EV5 CPU Chip
Internal Specification

The EV5 CPU Chip is a high-performance, single-chip implementation of the Alpha Architecture.

Revision/Update Information: This is Revision 0.0 of this specification.

Product Manager: John Fortune, RICKS::FORTUNE
Engineering Manager: Paul Rubinfeld, ROCK::RUBINFELD

Semiconductor Engineering Group
Digital Equipment Corporation, Hudson, Massachusetts

This is copy number

February 1992

Contents

FIGURES

TABLES

Chapter 1

The Ibox

1.1 Overview

The basic IBOX operation is as follows.

In stage S-1 the I_IDX%IC_INDEX_H<12:2> is generated and provided to the ICache. The carry of the incrementer is passed to the S0 PC<42:13> adder.

In stage S0 the virtual ICache reads 4 instructions and sends them to the IB. Also in S0 the pc adder calculates the PC of the fetched instruction block and sends it to the PC silo which has analogous control with the IB. At the end of S0 the ICache hit starts, checking the TAG just read with the PC just calculated.

In S1, the IB selects one of the 2 Instruction buffers and forwards the data to the Slot logic if possible. The IB receives I_HIT%IC_HIT_1A_H, calculates I_IBS%IB_STALL_1B_H and informs the fetch logic. In parallel the Branch taken adders calculate the 4 possible branch path indexes, picking the correct one at the end of S1 to send to the ICache. The ITB checks all ICache fetches for access and generates ICache Miss PAs for the Scache. The Slot logic starts in the second half of S1. It calculates the mux selects to send each instruction to the correct Pipe.

In S2, the Slot logic drives the instructions to the correct pipes. If some instructions don't slot a stall is generated to the IB stage, and the remaining instructions will try again in the next cycle. At the end of S2 the register numbers are predecoded for register reads and dirty checks.

In S3, the dirty logic for the register file checks for conflicts between the current set of instruction and in previously issued instructions. Instruction Valid is forwarded to the appropriate pipes for issued instructions, S3_Freeze is sent to earlier stages of the IBOX pipe if some instruction failed to issue. Also in S3 the alternate PC for predicted branch is calculated (ie the path the branch predictor didn't take).

In S4, the PC miss predict compare is done.

In S5, the mispredicted PC is loaded into the fetch index to restart the pipe.

In S6 the only IBOX action is to pipe state.

In S7, the trap logic prioritizes trap conditions and generates the trap vector. The Exception PC is loaded.

Figure 1–1: Simple Block Diagram

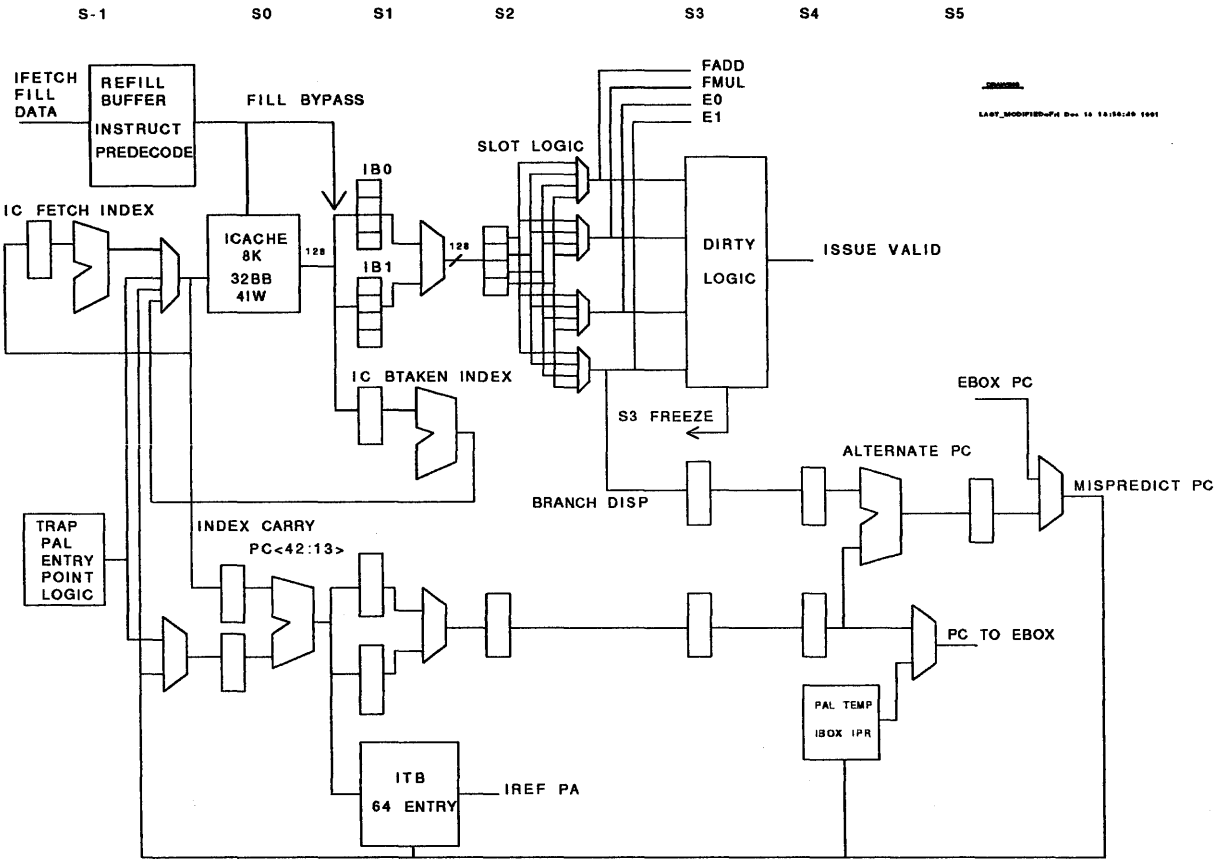
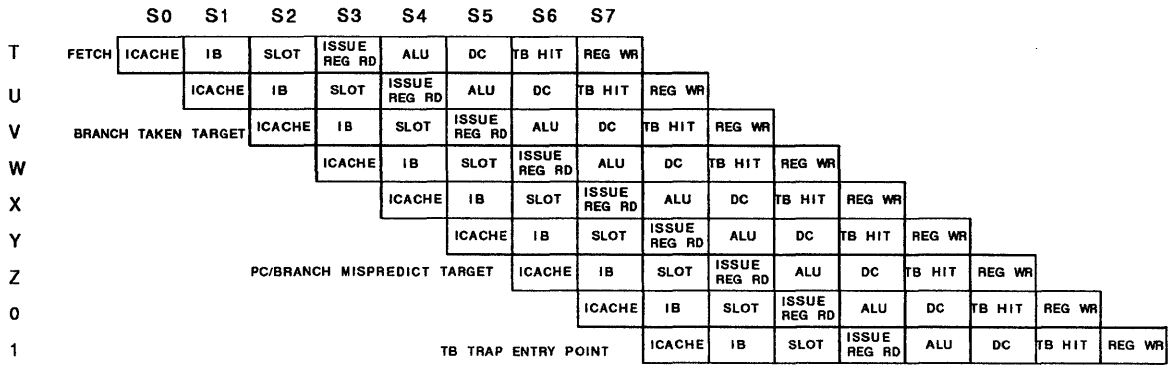


Figure 1–2: Waterfall



This water fall shows the location of the IBOX stages in the pipe. The fetch stream changes at 3 different times. The waterfall shows the cycles the fetch stream would be restarted if the named event happened in instruction T.

1. S2: branch predicted taken
2. S6: Branch or PC miss predict, ITB miss/ACV (this is the S5 trap)
3. S8: DTB Trap or any other error. (this is the S7 trap)

The EV5 Ibox contains the following functionality :

- An 8K byte direct mapped, 32 byte block, virtual ICache with 4 32 byte refill buffers for streaming prefetch. The ICache reads 4 instructions per cycle.
- The ICache Fetch logic including -
  - Fetch adder
  - Refill buffer adder
  - Prefetch adder
  - Refill buffer tags (4)
- A 64 entry, fully associative, full hint bit support, 7 bit ASN, ITB.
- A 2K*2 bit Branch History Table and the History Update logic.
- The Flow Prediction logic including -
  - Branch Predictor
  - Target Calculation logic
  - Return Prediction Stack
- All the PC logic including -
  - Fetch PC logic
  - PC silo
  - Execution PC logic
- An 8 instruction IB that holds 2 sets of 4 instruction ICache fetches.
- The Instruction Slot logic which includes -
  - functional unit slotting.
- The Instruction issue logic which includes -
  - The register scoreboard:
    - A 1 bit shift register per pipe per register.
    - The DCache miss bit, 1 bit per register.
  - The Bypass control for the EBOX and FBOX and the register file write control.
  - The register serialization checks.
  - The MB/STC/RC flag
- IPRs
  - Ibox Control
  - ITB Control
  - PAL Temps
- Pal Entry point logic

1.2 Functional Description

1.2.1 ICache

1.2.2 Instruction Fetch

The Instruction Fetch block is responsible for providing cache indexes, hit signals, prefetch addresses and fill data. The goal of the fetch logic is to provide a continuous supply of instructions to the Instruction Buffer (IB) pipe. Whenever the IB has an available set of instruction buffers the fetch logic attempts to provide the predicted instruction stream. If it is later determined that the stream was mispredicted the fetch logic receives the correct calculated branch target from the I_WPC section and resumes fetching from the ICache. If any other I-stream flow change is encountered, (exception, trap, interrupt, re-play, etc.) the fetch logic is provided the target index and fetching resumes from the ICache.

The fetch logic controls a four entry direct mapped Refill Buffer which is used to assist in streaming fetches which miss in the ICache. Data which is prefetched is held in the refill buffer until requested by the IB or overwritten by a new I-stream. The Refill Buffer consists of four 32 byte blocks and four associated tag entries. The 128 bytes are accessed as eight octaword buffers, with two sub-blocks per Refill Buffer Tag entry.

The instruction fetch logic deals primarily with with ICache indexes (i.e. PC<12:2>. There are three separate indexes which make up the instruction fetch logic; the ICache index (ic_index), the refill buffer index (rfb_index), and the prefetch index (pf_index). See Figure 1–9.

The instruction fetch logic is controlled by two sequencers; a fetch sequencer and a pre-fetch sequencer. The fetch sequencer is primarily responsible for the sequencing pertaining to calculating the ic_index and rfb_index, hitting in the ICache and refill buffer, I-stream flow changes, and recieving fill data. The pre-fetcher is primarily responsible for the sequencing pertaining to requesting data when accesses miss in both the ICache and refill buffer , pre-fetching I-stream data, and calculating the pf_index.

The instruction fetch logic is also responsible for reporting the tag compare information and generating ICache hit and Refill Buffer hit signals. The instruction fetch logic maintains the Refill Buffer Tag Store. The instruction fetch logic is responsible for providing the valid bits to the ICache. The fetch logic provides the two valid bits associated with each ICache Tag write. The ICache is responsible for clearing the valid bits.

Table 1–1: Instruction Fetch Logic Signal Interface

Signal	Description/Notes
Fetcher and ICache Signals	
I%J_IC_INDEX_ZB_H<12:4>	ICache Read/Write Index.
I%J_IC_INDEX_ZB_L<12:4>	ICache Read/Write Index.
I%J_VALID_ZB_H<1:0>	ICache Valid bits.
I%J_IC_CMD_A_H	Asserion indicates Write, de-assertion indicates read.

Table 1–1 (Cont.): Instruction Fetch Logic Signal Interface

Signal	Description/Notes
Fetcher and ICache Signals	
I%J_BYPASS_IC_A_H	Asserion indicates J%I_ISTR_DATA_0B_H is coming from the Refill Buffer, De-asserion indicates that J%I_ISTR_DATA_0B_H is coming from the ICache.
I%J_RFB_RD_IDX_1B_H<6:4>	Refill Buffer Data Read Index.
I%J_RFB_WR_IDX_A_H<2:0>	Refill Buffer Data Write Index, valid one cyle before the write.
I%J_RFB_WR_A_H	Refill Buffer Data Write Enable, valid one cyle before the write.
Fetcher and I_IBS Signals	
I_IBS%IB_STALL_1B_H	Assertion of this signal forces the Instruction Fetcher to stall the fetching of instructions.
I_IDX%IC_DATA_VALID_1A_H	Assertion indicates that the data on J%IC_DATA_0B_H is valid.
I_IDX%CURRENT_IDX_1A_H<3:2>	The longword address of the instruction fetch.
I_IDX%CURRENT_IDX_1A_H<0>	Assertion indicates that the instructions fetched are PAL instructions.
J%I_ISTR_DATA_0B_H<127:0>	This is the istream data which is sent to the IB.
Fetcher, I_WPC, and I_TRP Signals	
I_TRP%SEL_EXCEPTION_PC_A_H	Assertion indicates select I_TRP%EXCEPTION_PC_A_H<12:0>; de-assertion indicates select I_WPC%REPLAY_IDX_4A_H<12:4,0>, I_TRP%REPLAY_POS_4A_H<1:0>.
I_TRP%SEL_S6_REPLAY_A_H	Assertion indicates that I_WPC%REPLAY_IDX_6A_H<12:4,0> and I_TRP%REPLAY_POS_6A_H<1:0> should be selected. De-assertion indicates that I_WPC%BR_PC_MPRED_IDX_5A_H<12:0> should be selected.
I_WPC%REPLAY_IDX_4A_H<12:4,0>	4A replay index.
I_TRP%REPLAY_POS_4A_H<1:0>	4A replay position.
I_TRP%EXCEPTION_PC_A_H<12:0>	Exception pc.
I_WPC%BR_PC_MPRED_IDX_5A_H<12:0>	Branch, PC Mispredict index.
I_WPC%REPLAY_IDX_6A_H<12:4>	7A replay index.
I_TRP%REPLAY_POS_6A_H<1:0>	7A replay position.
I_TRP%SEL_EXC_EREPLAY_A_H	Assertion indicates that the I_TRP%SEL_EXCEPTION_PC_A_H signal is valid and the next fetch index should be either I_TRP%EXCEPTION_PC_A_H<12:0> or I_WPC%REPLAY_IDX_4A_H<12:4,0>, I_TRP%REPLAY_POS_4A_H<1:0>. It must not be asserted if I_TRP%SEL_BR_LREPLAY_A_H is asserted.
I_TRP%SEL_BR_LREPLAY_A_H	Assertion indicates that the I_TRP%SEL_S6_REPLAY_A_H signal is valid and the next fetch index should be either I_WPC%REPLAY_IDX_6A_H<12:4,0>, I_TRP%REPLAY_POS_6A_H<1:0> or I_WPC%BR_PC_MPRED_IDX_5A_H<12:0>.It must not be asserted if I_TRP%SEL_EXC_EREPLAY_A_H is asserted.
I_TRP%TRAP_POSTED_A_H	Assertion indicates that a trap has occurred. It must be asserted if either I_TRP%SEL_EXC_EREPLAY_A_H or I_TRP%SEL_BR_LREPLAY_A_H are asserted.

Table 1–1 (Cont.): Instruction Fetch Logic Signal Interface

Signal	Description/Notes
Fetcher and I_FPR Signals	
I_FPR%FLOW_CHANGE_1A_H	Assertion indicates that I_FPR%PREDICTED_IDX_1A_H<12:2> should be selected as the next fetch index. Must be conditioned with ICache Hit or Refill Buffer Hit.
I_FPR%PREDICTED_IDX_1A_H<12:2>	Predicted index (Branch Taken, CALL_PAL, or Stack).
Fetcher and MBOX Signals	
I%M_IREF_IDX_1B_H<6:5>	Index for the 1 of 4 MAF entries.
I%M_IREF_REQ_2B_H	Assertion indicates that the MBOX should begin a IREF arbitration using I%M_IREF_ADDR_2A_H<39:4> as the physical address.
I%M_IREF_ADDR_2A_H<39:4>	Physical address for IREF access.
Fetcher and CBOX Signals	
C%I_IFB_DATA_VALID_8B_H	Assertion indicates that the S%I_IFB_DATA_9B_H<127:0> will be valid in the following cycle.
C%I_IFB_INDEX_8B_H<2:0>	Indicates which one of 8 possible octawords is being returned on the S%I_IFB_DATA_9B_H<127:0>.
C%I_IFB_LAST_FILL_8B_H	Assertion indicates that this fill is the last of the two octawords associated with the C%I_IFB_INDEX_8B_H<2:0>.
S%I_IFB_DATA_9B_H<127:0>	SCache Fill data.

1.2.2.1 Instruction Fetch Flow

The instruction fetching begins with the longword (4 bytes) index (**I%J_IC_INDEX_ZB_H<12:2>**) required for ICache accesses. The **I%J_IC_INDEX_ZB_H** is set-up to the 0A rising edge of **K%CLOCK**. The the "next" ic_index can be sourced from 1 of 7 places from the Instruction Fetch section's (I_IDX) perspective. These sources are:

- **I_FPR%PREDICTED_IDX_1A_H<12:0>** - Predicted Index.
- **I_IDX_FIC%INC_INDEX_ZA_H<12:0>** - Incremented Index.
- **I_IDX_FIC%REC_INDEX_0A_H<12:0>** - Recirculated Index.
- **I_IDX_FIC%FILL_INDEX_1A_H<12:0>** - Fill Index.
- **I_WPC%REPLAY_IDX_4A_H<12:0>** - 4A Replay Index.
- **I_TRP%EXCEPTION_PC_A_H<12:0>** - Exception Index.
- **I_WPC%BR_PC_MPRED_IDX_5A_H<12:0>** - 5A Branch, PC Mispredict Index.
- **I_WPC%REPLAY_IDX_6A_H<12:0>** - 6A Replay Index.

The priority of selection of the "next" index is as follows:

- The Exception, 4A, 5A, and 6A Indexes have the highest priority in the selection of a "next" index. These indexes are selected by the assertion/de-assertion of **I_TRP%SEL_EXCEPTION_PC_A_H, I_TRP%SEL_EXC_EREPLAY_A_H, I_TRP%SEL_S6_REPLAY_A_H, I_TRP%SEL_BR_LREPLAY_A_H,** and **I_TRP%TRAP_POSTED_A_H.** The trap logic ensures that only one of these indexes can be selected each cycle.
- **I_IBS%IB_STALL_A_H** will force the ic_index to recirculate while asserted.
- **I_FPR%FLOW_CHANGE_1A_H** conditioned with **I_IDX%ISTR_DATA_VALID_1A_H** and NOT(**I_IDX%FETCH_BUBBLE**) will cause selection of the predicted index.
- If none of the above conditions exist then the "next" index is selected by the sequencer to either increment, recirculate, or return to the index which missed and wait for the fill.

The refill buffer index and the prefetch index are in sync with the ic_index when the ICache accesses hit in the ICache or when the I-stream flow is re-directed.

See Figure 1–8 and Figure 1–9 for the muxing heirarchy for the instruction fetch index.

When fetching contiguous ICache blocks which hit in the cache, the **I%J_IC_INDEX_ZB_H** is incremented in 16 byte steps each cycle. (I.E. bit 4 is incremented) The **I%J_IC_INDEX_ZB_H** is incremented on aligned octaword boundaries. (I.E. bits (3:2) are always zero when the index is sourced from the incrementer.)

When the Flow Prediction logic (I_FPR) encounters a valid I-stream flow change instruction, **I_FPR%FLOW_CHANGE_1A_H** is asserted. This will force the mux for **I%J_IC_INDEX_ZB_H** to select **I_FPR%PREDICTED_IDX_1A_H** as the next index. The sequencer will assert **I_IDX%FETCH_BUBBLE** to indicate to the other sections that a bubble is present. The rfb_index and pf_index are re-sync'ed with the ic_index and fetching continues by looking into the ICache first.

When **I_IBS%IB_STALL_1B_H** asserts the **I%J_IC_INDEX_ZB_H** must be held. This is accomplished by re-circulating the previous index. The ICache is responsible for holding the data to be stalled. If the index that is stalled is index(N), the Icache will be holding data(N-1).

When the Trap logic (I_TRP) encounters a I-stream flow change the selection of the next index is determined by **I_TRP%SEL_EXCEPTION_PC_A_H, I_TRP%SEL_PC_7A_H,** and **I_TRP%SEL_PC_5A_H** as described in the table below. The sequencer will assert **I_IDX%FETCH_BUBBLE** to indicate to the other sections that a bubble is present.

Figure 1–3: Fetch Index Mux Selects for Trap, Exception, Replay(4A,6A)

sel_exc_ereplay	sel_br_lreplay	sel_exception	sel_s6_replay	next index	
0	0	X	X	determined by fetch seq.	** NOTE sel_exc sel_br_ guarente to be m trap_po be asse of the a are ass
1	0	0	X	i_wpc%replay_idx_4a_h	
1	0	1	X	i_trp%exception_pc_a_h	
0	1	X	0	i_wpc%replay_idx_4a_h	
0	1	X	1	i_wpc%replay_idx_6a_h	

When there is an ICache miss the **I%J_IC_INDEX_ZB_H** is returned to the index which missed. The index is then held until the ICache is filled from the refill buffer. The fill may come in the next cycle if the data is presently in the refill buffer or it will arrive sometime later depending on where in the memeory system the data presently resides. When the ICache and IB are filled the **I%J_IC_INDEX_ZB_H** can be incremented.

When there is an ICache miss, then the refill buffer hit signal is checked. If it indicates a refill buffer hit, the data is provided by the refill buffer and written to the ICache and IB in the following cycle. While accesses hit in the refill buffer data can be sent to the ICache and IB in consecutive cycles. While hitting in the refill buffer the **I%J_IC_INDEX_ZB_H** and **I%J_RFB_RD_IDX_1B_H** are no longer in sync. The **I%J_RFB_RD_IDX_1B_H** is ahead of the **I%J_IC_INDEX_ZB_H** inorder to continue streaming data to the ICache and IB. After hitting in the refill buffer, the fetcher continues to fetch instructions from the refill buffer until the access misses in the refill buffer.

Figure 1–4: IC_Index and RFB_Index on IC_Miss and RFB_Hit

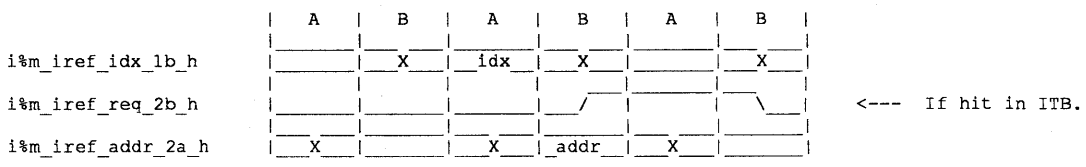
ic_index	0	1	2	1	2	3	4	
rfb_index	0	1	2	3	4			
ic_hit			0	X	D	D		
rfb_hit			D	1	2	3	4	
istr_data			0	D	1	2	3	4

** Note **
X - Miss
D - Don't Care

Once the fetcher misses in the refill buffer after missing in the Icache or when there is a miss in both the ICache and the refill buffer, the pre-fetcher attempts to make a request to the MBOX to get the data from either the SCache, BCache, or memory sub-system. Conditions which would prevent a request from being sent are if the number of outstanding requests is greater than 2, or if the refill buffer entry you wish to use is pending or if you are in the shadow of a predicted pc while in PALmode. If either of these conditions are true the request will not be sent until they are no longer true. The ic_index and the rfb_index of the block that missed is held while the request is being serviced.

The interface with the MBOX pertaining to the requesting of fill data consists of 3 signals/busses: **I%M_IREF_IDX_1B_H<6:5>, I%M_IREF_REQ_2B_H,I%M_IREF_ADDR_2A_H<39:4>**. At the time which the IBOX sends **I%M_IREF_IDX_1B_H<6:5>** it is not known whether or not the access hits in the IBOX Translation Buffer (ITB). Therefore the protocall is that the IBOX sends **I%M_IREF_IDX_1B_H<6:5>** indicating a request to 1 of the 4 Miss Address File (MAF) entries in the MBOX. After it is determined whether you hit in the ITB in stage 2A, the IBOX will conditionally send **I%M_IREF_REQ_2B_H** in stage 2B if the access hit in the ITB. The physical address (**I%M_IREF_ADDR_2A_H<39:4>** is read from the ITB in stage 2A. If the access is not to virtual address space than there will be no translation of the address and the **I%M_IREF_REQ_2B_H** will be sent without regard to the ITB.

Figure 1–5: Ibox requests to MBOX



The pre-fetcher will then send requests to pre-fetch the next blocks of instructions. The pre-fetcher can have a maximum of 3 requests outstanding at any time. this is due to the fact that the refill buffer is direct mapped and the pre-fetcher can not attempt to pre-fetch a block for a refill buffer entry which has been requested but has not yet been returned. Therefore the pre-fetcher monitors the ic_index and ensures that the pf_index does not step on the entries which the fetcher is waiting for.

One cycle before the fill data is driven to the ICache on **S%I_IFB_DATA_9B_H<127:0>**, the CBOX sends **C%I_IFB_DATA_VALID_8A_H** indicating that valid fill data will coming. Along with **C%I_IFB_DATA_VALID_8B_H** the CBOX sends a 3 bit fill index (**C%I_IFB_INDEX_8B_H<2:0>**) to indicate which of a possible 8 octawords is being sent and **C%I_IFB_LAST_FILL_8B_H** to indicate if this fill is the last of the two octawords associated with the **C%I_IFB_INDEX_8B_H<2:0>**. This valid signal is used to provide the write strobe to the Refill Buffer Data Latches as well as set the valid bit for the refill buffer entry. All fills are written into the refill buffer. If the fill is the octaword which was requested (the demand fetch data) the fill data is written thru the refill buffer and directly into the ICache and IB in parallel. At any point which the data that is being filled is not the demand fetch data, the fill data is written only into the Refill Buffer. The ICache index and Refill Buffer Index are both incremented after the demand fetch data is returned. If the incremented index hits in the Refill Buffer the data is sent to the IB in the next cycle. If the index does not hit in either the ICache or the Refill Buffer, the fetcher recirculates the demand index until the next octaword of fill data is returned.

Figure 1–6: Signal Protocall for Fills

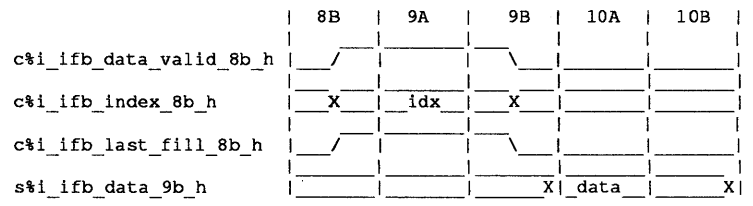


Figure 1–7: IFB Fills with RFB Hits

	X		Y		Z		0		1		2		3	
	A	B	A	B	A	B	A	B	A	B	A	B	A	B
ic_index	.		.		.	2	.	0	.	1	.	2	.	3
rfb_index	.	0	.	0	.	1	.	2	.	3	.		.	
ic_hit	X	.	X	.	X	.	X	.	X	.	X	.	X	.
rfb_hit	X	.	X	.	X	.	0	.	1	.	2	.	3	.
ifb_valid	1	.	0	.	2	.	3	.	.		.		.	
rfb_valid	.		.		0	.	1	.	2	.	3	.	.	
ifb_data	.		.	1	.	0	.	2	.	3	.		.	
istr_data	.		.		.		.	0	.	1	.	2	.	3

** Note **
X - Miss
D - Don't Care

In order to try to prevent prefetching data which is already in the ICache, an effort will be made to to look in the ICache while waiting for return data to see if there is a hit. If there is a hit, prefetching will stop, the indexes will be re-sync'ed and accesses will be made to the ICache. Outstanding prefetches are required to complete and will be written into the refill buffer. If there is not a hit in the ICache prefetching will continue. The only time that the fetcher will be looking at ICache hit is when the sequencer is fetching out of the ICache or when the fetcher is waiting for data to be returned.

When a condition which alters the flow of contiguous block accesses occurs each of the indexes are re-sync'ed to the target index and fetching begins by looking into the ICache first. Outstanding requests will complete and will be written into the refill buffer.

When the prefetcher is stalled due to the fact that there are 3 outstanding fill requests, the next request can not be made until both octawords associated with a previous request have been returned.

ICache fills and their associated tags are not dependent upon parity checks. The parity information is stored and a trap is taken later in the pipe.

1.2.2.2 Prefetch Addressing

The Prefetch Index (pf_index) is calculated in S1B. This is a stage later than the icache fetch and refill buffer indexes. The pf_index stays in sync with the ic_index until there is an ICache miss. If there is an ICache miss and a refill buffer hit, the pf_index remains in sync with the rfb_index. At the point which there is a refill buffer miss after an ICache miss, the pf_index begins to increment independently. The pf_index increments in 32 byte steps while in prefetch mode. (I.E. increment bit 5)

The prefetcher is the source for requests to the MBOX. Accesses which miss in both the ICache and refill buffer are requested to be filled from either the SCache, BCache, or memory sub-system. The MBOX permits no more than 3 outstanding requests at a time. The pf_index is held once the maximum number of outstanding requests have been sent. Once both octawords of a request have been returned, another request can be made and the pf_index is incremented. Requests can not be made for indexes corresponding to refill buffer entries which are pending due to a previous request. The prefetcher must wait for all 32 bytes cooresponding to the pending entry to be returned before the new request can be made.

The pf_index I_IDX%IREF_IDX_1B_H<12:4> is merged with the upper bits of the translated PC (I_ITB%PA_1B_H<42:13>) to form the full physical address to be sent to the MBOX for memory requests. (I%IREF_ADDR_2A_H<42:4>)

The pf_index is re-sync'ed with the ic_index on a change of the instruction flow.

The prefetcher does not prefetch across page boundaries.

1.2.2.3 I-Cache Hit Logic

The ICache tags and refill buffer tags are looked up and compared to the PC in parallel. If there is an ICache hit, the ICache index is incremented and fetching continues. The ICache tags are available late in S0B. In the remaining S0B phase the bit-by-bit XOR is calculated, comparing the ICache tag to the PC. The ASN and ASM for the entry are contained in the tag, and is compared to the ASN of the process. In early S1A, the results of all the individual compares is determined, the ASN compare being conditioned by the ASM bit of the entry not being set. Late in S1A the signal I_HIT%IC_HIT_1A_H is valid. The ic_hit signal can be forced deasserted or asserted either thru IPR control or as necessary during normal operation.

In order to more accurately store the valid bits for each block, if the tag of an access matches but the valid bit is not set, the tow valid bits of the entry are copied into the refill buffer tag so that when the data, tag, and valid bits are written into the ICache the valid bit cooresponding to the "other" half of the block will be preserved.

During the interval where the instruction fetcher is waiting for data to be returned, the ICache Tag is checked and if the index hits, prefetching is suspended.

1.2.2.4 Refill Buffer Hit Logic

The Refill Buffer Tags are available late in S0B. In the remaining S0B phase the bit-by-bit XOR is calculated, comparing the Refill Buffer Tag to the PC. The ASN and ASM for the entry are contained in the tag, and is compared to the ASN of the process. In early S1A, the results of all the individual compares is determined, the ASN compare being conditioned by the ASM bit of the entry not being set. Late in S1A the signal **I_HIT%RFB_HIT_1A_H** is valid. The rfb_hit signal can be forced deasserted or asserted either thru IPR control or as necessary during normal operation.

The Refill Buffer Tag is written in the S2A following a refill buffer miss which requires a request to be made for a fill from the MBOX. The refill buffer tag section contains two valid bits associated with each of the four tag entries. In the case of an ICache miss where the ICache tag matches but is invalid the valid bits are copied from the ICache and are set as each of the two 16 byte blocks are returned. In the case where the ICache tag does not match, both valid bits are cleared and are then set as each of the two 16 byte blocks are returned.

The Refill Buffer Hit logic is used to identify indexes which were previously requested but have not yet been returned. Short forward branches in blocks which miss in the cache are an example of this. In these cases, **I_HIT%RFB_TAG_MAT_1A_H** is asserted and a second request is not made.

Each of the four refill buffer entries has a pending bit associated with it. The pending bit is set when the request is sent to the MBOX. The pending bit cleared when all 32 bytes associated with the particular refill buffer entry have been returned. No new requests can be made for an index which is mapped to a refill buffer entry which is pending.

Figure 1–8: ICache Index Mux

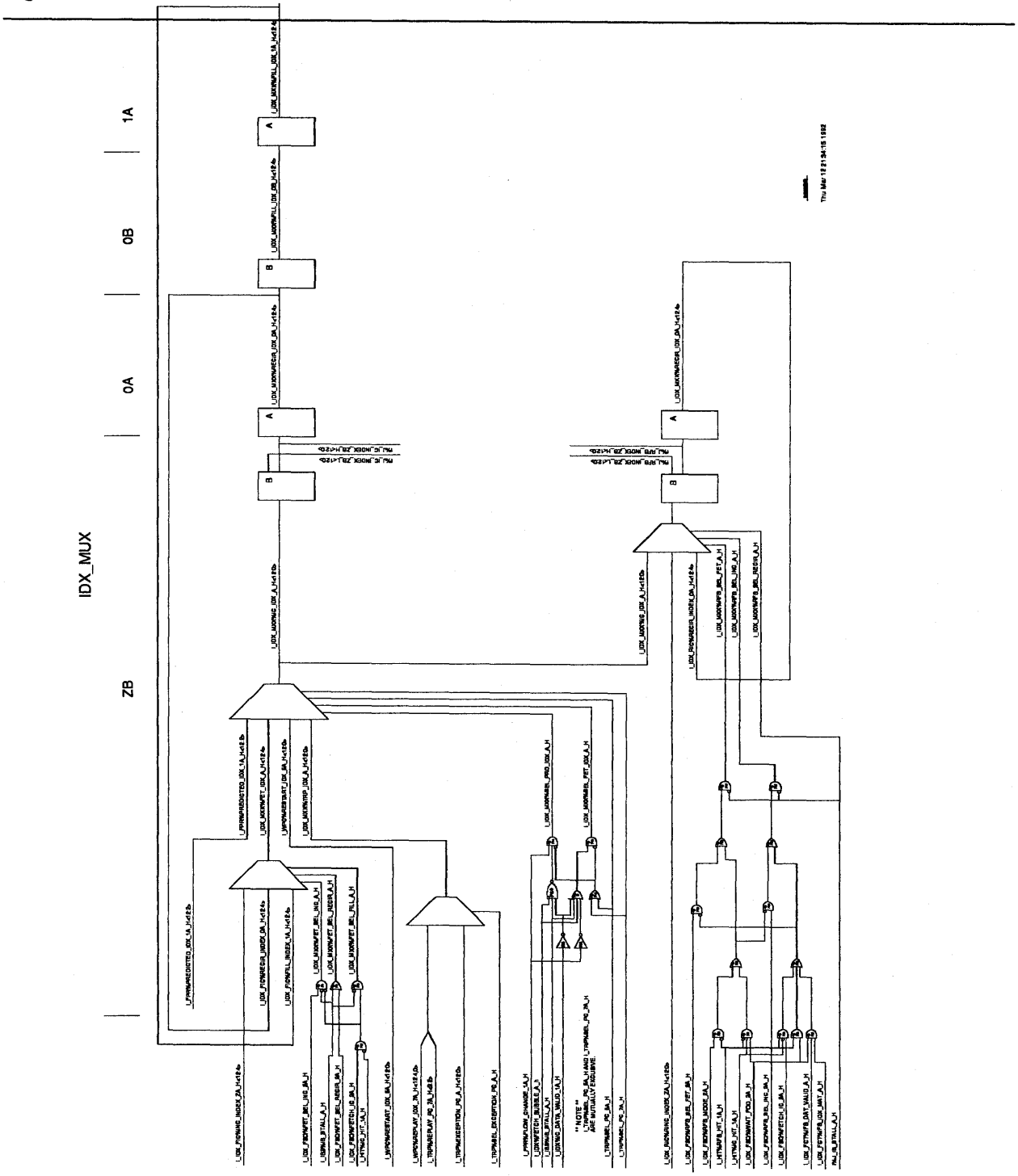


Figure 1–9: Fetch/Prefetch Logic

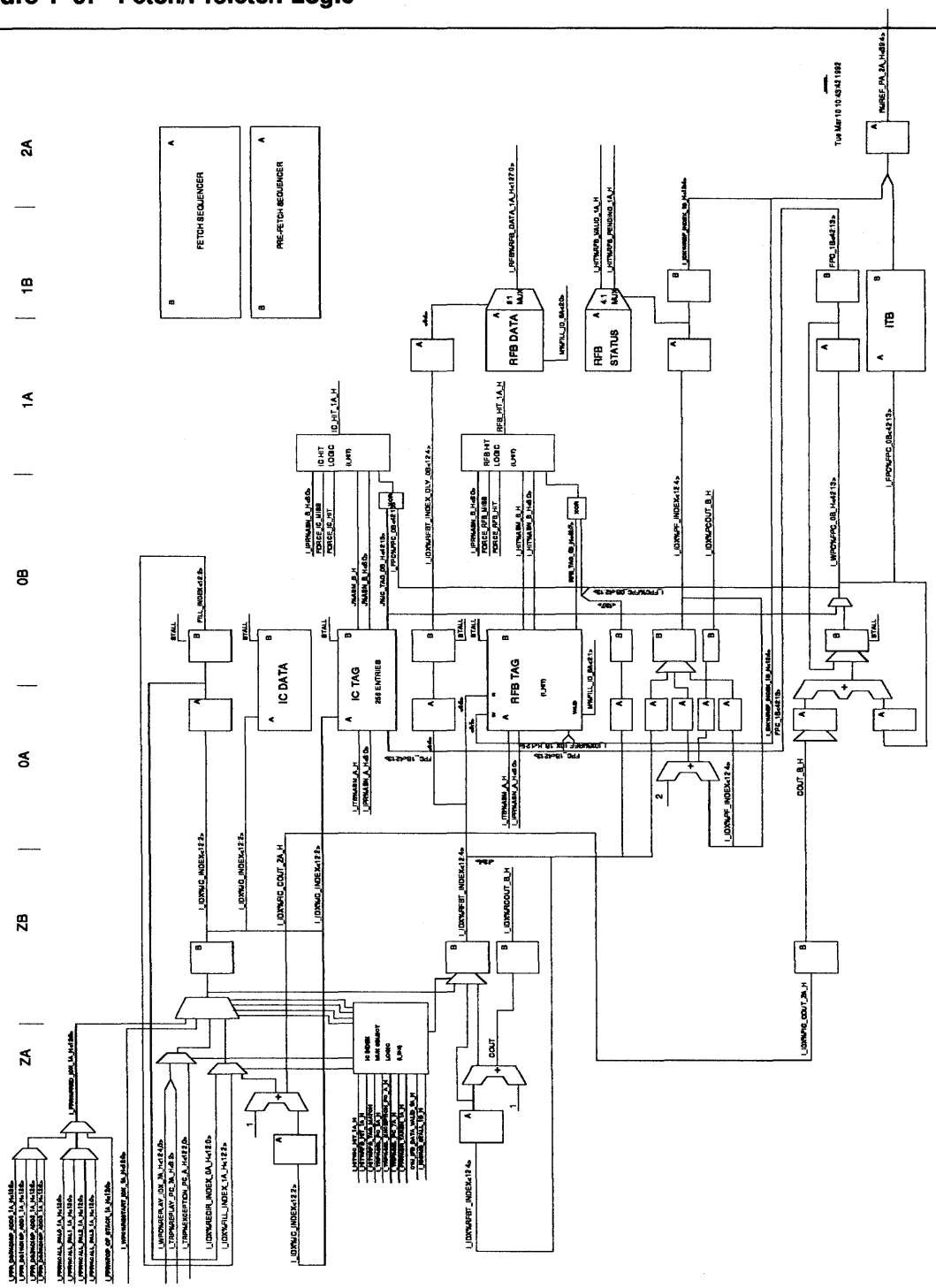


Figure 1–10: IBOX FETCHER SEQUENCER

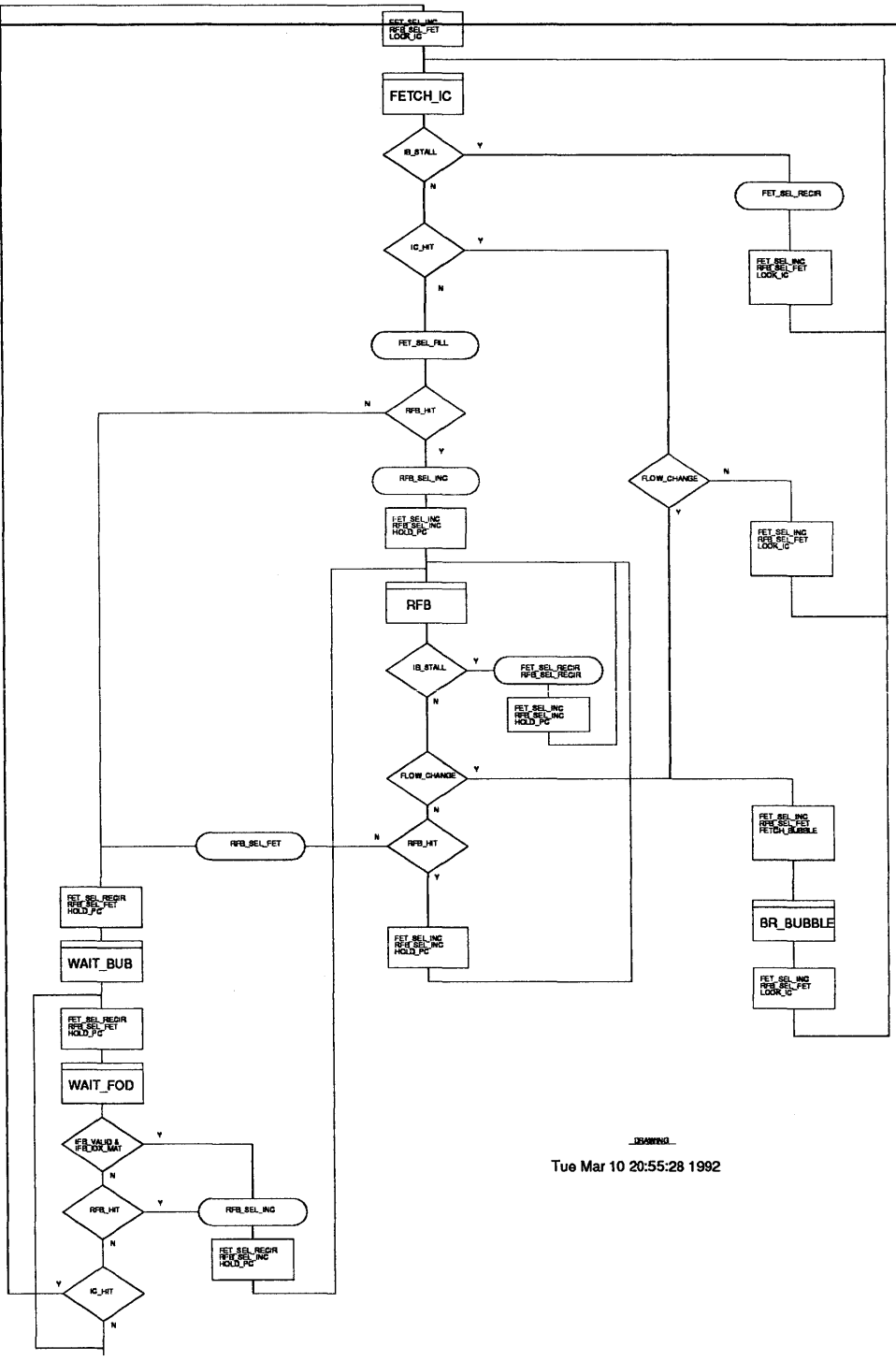
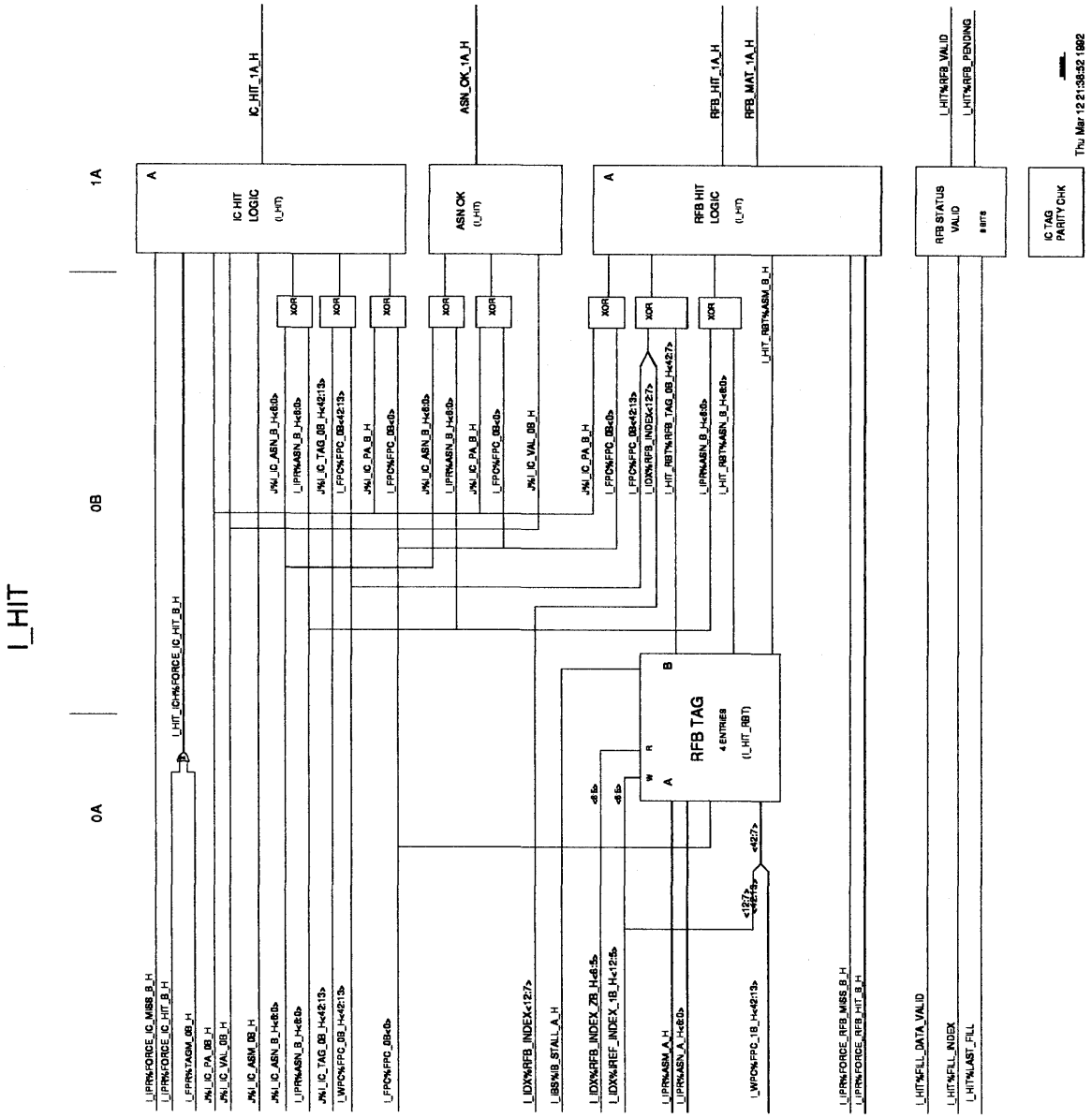


