDs of the controller.

In this section, we will explore the implementation of the control logic. The discussion will focus on the state machine definition. Certain conventions are followed throughout the discussion. These conventions are based on the state machine compiler used to generate the PLD equations. This compiler uses the exclamation point (!) to indicate the low or "0" condition of a signal. It uses the number symbols (#) to indicate that the signal is active low. For example, !ADS# indicates that the ADS signal is both low and active. The # symbol indicates that a signal is active when low. So symbol !ALD means that the ALD signal is not active. These symbols are used to indicate state transitions as shown in Figure 24. The state transition in Figure 24 depends on three signals: ADS#, ALD, and RAS#. The equation indicates that if both ADS# and ALD are active or if RAS# is not ctive at the next clock edge. the transition from S0 to S1 takes place. In the transition between S0 and S1, the Y# signal is activated. The definition of states indicates which outputs are changed in the transition. These conventions are used to describe the control state machines in the next section.

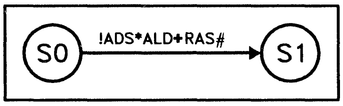


Figure 24. State Transition Example

6.1 Cycle Tracking Logic

The cycle tracking logic is contained in one PLD. The five state machines implemented in this PLD start and end DRAM cycles, control refresh timing and control the address registers. These state machines, along with the MRDY# state machine comprise the first level of control logic. All other control state machines depend on this first level to generate signals at the proper time.

The signals generated by this PLD are the following:

CIP# - Cycle in Progress
ALD - Address Latch Disable
CT - Cycle Track
RFACK - Refresh Acknowledge
PCHG- RAS Precharge Count

The primary cycle tracking state machine is shown in Figure 25. This state machine generates the CIP# and M# signals. CIP# indicates that the CPU has started a cycle. When it is active, the rest of the logic samples the CPU control and MEMCS# signals. If the current cycle is not to DRAM, it will be ignored and CIP# will be deactivated.

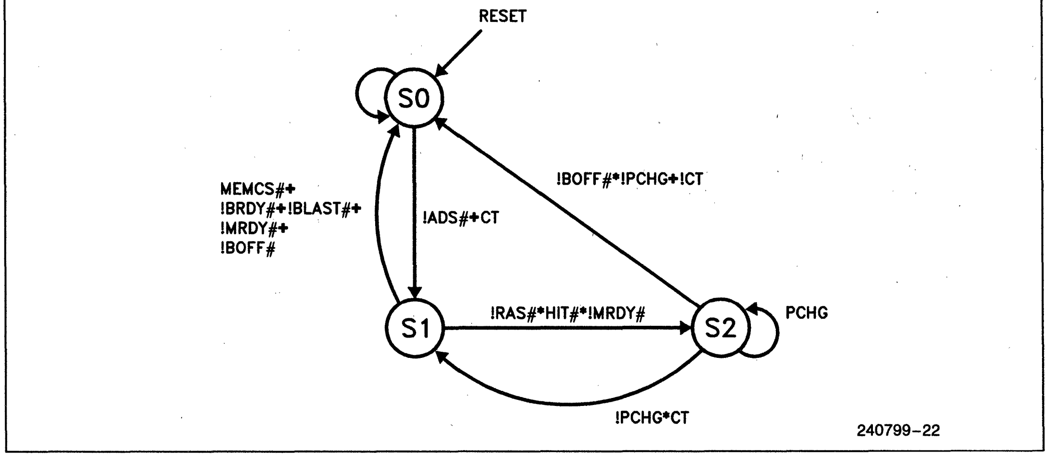


Figure 25. Cycle in Progress State Diagram

This function is defined by the S0 and S1 states in Figure 25. As shown, CIP# is activated when either ADS# or CT are sampled active. If the cycle is not to a DRAM address, the MEMCS# signal will not be active in the next clock. In this case, CIP# is deactivated to wait for the next ADS#. If the cycle is to DRAM, CIP# stays active until the end of the bus cycle. The bus cycle is terminated by one of three circumstances. All write cycles are terminated with the MRDY# signal. Read cycles are terminated by BRDY# and by BLAST#. The cycle can be aborted by BOFF#. Any of these three events causes CIP# to be deactivated (S1 to S0).

Two special cases are also handled by this state machine. When AHOLD is active in the same clock as ADS#, MEMCS# is not valid. In this case, the CIP# signal is not activated until AHOLD is deasserted. The state machine remains in S0 when AHOLD is active.

The second case is a write miss cycle. During a write miss, CIP# must be active for the cycle to complete. CIP# is active in this case after MRDY# is returned to the CPU. Cycles that start during the time CIP# is active must be tracked by the CT state machine. The M# signal indicates to the CT state machine that the cycles must be tracked.

The state in which M# and CIP# are both active is S2. This state is entered when MRDY# and RAS# are active and HIT# is inactive. By using MRDY# to qualify this transition, S2 is entered only during write cycles. Therefore, M# is only activated during write miss cycles. Note that any cycle will be recognized by the CT state machine when M# is active.

The CT state machine is shown in Figure 26. This state machine tracks cycles that start while the CIP# state machine is busy. It tracks CPU cycles that start during refresh cycles as well as to the two cases mentioned above.

This state machine tracks one cycle. Any cycle that starts while CIP# is busy is not terminated immediately. The MRDY# and MBRDY# signals are delayed until the previous cycle is finished. Therefore, anytime CT is active, there is only one cycle pending.

CT is deactivated when the pending cycle is recognized by the CIP# state machine. This event is indicated by CIP# active and M# inactive. When this event occurs, the CT state machine transitions to S0 deactivating CT.

The ALD signal is also active only during DRAM cycles. Therefore, its state machine is very similar to that of CIP#. As with CIP#, ALD is asserted when ADS# is sampled active. If the cycle is not to a DRAM address, ALD is deasserted. When a DRAM cycle is terminated, ALD is also deasserted. The S0- to-S1 transition is quite similar to that of CIP#.

The difference between the two state machines is revealed during write miss cycles. The S1-to-S2 transition is made if a write miss occurs. ALD must be held active during a write miss until RAS# is active. In this way the row address is held even if another cycle occurs. The combination of CIP# being active while PCHG is inactive indicates that RAS# will be active in this clock. ALD must be deactivated in this clock to allow the next address to be latched. ALD is re-activated if

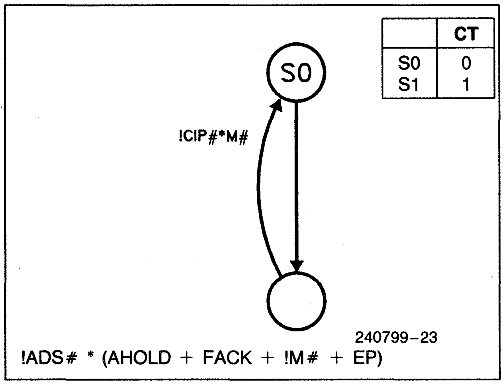


Figure 26. Cycle Tracking State Machine

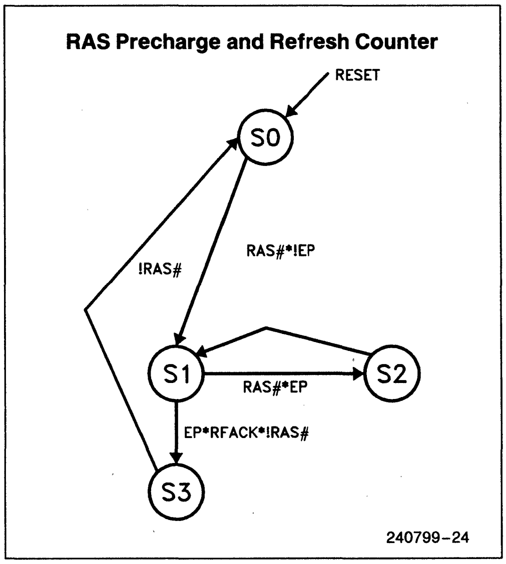


Figure 27. Precharge State Machine

another cycle has started during the write miss process. CIP# and MEMCS# are sampled during S0 for this purpose.

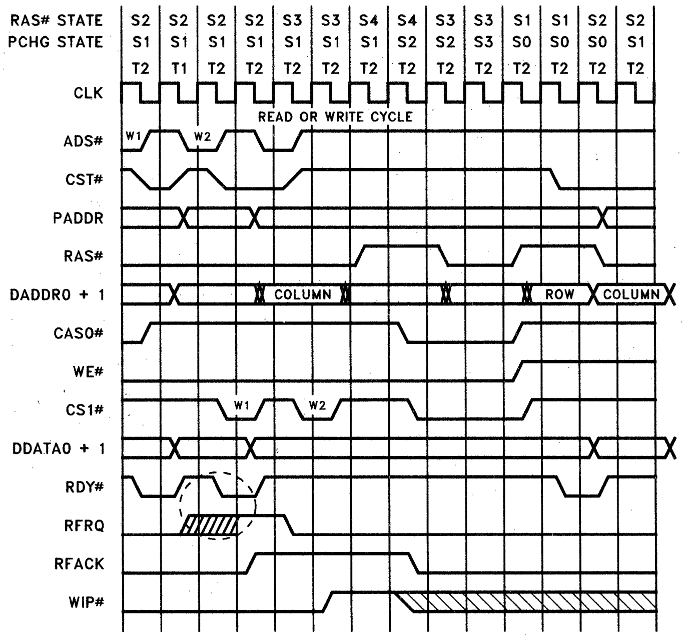
The PCHG state machine provides two functions. It determines the time RAS# is inactive during a miss or refresh cycle, and it determines the timing of refresh cycles. Figure 27 shows the state transitions of the PCHG state machine. Because the timing of this signal is not obvious, Figure 28 has been included. It shows a refresh cycle which occurs following a write cycle.

After RAS# is active the PCHG signal is activated. State S1 is maintained then until RAS# is deactivated. RAS# is only deactivated during a miss or refresh cycle or, of course, if RESET is asserted. During a miss cycle the transition to S0 is made deactivating PCHG. RAS# is then activated, resulting in two CPU clocks of RAS# precharge time.

States S2 and S3 define the timing of refresh cycles. The transition to this sequence is made when RAS# is sampled inactive while EP active. EP indicates that the RAS# state machine has entered the refresh sequence.

RFACK# initiates the refresh sequence. It indicates that the control logic is ready to accept a refresh request. The RFRQ signal is sampled at the end of a DRAM cycle or during idle clocks. Note that RFRQ cannot be recognized during a write miss.

RFACK# is deactivated after RAS# is deactivated at the beginning of the refresh sequence (See Figure 27 and Figure 28).



240799-25

Figure 28. Refresh State-Timing Example

6.2 RAS# Logic

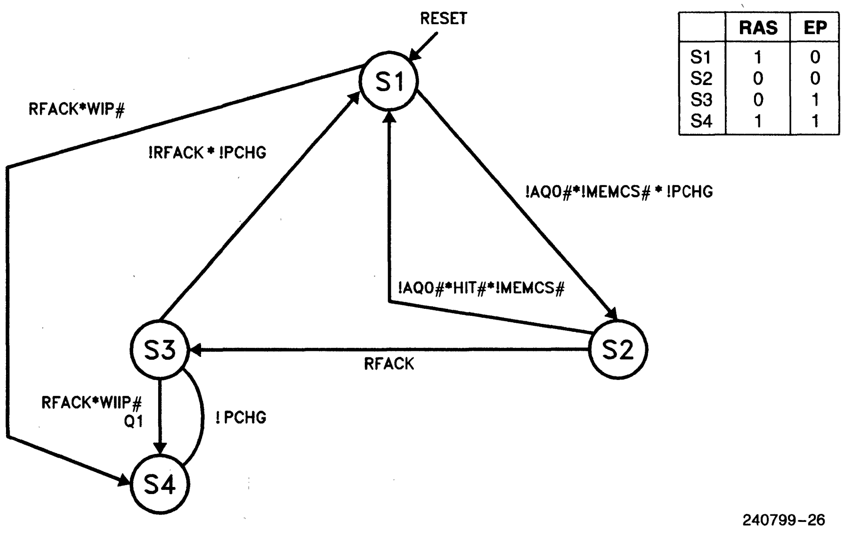
The RAS# logic for both memory banks occupies one PLD. Four RAS# signals are generated: RAS0#–RAS3#. These signals are generated to divide loading. Their timing is identical. The state machine for RAS is relatively simple and is shown in Figure 29.

States S0 and S1 are used to implement RAS# function for normal cycles. After RESET, the state machine waits for the first bus cycle. The first bus cycle is signaled by the CIP# signal. When CIP#, MEMCS# and PCHG are sampled active, RAS# is asserted. RAS# stays active until a miss or refresh cycle occurs.

A miss cycle is indicated when the HIT# signal is driven inactive. It is qualified by CIP# and MEMCS# being active. In this way, RAS# is only deactivated during DRAM cycles.

Once RAS# is deasserted during a miss cycle, it stays high until PCHG is sampled active. This function implements the RAS# precharge time. CIP# and MEMCS# will still be active during read miss cycles. Therefore, RAS# will be asserted in the next clock. For write miss cycles the WIP# signal must be used to restart RAS#. With a write miss, a non-DRAM cycle can occur before RAS# is asserted. WIP# is the only valid indication that a DRAM cycle has occurred in this case. WIP# is combined with MEMCS# to create the CSWIP# term which indicates a valid RAS# cycle.

When a refresh cycle occurs, the RAS# state machine transitions to S2. S2 and S3 are devoted to the refresh function. When RFACK is sampled active, the transition occurs. The refresh sequence shown in Figure 28 illustrates the function of these two states. Note that after a refresh cycle, RAS# is left inactive. The transition from S0 to S4 allows for refresh cycles that start when RAS# is inactive.



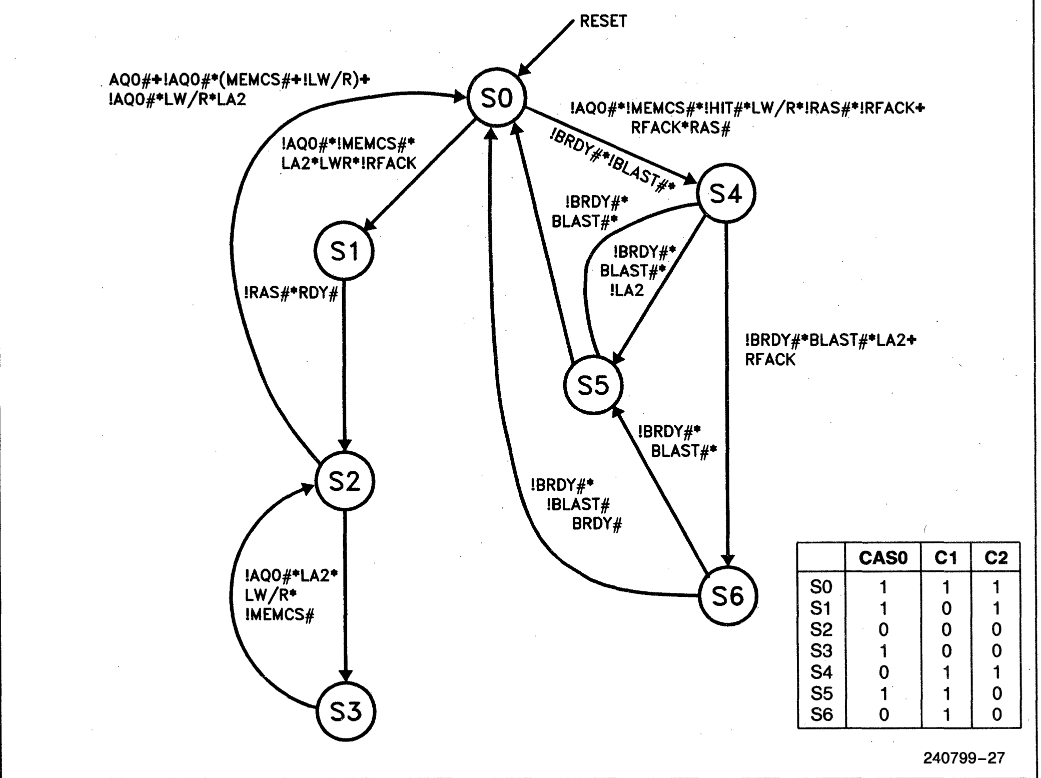
	RAS	EP
S1	1	0
S2	0	0
S3	0	1
S4	1	1

Figure 29. RAS State Machine

6.3 CAS# Logic

Two separate PLDs implement the CAS# function. These PLDs generate the CAS# signals for bank 0 and bank 1, respectively. The state machines which generate these signals are separate and independent. Each generates two CAS# signals. CAS00# and CAS01# for bank 0, and CAS10# and CAS11# for bank1. These signals drive separate DRAM modules due to drive requirements.