Figure 1–11: IBOX HIT Logic

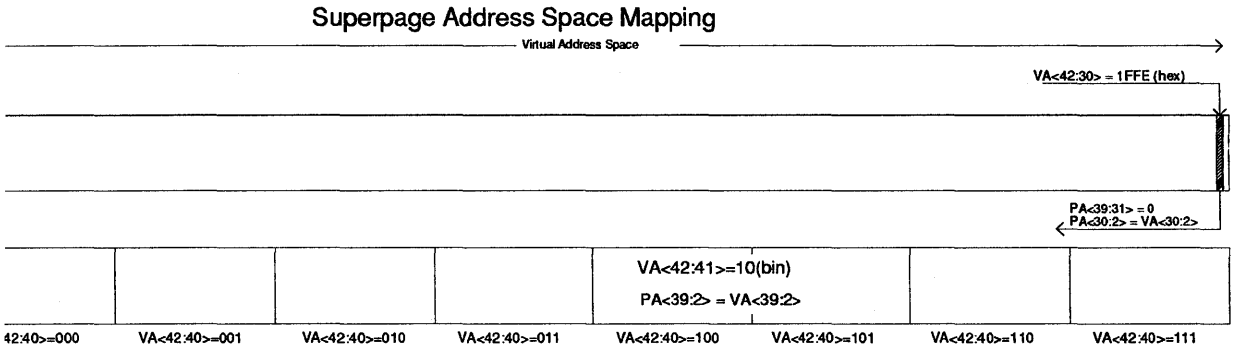


1.2.3 ITB

The Ibox contains a 64 entry fully associative translation buffer (ITB). The ITB contains instruction stream address translations and protection information for the referenced pages. The ITB supports the granularity hint option to map, under software control, either 1, 8, 64, or 512 physically/virtually contiguous 8 Kbyte pages with a single ITB entry, as defined by the ALPHA SRM. The ITB is maintained by PALcode, and the ITB can be updated only while in PALmode. PALcode is responsible for ensuring that a particular virtual address is never mapped to more than one ITB entry. The ITB is accessed using a not-last-used (NLU) algorithm, when written and when under IPR access control. The PC is not translated and protection is not checked while in PALmode.

The ITB supports the translation extention referred to as a super page. The mappings for superpages provide virtual to physical address translation for two specific regions of the virtual address space. The first superpage mapping is defined by the virtual address (PC) bits [42:41] = 10 (BIN). In this mode, the virtual address bits [39:2] are directly mapped to the physical address bits [39:2]. The second superpage mapping is defined by the virtual address (PC) bits [42:30] = 1FFE (HEX). In this mode, the physical address bits [39:31] = 0 and the virtual address bits [30:2] are directly mapped to the physical address bits [30:2]. Superpage translation is allowed only in kernel mode. If a superpage translation is attempted while not in kernel mode, an access violation fault will occur. This is accomplished by forcing the KRE bit to "1" and forcing the URE, SRE, and ERE bits to "0" on a superpage translation. Supaer page translation is performed only if the super page enable bit (SPE) is set in the ICSR IPR. See Section 1.2.10.17.

Figure 1–12: Superpage



All 64 enties of the ITB supprot each of the four page size options defined by the granularity hint (GH) bits, whioch are logically contained in the PTE. The two GH bits are decoded at the time when the PTE is written. The decoded GH bits are then used in the writting of the TAG portion of the entry to selectively determine the size of the entries page and therfore define which bits of the virtual address are compared for translation and which bits of the address are translated. The GH bits anable the comparision and translation of bits [21:13] of the virtual address.

Table 1–2: Granularity Hint Bit Mapping

GH<1>	GH<0>	Page Size	Physical Address of Page	Address within Page
0	0	8K bytes	PA<39:13>	PA<12:0>
0	1	64K bytes	PA<39:16>	PA<15:0>
1	0	512K bytes	PA<39:19>	PA<18:0>
1	1	4096K bytes	PA<39:22>	PA<21:0>

The ITB supports a seven bit Address Space Number (ASN) and single bit Address Space Match (ASM) for address comparisons. The ASN of the tag entry is compared to the ASN of the current process only if the ASM bit of the entry is not set. If the ASM bit of the entry is set, the ITB entry will match all ASN's. The comparision of the process ASN and the ASN of the entries is logically performed one cycle before the address comparision. This is done to reduce the load on the address match wire. The ASM of the entry is stored with the TAG portion of the entry, while the ASM is logically stored as part of the PTE.

There is one valid bit associated with each ITB entry and is determined by the associated MTPR instructions. (TBIAP,TBIA,TBIS,TBISI)

The ITB supports writes to the architectually defined TBIAP register by means of PALcode. PALcode should perform a MTPR to the ITB_IASM IPR. This write will have the effect of invalidating all ITB entries which the ASM bit of the PTE is not set.

The ITB supports writes to the architectually defined TBIA register by means of PALcode. PALcode should perform a MTPR to the ITB_ZAP IPR. This write will have the effect of invalidating all ITB entries, and resetting the NLU pointer to its initial state.

The ITB supports writes to the architectually defined TBIS register by means of PALcode. PALcode should perform a MTPR to the ITB_IS IPR. PALcode must ensure that the VA to be invalidated is present on **E%PC_3B_H<63:0>**.

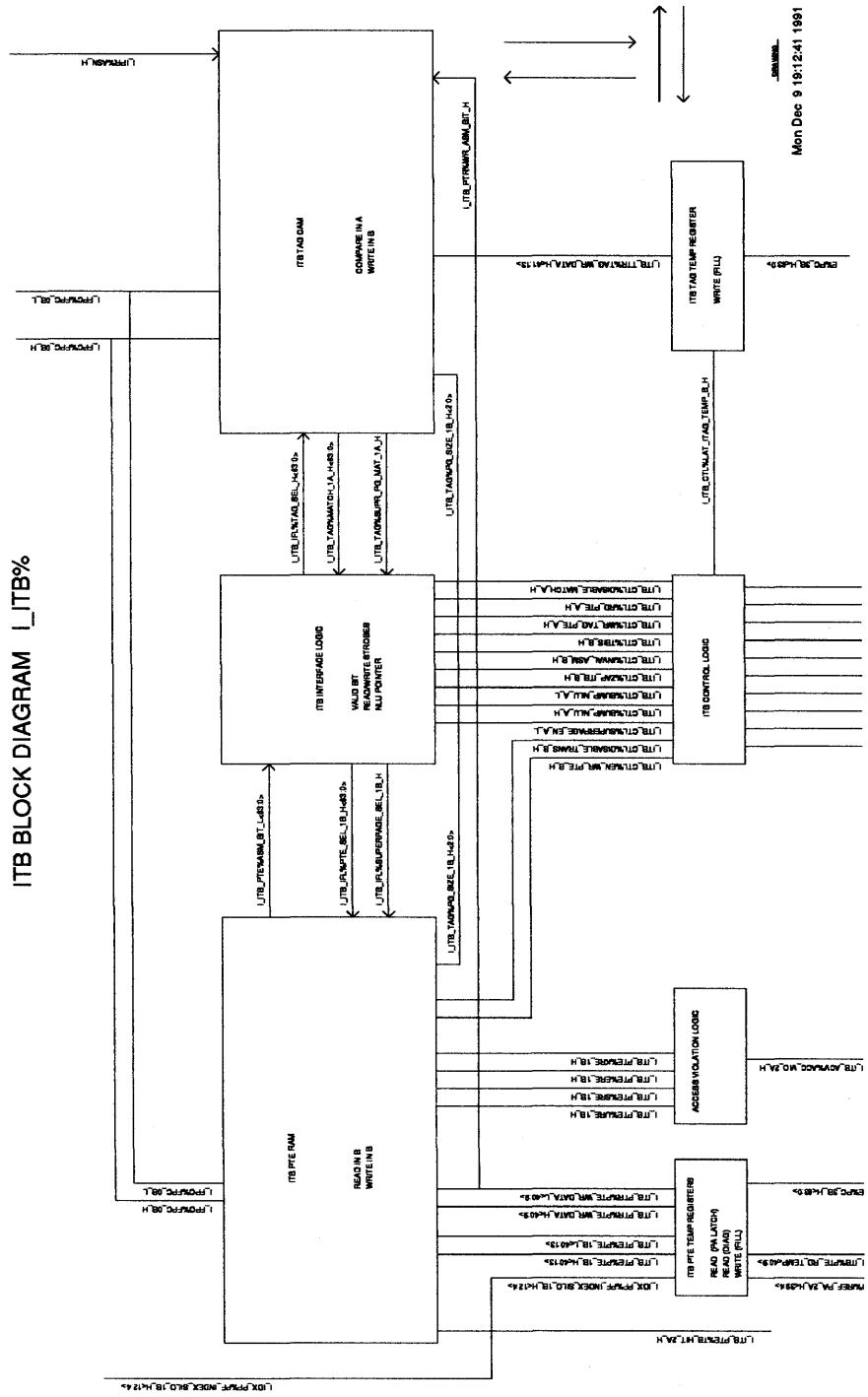
The ITB supports writes to the architectually defined TBISI register by means of PALcode. PALcode should perform a MTPR to the ITB_IS IPR. PALcode must ensure that the VA to be invalidated is present on **E%PC_3B_H<63:0>**.

When in non-PALmode, the ITB is looked up every cycle in stage 1 of the pipe. The ITB reference is performed on **I_WPC%FPC_0B_H<42:13>** (the virtual PC in non-PALmode), which must be set-up to the 1A edge of CLK. During the 1A phase the virtual address of the reference is compared to all the cached virtual addresses. If the Page Table Entry (PTE) associated with the virtual address is in the ITB, then the Page Frame Number (PFN) of the PTE is driven out of the ITB in 1B on **I_ITB%PA_1B_H<42:13>**. The lower bits of the address (PC), **I_IDX%IREF_IDX_1B_H<12:4>**, are not translated and are merged with the translated address to make up the full physical address. The physical address is latched in 1A and driven to the MBOX as **I%IREF_ADDR_2A_H<42:4>**. The protection bits for the page associated with the PTE are checked with the process privileges during stage 2 and produce **I_ITB%ACC_VIO_2A_H**. **I_ITB%ITB_HIT_2A_H** is determined during stage 2 as well.

Address translation is not performed if the address is physical. This is determined by bit [0] of the address; if set, the address is physical (I.E. PAL-mode) and no translation is performed. Bits [42:40] are not used.

The ITB supports diagnostic reads and writes of the PTE portion of the ITB entries only in PALmode. The entry read or written is determined by the NLU pointer.

Figure 1–13: ITB Block



1.2.4 Branch History Table

Conditional Branches can alter the normal sequential flow of program execution. In EV-5, when the instructions fetched include Conditional Branches, the condition determining the outcome of these branches is not known until atleast S4. Rather than waiting till S4 to determine the outcome of the branch and then resuming the i-stream fetch, EV-5 predicts the outcome of the branch. If the branch is predicted flow-through, it continues fetching instructions from the sequential stream. If the branch is predicted taken, it takes a cycle of penalty to determine the branch target and then starts fetching instructions from the target address.

The branch prediction scheme employed by EV-5 is based on maintaining a 2Kx2 bit Branch History Table (BHT).

The Branch History Table is accessed using the fetch index **I%J_BHT_IDX_ZB_H**<12:4>. When the i-stream fetcher of the IBOX is fetching a subblock of 4 instructions from the ICache, corresponding 8-bit branch history **J%I_BR_HIST_0B_H**<7:0> is fetched from the BHT and supplied to the IBOX directly. However, if the fetcher is fetching the instructions from the refill buffer, the index supplied to the BHT **I%J_BHT_IDX_ZB_H** is a stage ahead relative to the index supplied to the Refill Buffer. Hence, to align the fetched history with the fetched instructions, the history fetched from the BHT is siloed for one stage and then supplied to the IBOX. History bits driven to the ibox are zeroed if the BHT is disabled or the IBOX fetcher is in PAL mode.

Of the 2 bits per instruction, bit<1> of the history is used as a prediction hint by the Branch Predictor logic. If bit<1> is set, the conditional branch is predicted TAKEN and if bit<1> is clear, it is predicted NOT TAKEN. Bit<0> of the history is used to determine the new value of bit<1> when the history is updated.

The 8-bit branch history **J%I_BR_HIST_0B_H**<7:0> fetched from the BHT is siloed by the IBOX. As the instructions flow in the pipe, associated branch history tracks their flow. The siloed history is later used for history updates as explained below. When a conditional branch is issued to the EBOX(or FBOX), the branch logic inside the EBOX(or FBOX) checks the branch condition and determines whether it is taken or not. Based on taken/not_taken feedback from the EBOX(or FBOX), branch history for this conditional branch gets updated and the updated history is written back into BHT. These Updates to the BHT are controlled by History UPdate logic (HUP).

1.2.4.1 HUP Logic

In S4 the EBOX(or FBOX) examines the branch condition and determines branch outcome. The new branch history corresponding to this instruction is calculated according to 2-bit Counter scheme:

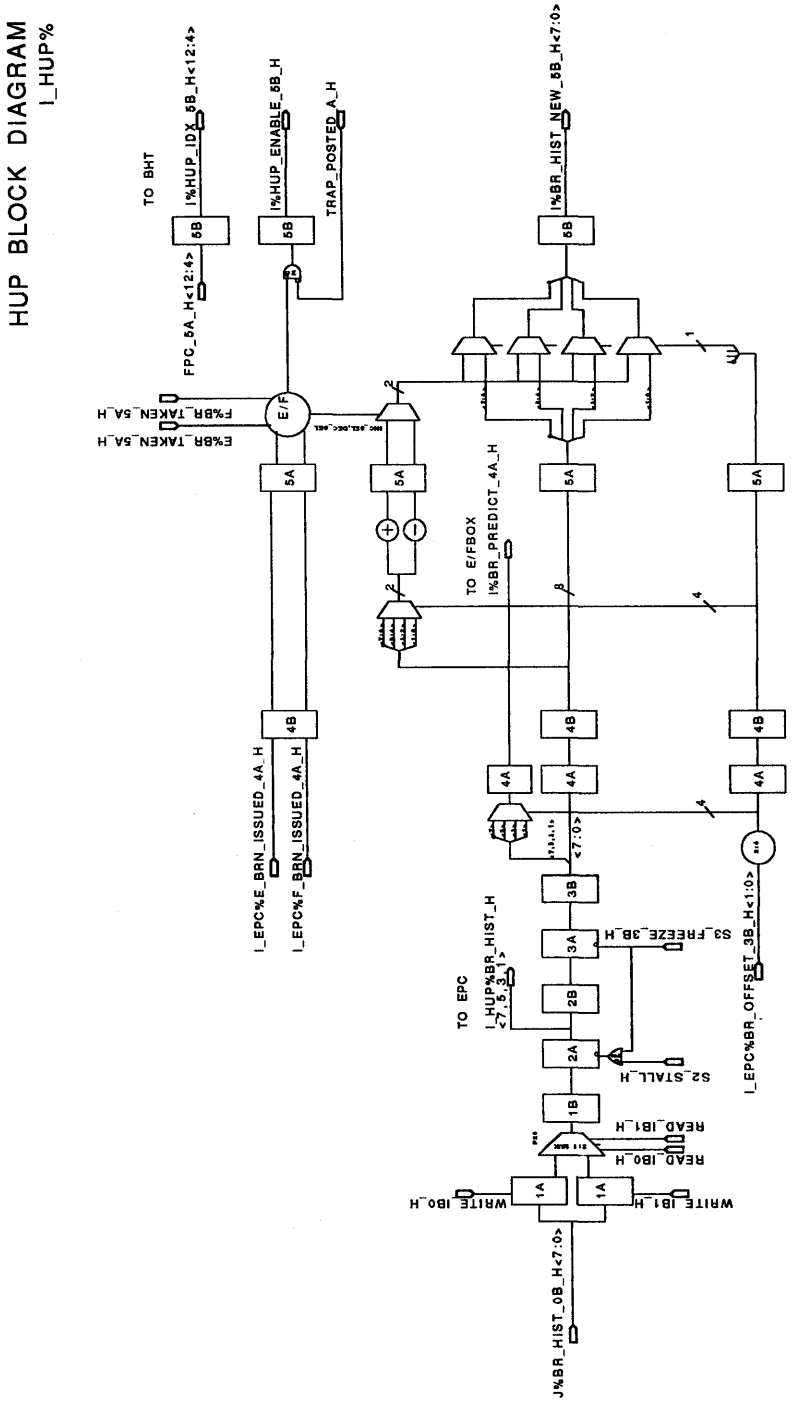
```
case (branch_taken):
        if (branch_history<1:0> != 0x3) new_branch_history<1:0> = branch_history<1:0> + 1;
case (branch_not_taken):
        if (branch_history<1:0> != 0x0) new_branch_history<1:0> = branch_history<1:0> - 1;
```

The BHT supports updates only on a subblock basis, i.e. 8 bits at a time. Hence, the 2-bit new_branch_history is merged with 6-bit siloed history for the other 3 instructions in the same subblock. The resulting 8-bit branch history **I%J_BHT_NEW_5B_H**<7:0> is written into the Branch History Table in S6.

The BHT supports one fetch operation and one update operation every cycle. If the update is occuring to the same location as the one being fetched, i.e. the update index **I%J_HUP_IDX_5B_H** and the fetch index **I%J_BHT_IDX_ZB_H** match and **I%J_HUP_EN_5B_H** is active, the updated history **I%J_BHT_NEW_5B_H**<7:0> is bypassed to the IBOX as the fetched history **J%I_BR_HIST_0B_H**<7:0>.

History updates are not performed in the PAL mode. However, to allow the initialization of BHT during the testing, history updates are performed if BHT is disabled. Note that the BHT is not initialized on ICache fills.

Figure 1–14: Branch History Logic



1.2.5 I-Stream Flow Prediction

Control instructions in the i-stream can alter the sequential fetch path that the i-stream fetcher assumes as the default. EV-5 includes Flow Prediction logic to decode such control flow instructions in the i-stream, predict whether they would change the flow or not and if they are predicted to change the flow, generate the target address from which the fetch should resume.

The Flow Prediction logic can be divided into 3 subsections:

1. Branch Predictor Logic, which decodes the i-stream, detects the change of flow and controls the target calculations.
2. Target Calculation Logic, which determines the branch target address.
3. Return Prediction Stack, which stores subroutine return addresses.

1.2.5.1 Branch Predictor

Branch Predictor logic serves three functions:

1. To detect the change in i-stream flow as a result of branches, jumps, subroutine calls and returns.
2. To control the return prediction stack as result of detected subroutine calls and returns.
3. To control the calculation of target address from which i-stream fetch should be restarted.

In S1 the IB gets a subblock containing upto 4 instructions from the ICache or the Refill Buffer. In the same stage, the Branch Predictor gets associated predecode, displacement and branch history information. In addition, Branch Predictor also gets current index **I_IDX%CURRENT_IDX_0B_H**<12:2> so that it can determine which of the 4 instructions being loaded into the IB are indeed in the i-stream and therefore, valid.

Starting sequentially from the first valid instruction, the Branch Predictor looks for unconditional flow-change instructions (CALL_PAL, HW_REI, JMP/JSR/RET/JSR_COROUTINE (Opcode 1A), BR, BSR) or conditional flow change instructions (Bxx: Integer conditional Branches, FBxx:Floating conditional Branches) whose branch history information flags 'Predict Taken'. If any such instruction is detected in the fetched subblock, it signals a flow change from the sequential stream. The instructions sequentially after this instruction are not in the i-stream and are therefore, invalid.

If the instruction that caused the flow change is CALL_PAL, BSR, JSR or JSR_COROUTINE, it generates a stack PUSH operation, pushing the return address (PC + 1) onto the Return Prediction Stack.

If that instruction is a HW_REI, RET or JSR_COROUTINE, it generates a stack POP operation, popping the return address out of the Return Prediction Stack.

Branch Predictor generates all the necessary control signals for calculating the correct target address.

1.2.5.2 Target Calculation

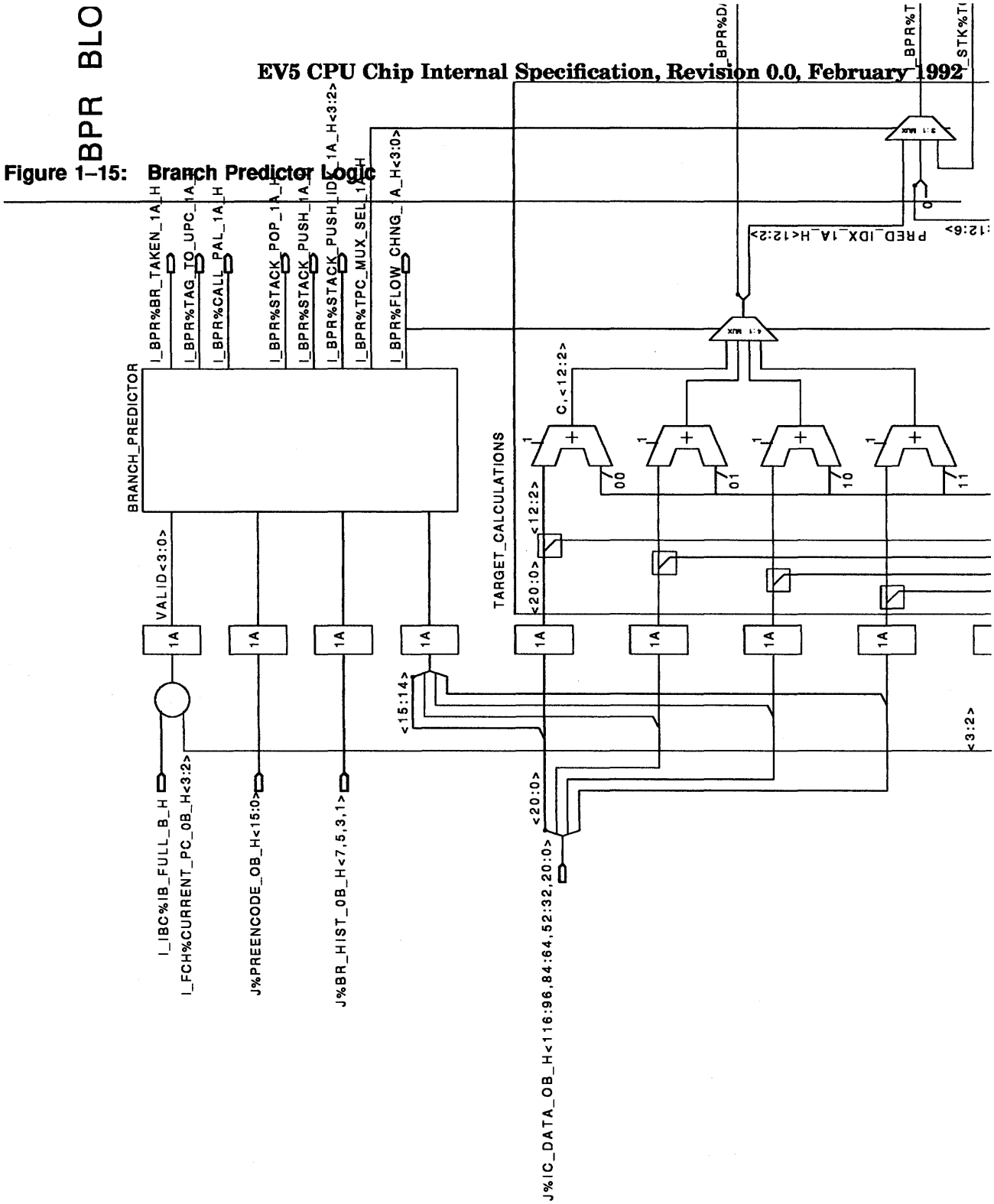
The target address TPC<42:0> from which the fetch should resume is determined according to the type of the instruction causing the i-stream flow change. The following algorithm describes in detail how the target address is determined. In the implementation, the index part of the target address (TPC<12:0>), called **I_FPR%PREDICTED_IDX_1A_H**<12:0> is calculated in S1-A and is jammed into the ICache Index Mux in S1-B to restart the fetch in the next cycle. In this cycle, which is S0 of the target, the remaining part of the target address (TPC<42:13>) is calculated and is called **I_WPC_FPC%FPC_0B_H**<42:13>.

```
switch (Instruction_type) {

       case CALL_PAL:

              TPC<0>     = 1;                    /* Set the PAL mode bit */
              TPC<5:1>   = 0;
              TPC<11:6>  = INSTR<5:0>;           /* This encoding allows 64 CALL_PAL functions, each with
                                                    a code region size of 64 bytes */
              TPC<12>    = INSTR<7>;             /* Previleged instruction encoding */
              TPC<13>    = 1;
              TPC<42:14> = PAL_BASE_IPR<42:14>   /* Since CALL_PAL instr uses PAL_BASE_IPR at the fetch
                                                    end, there should appropriate number of NOPs between
                                                    MTPR to PAL_BASE_IPR and next CALL_PAL */

       case RET, JSR_COROUTINE:

              TPC<12:0>  = TOP_OF_STACK<12:0>    /* POP the return prediction stack */
              TPC<42:13> = IC_TAG<42:13>         /* Predict that ICache Tag = Upper bits of the target PC.
                                                    PC comparison is done in S5 to verify this */

       case JMP, JSR:

              TPC<0>     = Current_PC<0>         /* keep the same mode */
              TPC<1>     = 0;
                                                 /* Use Displacement Hints */
              TPC<12:2>  = Current_PC<12:2> + (INSTR<10:0> << 2) + 1;
              TPC<42:13> = IC_TAG<42:13>         /* ICache Tag = Upper PC */

       case BSR, BR, Bxx, FBxx:

              TPC<0>     = Current_PC<0>         /* keep the same mode */
              TPC<1>     = 0;
              TPC<12:2>  = Current_PC<42:2> + (SEXT(INSTR<20:0>) << 2) + 1;

}

Note: 1) PC<0> = PAL mode.
      2) PC<1> = 0.
      3) Current_PC<42:0> = PC of the instruction causing the flow change.
```

As shown in the EV-5 waterfall chart, a change of i-stream flow leads to one bubble in the fetch sequencing. This bubble gets pressed out in the IB stage or the SLOT stage if there is SLOT or ISSUE stall for at least one cycle.

The block diagram of the Branch Predictor along with some details of the target calculations is given below.

Figure 1–15: Branch Predictor Logic



1.2.5.3 Return Prediction Stack

EV-5 includes a Return Prediction Stack to predict the return address on a return from subroutine. The EV-5 Return Prediction Stack is 12 entries deep and has pointer-repair logic to maintain its correctness in the presence of exceptions (Figure 1–17). The EV-5 implementation of the stack provides the index bits (PC<12:2>) and the PAL-mode bit (PC<0>).

Alpha archiecture provides four instructions to make a subroutine call: BSR, JSR, JSR_ COROUTINE and CALL_PAL. Return from a subroutine is made through the instructions RET, JSR_COROUTINE or HW_REI. The return location is the PC of the instruction after the calling instruction.

When an exception occurs, EV-5 makes a call to a PAL subroutine requesting the service of a priviledged exception handler. The return from this PAL service routine is made through a HW_ REI instruction.The return location is the PC of the instruction that caused the exception.

The Branch Predictor detects subroutine call and return instructions in the i-stream in S1. On detection of a subroutine call instruction, the Branch Predictor generates a stack PUSH operation and pushes longword-incremented PC<12:0> onto the stack. In the implementation, the stack pointer is updated in S1, but the return address is written to the stack in S2. On detection of a return, branch predictor generates a stack POP operation and captures the current top of the stack as the return address.

Note that the above Branch Predictor-intiated stack operations are actually speculative. The instructions prior to the one generating a PUSH or POP can be mispredicted, replayed or trapped. If an instruction is mispredicted (or replayed or trapped), all the PUSHes and POPs that occured after the mispredicted instruction have to be undone. EV-5 Return Prediction Stack has the pointer-repair logic for this purpose.

The pointer-repair logic includes two additional stack pointers, called the 5B pointer and the 7B pointer to 'repair' the stack after an S5 exception and an S7 exception respectively. 5B stack pointer is updated when an instruction reaches S5 successfully. Hence, the 5B pointer represents the correct state that the stack should be restored to if an S5 exception occurs. Similary, the 7B stack pointer is updated when an instruction reaches S7 successfully. Hence, 7B pointer represents the correct state that the stack should be restored to if an S7 exception occurs.

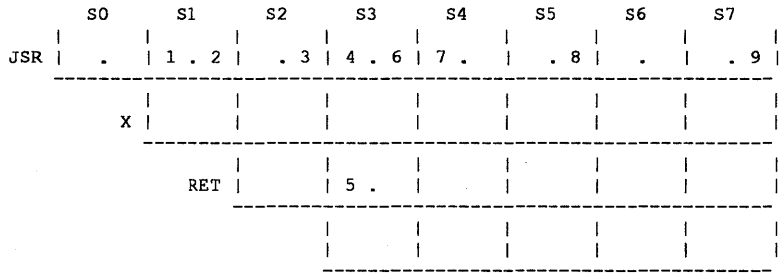
On an S5 exception, therefore, the 5B pointer is copied back to 1B pointer and that constitutes the 5B stack repair. Similarly, on an S7 exception, the 7B pointer is copied back to 1B and 5B pointers and that constitutes the 7B stack repair.

Certain exceptions (ITB miss exception in 5B, any PAL service routine exception in 7B) require that a return address be also pushed on to the prediction stack. If so, the pointers are updated to reflect the new PUSH at the time of the stack repair, but the return address is written to the top of the stack in the next cycle.

The stack pointers are implemented to form circular ring. Hence, the stack wrapps up on overflow or underflow.

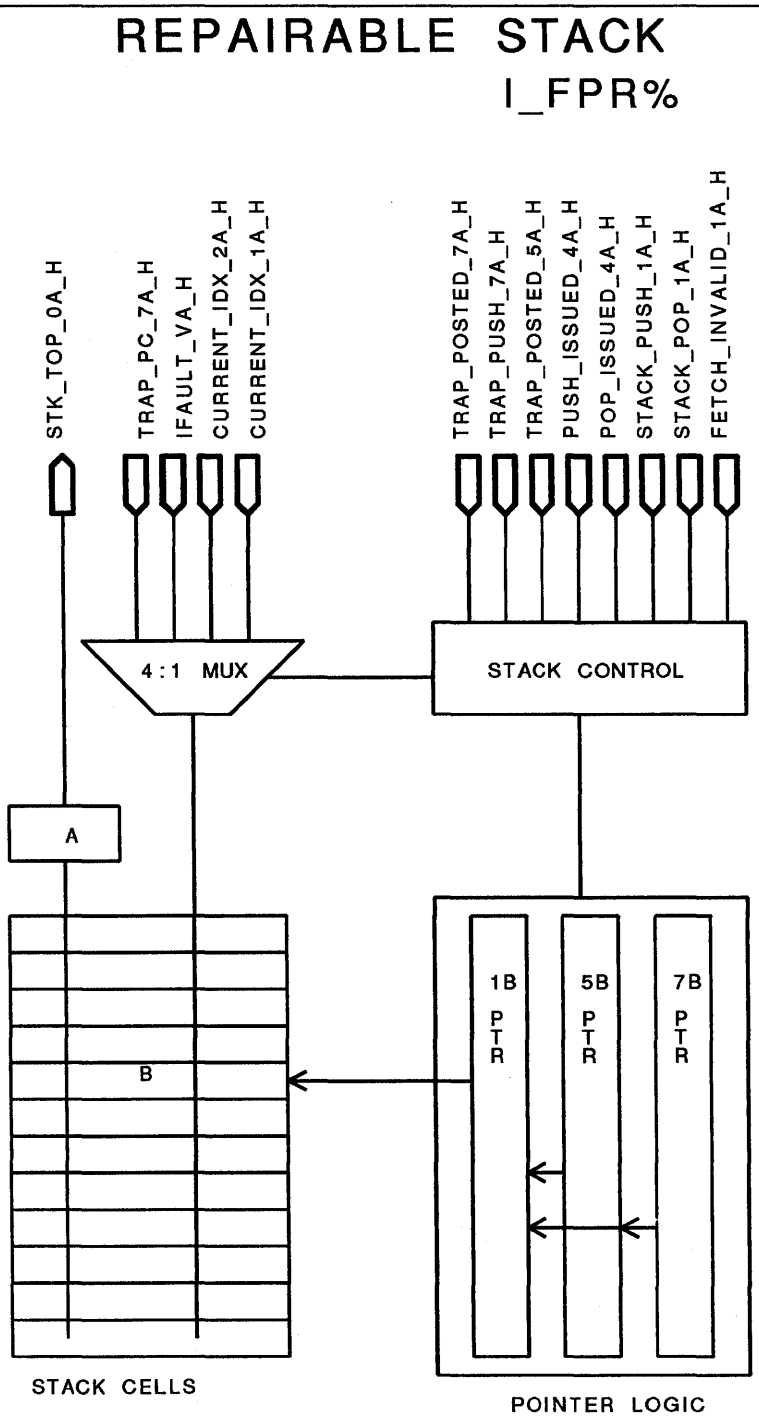
The waterfall chart shown below illustrates the operation of the Return Prediction Stack for a simple case of JSR followed by a RET(Figure 1–16).

Figure 1–16: Return Stack Operation



1. Branch Predictor detects JSR and generates a PUSH.
2. Stack Pointer 1B advances to new top of the stack.
3. Return Address is written at new top of the stack.
4. The pushed return address appears at the top of the stack.
5. Branch Predictor detects RET and generates a POP.
6. The return address appearing at the top of the stack is popped.
7. JSR is issued.
8. Stack Pointer 5B advances to respond to the issued JSR.
9. Stack Pointer 7B advances to respond to the successfully completed JSR.

Figure 1–17: Return Prediction Stack



1.2.6 PC

EV-5 PC logic keeps track of the PC of the instructions moving in the pipeline.

The PC logic functionality can be divided into two sub-sections: The FPC (Fetch PC), which calculates the PC when the instructions are fetched from the ICache or the Refill Buffer, and the EPC (Execution PC) which calculates the PC when the instructions are executed. EPC also compares the PC it calculated with a siloed version of the the PC that the FPC calculated and signals a PC_MISPREDICT trap if they do not match.

1.2.6.1 Fetch PC

Fetch PC sub-section generates the PC associated with each subblock of instructions being supplied to the IB. This PC, called FPC<42:0> is calculated in a pipelined fashion in two stages: SZ and S0.

In the stage SZ, the index part of the PC (FPC<12:0>) called **I%J_IC_INDEX_ZB_H<>** is calculated so that the ICache fetch can be done in S0. In the stage S1 when the ICache fetch is in progress, the tag part of the PC (FPC<42:13>) called **I_WPC_FPC%FPC_0B_H**<42:13> is calculated. As a result, when the fetched instructions enter the IB stage S1, the associated PC (FPC<42:0>) is available for tag comparison (**I_WPC%FPC_FOR_TAG_0B_H**<42:13>, ITB look-up (**I_WPC_FPC%FPC_FOR_ITB_0B_H**<42:13> or PC-silo (**I_WPC_FPC%FPC_0B_H**<42:13>.

The calculation of FPC is based on the the type of event occuring in a particular cycle. These details are given in the algorithm described below.

```
/* FPC Calculation */

switch(Event_Type) {

      case (7A trap or replay):                  /* Trap or replay is selected based on instr-order */

            FPC<42:0> = TRAP_PC<42:0>;           /* trap PC is the PAL entry point of trap handler */
            FPC<42:0> = REPLAY_PC_7A<42:0>;      /* replay PC is the PC of the instr to be replayed */


      case (5A Bxx mispredict, load-use replay or JSR/HW_REI PC mispredict):

            FPC<42:0> = BR_ALT_PC<42:0>;         /* Correct Target PC calculated by the EPC on
                                                    Conditional Branch Mispredict from E/FBOX */
            FPC<42:0> = REPLAY_PC_5A<42:0>;      /* PC of the load-use instruction to be replayed */

            FPC<42:0> = JSR_HW_REI_PC<42:0>;     /* PC from EBOX (JSR), or PC from Exception Address
                                                    Register (HW_REI) on PC mispredict */
      case (ICache miss):

            FPC<42:0> = Current_PC<42:0>;        /* Freeze the PC of the instruction missing in the ICache *

      case (I-stream Flow Change):

            FPC<42:0> = TPC<42:0>;               /* Target PC predicted by the branch predictor. TPC
                                                    is speculative and hence, it is later subjected
                                                    to branch mispredict and PC mispredict checks */
      default:

            FPC<42:4> = Current_PC<42:4> + 1; /* Next sequential subblock of 4 instructions */
            FPC<3:2>  = 0;
            FPC<1:0>  = Current_PC<1:0>;         /* preserve the PAL mode bit */

}

Note:  1. JMP/JSR/RET/JSR_COROUTINE instructions have the same opcode(1A).
          The discussion above refers to all of them by the single keyword "JSR".
       2. Events with higher priority are listed earlier.
```

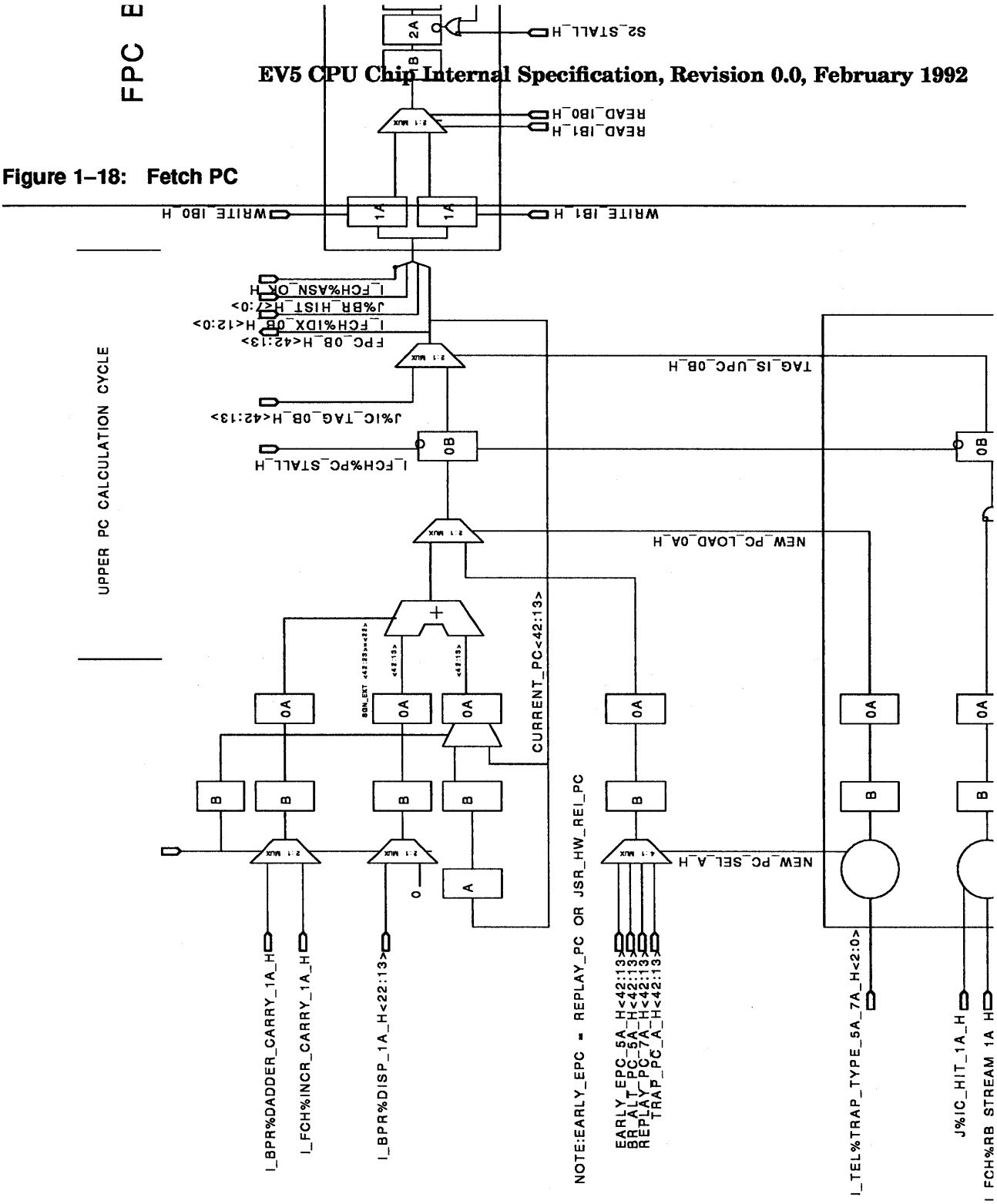
The calculated FPC is siloed upto S7 in the PC datapath. The siloed FPC is used for various purposes such as calculating BR_ALT_PC or REPLAY_PC_XA, loading EXC_ADDR_REG, supply index for history updates, etc.

Note that the FPC<42:0> calculated in the event of I-stream flow change is actually speculative. As explained in the Section 1.2.5, TPC<42:0> is calculated based on the prediction of conditional branch outcome, Return Prediction Stack or ICache tag. Since the prediction can be wrong, the TPC can be wrong and hence the FPC can be wrong.

In the case of a conditional branch, FPC<42:0> is the predicted target from which the I-stream fetch continues. When that conditional branch is issued, the EBOX(FBOX) branch logic determines the branch outcome and compares it with the predicted outcome. A Mranch Mispredict trap is signaled in 5A if the prediction turns out to be wrong. The fetch is restarted from BR_ALT_PC<42:0>.

In the case of a JSR or a HW_REI, FPC<42:13> is predicted to be the ICache Tag<42:13>. If the prediction turns out to be wrong, or there is an ASN mismatch, PC mispredict logic signals a PC mispredict trap in 5A and the fetch is restarted from JSR_HW_REI_PC<42:0>.

Figure 1–18: Fetch PC



1.2.6.2 Execution PC

EPC generates BR_ALT_PC<42:0>, receives JSR_HW_REI_PC<42:0> and performs PC mispredict calculation.

1.2.6.2.1 BR_ALT_PC

Every time a control instruction is issued, an associated BR_ALT_PC is generated in S4. BR_ALT_PC has two meanings depending on the type of the control instruction. If the control instruction is a conditional branch predicted Not Taken, BR_ALT_PC is the PC of the conditional branch target. Otherwise, it is the PC of the instruction after the associated control instruction.

BR_ALT_PC generation logic examines the pipes FA and E1 in S3. It identifies the type of control instruction and uses siloed branch history to calculate the BR_ALT_PC:

```
if (FBxx/Bxx, predicted Not Taken)
      BR_ALT_PC<42:0> = PC of the branch + displacement + 4;

else
      BR_ALT_PC<42:0> = PC of the control instruction + 4;
```

BR_ALT_PC is an all-in-one address for multiple tasks. If the control instruction is a conditional branch and the EBOX or FBOX signals a Branch Mispredict, BR_ALT_PC is the address from which the i-stream fetch is to be restarted. If it is a JSR (Opcode 1A), BR_ALT_PC is the PC to be supplied to the EBOX. Finally, if it is a CALL_PAL, BR_ALT_PC is the address to be latched into the Exception Address Register.

1.2.6.2.2 JSR_HW_REI_PC

A JSR instruction (Opcode 1A) causes the i-stream to jump to an address pointed to by an integer register. When a JSR instruction is slotted, this address appears on the EBOX bus **E%PC_3B_H**<63:0>. HW_REI is similar, except that the address is stored in the IBOX Exception Address Register and it appears on the IBOX bus **I_IPR%EXC_ADDR_REG_B_H**<63:0>. The Issue Decode Logic inside the PC section decodes the type of the control instruction slotted in S3 and selects one of the above two PCs as the JSR_HW_REI_PC. Since a JSR instruction does not affect the existing PAL mode setting, **E%PC_3B_H**<1:0> are ignored and instead **I_WPC_SIL%FPC_3B_H**<1:0> is used to form JSR_HW_REI_PC<1:0>. **E%PC_3B_H**<63:43> are checked for correct sign extension. If there is a sign-extension error, the error is siloed till S7 and then reported as an i-stream Access Violation Fault.

1.2.6.2.3 PC Mispredict

The JSR_HW_REI_PC is the right PC where the i-stream should jump to. When a JSR or HW_REI is issued, the Issue Decode Logic posts a PC Comparison in S4-A to compare the right PC with the PC predicted by the Fetch PC logic.

The PC predicted by the Fetch PC logic is available in the PC silo. However, this PC can currently be in S3 or in S2 depending on whether the i-stream flow-change bubble is pressed or not. The PC Mispredict logic relies on the 'Target' bit supplied by the IB to determine where the PC is. If the Target bit is TRUE at the time the JSR or HW_REI is issued, PC is already in S3 and hence the comparison can be done right in the same cycle in phase B. However, if the target bit is FALSE, PC is yet to come in S3. The PC comparison is then postponed till next cycle. In any case, if a trap is posted in a cycle the PC comparison is to be done, the PC comparison is cancelled.

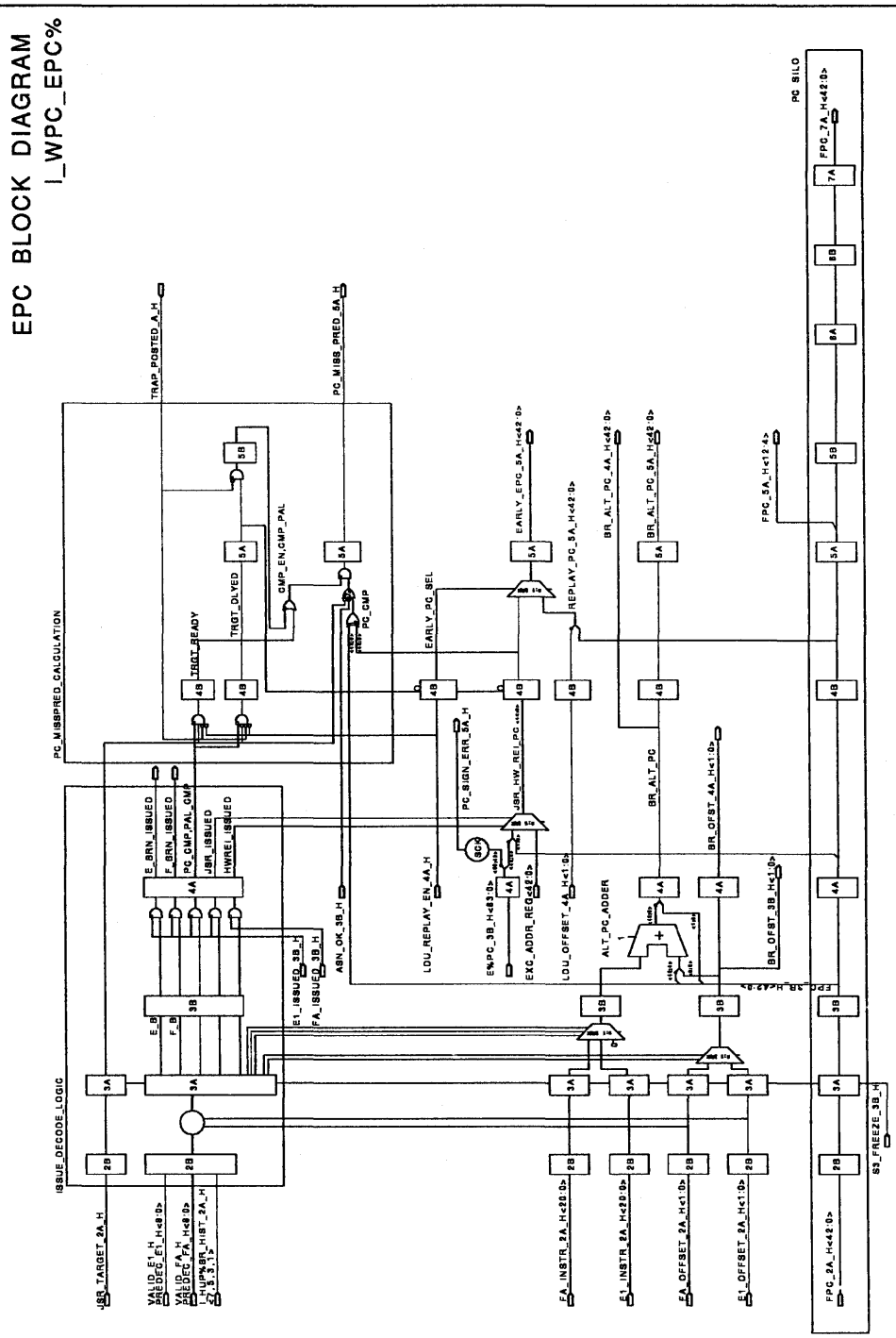
The address comparison takes place in phase B. Bits <42:2> are compared. If there is an address mismatch, a PC mispredict trap **I_WPC%PC_MISPREDICT_5A_H** is signaled. Note that this address mismatch could be because of the following three reasons:

1. JSR hint bits (INSTR<10:0> were wrong.
2. Return Prediction Stack was corrupted.
3. The address was not in the ICache and hence, the assumption of FPC<42:13> = IC_Tag<42:13> was wrong.

In addition to the address mismatch, a PC mispredict trap is also signaled if any of the following conditions was TRUE:

1. Target bit was FALSE when the PC comparison was done.
2. ASN_HIT was FALSE.
3. For a HW_REI PC comparison, PAL mode bit (<0>) did not match.

Figure 1–19: Execution PC



1.2.7 Instruction Buffer(IB)

The IB stage consists of two four-entry instruction buffers. The two buffers form a circular FIFO queue. The IB control logic maintains a set of read/write pointers which control the operation of the IB. The IB is written in S1A of the pipe, whenever the output of the ICACHE or the Refill Buffer is valid. The IB is not written during the bubble cycle caused by any one of the following flow-change instructions:

- Conditonal branch predicted taken.
- Unconditional branch, JMP, CALLPAL or HW_REI.

If the slot stage is finished with the last set of instructions it was operating on, the IB supplies it with a set of four new instructions in S1B. The IB stage can fill up if the slot stage does not accept instructions from the IB, at the rate at which the ICACHE/RFB delivers them. The IB then asserts a backpressure signal **I%J_IB_STALL_A_H** to the cache/fetcher indicating that it cannot accept any further data. The signal **I%J_IB_STALL_A_H** is asserted in S2A whenever the IB clocks in valid data in S1A, causing both buffers to go full and if the slot stage is not done with the last set of four instructions that it was operating on. As an optimization **I%J_IB_STALL_A_H** is deasserted in S2A if a flow-change instruction was latched by the IB in S1A. This can possibly supress some bubbles in the branch-taken path.

The IB stage maintains a valid bit for all 8 instructions in the 2 buffers. Whenever an istream reference is not octaword aligned (as in the case of a branch to the middle of an octaword), these valid bits can be used to selectively invalidate some instructions from a group of four.

The IB stage also has a flow-change bit for all 8 instructions. The branch prediction logic detects the presence of a flow change instruction in the set of four instructions presented to the IB. This information is stored along with the instruction in the IB and is used by the SLOT stage to invalidate all instructions following a flow change instruction.

The IB control logic controls the movement of addresses along the PC pipe. This ensures that the PC and the instruction move in lock-step. In addition to the PC, the IB control also controls the movement of other miscellaneous information associated with an instruction. Table 1–3 contains a list of all miscellaneous bits that are piped along with the instruction.

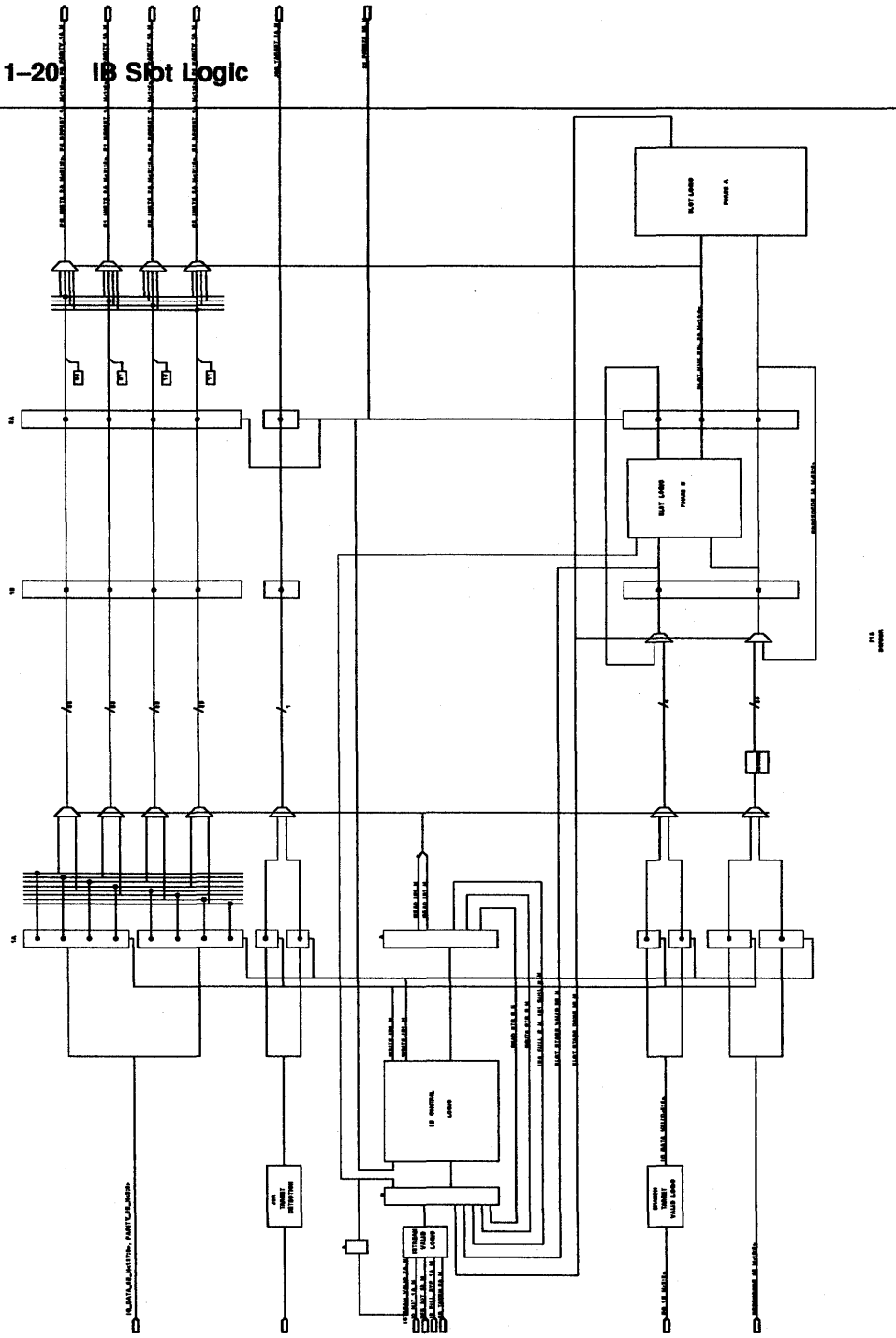
Table 1–3: MISCELLANEOUS IB BITS

Name	Width	Description
ASNOK	1	On JMP/HW_REI instructions indicates a predicted ASN field match
JSR_TARGET	1	Instr stream is the target of a JSR
PAL_MODE	1	PAL Mode
IACCVIO	1	IStream ACCVIO
ITBMISS	1	ITBMiss - trap later

1.2.7.1 HW_REI - stall prefetch

The hardware supports a special encoding of the HW_REI instruction which inhibits prefetching. Whenever this special HW_REI instruction is clocked into the IB in S1A, the IB control logic asserts I%J_IB_STALL_A_H in S2A. The stall signal remains asserted until 3 cycles after the HW_REI is issued i.e. when the HW_REI instruction is about to enter S6 of the execution pipeline. This special instruction has been devised in order to synchronize Ibox changes (such as ITB writes which take place in S6) with the HW_REI. Using the the special HW_REI instruction after an MTPR ITB_TAG or MTPR ITB_PTE will ensure that instructions following the HW_REI, do not access the ITB until after the ITB write is complete.

Figure 1–20 IB Slot Logic



1.2.8 Instruction Slotting

The SLOT stage of the IBOX pipeline receives a set of four instructions and valid bits from the IB stage in S1B. The function of the SLOT stage is to route each instruction to the appropriate integer/floating point execution unit required by the instruction. The slotting process begins with the first valid instruction and is carried out "in-order" with respect to the actual instruction ordering. In the event that the appropriate execution unit is not available for an instruction (i.e. an earlier instruction has already been slotted to that pipe), the slotting process stops and no attempt is made to slot instructions that follow in that cycle. Figure 1–21

The SLOT stage logic is assisted by five pre-decode bits calculated for each instruction during the ICACHE fill operation. These predecodes identify, among other things, the pipe(s) that a particular instruction requires for execution.

The SLOT stage routes all 32 bits of the instruction to the appropriate pipe in S2A. In addition it also sends a two bit encoding for each instruction that indicates the position of the instruction within the block of 4 instructions. The first of the 4 instructions is tagged as 00#2, the second as 01#2, etc. This encoding is used by the Issue stage to prevent out of order issue of instructions.

The SLOT stage also generates a valid bit for each of the four execution slots. The Issue stage receives the valid bits in S2B and disables the instruction-issue for the non-valid pipes.

Instructions that are not slotted in a cycle are retried in the following cycle. This process continues until all valid instructions in the SLOT stage have been slotted and have advanced to the issue stage. Until this happens the SLOT stage does not accept any further instructions from the IB stage.

1.2.8.1 Special Slotting Rules:

Instructions can belong to any of the following categorties:

- Requires Integer Pipe E0 (Class E0)
- Requires Integer Pipe E1 (Class E1)
- Requires any one of the Integer Pipes E0/E1 (Class EE)
- Requires the Floating Add Pipe FA (Class FA)
- Requires the Floating Mul Pipe FM (Class FM)
- Requires any one of the Floating Pipes FA/FM (Class FE)
- Requires NO execution slots- instruction is a UNOP (Class UNOP)

The slotting of an EE class instruction is dependent on whether a particular instruction sequence is detected in the group of four instructions to be slotted. If the instruction sequence EE..EE or EE..E1 is detected, an attempt is always made to first slot the EE class instruction to pipe E0. If pipe E0 is unavailable, the instruction is then slotted to pipe E1. If the sequences EE..EE or EE..E1 are not present, an attempt is first made to slot the EE class instruction to pipe E1. IF pipe E1 is unavailable, the instruction is then slotted to pipe E0.

If an instruction is capable of being issued to either the FA or FM pipes, an attempt is always to made to first slot it to pipe FA.

In addition to execution unit conflicts, any one of the following situations can defer or stop instruction slotting.

1. Slotting stops when a flow-change instruction is encountered. All instructions that follow the flow-change instruction are killed. The instructions which belong to this category are:
   - Predicted taken Floating/Integer Conditonal branches.
   - Unconditional Floating/Integer branches, JSR, JSR_COROUTINE, JMP, RET, CALLPAL or HW_REI.
2. Slotting is deferred when a second branch-class instruction is encountered (i.e any Branch, JSR, JSR_COROUTINE, JMP, RET, CALLPAL or HWREI). The slotting of the second branch-class instruction and all following instructions is deferred to the next cycle. Splitting of branch-class instructions is required in order to avoid having to track multiple alternate PCs in the event of misprediction and also simplifies some other control.
3. Slotting is deferred when a Load-class instruction (i.e Floating or Integer Load) is encountered following a Store-class (i.e Floating or integer Store, Opcode = 0x18 and LDX_L). Slotting of the Load-class instruction and all instructions that follow is deferred to the next cycle. This is done because issuing Loads and Stores simultaneously causes Dcache resource conflicts.
4. Slotting is deferred when a Store-class instruction (i.e Floating or Integer Store, Opcode = 0x18 and LDX_L) is encountered following a Load-class (i.e Floating or integer Load).
5. Slotting is deferred when the slot stage detects instructions sequences of the type I-F-I-I or F-I-I-I, where F refers to a floating instruction and I refers to an integer instruction. The slotting of the second integer instruction (i.e the third instruction) is deferred to the next cycle. This facilitates the compiler use of EV4 padding rules for EV5.

Figure 1–21: Instruction Slotting

EV5 Opcodes, What they do, where they go....

Notes:

Opc. --> Opcode (bits 31:26)
Instr.(s)/Mnemonic --> Contains the Mnemonics associated with each field... Also contains function field decodes when appropriate
Primary Exec. Boxes --> boxes that "operate" on the instruction
Pre-Decodes --> 4 bit predecode value stored in the ICACHE with the instruction.
Prod. --> Register results produced
Cons. --> Register results consumed
Bubbles --> Number of pipe bubbles inserted until the "Prod." register is available.
A 0 indicates that the result is available to the next cycle. A "none" indicates that no destination register is locked and therefore this instruction will not block multiple issue due to conflicts in the current issue cycle.

Opc.	Instr.(s) /Mnemonic	Exec. Boxes	Pred. Value	Issue Pipes	Prod.	Cons.	Bubbles	Comment
00	CAL_PAL	IBOX	1010	E1	none	none	none	IBOX stalls further issuing until all the pipes have have drained and all loads have gotten past the trap point. The IBOX then looks up the dispatch address in a table and then alters the flow to that address. The EBOX pipe basically treat this as a NOP... ****Moved to PIPE E1 only*****
01-07	reserved	IBOX	0001?	E0				Reserved Instruction Fault on Issue, slot maps to E0
08	LDA	EBOX	0011	E0 or E1	Ra	Rb	0	Ra = Rb + disp (Not really a LOAD!! -- totally EBOX)
09	LDAH	EBOX	0011	E0 or E1	Ra	Rb	0	Ra = Rb + (disp*65535) (Not really a LOAD!! -- totally EBOX)
0A	reserved	IBOX	0001?	E0				Reserved Instruction Fault on Issue, slot maps to E0
0B	LDQ_U	EBOX, MBOX	0010	E0 or E1	Ra	Rb	2 hit	Unaligned integer load... The EBOX treats this like a normal load. Alignment Trap not generated..
0C-0E	reserved	IBOX	0001?	E0				Reserved Instruction Fault on Issue, slot maps to E0
0F	STQ_U	EBOX, MBOX	0000	E0		Ra,Rb	none	Unaligned integer store... The EBOX treats this like a normal store. Alignment Trap not generated.
10	Iarith (binary func)	EBOX	0011					
	(xxx0xx)			E0 or E1	Rc	Ra,Rb	0	Integer Adds/Subs including scaled... either pipe, 1 cycle latency on Rc
	ADDi							
	SnADDi							
	SUBi							
	SnSUBi							
	(xxx1xx)			E0 or E1	Rc	Ra,Rb	1	Integer Compares ***Now in either E pipe but still has 2 cycle latency****
	CMPxx							
	CMPUxx							
	CMPBGE							
11	Ilogs (binary func)	EBOX	0011					
	(xxx0xx)			E0 or E1	Rc	Ra,Rb	0	Integer Logicals 1 cycle latency, either Pipe
	AND							
	BIS							BIS R31,R31,R31 will be the preferred method of creating integer NOPs in EV5!!!
	XOR							
	BIC							
	ORNOT							
	EQV							
	(xxx1xx)			E0 or E1	Rc	Ra,Rb	1	Integer CMOVs, two cycle latency, either pipe
	CMOVxx							

Figure 1–21 Cont'd on next page

Figure 1–21 (Cont.): Instruction Slotting

Opc.	Instr.(s) /Mnemonic	Exec. Boxes	Pred. Value	Issue Pipes	Prod.	Cons.	Bubbles	Comment
12	Ishft SLL SRL SRA EXTxx INSxx MSKxx ZAPNOT ZAP	EBOX	0001	E0	Rc	Ra,Rb	1	Integer Shifts ***MOVED to PIPE E0****
13	IMUL MUL1 UMULH	EBOX	0001	E0	Rc	Ra,Rb	MULL 7 UMULH MULQ 9	Integer Multiplies Latency based upon function field... No IBOX issues around latency since IMUL bit controls... ****MOVED to PIPE E0****
14	reserved	IBOX	0001?	E0				Reserved Instruction Fault on Issue, slot maps to E0
15	VAX_FP (binary func) (xxxxxxxxx00) (xxxxxxxxx01) (xxxxxxxx1xx) ADDF ADDG SUBF SUBG CMPGxx CVTGxx CVTDG CVTQF	FBOX	0110	FA	Fc	Fa,Fb	4	Add/Subs/Compares/Converts.... SRM requires that Fa be F31 on Converts
	(xxxxxxx0010) MULF MULG		0101	FM	Fc	Fa,Fb	4	Floating Multiplies
	(xxxxxxx0011) DIVF DIVG		0110	FA	Fc	Fa,Fb	n?	Floating Divides, latency depends upon the data type in the function field. The IBOX doesn't care since all divides set the DIVIDE bit dirty.
16	IEEE_FP (binary func) (xxxxxxxxx00) (xxxxxxxxx01) (xxxxxxxx1xx) ADDS ADDT SUBS SUBT CMPTxx CVTQS CVTQT CVTTx	FBOX	0110	FA	Fc	Fa,Fb	4	Adds/Subs/Compares/Converts SRM requires that Fa be F31 on Converts... ??Do we understand rounding to +/- infinity??
	(xxxxxxx0010) MULS MULT		0101	FM	Fc	Fa,Fb	4	Floating Multiplies
	(xxxxxxx0011) DIVS DIVT		0110	FA	Fc	Fa,Fb	n?	Floating Divides... The latency depends upon the data specified in the function field. The IBOX doesn't care what the acutal latency is since they all set the divide bit.

Figure 1–21 Cont'd on next page

Figure 1–21 (Cont.): Instruction Slotting

Opc.	Instr.(s) /Mnemonic	Exec. Boxes	Pred. Value	Issue Pipes	Prod.	Cons.	Bubbles	Comment
17	DI_FP (hex func.)	FBOX						Datatype Independent Floating Point Ops
	(all excpt. 020)		0110	FA	Fc	Fa,Fb	4	Floating CMOV, Mx_FPCR, Int-Int Converts, CPYS negate, CPYS & exponent. To simplify decoding, the MT_FPCR has a latency of 5 even though it has no register destination. In addition, an MF_FPCR will stall on Fa,Fb dirty even though it doesn't actually read the regs... This is acceptable since the SRM requires DRAINTs around the Mx_FPCR instructions. the SRM requires that Fa is F31 on the Converts...
	CPYSN							
	CPYSE							
	FCMOVxx							
	MT_FPCR							
	MF_FPCR							
	CVTQL/x							
	CVTLQ							
	(020)							Copy Sign. Using CPYS F31,F31,F31 results in a floating point NOP. Therefore, the CPYS opcode will be slotted to either available pipe to effect the NOP case. As a result of the NOP decision, real CPYS will be done in the first available pipe, FA or FM.
	CPYS		0111	FM or FA	Fc	Fa,Fb	4	
18	MISC. (hex funct.)		0001					Goofy Miscellaneous Instructions... Most can be treated as either EBOX Nops or special cases of LOADS (see FETCHx below)... Issue??? Do we need to decode all 16 bits of the function field... Could we simply check the top 4-8 bits as in EV4????? Issue??? The current plan is dirty check these like special cases of LOADs for the Rb source. The Ra dest will not set the load/miss bit unless the inst. is RPCC,RC, or RS. Having the unused register src/ dest. fields set to R31 will avoid unnecessary dirty check stalls in the IBOX.
	Draint (0000)	IBOX		E0			x	Stops issue until all 4 pipes have drained. Outstanding Loads DO NOT have to complete Slotting routes this to pipe E0 which can either NOP it or do the FETCHx routine.
	MB (4000) (4400) (4800) (4C00)	IBOX, MBOX		E0			x	Stops issue of MBOX instructions until the MBOX clears a flop in the IBOX. Slotting routes this to pipe E0 which can either NOP or do the FETCHx routine. ??Issue?? A 2nd MB while the first is still pending should stall issue??. This allows for DVT software to completely freeze the machine. When the first MB finishes and clears the flop, the 2nd will be executed and will reset the flop until the MBOX clears it a 2nd time
	FETCHx (8000) (A000)	MBOX, EBOX		E0		Rb	none	The Ebox sends (Rb + 0) to the MBOX which then fetches the page. The EBOX treats this like a load except for zeroing the displacment field in the addition.
	RPCC (C000) RC (E000) RS (F000)	MBOX, EBOX		E0	Ra		2	IBOX will lock the Ra destination as if these were LOADs An implied MB will occur upon issue. The MBOX will clear the MB flag to reenable MBOX instructions and will assert DC_HIT or a proper FILL sequence to load and unlock Ra. The IBOX will allow these two clears to occur on different cycles if necessary.

Figure 1–21 Cont'd on next page

Figure 1–21 (Cont.): Instruction Slotting

Opc.	Instr.(s) /Mnemonic	Exec. Boxes	Pred. Value	Issue Pipes	Prod.	Cons.	Bubbles	Comment
19	HW_MFPR MBOX,CBOX, Scache IPRs	MBOX, EBOX IBOX	0011	E0	Ra		2 hit	If the IPR is in the M or C Boxes, this looks like a load to the I and E boxes. The MBOX will either return the data 2 cycles later at HIT time or will assert a MISS and a "FILL" when the data is available.
	IBOX IPRs, PAL_TEMPS			E1			0	If the IPR is in the IBOX, the data is provided on the I%PC_BUS which the EBOX muxes inplace of the Logic box results in S4. The data is therefore bypassable to the next cycle as if it was a logic box result. A few IBOX IPRs might not able to be read in 1 cycle. Any of these registers would use a special IPR_TEMP IPR register as an intermediate step in a two step read operation NOTE: It is up to PAL code to insure that the proper slot is used depending upon the source box. i.e. if an IBOX IPR or PAL temp is being read, PAL code MUST pad the instruction block with a leading integer NOP or an unrelated integer op to use the E0 pipe. Likewise if the IPR is in the MBOX, PAL code must insure that the MFPR is the first integer instruction in the block so that it gets the E0 pipe. NOTE; It is possible (in fact it is a feature) to dual issue MFPRs as long as the destinations are not identical and one is in the IBOX while the other is in the MBOX, CBOX, or Scache.
1A	JSR	EBOX, IBOX	1001	E1	Ra	Rb	0	The IBOX provides the old PC in 3B over the I%PC bus. This can then be bypassed to the next cycle. The EBOX provides the Rb value (the target PC) in 3B over the E%PC_BUS. In theory, dirty logic could allow a 1 latency on the Ra dest, however, I need to check to see if this is even a needed case.
1B	HW_LD	EBOX, MBOX	0010	E0 or E1	Ra	Rb	2 hit	To the IBOX and the EBOX this looks almost like a normal load. The only difference is that the displacement field is only 12 bits wide. NOTE: PAL Code will be restricted from dual issuing HW_LD with the Locked option and any other MBOX inst. In addition, the HW_LD_L must be routed by E0 by explicit code padding.
1C	reserved	IBOX	0001?	E0				Reserved Instruction Fault on Issue..slot maps to E0/E1?
1D	HW_MTPR MBOX,CBOX, Scache IPRs	EBOX, IBOX, MBOX	0011	E0		Ra	none	For MBOX IPRs, the EBOX pipe will provide the data on the E%VA bus in 4B and over the E%ST_DATA_BUS.
	IBOX IPRs, PAL_TEMPs			E1				For IBOX IPRs, the EBOX pipe will provide the data on the E%PC bus in 3B. NOTE: It is possible (in fact it is a feature) to be able to to dual issue an MTPRs. One must be in the IBOX and the other must be in the MBOX,CBOX, or Scache NOTE: It is up to PAL code to insure that the proper slot is used depending upon the source box. i.e. if an IBOX IPR or PAL temp is being written, PAL code MUST pad the instruction block with a leading integer NOP or an unrelated integer op to use the E0 pipe. Likewise if the IPR is in the MBOX, PAL code must insure that the MFPR is the first integer instruction in the block so that it gets the E0 pipe.

Figure 1–21 Cont'd on next page

Figure 1–21 (Cont.): Instruction Slotting

Opc.	Instr.(s) /Mnemonic	Exec. Boxes	Pred. Value	Issue Pipes	Pred.	Cons.	Bubbles	Comment
1E	HW_REI "normal"	IBOX	1100	E1			none	The EBOX NOPs this.... The "normal" HW_REI uses the 1100 predecode
	"special"		1110					The "special" HW_REI uses the 1110 predecode
1F	HW_ST	EBOX, MBOX	0000	E0		Ra,Rb	none	This looks almost like a normal STORE to the EBOX pipe except that the displacement field is only 12 bits wide. NOTE: The MBOX must clear the MB flop to resume MBOX instruction issuing when the _C option is used.
20-23	Floating LDs LDF LDG LDS LDT	EBOX, MBOX, FBOX	0010	E0 or E1	Fa	Rb	2 hit	The EBOX calculates the address the same way as if this was an Integer Load. The MBOX determines that this is a floating load and routes the data appropriately.
24-27	Float. Stores STF STG STS STT	EBOX, MBOX, FBOX	0100	E0		Fa,Rb	none	The FBOX sends the Fa register to the MBOX on it's STORE bus while the E0 pipe calculates the address in the same manner as for integer stores.
28-29	Int. Loads LDL LDQ	EBOX, MBOX	0010	E0 or E1	Ra	Rb	2 hit	The EBOX calculates the address and sends the E%VA bus to the MBOX in 4B. Dirty logic will lock the dest. for two cycles until HIT time. At that point, a LOAD-MISS-REPLAY may occur if the data is bypassed but found to MISS. Misses will lock the register until the MBOX indicates a FILL has occurred to that particular destination.
2A-2B	Load Locked LDL_L LDQ_L	EBOX, MBOX	0000	E0				Similar to normal Integer Loads but will inhibit dual issue of MBOX instructions and will only slot to the E0 pipe....
2C-2D	Int. Stores STL STQ	EBOX, MBOX	0000	E0		Ra,Rb	none	The EBOX calculates the address and sends it to the MBOX on the E%VA bus in 4B. The data is sent on the E%ST_DATA bus in 4A/4B.
2E-2F	Store Cond. STL_C STQ_C	EBOX, MBOX	0000	E0	Ra	Ra,Rb	2 hit*	Similar to normal stores except that MBOX instructions will be inhibited by an implicit MB. The MBOX will clear the MB flop when it is ready to resume accepting instructions.. The destination register Ra will be be locked by the dirty logic until a FILL occurs to the appropriate register. DC_HIT should NOT assert at HIT time on these instructions.
30	BR	EBOX, IBOX	1101	E1	Ra		0	The IBOX sends the PC in 3B over the I%PC bus.
31-33	Float. Branch. FBEQ FBLT FBLE	FBOX, IBOX	1111	FA		Fa	0	The FBOX sends BR_SUCC and BR_MISPRED in 5A.
34	BSR	EBOX, IBOX	1011	E1	Ra		0	The IBOX sends the PC in 3B over the I%PC bus
35-37	Float. Branch. FBNE FBGE FBGT	FBOX, IBOX	1111	FA		Fa	0	The FBOX sends BR_SUCC and BR_MISPRED in 5A.
38-3F	Int.Cond.Br.	IBOX, EBOX	1000	E1		Ra	0	The EBOx sends BR_SUCC and BR_MISPRED in 5A.