Figure 30 shows the state diagram for the bank 0 CAS# function. The states on the left side of the diagram implement the write function. The states on the right implement the read function. As with RAS#, the state machine waits until CIP# indicates that a cycle has started. When CIP# is active, the state of the latched version of W/R# determines which sequence is started.



	CAS0	C1	C2
S0	1	1	1
S1	1	0	1
S2	0	0	0
S3	1	0	0
S4	0	1	1
S5	1	1	0
S6	0	1	0

Figure 30. CAS State Machine

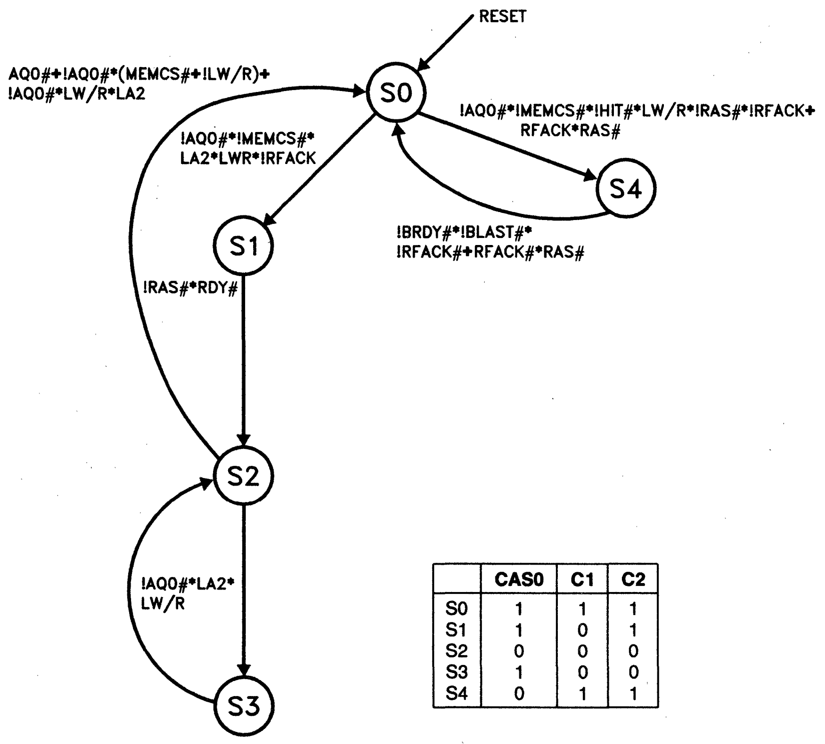
If the cycle is a read, S4 is entered. If the cycle is a write, LA2 is sampled to determine if the cycle is to bank 0. If LA2 is low, S1 is entered. Note that this function is the same for the bank 1 state machine. The only difference is the state of LA2, which starts the write sequence.

During a write cycle, CAS# is held inactive until the clock after RDY# is asserted. The state machine also waits in S1 during a write miss cycle. CAS# is asserted during S2. In this state, several events can occur. First, the CPU may not start another bus cycle. Second, it may start a bus cycle other than a DRAM cycle. Third, it may initiate a read cycle, and fourth, it may begin a write cycle to bank 1. If any of these events occur, S1 is entered. If another write cycle starts to the same bank, however, S3 is entered.

The case of sequential writes to the same bank involves S2 and S3 only. An unlimited number of write cycles can occur in the same bank. If the DRAM row is same, they will occur without wait-states. If a write miss occurs, RAS# will be deasserted, and the transition from S3 to S1 takes place.

During read cycles, the CAS# signals for bank 0 and bank 1 are activated at the same time. Therefore, the state machines enter S4 at the same clock. At this point, however, the state of LA2 determines which state machine enters S5. In S5, CAS# is deasserted to prepare that bank for the next access. If S6 is entered, the data from that bank has not yet been accessed. CAS# must be held active, in this case, until the data is sampled by the CPU. From S6, the next transition will be to S5 to continue the cycle, or S0 to terminate the cycle. If this bank was accessed first, the cycle will terminate from this state.

The read sequence is much simpler if static column mode DRAMs are used. The state sequence for static column mode is shown in Figure 31. The write sequence in this diagram is exactly the same as for the page mode CAS# control logic. The read function, however, requires only two states. From S0, the transition is made to S5 any time that a DRAM read cycle starts. Note that LA2 is not used to qualify this transition. Therefore, the CAS# signals for bank 0 and bank 1 are active at the same time.



	CAS0	C1	C2
S0	1	1	1
S1	1	0	1
S2	0	0	0
S3	1	0	0
S4	0	1	1

240799–28

Figure 31. Static Column CAS State Machine

6.4 Write Control Logic

The posted write implementation requires logic support for a few key functions. These functions are required mainly to support posting with interleaved memory. Three types of signals are generated to implement these functions:

Multiplexer Select - These signals control the address multiplexers when RAS# is active. During write cycles, they must be active to select the write address path. These signals stay active during read cycles which are immediately preceded by a write. They are deactivated, when the write cycle is complete. Once they are deactivated the read cycle may proceed as the read path is selected.

Write Enable - These signals are combined with the byte enable CPU outputs (BE0#-BE3#) to create the WBE# signals. The WBE00#-WBE03# signals control which byte is written in bank 0 during a write cycle. The WBE10#-WBE13# signals perform the same function for bank 1.

Write In Progress - This signal is active when a write cycle has been started by either DRAM bank. It is active when either C01# or C11# is active. C01# and C11# are state outputs from the CAS# state machine which indicates that a write cycle is being performed. C01# is generated for bank 0 and C11# for bank 1. WIP# is only required for interleaved memory systems. The C01# (or C11#) output would be sufficient for a non-interleaved (single bank) system.

The state machines which generate these signals are shown in figure 32. The state diagram for the MEN0# signal is shown. This signal enables the address multiplexer for bank 0. MEN0# is activated whenever a write cycle occurs to an address with A2 low (0). The MEN1# function is the same except that it is activated when A2 is high (1). The AQ0#, MEMCS# and LW/R# signals are used to indicate a valid write cycle.

The MEN# signals are deactivated when the write cycle is complete. The cycle is complete when CAS# for that bank is sampled active. For bank 0, C01# is used to indicate that a write is in progress. MEN0# is held active when C01# is active. When CAS00# is sampled active, CIP# is checked to determine if another valid write to the same bank has occured. If so, MEN0# stays active until CAS00# is sampled active. This function keeps the write address path open during consecutive writes to the same bank.

The WE# state machine is very similar to that of the MEN# state machine. When a write cycle starts, WE0# is activated in the same manner as MEN0#. The write enable signals, however, must stay active one clock longer than the MEN# signals. Therefore, the WE# signal is not deactivated until C01# is sampled inactive.

WIP# is generated in part by combinatorial logic so that it can be active in the same clock as the C01# and C11# signals. WIP# must be active in this clock to ensure that a write miss is completed before a refresh cycle takes place. WIP# must also be held active one clock after C01# and C02# are sampled inactive. This timing ensures the proper sequence for subsequent read cycles. The logic equation and state machine for WIP# are shown in Figure 32.

6.5 Burst Address Logic

The burst address logic generates the B1MA0 and B0MA0 signals. These signals are connected directly to the low order address inputs of the DRAMs. Because of the direct connection, these signals must perform several different functions. They must multiplex the low order row and column addresses, multiplex the write and read addresses and generate the burst address during read cycles.

These functions are performed separately for each bank by two PLDs. Each PLD generates two identical signals to reduce the drive requirements. These signals are connected directly to two bytes of the DRAM array. The signals are generated partly by combinatorial logic and partly by the state machine.

The logic equations and state diagram for this function are shown in Figure 33. The state machine generates the burst address for read cycles. The logic equations handle the multiplexing functions.

The burst address is generated after a burst read cycle has started. Note that the i486 CPU cache need not be enabled for burst cycles to occur. Cycles such as 64-bit floating-point operand reads will burst if BRDY is returned to the processor. S0 and S3 track the state of the A3 CPU address output. When a burst read cycle starts, S1 or S2 is entered. The B0MA0 address output will then change its state when MBRDY# and DATASEL are both low. This function is the burst address for bank 0. The B1MA0 address output changes its state when MBRDY# is low and DATASEL is high. This function is the burst address for bank1. The only difference in the two PLDs is the value of DATASEL used to determine the time of which the burst address changes its state.

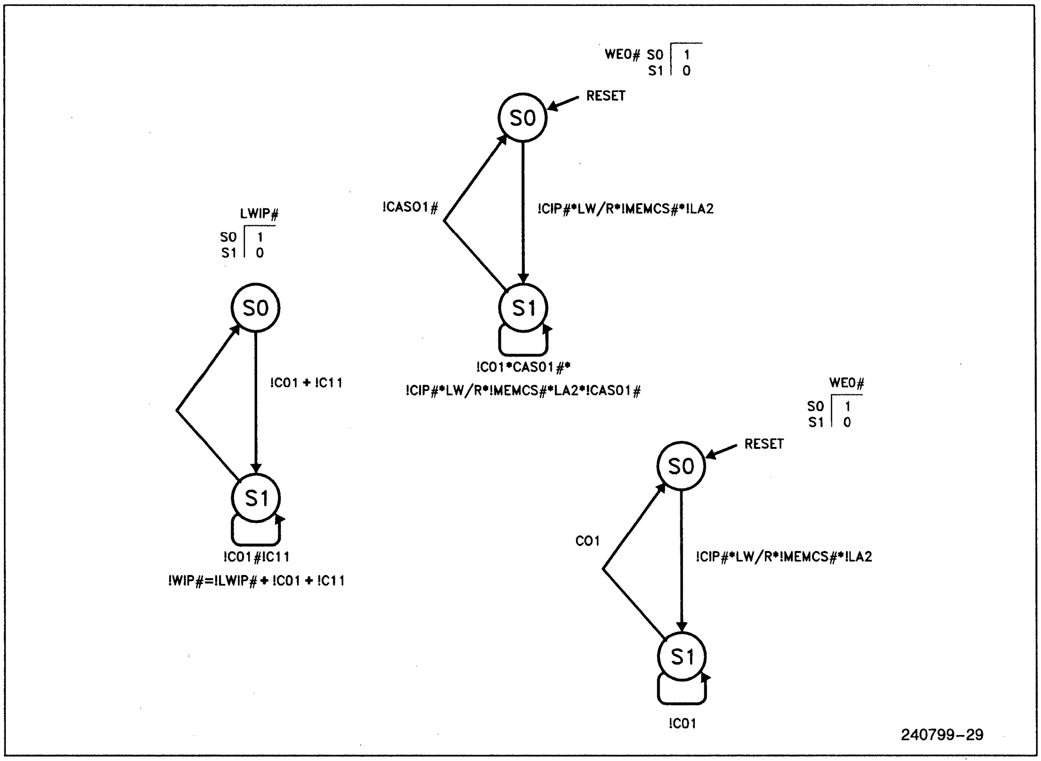
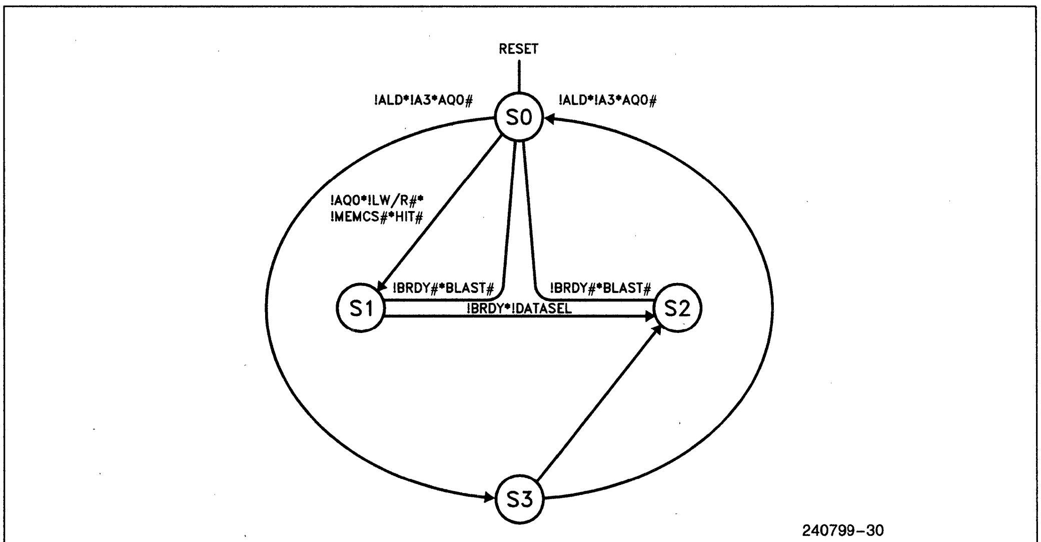


Figure 32. State Machines for MEN0#, WIP#, and WE0#



!B00MA0 = !WE0# * !LA313 + WEO# * RAS# * !LA313 * WE0# * !RAS# * IB0A

RAS Precharge and Refresh Counter

Figure 33. Burst Address Generation

The S0 and S3 states are required only to ensure that the burst address outputs are valid during the T2 of any read cycle. Figure 17 shows the timing of a burst read hit cycle. In the first access of this cycle, the burst address must be valid in the first T2 to satisfy the address access time requirements of the DRAM. The value of A3 is sampled with ALD to statisfy this requirement. In this way, the burst address state machine always starts from the correct value of A3. If another wait state is added to this access, this function is not required.

The logic equations which provide the multiplexor function are very simple. The first term of the equations shown in Figure 33 enables the write path. The write enable signals are used to enable this path. When WE0 is active, for example, the value of the multiplexor output is passed through to the DRAM. The second term allows the row address A13 to be passed to the DRAM during a read page miss. This term is also qualified by the write enable signals. In this way, the write address is not disabled early during a read miss. The third term enables the burst address output from the state machine onto the address pins.

7.0 SUMMARY

We have discussed an example memory subsystem for the i486TM CPU. The material has been presented as a design guide for systems under development or as an optimization for existing systems. We have discussed several key functions which will be summarized in this section. We will also discuss some important timing restrictions. The key functions discussed include an external or second level cache, posted write cycles, and interleaved DRAM banks.

The interleaving technique is used to support the burst bus feature of the i486 CPU. The use of this technique allows the DRAM to supply a DWORD every clock during burst cycles. Interleaving proves to be very useful in i486 CPU memory designs. Without its use DRAM timings such as tPC (Page Mode Cycle time) and tCP (CAS Precharge time) would prevent zero wait state access at 33 MHz.