Figure 1–21 Cont'd on next page

Figure 1–21 (Cont.): Instruction Slotting

Revision History:

Date	Who	What
13-Nov-91	rpp	Created the table
14-Nov-91	rpp	Changes from IBOX meeting 13-Nov a. Converts are required by SRM to have Fa as F31 which simplifies the decoding of the floating point operates b. Clarify dirty checks on OPC 18 instructions c. Add Homayoon's request that a 2ns MB causes a complete pipeline stall until the first MB finishes d. Change HW_MFPR pipes from "E0 and E1" to "E0 or E1" e. Change HW_LD pipes from "E0 and E1" to "E0 or E1" f. Change HW_REI pipes from "E0 and E1" to "E1 only" g. Change HW_ST pipes from "E0 and E1" to "E0 only" h. Fix mistake on Flt. Store. pipes "E0 and FA" to "E0 and FM" i. Fix mistake on Flt. Branch latency, should be 0 bubbles j. Fix mistake on Int. Cond. Branch latency, should be 0 bubbles.
25-Nov-91	rpp	Clarified HW_MFPR operation for both IBox and Mbox instructions. Added timing for IBOX IPRs and PAL_TEMPs
09-Dec-91	rpp	Updates from IBOX/Arch meeting of 09-DEC. Changes are: a. CAL_PAL moved from "E0 and E1" to "E0" b. reserved opcodes moved from "E0 or E1" to "E0". c. STQ_U moved from "E1" to "E0", this was a mistake in the 25-Nov rev.... d. add pipe E0 to the CMPxx instructions. e. move shifts from "E1" to "E0" f. move IMULs from "E1" to "E0" g. change CPYS to go to either pipe as the new floating NOP. h. add a comment defining BIS R31,R31,R31 as the integer NOP. i. changed all misc. instructions (opcode 18) to pipe "E0" from pipe "E1", also changed several of the comment fields j. changed HW_MFPRs from the MBOX to "E0" from "E0 or E1". All non-IBOX IPRs will return their data on the E0 pipe if the return data at hit time. Added a comment about dual issuing HW_MFPRs. k. changed HW_MTPRs to the following: IBOX and PAL_TEMP are now in "E1". MBOX etc. are in "E0". Added a note about support for dual issuing MTPRs l. added a comment on Load_Locked. Can we issue this to either pipe.
13-Dec-91	rpp	Converted into DECWRITE to get PostScript for the DOCUMENT Spec
17-Dec-91	rpp/vr	Added Predecode values and corrected some of the Comments....
03-Mar-92	rpp	Updates for the 2nd IBOX review

1.2.9 Instruction Issue

The "slotted" instructions are presented to the Issue Stage in S2B where register conflict (also known as Dirty) checks are performed in S3. In addition, final resource availability and serialization of the instructions are performed prior to releasing the instructions to the various execution units.

The Issue Stage's primary output is the signal **I%Z_STALL_3B** which is used to freeze the EBOX and FBOX register file addresses, opcodes, and the earlier stages of the IBOX. The Issue stage is also responsible for signaling the appropriate data bypasses to the EBOX and the FBOX and for driving addresses for reading and writing both the EBOX and FBOX register files.

A block diagram of the Issue Stage is given in Figure 1–22 which is included in the large pull outs.

1.2.9.1 Interface with the Slot Stage

The Slot stage sends the instructions to the Issue stage over the **I_IBS%FM_INST_2A<31:0>**, **I_IBS%FA_INST_2A<31:0>**, **I_IBS%E0_INST_2A<31:0>**, **I_IBS%E1_INST_2A<31:0>** lines.

In addition, the slot stage indicates the relative logical ordering of the four instructions on the **I_IBS%E0_POS_2A<1:0>**, **I_IBS%E1_POS_2A<1:0>**, **I_IBS%FA_POS_2A<1:0>**, **I_IBS%FM_POS_2A<1:0>** lines. The value 00#2 indicates the first instruction of the block of four while 11#2 indicates the fourth instruction.

A valid signal (**I_IBS%E0_VALID_2A**, **I_IBS%E1_VALID_2A**, **I_IBS%FA_VALID_2A**, **I_IBS%FM_VALID_2A**) is also sent from the slotting stage for each instruction to indicate which of the slots contain valid instructions.

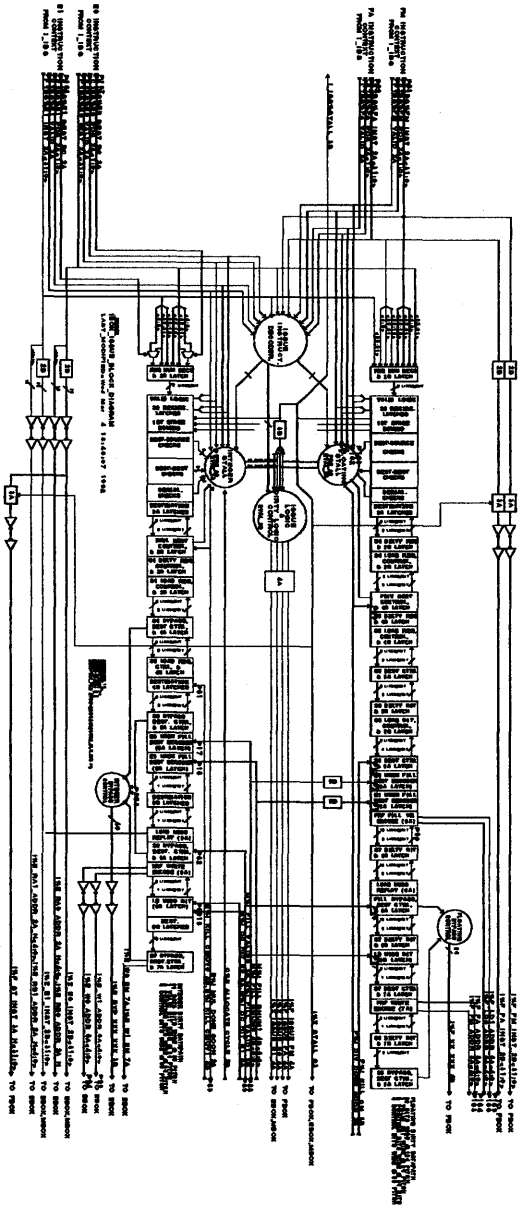
Finally the Slot stage sends a pair of signals (**I_IBS%E0_DEST_RC_2A** and **I_IBS%E1_DEST_RC_2A**) which indicate whether the A field (<25:21>) or the C field (<4:0>) should be used as the destination register for the corresponding integer instructions. Only instructions with opcodes between 10#16 and 17#16 will use the C field. Since all floating instructions use the C field as the destination register, the slotting stage does not have to calculate a mux control for them.

1.2.9.2 Instruction Interface with the E,F,M Boxes

The Issue stage sends the instruction context information to the E,F,M Boxes. It contains 2B latches and buffer/repeaters which output the integer instruction buses as **I%Z_E0_INST_2B<31:0>** and **I%Z_E1_INST_2B<31:0>** to the EBOX and MBOX. The RA and RB fields of both integer instructions are sent directly to the EBOX register file in phase 2A prior to the 2B latch to ease a critical path. The buses **I%E_RA0_ADDR_2A_H<4:0>**, **I%E_RB0,ADDR_2A_H<4:0>**, **I%E_RA1_ADDR_2A_H<4:0>**, and **I%E_RB1_ADDR_2A_H<4:0>** contain the register file address information.

The instruction context information for the FBOX instructions is latched into 2B and conditionally into 3A when **I%Z_STALL_3B** is not asserted. This data is then buffered and sent to the FBOX as **I%F_FA_INST_3A_H<31:0>** and **I%F_FM_INST_3A_H<31:0>**. In addition, the E0 instruction data is latched and send to the FBOX as **I%F_ST_INST_3A_H<31:0>** so that the FBOX can process the data for floating stores.

Figure 1–22: Instruction Issue-Block Diagram



1.2.9.3 Dirty Checks

The Dirty Checks are used prevent the issue of an instruction until it can complete without encountering any data dependencies. Each of the 31 integer registers and each of the 31 floating point registers is allocated a bit in the appropriate Dirty Logic Datapath.

The two Dirty Logic Datapaths mimic the flow of the execution datapaths for pipe length and bypassabilities. Additional stages are present in each datapath to account for IMUL and FDIV destinations, for LOADs that have missed in the D-Cache, and to support the PAL_SHADOW registers. As instructions are issued, the associated destination registers are decoded and latched into the appropriate bit of the datapaths. These destination bits are SILOed down the dirty logic datapaths to indicate that data bound for the corresponding register is present in the execution unit(s) at the corresponding point.

During S3, the four current instructions are checked for conflicts against those instructions that are currently executing in the four execution unit pipelines, against any Loads that have not completed, and against each other.

1.2.9.3.1 DEST-SOURCE Checks

A DEST-SOURCE check is done for each of the two possible source operands of each of the up to four slotted instructions during S3 of the IBOX pipe. These checks are performed by decoding the source register numbers for the current instructions and checking to see if a corresponding destination bit has been set in any of the pipeline stages, the IMUL/FDIV units, or for a LOAD that has missed in the D-Cache.

The source register numbers for integer instructions are not checked against operate destination register numbers currently in S5 or beyond of the two EBOX pipelines since all integer operate results are bypassable by the end of the S5 stage. Matches between integer source registers and integer destination registers in the S4 stage are qualified by the type of instruction executing in S4. Most integer instructions are bypassable by the end of S4 however some (like shifts) are not available until S5 (see Figure 1–21 for a list of instructions and their respective latencies).

All of the 8 source registers are checked against any LOADs that are currently executing in S4 or S5 since LOAD data is not available until S6. In addition, the source registers are checked against any load that has advanced into S7 and missed in the D-Cache. Instructions with source registers that attempt to use data from a LOAD instruction that is in S6 will be issued and will either complete successfully if the LOAD hits in the D-Cache or will generate a LDU-REPLAY as detailed in Section 1.2.9.17 if the LOAD misses.

If a match is found on any of these checks, the instruction with the matching source register is stalled. This stall is termed a DEST-SOURCE stall.

1.2.9.3.2 DEST-DEST Checks

In addition to the DEST-SOURCE checks detailed above, some DEST-DEST checks are performed. These checks are used to preserve the correct ordering of the eventual write operations into the register file. The only checks that need to be performed are for units/instructions that can write the register file out-of order, such as IMUL, FDIV, integer loads that could miss in the D-Cache, and all Floating Loads (since LDFs write the FBOX register file two cycles earlier than floating operates). If any of the following tests fail, the failing instruction cannot issue. The created stall condition is termed a DEST-DEST stall.

1. If either the E0 or E1 instruction's destination register is the same as IMUL unit's destination register, the E0 or E1 instruction is stalled.
2. If either the FA of FM instruction's destination register is the same as FDIV unit's destination register, the FA or FM instruction is stalled.
3. If either the E0 or E1 instruction is a floating LOAD with a destination register that matches the FDIV destination register, the E0 or E1 instruction is stalled.
4. If either the E0 or E1 instruction is a floating LOAD with a destination register that matches the S4 or S5 destination registers in either the FA or FM pipes, the E0 or E1 instruction is stalled.
5. If any of the instructions have a destination register that is the same as the destination register of any Load that has missed in the D-Cache, the instruction is stalled until a FILL occurs returning the register data.
6. If any of the instructions have a destination register that is the same as the destination register of any Load instruction that is currently executing in S4 or S5, the the matching instruction is stalled until the conflicting Load reaches S6 at which point the instruction will be issued. The instruction will either complete successfully if the conflicting Load hits in the D-Cache or a LDU-Reply will occur (see Section 1.2.9.17).

1.2.9.3.3 Current Issue Conflicts

The destination and both source registers of each of the four current instructions are checked against the destination registers of the other three currently slotted instructions to determine if any conflicts exist in the block. If a match occurs between a source/destination and the destination of another instruction which logically proceeds it (as determined by the **I_IBS%E0_POS_2A<1:0>**, etc. lines), the logically latter instruction is stalled. The created stall is termed a Current-Issue stall.

1.2.9.4 Resource Availability Checks

The Issue stage performs a series of checks to insure that the proper execution resources are available prior to releasing an instruction for execution. The results of the following checks are ORed together for each of the four slotted instructions to generate a RESOURCE_STALL signal for each instruction.

1.2.9.4.1 IMUL_BUSY

When an integer multiply is issued, the destination register number is stored in a register in the integer dirty datapath. Subsequent IMUL instructions will check to see that this register is empty prior to issuing. A stall occurs on an IMUL instruction if the multiplier is busy with a previous instruction. It is possible to overlap the final cycle of one multiply with the first cycle of the next multiply if there are no register dependencies. The Issue stage will clear the IMUL_DEST register one cycle prior to the actual end of the first multiply operation to support this feature. The EBOX will send the signal **E%I_MUL_DONE_SOON_2A** to indicate that the multiplier will produce S5 results in 3 cycles.

1.2.9.4.2 IMUL_DONE_SOON

The result of an integer multiply is muxed into the E0 EBOX pipeline at the S5 stage. There must not be a valid instruction in S5 of the E0 pipe when the multiply result becomes available or a data collision will occur. The Issue logic will stall the E0 pipe for one cycle to allow the multiply results to mux into the normal flow. The signal **E%I_MUL_DONE_SOON_2A** causes this stall. The EBOX will signal the MUL_DONE_SOON indication in 2A, in the next cycle (3A), the E0 instruction will be unconditionally stalled to introduce the necessary bubble.

1.2.9.4.3 FDIV_BUSY

A FDIV_BUSY stall occurs when a floating divide is scheduled to execute prior to the completion of a previous floating divide. This stall is similar to the IMUL_BUSY stall described in Section 1.2.9.4.1. The FBOX will send the signal **F%I_FDIV_DONE_SOON_2A** to clear the FDIV_DEST register in the floating dirty datapath. As with the IMUL unit, it is possible to overlap the final cycle of a previous divide with the first cycle of the next divide in the absence of register dependencies.

1.2.9.4.4 FDIV_DONE_SOON

The Floating Divider results are multiplexed into the Floating Add pipe in place of the normal FA results. A bubble is required in the FADD pipe to avoid a data collision. The signal **F%I_FDIV_DONE_SOON_2A** will cause a stall to occur on the FA slotted instruction in the next cycle. This stall is similar to the IMUL_DONE_SOON stall described in Section 1.2.9.4.2.

1.2.9.4.5 STORE_STALL

The Store SILO in the MBOX requires that Store(s) cannot be followed by Load instructions in the second subsequent cycle. The Issue logic in the IBOX will set a bit in a two cycle shift register whenever a STORE is issued. If the low order bit of this register is set, any LOAD type instruction will be stalled.

1.2.9.4.6 FILL_STALL

Fill data returning from the MBOX to the EBOX register file is multiplexed into the appropriate EBOX datapaths in place of operate results at the S6 stage. If an operate instruction is present in S6 when a fill occurs, a data collision would result. To avoid this collision, the CBOX will send a request to the IBOX Issue stage indicating that a FILL operation MIGHT occur. The IBOX will stall both the E0 and E1 pipes inorder to insert the necessary bubbles when this signal **C%I_ALLOCATE_CYCLE_2B** is asserted. LOADs that HIT in the DCache will not require any additional pipe bubbles, since the pipe is already reserved for the MBOX data by the instruction itself.

Fills for Floating Loads will not require these bubbles since the FBOX register file contains two write ports dedicated to LOAD data. In the event that the MBOX has a conflict between a floating fill and a floating LOAD hit in the DCache that want to use the same Load bus, the second LOAD will be "FORCED_MISS"ed and the earlier LOAD's Fill data will be returned.

1.2.9.4.7 DRAINT_Stall

A number of conditions, including issuing the DRAINT or CALL_PAL instructions, or entering PALMode due to a TRAP will cause a DRAINT operation. When a DRAINT occurs, the DRAINT_FLAG is set in the Issue stage of the IBOX. All issuing is stalled while this bit is set. All previously issued instructions must complete to the S7 trap point prior to resuming issuing. In addition, the IMUL and FDIV units must finish any current operation before the DRAINT_FLAG is cleared. Conditions causing the DRAINT_FLAG to be set and the requirements for clearing it are listed in Section 1.2.9.13.

1.2.9.4.8 MB_STALL

A number of conditions, including issuing the MB instruction, will set the MB_FLAG in the Issue stage of the IBOX. When this flag is set, any slotted instruction that requires the MBOX will be stalled. Instructions requiring the MBOX are listed in Figure 1–21. Conditions causing the MB_FLAG to set and reset are listed in Section 1.2.9.12.

1.2.9.4.9 MB_MB_STALL

If a second MB instruction occurs while the MB_FLAG is set, all four instructions are unconditionally stalled. This allows an easy method for software (i.e. DVT code) to freeze the machine. Unfreezing the machine occurs when the first MB is cleared. Since clearing the MB can be accomplished from the pins, this allows for hardware to synchronize stopping and starting instruction issue. When the first MB is cleared, the second MB instruction will issue and will set the MB_FLAG allowing any subsequent non-MBOX instructions to issue. A second clear of the MB_FLAG must occur to remove the second of the MB instructions to allow MBOX instructions to issue.

1.2.9.5 Instruction Stall

Each of the four slotted instructions has four possible stall conditions associated with it: a DEST-SOURCE stall, a DEST-DEST stall, a Current-Issue stall, and a Resource stall. These four lines are ORed together for each instruction to create a single stall indication for each of the four instructions. If a valid instruction has not been slotted (as determined by the **I_IBS%E0_VALID_2A** etc. lines), the corresponding stall condition is not asserted.

If any of the four instructions signals that a STALL is necessary, the **I%Z_STALL_3B** signal is raised and sent to the earlier stages of the IBOX and to the other boxes. The Issue stage will have latched the opcodes and other information sent from the slot stage into a set of conditional holding latches. If a stall occurs, these latches will not update but will hold the instruction context for all four of the current instructions. Local valid bits will be cleared for those instructions that have been issued (as indicated by the **I%Z_ISSUE_E0_4A**, etc. lines). In the next cycle, the Issue stage will attempt to issue the remaining un-issued instructions. The Dirty Logic will assert **I%Z_STALL_3B** for as many cycles as necessary to resolve all the conflicts and successfully issue each instruction in the block. Each of the four **I%Z_ISSUE_XX_4A** lines will only assert for a single cycle in each block. The cycle when **I%Z_ISSUE_XX_4A** asserts, is the cycle that a particular instruction was actually released to the pipe.

1.2.9.6 Serialization

Assuming that one or more of the instructions was stalled, a serialization step is necessary to block the issue of all instructions that logically followed the stalling instruction even if there are no register or resource conflicts on those following instruction(s). This is necessary to preserve write ordering of the result data. This step is performed during 3B with the result of creating four lines (**I%Z_ISSUE_E0_4A, I%Z_ISSUE_E1_4A, I%F_ISSUE_FA_4A, I%F_ISSUE_FM_4A**). These signals indicate to the IBOX, EBOX, FBOX, and MBOX which of the four instructions are able to issue. These signals will be latched and will travel down the the dirty logic datapaths as valid bits to qualify future stall and bypass calculations. The signal **I%Z_ISSUE_E0_4A** is also sent to the FBOX as **I%F_ST_ISSUE_4A** to indicate that a floating store has been issued in the E0 pipe.

1.2.9.7 Bypasses

In addition to coordinating the release of instructions to the various execution units, the Issue stage is responsible for steering the correct data in the execution pipelines through a number of built in bypasses. The IBOX Issue stage detects that a destination register matches one (or more) of the source registers using the same datapaths used for the dirty checks. Bypasses are signaled to the EBOX/FBOX regardless of the state of the **I%Z_ISSUE_XX_4A** and **I%Z_STALL_3B** lines. If a stall is occurring, the source register numbers used in the bypass calculations will be recirculated and the correct bypasses will be updated and signaled in the subsequent cycles until the instruction requiring the bypass is actually able to execute.

1.2.9.7.1 EBOX Bypasses

In the two EBOX pipes, each stage's (S4-S7) destination data is bypassable back to replace any of the four operand registers (E0A, E0B, E1A, E1B). In addition, the S4 stage of each pipe has two possible destination registers, the adder output or the logic box output. (RPP—Is this still true?, does the dirty logic need to pick the LU or ADDer or will the EBOX figure this out for itself??) There are 2 pipes * (4 + 1 stages) * 4 source registers = 40 different bypass possibilities.

For the case of two cycle EBOX instructions like SHIFTs, an S4 bypass may be asserted, but the DEST-SOURCE checks will stall without issuing the following instruction since the data is not actually available from the shifter until S5. Likewise IMUL results are not bypassable until they reach the S5 stage which occurs three cycles following the **E%MUL_DONE_SOON_2A** signal.

It is possible that a particular register destination might exist in more than one stage of the pipe simultaneously. This can be caused, for example, by issuing two instructions that write the same register in back to back cycles. In this case, if the same register was then used as a source in a following cycle, two different bypasses would be indicated. The bypass signals will be prioritized prior to sending them to the EBOX such that the bypass from the logically latest instruction will be used as the source register for the issuing instruction. Since the Current-Issue checks (Section 1.2.9.3.3) will prevent the same destination register from appearing in both pipes in the same cycle at the same stage, it is always possible to select the latest instruction to use as the bypass source. If an instruction uses a literal value rather than a register file location as the B source, the issue stage will detect this and signal the EBOX that the literal field should be used instead of the register file or bypasses. Table 1–4 contains a list of the EBOX bypasses and their relative priorities.

Table 1–4: EBOX Bypass MUX control Signals

Bypass	Priority	Description
E0A Source Bypasses		
I%E_BYP_E0S4_E0A_3B	1	Use the E0 Pipe's S4 result
I%E_BYP_E1S4_E0A_3B	1	Use the E1 Pipe's S4 result
I%E_BYP_E0S5_E0A_3B	2	Use the E0 Pipe's S5 result
I%E_BYP_E1S5_E0A_3B	2	Use the E1 Pipe's S5 result
I%E_BYP_E0S6_E0A_3B	3	Use the E0 Pipe's S6 result
I%E_BYP_E1S6_E0A_3B	3	Use the E1 Pipe's S6 result
I%E_BYP_E0W_E0A_3B	4	Use the E0 Pipe's S7 result
I%E_BYP_E1W_E0A_3B	4	Usethe E1 Pipe's S7 result
I%E_USE_E0A_3B	5	Use the Register File Contents
E0B Source Bypasses		
I%E_USE_E0_LIT_3B	1	Use the Literal Field
I%E_BYP_E0S4_E0B_3B	2	Use the E0 Pipe's S4 result
I%E_BYP_E1S4_E0B_3B	2	Use the E1 Pipe's S4 result
I%E_BYP_E0S5_E0B_3B	3	Use the E0 Pipe's S5 result
I%E_BYP_E1S5_E0B_3B	3	Use the E1 Pipe's S5 result
I%E_BYP_E0S6_E0B_3B	4	Use the E0 Pipe's S6 result
I%E_BYP_E1S6_E0B_3B	4	Use the E1 Pipe's S6 result
I%E_BYP_E0W_E0B_3B	5	Use the E0 Pipe's S7 result
I%E_BYP_E1W_E0B_3B	5	Usethe E1 Pipe's S7 result
I%E_USE_E0B_3B	6	Use the Register File Contents
E1A Source Bypasses		
I%E_BYP_E0S4_E1A_3B	1	Use the E0 Pipe's S4 result
I%E_BYP_E1S4_E1A_3B	1	Use the E1 Pipe's S4 result
I%E_BYP_E0S5_E1A_3B	2	Use the E0 Pipe's S5 result
I%E_BYP_E1S5_E1A_3B	2	Use the E1 Pipe's S5 result
I%E_BYP_E0S6_E1A_3B	3	Use the E0 Pipe's S6 result
I%E_BYP_E1S6_E1A_3B	3	Use the E1 Pipe's S6 result
I%E_BYP_E0W_E1A_3B	4	Use the E0 Pipe's S7 result
I%E_BYP_E1W_E1A_3B	4	Usethe E1 Pipe's S7 result
I%E_USE_E1A_3B	5	Use the Register File Contents
E1B Source Bypasses		

Table 1–4 (Cont.): EBOX Bypass MUX control Signals

Bypass	Priority	Description
I%E_USE_E1_LIT_3B	1	Use the Literal Field
I%E_BYP_E0S4_E1B_3B	2	Use the E0 Pipe's S4 result
I%E_BYP_E1S4_E1B_3B	2	Use the E1 Pipe's S4 result
I%E_BYP_E0S5_E1B_3B	3	Use the E0 Pipe's S5 result
I%E_BYP_E1S5_E1B_3B	3	Use the E1 Pipe's S5 result
I%E_BYP_E0S6_E1B_3B	4	Use the E0 Pipe's S6 result
I%E_BYP_E1S6_E1B_3B	4	Use the E1 Pipe's S6 result
I%E_BYP_E0W_E1B_3B	5	Use the E0 Pipe's S7 result
I%E_BYP_E1W_E1B_3B	5	Usethe E1 Pipe's S7 result
I%E_USE_E1B_3B	6	Use the Register File Contents

1.2.9.7.2 FBOX Bypasses

The FBOX bypasses are somewhat simpler since FBOX results are only bypassable at the register file write stage. In the FBOX, the FA result can be bypassed back as any one of the four source registers or as data for the store port. The FM result can be bypassed back similarly. In addition, the two Load buses can be bypassed to any of the four source registers or the store port. Therefore we have 4 destination registers * 5 source registers = 20 bypasses.

The Current-Issue checks (Section 1.2.9.3.3) will prevent the FA and FM pipes' destination register numbers from matching so that only a single bypass can be asserted for each of the 4 source registers slotted to the FBOX pipes due to the execution pipes. The DEST-DEST checks (number 4 in Section 1.2.9.3.2) will prevent either of the LOAD bus destination registers in 7A from matching either the FA or FM output destination register numbers currently in 9A. Therefore no priority encoding of bypass signals to the FBOX is necessary. Table 1–5 contains a list of the FBOX bypasses.

Table 1–5: FBOX Bypass MUX control Signals

Bypass	Description
FAA Source Bypasses	
I%F_FA_FAA_3B	Use the Adder results
I%F_FM_FAA_3B	Use the Multiplier results
I%F_LD0_FAA_3B	Use the data on Load Port 0
I%F_LD1_FAA_3B	Use the data on Load Port 1
FAB Source Bypasses	
I%F_FA_FAB_3B	Use the Adder results

Table 1–5 (Cont.): FBOX Bypass MUX control Signals

Bypass	Description
I%F_FM_FAB_3B	Use the Multiplier results
I%F_LD0_FAB_3B	Use the data on Load Port 0
I%F_LD1_FAB_3B	Use the data on Load Port 1
FMA Source Bypasses	
I%F_FA_FMA_3B	Use the Adder results
I%F_FM_FMA_3B	Use the Multiplier results
I%F_LD0_FMA_3B	Use the data on Load Port 0
I%F_LD1_FMA_3B	Use the data on Load Port 1
FMB Source Bypasses	
I%F_FA_FMB_3B	Use the Adder results
I%F_FM_FMB_3B	Use the Multiplier results
I%F_LD0_FMB_3B	Use the data on Load Port 0
I%F_LD1_FMB_3B	Use the data on Load Port 1
STORE Port Bypasses	
I%F_FA_ST_3B	Use the Adder results
I%F_FM_ST_3B	Use the Multiplier results
I%F_LD0_ST_3B	Use the data on Load Port 0
I%F_LD1_ST_3B	Use the data on Load Port 1

1.2.9.8 Register File Writes

The IBOX Issue stage provides both the EBOX and FBOX register files with write addresses and write strobes. The destination register numbers are piped in the two dirty datapaths and the accompanying valid bits are used to create the address and strobes respectively.

Destination register numbers for operates or loads entering S6 of the integer dirty datapath are encoded and sent to the EBOX register file over the **I%E_W0_ADDR_6A<4:0>** and **I%E_W1_ADDR_6A<4:0>** lines. The corresponding valid bits are buffered and sent as **I%E_W0_EN_7A** and **I%E_W1_EN_7A**. If the valid bit(s) have been cleared by either not having issued an instruction in that cycle or by a pipe abort condition, no write strobe is issued and therefore the register file is not updated. Also, the write strobe is not issued if the instruction writing the register file was a LOAD that missed in the D-Cache.

Destination register numbers entering S8 of the floating dirty datapath are encoded and sent to the FBOX register file over the **I%F_FA_ADDR_8A<4:0>** and **I%F_FM_ADDR_8A<4:0>** lines. The corresponding valid bits are buffered and sent as **I%F_WE_FA_9A** and **I%F_WE_FM_9A**. As with the EBOX registers, if the valid bit(s) have been cleared, the register is not updated.

The FBOX register file also has two separate write ports for Load data. The Issue stage will send register addresses for these two ports over the **I%F_LD0_ADDR_6A<4:0>** and **I%F_LD1_ADDR_6A<4:0>** lines. These are 6A signals to support the 7B write from the MBOX. For floating Loads that HIT in the DCache, the address is provided by the IBOX Issue logic by decoding the instruction. For FILLs, the MBOX will provide the register destination number to the Issue stage in 5A (see Section 1.2.9.9). This destination number will be siloed, muxed inplace of the IBOX generated address, and sent to the FBOX in 6A. The write strobes are **I%F_WE_LD0_7A** and **I%F_WE_LD1_7A**. These strobes are created by examining the **M%I_DC_HIT_E0_6A, M%I_DC_HIT_E1_6A, M%I_FILL_VALID0_5B**, and **M%I_FILL_VALID1_5B** lines as appropriate to the pipe and hit/miss status of the instruction.

1.2.9.9 LOADs and STOREs

LOAD and STORE instructions require some additional effort and special casing due to their uncertain latency (LOADs), special pipe requirements (floating LOADs, and STOREs), and because of MBOX requirements.

1.2.9.9.1 Additional LOAD Checks

The destination register of a load instruction will be locked against all reads/writes for two cycles (S4 and S5). This means that any following instruction will not be allowed to issue if it uses the Load's destination register as either a source or a destination.

If an issuing (S3) instruction references the destination register of a load that is entering S6 as either it's destination or one of its two sources, the Load is assumed to hit in the D-Cache and the appropriate bypasses and issue lines are asserted. If it turns out that the Load misses in the D-Cache, an LDU-Replay occurs, see Section 1.2.9.17.

When a load misses in the D-Cache, the destination register is added to a list of outstanding load misses held in the dirty datapaths. Any register locked by this list will cause stalls when an issuing instruction attempts to reference the register as either a source or a destination.

1.2.9.9.2 Floating Loads

Floating Loads present a special problem since they are integer instructions that return results to the floating register file. Therefore, for floating loads, the dirty logic must check the base address (an integer register) using the DEST-SOURCE checks of the integer dirty pipe (this is the same as for integer LOADs). However, the destination register (a floating register) must be checked using the DEST-DEST checks of the floating dirty pipe. The current issue checks that must be performed are a hybrid of the integer and floating current issue checks. The base address register (Rb) must not conflict with the other integer instruction's destination, if it does, the Current Issue check will stall the floating load if it logically follows the conflicting integer instruction. The floating load's destination register (Fa) must not conflict with either of the floating execute instruction's destination registers. The logically latest of the floating load or the conflicting floating operate will be stalled by the Current Issue checks to maintain register file write ordering (see Section 1.2.9.3.3). If the floating load's destination register conflicts with one of the floating operate's sources, and the floating load logically preceeds the operate, then the floating operate is stalled. Therefore a condition resulting in a stall of a Floating LOAD could be generated in either the integer or floating dirty datapath and the floating load (while technically an integer instruction by issue pipe) can cause stalls of floating point instructions.

1.2.9.9.3 Floating Stores

Like Floating LOADs, Floating STOREs are integer instructions that use the floating register file. The base address register of a floating store (Rb) is checked using the integer dirty datapath for DEST-SOURCE and Current Issue conflicts similar to the Floating Load case described above. The store data register (Fa) is checked using the floating dirty datapath for DEST-SOURCE conflicts and Current Issue conflicts. Since the Fa register is a source rather than a destination, the floating store cannot cause other instructions to stall (in either datapath). However, conditions in either the integer or the floating point dirty datapaths, can lead to stalls of the floating store.

1.2.9.9.4 LOAD HITs

Register file destination numbers for LOADs that HIT will be encoded from the destinations being piped along in the dirty datapaths. These encoded values will be sent to the register files in 6A as described in Section 1.2.9.8 prior to learning of their hit status. The Hit signal(s) (**M%I_DC_HIT_E0_6A** and **M%I_DC_HIT_E1_6A**) will arrive at the Issue logic in late 6A and will be used to qualify the write strobes sent in 7A. The "E0" hit line will indicate that data is returning to either the E0 integer pipe over the **M%E_LD_DATA0_6A_H<31:0>** bus or to the floating register file's LD0 write port over the "0" (rpp-fill in the name of the bus when I know what it is) bus. The "E1" hit line operates in a similar manner for the E1 integer pipe (**M%E_LD_DATA1_6A_H<31:0>**) and the LD1 port of the floating register file. The IBOX issue logic will determine which register file (E or F) is to be written in each case by examining siloed bits associated with each pipe which indicate whether the current S6 instruction is a LOAD and whether it is an integer or a floating point.

1.2.9.9.5 LOAD Fills

When a load misses in the D-Cache, the absence of the HIT signal(s) will prevent a write strobe from being issued to the appropriate register file. Instead the register destination number will be added to the list of LOAD_MISSes in the appropriate dirty logic datapath. This list is used in the DEST-SOURCE and DEST-DEST checking as described in Section 1.2.9.3. When FILL data is returned from the MBOX, the appropriate register is removed from the list.

The register addresses for returning fill data are provided by the MBOX in 4B over the **M%I_FILL_RNUM0_4B<6:0>** and **M%I_FILL_RNUM1_4B<6:0>** lines. The "0" bus is used for addresses corresponding to data being returned to either the E0 EBOX pipe (over **M%E_LD_DATA0_6A<31:0>** or to the LD0 port of the floating register file (rpp-fill in name). The "1" bus likewise corresponds to the E1 EBOX pipe and the LD1 port of the floating register file. Unlike with HITs where the MBOX must return the data using the same pipe in which the instruction was issued ("0" or "1"), FILL data may be returned to either of the two E/F box pipes/ports. Fill data is marked valid by the FBOX by the assertion of the lines **M%I_FILL_VALID0_5B** and **M%I_FILL_VALID1_5B**.

The IBOX Issue stage will determine whether the fill data is bound for the E or F box by examining bit 5 of the MBOX supplied register address. This bit will be a "1" for an FBOX fill or a "0" for an EBOX fill. Bit 6 of the rnum is the "Pal_Shadow" bit. It is a "1" for fill addresses corresponding to a PAL_SHADOW register rather than a normal EBOX register. Bit 6 is ignored, if Bit 5 is set, corresponding to an FBOX fill.

When a valid fill is sent from the MBOX (i.e. a valid register destination and "fill_valid" are present), the IBOX will multiplex the MBOX specified register destination address into the appropriate dirty logic datapath and then drive out the correct register address in the same manner as done for D-Cache Hits.

1.2.9.9.6 EBOX LD MUX

LOAD data is multiplexed into the normal EBOX result SILO at the S6 stage. The IBOX Issue stage signals the EBOX whether it should SILO it's S5 results into S6 or accept MBOX data. This is accomplished by the two lines **I%E_USE_LD0_6A** and **I%E_USE_LD1_6A**. Since these are 6A signals, it is not known whether the MBOX data multiplexed will be valid. The EBOX register file will not be updated in S7 if MBOX FILL data was not valid or not a HIT. The Issue stage will assert the appropriate "use_ldx" line when an integer LOAD advances into S6. It will assert both lines in response to the CBOX allocate cycle command (**C%I_ALLOC_CYCLE_2B**).

1.2.9.10 EBOX IMUL MUX

The EBOX integer multiplier indicates that it has finished it's operation and has data available by asserting the **E%I_MULL_DONE_SOON_2A** line. As mentioned in Section 1.2.9.4.2, the E0 instruction is stalled for a cycle, to allow this instruction to merge into the E0 pipe at the S5 stage three cycles later. However, if a FILL_STALL (Section 1.2.9.4.6) is signaled by the CBOX (**C%I_ALLOCATE_CYCLE_2B**) at the same time, the FILL_STALL has priority. Therefore the IMUL unit cannot mux it's results into the E0 pipe. The final stage of the IMUL unit contains a static latch will hold the data until the next IMUL finishes. Therefore this data is available for multiplexing into the E0 pipe at any point until the next IMUL is issued and completes. The IBOX Issue stage will select the IMUL result by asserting the **I%E_SEL_MUL_5B** signal in the first cycle that the FILL_STALL does not occur. In this cycle, the E0 pipe will be reserved by the IMUL_DONE_SOON stall so that a data collision does not occur.

The floating divider does not present a similar problem since FILLs are returned to the FBOX register file via separate dedicated write ports.

(rpp— Is there a potential problem here if the CBOX allocates N cycles in a row???)

1.2.9.11 Conditional Move

The integer and floating point CMOV instructions are jointly performed by the Issue stage of the IBOX and the E/FBoxes. The Issue stage issues the instruction and signals the appropriate box with the **I%Z_ISSUE_XX_4A** lines. The E/FBoxes, assume that the CMOV will be successful and copy the Rb data to the destination. At the same time the "conditional" test on Ra is carried out. The sucess of this test is indicated to the IBOX Issue stage using the lines **E%I_KILL_CMOV0_4B**, **E%I_KILL_CMOV1_4B**, and **F%I_KILL_CM_5A_H**. If one of these signals asserts, it indicates that the conditional test failed. The Issue stage will clear the appropriate valid bit associated with the instruction thus "killing" it. If the test is sucessful, the instruction is allowed to proceed down the pipeline (being bypassed if necessary) and eventually written into the register file.

1.2.9.12 Memory Barriers

Issuing any of the following instructions will set the MB_FLAG in the Issue stage which will prevent further issuing of MBOX instructions:

Table 1–6: Instructions Setting the MB_FLAG

Opcode(hex)	Mnenomic	Description
18.4xxx	MB	Memory Barrier
2E	STL_C	Store Long Conditional
2F	STQ_C	Store Quad Conditional
18.Cxxx	RPCC	Read Process Cycle Counter
18.Exxx	RC	READ and CLEAR
18.Fxxx	RS	READ and SET
1F.	HW_ST	Hardware Store, only sets the MB_FLAG if the "_C" option is present

The MB flag will remain set until the MBOX clears it via the **M%I_MB_CLEAR_2B** signal. (rpp– Does this still come solely from the MBOX?? or does the CBOX ack MBs while the MBOX acks the rest?)

Need to work out what happens during the various pipe abort conditions -rpp

While the MB_FLAG is set, all MBOX associated instructions are stalled at the Issue stage. Since EV5 does not issue instructions out-of-order, this first stalled MBOX instruction will stall all further instruction issue until the MB_FLAG is cleared by the MBOX. The following table lists instructions that will be stalled while the MB_FLAG is set.

Table 1–7: MBOX Instructions stalling while MB_FLAG is set

Opcode	Mnemonic	Description
0B	LDQ_U	Load Quad Unaligned
0F	STQ_U	Store Quad Unaligned
18.4xxx	MB	Memory Barrier—2nd MB stalls the machine see Section 1.2.9.3
18.8xxx	FETCH	Fetch Instruction
18.Axxx	FETCHM	Fetch with Modify Intent
18.Cxxx	RPCC	Read Process Cycle Counter
18.Exxx	RC	Read and Clear
18.Fxxx	RS	Read and Set
19	HW_MFPR	Hardware Move from Processor Register –(rpp-Should this stall under MB for both I,M boxes, just for the MBOX, or for neither?)
1B	HW_LD	Hardware Load
1D	HW_MTPR	Hardware Move to Processor Register—(rpp see note on HW_MFPR)
1F	HW_ST	Hardware Store
20	LDF	Load F
21	LDG	Load G
22	LDS	Load S
23	LDT	Load T

Table 1–7 (Cont.): MBOX Instructions stalling while MB_FLAG is set

Opcode	Mnemonic	Description
24	STF	Store F
25	STG	Store G
26	STS	Store S
27	STT	Store T
28	LDL	Load Long
29	LDQ	Load Quad
2A	LDL_L	Load Long Locked
2B	LDQ_L	Load Quad Locked
2C	STL	Store Long
2D	STQ	Store Quad
2E	STL_C	Store Long Conditional
2F	STQ_C	Store Quad Conditional

1.2.9.13 DRAINT

The Issue stage contains a one bit DRAINT_FLAG which is used to indicate that a DRAINT operation (explicit or implicit) is occuring. No instructions are issued while the DRAINT_FLAG is set.

1.2.9.13.1 Setting the DRAINT_FLAG

The DRAINT_FLAG is explicitly set by issuing the DRAINT instruction. It is also set (implicitly) by the issue of a CALL_PAL instruction or by the dispatch of a TRAP to the appropriate PALCode routine. The DRAINT_FLAG is set when the DRAINT or CALL_PAL is actually issued. This implies that it is not set until the instruction is next in line to issue. Therefore if a prior instruction in the issue block causes a STALL, the DRAINT_FLAG is not set in that cycle. DRAINTs and CALL_PALs may multiple issue with other instructions however by definition they will be the last instruction in the multiple issue block. When a DRAINT is issued, subsequent (and therefore) unissued instructions in the S3 Issue block will be stalled immediately (CALL_PALs will be the last valid instruction slotted in an issue block since they change the PC). Neither DRAINT or CALL_PAL instructions will set any valid bits in the dirty logic pipelines when they issue, the only effect as far as the issue stage is concerned is the setting of the DRAINT_FLAG. When a TRAP is posted, the change from "native" mode to PALMode as indicated by bit 0 of the PC (RPP-get the signal name from NITAL), an implicit DRAINT is executed. All instruction issuing is stalled in that cycle.

1.2.9.13.2 Clearing the DRAINT_FLAG

Each cycle, the valid bits in both the floating and integer dirty datapaths are examined. The presence of a valid bit indicates that a valid instruction is executing in one of the pipelines. The presence of any set valid bit corresponding to stages 4-7 of either of the EBOX pipes or to stages 4-9 of either of the FBOX pipes means that the DRAINT operation has not completed. (NOTE: The valid bits for instructions that logically follow a TRAP are cleared when the TRAP is posted—see Section 1.2.9.15.1). In addition, if a valid instruction is executing in either the Integer Multiplier or the Floating Divider, the DRAINT operation is not complete. The DRAINT_FLAG is cleared at the end of the cycle following the cycle when all of the valid bits in the two dirty logic pipelines, the IMUL, and the FDIV, become clear. This means that any TRAPS associated with the executing instructions have reached their reporting points and the TRAP logic in the IBOX has had a cycle to take the appropriate response if a TRAP has occurred.

1.2.9.13.3 DRAINT Latency

Instruction issuing is resumed at the beginning of the cycle following the cycle in which the DRAINT_FLAG is cleared. There is an explicit SET overrides RESET logic for the DRAINT_FLAG such that a minimum DRAINT operation appears to require two cycles. In the first cycle, the DRAINT (or CALL_PAL) instruction is issued and the DRAINT_FLAG is set. This corresponds to the initial cycle that a TRAP is posted. In the second cycle, if all the valid bits are clear, the DRAINT_FLAG is cleared. However, this does not occur until late in the cycle at which point a STALL decision for that cycle has already been made. Therefore instruction issuing resumes at the beginning of the third cycle.

Typically the minimum DRAINT operation does not occur. The pipeline must have been fully drained prior to issuing the DRAINT (or CALL_PAL) to achieve the minimum latency. For well scheduled integer code (without IMULs), the typical latency of a DRAINT is 5 or 6 cycles. This corresponds to the following:

0. Dual Issue of an integer operate and the DRAINT.
1. S4 of the Integer Operate
2. S5 of the Integer Operate
3. S6 of the Integer Operate
4. S7 of the Integer Operate (Overflow is reported to the TRAP logic)
5. The DRAINT_FLAG is cleared late in the cycle
6. Issuing resumes.

The latency is reduced to 5 cycles, if the DRAINT does not dual issue with an integer operate.

Typical DRAINT latency on floating point code is two cycles longer (i.e. 7 or 8 cycles) since the floating pipe is two cycles longer than the integer pipes.

What do we do about error conditions,aborts, etc???—rpp

1.2.9.14 Illegal/Reserved Opcodes

(RPP- It is currently a little unclear how much of this is detected by the issue stage and how much by the Slot stage. This description provides the details of having the Issue Stage do all the detecting/reporting of illegal opcodes. The final implementation may have some/all of the decoding done in the slot stage with the issue stage simply providing the issue timing for the appropriate signals to the TRAP logic).

The Issue Stage of the IBOX detects and reports several kinds of illegal/reserved opcodes to the TRAP logic. The Issue Stage reports on these conditions at the same time that the offending instruction issues. This helps preserve the "exactness" of the potential exceptions generated by these conditions. The TRAP logic must abort the instruction (and any subsequent instructions (Section 1.2.9.15.1) by generating the appropriate TRAP. The following conditions are detected and reported.

1.2.9.14.1 Opcodes Reserved to Digital

These opcodes (01-07, 0A, 0C-0E, 14, and 1C) are routed by the Slot stage to Pipe E0. The Issue Stage will assert the signal **I_ISS%OPCDEC_4A** to the TRAP logic when one of these opcodes "issues". In addition, the relative position of this instruction within the current block of four will be indicated by the **I_ISS%E0_POS_4A<1:0>** lines.

1.2.9.14.2 PAL Instruction in "native" mode

The PAL opcodes (19, 1B, 1D, 1E, 1F) are detected by the instruction decoder in the Issue stage. When one of these opcodes reaches the issue stage and does not encounter any stalls, the instruction issues and the appropriate registers are marked dirty. If the machine is operating in "native" mode when the instruction issues as indicated by bit 0 of the S3 PC (rpp-check with Nital for the correct signal) instead of PALMode, the signals **I_ISS%OPCPAL_E0_4A** and/or **I_ISS%OPCPAL_E1_4A** are asserted to the TRAP logic. The instruction will issue during the 3B/4A phase, so the TRAP logic must signal a valid TRAP back to the Issue stage in order to clear the valid bits associated with this instruction and to free up the dirty registers. The position of the OPCPAL instruction within the current block can be determined by examining the **I_ISS%E0_POS_4A<1:0>** and **I_ISS%E1_POS_4A<1:0>** lines.

1.2.9.14.3 Priviledged CALL_PALs

When a CALL_PAL with a priviledged function field is issued, the Issue stage signals the TRAP logic using the **I_ISS%PRIV_PAL_4A<1:0>** line. The position of this instruction may be determined by examining the **I_ISS%E1_POS_4A<1:0>** lines. Since a CALL_PAL does not dirty any registers, no registers are dirty locked by the issuing of this instruction. The Issue stage will indicate that a priviledged CALL_PAL has been executed regardless of the current mode of the machine. The TRAP logic will determine if a TRAP is required (i.e. not in KERNAL mode).

1.2.9.14.4 Illegal CAL_PAL functions

When a CALL_PAl with a function field outside of the legal range (0-3F and 80-BF are the legal range), is issued, the Issue stage signals the TRAP logic with the **I_ISS%BAD_PAL_4A** line. The position of this instruction may be determined by the **I_ISS%E1_POS_4A<1:0>** lines.

1.2.9.14.5 Floating Point

(rpp–I need to check on this... Does the issue stage need to do anything?)

1.2.9.15 Aborting Instructions

1.2.9.15.1 TRAPs, REPLAYs, and INTERRUPTS

When an instruction is issued, the **I%ISSUE_XX_4A** lines are latched and travel down the appropriate dirty logic pipeline as valid bits. (Section 1.2.9.6). The TRAP logic will send the signals **I_TRP%ABORT_S2_THRU_S7_B_H** or **I_TRP%ABORT_S2_THRU_S5_B_H** when an interrupt, error, replay, or trap occurs. These signals will be used to clear the siloed valid bits in the dirty datapaths. In addition, issuing is suspended for two cycles. This allows time for the earlier stages of the IBOX to cycle to the correct state without releasing any additional instructions into the execution units.

The actual valid bits cleared depends upon which of the TRAP signals is indicated and upon the position information sent from the TRAP logic sent over the (rpp-get the name from Vidya) lines. For S7 traps, (indicated by the **I_TRP%ABORT_S2_THRU_S7_B_H** signal), all valid bits in stages 3-6 are cleared. In addition, valid bits for instructions in S7 that follow the TRAPping instruction cleared. For S5 traps, (**I_TRP%ABORT_S2_THRU_S5_B_H**) only the valid bits in S3, S4, and for S5 instructions following the TRAPping instruction are cleared.

Since the register file writes in the EBOX (FBOX) use an S7 (S9) version of these siloed valid bits as the write strobe, EBOX and FBOX operates for the aborted instructions are effectively killed when the valid bits are cleared. The FBOX also uses the write strobe(s) to update the FPCSR register with the exception status. If the write strobe doesn't occur, the instruction is assumed to have been aborted by a previous exception and the register is not updated. Since some TRAPS are reported in S7, the EBOX register file write strobes cannot be aborted in time. Therefore the MBOX (the source of the S7 traps) will send a pair of register file write aborts to the EBOX register file to abort the S7 instructions.

Instructions executing in the integer Multiplier (and the floating Divider) require a special abort sequence. When an instruction is issued to either of these units, it sets a timer in the Issue stage. The timer is used to determine the state of the inprogress instruction relative to the trapping event. If the instruction in one of these two units logically follows the faulting instruction in execution order, the unit must be aborted. This state is detected by the issue logic and an abort signal is sent (**I%E_ABORT_IMUL, I%F_ABORT_FDIV**) to the appropriate box. In addition, the IMUL_DEST (or FDIV_DEST) register in the dirty logic is cleared to free up the unit for future instructions. These abort signals may be sent in any of the first four cycles of the IMUL/FDIV instruction's execution.

The Trap logic in the IBOX will be responsible for driving the appropriate signals to the MBOX to abort any issued MBOX instructions. The valid bits for any instructions following the faulting instruction will have been cleared in the Issue stage so that no data is expected to be returned to the register file(s) for these instructions. LOADs that have missed in the D-Cache must have logically preceeded any trapping instruction and therefore will be allowed to complete when the data is returned by the MBOX. The list of outstanding misses will not be cleared until the fill data returns, therefore PALCode may experience some unanticpated stalls if the code attempts to use a register that is still locked from prior loads.

1.2.9.15.2 ERROR aborts

Certain conditions, like RESET, expiration of the S3_STALL_TIMEOUT counter, and (others TBD-rpp) will cause all of the valid bits in the Issue logic to be cleared These conditions will also cause the oustanding miss register list to be cleared and aborts to be sent to the IMUL and FDIV units.

1.2.9.16 Special Stuff

RPP-This section will be a collection of all the other hoops that the Issue stage jumps through that don't fit into the other sections.

1.2.9.17 LOAD MISS-AND-USE Replay

This section is waiting some clarification of the latest set of changes prior to being included.

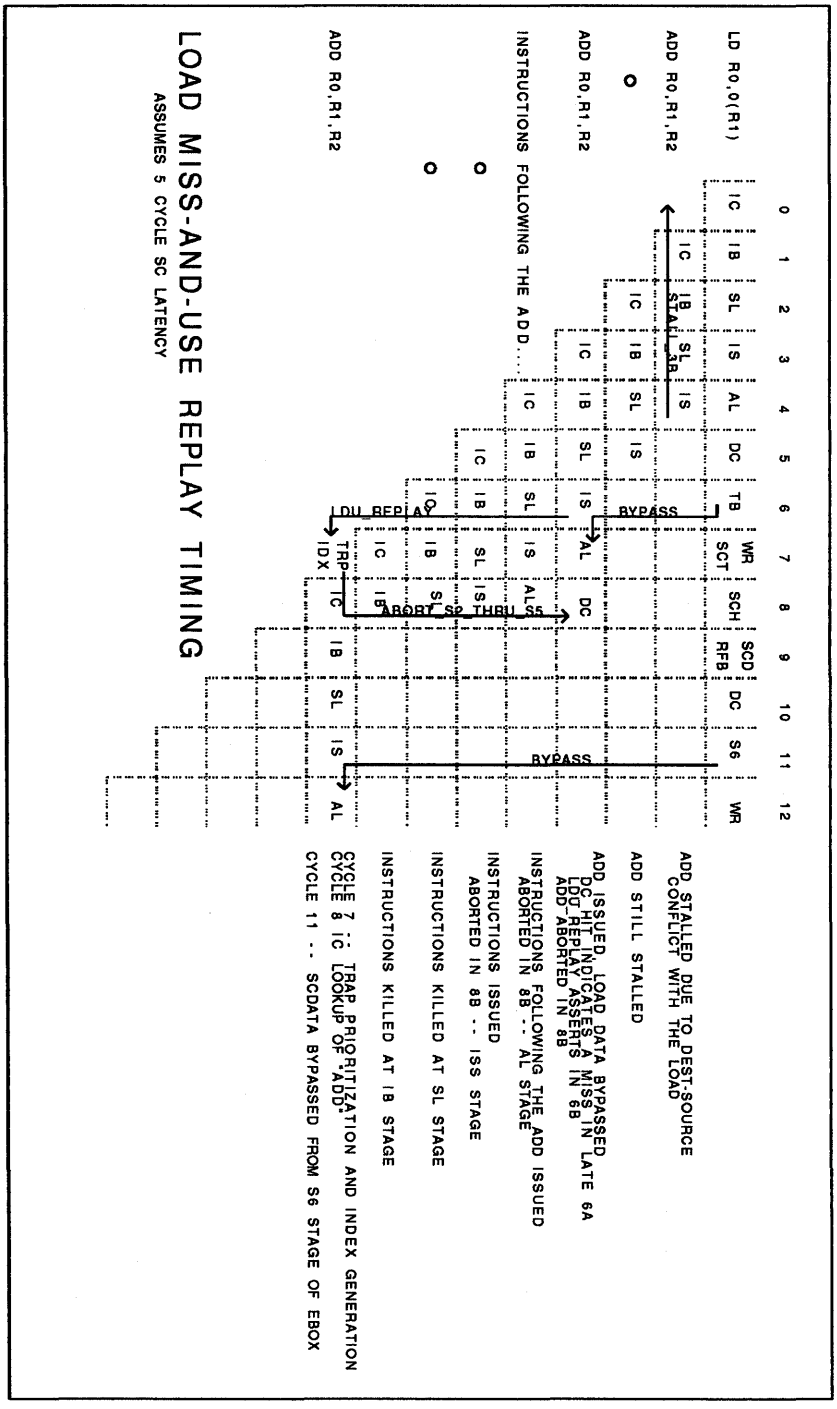
The DCache HIT signals from the MBOX (**M%I_DC_HIT_E0_6A** and **M%I_DC_HIT_E1_6A** do not arrive in the Issue logic early enough for the IBOX to signal the correct stall or bypass if a current issuing S3 instruction has a data dependency on one of the LOADs currently in 6A. To avoid adding an additional cycle of latency to the LOAD-HIT path, special LDU_REPLAY logic has been built into the integer dirty datapath. In the case described, where an issuing instruction a source register that matches the destination of a LOAD currently is S6, the LOAD is assumed to HIT and the instruction is issued. If the guess was correct, everything proceeds as normal. If the guess proves incorrect, an LDU_REPLAY is asserted. The signal **I_ISS%LDU_REPLAY_3B** is asserted to the EPC and TRAP sections of the IBOX. In addition, the position in the block of 4 of the instruction that was incorrectly issued is sent on the lines **I_ISS%LDU_OFFSET_4A<1:0>**. An LDU_REPLAY also occurs if one of the issuing instructions has a DEST-DEST conflict with the LOAD that missed in the DCache.

The TRAP logic will abort the Issue stage pipe at the S5 TRAP time assuming that no higher priority TRAP occurs. This will cause the incorrectly issued instruction and all instructions issued in it's shadow to be aborted.

The EPC section will create the PC of the incorrectly issued instruction by appending together the PC of the issue block and the offset. This PC will be sent to the IDX section which will start a new ICACHE access at the point of the incorrectly issued instruction.

The incorrectly issued instruction will arrive back at the issue point 5 cycles later just as the data is being returned from the SCache (if the SCache data was a Hit). If the SCache MISSes, the instruction is stalled until the data arrives. Figure 1–23 shows a timing diagram for the LDU_REPLAY path.

Figure 1–23: LOAD MISS-AND-USE Replay Timing



1.2.9.18 PAL Shadow Support

The EBOX has an additional 8 "PAL_SHADOW" registers that can be enabled in PAL mode to replace registers 8-15 of the normal EBOX register file. Mapping of the PAL_SHADOW bank is enabled when the machine is in PALMode and the ICSR<SDE> bit is set. PALMode is determined by examining bit 0 of the 3A PC (rpp-check with Nital for the correct version).

1.2.9.18.1 EBOX Register File Control

The EBOX register file selects the PAL_SHADOW bank registers for reads by examining the **I%E_RD_PAL_SHADOW_2A_H** line which is created by ANDing the PALMode and SDE bits together. If this signal is active, all four source operands will be read from the PAL_SHADOW bank if they are in the PAL_SHADOW's range, otherwise they will be read from the normal EBOX registers.

The Issue Stage will silo the **I%E_RD_PAL_SHADOW_2A_H** bit down the dirty logic datapath with the valid and position bits. The siloed version of this signal is used to select the PAL_SHADOW bank for register file writes. Two signals are generated in S6 to control the bank selection for register file writes, **I%E_W0_PAL_SHADOW_6A** and **I%E_W1_PAL_SHADOW_6A**. These signals select the PAL_SHADOW bank for the W0 and W1 write ports respectively. If one of these signals is active, and the register address falls in the PAL_SHADOW range, the PAL_SHADOW register is updated when the write strobe (**I%E_W0_EN_7A**, etc.) is generated. If the register falls outside the PAL_SHADOW bank, then the normal EBOX register is updated regardless of the state of the **I%E_WX_PAL_SHADOW_6A** lines.

1.2.9.18.2 Dirty Checks for the PAL_SHADOW registers

The Issue Stage peforms the normal dirty and bypass checks (see Section 1.2.9.3) on instructions issued while the PAL_SHADOW bank is enabled. The Issue stage doesn't separate a reference to a PAL_SHADOW register from that to the corresponding normal register. Stalls and Bypasses may be asserted if, for example, one instruction references Shadow R8 and another references "normal" R8. This implies that all writes to PAL_SHADOW registers must complete before switching to the "normal" bank and that all writes to "normal" registers must complete prior to switching to the PAL_SHADOW bank. The only exception to this rule is for LOADs that miss in the DCache. The PAL_SHADOW bit that is siloed with the instruction is used in setting the bits in the DCache Miss register of the integer dirty datapath. In addition, the MBOX reads the signal **I%M_PAL_SHADOW_EN_2A** and stores this bit in the Miss Address File. The MBOX will return this bit as bit 6 of the register address when the fill data returns. Therefore, it is possible to determine the correct register address (PAL_SHADOW or normal) for the EBOX register file. The PAL_SHADOW bit of the write addresses sent to the EBOX register file (**I%E_WX_PAL_SHADOW_6A**) are set to match the MBOX provided bit 6 for FILLs.

1.2.9.18.3 Switching betweeen PAL_SHADOW and NORMAL banks

There are two ways that the PAL_SHADOW bank can be enabled. The first case is switching from "native" mode to PALMode by either issuing a CALL_PAL instruction or by the posting of a TRAP. Switching from "native" mode to PALMode requires that an implicit DRAINT be performed, therefore the pipeline is drained of any operate instructions that have destinations to the "normal" registers. At the point that issuing resumes, the PAL_SHADOW bank is enabled and no outstanding writes (due to operates) exist for the "normal" registers that lie in the PAL_SHADOW address range.

The second way to enable the PAL_SHADOW bank is to set the ICSR<SDE> bit while in PALMode. In this case, up to 5 cycles of operates (S3-S7) may be in the pipeline with destinations that should be to "normal" registers. These operates could cause incorrect data to be bypassed to subsequent instructions that reference PAL_SHADOW registers with corresponding register numbers. Therefore, PALCode must ensure that 5 cycles are allowed between an operate instruction that will write a "normal" register in range R8-R15 and an instruction that will use the PAL_SHADOW version of this register as a source operand.

There are also two ways that the PAL_SHADOW bank can be disabled. The first is to switch from PALMode back to "native" mode. As in the case outlined in the previous paragraph, operates in the two EBOX pipes with PAL_SHADOW destinations could cause incorrect bypasses to source registers that are actually in the "normal" bank. Since PALCode has no control over the application instructions following an HW_REI (switch back to "native" mode), PALCode must avoid writing a PAL_SHADOW register in any of the 5 cycles prior to and including the cycle when a HW_REI issues that switches back to "native" mode. This restriction can also be met by issuing a DRAINT just before the HW_REI.

The second way of disabling the PAL_SHADOW bank is to clear the ICSR<SDE> bit. This case the exact analog of the case where the bit is set. Therefore PALCode must not reference a register in the range R8-R15 within 5 cycles of a write to the corresponding PAL_SHADOW register.

1.2.10 IBOX IPR's and PAL_TEMP registers

The PC bus is used for data movement to and from the IPR's/PAL_TEMP registers.

An IPR/PAL_TEMP is read in S3 and sent to the EBOX on the IBOX PC bus where it is muxed in with the output of the logic box at the end of the E1 pipe. This makes the data bypassable in a cycle.

To write an IPR/PAL_TEMP, the data is read from the EBOX gpr in S3 and sent over the EBOX PC bus in S4. The actual write of the IPR/PAL_TEMP register takes place in S5. There is no siloing of write data to wait out trap shadows. The Ibox will stop writes to the IPR once a trap happens. Given that the latest traps happen in S6 and that IPR writes happen in S5, S6 traps cannot block IPR writes. Only Mbox instructions can trap in S6, which therefore implies that any HW_MTPR that dual issues with and is after an Mbox instruction, will not be aborted if the Mbox instruction is aborted. Floating branch mispredicts which trap in S5 also do not abort IPR writes. are posted in S7.

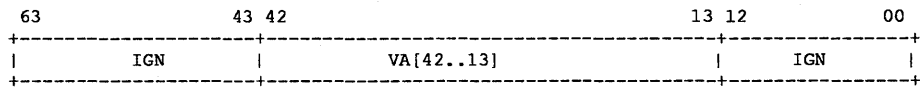
NOTE

Unless explicitly stated, IPRS are not cleared or set by hardware on chip or on timeout reset.

1.2.10.1 ITB_TAG

The ITB_TAG register is a write only register. This register is written by hardware on an ITBMISS/IACCVIO, with the tag field of the faulting VA. To ensure the integrity of the ITB, the TAG and PTE fields of an ITB entry are updated simultaneously by a write to the ITB_PTE register. This write causes the contents of the ITB_TAG register to be written into the tag field of the ITB location, which is determined by a NLU algorithm. The PTE field is obtained from the MTPR ITB_PTE instruction.

Figure 1–24: Istream TB Tag, ITB_TAG

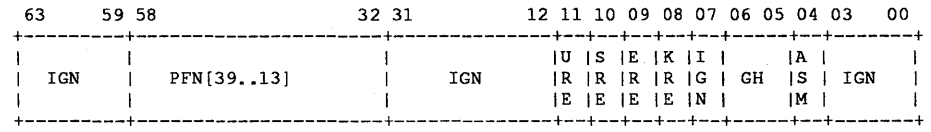


1.2.10.2 ITB_PTE

The ITB_PTE register is a read/write register. A write to this register, writes both the PTE and TAG fields of an ITB location determined by a not-last-used algorithm. The TAG and PTE fields are updated simultaneously to insure the integrity of the ITB. A write to the ITB_PTE register increments the NLU pointer, which allows for writing the entire set of ITB PTE and TAG entries. The TAG field of the ITB location is determined by the contents of the ITB_TAG register. The PTE field is available in the MTPR ITB_PTE instruction. Writes to this register use the memory format bits as described in the Open VMS memory management chapter of the Alpha SRM.

Note: The NLU pointer is bumped in trap shadows.

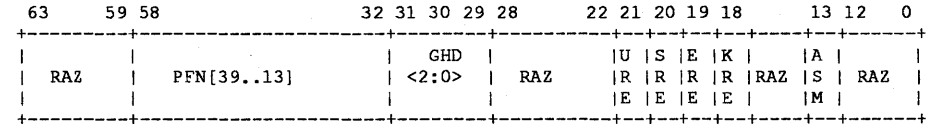
Figure 1–25: Istream TB PTE Write Format, ITB_PTE



A read of the ITB_PTE requires two instructions. A read of the ITB_PTE register, returns the PTE pointed to by the NLU pointer to the ITB_PTE_TEMP register and updates the NLU pointer according to the not-last-used algorithm. A zero value is returned to the integer register file. A second read of the ITB_PTE_TEMP register returns the PTE the the general purpose integer register file.

Note: The NLU pointer is bumped in trap shadows.

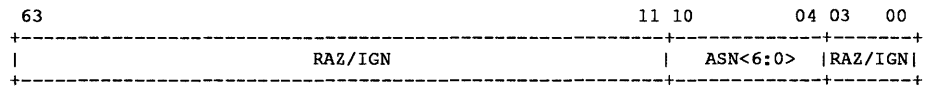
Figure 1–26: Istream TB PTE Read Format, ITB_PTE



1.2.10.3 Address Space Number, ITB_ASN

The ITB_ASN register is a read/write register which contains the Address space number (ASN) of the current process.

Figure 1–27: Address Space Number Read/Write Format, ITB_ASN



1.2.10.4 ITB_PTE_TEMP

The ITB_PTE_TEMP register is a read-only holding register for ITB_PTE read data. A read of the ITB_PTE register returns data to this register. A second read of the ITB_PTE_TEMP register returns data to the integer general purpose register file.

Figure 1–28: Istream TB PTE Temp Read Format, ITB_PTE_TEMP

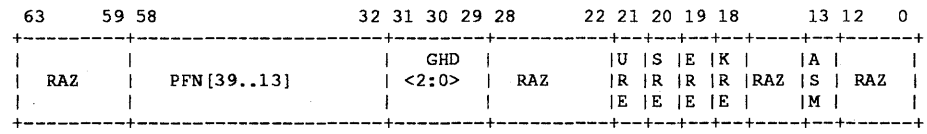


Table 1–8: Description of GHD bits in ITB_PTE_TEMP read format

Name	Extent	Type	Description
GHD	31	RO	Is set if GH(granularity hint) equals 11.
GHD	30	RO	Is set if GH(granularity hint) equals 10 or 11.
GHD	29	RO	Is set if GH(granularity hint) equals 01, 10 or 11.

1.2.10.5 Istream TB Invalidate All Process, ITB_IAP

This is a write-only register. Any write to this register invalidates all ITB entries, whose ASM bit equals zero.

1.2.10.6 IStream TB Invalidate All, ITB_IA

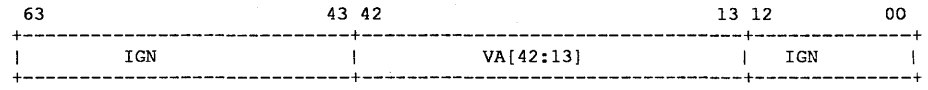
This is a write-only register. Any write to this register invalidates all ITB entries, and resets the ITB NLU pointer to its initial state. RESET Palcode must execute an MTPR ITB_IA instruction in order to initialize the NLU pointer.

1.2.10.7 ITB_IS

This is a write-only register. Writing a virtual address to this IPR invalidates the ITB entry that meets any one of the following criteria:

- An ITB entry whose VA field matches ITB_IS<42:13> and whose ASN field matches ITB_ASN<10:4>.
- An ITB entry whose VA field matches ITB_IS<42:13> and whose ASM bit is set.

Figure 1–29: ITB_IS



1.2.10.8 Formatted Faulting VA register, IFAULT_VA_FORM

This is a read-only register which contains the formatted faulting virtual address on an ITBMiss/IACCVIO. The formatted faulting address generated depends on whether NT super page mapping is enabled through the SPE <0> bit of the ICSR.

Figure 1–30: IFAULT_VA_FORM in non NT mode

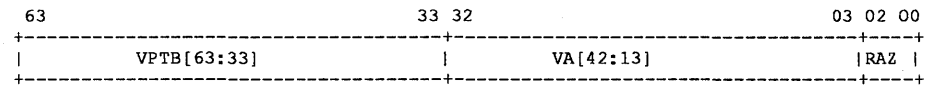
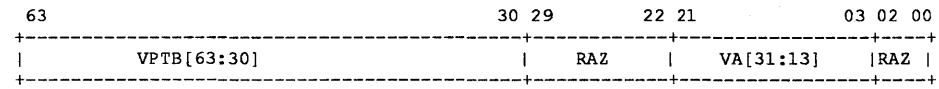


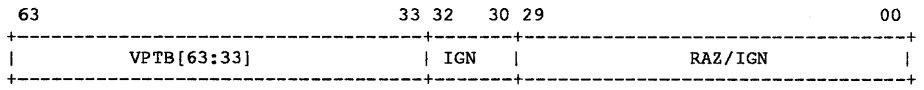
Figure 1–31: IFAULT_VA_FORM in NT mode



1.2.10.9 Virtual Page Table Base register, IVPTBR

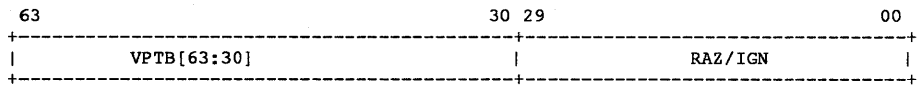
This is a read-write register.

Figure 1–32: IVPTBR in non NT mode



Bits <32:30> are undefined on a read of this register in non NT mode.

Figure 1–33: IVPTBR in NT mode



1.2.10.10 Icache Parity Error Status register, ICPERR_STAT

This is read/write register that contains information about an Icache Parity error. The error status bits may be cleared by writing a 1 to the appropriate bits.

Figure 1–34: ICPERR_STAT Read format

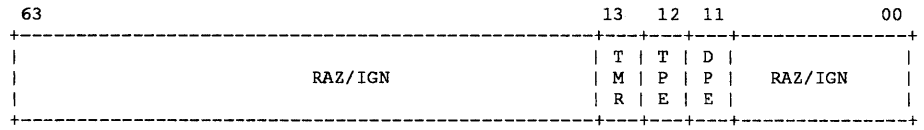


Table 1–9: ICPERR_STAT Field Descriptions

Name	Extent	Type	Description
DPE	11	W1C	Data parity error.
TPE	12	W1C	Tag parity error.
TMR	13	W1C	Timeout reset error.

1.2.10.11 ICache Flush Control register, IC_FLUSH_CTL

This is a write-only register. Writing any value to this register flushes the entire Icache.

1.2.10.12 Exception Address register, EXC_ADDR

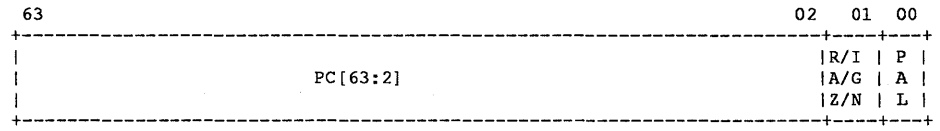
The EXC_ADDR register is a read-write register used to restart the machine after exceptions or interrupts. The HW_REI instruction causes a return to the instruction pointed to by the EXC_ADDR register. This register can be written both by hardware and software. Hardware writes happen as a result of exceptions/interrupts and CALLPAL instructions. Hardware writes which occur as a result of exceptions/interrupts take precedence over all other writes.

In case of an exception/interrupt, hardware writes a PC to this register in S6 of the execution pipeline. In case of precise exceptions, this is the PC of the instruction that caused the exception. In case of imprecise exceptions/interrupts, this is the PC of the next instruction that would have issued if the exception/interrupt was not reported.

In case of a CALLPAL instruction, the PC of the instruction after the CALLPAL is written to EXC_ADDR in S5. Software writes of the register through the HW_MTPR instruction also take place in S5. At a given time only a CALLPAL or HW_MTPR instruction will attempt to write EXC_ADDR as both these instructions are slotted to the E1 pipe.

BIT <0> of this register is used to indicate PAL mode. On a HW_REI the mode of the machine is determined by BIT <0> of the EXC_ADDR register.

Figure 1–35: EXC_ADDR Read/Write format



1.2.10.13 Exception Summary register, EXC_SUM

The exception summary register records the different arithmetic traps that have occurred since the last time EXC_SUM was written. Any write to this register clears bits <16:10>.

Figure 1–36: Exception Summary register Read Format, EXC_SUM

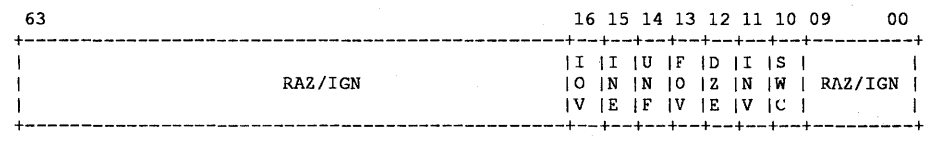


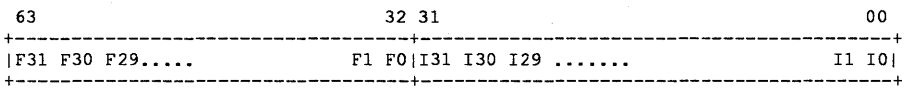
Table 1–10: EXC_SUM Field Descriptions

Name	Extent	Type	Description
SWC	10	WA	Indicates Software completion possible. This bit is set after a floating point instruction containing the /S modifier completes with an arithmetic trap and all previous floating point instructions that trapped since the last MTPR EXC_SUM also contained the /S modifier. The SWC bit is cleared whenever a floating point instruction without the /S modifier completed with an arithmetic trap. The bit remains cleared regardless of additional arithmetic traps until the register is written via an MTPR instruction. The bit is always cleared upon any MTPR write to the EXC_SUM register.
INV	11	WA	Indicates invalid operation.
DZE	12	WA	Indicates divide by zero.
FOV	13	WA	Indicates floating point overflow.
UNF	14	WA	Indicates floating point underflow.
INE	15	WA	Indicates floating inexact error.
IOV	16	WA	Indicates Fbox convert to integer overflow or Integer Arithmetic Overflow.

1.2.10.14 Exception Mask Register, EXC_MASK

The exception mask register records the destinations of instructions that have caused an arithmetic trap, since the last time EXC_MASK was cleared. The destination is recorded as a single bit mask in the 64 bit IPR representing F0-F31 and I0-I31. A write to EXC_SUM clears the EXC_MASK register.

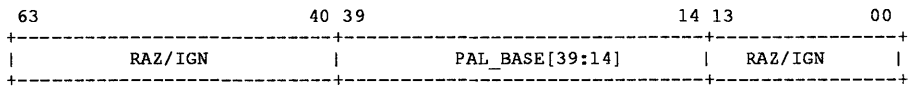
Figure 1–37: Exception Mask register Read Format, EXC_MASK



1.2.10.15 PAL Base Register, PAL_BASE

The PAL_BASE register is a read/write register which contains the base address for PALcode. The register is cleared by hardware on reset.

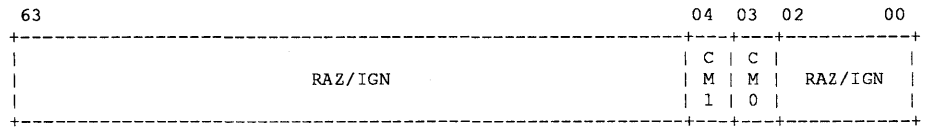
Figure 1–38: PAL_BASE



1.2.10.16 Processor Status, PS

The processor_status register is a read/write register containing the current mode bits of the architecturally defined PS.

Figure 1–39: Processor Status, PS



1.2.10.17 Ibox Control/Status Register, ICSR

This is a read-write register which contains Ibox related control and status information.

Figure 1–40: Ibox Control/Status Register ICSR

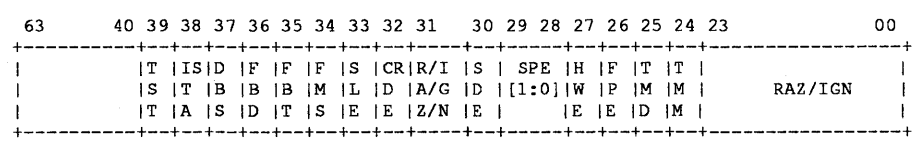


Table 1–11: ICSR Field Descriptions

Name	Extent	Type	Description
TMM	24	RW,0	If set, the timeout counter counts 5K cycles before asserting timeout reset. If clear, the timeout counter counts 1 billion cycles before asserting timeout reset.
TMD	25	RW,0	If set, disables the timeout counter.
FPE	26	RW,0	If set floating point instructions may be issued. When clear floating point instructions cause FEN exceptions.
HWE	27	RW,0	If set, allows PALRES instructions to be issued in kernel mode.

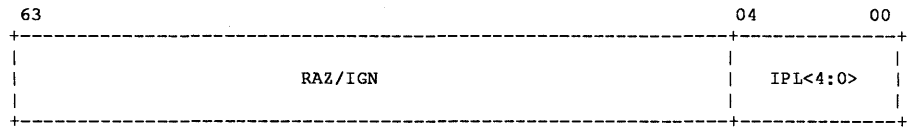
Table 1–11 (Cont.): ICSR Field Descriptions

Name	Extent	Type	Description
SPE	29:28	RW,0	If SPE<1> is set, it enables super page mapping of istream virtual addresses VA<39:13> directly to physical address PA<39:13> if VA<42:41> = 10. Virtual address bit VA<40> is ignored in this translation. Access is allowed only in kernel mode. SPE<0> when set, enables super page mapping of istream virtual addresses VA<42:30>=1FFE (Hex) directly to physical address PA<39:30>= 0(Hex). VA<30:13> is mapped directly to PA<30:13>. Access is allowed only in kernel mode.
SDE	30	RW,0	If set, enables PAL shadow registers.
CRDE	32	RW,0	If set, enables correctable error interrupts.
SLE	33	RW,0	If set, enables serial line interrupts.
FMS	34	RW,0	If set, forces miss on Icache references.
FBT	35	RW,0	If set, forces bad Icache tag parity.
FBD	36	RW,0	If set, forces bad Icache data parity.
DBS	37	RW,1	This bit controls the selection of the multiplexer for the debug port. If set the debug port sees bits <11:4> of the siloed PC. If cleared, the packet from the MBOX is selected.
ISTA	38	RO	Reading this bit indicates ICACHE BIST status. If set, ICACHE BIST was successful.
TST	39	RW,0	Writing a 1 to this bit causes the TEST_STATUS_H pin of the chip to be asserted.

1.2.10.18 Interrupt Priority Level Register, IPL

This is a read/write register containing the value of the architecturally specified IPL register. Whenever hardware detects an interrupt whose target IPL level is greater than the value in IPL<4:0>, an interrupt is taken.

Figure 1–41: Interrupt Priority Level Register, IPL



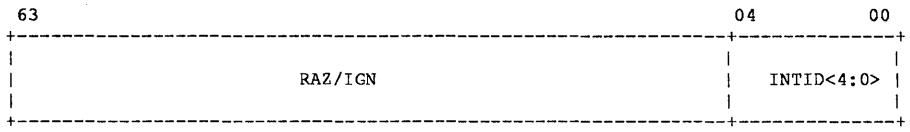
1.2.10.19 Interrupt Id Register, INTID

This is a read only register. It is written by hardware with the target IPL of the highest priority pending interrupt. The hardware recognizes an interrupt if this IPL is greater than the IPL given by IPL<4:0>. Interrupt service routines may use the value of this register to determine the cause of the interrupt. PAL code, for the interrupt service, must ensure that the IPL level

in INTID is greater than the IPL level specified by the IPL register. This restriction is required because a level sensitive hardware interrupt may disappear before the interrupt service routine is entered (passive release).

The contents of INTID are not correct on a HALT interrupt, as this particular interrupt does not have a target IPL at which it can be masked. When a HALT interrupt occurs INTID indicates the next highest priority pending interrupt. PAL code for interrupt service must check the ISR to determine if a HALT interrupt has occured.

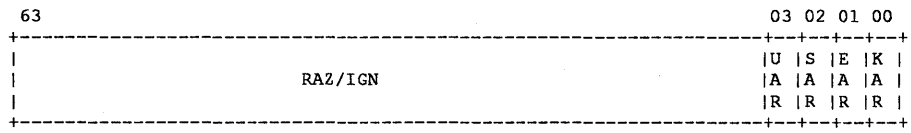
Figure 1–42: Interrupt Id Register, INTID



1.2.10.20 Aynchronous System Trap Request Register, ASTRR

The Asynchronous System Trap Request Register is a read/write register which contains bits to request AST interrupts in each of the four processor modes(USEK). In order to generate an AST interrupt, the corresponding enable bit in the ASTER must be set and the current processor mode given in PS<4:3> should be equal or higher than the mode associated with the AST request.

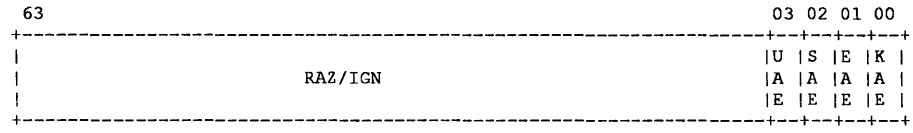
Figure 1–43: Asynchronous System Trap Request Register, ASTRR



1.2.10.21 Aynchronous System Trap Enable Register, ASTER

The Asynchronous System Trap Enable Register is a read/write register which contains bits to enable corresponding AST interrupt requests.

Figure 1–44: Asynchronous System Trap Enable Register, ASTER format



1.2.10.22 Software Interrupt Request Register. SIRR

The Software Interrupt Request Register is a read/write register used to control software interrupt requests. A software request for a particular IPL may be requested by setting the appropriate bit in SIRR<15:1>.

Figure 1–45: Software Interrupt Request Register, SIRR write format

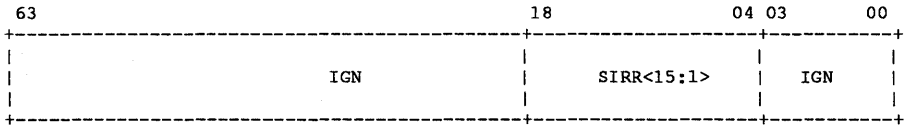


Table 1–12: SIRR Field Descriptions

Name	Extent	Type	Description
SIRR	18:4	RW	Request software interrupts.

1.2.10.23 HW Interrupt Clear register, HWINT_CLR

This is a write-only register, used to clear edge-sensitive hardware interrupt requests.

Figure 1–46: Hardware Interrupt Clear Register, HWINT_CLR

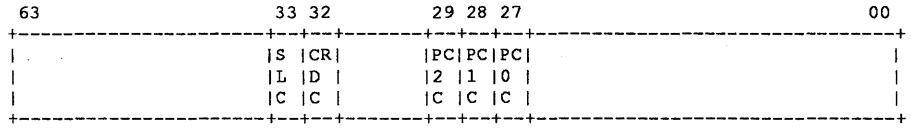


Table 1–13: HWINT_CLR Field Descriptions

Name	Extent	Type	Description
PC0C	27	W1C	Clears perf counter 0 interrupt requests.
PC1C	28	W1C	Clears perf counter 1 interrupt requests.
PC2C	29	W1C	Clears perf counter 2 interrupt requests.
CRDC	32	W1C	Clears correctable read data interrupt requests.
SLC	33	W1C	Clears serial line interrupt requests.

1.2.10.24 Interrupt Summary register, ISR

The Interrupt Summary register is a read only register which contains information about all pending hardware/software/AST interrupt requests.