Data registers are also used to improve average write cycle latency. These registers hold write data during posted write cycles. Write posting can improve average write latency to under 3 clocks for many applications. This improvement is important in i486 CPU based systems because 65% to 70% of all bus cycles are writes. Without using a latency improvement technique such as write posting average write latency will be above 5 clocks.

The write posting technique also improves memory performance in other ways. Write cycles, particularly DRAM page misses, can be overlapped with read hit cycles in the second level cache. This fact greatly reduces the delay caused by read cycles which immediatly follow write cycles.

Analysis of this memory subsystem design has shown that use of these features has resulted in a low latency response to the CPU. Over several important applications the following characteristics have been recorded. The average clock cycles required to complete the first read is 3.5 clocks. Subsequent cycles of a burst are always processed in one clock. Write cycles average 2.5 clocks. These average counts result from the following DRAM access rates. Read accesses from the cache always occur in zero wait states.

Table 3. Dram Function Latencies

DRAM Function	First Access Burst	Subsequent Burst Accesses	Write Cycles
Page Hit	3	1	2
Page Miss	7	1	5*

NOTE:
*Write miss latencies occur only during cycles subsequent to a write miss cycle.

7.1 Timing Restrictions

A few DRAM timing restrictions must be mentioned. These timings become critical at 33 MHz. These timings are critical due primarily to the latency of the first cycle of a read page hit. Since three clocks are used the following timing restrictions exist.

tRAC = Data access time from RAS# active

tCAA = Data access time from column address valid

tCAC = Data access time from CAS# active

tRP = RAS# precharge time

At 33 MHz

tRAC = 71.5 ns

tCAA = 37.5 ns

tCAC = 34 ns

tRP = 60.6 ns

At 25 MHz

tRAC = 101.5 ns

tCAA = 51 ns

tCAC = 61.5 ns

tRP = 80 ns

APPENDIX A
PLD CODES AND SCHEMATICS

A.1 PLD DEVICES

Many design examples in this manual use PLDs (Programmable Logic Devices) which can be programmed by the user to implement random logic. A PLD device can be used as a state machine or a signal decoder, for example. The advantages of PLDs include the following:

1. PLD pinout is determined by the designer, which can simplify board layout by moving signals as required.
2. PLDs are inexpensive as compared to dedicated bus controllers.

Intel EPLDs (Erasable Programmable Logic Devices) have the following additional advantages:

1. Programmability/erasability allows EPLD functions to be changed easily, simplifying prototype development.
2. Since EPLDs are implemented in CMOS technology, they can consume an order of magnitude less power than bipolar PLDs. Power-conscious applications can benefit greatly from using EPLDs.
3. Since the EPROM cell size is an order of magnitude smaller than an equivalent bipolar fuse, EPLDs can implement more functions in the same package. This higher integration can result in a lower overall component count for a design. The added flexibility can also mean that an extremely low number of "raw" (unprogrammed) devices need to be stocked versus bipolar PLDs.
4. Once an EPLD design has been tested, plastic OTP (One-Time Programmable) versions of the device can be used in a production environment.

PLDs have the following tradeoffs:

1. Most PLDs do not have buried (not connected to outputs) registers. For some state machine applications, this means using an otherwise available output pin to store the current state.
2. The drive capability of CMOS EPLDs may be insufficient for some applications. While the trend is towards use of CMOS throughout a system, in cases where high current levels are required, some additional buffering may be required with EPLDs.

A PLD consists logically of a programmable AND array whole output terms feed a fixed OR array. Any sum-of-products equations, within the limits of the number of PLD inputs, outputs, and equation terms, can be realized by specifying the correct AND array connections. Figure B-1 shows an example of two PLD equations and the corresponding logic array. Note that every horizontal line in the AND array represents a multi-input AND gate; every vertical line represents a possible input to the AND gate. An X at the intersection of a horizontal line and a vertical line represents a connection from the input to the AND gate.

The sum-of-products is then routed to a configurable macrocell. The macrocell in Figure B-2 can be configured as a combinational output or registered output. The output can be active high or active low. A separate AND term controls the output buffer.

Designing with PLDs consists of determining where Xs must be placed in the AND array and how to configure the macrocell. This task is simplified by logic compilers, such as iPLS II (Intel's Programmable Logic Software II) or ABEL. Logic compilers accept input in the form of sum-of-product equations and translate the input into a JEDEC programming file that can be used by programming hardware/software.

Intel PLDs are described in the *Programmable Logic Handbook*. Three Intel PLDs have been used in this manual to implement state machine and decode functions. These PLDs include:

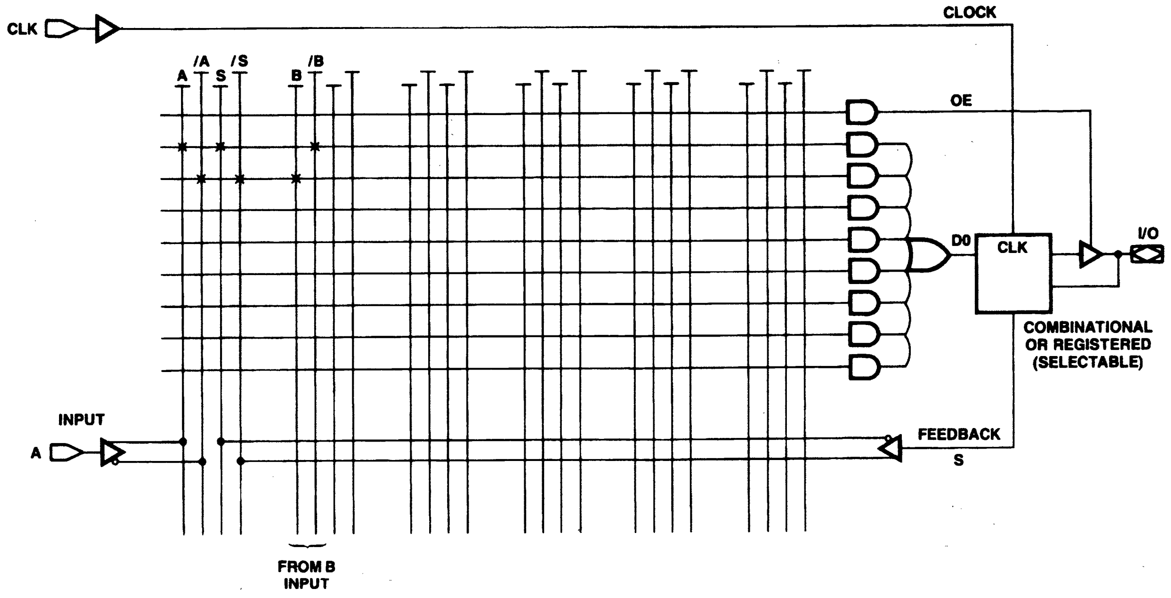
- 85C220—fast 20-pin superset of 16 x 8 type bipolar and CMOS PLDs.
- 85C224—fast 24-pin superset of 20 x 8 type bipolar and CMOS PLDs.
- 85C508—fast address decode PLD with integral transparent latches.

The 85C220 and 85C224 PLDs are both available at clock speeds to support fast state-machines in i486 systems. The 85C508 provides a fast Enable-to-Output time with a minimal system setup time.

BOOLEAN EQUATION:

D = A * S * /B
+ /A * /S * B

EPLD IMPLEMENTATION:



240799-31

Figure A-1. PLD Equation and Device Implementation

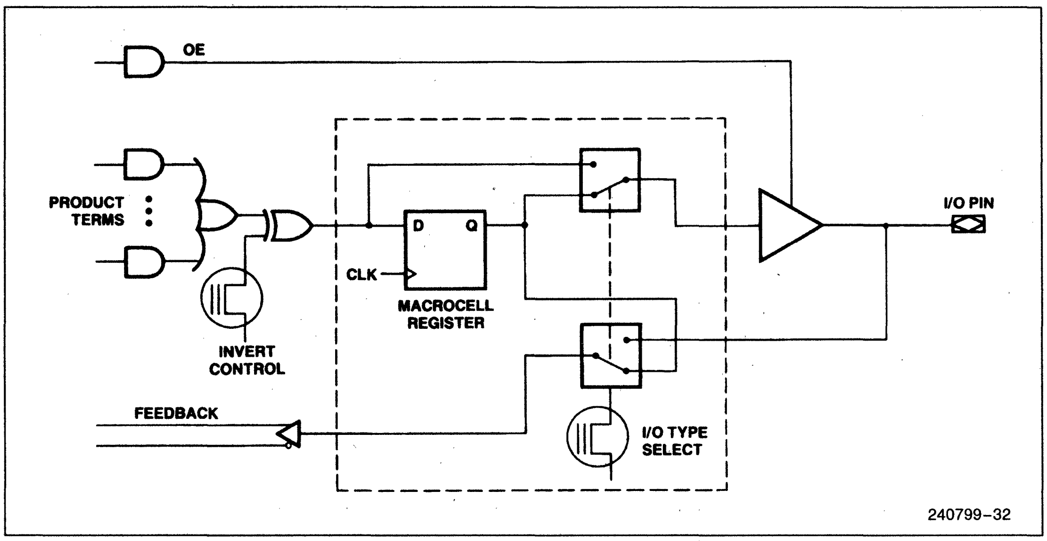


Figure A-2. 85C220/85C224 EPLD Macrocell Architecture

```
module  SC_MODE_DRAM_CTRL_4  flag '-r4'

title    'STATIC COLUMN MODE DRAM CONTROLLER - PLD 4, INTEL CORPORATION'
" This pld generates MRDY and MBRDY
" Implemented with Intel 85C224 EPLD.

  SCk  device          'E224';

  x          =    .X.;                " ABEL 'don't care' symbol
  c          =    .C.;                " ABEL 'clocking input' symbol

" Inputs

  CLK pin 1;  "P4 input CLK"
  M~          pin  2; "Miss Indicator
  CIP~        pin  3; "Cycle OK
  MEMCS~      pin  4; "Latched A2.
  HIT~        pin  5; "DRAM Page Hit Signal
  RFACK       pin  6; "Refresh acknowledge"
  ADS~        pin  7; "CPU ADS~
  W_R         pin  8; "CPU W/R
  RESET       pin  9; "System Reset
  dum1        pin 10; "Write in progress
  BOFF~       pin 11; "CPU Backoff input
  WIP~        pin 14; "CPU Burst Last output
  CAS~        pin 15; "Row Address strobe
  BLAST~      pin 22;
  RAS~        pin 23; "Any CAS# signal

" Output

  dum0        pin 16;
  MT          pin 17; " BRDY state miss tracking
  MRDY~       pin 18; " Memory RDY (modified with other RDYs)
  DALE~       pin 19; " Decode Latch enable
  LWR         pin 20; " Internally latched W/R# for rdy
  BRDY~       pin 21; " Processor BRDY~

 state_diagram [MRDY~]

   state [1]:   if (!RFACK & !ADS~ & W_R & !RAS~ & M~) # (!CIP~ & LWR &
                !MEMCS~ & !RFACK & M~) then [0] else [1];

   state [0]:   goto [1];

 state_diagram [BRDY ~, MT]

   state [1, 1]: if !CIP~ & !HIT~ & !MEMCS~ & !LWR & !RFACK & WIP~ & !RAS~
                    then [0, 1] else if !CIP~ & !MEMCS~ & HIT~ & !LWR #
                    !CIP~ & !MEMCS~ & RAS~ & !LWR then [1, 0];
```

240799–33

```
  state [1, 0]: if RESET then [1, 1] else
                   If WIP~ & !RFACK & !CAS~ then [0, 1];

  state [0, 1]: if RESET # !BOFF~ # !BLAST ~ then [1, 1] else [0, 1];

state_diagram [DALE~]

  state [0]:    if RESET then [0] else
                if !ADS~ then [1] else [0];

  state [1]:    if RESET # !BOFF~ then [0] else
                if !CIP~ then [0] else [1];

state_diagram [LWR]

  state [0]:    if RESET then [0] else
                if !ADS~ & W_R then [1] else [0];

  state [1]:    if RESET # !BOFF~ then [0] else
                if !ADS~ & !W_R then [0] else [1];

test_vectors

([CLK,M~,CIP~,MEMCS~,HIT~,RFACK,ADS~,W_R,RESET,WIP~,BOFF~,BLAST~]
 ->    [RAS~,MRDY~,DALE~,LWR,BRDY~])

"C M A M H R A W R W B B R      M D L B
"L ~ Q E I F D _ E I O L A     R A W R
"K   0 M T A S R S P F A S      D L R D
"      ~ C ~ F ~     E    F S ~    Y E    Y
"        S   K       T   ~ T          ~ ~    ~
"      ~                 ~
"
"
"


  [c, x, x, x, x, x, 1, x, 1, x, x, x, x]  -> [x, x, x, x];
  [c, x, 1, 1, x, x, 1, x, 1, x, x, x, x]  -> [1, 0, 0, 1];
  [c, 1, 1, 1, x, 0, 1, x, 0, 0, 1, 1, 1]  -> [1, 0, 0, 1];
  [c, 1, 1, 1, x, 0, 1, x, 0, 0, 1, 1, 1]  -> [1, 0, 0, 1];
  [c, 1, 1, 1, x, 0, 0, 1, 0, 0, 1, 1, 1]  -> [1, 1, 1, 1];
  [c, 1, 0, 0, 0, 0, 1, x, 0, 0, 1, 1, 1]  -> [0, 0, x, 1];
  [c, 1, 0, 0, 0, 0, 1, x, 0, 1, 1, 1, 0]  -> [1, 0, 1, 1];
  [c, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0]  -> [0, 1, 1, 1];
  [c, 1, 0, 0, 0, 0, 1, x, 0, 1, 1, 1, 0]  -> [1, 0, x, 1];
  [c, 1, 1, x, 0, 0, 0, 1, 0, 1, 1, 1, 0]  -> [0, 1, 1, 1];
  [c, 1, 0, 0, 0, 0, 1, x, 0, 1, 1, 1, 0]  -> [1, 0, x, 1];
  [c, 1, 1, x, 0, 0, 1, x, 0, 1, 1, 1, 0]  -> [1, 0, x, 1];
  [c, 1, 1, 1, x, 0, 1, x, 0, 1, 1, 1, 0]  -> [1, 0, x, 1];

end SC_MODE_DRAM_CTRL_4;
```

240799–34

```
module  SC_MODE_DRAM_CTRL_3  flag '-r4'

title    'STATIC COLUMN MODE DRAM CONTROLLER - PLD 3, INTEL CORPORATION'
" This PLD generates RAS
" Implemented with the Intel 85C220 EPLD.

  SC3     device    'E0320';

  x         =    .X.;              " ABEL 'don't care' symbol
  c         =    .C.;              " ABEL 'clocking input' symbol

" Inputs

  CLK pin  1;  "P4 input CLK"
  M~           pin  2; "Refresh Acknowledge
  CIP~         pin  3; "Cycle OK
  MEMCS~       pin  4; "Latched A2.
  HIT~         pin  5; "DRAM Page Hit Signal
  RFACK        pin  6; "Backoff input to P4"
  PCHG         pin  7; "RAS precharge count
  WIP~         pin  8; "Write in Progress
  RESET        pin  9; "System Reset
  Q1           pin 12; "RAS refresh count
" Output

  RAS2~        pin 13; "
  RAS1~        pin 14; " RAS byte 0,2
  EP           pin 15; " state variable
  EP1 pin 16; " state variable
  RAS0~        pin 17; " RAS byte 1,3
  RAS3~        pin 18; "
  CSWIP~       pin 19; "

state_diagram [RAS0~,RAS1~,EP]

  state [1, 1, 0]:    if RESET then [1, 1, 0] else
                      if !CIP~ & !CSWIP~ & !PCHG then [0, 0, 0] else
                      if RFACK & WIP~ then [1, 1, 1] else
                      [1, 1, 0];

  state [0, 0, 0]:    if RESET then [1, 1, 0] else
                      if RFACK then [0, 0, 1] else
                      if !CIP~ & HIT~ & !MEMCS~ then [1, 1, 0]
                      else [0, 0, 0];

  state [0, 0, 1]:    if RESET then [1, 1, 0] else
                      if !RFACK & !PCHG then [1, 1, 0] else
                      if RFACK & !WIP~ # !RFACK & PCHG then
                      [0, 0, 1] else if RFACK & WIP~ & !Q1 then [1, 1, 1];

  state [1, 1, 1]:    if RESET then [1, 1, 0] else
                      if !PCHG then [0, 0, 1] else [1, 1, 1];
```