Figure 1–47: Interrupt Summary Register, ISR read format

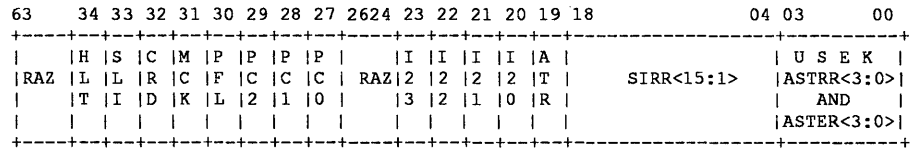


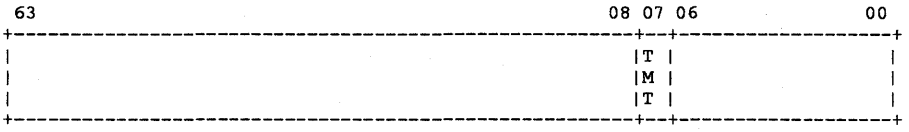
Table 1–14: ISR read format Field Descriptions

Name	Extent	Type	Description
ASTRR[3:0]	3:0	RO	AST requests 3 through 0 (USEK) at IPL 2.
SIRR[15:1]	18:4	RO,0	Software interrupt requests 15 through 1 corresponding to IPL 15 through 1.
ATR	19	RO	Is set if any AST request and corresponding enable bit is set and if the processor mode is equal to or higher than the AST request mode.
I20	20	RO	External hardware interrupt at IPL 20.
I21	21	RO	External hardware interrupt at IPL 21.
I22	22	RO	External hardware interrupt at IPL 22.
I23	23	RO	External hardware interrupt at IPL 23.
PC0	27	RO	External hardware interrupt - Performance counter 0 (IPL 29).
PC1	28	RO	External hardware interrupt - Performance counter 1 (IPL 29).
PC2	29	RO	External hardware interrupt - Performance counter 2 (IPL 29).
PFL	30	RO	External Hardware interrupt - Powerfail (IPL 30).
MCK	31	RO	External Hardware interrupt - system machine check (IPL 31).
CRD	32	RO	Correctable ECC errors (IPL 31).
SLI	33	RO	Serial line interrupt.
HLT	34	RO	External Hardware interrupt - halt .

1.2.10.25 Serial line transmit, SL_XMIT

The serial line transmit register is a write-only register used to transmit bit-serial data off chip under the control of a software timing loop. The value of the TMT bit is transmitted off chip on the SROM_CLK_H pin. In normal operation mode (not in debug-mode), the SROM_CLK_H pin is overloaded and serves both the serial line transmission and the Icache serial ROM interface.

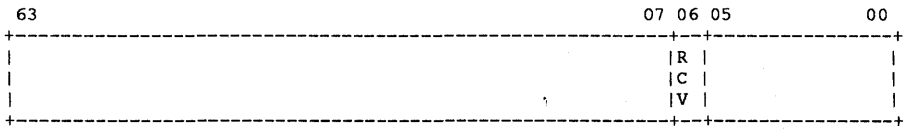
Figure 1–48: Serial line transmit Register, SL_XMIT



1.2.10.26 Serial line receive, SL_RCV

The serial line receive register is a read-only register used to receive bit-serial data under the control of a software timing loop. The RCV bit in the SL_RCV register is functionally connected to the SROM_DAT_H pin. A serial line interrupt is requested whenever a transition is detected on the SROM_DAT_H pin and the SLE bit in the ICSR is set. During normal operations (not in test-mode), the SROM_DAT_H pin is overloaded and serves both the serial line reception and the ICache serial ROM interface.

Figure 1–49: Serial line receive Register, SL_RCV



1.2.11 Traps and Interrupts

For the purpose of this discussion,traps are events that cause a change in control flow other than those caused by explicit transfer of control instructions in the istream. Traps are caused when an instruction (and instructions that follow) are not allowed to complete because of one of the following reasons:

- Replay Traps: An instruction that has passed the issue stage (and therefore cannot be stalled anymore) encounters a resource conflict and must be retried. The traps belonging to this category are MBOX_UNAVAIL and LD_USE_REPLAY and CORR_ECC_REPLAY.
- The istream was incorrectly predicted and must therefore be corrected. - BR/PC Mispredicts.
- Exceptions: An instruction encounters an exception condition which needs to be serviced. The exception is precise if the instruction causing the exception, and all instructions that follow are aborted before the exception is serviced. The exception is imprecise if the trapping instruction is allowed to complete, but some later instruction and all instructions that follow

it are aborted cleanly. From the point of view of the implementation the imprecise exception in effect looks like a precise exception tied to a later instruction in the istream. Once the exception is serviced control returns to the istream with the exception causing instruction.

- Interrupts - An interrupt has been detected, and must be serviced before control may return to the interrupted istream. This is achieved by associating the interrupt with some instruction in the istream and causing that instruction to trap.

For a detailed description of exceptions and interrupts refer to Chapter 9 of the ALPHA SRM.

Whenever a trap is detected in EV5 the following actions take place:

- All instructions in the shadow of the trap are aborted. The trap shadow consists of all instructions including and after the trapping instruction (or some later instruction, as in the case of imprecise exceptions) that have entered some stage of the execution pipeline.
- A new Icache fetch address is generated and supplied to the fetcher. In case of replay traps this is the address of the trapping instruction and in case of istream mispredicts it is the corrected istream address. In the case of exceptions/interrupts this is a PAL entry point associated with the exception/interrupt.

  If the event is an exception/interrupt the following additional steps are taken.
- An address is written to the EXC_ADDR register in S6 and is also pushed on the prediction stack for use by following HW_REI instructions. In case of precise exceptions this is the address of the trapping instruction. In case of imprecise exceptions this is the address of some instruction in the istream that follows the trapping instruction or in the case of interrupts some instruction in the istream which has been chosen as the cutoff point. The requirement is that all instructions before this address must complete and no instruction at or after this location must be allowed to complete.
- When the fetcher receives a restart address because of an exception/interrupt, the PAL mode bit, which is the LSB of the PC, is set to indicate that a PAL flow is now beeing executed. The PAL mode bit is then piped along with the rest of the PC.
- The trap logic makes a request to the issue stage for a TRAPB. This ensures that the issue stage will drain out all previous exceptions before allowing execution to continue.

The following is a table of all events that cause traps in EV5.

Table 1–15: IBOX TRAPS, ENTRY POINTS and INTERRUPT

Name	Description
RESET	Reset
MCHK	Uncorrectable hardware error
ARITH	Arithmetic Exception
INTERRUPT	Hardware, Software or AST interrupts.
ITBMISS	Istream Translation buffer miss
IACCVIO	Istream Access violation, also includes sign_check error on PC
OPCDEC	Illegal Opcode includes: - Opcodes: 01-07, 0A, 0C-0E, 14, 1C

Table 1–15 (Cont.): IBOX TRAPS, ENTRY POINTS and INTERRUPT

Name	Description
	- privileged CALLPAL instr in non-kernel mode.
	- CALLPAL instr outside the range 0-3F or 80-BF
	- PALRES instr attempted in native mode (non-PAL) and HWE bit in ICSR or kernel mode not set.
FEN	Floating Point Operation attempted with:
	- FP Instructions(LD, ST and Operates) disabled through FPE bit in ICSR
	- FP IEEE operation with datatype other than S, T or Q
DTBMISS_SGL	DTBMiss - Dstream Translation Buffer Miss.
DTBMISS_DBL	DTBMiss - DTBMiss in ITBMiss or DTBMISS_SGL flow.
DUNALIGN	Dstream unaligned VA
DFAULT	Dstream access violation or BAD VA
MBOX_UNAVAIL	Includes MAF full, WB conflict and WB full
CORR ECC REPLAY	A dstream correctable ECC error was detected.
BR MISPRED	Branch Target Mispredict
PC MISPRED	PC Mispredict on target of JMP class instr.
LD USE REPLAY	Attempt to use load data which missed in Dcache.

1.2.11.1 Trap Prioritization and cross-products

The traps in EV5 belong to one of the following categories based on the time of posting.

- S4 traps - Trap detected when the instruction that caused the trap is in S4 of the execution pipeline.
- S5 traps - Trap detected when the instruction that caused the trap is in S5 of the execution pipeline.
- S6 traps - Trap detected when the instruction that caused the trap is in S6 of the execution pipeline.
- Asynchronous events (ASYNC) - The instruction that caused the trap is no longer in stages S0-S6 of the execution pipeline or, as in the case of interrupts the trap causing event is not tied to any particular instruction in the pipeline. This class includes, amongst others, imprecise exceptions and interrupts.

When multiple trap signals are asserted simultaneously the trap signal associated with the earlier instruction takes precedence. ASYNC traps therefore take precedence over both S6, S5 and S4 traps asserted at the same time. A single instruction may generate multiple traps. Table 1–16 defines the prioritization used to determine the highest priority trap in such an event.

1.2.11.1.1 Asynchronous traps

ASYNC traps are synchronized with the execution pipeline by piggybacking them on some instruction currently in the execution pipeline. The ASYNC trap is brought into the execution pipeline in S5. An attempt is made to post the ASYNC trap by tying it to the first *valid* instruction in the S5 stage. The attempt to post the trap is successful if the following conditions are met:

- The first instruction in the S5 stage is not in the shadow of an "non-exception" trap.
- The first instruction in the S5 stage is not the target of a mispredicted JSR.

If all of the above conditions are met the attempt to post the ASYNC trap is successful. If not, the trap is deferred, until the conditions are satisfied. If the trap attempt is successful, the EXC_ADDR register is loaded in S6 and the return address is pushed on the stack at the same time. The fetcher is restarted at the PAL entry point associated with the ASYNC event. If the attempt to post the ASYNC trap succeeded in the shadow of an exception, the EXC_ADDR register and the stack already contain the right address. The fetcher is restarted but EXC_ADDR is not loaded and no-address is pushed on the stack.

Table 1–16: Trap Prioritization

Name	Trap Time	Category	Priority	Restart Address
RESET	Async	Exception	1	PAL offset: 0000
MCHK	Async	Exception	2	PAL offset: 0080
ARITH	Async	Imprecise Exception	3	PAL offset: 0100
INTERRUPT	Async	Interrupt	4	PAL offset: 0580
CORR_ECC_REPLAY	Async	Replay	5	Replay first valid instr in S6.
OPCDEC	S5	Precise Exception	6	PAL offset: 0280
FEN	S5	Precise Exception	6	PAL offset: 0200
DTBMISS_SGL	S6	Precise Exception	7	PAL offset: 0480
DTBMISS_DBL	S6	Precise Exception	7	PAL offset: 0500
DUNALIGN	S6	Precise Exception	8	PAL offset: 0300
DFAULT	S6	Precise Exception	9	PAL offset: 0380
MBOX_UNAVAIL	S6	Replay	10	Replay trapping instr.
ITBMISS	S4	Precise Exception	11	PAL offset: 0400
IACCVIO	S4	Precise Exception	12	PAL offset: 0180
BR MISPRED	S5	Istream mispr.	13	Correct Br target
PC MISPRED	S4	Istream mispr.	13	Correct Jmp Target
LD USE REPLAY	S4	Replay	13	Replay trapping instr

1.2.11.2 Aborting Ibox pipe stages on traps

Whenever a trap is detected all instructions in the trap shadow are aborted by the following mechanism.

- The trap logic asserts I_TRP%TRAP_POSTED_A_H which causes pipe stages S0-S3 to be flushed.
- The trap logic aborts register file writes in the shadow by asserting the signals **I_TRP%ABORT_E0_6A_H, I_TRP%ABORT_E1_6A_H, I_TRP%ABORT_LD0_6A_H, I_TRP%ABORT_LD1_6A_H, I_TRP%ABORT_FA_6A_H, I_TRP%ABORT_FM_6A_H** through the shadow.
- Writes to Ibox IPR's are aborted by asserting the signal **I_TRP%ABORT_IPR_WRITE_5A_H** through the shadow.

1.2.11.3 Aborting Mbox pipe stages on traps

At the IBOX end all precise traps except those generated by the MBOX are known by the end of S5A, The ASYNC traps are brought into S5 and are therefore also known by the end of S5A. Based on the trap information two kill signals **I%M_KILL_E0_5B_H** and **I%M_KILL_E1_5B_H** are generated towards the end of S5B, which tell the MBOX if one or both the pipes are to be aborted.

1.2.11.4 Generating Restart addresses

A restart address is generated based on the highest priority trap signal. There are four categories of trap restart addresses.

- Branch/PC Mispredict: Restart address is **I_WPC%BR_PC_MPRED_IDX_5A_H.**
- S4 replays(LD_USE_REPLAY): Restart address is **I_WPC%REPLAY_IDX_6A_H.**
- S6 Replay trap(MBOX UNAVAIL): Restart address is **I_WPC%REPLAY_IDX_4A_H.**
- Exceptions/interrupts: Restart address is **I_TRP%EXCEPTION_PC_A_H.**

If one or more exception/interrupt signals are asserted, a PAL entry point **I_TRP%EXCEPTION_PC_A_H** is generated based on the highest priority exception/interrupt asserted.

The PAL entry point is a 40 bit physical address formed as shown.

Figure 1–50: PAL_ENTRY

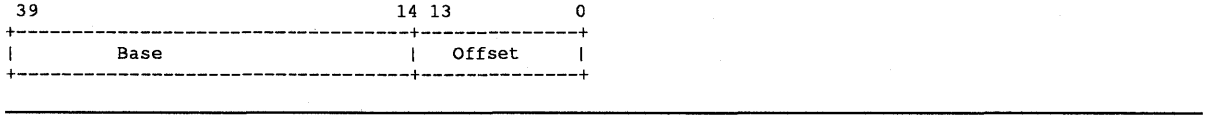


Table 1–17: PAL_ENTRY

Name	Extent	Description
Base	39:14	From Pal_Base Register
Offset	13:0	PAL Entry Offset

The 14 bit PAL entry offset associated with the various exceptions is listed in Table 1–16. The trap vectors are assigned at fixed intervals of 128 bytes.

The trap logic generates control signals which determines which one of the four restart addresses is to be used.

1.2.11.5 INTERRUPTS

The EV5 chip supports three sources of interrupts: hardware, software and asynchronous system traps (AST). There are 7 level sensitive hardware interrupts sourced by pins, 2 edge sensitive hardware interrupts for performance counters sourced by pins, 15 software interrupts sourced by an internal IPR (SIRR) and 4 AST interrupts (one for each processor mode) sourced by a second internal IPR (ASTRR). Associated with each interrupt source is a target interrupt priority level(IPL). An interrupt is masked when its target IPL is less than the value specified by the IPL IPR. Additional masking capability is provided for the AST interrupts. The AST interrupts can be masked by clearing the corresponding enable bits of the ASTER IPR. Also AST interrupt requests need to be qualified with the current processor mode. An AST interrupt request is made only when the mode (USEK) associated with the request is equal to or lower than the current processor mode.

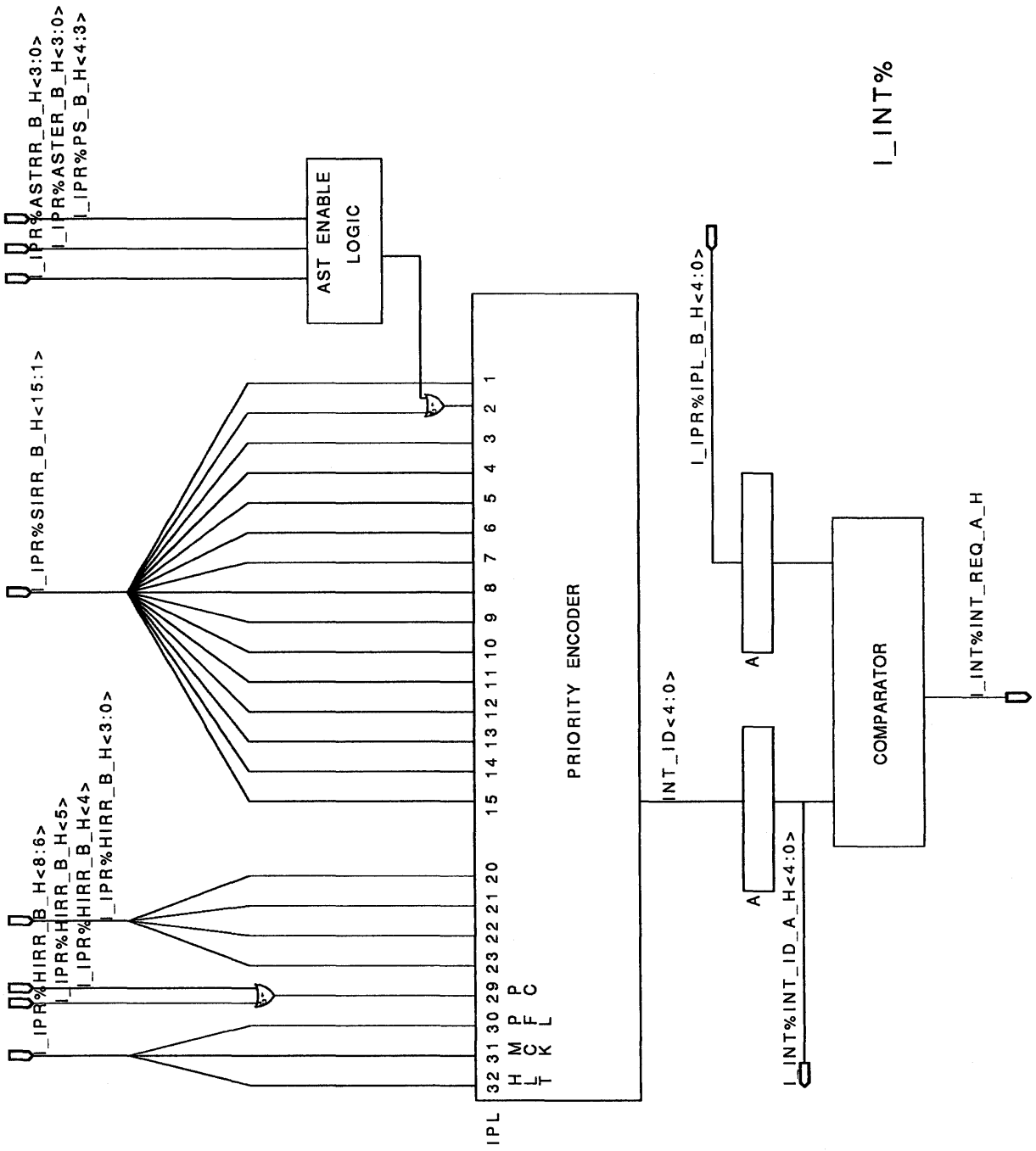
1.2.11.5.1 Interrupt Generation Logic

The interrupt generation logic priority encodes the interrupt requests from all possible interrupt sources, and selects the highest priority pending interrupt. The highest priority pending interrupt is the interrupt request with the highest target IPL value. This value is latched in the INTID IPR. A comparator determines whether the value of the highest pending interrupt is greater than the value stored in the IPL IPR. If so, an interrupt request is made to the trap logic if the machine is not currently executing PAL. All interrupt requests are masked in PALmode. The trap logic prioritizes the different trap/interrupt sources and eventually a PAL entry point for the interrupt service routine is taken. A level sensitive hardware interrupt may deassert before the PAL entry point for the interrupt request is taken. This is termed a passive release. PALcode for interrupt service must therefore check to see if the value in the INTID IPR is greater than the value stored in the IPL IPR. The INTID IPR is continuously updated and can therefore be used to determine if a passive release ocurred. PALcode may also use the INTID IPR to determine the nature of the interrupt i.e. software, hardware, AST. The following table lists the different interrupt sources and their target IPL values.

Table 1–18: Interrupt Priority Level Effect

Interrupt Source	Target IPL (decimal)
Halt	Masked only by executing in PAL mode
System machine check interrupt and internally detected correctable error interupt.	31
Power fail interrupt	30
Performance counters	29
External interrupt 23(I/O interrupt at IPL 23)	23
External interrupt 22(I/O interrupt at IPL 22; interprocessor interrupt; timer interrupt)	22
External interrupt 21(I/O interrupt at IPL 21)	21
External interrupt 20(I/O interrupt at IPL 20)	20
Software Interrupt Request 15	15
Software Interrupt Request 14	14
Software Interrupt Request 13	13
Software Interrupt Request 12	12
Software Interrupt Request 11	11
Software Interrupt Request 10	10
Software Interrupt Request 9	9
Software Interrupt Request 8	8
Software Interrupt Request 7	7
Software Interrupt Request 6	6
Software Interrupt Request 5	5
Software Interrupt Request 4	4
Software Interrupt Request 3	3
Software Interrupt Request 2	2
AST pending (for current or more privileged mode)	2
Software Interrupt Request 1	1

Figure 1–51: IBOX TNTERRUPT LOGIC



1.2.11.6 ERRORS

This section is *******TBD********

1.3 Reset and Initialization

1.4 Error Handling and Recording

1.5 Test Aspects

1.6 Performance Monitoring Features

1.7 Issues

1.8 Revision History

Table 1–19: Revision History

Who	When	Description of change
jbk	12/10/91	cold start
npp	12/30/91	pc,stack,branch portions added
vr	1/07/91	IB/slot/IPR/traps sections added

Chapter 2

The Ebox

2.1 Overview-Block Diagram

The EV5 Ebox is the execution unit which performs the integer arithmetic, logical, and byte-manipulation instructions of the Alpha instruction set. It also partially executes memory (LOAD/STORE) and instruction flow control instructions.

The EV5 Ebox is leveraged from the EV4 design as much as possible, but it is mostly a new design. An additional pipe has been added, and most of the functional units will be redesigned. The EV4 Z-bit logic has been removed, reducing much complexity in the branch logic and at the output of the execution units. Other changes include: the multiplier retires 8 bits per cycle, the Ebox is now responsible for virtual address generation in load/store instructions, 8 PAL shadow registers have been added to the register file, the register file does not have any write ports dedicated to the Mbox for load and fill data, and the register file does not have a read port dedicated to store instructions.

The Ebox consists of a 40 entry register file and two instruction execution pipelines. The pipelines E0 and E1 are four stages long, semi-symmetrical, fully bypassable, and they operate in parallel. This permits two instructions to be issued to the Ebox each cycle. Instructions may also be issued to the Ebox in parallel with instructions issued to the Fbox.[1] To reduce the latency of each instruction to the time required for its execution, a bypass path exists from the output of each stage in each pipe to each input of both pipes. There is no direct path, however, between the Ebox and the Fbox. Both pipes contain the hardware required to execute most instructions. The exceptions are shift, byte-manipulation, store, and multiply instructions, which must be issued to pipe E0; and the instruction flow control instructions, which must be issued to pipe E1. Some MxPR instructions are also restricted based on the location of the register: MxPR instructions involving registers in the Ibox are only executed in pipe E1, and MTPR instructions to registers in the Mbox are only executed in pipe E0.

A table of the instructions handled by each pipe is shown in Table 2–1.

[1] See the Ibox spec for detailed information on instruction issuing.

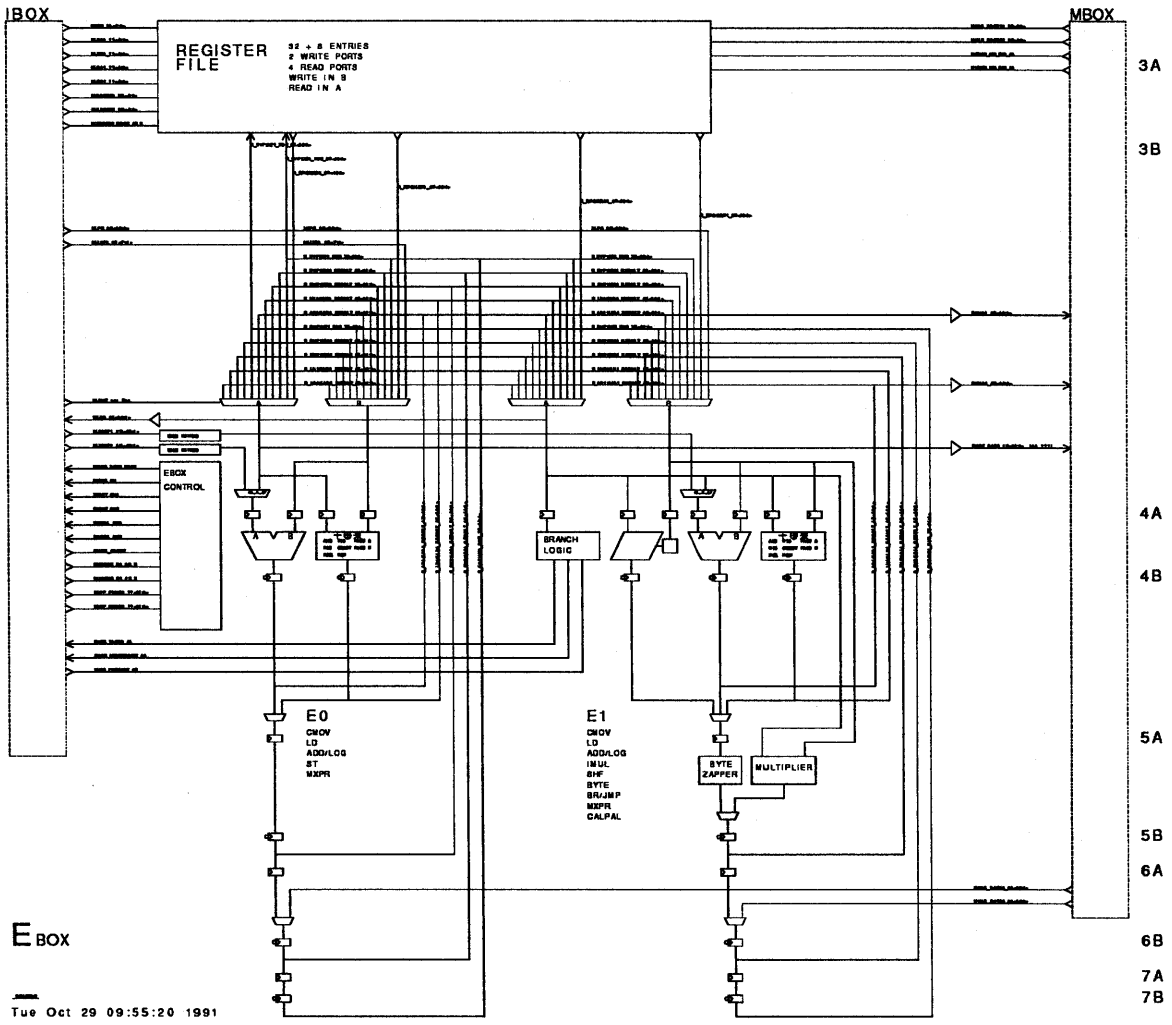
Table 2–1: Instruction Matrix

Instruction	Pipe E0	Pipe E1
STx	YES	NO
LDx	YES	YES
STx_C	YES	NO
LDx_L	YES	NO
JMP	NO	YES
JSRx	NO	YES
RET	NO	YES
BRx	NO	YES
ADDx	YES	YES
SUBx	YES	YES
CMPx	YES	YES
CMPBGE	YES	YES
S4ADDx	YES	YES
S4SUBx	YES	YES
S8ADDx	YES	YES
S8SUBx	YES	YES
AND	YES	YES
BIS	YES	YES
BIC	YES	YES
XOR	YES	YES
ORNOT	YES	YES
EQV	YES	YES
CMOVx	YES	YES
SLL	YES	NO
SRx	YES	NO
EXTx	YES	NO
INSx	YES	NO
MSKx	YES	NO
ZAPx	YES	NO
MULx	YES	NO
UMULH	YES	NO
FETCHx	YES	NO
RPCC	YES	NO
Rx	YES	NO
CALL_PAL	NO	YES
DRAINT	YES	NO

Table 2–1 (Cont.): Instruction Matrix

Instruction	Pipe E0	Pipe E1
MB	YES	NO

Figure 2–1: Ebox Block Diagram



2.2 Functional Description

2.2.1 Register File

The Ebox register file contains the 32 integer registers (R00 thru R31), as described in the Alpha SYSTEM REFERENCE MANUAL, and 8 implementation specific PAL shadow registers (SR08 thru SR15). Each register is 64 bits wide and has 4 read ports and 2 write ports.

Three signals control whether the PAL shadow registers are addressed or not, 1 read and 2 PAL shadow address signals. The read PAL control signal selects between reading from reading R08-R15 or SR08-SR15. Because there can be loads pending, there are separate write PAL signals, one for each write port. The PAL shadow address signals operate in the same way the normal address signals do - all control is left to the Ibox.

2.2.2 Bypass Logic

The Ebox bypass logic is controlled by the Ibox. Each Ebox pipe can bypass data being written into the register file or any of the intermediate pipe stage results onto the A or B operand of either pipe. The bypass scheme is shown in the block diagram in Figure 2–1.

The results of most operations are available for bypassing in 4B. The exceptions are shift & byte operations, CMOVx, CMPBGE, and CMPx, which are valid in 5B, and multiplies, which take many cycles.

2.2.3 Adder

Each Ebox pipe contains an adder. Each adder performs the operate instructions ADDx, SUBx, and generates memory instruction addresses for the Mbox. It should be noted that memory instruction addresses do not need to be bypassed because the address, with the exceptions of LDA and LDAH, is not the result of the operation.

The adder accepts three operands: **AX** and **BX** from the bypass logic, and a sign extended displacement **DISP**. From these two inputs are selected for the adder. The **A** input can be **AX**, **AX** shifted left two bits, **AX** shifted left three bits, or the sign extended **DISP**. The **B** input can be **BX** or the one's complement of **BX**. In addition, there is a carry-in to the adder. See Figure 2–2 for details about adder inputs for various instructions.

The adder produces two outputs: the main datapath result of the add operation and an eight bit byte carry-out field (**BYTE_COUT**). The main datapath result is always the 64 bit sum **A + B + CIN**, or a 32 sign extended version of **A<31:0> + B<31:0> + CIN**.

BYTE_COUT is a vector containing the carry-outs from byte operations in the adder. For x = 1 to 7, **BYTE_COUT<X>** is the carry-out of **A<8X + 7 : 8X> + B<8X + 7 : 8X> + 1**. **BYTE_COUT<0>** is the carry-out of **A<0> + B<0> + CIN**. **BYTE_COUT** is only used for the byte compare operation.

The adder produces a carry-out of the quadword add and the true sign of the quadword operation, **QW_SIGN**. **QW_SIGN** is only used for the signed compare operations, and the carry-out is only used for the unsigned compare operations.

The adder asserts a quadword overflow signal if the 64 bit operation **A + B + CIN** produces a result that does not fit in 64 bits, and it asserts a longword overflow signal if the 32 bit operation **A<31:0> + B<31:0> + CIN** produces a result that does not fit in 32 bits. These overflow signals do not indicate an integer overflow exception by themselves. They must first be combined with opcode information and integer overflow enable.

Figure 2–2: Summary of Adder Control

NU = not used
IG = ignored

Opcode	Scale Mux	Bin	Cin	Byte Cout	63 Cout	31 Ovf	63 Ovf	LW sext	Result
LDA,	SEXT disp<15:0>	B	0	NU	NU	IG	IG	no	USED
LDAH	SEXT (disp<15:0><<16)	B	0	NU	NU	IG	IG	no	USED
LDL, LDQ	SEXT disp<15:0>	B	0	NU	NU	IG	IG	no	USED
LDQ_U	SEXT disp<15:0>	B	0	NU	NU	IG	IG	no	USED
LDL_L, LDQ_L	SEXT disp<15:0>	B	0	NU	NU	IG	IG	no	USED
STL_C, STQ_C	SEXT disp<15:0>	B	0	NU	NU	IG	IG	no	USED
STL, STQ	SEXT disp<15:0>	B	0	NU	NU	IG	IG	no	USED
STQ_U	SEXT disp<15:0>	B	0	NU	NU	IG	IG	no	USED
Bxx	nothing...								
BR, BSR	nothing...								
JMP, MSR, RET, JSR_CO..	nothing...								
ADDL	SHIFT 0	B	0	NU	NU	VALID	IG	yes	USED
S4ADDL	SHIFT 2	B	0	NU	NU	IG	IG	yes	USED
S8ADDL	SHIFT 3	B	0	NU	NU	IG	IG	yes	USED
ADDQ	SHIFT 0	B	0	NU	NU	IG	VALID	no	USED
S4ADDQ	SHIFT 2	B	0	NU	NU	IG	IG	no	USED
S8ADDQ	SHIFT 3	B	0	NU	NU	IG	IG	no	USED
CMPEQ	SHIFT 0	!B	1	NU	NU	IG	IG	no	USED
CMPLT	SHIFT 0	!B	1	NU	NU	IG	USED	no	USED
CMPLE	SHIFT 0	!B	1	NU	NU	IG	USED	no	USED
CMPULT	SHIFT 0	!B	1	NU	USED	IG	IG	no	USED
CMPULE	SHIFT 0	!B	1	NU	USED	IG	IG	no	USED
MULL, MULQ, UMULH	nothing...								

Figure 2–2 Cont'd on next page

Figure 2–2 (Cont.): Summary of Adder Control

Opcode	Scale Mux	Bin	Cin	Byte Cout	63 Cout	31 Ovf	63 Ovf	LW sext	Result
SUBL	SHIFT 0	!B	1	NU	NU	VALID	IG	yes	USED
S4SUBL	SHIFT 2	!B	1	NU	NU	IG	IG	yes	USED
S8SUBL	SHIFT 3	!B	1	NU	NU	IG	IG	yes	USED
SUBQ	SHIFT 0	!B	1	NU	NU	IG	VALID	no	USED
S4SUBQ	SHIFT 2	!B	1	NU	NU	IG	IG	no	USED
S8SUBQ	SHIFT 3	!B	1	NU	NU	IG	IG	no	USED
AND, BIS, XOR, BIC, ORNOT, EQV	nothing...								
CMOVxx	nothing...								
SLL, SRL	nothing...								
SRA	nothing...								
CMPBGE	SHIFT 0	!B	1	USED	NU	IG	IG	no	IG
EXTxx	nothing...								
INSxx	nothing...								
MSKxx	nothing...								
ZAPx	nothing...								
FETCH, TBIS	disp = 0	B	0	NU	NU	IG	IG	no	USED
HW_LD, HW_ST	SEXT disp<11:0>	B	0	NU	NU	IG	IG	no	USED

2.2.4 Logic Unit

Each Ebox pipe contains a logic unit. Each logic unit implements all the integer logical functions (See Table 2–4), the ability to pass A, the ability to pass B, a zero detector, and logic to examine bits <63> and <0> of A.

The main section of the logic unit contains the functionality for the logical functions and the ability to pass A. The output of this section passes through the zero detector, which is used to detect an A operand of zero. The zero detector is also used on the result of A XOR B to determine equivalence between the two operands. B cannot be passed through the zero detector. B can be passed through the logic unit while a zero detect is being performed on A.

The logic unit is used for all integer logical instructions, all integer compare instructions, all unsigned integer compare instructions, and all conditional move instructions. The logic unit must also be used to perform a shift of zero on A; this impacts the shift, extract byte, and insert byte instructions.

2.2.5 Shifter

The shifter is located in pipe E0 and is a right shifter only. The input to the shifter is a 128 bit vector that is constructed from the outputs of two datapath input muxes. It is composed of various combinations of operand A, sign extensions of A<63>, and 0. For the specific combinations used for each shift related instruction see Table 2–5, Table 2–6, and Table 2–8.

The amount of the right shift is determined from bits <5:0> of operand B and the intended direction of the shift. See the aforementioned tables for the right shift amounts used for each instruction.

The major limitation of the shifter is that it can not perform a shift of zero. When operand A must be shifted by zero, the logic unit is used to pass A in lieu of a shift.

The shifter consumes enough time so that shift instructions cannot complete in one cycle. All instructions which use the shifter have a latency of two cycles.

All shift instructions use the shifter, as well as all extract byte and insert byte instructions. The extract and insert byte instructions are only partially executed in the shifter.

2.2.6 Byte Zapper

There is a byte zapper in pipe E0 of the Ebox. This unit is used during all compare, unsigned compare, compare byte, shift, extract byte, insert byte, mask byte, and zap byte instructions. It simply masks different fields of a result passed from another execution unit in the Ebox; forcing all masked bits to zero. Generally, the mask resolution is at the byte level, except that bits <7:1> can also be masked for the compare instructions.

All the instructions which use the zapper have a latency of two cycles, and the zapper operates in the second of these two. Therefore, it does not receive its operands from the bypass logic as all the other execution units in the Ebox do.

For a compare operation, the zapper passes the result of the compare in bit <0>; all other bits are forced to zero. The result of a compare byte operation are passed in bits <7:0> with the other bits masked. The zapper does nothing on a shift operation, passing all bits. See Table 2–7, Table 2–9, Table 2–10, and Section 2.3.21 for zapper operation details for the extract, insert, mask, and zap byte instructions respectively.

Pipe E1 contains a less complex byte zapper that is only used for the various compare instructions.

The zappers are in the normal path of data flow in each pipe. For all instructions other than those mentioned the zappers will pass data untouched.

2.2.7 Multiplier

The Ebox multiplier performs all of the integer multiply operations. MULL operations have a latency of 8 cycles and MULQ & UMULH take 13 cycles to complete. The multiplier array retires 8 bits per cycle. The other latency cycles are accounted for in the following way: two cycles are used at the front end to allow the booth decoder to get started and to do the first array calculation based on the LSB of the multiplier; at the backend, in the case of MULL, two cycles are used to do a final addition of the carry & sum bits and calculate overflow, whereas MULQ and UMULH take an extra addition cycle for a total of three backend cycles.

The multiplier result is can be inserted into the E0 pipe in 5B. The Ibox controls which cycle this occurs. Overflow is reported to the Ibox in the following 6A.

2.2.8 Branch Condition Logic

The branch logic is used to execute the conditional branch instructions. It receives operand A as data and the condition type and the branch prediction information as control. This unit contains logic to examine bits <63> and <0> of A, and it contains a zero detector which operates on A. This is different from the EV4 implementation in that there is no Z-bit that arrives with the operand. Therefore, a zero detector is necessary.

The branch logic determines whether the branch condition is met, and whether the branch prediction is correct. For details about conditional branch, see Section 2.3.33.

2.3 Instruction Flows

This section discusses the execution of instructions in the Ebox pipelines. The execution flows for most Ebox instructions require only one cycle and actively use only one stage of the pipeline. Whenever this is not the case, the number of cycles needed to execute the instruction will be given.

The possible operands for an instruction include **RA**, **RB**, **#B** (literal), and **DISP**. **RA** and **RB** are the contents of integer register 'a' and 'b' respectively. The value for **RA** and/or **RB** may come from the register file or from one of the multiple bypass paths in the Ebox. The displacement operand (**DISP**) usually refers to the displacement field of the memory format instruction; that is, bits <15:0> of the instruction longword. The exception is in the case of HW_LD/HW_ST and is discussed in context. A literal (**#B**) can be used in place of **RB** for operate instructions. It comes from bits <20:13> of the instruction longword.

The Ibox controls the selection of **RA** and **RB**. The Ibox also controls the destination register for each instruction and provides the register file write enable.

Unless indicated otherwise, the result of each instruction flows down the pipe until it is written into the register file in cycle 7.

2.3.1 Compare (CMPEQ, CMPLT, CMPLE)

These instructions are executed in the adder, logic unit, and byte zapper of either pipe.

The bypass mux_A selects **RA**; the bypass mux_B selects either **RB** or **#B**, as indicated in the instruction.

The logic unit performs a bit-wise XOR of **A** and **B**. A zero detect is then performed on the result of the XOR. If the result of the XOR is zero, then **LOGIC_Z** is set to indicate that **A** is equal to **B**.

RA is selected as the **A** input to the adder, and the one's complement of **RB** is selected as the **B** input. The carry-in is set. The adder computes **A + B + CIN**. The true sign of the result, **QW_SIGN**, is calculated (**QW_SIGN** = **A<63>** XOR **B<63>** XOR carry-out of the add).

The result of the compare is determined from the outputs of the logic unit and the adder. See Table 2–2.

Table 2–2: Compare

LOGIC_Z	QW_SIGN	CMPEQ	CMPLT	CMPLE
0	0	0	0	0
0	1	0	1	1
1	0	1	0	1

This result is passed to the byte zapper. The byte zapper forces bits <63:1> to zero and passes the result of the compare in bit <0>.

The final result is the output of the byte zapper.

The latency of these compare instructions is two cycles.

No exceptions are possible for these instructions.

2.3.2 Compare Unsigned (CMPULT, CMPULE)

These instructions are executed in the adder, logic unit, and byte zapper of either pipe.

The bypass mux_A selects **RA**; the bypass mux_B selects either **RB** or **#B**, as indicated in the instruction.

The logic unit performs a bit-wise XOR of **A** and **B**. A zero detect is then performed on the result of the XOR. If the result of the XOR is zero, then **LOGIC_Z** is set to indicate that **A** is equal to **B**.

RA is selected as the **A** input to the adder, and the one's complement of **RB** is selected as the **B** input. The carry-in is set. The adder computes **A + B + CIN** and produces a carry-out, **COUT**.

The result of the compare is determined from the outputs of the logic unit and the adder. See Table 2–3.

Table 2–3: Compare

LOGIC_Z	Cout	CMPULT	CMPULE
0	0	0	0
0	1	1	1
1	1	0	1

This result is passed to the byte zapper. The byte zapper forces bits <63:1> to zero and passes the result of the compare in bit <0>.

The final result is the output of the byte zapper.

The latency of these compare instructions is two cycles.

No exceptions are possible for these instructions.