240799-35

```
    state [0, 1, 0]:    goto [1, 1, 0];
    state [0, 1, 1]:    goto [1, 1, 0];
    state [1, 0, 0]:    goto [1, 1, 0];
    state [1, 0, 1]:    goto [1, 1, 0];

  state_diagram [RAS2~,RAS3~,EP1]

    state [1, 1, 0]:    if RESET then [1, 1, 0] else
                        if !CIP~ & !CSWIP~ & !PCHG then [0, 0, 0] else
                        if RFACK & WIP~ then [1, 1, 1] else
                        [1, 1, 0];

    state [0, 0, 0]:    if RESET then [1, 1, 0] else
                        if RFACK then [0, 0, 1] else
                        if !CIP~ & HIT~ & !MEMCS~ then [1, 1, 0]
                        else [0, 0, 0];

    state [0, 0, 1]:    if RESET then [1, 1, 0] else
                        if !RFACK & !PCHG then [1, 1, 0] else
                        if RFACK & !WIP~ # !RFACK & PCHG then
                        [0, 0, 1] else if RFACK & WIP~ & !Q1 then [1, 1, 1];

    state [1, 1, 1]:    if RESET then [1, 1, 0] else
                        if !PCHG then [0, 0, 1] else [1, 1, 1];

    state [0, 1, 0]:    goto [1, 1, 0];
    state [0, 1, 1]:    goto [1, 1, 0];
    state [1, 0, 0]:    goto [1, 1, 0];
    state [1, 0, 1]:    goto [1, 1, 0];

  equations

    !CSWIP~ = (!MEMCS~ # !WIP~)& !RESET;

  test_vectors

    ([CLK,M~,CIP~,MEMCS~,HIT~,RFACK,PCHG,WIP~,Q1,RESET] ->
    [RAS0~,RAS1~,EP,RAS2~,RAS3~,EP1])

  "  C M A M H R P W Q R   R R E R R E
  "  L ~ Q E I F C I 1 E   A A P A A P
  "  K   0 M T A H P   S   S S   S S 1
  "    ~ C ~ C G ~     E   0 1   2 3
  "      S   K         T
  "      ~
  "
  "
  "

    [c, x, x, x, x, x, 1, x, x, 1] -> [x, x, x, x, x, x];
    [c, x, x, x, x, x, 1, x, x, 1] -> [1, 1, 0, 1, 1, 0];
    [c, x, x, x, x, x, 1, x, x, 1] -> [1, 1, 0, 1, 1, 0];
    [c, x, x, x, x, x, 1, x, x, 1] -> [1, 1, 0, 1, 1, 0];
```

240799-36

```
    [c, x, x, x, x, x, 1, x, x, 1] -> [1, 1., 0, 1, 1, 0];
    [c, x, x, x, x, x, 1, x, x, 1] -> [1, 1, 0, 1, 1, 0];
    [c, x, x, x, x, x, 1, x, x, 1] -> [1, 1, 0, 1, 1, 0];
    [c, 1, 1, x, x, 0, 0, 0, 0, 0] -> [1, 1, 0, 1, 1, 0];
    [c, 1, 1, x, x, 0, 0, 0, 0, 0] -> [1, 1, 0, 1, 1, 0];
    [c, 1, 0, 0, 0, 0, 0, 0, 0, 0] -> [0, 0, 0, 0, 0, 0];
    [c, 1, 0, 0, 0, 0, 0, 0, 0, 0] -> [0, 0, 0, 0, 0, 0];
    [c, 1, 1, x, x, 0, 0, 0, 0, 0] -> [0, 0, 0, 0, 0, 0];
    [c, 1, 0, 0, 0, 0, 0, 0, 0, 0] -> [0, 0, 0, 0, 0, 0];
    [c, 1, 1, x, x, 0, 0, 0, 0, 0] -> [0, 0, 0, 0, 0, 0];
    [c, 1, 0, 0, 0, 0, 0, 0, 0, 0] -> [0, 0, 0, 0, 0, 0];
    [c, 1, 1, x, x, 0, 0, 0, 0, 0] -> [0, 0, 0, 0, 0, 0];
    [c, 1, 1, x, x, 0, 0, 0, 0, 0] -> [0, 0, 0, 0, 0, 0];
    [c, 1, 1, x, x, 0, 0, 0, 0, 0] -> [0, 0, 0, 0, 0, 0];

end SC_MODE_DRAM_CTRL_3;
```

240799-37

```
module  SC_MODE_DRAM_CTRL_1  flag '-r4'

title    'STATIC COLUMN MODE DRAM CONTROLLER - PLD 1, INTEL CORPORATION
" Cycle Tracking Logic
" Implemented with Intel 85C224 EPLD.
  SCy  device     'E224';

  x           =    .X.;                   " ABEL 'don't care' symbol
  c           =    .C.;                   " ABEL 'clocking input' symbol

" Inputs

  CLK pin 1; "P4 input CLK"
  BLAST~      pin  2; "P4 BLAST output
  MEMCS~      pin  3; "Memory Chip Select
  AHOLD       pin  4; "Address HOLD input to P4"
  HIT~        pin  5; "DRAM Page Hit Signal
  BOFF~       pin  6;'Backoff input to P4"
  ADS~        pin  7; "Address Status output of P4"
  RFRQ        pin  8; "Refresh Request Signal
  RESET       pin  9; "System Reset
  BRDY~       pin 10; "Processor burst ready pin.
  MRDY~       pin 11; "Memory ready
  RAS~        pin 14; "Row Address Strobe
  EP          pin 23; "Refresh indicator - count on RAS~ low

" Output

  RFACK~      pin 15; "Refresh acknowledge
  CIP~        pin 16; " ADS~ active indicator
  M~          pin 17; " AQ0~ Miss state indicator
  CT          pin 18; " AHOLD with ADS~ indicator
  PCHG        pin 19; " Precharge state indicator
  Q1          pin 20; " Precharge state indicator
  ALD pin 21; " Address Latch Disable
  adlst~      pin 22; " ADL state variable

state_diagram [CIP~, M~]

  state [1, 1]: if RESET then [1, 1] else
                if AHOLD # !RFACK~ # EP then [1, 1] else
                if !ADS~ # CT then [0, 1] else [1, 1];

  state [0, 1]: if RESET # !BOFF~ # MEMCS~
                  then [1, 1] else
                  if HIT~ & !RAS~ & !MRDY~ then [0, 0] else
                  If (!MRDY~ # (!BRDY~ & !BLAST~)) then [1, 1]
                  else [0, 1];

  state [0, 0]: if RESET # !BOFF~ then [1, 1] else
                  if !PCHG & (CT # !ADS~) then [0, 1] else
                  if !PCHG & !CT then [1, 1] else
                  [0, 0];
```

240799-38

```
  state [1, 0]: goto [1, 1];

state_diagram [PCHG, Q1]

  state [0, 0]:       if RESET then [0, 0] else
                        if !RAS~ then [1, 0] else
                        if RAS~ & !RFACK~ then [0, 1] else [0, 0];

  state [1, 0]:       if RESET then [0, 0] else
                        if RAS~ & !EP then [0, 0] else
                        if RFACK~ & EP & !RAS~ then     [1, 1] else
                        if RAS~ & EP then [0, 1] else [1, 0];

  state [0, 1]: goto [1, 0];

  state [1, 1]: goto [0, 0];

state_diagram [CT]

  state [0]:          if RESET then [0] else
                      if !ADS~ & (AHOLD # !RFACK~ # !M~ # EP) then [1] else [0];

  state [1]:          if RESET # !BOFF~ then [0] else
                      if !CIP~ & M~ then [0] else [1];

state_diagram [RFACK~]

  state[1]:           if RESET then [1] else
                      if !CIP~ & RFRQ & !MRDY~ & !HIT~ then [0] else
                      if !CIP~ & RFRQ & (!BRDY~ & !BLAST~) #
                      RFRQ & CIP~ & ADS~ then [0] else [1];

  state[0]:           if RESET # !BOFF~ then [1] else
                      if RAS~ then [1] else [0];

state_diagram [ALD, adlst~]

  state [0, 1]:       if RESET then [0, 1] else
                        if !ADS~ # !CIP~ & !MEMCS~ then [1, 0] else [0, 1];

  state [1, 0]:       if RESET then [0,1] else
                        if !CIP~ & MEMCS~ then [0, 1] else
                        if HIT~ & !MRDY~ then [1, 1] else
                        if !HIT~ & !MRDY~ then [0, 1] else
                        if !BRDY~ & !BLAST~ then [0, 1] else [1, 0];

  state [1, 1]:       if RESET then [0, 1] else
                        if !CIP~ & (!PCHG # MEMCS~) then [0, 1] else [1, 1];

  state [0, 0]: goto [0, 1];

test_vectors
```

240799-39

5

```
  ([CLK,BLAST~,MEMCS~,AHOLD,HIT~,BOFF~,ADS~,RFRQ,RESET] ->
  [BRDY~,MRDY~,RAS~,EP,RFACK~,CIP~,M~,CT,PCHG,Q1,ALD,adlst~])

"CBMAHBARRBRRE RAMCPQAa
"LLEHIODFERDAP FQ ~ TC1Ld
"KAMOTFSRSDYS~ A0   H DI
"  SCL~F~QEY~~  C~  G  s
"  TSD  ~    T~     K    t
"  ~ ~                   ~
"
"
"

  [c, x, x, x, x, x, 1, x, 1, x, x, x, x] -> [1, 1, 1, 0, 0, 0, 0, 1];
  [c, x, x, x, x, x, 1, x, 1, x, x, x, x] -> [1, 1, 1, 0, 0, 0, 0, 1];
  [c, 1, x, 0, x, 1, 1, 0, 1, 0, 1, 1, 0] -> [1, 1, 1, 0, 0, 0, 0, 1];
  [c, 1, x, 0, x, 1, 0, 0, 0, 0, 1, 1, 0] -> [1, 0, 1, 0, 0, 0, 1, 0];
  [c, 1, x, 0, x, 1, 1, 0, 1, 0, 1, 1, 0] -> [1, 0, 1, 0, 0, 0, 1, 0];
  [c, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0] -> [1, 0, 1, 0, 0, 0, 1, 0];
  [c, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0] -> [1, 0, 1, 0, 1, 0, 1, 0];
  [c, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0] -> [1, 0, 1, 0, 1, 0, 1, 0];
  [c, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0] -> [1, 0, 1, 0, 1, 0, 1, 0];
  [c, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0] -> [1, 0, 1, 0, 1, 0, 1, 0];
  [c, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0] -> [1, 0, 1, 0, 1, 0, 1, 0];
  [c, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0] -> [1, 1, 1, 0, 1, 0, 0, 1];
  [c, x, x, 0, x, 1, 1, 0, 1, 0, 1, 0, 0] -> [1, 1, 1, 0, 1, 0, 0, 1];

end SC_MODE_DRAM_CTRL_1;
```

240799-40

```
module  SC_MODE_DRAM_CTRL_7  flag '-r4'

title  'STATIC COLUMN MODE DRAM CONTROLLER - PLD 7, INTEL CORPORATION'
" This PLD generates DATASL and WE
" Implemented with the Intel 85C220 EPLD.

  SC7  device      'E0320';

  x          =    .X.;                  " ABEL 'don't care' symbol
  c          =    .C.;                  " ABEL 'clocking input' symbol

" Inputs

  CLK pin 1; "P4 input CLK"
  BRDY~      pin  2; "Burst Ready
  CIP~       pin  3; "Cycle OK
  MEMCS~     pin  4; "memory select
  LA2 pin 5; "Latched A2.
  CAS00~     pin  6; "CAS output Bank1
  CAS10~     pin  7; "CAS output Bank1
  LW_R       pin  8; "CPU W/R latched~
  RESET      pin  9; "System Reset
  BLAST~     pin 12; "CPU BLAST~ output
  BOFF~      pin 13; "CPU Backoff input
  HIT~       pin 19;
" Output

  DATASEL    pin 14; " Bank select for reads
  RS~        pin 15; " state variable
  RALE~      pin 16; " state variable
  WE~        pin 17; " Write Enable posted writes
  BSEL       pin 18; " Selects read or write data path

state_diagram [DATASEL, RS~]

  state [1, 1]: if RESET then [1, 1] else
                  if !CIP~ & !LA2 & !LW_R & !MEMCS~ then [0, 0] else
                  if !CIP~ & LA2 & !LW_R & !MEMCS~ then [1, 0] else [1, 1];

  state [1, 0]: if RESET # !BOFF~ # (!BRDY~ & !BLAST~) then [1, 1] else
                  if !BRDY~ & BLAST~ then [0, 0] else [1, 0];

  state [0, 0]: if RESET # !BOFF~ # (!BRDY~ & !BLAST~) then [1, 1] else
                  if !BRDY~ & BLAST~ then [1, 0] else [0, 0];

  state [0, 1]: goto [1, 1];

state_diagram [WE~]

  state [1]:    if RESET then [1] else
                if LW_R & !CIP~ & !MEMCS~ then [0] else [1];
```

240799–41

5

```
   state [0]:     if RESET # !BOFF~ then [1] else
                  if LW_R & !CIP~ & !MEMCS~ then [0] else
                  if CAS00~ + CAS10~ then [1];

state_diagram [RALE~]

   state [0]:     if RESET then [0] else
                  if !CIP~ & HIT~ & !MEMCS~ then [1] else [0];

   state [1]:     if RESET # !BOFF~ then [0] else
                  if !HIT~ then [0] else [1];

end SC_MODE_DRAM_CTRL_7;
```

240799-42

```
module   SC_MODE_DRAM_CTRL_11 flag '-r4'

title  'STATIC COLUMN MODE DRAM CONTROLLER - PLD 11, INTEL CORPORATION'
" This PLD generates the mux enables write enables and WIP#
" Implemented with the Intel 85C220 EPLD.

  SCw   device      'E0320';

  x           =     .X.;              " ABEL 'don't care' symbol
  c           =     .C.;              " ABEL 'clocking input' symbol

" Inputs

  CLK pin 1; "P4 input CLK"
  LA2 pin 2; "Latched A2.
  CIP~        pin  3; "Cycle OK
  MEMCS~      pin  4; "Memory Chip select.
  RESET       pin  5; "DRAM Page Hit Signal
  LW_R        pin  6; "latched CPU W/R#
  C01 pin 7; "Write indication Bank0
  CAS01~      pin  8; "
  C11 pin 9; "Write indication Bank1
  CAS11~      pin 19; "

" Output

  WIP~        pin 12; "New Wip signal comb
  MEN0~       pin 13; "Mux enables
  WE0~        pin 14; "
  LWIP~       pin 15; "Latched WIP~
  dum         pin 16; "
  WE1~        pin 17; "
  MEN1~       pin 18; " Mux enable Bank1

state_diagram [WE0~]

  state [1]:  if RESET then [1] else
              if !CIP~ & LW_R & !MEMCS~ & !LA2 then [0];

  state [0]:  if RESET then [1] else
              if !C01 then [0] else
              if C01 then [1];

  state_diagram [WE1~]

  state [1]:  if RESET then [1] else
              if !CIP~ & LW_R & !MEMCS~ & LA2 then [0];

  state [0]:  if RESET then [1] else
              if !C11 then [0] else
              if C11 then [1];

state_diagram [LWIP~]
```

240799–43

```
  state [1]:    if !C01 # !C11 then [0] else [1];

  state [0]:    if RESET then [1] else
                if !C01 # !C11 then [0] else [1];

state_diagram [MEN0~]

  state [1]:    if RESET then [1] else
                if !CIP~ & LW_R & !MEMCS~ & !LA2 then [0];

  state [0]:    if RESET then [1] else
                if !C01 & CAS01~ then [0] else
                if !CIP~ & LW_R & !MEMCS~ & !LA2 & !CAS01~ then [0] else
                if !CAS01~ then [1];

state_diagram [MEN1~]

  state [1]:    if RESET then [1] else
                if !CIP~ & LW_R & !MEMCS~ & LA2 then [0];

  state [0]:    if RESET then [1] else
                if !C11 & CAS11~ then [0] else
                if !CIP~ & LW_R & !MEMCS~ & !LA2 & !CAS11~ then [0] else
                if !CAS11~ then [1];

equations

                !WIP~ = !LWIP~ # !C01 # !C11;

"test_vectors

"([CLK,M IO~,CIP~,MEMCS~,HIT~,RFACK,ADS~,W_R,RESET,CAS0~,BOFF~,BLAST~]
"->      [RAS~,MRDY~,DALE~,LWR,BRDY~])
"
"CMAMHRAWRCBBR  MDLB
"L_QEIFD_EAOLA  RAWR
"KI0MTASRSSFAS  DLRD
"  0~C~F~ E0FS~ YE Y
"    S K  T~~T  ~~ ~
"    ~      ~
"
"
"
"
" [c, x, x, x, x, x, 1, x, 1, x, x, x, x] -> [x, x, x, x];
" [c, x, 1, 1, x, x, 1, x, 1, x, x, x, x] -> [1, 0, 0, 1];
" [c, 1, 1, 1, x, 1, 1, x, 0, 1, 0, 1, 1] -> [1, 0, 0, 1];
" [c, 1, 1, 1, x, 1, 1, x, 0, 1, 0, 1, 1] -> [1, 0, 0, 1];
" [c, 1, 1, 1, x, 1, 0, 1, 0, 1, 0, 1, 1] -> [1, 1, 1, 1];
" [c, 1, 0, 0, 0, 1, 1, x, 0, 1, 0, 1, 1] -> [0, 0, x, 1];
" [c, 1, 0, 0, 0, 1, 1, x, 0, 1, 0, 1, 0] -> [1, 0, x, 1];
" [c, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0] -> [0, 1, 1, 1];
" [c, 1, 0, 0, 0, 1, 1, x, 0, 0, 0, 1, 0] -> [1, 0, x, 1];
```

240799–44

```
" [c, 1, 1, x, 0, 1, 0, 1, 0, 1, 0, 1, 0]  -> [0, 1, 1, 1];
" [c, 1, 0, 0, 0, 1, 1, x, 0, 1, 0, 1, 0]  -> [1, 0, x, 1];
" [c, 1, 1, x, 0, 1, 1, x, 0, 1, 0, 1, 0]  -> [1, 0, x, 1];
" [c, 1, 1, 1, x, 1, 1, x, 0, 1, 0, 1, 0]  -> [1, 0, x, 1];
"
end SC_MODE_DRAM_CTRL_11;
```

240799-45

5

```
module  SC_MODE_DRAM_CTRL_11 flag '-r4'

title  'STATIC COLUMN MODE DRAM CONTROLLER - PLD 11, INTEL CORPORATION'
" This PLD generates the mux enables write enables and WIP#
" Implemented with the Intel 85C220 EPLD.

  SCw  device  'E0320';

  x           =    .X.;                " ABEL 'don't care' symbol
  c           =    .C.;                " ABEL 'clocking input' symbol

" Inputs

  CLK pin 1; "P4 input CLK"
  LA2 pin 2; "Latched A2.
  CIP~        pin  3; "Cycle OK
  MEMCS~      pin  4; "Memory Chip select.
  RESET       pin  5; "DRAM Page Hit Signal
  LW_R        pin  6; "latched CPU W/R#
  C01 pin 7; "Write indication Bank0
  CAS01~      pin  8; "
  C11 pin  9; "Write indication Bank1
  CAS11~      pin 19; "

" Output

  WIP~        pin 12; "New Wip signal comb
  MEN0~       pin 13; " Mux enables
  WE0~        pin 14; "
  LWIP~       pin 15; "Latched WIP~
  dum         pin 16; "
  WE1~        pin 17; "
  MEN1~       pin 18; " Mux enable Bank1

state_diagram [WE0~]

  state [1]:  if RESET then [1] else
              if !CIP~ & LW_R & !MEMCS~ & !LA2 then [0];

  state [0]:  if RESET then [1] else
              if !C01 then [0] else
              if C01 then [1];

state_diagram [WE1~]

  state [1]:  if RESET then [1] else
              if !CIP~ & LW_R & !MEMCS~ & LA2 then [0];

  state [0]:  if RESET then [1] else
              if !C11 then [0] else
              if C11 then [1];

state_diagram [LWIP~]
```

240799-46

```
   state [1]:    if !C01 # !C11 then [0] else [1];

   state [0]:    if RESET then [1] else
                 if !C01 # !C11 then [0] else [1];

state_diagram [MEN0~]

   state [1]:    if RESET then [1] else
                 if !CIP~ & LW_R & !MEMCS~ & !LA2 then [0];

   state [0]:    if RESET then [1] else
                 if !C01 & CAS01 ~ then [0] else
                 if !CIP~ & LW_R & !MEMCS~ & !LA2 & !CAS01~ then [0] else
                 if !CAS01~ then [1];

state_diagram [MEN1~]

   state [1]:    if RESET then [1] else
                 if !CIP~ & LW_R & !MEMCS~ & LA2 then [0];

   state [0]:    if RESET then [1] else
                 if !C11 & CAS11~ then [0] else
                 if !CIP~ & LW_R & !MEMCS~ & !LA2 & !CAS11~ then [0] else
                 if !CAS11~then [1];

equations

     !WIP~ = !LWIP~ # !C01 # !C11;

"test_vectors

"([CLK,M_IO~,CIP~,MEMCS~,HIT~,RFACK,ADS~,W_R,RESET,CAS0~,BOFF~]
"  ->     [BLAST~,RAS~,MRDY~,DALE~,LWR,BRDY~])
"
C M A M H R A W R C B B R   M D L B
"L _ Q E I F D _ E A O L A   R A W R
"K I 0 M T A S R S S F A S   D L R D
"  O ~ C ~ F ~  E 0 F S ~   Y E  Y
"      S  K      T ~ ~T      ~ ~  ~
"      ~            ~
"
"
"
"
" [c, x, x, x, x, x, 1, x, 1, x, x, x, x]  -> [x, x, x, x];
" [c, x, 1, 1, x, x, 1, x, 1, x, x, x, x]  -> [1, 0, 0, 1];
" [c, 1, 1, 1, x, 1, 1, x, 0, 1, 0, 1, 1]  -> [1, 0, 0, 1];
" [c, 1, 1, 1, x, 1, 1, x, 0, 1, 0, 1, 1]  -> [1, 0, 0, 1];
" [c, 1, 1, 1, x, 1, 0, 1, 0, 1, 0, 1, 1]  -> [1, 1, 1, 1];
" [c, 1, 0, 0, 0, 1, 1, x, 0, 1, 0, 1, 1]  -> [0, 0, x, 1];
" [c, 1, 0, 0, 0, 1, 1, x, 0, 1, 0, 1, 0]  -> [1, 0, x, 1];
" [c, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0]  -> [0, 1, 1, 1];
" [c, 1, 0, 0, 0, 1, 1, x, 0, 0, 0, 1, 0]  -> [1, 0, x, 1];
```

240799–47

5

```
" [c, 1, 1, x, 0, 1, 0, 1, 0, 1, 0, 1, 0]  -> [0, 1, 1, 1];
" [c, 1, 0, 0, 0, 1, 1, x, 0, 1, 0, 1, 0]  -> [1, 0, x, 1];
" [c, 1, 1, x, 0, 1, 1, x, 0, 1, 0, 1, 0]  -> [1, 0, x, 1];
" [c, 1, 1, 1, x, 1, 1, x, 0, 1, 0, 1, 0]  -> [1, 0, x, 1];
"

end SC_MODE_DRAM_CTRL_11;
```

240799-48

```
module   SC_MODE_DRAM_CTRL_8   flag '-r4'

title  'STATIC COLUMN MODE DRAM CONTROLLER - PLD 8, INTEL CORPORATION'
" This PLD generates CAS1 (CAS for bank 1)
" Implemented with the Intel 85C220 EPLD.

  SC8   device   'E0320';

  x            =     .X.;                    " ABEL 'don't care' symbol
  c            =     .C.;                    " ABEL 'clocking input' symbol

" Inputs

  CLK pin 1; "P4 input CLK"
  RFACK        pin  2; "Refresh Acknowledge
  CIP~         pin  3; "Cycle OK
  LA2 pin 4; "Latched A2.
  HIT~         pin  5; "DRAM Page Hit Signal
  BOFF~        pin  6; "Backoff input to P4"
  LW_R~        pin  7; "
  RAS~         pin  8; "
  RESET        pin  9; "System Reset
  RDY~         pin 12; " Processor RDY#
  MEMCS~       pin 13; " Memory Chip Select
  BRDY~        pin 18; " Processor BREADY#
  BLAST~       pin 19; " Processor BLAST#

" Output

  CAS10~       pin 14; " CAS1 byte 0,2
  C1           pin 15; " state variable
  C2           pin 16; " state variable
  CAS11~       pin 17; " CAS1 byte 1,3

state_diagram [CAS10~,CAS11~,C1,C2]

  state [1, 1, 1, 1]:  if RESET # !BOFF~ then [1, 1, 1, 1] else
                          if !RFACK & !CIP~ & LA2 & LW_R~ & !MEMCS~ then
                          [1, 1, 0, 1] else if !RFACK & !CIP~ & !LW R~ & !RAS~
                          & !HIT~ & !MEMCS~ # (RFACK & RAS~) then
                          [0, 0, 1, 1] else [1, 1, 1, 1];

  state [1, 1, 0, 1]:  if RESET # !BOFF~ then [1, 1, 1, 1] else
                       if !RAS~ & RDY~ then [0, 0, 0, 0] else
                       [1, 1, 0, 1];

  state [0, 0, 0, 0]:  if RESET # !BOFF~ then [1, 1, 1, 1] else
                          if !CIP~ & LA2 & LW_R~ & !MEMCS~ then [1, 1, 0, 0] else
                          if CIP~ # (!CIP~ & (MEMCS~ # !LW_R~)) # (!CIP~ & LW_R~&
                          !LA2) then [1, 1, 1, 1] else [0, 0, 0, 0];

  state [1, 1, 0, 0]:  if !RAS~ then [0, 0, 0, 0] else [1, 1, 0, 0];
```

240799-49

```
    state [0, 0, 1, 1]:  if RESET # !BOFF~ then [1, 1, 1, 1] else
                         if !BRDY~ & !BLAST~ & !RFACK then
                         [1, 1, 1, 1] else
                           if !BRDY~ & BLAST~ & LA2 then [1, 1, 1, 0] else
                           if !BRDY~ & BLAST~ & !LA2 then [0, 0, 1, 0] else
                           if RFACK then [0, 0, 1, 0] else
                           if BRDY~ & !RFACK then [0, 0, 1, 1];

    state [1, 1, 1, 0]:  if RESET then [1, 1, 1, 1] else
                           if !BOFF~ then [1, 1, 1, 0] else
                           if !BRDY~ & BLAST~ then [0, 0, 1, 1] else
                           if !BRDY~ & !BLAST~ then [1, 1, 1, 1] else
                           [1, 1, 1, 0];

    state [0, 0, 1, 0]:  if RESET # !BOFF~ then [1, 1, 1, 1] else
                           if !BRDY~ & BLAST~ then [1, 1, 1, 0] else
                           if !BRDY~ & !BLAST~ # BRDY~ then [1, 1, 1, 1];

test_vectors

  ([CLK,RFACK,CIP~,LA2,HIT,~BOFF~,LW_R~,RAS~,RESET,RDY~,MEMCS~,BRDY~]
->     [BLAST~,CAS10~,C1,C2,CAS11~])

"CRALHBLRRRMBB   CCCC
"LFQAIOWAEDERL   A12A
"KA02TFRSSYMDA   S  S
" C~ ~F~ ~E~CYS  0  0
" K  ~   T S  T 0 1
"         ~   ~  ~   ~
"
"
"

   [c, x, x, x, x, x, 1, x, 1, x, x, x, x] -> [x, x, x, x];
   [c, x, 1, 1, x, x, 1, x, 1, x, x, x, x] -> [1, 1, 1, x];
   [c, x, 1, 1, x, 1, 1, 0, 0, 1, 0, 1, 1] -> [1, 1, 1, 1];
   [c, 0, 1, 1, x, 1, 1, 0, 0, 1, 0, 1, 1] -> [1, 1, 1, 1];
   [c, 0, 1, 1, x, 1, 1, 0, 0, 1, 0, 1, 1] -> [1, 1, 1, 1];
   [c, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1] -> [1, 0, 1, 1];
   [c, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1] -> [1, 0, 1, 1];
   [c, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1] -> [0, 0, 0, 0];
   [c, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1] -> [1, 0, 0, 1];
   [c, 0, 1, x, 0, 1, 1, 0, 0, 1, 0, 1, 1] -> [0, 0, 0, 0];
   [c, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1] -> [1, 0, 0, 1];
   [c, 0, 1, x, 0, 1, 1, 0, 0, 1, 0, 1, 1] -> [0, 0, 0, 0];
   [c, 0, 1, 0, x, 1, 1, 0, 0, 1, 0, 1, 1] -> [1, 1, 1, 1];
end SC_MODE_DRAM_CTRL_8;
```