2.3.3 Compare Byte (CMPBGE)

This instruction is executed in the adder and byte zapper of either pipe.

The bypass mux_A selects **RA**; the bypass mux_B selects either **RB** or **#B**, as indicated in the instruction.

RA is selected as the **A** input to the adder, and the one's complement of **RB** is selected as the **B** input. A carry-in to each byte is set. The adder adds each byte of **A** to the complement of each byte of **B** and to a carry-in to that byte for all eight bytes. A carry-out is generated for each byte, and these are passed to the byte zapper.

The byte zapper forces bits <63:8> to zero and passes the byte carry-outs on bits <7:0>.

The final result is the output of the byte zapper.

The latency of this instruction is two cycles.

No exceptions are possible for these instructions.

2.3.4 Logical Functions (AND, BIS, XOR, BIC, ORNOT, EQV)

These instructions are executed in the logic unit of either pipe.

The bypass mux_A selects **RA**; the bypass mux_B selects either **RB** or **#B**, as indicated in the instruction.

The logic unit performs the appropriate logical operation on **A** and **B** in a bitwise fashion. The operations for each opcode are given in the Alpha SRM and repeated here for convenience in Table 2–4.

Table 2–4: Logical Functions

Opcode	Function
AND	A AND B
BIS	A OR B
XOR	A XOR B
BIC	A AND (NOT B)
ORNOT	A OR (NOT B)
EQV	A XOR (NOT B)

If the instruction is issued to pipe E1, the result of the logic unit must be muxed with the **I%PC_4B** bus before dropping into the normal pipe flow.

No exceptions are possible for these instructions.

2.3.5 Conditional Move (CMOVEQ, CMOVNE, CMOVLT, CMOVLE, CMOVGT, CMOVGE, CMOVLBC, CMOVLBS)

These instructions are executed in the logic unit of either pipe.

The bypass mux_A selects **RA**; the bypass mux_B selects either **RB** or **#B**, as indicated in the instruction.

The logic unit passes **B** to its output. This is the result that may be written to the destination register and flows down the pipe normally. The logic unit also tests the **A** value to determine whether **B** will be written. Bits **A<63>** and **A<0>** are examined, and a zero detect (**LOGIC_Z** set if **A** = 0) is performed on **A**. With this information, it determines whether the move condition is true. This logic is summarized in Figure 2–3.

Figure 2–3: Conditional Move Conditions

LOGIC_Z	A<63>	A<0>	CMOVEQ	CMOVNE	CMOVLT	CMOVLE	CMOVGT	CMOVGE	CMOVLBC	CMOVLBS
0	0	0	0	1	0	0	1	1	1	0
0	0	1	0	1	0	0	1	1	0	1
0	1	0	0	1	1	1	0	0	1	0
0	1	1	0	1	1	1	0	0	0	1
1	0	0	1	0	0	1	0	1	1	0

If the condition is false, the appropriate **E%KILL_CMOVX** signal for the pipe is asserted. This indicates to the Ibox that the value of **B** should not be written to the destination register, and that it should also not be bypassed back into the Ebox pipe. Because of this hand-shaking with the Ibox, these instructions have a latency of two cycles.

No exceptions are possible for these instructions.

2.3.6 Add Longword (ADDL)

This instruction is executed in the adder of either pipe.

The bypass mux_A selects **RA**. The bypass mux_B selects either **RB** or **#B**, as indicated in the instruction.

RA is selected as the **A** input to the adder. **RB** or **#B** is selected as the **B** input, and the carry-in is cleared.

The adder adds **A<31:0>**, **B<31:0>**, and **CIN**. The result is sign extended from bit <31>, that is, bits <63:32> of the result are given the same value as bit <31> of the result. Bits <63:32> of **A** and **B** are ignored.

The Ebox can generate integer overflow on this instruction. This exception is produced when integer overflow is enabled and the resultant sum does not fit in bits <31:0> of the sum.

2.3.7 Scaled Add Longword (S4ADDL, S8ADDL)

These instructions are executed in the adder of either pipe.

The bypass mux_A selects **RA**. The bypass mux_B selects either **RB** or **#B**, as indicated in the instruction.

For S4ADDL, **RA**, shifted left by two, is selected as the **A** input to the adder. For S8ADDL, **RA** is shifted left by three. **RB** or **#B** is selected as the **B** input, and the carry-in is cleared. No checks are performed to determine whether any significant bits of **RA** are lost during the shifting.

The adder adds **A<31:0>**, **B<31:0>**, and **CIN**. The result is sign extended from bit <31>, that is, bits <63:32> of the result are given the same value as bit <31> of the result. Bits <63:32> of **A** and **B** are ignored.

No exceptions are possible for these instructions.

2.3.8 Add Quadword (ADDQ)

This instruction is executed in the adder of either pipe.

The bypass mux_A selects **RA**. The bypass mux_B selects either **RB** or **#B**, as indicated in the instruction.

RA is selected as the **A** input to the adder. **RB** or **#B** is selected as the **B** input, and the carry-in is cleared.

The adder adds **A<63:0>**, **B<63:0>**, and **CIN** to produce the result.

The Ebox can generate integer overflow on this instruction. This exception is produced when integer overflow is enabled and the resultant sum does not fit in bits <63:0> of the sum.

2.3.9 Scaled Add Quadword (S4ADDQ, S8ADDQ)

This instruction is executed in the adder of either pipe.

The bypass mux_A selects **RA**. The bypass mux_B selects either **RB** or **#B**, as indicated in the instruction.

For S4ADDQ, **RA**, shifted left by two, is selected as the **A** input to the adder. For S8ADDQ, **RA** is shifted left by three. **RB** or **#B** is selected as the **B** input, and the carry-in is cleared. No checks are performed to determine whether any significant bits of **RA** are lost during the shifting.

The adder adds **A<63:0>**, **B<63:0>**, and **CIN** to produce the result.

No exceptions are possible for these instructions.

2.3.10 Subtract Longword (SUBL)

This instruction is executed in the adder of either pipe.

The bypass mux_A selects **RA**. The bypass mux_B selects either **RB** or **#B**, as indicated in the instruction.

RA is selected as the **A** input to the adder. The one's complement of **RB** or **#B** is selected as the **B** input, and the carry-in is set.

The adder adds **A<31:0>**, **B<31:0>**, and **CIN**. The result is sign extended from bit <31>, that is, bits <63:32> of the result are given the same value as bit <31> of the result. Bits <63:32> of **A** and **B** are ignored.

The Ebox can generate integer overflow on this instruction. This exception is produced when integer overflow is enabled and the resultant difference does not fit in bits <31:0> of the difference.

2.3.11 Scaled Subtract Longword (S4SUBL, S8SUBL)

These instructions are executed in the adder of either pipe.

The bypass mux_A selects **RA**. The bypass mux_B selects either **RB** or **#B**, as indicated in the instruction.

For S4SUBL, **RA**, shifted left by two, is selected as the **A** input to the adder. For S8SUBL, **RA** is shifted left by three. The one's complement of **RB** or **#B** is selected as the **B** input, and the carry-in is set. No checks are performed to determine whether any significant bits of **RA** are lost during the shifting.

The adder adds **A<31:0>**, **B<31:0>**, and **CIN**. The result is sign extended from bit <31>, that is, bits <63:32> of the result are given the same value as bit <31> of the result. Bits <63:32> of **A** and **B** are ignored.

No exceptions are possible for these instructions.

2.3.12 Subtract Quadword (SUBQ)

This instruction is executed in the adder of either pipe.

The bypass mux_A selects **RA**. The bypass mux_B selects either **RB** or **#B**, as indicated in the instruction.

RA is selected as the **A** input to the adder. The one's complement of **RB** or **#B** is selected as the **B** input, and the carry-in is set.

The adder adds **A<63:0>**, **B<63:0>**, and **CIN** to produce the result.

The Ebox can generate integer overflow on this instruction. This exception is produced when integer overflow is enabled and the resultant difference does not fit in bits <63:0> of the difference.

2.3.13 Scaled Subtract Quadword (S4SUBQ, S8SUBQ)

These instructions are executed in the adder of either pipe.

The bypass mux_A selects **RA**. The bypass mux_B selects either **RB** or **#B**, as indicated in the instruction.

For S4SUBQ, **RA**, shifted left by two, is selected as the **A** input to the adder. For S8SUBQ, **RA** is shifted left by three. The one's complement of **RB** or **#B** is selected as the **B** input, and the carry-in is set. No checks are performed to determine whether any significant bits of **RA** are lost during the shifting.

The adder adds **A<63:0>**, **B<63:0>**, and **CIN** to produce the result.

No exceptions are possible for these instructions.

2.3.14 Multiply Longword (MULL)

This instruction is executed in the multiplier, and it must be issued to pipe E0.

The bypass mux_A selects **RA**. The bypass mux_B selects either **RB** or **#B**, as indicated in the instruction.

The multiplier produces a sign extended 32 bit product of **A** and **B**. At ***TBD*** cycles before the multiply is complete, the Ebox asserts **E%MUL_DONE_SOON_H**. Then, once the product has been completed, the multiplier stores it until another multiply instruction is issued or until **I%MUL_ABORT** is asserted. The Ibox asserts **I%SEL_MUL_5B_H** to mux the product into the E0 pipe, from which it can be bypassed from cycle 5 onward.

The minimum latency for MULL is 8 cycles.

The Ebox can generate integer overflow on this instruction. This exception is produced when integer overflow is enabled and the resultant product does not fit in bits <31:0> of the sum.

2.3.15 Multiply Quadword (MULQ)

This instruction is executed in the multiplier, and it must be issued to pipe E0.

The bypass mux_A selects **RA**. The bypass mux_B selects either **RB** or **#B**, as indicated in the instruction.

The multiplier produces a 64 bit product of **A** and **B**. At ***TBD*** cycles before the multiply is complete, the Ebox asserts **E%MUL_DONE_SOON_H**. Then, once the product has been completed, the multiplier stores it until another multiply instruction is issued or until **I%MUL_ABORT** is asserted. The Ibox asserts **I%SEL_MUL_5B_H** to mux the product into the E0 pipe, from which it can be bypassed from cycle 5 onward.

The minimum latency for MULQ is 13 cycles.

The Ebox can generate integer overflow on this instruction. This exception is produced when integer overflow is enabled and the resultant product does not fit in bits <64:0> of the sum.

2.3.16 Multiply Unsigned Quadword High (UMULH)

This instruction is executed in the multiplier, and it must be issued to pipe E0.

The bypass mux_A selects **RA**. The bypass mux_B selects either **RB** or **#B**, as indicated in the instruction.

The multiplier produces the high order 64 bits of the 128 bit product of **A** and **B** multiplied as unsigned numbers. At ***TBD*** cycles before the multiply is complete, the Ebox asserts **E%MUL_DONE_SOON_H**. Then, once the product has been completed, the multiplier stores it until another multiply instruction is issued or until **I%MUL_ABORT** is asserted. The Ibox asserts **I%SEL_MUL_5B_H** to mux the product into the E0 pipe, from which it can be bypassed from cycle 5 onward.

The minimum latency for UMULH is 13 cycles.

No exceptions are generated for this instruction.

2.3.17 Shift (SLL, SRL, SRA)

These instructions are executed in the shifter and in the logic unit of pipe E0. They may only be issued to pipe E0.

The bypass mux_A selects **RA**; the bypass mux_B selects either **RB** or **#B**, as indicated in the instruction.

The logic unit passes **A** to its output.

The shifter is a right shifter only. The input to the shifter is a 128 bit vector that is constructed from the outputs of two datapath input muxes. The correct right shift amount is determined from bits <5:0> of **RB** and the direction of the shift. A summary of the shifter input logic is given in Table 2–5.

Table 2–5: Shifter Inputs

Opcode	Input<127:64>	Input<63:0>	Right Shift Amount
SLL	A	0	NOT(B<5:0>) + 1
SRL	0	A	B<5:0>
SRA	SEXT(A<63>)	A	B<5:0>

The result is the output of the shifter, with one exception. If the shift amount is zero, the result is the output of the logic unit. This is due to the inability of the shifter to perform a shift of zero. The output of the shifter passes through the byte zapper, which does nothing to modify it.

The latency of these shift instructions is two cycles.

No exceptions are possible for these instructions.

2.3.18 Extract Byte (EXTBL, EXTWL, EXTLL, EXTQL, EXTWH, EXTLH, EXTQH)

These instructions are executed in the shifter and in the logic unit and byte zapper of pipe E0. They may only be issued to pipe E0.

The bypass mux_A selects **RA**; the bypass mux_B selects either **RB** or **#B**, as indicated in the instruction.

The logic unit passes **A** to its output.

The shifter is a right shifter only. The input to the shifter is a 128 bit vector that is constructed from the outputs of two datapath input muxes. The correct right shift amount is determined from bits <2:0> of **RB** and the direction of the shift. A summary of the shifter input logic is given in Table 2–6.

Table 2–6: Shifter Inputs for the Extract Byte Instructions

Opcode	Input<127:64>	Input<63:0>	Right Shift Amount
EXTxL	0	A	B<2:0> & 000#2
EXTxH	A	0	B<2:0> & 000#2

The shift result is the output of the shifter, with one exception. If the shift amount is zero, the shift result is the output of the logic unit. This is due to the inability of the shifter to perform a shift of zero.

The output of the shifter passes through the byte zapper, which forces certain bytes of the output to zero. A summary of the byte zapper operation for the extract byte instructions is given in Table 2–7

Table 2–7: Byte Zapper Operation for the Extract Byte Instructions

Opcode	Bits Cleared	Bits Passed
EXTBL	<63:8>	<7:0>
EXTWL/EXTWH	<63:16>	<15:0>
EXTLL/EXTLH	<63:32>	<31:0>
EXTQL/EXTQH	none	<63:0>

The final result is the output of the byte zapper.

The latency of these extract instructions is two cycles.

No exceptions are possible for these instructions.

2.3.19 Insert Byte (INSBL, INSWL, INSLL, INSQL, INSWH, INSLH, INSQH)

These instructions are executed in the shifter and in the logic unit and byte zapper of pipe E0. They may only be issued to pipe E0.

The bypass mux_A selects **RA**; the bypass mux_B selects either **RB** or **#B**, as indicated in the instruction.

The logic unit passes **A** to its output.

The shifter is a right shifter only. The input to the shifter is a 128 bit vector that is constructed from the outputs of two datapath input muxes. The correct right shift amount is determined from bits <2:0> of **RB** and the direction of the shift. A summary of the shifter input logic is given in Table 2–8.

Table 2–8: Shifter Inputs for the Insert Byte Instructions

Opcode	Input<127:64>	Input<63:0>	Right Shift Amount
INSxL	A	0	NOT(B<2:0> & 000#2) + 1
INSxH	0	A	NOT(B<2:0> & 000#2) + 1

The shift result is the output of the shifter, with one exception. If the shift amount is zero, the shift result is the output of the logic unit. This is due to the inability of the shifter to perform a shift of zero.

The output of the shifter passes through the byte zapper, which forces certain bytes of the output to zero. A summary of the byte zapper operation for the insert byte instructions is given in Table 2–9

Table 2–9: Byte Zapper Operation for the Insert Byte Instructions

Opcode	B<2:0>	Bits Cleared	Bits Passed
INSBL	0	<63:8>	<7:0>
INSBL	1	<63:16,7:0>	<15:8>
INSBL	2	<63:24,15:0>	<23:16>
INSBL	3	<63:32,23:0>	<31:24>
INSBL	4	<63:40,31:0>	<39:32>
INSBL	5	<63:48,39:0>	<47:40>
INSBL	6	<63:56,47:0>	<55:48>
INSBL	7	<55:0>	<63:56>
INSWL	0	<63:16>	<15:0>
INSWL	1	<63:24,7:0>	<23:8>
INSWL	2	<63:32>,15:0	<31:16>
INSWL	3	<63:40,23:0>	<39:24>
INSWL	4	<63:48,31:0>	<47:32>
INSWL	5	<63:56,39:0>	<55:40>
INSWL	6	<47:0>	<63:48>
INSWL	7	<55:0>	<63:56>
INSWH	0-6	<63:0>	none
INSWH	7	<63:8>	<7:0>
INSLL	0	<63:32>	<31:0>
INSLL	1	<63:40,7:0>	<39:8>
INSLL	2	<63:48,15:0>	<47:16>
INSLL	3	<63:56,23:0>	<55:24>
INSLL	4	<31:0>	<63:32>
INSLL	5	<39:0>	<63:40>
INSLL	6	<47:0>	<63:48>
INSLL	7	<55:0>	<63:56>
INSLH	0-4	<63:0>	<none>
INSLH	5	<63:8>	<7:0>
INSLH	6	<63:16>	<15:0>
INSLH	7	<63:24>	<23:0>
INSQL	0	<none>	<63:0>
INSQL	1	<7:0>	<63:8>
INSQL	2	<15:0>	<63:16>

Table 2–9 (Cont.): Byte Zapper Operation for the Insert Byte Instructions

Opcode	B<2:0>	Bits Cleared	Bits Passed
INSQL	3	<23:0>	<63:24>
INSQL	4	<31:0>	<63:32>
INSQL	5	<39:0>	<63:40>
INSQL	6	<47:0>	<63:48>
INSQL	7	<55:0>	<63:56>
INSQH	0	<63:0>	<none>
INSQH	1	<63:8>	<7:0>
INSQH	2	<63:16>	<15:0>
INSQH	3	<63:24>	<23:0>
INSQH	4	<63:32>	<31:0>
INSQH	5	<63:40>	<39:0>
INSQH	6	<63:48>	<47:0>
INSQH	7	<63:56>	<55:0>

The final result is the output of the byte zapper.

The latency of these insert instructions is two cycles.

No exceptions are possible for these instructions.

2.3.20 Mask Byte (MSKBL, MSKWL, MSKLL, MSKQL, MSKWH, MSKLH, MSKQH)

These instructions are executed in the logic unit and byte zapper of pipe E0. They may only be issued to pipe E0.

The bypass mux_A selects **RA**; the bypass mux_B selects either **RB** or **#B**, as indicated in the instruction.

The logic unit passes **A** to its output.

The output of the logic unit passes through the byte zapper, which forces certain bytes of the output to zero. A summary of the byte zapper operation for the mask byte instructions is given in Table 2–10

Table 2–10: Byte Zapper Operation for the Mask Byte Instructions

Opcode	B<2:0>	Bits Cleared	Bits Passed
MSKBL	0	<63:8>	<7:0>
MSKBL	1	<63:16,7:0>	<15:8>
MSKBL	2	<63:24,15:0>	<23:16>
MSKBL	3	<63:32,23:0>	<31:24>
MSKBL	4	<63:40,31:0>	<39:32>

Table 2–10 (Cont.): Byte Zapper Operation for the Mask Byte Instructions

Opcode	B<2:0>	Bits Cleared	Bits Passed
MSKBL	5	<63:48,39:0>	<47:40>
MSKBL	6	<63:56,47:0>	<55:48>
MSKBL	7	<55:0>	<63:56>
MSKWL	0	<63:16>	<15:0>
MSKWL	1	<63:24,7:0>	<23:8>
MSKWL	2	<63:32>,15:0	<31:16>
MSKWL	3	<63:40,23:0>	<39:24>
MSKWL	4	<63:48,31:0>	<47:32>
MSKWL	5	<63:56,39:0>	<55:40>
MSKWL	6	<47:0>	<63:48>
MSKWL	7	<55:0>	<63:56>
MSKWH	0-6	<63:0>	none
MSKWH	7	<63:8>	<7:0>
MSKLL	0	<63:32>	<31:0>
MSKLL	1	<63:40,7:0>	<39:8>
MSKLL	2	<63:48,15:0>	<47:16>
MSKLL	3	<63:56,23:0>	<55:24>
MSKLL	4	<31:0>	<63:32>
MSKLL	5	<39:0>	<63:40>
MSKLL	6	<47:0>	<63:48>
MSKLL	7	<55:0>	<63:56>
MSKLH	0-4	<63:0>	<none>
MSKLH	5	<63:8>	<7:0>
MSKLH	6	<63:16>	<15:0>
MSKLH	7	<63:24>	<23:0>
MSKQL	0	<none>	<63:0>
MSKQL	1	<7:0>	<63:8>
MSKQL	2	<15:0>	<63:16>
MSKQL	3	<23:0>	<63:24>
MSKQL	4	<31:0>	<63:32>
MSKQL	5	<39:0>	<63:40>
MSKQL	6	<47:0>	<63:48>
MSKQL	7	<55:0>	<63:56>

Table 2–10 (Cont.): Byte Zapper Operation for the Mask Byte Instructions

Opcode	B<2:0>	Bits Cleared	Bits Passed
MSKQH	0	<63:0>	<none>
MSKQH	1	<63:8>	<7:0>
MSKQH	2	<63:16>	<15:0>
MSKQH	3	<63:24>	<23:0>
MSKQH	4	<63:32>	<31:0>
MSKQH	5	<63:40>	<39:0>
MSKQH	6	<63:48>	<47:0>
MSKQH	7	<63:56>	<55:0>

The final result is the output of the byte zapper.

The latency of these mask instructions is two cycles.

No exceptions are possible for these instructions.

2.3.21 Zap Byte (ZAP, ZAPNOT)

These instructions are executed in the logic unit and byte zapper of pipe E0. They may only be issued to pipe E0.

The bypass mux_A selects **RA**; the bypass mux_B selects either **RB** or **#B**, as indicated in the instruction.

The logic unit passes **A** to its output.

The output of the logic unit passes through the byte zapper, which forces certain bytes of the output to zero. The mask used in the byte zapper is provided directly by the **B** operand for the zap instructions. For ZAP, the mask is **B<7:0>**. For each bit <x> that is set in **B**, bits <8x + 7 : 8x> of the logic unit output are cleared. The other bits are passed.

For ZAPNOT, the mask is NOT **B<7:0>**. For each bit <x> that is set in **B**, bits <8x + 7 : 8x> of the logic unit output are passed. The other bits are cleared.

The final result is the output of the byte zapper.

The latency of these zap instructions is two cycles.

No exceptions are possible for these instructions.

2.3.22 Load Address (LDA, LDAH)

These instructions are executed in the adder of either pipe.

The bypass mux_A is not used. **RB** is selected by the bypass mux_B.

The **A** input to the adder is the sign extended displacement. This sign extension is a direct sign extension of **DISP** for LDA, and a sign extension of (**DISP** shifted left by 16) for LDAH. **RB** is selected as the **B** input to the adder, and the carry-in to the adder is cleared.

The result is **A** + **B** + **CIN**, and is available after one cycle. No exceptions are possible for these instructions.

2.3.23 Load (LDL, LDQ)

The Ebox only partially executes these instructions. The virtual address for the load is generated in the Ebox, and the rest of the instruction is executed in the Mbox and possibly the Cbox. The Ebox portion of these instructions are executed in the adder of either pipe.

The bypass mux_A is not used. **RB** is selected by the bypass mux_B.

The sign extension of **DISP** is selected as the **A** input to the adder. **RB** is selected as the **B** input, and the carry-in is cleared.

The virtual address is **A** + **B** + **CIN**, which is available on one of the **E%VAX_4B** buses after one cycle.

The result of the load is delivered to the Ebox on one of the **M%LD_DATAX_6A** buses, and is muxed into the appropriate Ebox pipe in cycle 6. The Mbox is responsible for picking the pipe into which the data will be inserted, and it also controls the mux in the Ebox pipe which accomplishes this. In the case of LDL, the data is sign extended in the Mbox. If the data is returned late (such as the result of a Dcache miss), the Mbox must ensure that a bubble exists in the instruction flow of the appropriate pipe when data is returned.

The Ebox does not generate any of the possible exceptions for these instructions.

2.3.24 Load Unaligned (LDQ_U)

The Ebox only partially executes this instruction. The virtual address for the load is partially generated in the Ebox, and the rest of the instruction is executed in the Mbox and possibly the Cbox. The Ebox portion of this instruction is executed in the adder of either pipe.

The bypass mux_A is not used. **RB** is selected by the bypass mux_B.

The sign extension of **DISP** is selected as the **A** input to the adder. **RB** is selected as the **B** input, and the carry-in is cleared.

The virtual address is **A** + **B** + **CIN**, with bits <2:0> of the sum cleared. The Ebox does not clear bits <2:0> of the sum; the Mbox is expected to ignore these bits. The sum is available on one of the **E%VAX_4B** buses after one cycle.

The result of the load is delivered to the Ebox on one of the **M%LD_DATAX_6A** buses, and is muxed into the appropriate Ebox pipe in cycle 6. The Mbox is responsible for picking the pipe into which the data will be inserted, and it also controls the mux in the Ebox pipe which accomplishes this. If the data is returned late (such as the result of a Dcache miss), the Mbox must ensure that a bubble exists in the instruction flow of the appropriate pipe when data is returned.

The Ebox does not generate any of the possible exceptions for this instruction.

2.3.25 Load Locked (LDL_L, LDQ_L)

The Ebox only computes the virtual address for these instructions. This is done in the adder of pipe E0 only.

The bypass mux_A is not used. **RB** is selected by the bypass mux_B.

The sign extension of **DISP** is selected as the **A** input to the adder. **RB** is selected as the **B** input, and the carry-in is cleared.

The virtual address is **A + B + CIN**, which is available on the **E%VA0_4B** bus after one cycle.

The result of the load is delivered to the Ebox on one of the **M%LD_DATAX_6A** buses, and is muxed into the appropriate Ebox pipe in cycle 6. The Mbox is responsible for picking the pipe into which the data will be inserted, and it also controls the mux in the Ebox pipe which accomplishes this. In the case of LDL_L, the data is sign extended in the Mbox. If the data is returned on **M%LD_DATA1_6A**, or if the data is returned late (such as the result of a Dcache miss), the Mbox must ensure that a bubble exists in the instruction flow of the appropriate pipe.

The Ebox does not generate any of the possible exceptions for these instructions.

The Ebox does not contain nor control the locked_physical_address register or the lock_flag.

2.3.26 Store Conditional (STL_C, STQ_C)

The Ebox only partially executes these instructions, and is responsible for generating the virtual address. The Ebox also provides the data to be stored and receives the lock_flag.

These instructions are executed in the adder, and can only be issued to pipe E0.

RA is selected by the bypass mux_A. **RB** is selected by the bypass mux_B.

The sign extension of **DISP** is selected as the **A** input to the adder. **RB** is selected as the **B** input, and the carry-in is cleared.

The virtual address is **A + B + CIN**, which is available on **E%VA0_4B**. The data to be stored, **RA**, is available on **E%ST_DATA_3B** (4A?). In the case of STL_C, 64 bits of data are delivered on **E%ST_DATA_3B**; the Mbox must extract the lower longword to be stored in memory.

The Ebox does not generate any of the possible exceptions for these instructions.

The Ebox does not contain nor control the locked_physical_address register or the lock_flag.

The lock_flag must be delivered to the Ebox on one of the **M%LD_DATAX_6A** buses, and is muxed into the appropriate Ebox pipe in cycle 6. The Mbox is responsible for picking the pipe into which the data will be inserted, but if pipe E1 is chosen there must be an appropriate bubble in the pipe. The lock flag is written to the register **RA**. The Ibox keeps track of the destination register numbers.

2.3.27 Store (STL, STQ)

The Ebox only partially executes these instructions, and is responsible for generating the virtual address and delivering the data to be stored.

These instructions are executed in the adder, and can only be issued to pipe E0.

RA is selected by the bypass mux_A. **RB** is selected by the bypass mux_B.

The sign extension of **DISP** is selected as the **A** input to the adder. **RB** is selected as the **B** input, and the carry-in is cleared.

The virtual address is **A + B + CIN**, which is available on **E%VA0_4B**. The data to be stored, **RA**, is available on **E%ST_DATA_3B** (4A?). In the case of STL, 64 bits of data are delivered on **E%ST_DATA_3B**; the Mbox must extract the lower longword to be stored in memory.

The Ebox does not generate any of the possible exceptions for these instructions.

2.3.28 Store Unaligned (STQ_U)

The Ebox only partially executes this instruction, and is responsible for generating the virtual address and delivering the data to be stored.

This instruction is executed in the adder, and can only be issued to pipe E0.

RA is selected by the bypass mux_A. **RB** is selected by the bypass mux_B.

The sign extension of **DISP** is selected as the **A** input to the adder. **RB** is selected as the **B** input, and the carry-in is cleared.

The virtual address is **A + B + CIN**, with bits <2:0> of the sum cleared. The Ebox does not clear bits <2:0> of the sum; the Mbox is expected to ignore these bits. The sum is available on one of the **E%VAX_4B** buses after one cycle. The data to be stored, **RA**, is available on **E%ST_DATA_3B** (4A?).

The Ebox does not generate any of the possible exceptions for this instruction.

2.3.29 Hardware Load (HW_LD)

The Ebox only partially executes this instruction. The virtual address for the load is generated in the Ebox, and the rest of the instruction is executed in the Mbox and possibly the Cbox. The Ebox portion of this instructions is executed in the adder of either pipe.

The bypass mux_A is not used. **RB** is selected by the bypass mux_B.

The sign extension of **DISP** is selected as the **A** input to the adder. This sign extension is done from **DISP<11>**, unlike most memory format instructions. **RB** is selected as the **B** input, and the carry-in is cleared.

The virtual address is **A + B + CIN**, which is available on one of the **E%VAX_4B** buses after one cycle. The Ebox does not zero any of the low order bits in the virtual address.

The result of the load is delivered to the Ebox on one of the **M%LD_DATAX_6A** buses, and is muxed into the appropriate Ebox pipe in cycle 6. The Mbox is responsible for picking the pipe into which the data will be inserted, and it also controls the mux in the Ebox pipe which accomplishes this. If the QW bit of the instruction is cleared, the data is sign extended in the Mbox.

The Ebox does not generate any exceptions for this instructions.

2.3.30 Hardware Store (HW_ST)

The Ebox only partially executes this instruction, and is responsible for generating the virtual address and delivering the data to be stored.

This instruction is executed in the adder, and can only be issued to pipe E0.

RA is selected by the bypass mux_A. **RB** is selected by the bypass mux_B.

The sign extension of **DISP** is selected as the **A** input to the adder. This sign extension is done from **DISP<11>**, unlike most memory format instructions. **RB** is selected as the **B** input, and the carry-in is cleared.

The virtual address is **A + B + CIN**, which is available on **E%VA0_4B**. The Ebox does not zero any of the low order bits in the virtual address. The data to be stored, **RA**, is available on **E%ST_DATA_3B** (4A?). If the QW bit of the instruction is cleared, 64 bits of data are still delivered on **E%ST_DATA_3B**; the Mbox must extract the lower longword to be stored in memory.

The Ebox does not generate any exceptions for this instruction.

2.3.31 Hardware Move From Processor Register (HW_MFPR)

The Ebox only partially executes these instructions. The Ebox primarily just receives and stores the data.

If the IPR is located in the Mbox or Cbox, the instruction is issued to pipe E0. If the IPR is located in the Ibox, the instruction is issued to pipe E1.

The bypass muxes and execution units of the Ebox are not used.

If the IPR is located in the Mbox or Cbox, the data is delivered to the Ebox on one of the **M%LD_DATAX_6A** buses, and is muxed into the appropriate Ebox pipe in cycle 6. The Mbox is responsible for picking the pipe into which the data will be inserted, and it also controls the mux in the Ebox pipe which accomplishes this. If the data can not be delivered by cycle 6, it is returned later, much like a fill.

If the IPR is located in the Ibox, the data is delivered to the Ebox on the **I%PC_4B** bus. The Ebox must mux this data with the output of the logic unit in pipe E1. If the data cannot be delivered to the Ebox on time, the Ibox must re-issue the instruction when data is available.

The Ebox does not generate any exceptions for these instructions.

2.3.32 Hardware Move To Processor Register (HW_MTPR)

The Ebox only partially executes these instructions. The Ebox delivers the data to either the Ibox or the Mbox.

This instruction is executed in the adder. If the IPR is located in the Mbox or Cbox, the instruction is issued to pipe E0. If the IPR is located in the Ibox, the instruction is issued to pipe E1. The execution of these instructions is much like that for the FETCH instruction.

Bypass mux_A is not used. **RB** is selected by the bypass mux_B.

The displacement is forced to zero and then is selected as the **A** input to the adder. **RB** is selected as the **B** input, and the carry-in is cleared.

The adder performs A + B + CIN, which is equal to RB. In this way the adder passes RB to one of the E%VAX_4B buses.

If the IPR is located in the Mbox or Cbox, the data is available on E%VA0_4B and E%ST_DATA_3B (4B?) during the appropriate cycles. If the IPR is located in the Ibox, the data is available on E%PC_3B.

The Ebox does not generate any exceptions for these instructions.

2.3.33 Conditional Branch (BEQ, BNE, BLT, BLE, BGT, BGE, BLBC, BLBS)

The Ebox only partially executes these instructions. The Ebox tests the condition; the Ibox is responsible for generating the virtual address and updating the PC.

These instuctions are executed in the branch logic, and can only be issued to pipe E1.

RA is selected by the bypass mux_A. Bypass mux_B is not used.

The branch logic examines bits <63> and <0> of A, and performs a zero detect (BR_ZBIT set if A = 0) on A. With this information, it determines whether the branch condition is true. This logic is summarized in Figure 2–4.

Figure 2–4: Branch Conditions

BR_ZBIT	A<63>	A<0>	BEQ	BNE	BLT	BLE	BGT	BGE	BLBC	BLBS
0	0	0	0	1	0	0	1	1	1	0
0	0	1	0	1	0	0	1	1	0	1
0	1	0	0	1	1	1	0	0	1	0
0	1	1	0	1	1	1	0	0	0	1
1	0	0	1	0	0	1	0	1	1	0

The Ebox asserts E%BR_TAKEN_5A if the branch condition is true. If the branch condition does not match with I%BR_PREDICT_4A, the Ebox asserts E%BR_MISPREDICT_5A.

These instuctions can not produce an exception.

2.3.34 Unconditional Branch (BR, BSR)

The Ebox only partially executes these instructions. The Ebox stores the old PC.

These instructions do not require any of the Ebox execution units. They must be issued to pipe E1.

Neither bypass mux_A nor bypass mux_B is used.

The Ebox receives the value of the old PC on I%PC_4B. The Ebox is responsible for muxing this into the E1 pipe with the output of the E1 logic unit.

These instuctions can not produce an exception.

2.3.35 Jump (JMP, JSR, RET, JSR_COROUTINE)

The Ebox only partially executes these instructions. The Ebox provides the new PC, and stores the old one.

These instructions do not require any of the Ebox execution units. They must be issued to pipe E1.

Bypass mux_A is not used. **RB** is selected by the bypass mux_B.

The Ebox provides the value of the new PC on **E%PC_3B**. The lower two bits of the value on **E%PC_3B** should not be used by the Ibox. The Ebox receives the value of the old PC on **I%PC_4B**. The Ebox is responsible for muxing this into the E1 pipe with the output of the E1 logic unit.

These instuctions can not produce an exception.

2.3.36 Fetch (FETCH, FETCH_M)

The Ebox only partially executes these instructions. Their execution is performed in the adder, and they are issued to pipe E0.

Bypass mux_A is not used. **RB** is selected by the bypass mux_B.

The displacement is forced to zero and then is selected as the **A** input to the adder. **RB** is selected as the **B** input, and the carry-in is cleared.

The adder performs **A** + **B** + **CIN**, which is equal to **RB**. In this way the adder passes **RB** to one of the **E%VAX_4B** buses.

These instructions can not produce an exception.

2.3.37 Read Cycle Counter / VAX Compatibility (RPCC, RC, RS)

The Ebox only partially executes these instructions. These are treated much like loads, except that the Ebox does not need to generate a virtual address. These instructions are assumed to be issued to pipe E0 only.

Neither bypass mux is used, and no Ebox execution unit is used.

The result of the instruction is delivered to the Ebox on one of the **M%LD_DATAX_6A** buses, and is muxed into the appropriate Ebox pipe in cycle 6. The Mbox is responsible for picking the pipe into which the data will be inserted, and it also controls the mux in the Ebox pipe which accomplishes this. If the data is returned on **M%LD_DATA1_6A**, the Mbox must ensure that a bubble exists in the instruction flow of the pipe E1.

These instructions can not produce an exception.

2.3.38 Other Instructions

The Ebox treats all other instructions, including CAL_PAL, DRAINT, MB, HW_REI and all floating instructions, as NOPs.

2.4 Ebox Interfaces

2.4.1 Ibox Interface

The Ibox issues instructions to the Ebox, and sends the necessary opcode and data information that the Ebox does not already have.

The Ibox is responsible for the control of data flow in the Ebox. This means that the Ibox must determine whether the up-to-date copy of a register is in the register file or in one of the stages of one of the Ebox pipes. The Ibox owns the bypass mux controls to the extent that it dictates whether an operand comes from the register file or a pipe stage. If the source is a pipe stage, and if that stage has more than one source, the Ebox must choose the correct source for that pipe stage.

The Ibox also controls the Ebox register file. The Ibox indicates when the output of an Ebox pipe should be written to a register, and which register should be written. The register file read addresses are also supplied by the Ibox.

The Ibox and Ebox collaborate on certain instructions; Section 2.3 gives details about these instructions, but they are briefly discussed here for convenience. During conditional branch instructions, the Ebox determines the success or failure of the branch and the correctness of the branch prediction. The Ebox stores the old value of the PC for other branch and jump instructions, and it supplies the new PC for jump instructions.

The Ibox can directly abort a multiply operation, and it must reset the multiplier during system initialization (by asserting abort). The Ibox controls when the output of the Ebox multiplier is muxed into stage 5 of the E0 datapath.

The Ebox indicates success or failure for conditional move instructions. The Ibox acts on this information to determine whether to kill the move by not bypassing its result or writing the result to the register file.

The Ebox sends data to the Ibox during HW_MTPR instructions to an Ibox IPR, and receives data from the Ibox during HW_MFPR instructions to an Ibox IPR.

The Ebox reports integer overflow to the Ibox.

The following signals are driven by the Ebox to the Ibox.

- **E%MUL_DONE_SOON_H**
  This signal informs the Ibox that the Ebox multiplier result will be available for bypassing in *TBD* cycles.
- **E%MUL_BUSY_H**
  This signal informs the Ibox that a multiply instruction has been received by Ebox and its result is being calculated.
- **E%INT_OVF0_6A_H**
  This signal informs the Ibox that an integer overflow has occured in the adder in pipe E0.
- **E%INT_OVF1_6A_H**
  This signal informs the Ibox that an integer overflow has occured in the adder in pipe E1.

- **E%KILL_CMOV0_H**
  This signal informs the Ibox that a conditional move instruction issued to pipe E0 has failed its condition evaluation and no register file write should occur.
- **E%KILL_CMOV1_H**
  This signal informs the Ibox that a conditional move instruction issued to pipe E1 has failed its condition evaluation and no register file write should occur.
- **E%BR_TAKEN_5A_H**
  This signal informs the Ibox that a conditinal branch instruction has been successfully evaluated and that the branch can occur.
- **E%BR_MISPREDICT_5A_H**
  This signal informs the Ibox that the evaluation of a conditional branch instruction has been mis-predicted.
- **E%PC_3B_H<63:0>**
  This 64 bit bus carries new PC values and MTPR data from the Ebox to the Ibox.

The following signals are driven by the Ibox to the Ebox.

- **I%BYP_RA0_A0_L**
  This signal selects register file RA0 read port data onto **E_BYP%A0_3B<63:0>**.
- **I%BYP_S40_A0_L**
  This signal bypasses the result from stage 4 of pipe E0 onto **E_BYP%A0_3B<63:0>**.
- **I%BYP_S41_A0_L**
  This signal bypasses the result from stage 4 of pipe E1 onto **E_BYP%A0_3B<63:0>**.
- **I%BYP_S50_A0_L**
  This signal bypasses the result from stage 5 of pipe E0 onto **E_BYP%A0_3B<63:0>**.
- **I%BYP_S51_A0_L**
  This signal bypasses the result from stage 5 of pipe E1 onto **E_BYP%A0_3B<63:0>**.
- **I%BYP_S60_A0_L**
  This signal bypasses the result from stage 6 of pipe E0 onto **E_BYP%A0_3B<63:0>**.
- **I%BYP_S61_A0_L**
  This signal bypasses the result from stage 6 of pipe E1 onto **E_BYP%A0_3B<63:0>**.
- **I%BYP_W0_A0_L**
  This signal bypasses the result from stage 7 of pipe E0 onto **E_BYP%A0_3B<63:0>**.
- **I%BYP_W1_A0_L**
  This signal bypasses the result from stage 7 of pipe E1 onto **E_BYP%A0_3B<63:0>**.
- **I%BYP_RA1_A1_L**
  This signal selects register file RA1 read port data onto **E_BYP%A1_3B<63:0>**.
- **I%BYP_S40_A1_L**
  This signal bypasses the result from stage 4 of pipe E0 onto **E_BYP%A1_3B<63:0>**.
- **I%BYP_S41_A1_L**
  This signal bypasses the result from stage 4 of pipe E1 onto **E_BYP%A1_3B<63:0>**.
- **I%BYP_S50_A1_L**
  This signal bypasses the result from stage 5 of pipe E0 onto **E_BYP%A1_3B<63:0>**.
- **I%BYP_S51_A1_L**
  This signal bypasses the result from stage 5 of pipe E1 onto **E_BYP%A1_3B<63:0>**.