240799-50

```
module   PG_MODE_DRAM_CTRL_2   flag '-r4'

title    'PAGE MODE DRAM CONTROLLER - PLD 2, INTEL CORPORATION'
" This PLD generates CAS0
" Implemented with the Intel 85C220 EPLD.

  SC2   device   'E0320';

  x          =    .X.;                 " ABEL 'don't care' symbol
  c          =    .C.;                 " ABEL 'clocking input' symbol

" Inputs

  CLK pin 1; "P4 input CLK"
  RFACK       pin  2; "Refresh Acknowledge
  CIP~        pin  3; "Cycle OK
  LA2 pin 4; "Latched A2.
  HIT~        pin  5; "DRAM Page Hit Signal
  BOFF~       pin  6; "Backoff input to P4"
  LW_R~       pin  7; "
  RAS~        pin  8; "
  RESET       pin  9; "System Reset
  RDY~        pin 12; "Processor RDY#
  MEMCS~      pin 13; "Memory Chip Select
  BRDY~       pin 18; "Processor BREADY#
  BLAST~      pin 19; "Processor BLAST#

" Output

  CAS10~      pin 14; " CAS1 byte 0,2
  C1          pin 15; " state variable
  C2          pin 16; " state variable
  CAS11~      pin 17; " CAS1 byte 1,3

state_diagram [CAS10~, CAS11~, C1, C2]

  state [1, 1, 1, 1]:  if RESET # !BOFF~ then [1, 1, 1, 1] else
                         if !RFACK & !CIP~ & !LA2 & LW R~ & !MEMCS~ then
                         [1, 1, 0, 1] else if !RFACK & !CIP~ & !LW_R~ & !RAS~
                         & !HIT~ & !MEMCS~ # (RFACK & RAS~) then
                         [0, 0, 1, 1] else [1, 1, 1, 1];

  state [1, 1, 0, 1]:  if RESET # !BOFF~ then [1, 1, 1, 1] else
                       if !RAS~ & RDY~ then [0, 0, 0, 0] else
                       [1, 1, 0, 1];

  state [0, 0, 0, 0]:  if RESET # !BOFF~ then [1, 1, 1, 1] else
                       if !CIP~ & !LA2 & LW_R~ & !MEMCS~ then [1, 1, 0, 0] else
                       if CIP~ # (!CIP~ & (MEMCS~ # !LW_R)) # (!CIP~ & LW_R~ &
                       LA2) then [1, 1, 1, 1] else [0, 0, 0, 0];

  state [1, 1, 0, 0]:  if !RAS~ then [0, 0, 0, 0] else [1, 1, 0, 0];
```

240799-51

```
    state [0, 0, 1, 1]:  if RESET # !BOFF~ then [1, 1, 1, 1] else
                         if !BRDY~ & !BLAST~ & !RFACK then
                         [1, 1, 1, 1] else
                           if !BRDY~ & BLAST~ & !LA2 then [1, 1, 1, 0] else
                           if !BRDY~ & BLAST~ & LA2 then [0, 0, 1, 0] else
                           if RFACK then [0, 0, 1, 0] else
                           if BRDY~ & !RFACK then [0, 0, 1, 1];

    state [1, 1, 1, 0]:  if RESET then [1, 1, 1, 1] else
                           if !BOFF~ then [1, 1, 1, 0] else
                           if !BRDY~ & BLAST~ then [0, 0, 1, 1] else
                           if !BRDY~ & !BLAST~ then [1, 1, 1, 1] else
                           [1, 1, 1, 0];

    state [0, 0, 1, 0]:  if RESET # !BOFF~ then [1, 1, 1, 1] else
                           if !BRDY~ & BLAST ~ then [1, 1, 1, 0] else
                           if !BRDY~ & !BLAST~ # BRDY~ then [1, 1, 1, 1];

test_vectors

    ([CLK,RFACK,CIP~,LA2,HIT~,BOFF~,LW_R~,RAS~,RESET,RDY~,MEMCS~,BRDY~]
    ->      [BLAST~,CAS10~,C1,C2,CAS11~])

"CRALHBLRRRMBB    CCCC
"LFQAIOWAEDERL    A12A
"KA02TFRSSYMDA    S  S
" C~  ~F~~E~CYS  0  0
" K    ~  T  S  T  0  1
"             ~   ~   ~   ~
"
"
"

    [c, x, x, x, x, x, 1, x, 1, x, x, x, x] -> [x, x, x, x];
    [c, x, 1, 0, x, x, 1, x, 1, x, x, x, x] -> [1, 1, 1, 1];
    [c, 0, 1, 0, x, 1, 1, 0, 0, 1, 0, 1, 1] -> [1, 1, 1, 1];
    [c, 0, 1, 0, x, 1, 1, 0, 0, 1, 0, 1, 1] -> [1, 1, 1, 1];
    [c, 0, 1, 0, x, 1, 1, 0, 0, 1, 0, 1, 1] -> [1, 1, 1, 1];
    [c, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1] -> [1, 0, 1, 1];
    [c, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1] -> [1, 0, 1, 1];
    [c, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1] -> [0, 0, 0, 0];
    [c, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1] -> [1, 0, 0, 1];
    [c, 0, 1, x, 0, 1, 1, 0, 0, 1, 0, 1, 1] -> [0, 0, 0, 0];
    [c, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1] -> [1, 0, 0, 1];
    [c, 0, 1, x, 0, 1, 1, 0, 0, 1, 0, 1, 1] -> [0, 0, 0, 0];
    [c, 0, 1, 1, x, 1, 1, 0, 0, 1, 0, 1, 1] -> [1, 1, 1, 1];
end PG_MODE_DRAM_CTRL_2;
```

240799-52

```
module  SC_MODE_DRAM_CTRL_15 flag '-r4'

title  'STATIC COLUMN MODE DRAM CONTROLLER - PLD 15, INTEL CORPORATION'
" This PLD combines ready signals
" Implemented with the Intel 85C220 EPLD.

  SC15K device  'E0320';

  x         =    .X.;                  " ABEL 'don't care' symbol
  c         =    .C.;                  " ABEL 'clocking input' symbol

" Inputs

  MEMCS~    pin  1; "
  JRDY~     pin  2; "
  MRDY~     pin  3; "
  BRDY~     pin  4; "
  ALD pin 5; "
  CKEN~     pin  6; "
  SKEN~     pin  7; "
  BRDYO~    pin  8; "
  M~        pin  9; "miss indicator for CIP~
  CIP~      pin 11; " Cycle indicator

" Output

  WEN~      pin 12; "Write enable for write latches
  RDY~      pin 13; "to 486
  MRDYCS~   pin 14; "
  MALD~     pin 15; "Modified ALD for FF's
  dum10     pin 16; "
  PBRDY~    pin 17; "
  KEN~      pin 18; "
  DRDY~     pin 19; "

equations

  !MALD~ = (!MEMCS~ & !ALD);

  !RDY~ = (!MRDY~ & M~ & !MEMCS~) # !JRDY~;

  !MRDYCS~ = (!MRDY~ & M~ & !MEMCS~);

  !WEN~ = !CIP~ & M~;

  !DRDY~ = !BRDY~ # !MRDYCS~;

  KEN~ = SKEN~ & CKEN~;

  PBRDY~ = BRDY~ & BRDYO~;
"test_vectors
```

240799-53

5

```
" ([CLK,RESET] - >
" [RESETO])

"C R    R
"L E    E
"K S    S
"   E    E
"   T    T
"        O
"
"
"

" [c, 0] - > [x];
" [c, 0] - > [0];
" [c, 0] - > [0];
" [c, 0] - > [0];
" [c, 0] - > [0];
"[c, 0] - > [0];
" [c, 0] - > [0];
" [c, 1] - > [1];
" [c, 1] - > [1];
" [c, 1] - > [1];
" [c, 1] - > [1];
" [c, 1] - > [1];
" [c, 1] - > [1];
" [c, 1] - > [1];
" [c, 1] - > [1];
" [c, 0] - > [0];
" [c, 0] - > [0];
" [c, 0] - > [0];

end SC_MODE_DRAM_CTRL_15;
```

240799-54

```
module   SC_MODE_DRAM_CTRL_17 flag '-r4'

title  'STATIC COLUMN MODE DRAM CONTROLLER - PLD 17, INTEL CORPORATION'
" This PLD generates the A0 signal for bank 1
" Implemented with the Intel 85C224 EPLD.

  SC17 device  'E224';

  x          =    .X.;                  " ABEL 'don't care' symbol
  c          =    .C.;                  " ABEL 'clocking input' symbol

" Inputs

  CLK pin 1; "P4 input CLK"
  BRDY~      pin  2; "Burst Ready
  CIP~       pin  3; "Cycle OK
  MEMCS~     pin  4; "memory select
  LA313      pin  5; "Latched A2.
  DATASEL    pin  6; "Refresh acknowledge"
  RAS~       pin  7; "Row address strobe
  LW_R       pin  8; "CPU W/R latched~
  RESET      pin  9; "System Reset
  BLAST~     pin 10; "CPU BLAST~ output
  A3         pin 11; "CPU Backoff input
  ALD pin 14; "Address Latch disable
  dum1       pin 15;
  WE1~       pin 22; "Write enable
  dum2       pin 23; "Address Latch disable

" Output

  B10MA0     pin 21; "Bank 1 A0
  B1A pin 20; "Burst A3 bank0
  CS0~       pin 19; " state variable
  dun pin 18; " state variable
  dum        pin 17; " Burst A3 bank1
  B11MA0     pin 16; "Bank 1 A0

state_diagram [B1A, CS0~]

  state [1, 1]: if RESET then [1, 1] else
                  if CIP~ & !ALD & !A3 then [0, 1] else
                  if !CIP~ & !ALD & !A3 then [0, 1] else
                  if !CIP~ & !LW_R & !MEMCS~ & WE1~ then [1, 0] else [1, 1];

  state [0, 1]: if RESET then [1, 1] else
                  if CIP~ & !ALD & A3 then [1, 1] else
                  if !CIP~ & !ALD & A3 then [1, 1] else
                  if !CIP~ & !LW_R & !MEMCS~ & WE1~ then [0, 0] else [0, 1];

  state [1, 0]: if RESET # (!BRDY~ & !BLAST~) then [1, 1] else
                  if !BRDY~ & DATASEL then [0, 0] else [1, 0];
```

240799–55

```
  state [0, 0]: if RESET # (!BRDY~ & !BLAST~) then [1, 1] else
                   if !BRDY~ & DATASEL then [1, 0] else [0, 0];

equations

!B10MA0 = !WE1~ & !LA313 # WE1~ & RAS~ & !LA313 # WE1~ & !RAS~ & !B1A;

!B11MA0 = !WE1~ & !LA313 # WE1~ & RAS~ & !LA313 # WE1~ & !RAS~ & !B1A;

end SC_MODE_DRAM_CTRL_17;
```

240799-56

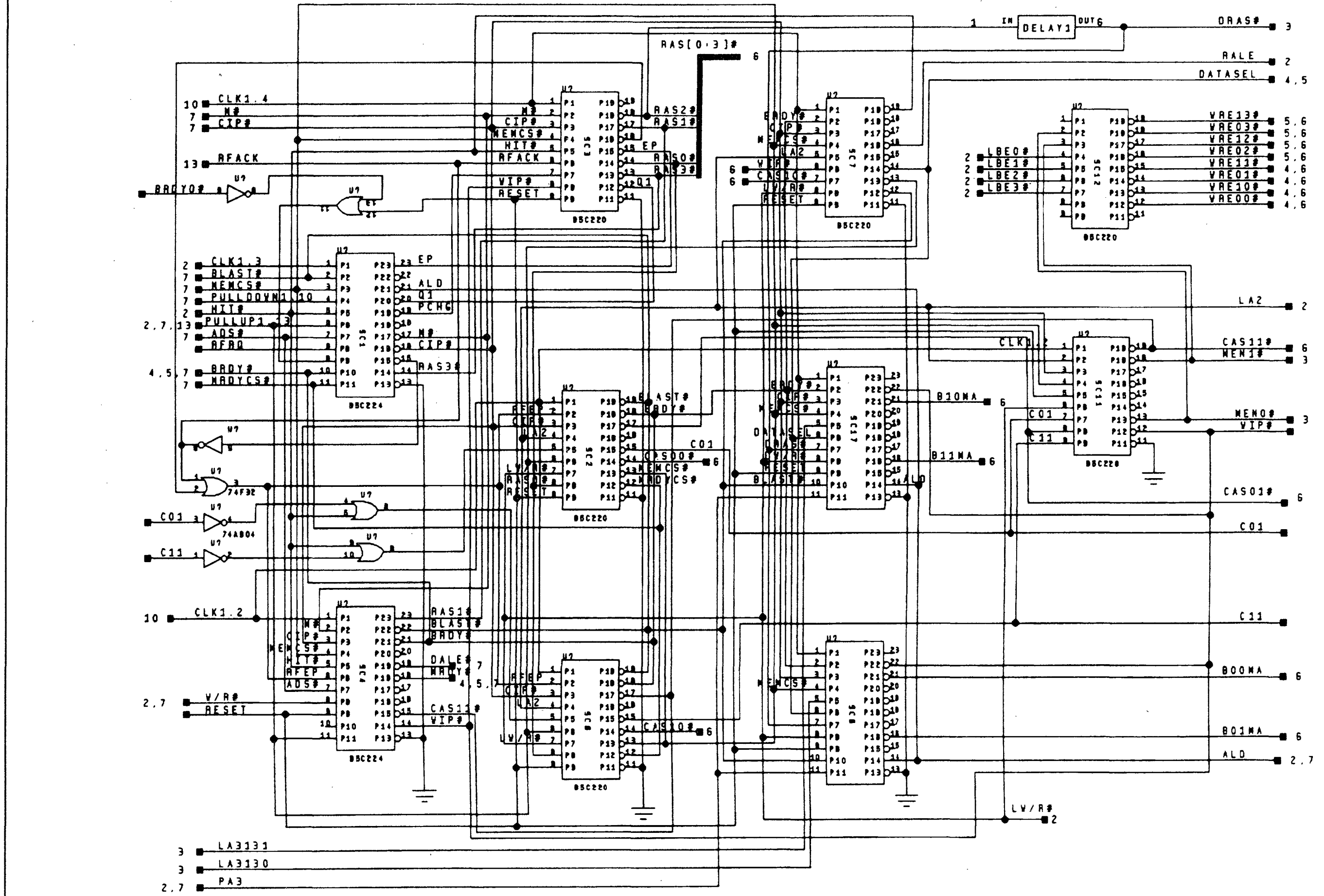
```
module   SC_MODE_DRAM_CTRL_6   flag '-r4'

title   'STATIC COLUMN MODE DRAM CONTROLLER - PLD 6, INTEL CORPORATION'
" This PLD generates A0 for bank 0
" Implemented with the Intel 85C224 EPLD.

  SC6   device   'E224';

  x          =     .X.;              " ABEL 'don't care' symbol
  c          =     .C.;              " ABEL 'clocking input' symbol

" Inputs

  CLK pin 1; "P4 input CLK"
  BRDY~      pin  2; "Burst Ready
  CIP~       pin  3; "Cycle OK
  MEMCS~     pin  4; "memory select
  LA313      pin  5; "Latched A2.
  DATASEL    pin  6; "Refresh acknowledge"
  RAS~       pin  7; "Row address strobe
  LW_R       pin  8; "CPU W/R latched~
  RESET      pin  9; "System Reset
  BLAST~     pin 10; "CPU BLAST~ output
  A3         pin 11; "CPU Backoff input
  ALD pin 14; "Address Latch disable
  dum1       pin 15;
  WE0~       pin 22; "Write enable
  dum2       pin 23; "Address Latch disable

" Output

  B00MA0     pin 21; "Bank 0 A0
  B0A pin 20; " Burst A3 bank 0
  CS0~       pin 19; " state variable
  dun pin 18; " state variable
  dum        pin 17; " Burst A3 bank1
  B01MA0     pin 16; "Bank 0 A0

state_diagram [B0A, CS0~]

  state [1, 1]: if RESET then [1, 1] else
                if CIP~ & !ALD & !A3 then [0, 1] else
                if !CIP~ & !ALD & !A3 then [0, 1] else
                if !CIP~ & !LW_R & !MEMCS~ & WE0~ then [1, 0] else [1, 1];

  state [0, 1]: if RESET then [1, 1] else
                if CIP~ & !ALD & A3 then [1, 1] else
                if !CIP~ & !ALD & A3 then [1, 1] else
                if !CIP~ & !LW_R & !MEMCS~ & WE0~ then [0, 0] else [0, 1];

  state [1, 0]: if RESET # (!BRDY~ & !BLAST~) then [1, 1] else
                if !BRDY~ & !DATASEL then [0, 0] else [1, 0];
```

240799-57

5

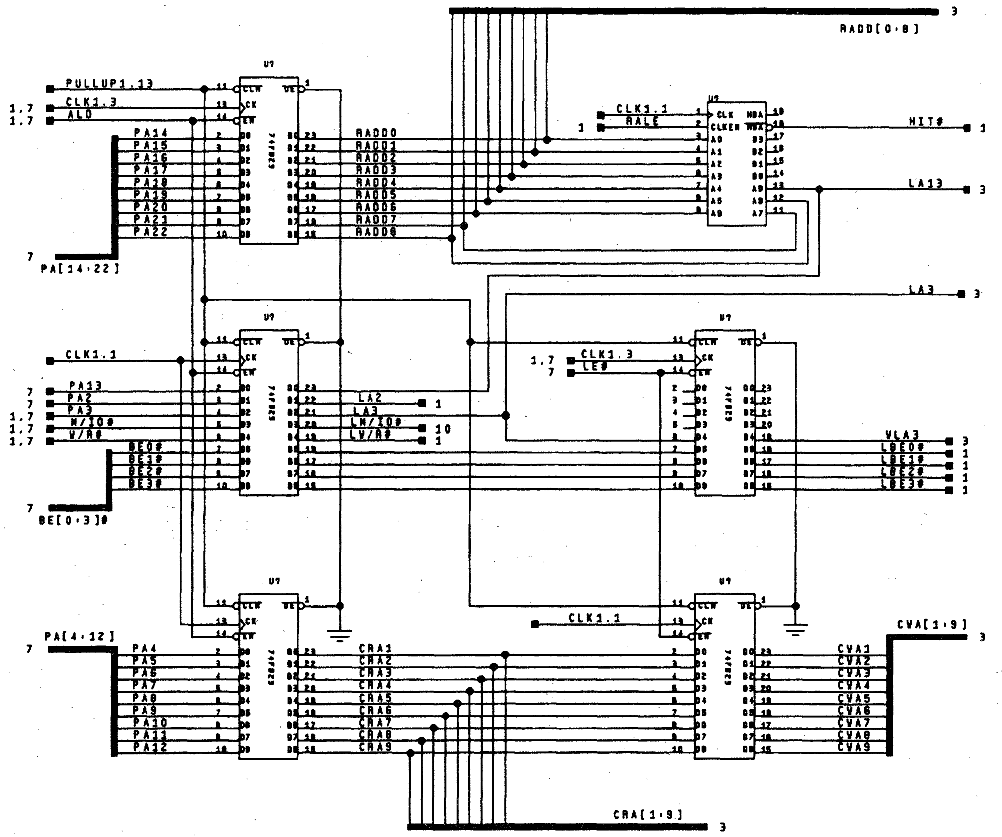
```
    state [0, 0]: if RESET # (!BRDY~ & !BLAST~) then [1, 1] else
                      if !BRDY~ & !DATASEL then [1, 0] else [0, 0];

equations

!B00MA0 = !WE0~ & !LA313 # WE0~ & RAS~ & !LA313 # WE0~ & !RAS~ & !B0A;

!B01MA0 = !WE0~ & !LA313 # WE0~ & RAS~ & !LA313 # WE0~ & !RAS~ & !B0A;

end SC_MODE_DRAM_CTRL_6;
```

240799-58



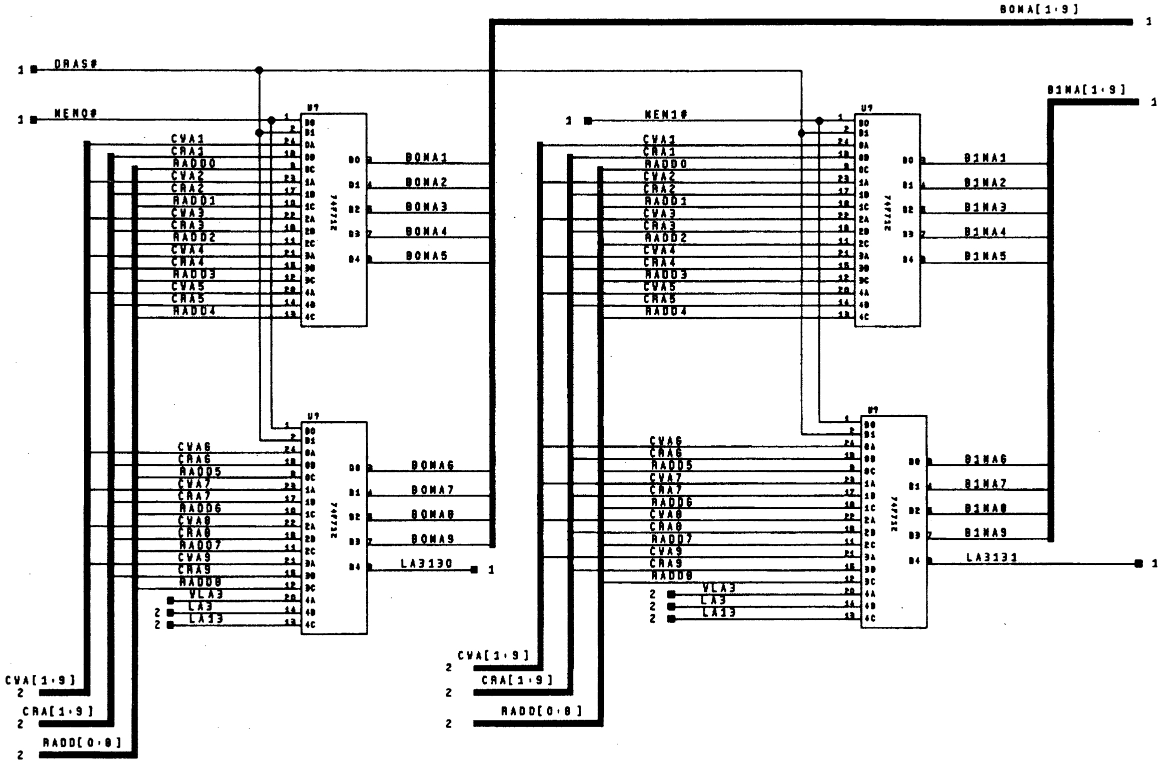
240799-59

Sheet 1 of 10



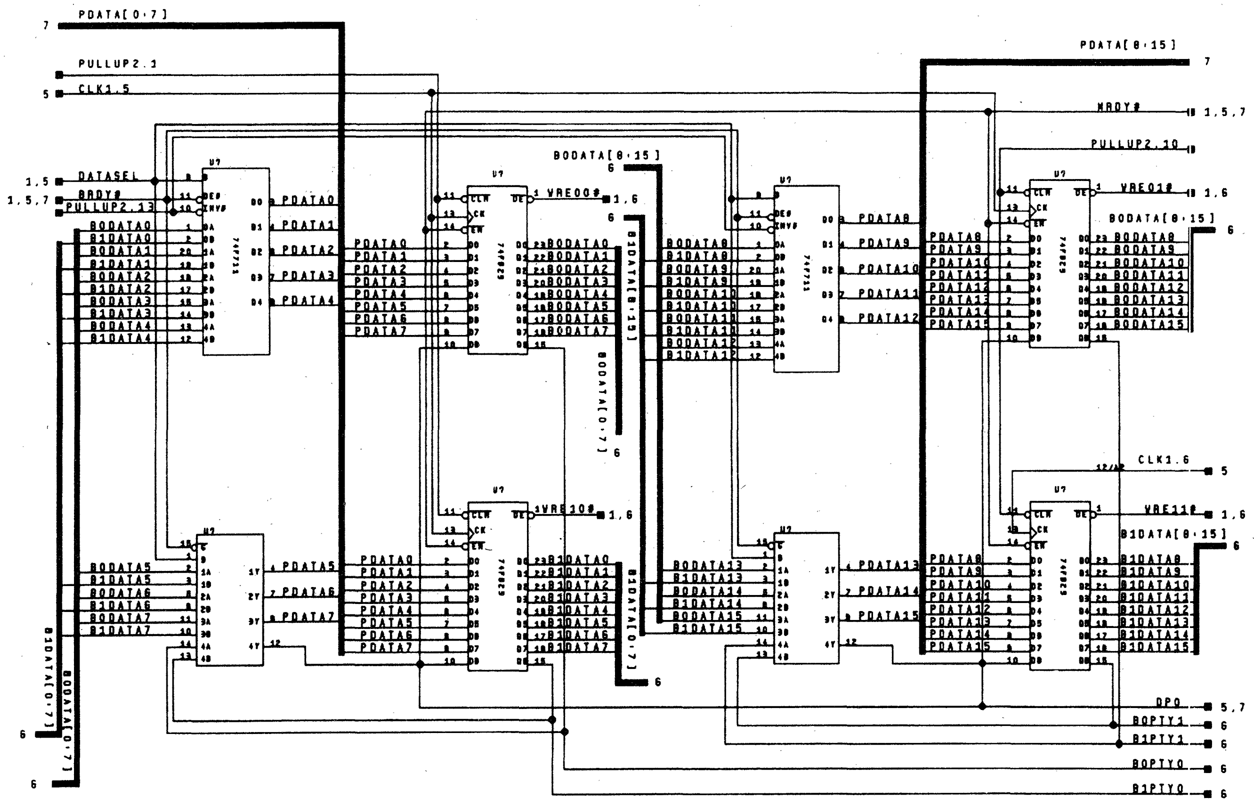
240799-60

Sheet 2 of 10



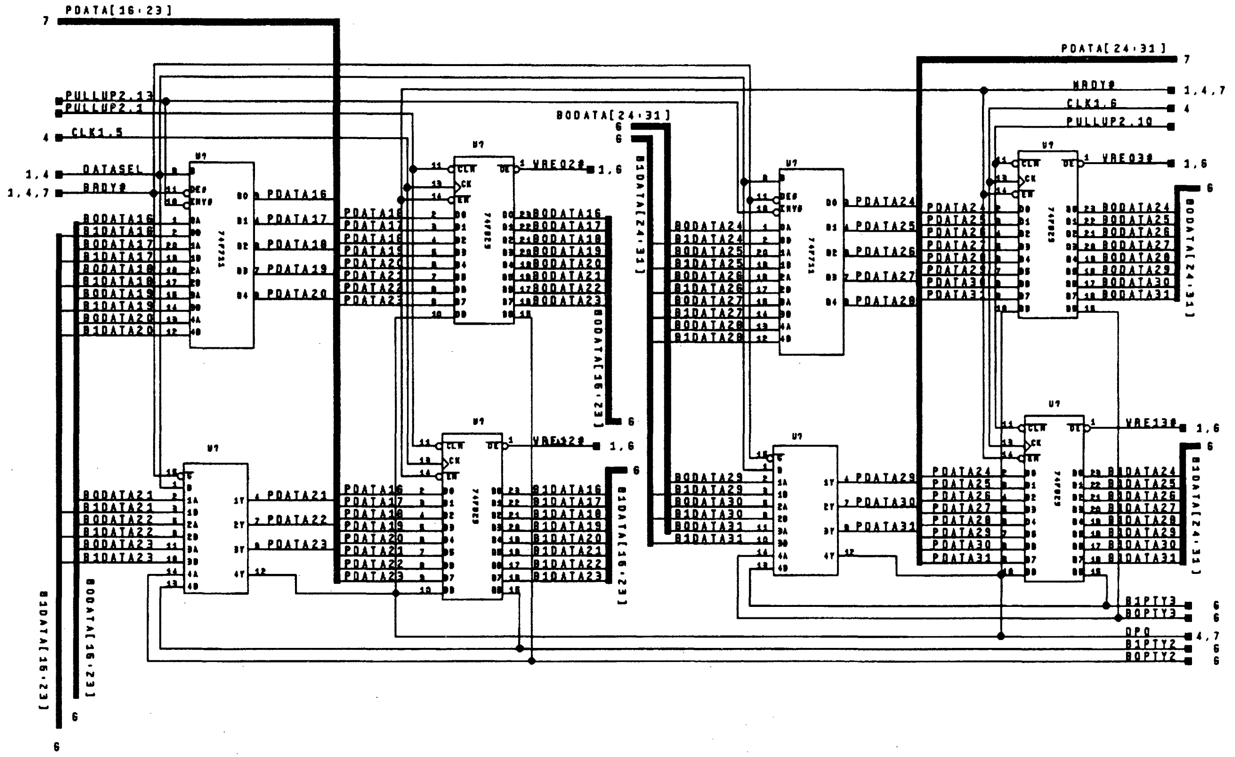
240799-61

Sheet 3 of 10



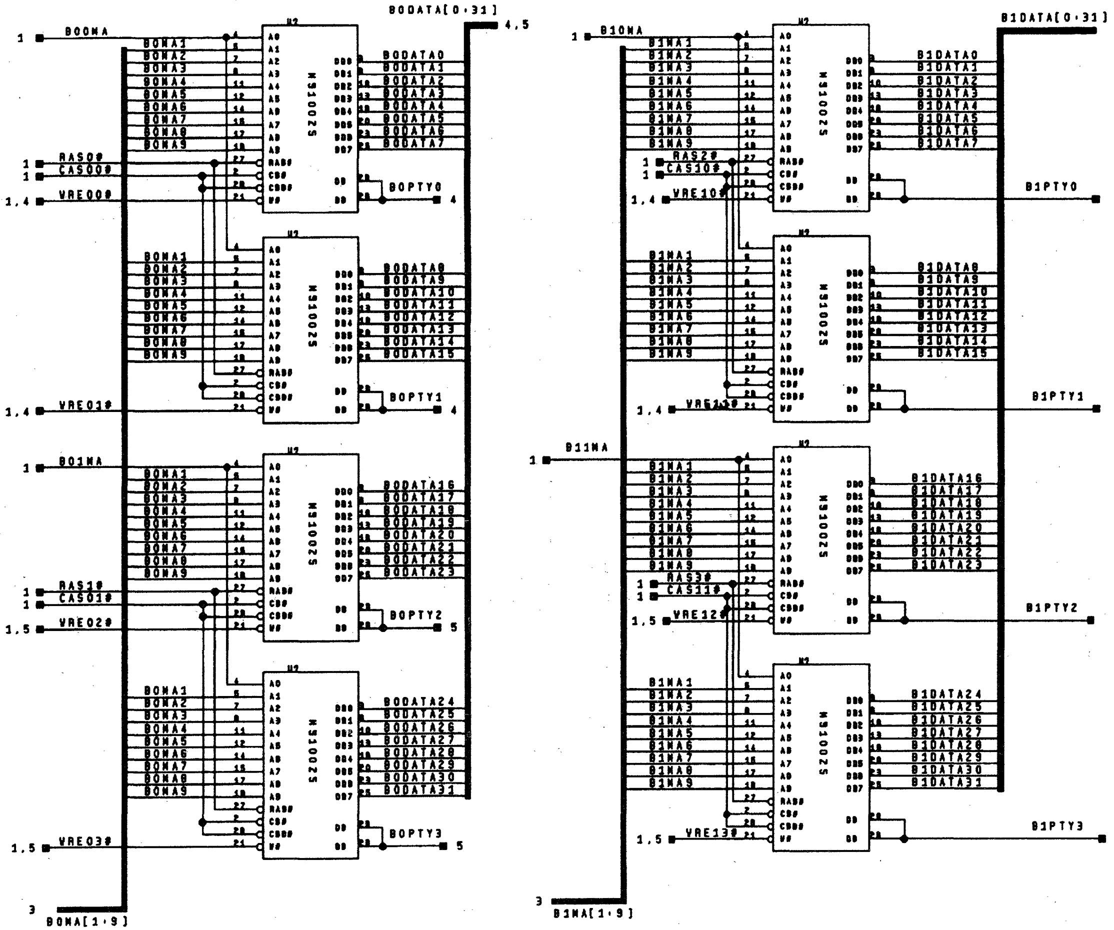
240799–62

Sheet 4 of 10



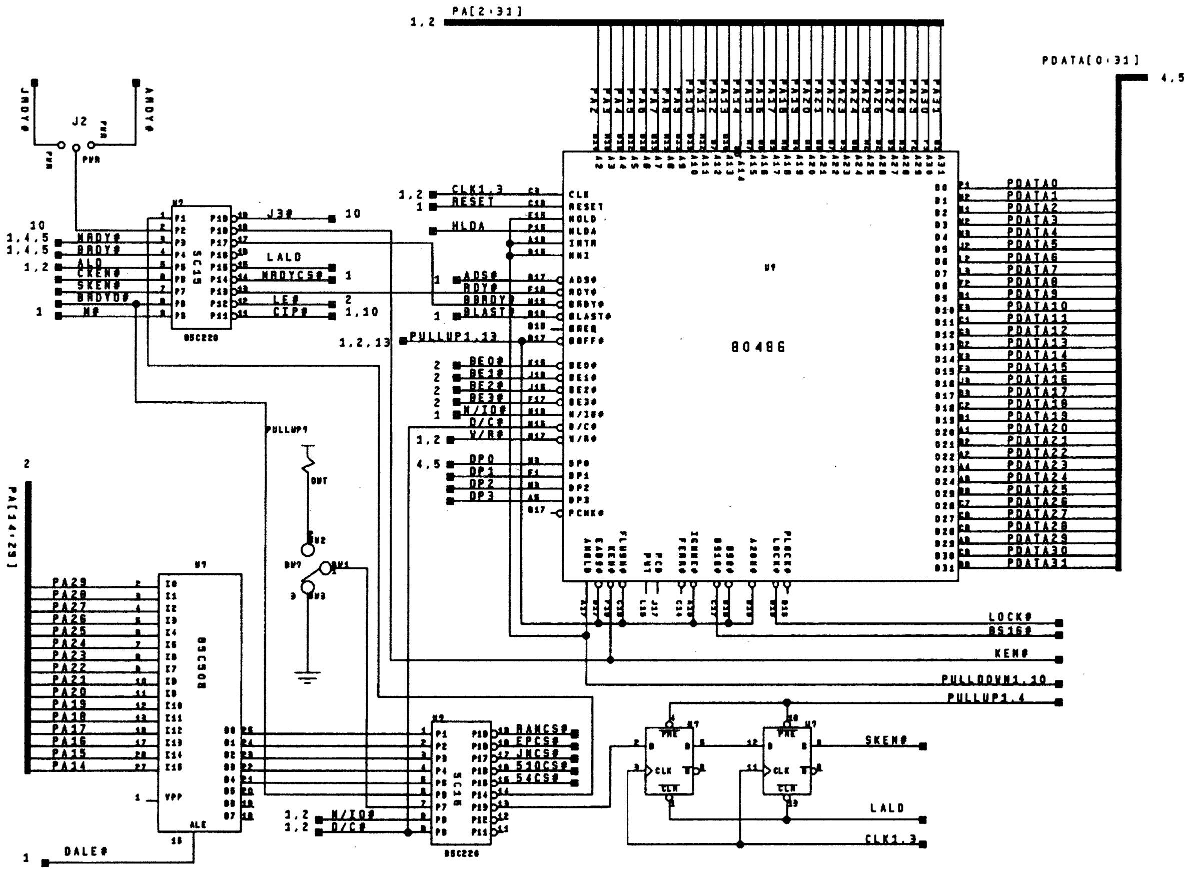
240799-63

Sheet 5 of 10



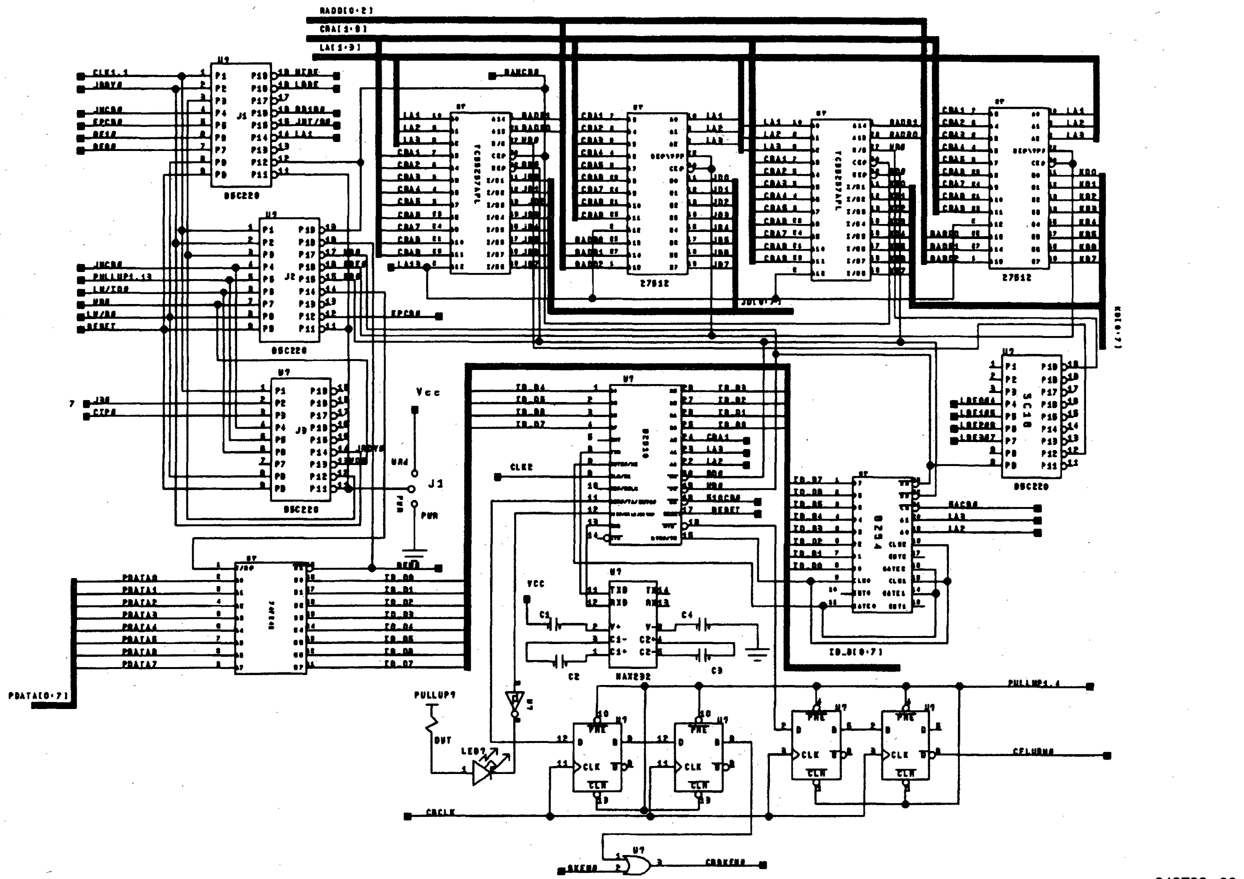
240799-64

Sheet 6 of 10



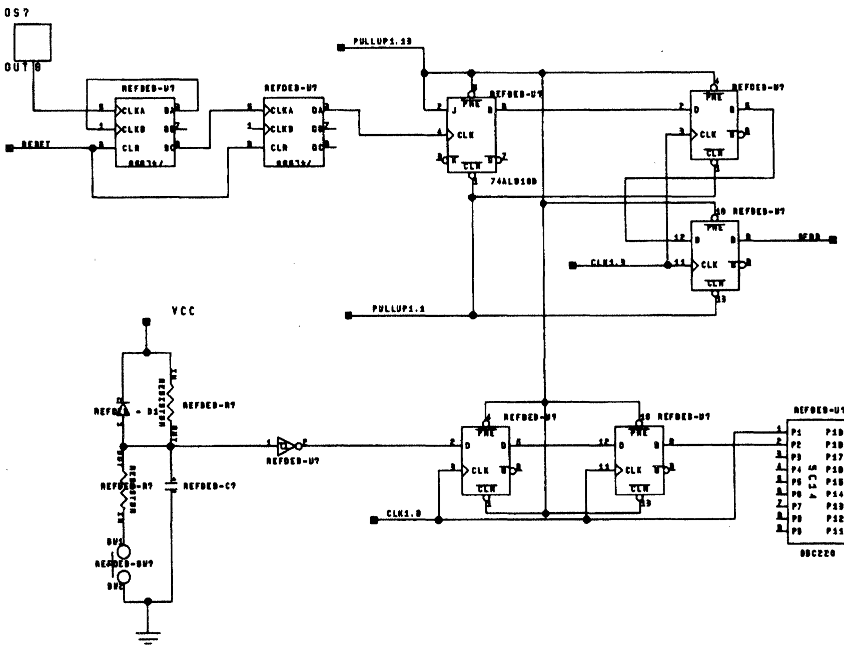
240799-65

Sheet 7 of 10



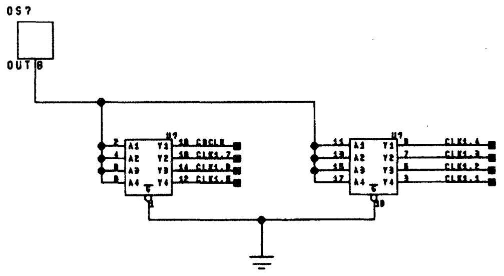
240799-66

Sheet 8 of 10



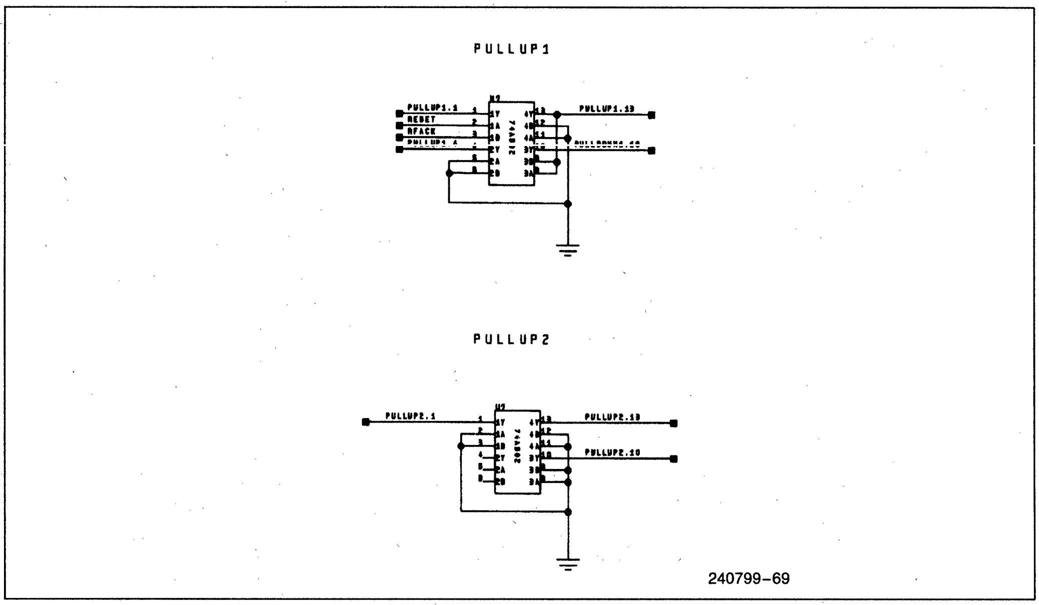
240799-67

Sheet 9 of 10

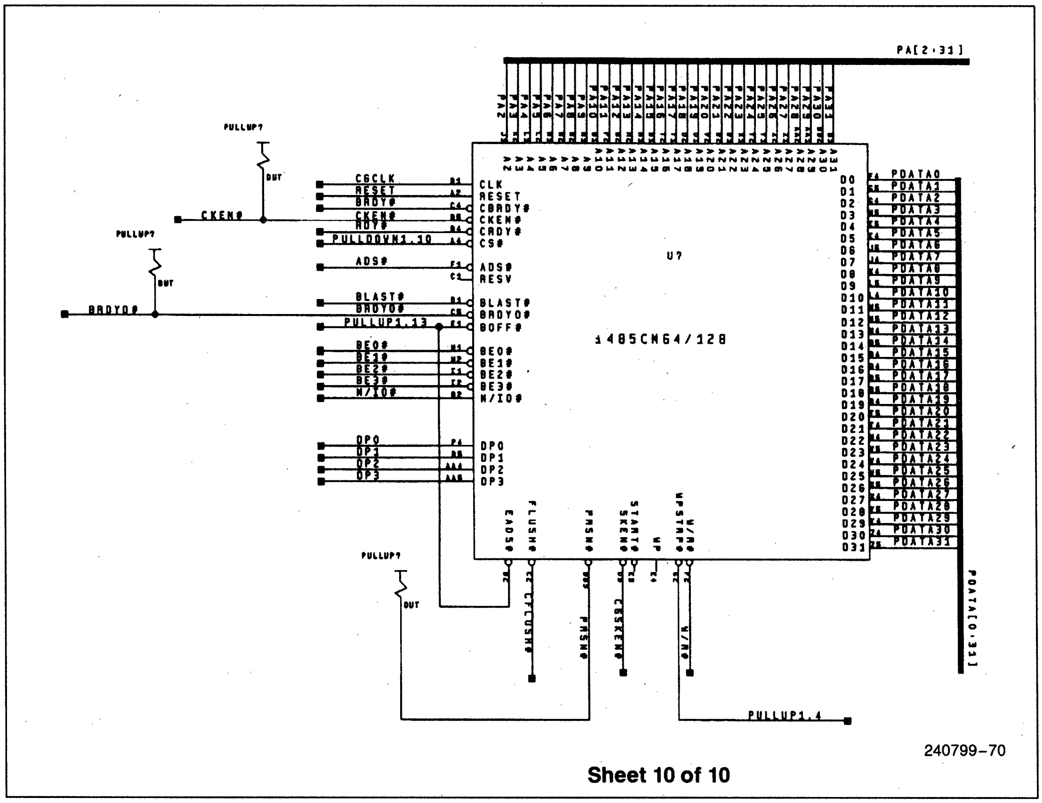


```
NOTE:
      CLK1.1    -> U10 PIN13, U13 PIN13, U12 PIN13, J1,J2,J3 PIN1
      CLK1.2    -> SC4 PIN1, SC8 PIN1, SC11 PIN1, SC2 PIN1
      CLK1.3    -> PROC. PINC3, SC1 PIN1, U8 PIN13, U87 PIN13, U83 PIN3&11
      CLK1.4    -> SC17 PIN1, SC7 PIN1, SC6 PIN1, SC3 PIN1
      CLK1.5    -> U30 PIN13, U25 PIN13, U29 PIN13, U21PIN13, U24 PIN13
      CLK1.6     -> U20 PIN13,U27PIN13,U33 PIN13, U85 PIN3&11, U86 PIN1
      CLK1.7    -> U45 PIN1, U46 PIN10, U47 PIN1, U49 PIN2, U56 PIN6, U57 PIN23
```

240799-68



240799-69



240799-70

Sheet 10 of 10