- **I%BYP_S60_A1_L**
  This signal bypasses the result from stage 6 of pipe E0 onto **E_BYP%A1_3B<63:0>**.
- **I%BYP_S61_A1_L**
  This signal bypasses the result from stage 6 of pipe E1 onto **E_BYP%A1_3B<63:0>**.
- **I%BYP_W0_A1_L**
  This signal bypasses the result from stage 7 of pipe E0 onto **E_BYP%A1_3B<63:0>**.
- **I%BYP_W1_A1_L**
  This signal bypasses the result from stage 7 of pipe E1 onto **E_BYP%A1_3B<63:0>**.
- **I%BYP_PC_A1_L**
  This signal selects the Ibox PC bus onto **E_BYP%A1_3B<63:0>**.
- **I%BYP_RB0_B0_L**
  This signal selects register file RB0 read port data onto **E_BYP%B0_3B<63:0>**.
- **I%BYP_S40_B0_L**
  This signal bypasses the result from stage 4 of pipe E0 onto **E_BYP%B0_3B<63:0>**.
- **I%BYP_S41_B0_L**
  This signal bypasses the result from stage 4 of pipe E1 onto **E_BYP%B0_3B<63:0>**.
- **I%BYP_S50_B0_L**
  This signal bypasses the result from stage 5 of pipe E0 onto **E_BYP%B0_3B<63:0>**.
- **I%BYP_S51_B0_L**
  This signal bypasses the result from stage 5 of pipe E1 onto **E_BYP%B0_3B<63:0>**.
- **I%BYP_S60_B0_L**
  This signal bypasses the result from stage 6 of pipe E0 onto **E_BYP%B0_3B<63:0>**.
- **I%BYP_S61_B0_L**
  This signal bypasses the result from stage 6 of pipe E1 onto **E_BYP%B0_3B<63:0>**.
- **I%BYP_W0_B0_L**
  This signal bypasses the result from stage 7 of pipe E0 onto **E_BYP%B0_3B<63:0>**.
- **I%BYP_W1_B0_L**
  This signal bypasses the result from stage 7 of pipe E1 onto **E_BYP%B0_3B<63:0>**.
- **I%BYP_LIT_B0_L**
  This signal selects the pipe E0 literal onto **E_BYP%B0_3B<63:0>**.
- **I%BYP_RB1_B1_L**
  This signal selects register file RB1 read port data onto **E_BYP%B1_3B<63:0>**.
- **I%BYP_S40_B1_L**
  This signal bypasses the result from stage 4 of pipe E0 onto **E_BYP%B1_3B<63:0>**.
- **I%BYP_S41_B1_L**
  This signal bypasses the result from stage 4 of pipe E1 onto **E_BYP%B1_3B<63:0>**.
- **I%BYP_S50_B1_L**
  This signal bypasses the result from stage 5 of pipe E0 onto **E_BYP%B1_3B<63:0>**.
- **I%BYP_S51_B1_L**
  This signal bypasses the result from stage 5 of pipe E1 onto **E_BYP%B1_3B<63:0>**.
- **I%BYP_S60_B1_L**
  This signal bypasses the result from stage 6 of pipe E0 onto **E_BYP%B1_3B<63:0>**.
- **I%BYP_S61_B1_L**
  This signal bypasses the result from stage 6 of pipe E1 onto **E_BYP%B1_3B<63:0>**.

- **I%BYP_W0_B1_L**
  This signal bypasses the result from stage 7 of pipe E0 onto **E_BYP%B1_3B<63:0>**.
- **I%BYP_W1_B1_L**
  This signal bypasses the result from stage 7 of pipe E1 onto **E_BYP%B1_3B<63:0>**.
- **I%BYP_LIT_B1_L**
  This signal selects the pipe E1 literal onto **E_BYP%B1_3B<63:0>**.
- **I%FREEZE_EBOX_3B_H**
  In the register file, this signal determines whether newly decoded or previously decoded registers are read (applies to all registers reads for that cycle).
- **I%LIT0_3B_H<7:0>**
- **I%LIT1_3B_H<7:0>**
- **I%INSTR0_EBOX_2B_H<26:0>**
  This 27 bit bus contains the opcode, function, literal, and displacement information for the E0 pipe instruction.
- **I%INSTR1_EBOX_2B_H<26:0>**
  This 27 bit bus contains the opcode, function, literal, and displacement information for the E1 pipe instruction.
- **I%ISSUE0_EBOX_4A_H**
  This signal informs the Ebox that a valid instruction has been issued to pipe E0.
- **I%ISSUE1_EBOX_4A_H**
  This signal informs the Ebox that a valid instruction has been issued to pipe E1.
- **I%MUL_ABORT_H**
  This signal tells the Ebox to abort a previously issued multiply instruction.
- **I%BR_PREDICT_4A_H**
  This signal informs the Ebox what the result of a conditional branch was predicted to be by the Ibox.
- **I%PC_4B_H<63:0>**
  This 64 bit bus carries the old PC value and MFPR data from the Ibox to the Ebox.
- **I%EW0_ADDR_6A_H<4:0>**
  These signals are the E0 pipe register file write port address.
- **I%EW1_ADDR_6A_H<4:0>**
  These signals are the E1 pipe register file write port address.
- **I%ERA0_ADDR_2A_H<4:0>**
  These signals are the RA0 register file read port address.
- **I%ERA1_ADDR_2A_H<4:0>**
  These signals are the RA1 register file read port address.
- **I%ERB0_ADDR_2A_H<4:0>**
  These signals are the RB0 register file read port address.
- **I%ERB1_ADDR_2A_H<4:0>**
  These signals are the RB1 register file read port address.
- **I%ERD_PAL_SHADOW_ADDR_2A_H**
  This signal determines whether registers R08-R15 or shadow registers SR08-SR15 are accessed. This applies to all register file read ports: RA0, RA1, RB0, RB1.

- **I%W0_EN_7A_H**
  This signal is the E0 pipe register file write enable.
- **I%W1_EN_7A_H**
  This signal is the E1 pipe register file write enable.
- **I%EW0_PAL_SHADOW_ADDR_6A_H**
  This signal determines whether registers R08-R15 or shadow registers SR08-SR15 are written from W0.
- **I%EW1_PAL_SHADOW_ADDR_6A_H**
  This signal determines whether registers R08-R15 or shadow registers SR08-SR15 are written from W1.

2.4.2 Mbox Interface

The Ebox/Mbox interface is mostly one of data transmission. The Ebox sends data to the Mbox on **E%ST_DATA_3B**<63:0> during store, store conditional, HW_ST, and HW_MTPR instructions to Mbox IPRs. The Ebox sends addresses to the Mbox on the **E%VAX_4B**<63:0> buses during store, store conditional, load, load locked, HW_LD, HW_ST, HW_MxPR operations to Mbox IPRs, and FETCH instructions. The Ebox receives data on the **M%LD_DATAX_6A**<63:0> buses during load, load locked, store conditional, HW_LD, HW_MFPR operations to Mbox IPRs, read from process cycle counter, and the VAX compatibility instructions. See Section 2.3 for details on these instructions.

Any time that the Mbox sends data to the Ebox on one of the **M%LD_DATAX_6A**<63:0> buses, it is the responsibility of the Mbox/Ibox to ensure that there is a bubble in the data flow of the appropriate Ebox pipe. The Mbox controls the muxes which bring the data into the pipes.

The following signals are driven by the Ebox to the Mbox

- **E%ST_DATA_3B_H<63:0>**
  This 64 bit bus carries store and MTPR data from the Ebox to the Mbox.
- **E%VA0_4B_H<63:0>**
  This 64 bit bus carries memory instruction virtual addresses and MTPR data from the Ebox to the Mbox. It is the buffered output of the E0 adder.
- **E%VA1_4B_H<63:0>**
  This 64 bit bus carries memory instruction virtual addresses and MTPR data from the Ebox to the Mbox. It is the buffered output of the E1 adder.

The following signals are driven by the Mbox to the Ebox.

- **M%BYP_LD0_S60_L**
  This signal inserts load data into stage 6 of pipe E0.
- **M%BYP_LD1_S61_L**
  This signal inserts load data into stage 6 of pipe E1.
- **M%LD_DATA0_6A_H<63:0>**
  This 64 bit bus carries returning fill data from the Mbox to pipe E0.
- **M%LD_DATA1_6A_H<63:0>**
  This 64 bit bus carries returning fill or MFPR data from the Mbox to pipe E1.

2.5 Exceptions, Traps, & Stalls

If integer overflow is enabled, the Ebox generates that exception if the result of an ADDL, SUBL, or MULL instruction does not fit in 32 bits, or if the result of an ADDQ, SUBQ, or MULQ instruction does not fit in 64 bits. The instruction must have been issued in order to result in overflow. The Ebox does not generate any other exceptions, nor does it generate overflow for any other instructions. The Ebox reports overflow to the Ibox and returns a truncated 32 bit result for longword operations or a truncated 64 bit result for quadword operations.

The Ebox does not initiate any traps.

The Ebox is stalled at cycle 3 when **I%FREEZE_EBOX_3B** is asserted. It cannot be stalled at any other point in either pipe. The output of the multiplier, however, is static and can be read until a new multiply instruction is issued or **I%MUL_ABORT** is asserted.

2.6 Reset and Initialization

The sequencer which controls the multiplier must be reset during initialization. This is accomplished by asserting **I%MUL_ABORT**.

2.7 Revision History

Table 2–11: Revision History

Who	When	Description of change
Dan Dever	23-JAN-1992	Added a description of instruction flows, an adder control chart, and the overview/introduction. Added to the descriptions of the adder, shifter, byte zapper, and the branch logic, and added narative to the interface section.
Harry Fair	06-NOV-1991	CREATED.

Chapter 3

The Fbox

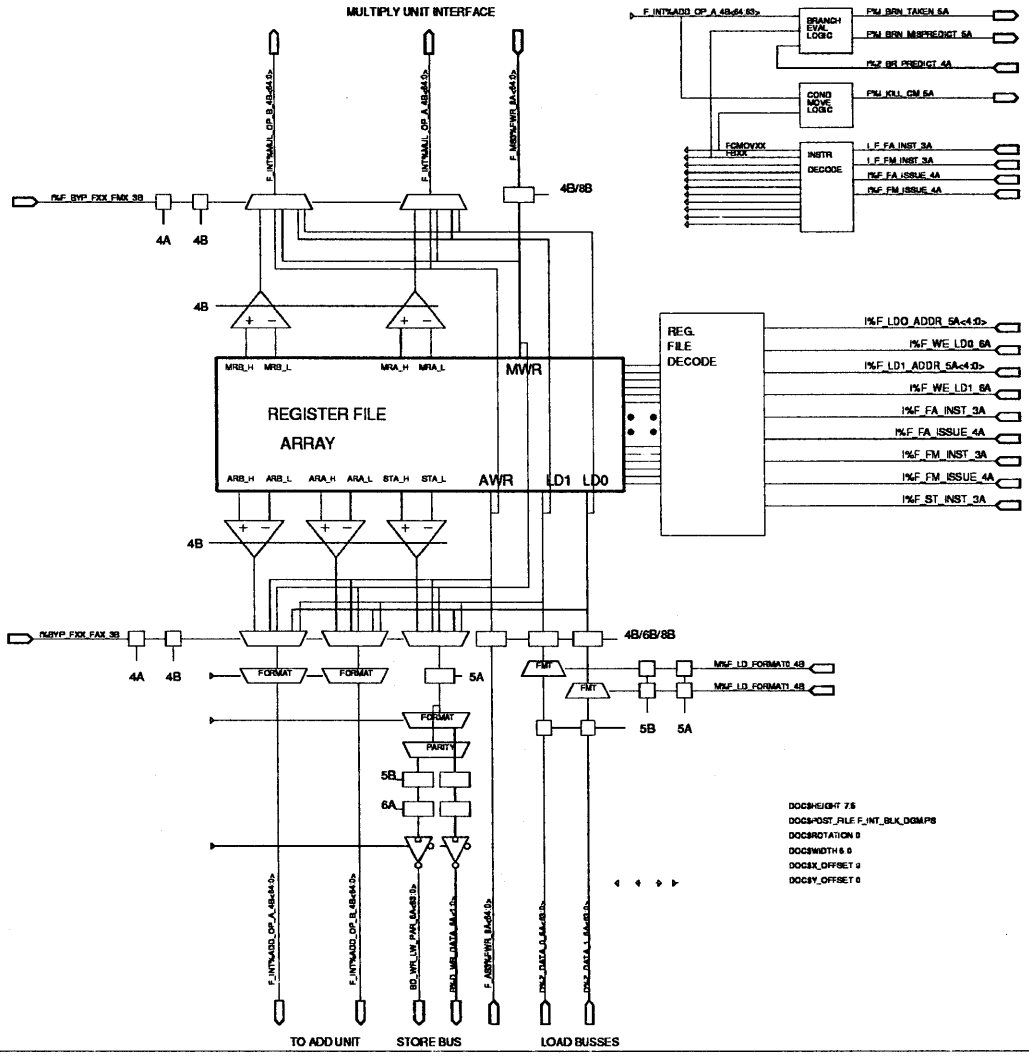
3.1 Overview-Block Diagram

```
FIGURE>(Ebox)
FIGURE_ATTRIBUTES>(KEEP\WIDE)
FIGURE_FILE>(POSTSCRIPT\ebox_blk.ps\43)
ENDFIGURE>
```

3.2 Functional Description

3.3 FBOX Interface

Figure 3–1: Fbox Interface Block Diagram



3.3.1 Interface Overview

3.3.1.1 External Interface

3.3.1.1.1 Floating Point Instruction Issue

The FBOX contains two pipelined functional units: an ADD pipe and a MULTPLY pipe. Each pipe may be issued one floating point ALPHA instruction per cycle from the IBOX. In addition to these functional units, the Fbox floating point register file contains 2 load ports and 1 store port. These ports allow up to 2 floating point loads and 1 floating point store to access the register file each cycle. Table 3–1 lists the instructions which may be issued to each pipe. Floating point load/store instructions are issued to the EBOX at a maximum of two loads or one store per cycle, joining the FBOX pipe for formatting and floating point register file reads and writes. Floating Point loads and stores are described in detail in Section 3.3.1.1.3.

Table 3–1: Floating Point Pipe Instruction Execution

ADD Pipe	ADD pipe (contd)	Multiply Pipe	Load Port	Store Port
ADDx	FBx	CPYS	LDT	STT
CMPx	FCMOVxx	MULx	LDS	STS
CPYSx	MF_FPCR		LDG	STG
CVTx	MT_FPCR		LDF	STF
DIVx	SUBx			

Instruction issue to the FBOX is initiated in cycle 3A by the IBOX placing source register addresses and opcode information on bus **I%F_FA_INST_3A** for an ADD pipe issue, bus **I%F_FM_INST_3A** for a MULTIPLY pipe issue, and bus **I%F_ST_INST_3A** for floating store issues. Instruction issue is completed in cycle 4A by the IBOX asserting **I%F_ISSUE_FA_4A** for ADD pipe issues and **I%F_ISSUE_FM_4A** for MULTIPLY pipe issues. No issue signal is required at the Fbox for floating stores.

3.3.1.1.2 Floating Point Instruction Retirement

Floating point instruction results are retired from both pipes in cycle 8B, at a maximum rate of one instruction per cycle per pipe. Instruction retirement involves writing results into the register file, recording any exceptions which occurred during instruction execution in the Floating Point Control Register (FPCR), and conditionally signalling exceptions to the IBOX.

Instructions are retired from the pipes in cycle 8B by sending register file write addresses and asserting the appropriate write enables. The IBOX places the destination register addresses on bus **I%F_FA_ADDR_7A**<4:0> for ADD pipe retirement and on bus **I%F_FM_ADDR_7A**<4:0> for MULTIPLY pipe retirement. The appropriate register file write enable must also be asserted: **I%F_WE_FM_8A** for MULTIPLY pipe and **I%F_WE_FA_8A** for ADD pipe retirement.

The floating point divider is located in Stage 1 of the Fbox ADD pipe. The divider is non-pipelined, and requires a variable, data-dependent number of cycles to complete. Retirement of floating divide instructions is initiated by the FBOX asserting **F%I_DIV_DONE_SOON_1B** seven cycles before the divide instruction is complete (ready to be retired). The IBOX detects this condition and prevents issue to the ADD pipe in that pipe slot. This allows the divider result to rejoin the ADD pipe without conflict. Six cycles after assertion of **F%DIV_DONE_SOON_1B**, the IBOX

drives the siloed destination register number on **I%F_FA_ADDR_7A**, followed by the RF write enable on **I%F_WE_FA_8A**. The divider fraction result rejoins the ADD pipe from the quotient register at the Fbox stage 3 input during cycle 7A. The fraction and exponent pass thru stage 3 and are written to the FBOX register file in cycle 8B. The signalling and recording of all divide-related exceptions are held until the divide instruction is retired in cycle 8B of the Fbox ADD pipe.

See Section 3.3.1.1.8 for details on abort handling in the FBOX.

3.3.1.1.3 Floating Point LOAD/STORE Issue and Retirement

Floating Point load (LDx) and store (STx) instructions are primarily executed by the EBOX. Their only interaction with the FBOX is during data formatting and FBOX register file reads and writes needed to complete the instruction.

Register file reads for floating point store instructions are accomplished using a dedicated register file port. The IBOX initiates a STx register file read in the FBOX by placing the source register address and opcode information on bus **I%F_ST_INST_3A**.

Store data is then recoded from register file format to memory format and EVEN longword parity is generated for the memory format data. Opcode information driven from the IBOX is used to generate control signals for the store bus data formatter. Memory format data and longword parity are driven to the DCACHE in cycle 6A on **B%D_WR_DATA_6A**<63:0> and **B%D_WR_LW_PAR_6A**<1:0>. For floating point stores of longword-length data (STF, STS), the longword store data and its corresponding parity bit are duplicated on the upper and lower halves of **B%D_WR_DATA_6A**<63:0> and **B%D_WR_LW_PAR_6A**<1:0>. The DCACHE then selects the appropriate longword based on store address bit 2. Store data and parity must be valid at the DCACHE input at the beginning of cycle 6B.

B%D_WR_DATA_6A<63:0> is a global tristate bus having two drivers: the MBOX and the FBOX. The DCACHE is the only receiver. The MBOX is the default bus driver and controls access to the bus: the FBOX is enabled to drive data in cycle 6A when the signal **M%F_FBOX_DRV_ENA_5A** was asserted in the previous cycle.

DCACHE load and fill data is sent to the FBOX on two 64 bit buses: **D%Z_DATA_0_5A**<63:0> and **D%Z_DATA_1_5A**<63:0>. Format information for the data is driven to the FBOX from the MBOX on **M%F_LD_FORMAT0_4B<2:0>** for load bus 0 and **M%F_LD_FORMAT1_4B<2:0>** for load bus 1. Bit 2 of each bus indicates VAX/NOT IEEE format, bit 1 indicates LW/NOT QW length datatype, and bit 0 indicates UPPER/NOT LOWER LONGWORD position for longword-length datatypes. The data is recoded from memory format to register file format in the FBOX during cycles 5B and 6A. The formatted data is driven to the FBOX register file during cycle 6A and is written during cycle 6B.

Register file writes for load instructions are controlled by the IBOX for both *loads* (DCACHE hits) and *fills* (DCACHE misses). During a load/fill, the IBOX places destination register file addresses on **I%F_LD0_ADDR_5A**<4:0> for load bus 0 and on **I%F_LD1_ADDR_5A**<4:0> for load bus 1. The load/fill sequence is completed by asserting **I%F_WE_LD0_6A** and **I%F_WE_LD1_6A** for loads on busses 0 and 1, respectively.

3.3.1.1.4 Operand Bypasses

To reduce pipeline latency by one cycle between data-dependent instructions, the FBOX provides for the bypassing of instruction results around the register file back into the FBOX pipe. Scoreboarding and scheduling of bypasses is performed completely by the IBOX. The FBOX simply muxes potential operand sources into each pipeline using control signals **I%F_BYP_FXX_FXX_3B** provided by the IBOX. Results from load bus 0, load bus 1, the ADD pipe, or the MULTIPLY pipe may be bypassed from the register file write stage (6B for loads/fills and 8B for operates) into the first stage (cycle 4B) of any floating point instruction.

3.3.1.1.5 Floating Point Branch Evaluation

The FBOX evaluates the outcome of floating point branch instructions (FBxx) as well as the validity of the corresponding IBOX branch prediction. The IBOX sends the predicted branch outcome to the FBOX via **I%Z_BR_PREDICT_4A**. The FBOX evaluates the actual branch outcome and signals the IBOX via **F%I_BR_MISPREDICT_5A** and **F%I_BR_TAKEN_5A**. **F%I_BR_MISPREDICT_5A** is asserted if the IBOX branch prediction was incorrect, and **F%I_BR_TAKEN_5A** is asserted if the branch condition evaluates as TRUE. The FBOX conditions the assertion of these signals with the issue signal, preventing spurious branch mispredict indications. The mispredict and taken signals are valid at the IBOX early in cycle 5A.

When the floating branch instruction slot reaches cycle 8A of the FBOX ADD pipe, it is possible for the IBOX to send a register file write enable (implementation-dependent). The FBOX does not require this write enable here (floating point branches do not take exceptions) but can handle it with certain constraints. See Section 3.3.1.1.8 for details.

3.3.1.1.6 Conditional Move Evaluation

The FBOX evaluates the outcome of each conditional move instruction (FCMOVxx) and signals the IBOX with the result. The IBOX uses this result to conditionally disable write enables during retirement of the instruction. The FBOX asserts **F%I_KILL_CM_5A** to signal the IBOX that the move should not be retired (i.e. register write should be KILLed). The FBOX conditions **F%I_KILL_CM_5A** with the ADD pipe issue signal so the IBOX receives it only when an FCMOVxx instruction has actually been issued.

3.3.1.1.7 Pipeline Stalls

As described in Section 3.3.1.1.1, the IBOX issues instructions to the FBOX by sending source register addresses and opcode information in cycle 3A. When a valid instruction has been sent to the FBOX in cycle 3A and it has issued properly, the IBOX asserts **I%F_FA_ISSUE_4A** or **I%F_FM_ISSUE_4A** for issues to the ADD and MUL pipes, respectively. Pipeline stall conditions are handled by the IBOX inhibiting the assertion of **I%F_FA_ISSUE_4A** and **I%F_FM_ISSUE_4A**. Thus, stall conditions in the CPU are transparent to the FBOX.

3.3.1.1.8 Pipeline Aborts

Many conditions may arise in which it is necessary to abort instructions in either FBOX pipe. This includes aborts of the floating point divider in the ADD pipe, whose latency is not deterministic and is much longer than the ADD pipe or the MULTIPLY pipe. In general, aborts can be caused by exception and non-exception conditions.

Non-exception aborts do not require the pipeline to be drained of all outstanding instructions before restarting the pipeline at a redirected address. Examples of non-exception abort conditions are branch mispredictions, subroutine call/return mispredictions, and cache misses. Data cache misses do not produce abort conditions but can cause pipeline stalls. Non-exception aborts are completely transparent to the FBOX.

Aborts caused by exceptions require the pipeline to be drained of all outstanding instructions before restarting the pipeline at a redirected address. The IBOX accomplishes aborts in both FBOX pipelines by disabling write enables during retirement of instructions which are being "drained". Absence of the write enable prevents any results from being written and any related exceptions from being recorded in the FPCR and sent to the IBOX. Thus, write enables function as a late abort mechanism for the FBOX. In addition, assertion of the signal **I->F_FDIV_ABORT** forces any instruction in the FBOX divider to be immediately aborted and the divider reset.

Certain instructions issued to the FBOX produce no Stage 3 result and do not signal exceptions. This class includes floating branches (FBXX) and divide instructions (DIVXX, as opposed to divide bubbles). Although these instructions do not require write enable to be asserted when they reach ADD pipe cycle 8A, it may be convenient for the IBOX to do so for implementation purposes.

For this class of instructions, the IBOX may assert **I%F_WE_FA_8A** when the instruction reaches FBOX Stage 3 (8A), provided the register file address specified is F31. The FBOX internally detects this class of instructions and inhibits both FPCR writes and exception signals to the IBOX, regardless of whether the write enable is asserted.

3.3.1.1.9 Exceptions

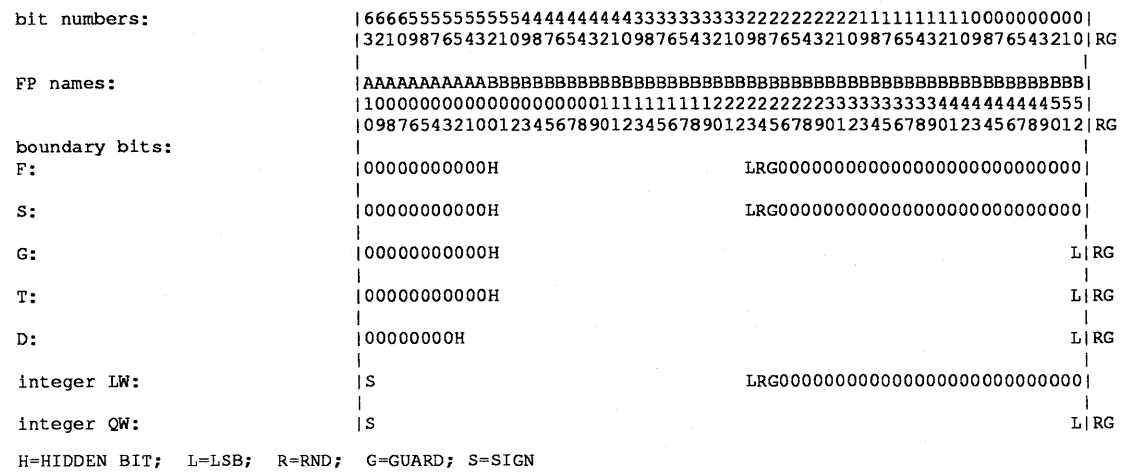
Exception signals from the ADD and MULTIPLY pipes are logically OR'ed and written into the FPCR when either **I%F_WE_FM_8A** or **I%F_WE_FA_8A** are asserted. Exception signals from both pipes are driven to the IBOX for use in updating the Exception Summary Register. ADD and MULTIPLY pipe exceptions are signalled only when **I%F_WE_FA_8A** and **I%F_WE_FM_8A** are asserted, respectively. The exception signals and their associated conditions are described in Table 3–8.

3.3.1.2 Internal Interface

3.3.1.2.1 Stage 1 Interface

Stage 1 of the ADD and MULTIPLY pipes each receive two operand busses from the FBOX interface section. The operand bus sources are selected by the interface bypass mux.

Figure 3–2: ADD Pipe Fraction Datapath Alignment/Format



The ADD pipe receives the A operand on **F_INT%ADD_OP_A_4B**<64:0> and the B operand on **F_INT%ADD_OP_B_4B**<64:0>. Operand A has been recoded in the interface to the ADD pipe fraction datapath operand format depicted in Figure 3–2. Operand B has been partially formatted and receives additional formatting in Stage 1 to get to its final fraction datapath format. For floating point datatypes (F,S,G,T,D), these busses represent the fraction portion, and the exponent is driven to stage 1 via **F_INT%ADD_EXP_A_4B**<10:0> and **F_INT%ADD_EXP_B_4B**<10:0>. For longword and quadword datatypes, the A and B operands are contained completely on **F_INT%ADD_OP_A_4B**<64:0> and **F_INT%ADD_OP_B_4B**<64:0>. The stage 1 normalization-shift-detect and trailing-zero sections use these busses directly.

Table 3–2: Exponent constants muxed onto Stage 1 Input Exponent Operand A

Instruction	Constant	Value (hexadecimal)
IEEE CVT float-float	IEEE Bias	3FF
VAX CVT float-float	VAX Bias	400
IEEE CVT float-quad	IEEE Bias + 52 (decimal)	433
VAX CVT float-quad	VAX Bias + 53 (decimal)	435

To allow the stage 1 exponent adder to begin in cycle 5A, special exponent bypass logic muxes several exponent constants onto the **F_INT%ADD_EXP_A_4B**<10:0> bus. As described in Table 3–2, these constants are used during CVT floating-floating IEEE and VAX, and CVT floating-quadword IEEE and VAX instructions. Due to implementation constraints, portions of these constants can "leak" onto the exponent field of the data in the fraction datapath during these instructions. The fraction data that gets modified in these cases is always 0, because the Alpha SRM requires that operand A be F31 for CVT instructions. During these CVTs the normalization-shift-detect logic is not using the operands. Therefore the Stage 1 output mux must select 0 for these cases rather than the original A operand, thus making the stage 1 exponent bypass details transparent to the remainder of the pipe. Figure 3–3 shows the conditions occurring in the interface and ADD Pipe Stage 1 for these cases.

The MULTIPLY pipe receives its A operand on **F_INT%MUL_OP_A_4B**<64:0> and the B operand on **F_INT%MUL_OP_B_4B**<64:0>. The format of these operands matches the floating point register file format.

3.3.1.2.2 Stage 3 Interface

Stage 3 of the ADD and MULTIPLY pipes supply result data to the register file and bypasses in register file format. The ADD pipe result bus is **F_AS3%FWR_8A**<64:0> and the MULTIPLY pipe result bus is **F_MS3%FWR_8A**<64:0>.

3.3.2 Interface Instruction Flows

Figure 3–3: STAGE 1 INPUT BYPASS/FORMAT/RESOURCE TABLE

OPERATION	LXD Y/N	TRZ Y/N	IN OP A	IN OP B	BYP OP A	BYP OP B	MINI-FORMAT FRACTION OP A	MINI-FORMAT FRACTION OP B	S1 F1 OUT	S1 F2 OUT	EXP FMT OP A	EXP FMT OP B	COMMENTS
EFF_ADD	N	Y	*	*	BYP	BYP	FP	FP	AF|BF	AF|BF	EXP	EXP	
EFF_SUB	Y	Y	*	*	BYP	BYP	FP	FP	AF|BF	FS|*F	EXP	EXP	valid for ediff
CMP*	N	N	*	*	BYP	BYP	FP	FP	0	0	EXP	EXP	
CVT IEEE FP-FP	N	Y	F31	*	X	BYP	X	FP	0	BF	0X3FF	EXP	
CVT VAX FP-FP	N	Y	F31	*	X	BYP	X	FP	0	BF	0X400	EXP	
CVTDG	N	Y	F31	*	X	BYP	X	QW	0	BF	0X400	D-EXP	
CVTQF VAX	Y	Y	F31	*	BYP	BYP	QW	QW	0	FS|BF	X	X	
CVTQF IEEE	Y	Y	F31	*	BYP	BYP	QW	QW	0	FS|BF	X	X	
CVTFQ VAX	N	Y	F31	*	X	BYP	X	FP	0	BF	0X435	EXP	
CVTFQ IEEE	N	Y	F31	*	X	BYP	X	FP	0	BF	0X433	EXP	
CVTLQ	N	N	F31	*	X	BYP	X	QW	0	BF	X	X	
CVTQL	N	Y	F31	*	X	BYP	X	QW	0	BF	X	X	
CPYS*	N	N	*	*	BYP		FP	FP	0	BF	EXP	EXP	
FCMOV*	N	N	*	*	BYP		FP	FP	0	BF	EXP	EXP	
MT_FPCR	N	N	*	*	BYP		FP	FP	0	BF	EXP	EXP	

QW : PASS ALL FRACTION BITS UNMODIFIED INTO LXD/TRZ/FIS.
FIS WILL FORMAT FRACTION PROPERLY FOR REST OF PIPE.

FP : bit B52 = !Z; ZERO EXPONENT FIELD OF FRACTION (FSGT types only)

EXP : EXTRACT EXPONENT FIELD OF FRACTION DP

D-EXP : EXTRACT EXPONENT FIELD OF FRACTION; ZERO UNUSED FRACTION BITS

A1 : STAGE 1 ADDER 1

A2 : STAGE 1 ADDER 2

AF : OPERAND A FRACTION

BF : OPERAND B FRACTION

FS : FRACTION SUM

LXD : NORMALIZATION-SHIFT-DETECT LOGIC

TRZ : TRAILING-ZERO-DETECT LOGIC

Table 3–3: ADD pipe: ADDx/CPYSx/CMPx/CVTx/FCMOVxx/FBXX/MX_FPCR/SUBx

Cycle	Description
3A	Instruction opcode and RF source addresses sent from IBOX on I%F_FA_INST_3A
3B	RF source address decode
4A	RF source operand read; Issue signal from IBOX on I%F_FA_ISSUE_4A
4B	RF sense amp; operand bypass; Pipe formatting
5A	F%I_BRN_TAKEN_5A and F%BRN_MISPREDICT_5A to IBOX for FBXX; F%KILL_CM_5A to IBOX for FCMOVxx
5A - 8B	——————— S1 - S3 ———————
7A	RF destination address from IBOX on I%F_FA_ADDR_7A
7B	RF decode
8A	Write enable from IBOX on I%F_WE_FA_8A
8B	RF write; FPCR write; Exceptions driven to IBOX; Operand bypass; Late abort in absence of RF write enable

Table 3–4: ADD pipe: DIVx

Cycle	Description
3A	Instruction opcode and RF source addresses sent from IBOX on I%F_FA_INST_3A
3B	RF source address decode
4A	RF source operand read
4B	RF sense amp; Operand bypass; Pipe formatting
5A	Operands driven to Divider in Stage 1
–	———— Divider runs for variable period ————
–	——— (I->F_FDIV_ABORT can be asserted at any time) ———
1B	"Divider done soon" signal sent to Ibox on F%I_DIV_DONE_SOON_1B
4A	No issue signal to ADD pipe from IBOX this cycle (DIV bubble)
5B	Exponent, controls re-enter ADD pipe S2 from Divider Hold Latch (DHL)
7A	RF destination address from IBOX on I%F_FA_ADDR_7A
7B	RF decode; Quotient register result enters Stage 3
8A	Write enable from IBOX on I%F_WE_FA_8A
8B	RF write; FPCR write; Exceptions driven to IBOX; Operand bypass; Late abort in absence of RF write enable

Table 3–5: MULTIPLY pipe: MULx/CPYS

Cycle	Description
3A	Instruction opcode and RF source addresses sent from IBOX on I%F_FM_INST_3A

Table 3–5 (Cont.): MULTIPLY pipe: MULx/CPYS

Cycle	Description
3B	RF source address decode
4A	RF source operand read
4B	RF sense amp; Operand bypass; Issue signal from IBOX on I%F_FM_ISSUE_4A
5A - 8B	——————— S1 - S3 ———————
7A	RF destination address from IBOX on I%F_FM_ADDR_7A
7B	RF decode
8A	Write enable from IBOX on I%F_WE_FM_8A
8B	RF write; FPCR write; Exceptions driven to IBOX; Operand bypass; Late abort in absence of RF write enable

Table 3–6: STORE port: STx

Cycle	Description
3A	Instruction opcode and RF source addresses sent from IBOX on I%F_ST_INST_3A
3B	RF source address decode
4A	RF source operand read
4B	RF sense amp; Operand bypass
5A	Drive data to store logic; Begin formatting and parity generation; Store bus enable from MBOX on M%F_FBOX_DRV_ENA_5A
5B	Finish store data formatting and parity generation
6A	Formatted data and parity driven to DCACHE on B%D_WR_DATA_6A<63:0> and B%D_WR_LW_PAR_6A<1:0>

Table 3–7: RF Load ports: LDx (LOADs and FILLs)

Cycle	Description
4B	RF dest. address driven from IBOX on I%F_LDX_ADDR_5A<>; Format info. from MBOX on M%F_LD_FORMATX_4B
5A	Load data driven from DCACHE on D%Z_DATA_X_5A<>; Decode RF dest. address and format info
5B	Format data into floating point RF format
6A	Complete exponent expansion/zero detect and drive to FBOX RF; WEs from IBOX on I%F_WE_LDX_6A
6B	RF write; Operand bypass; Absence of WE constitutes late abort

Table 3–8: FBOX Interface Signal List

Signal	Description
From FBOX:	
F%I_BRN_TAKEN_5A	Floating branch (FBx) condition evaluated TRUE
F%I_BR_MISPREDICT_5A	IBOX mispredicted branch condition result
F%I_DIV_DONE_SOON_1B	Divide result ready in seven cycles
F%I_KILL_CM_5A	Conditional move (FCMOVxx) condition evaluated FALSE
B%D_WR_DATA_6A<63:0>	FBOX store data bus
B%D_WR_LW_PAR_6A<1:0>	Fbox store data bus LW parity
F%I_FOV_FA_8B	Floating Overflow - ADD pipe
F%I_IOV_FA_8B	Integer Overflow - ADD pipe
F%I_FUN_FA_8B	Floating Underflow - ADD pipe
F%I_INE_FA_8B	Floating Inexact Result - ADD pipe
F%I_FDZ_FA_8B	Floating Divide by Zero - ADD pipe
F%I_INV_FA_8B	Invalid Operation - ADD pipe
F%I_SWC_FA_8B	Software Completion - ADD pipe
F%I_FOV_FM_8B	Floating Overflow - MULTIPLY pipe
F%I_FUN_FM_8B	Floating Underflow - MULTIPLY pipe
F%I_INE_FM_8B	Floating Inexact Result - MULTIPLY pipe
F%I_INV_FM_8B	Invalid Operation - MULTIPLY pipe
F%I_SWC_FM_8B	Software Completion - MULTIPLY pipe
To FBOX:	
D%Z_DATA_0_5A<63:0>	load data bus 0 from DCACHE
D%Z_DATA_1_5A<63:0>	load data bus 1 from DCACHE
I%F_BYP_FM_FMA_3B	Bypass controls
I%F_BYP_FA_FMA_3B	Bypass controls
I%F_BYP_FLD0_FMA_3B	Bypass controls
I%F_BYP_FLD1_FMA_3B	Bypass controls
I%F_BYP_FM_FMB_3B	Bypass controls
I%F_BYP_FA_FMB_3B	Bypass controls
I%F_BYP_FLD0_FMB_3B	Bypass controls
I%F_BYP_FLD1_FMB_3B	Bypass controls
I%F_BYP_FM_FAA_3B	Bypass controls
I%F_BYP_FA_FAA_3B	Bypass controls
I%F_BYP_LD0_FAA_3B	Bypass controls
I%F_BYP_LD1_FAA_3B	Bypass controls
I%F_BYP_FM_FAB_3B	Bypass controls
I%F_BYP_FA_FAB_3B	Bypass controls

Table 3–8 (Cont.): FBOX Interface Signal List

Signal	Description
I%F_BYP_LD0_FAB_3B	Bypass controls
I%F_BYP_LD1_FAB_3B	Bypass controls
I%F_FDIV_ABORT	Abort FBOX divide
I%F_LD0_ADDR_5A<4:0>	FBOX load bus 0 register address
I%F_LD1_ADDR_5A<4:0>	FBOX load bus 1 register address
I%F_WE_LD0_6A	FBOX load bus 0 write enable
I%F_WE_LD1_6A	FBOX load bus 1 write enable
I%F_WE_FA_8A	FBOX ADD pipe write enable
I%F_WE_FM_8A	FBOX MULTIPLY pipe write enable
M%F_LD_FORMAT0_4B<2:0>	FBOX load bus 0 format information
M%F_LD_FORMAT1_4B<2:0>	FBOX load bus 1 format information
I%F_FA_INST_3A	ADD pipe instruction - multiple bits
I%F_FM_INST_3A	MULTIPLY pipe instruction - multiple bits
I%F_ST_INST_3A	STORE port instruction - multiple bits
I%F_FA_ISSUE_4A	IBOX issue valid to ADD pipe
I%F_FM_ISSUE_4A	IBOX issue valid to MULTIPLY pipe
I%Z_BR_PREDICT_4A	IBOX predicted conditional branch taken

3.4 FBOX Multiplier Pipe

3.4.1 INTRODUCTION

In this section an overview of the FBOX Multiplier Pipe is presented. The multiplier pipe consists of a three stage pipelined execution unit.

The primary goal of this section is to demonstrate how various instructions can be executed using the three stage pipelined microarchitecture. A general overview and block diagram of each stage is given, followed by a description of the sequence of operations the FBOX performs to execute the floating point multiply instruction.

The appendices contain a description of the algorithms used to implement parallel rounding with addition, trailing 0 detection on the input operands, and a summary of the multiply pipe exception handling.

3.4.2 Multiply Pipe Overview

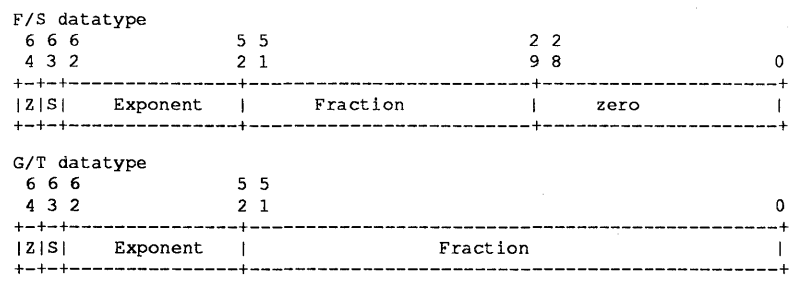
3.4.2.1 Interface

The register file contains 32 floating-point registers each 65 bits wide. The additional bit in each register is assigned to the Z bit. The bypass mux at the output of the register file selects the two operands from the register file, formatted load data, or the result (WR) sent from the pipeline. Depending on the datatype, the selected operand is formatted into the appropriate fraction, exponent, and sign fields of the pipelined stages.

The result of the floating-point operation is formatted to register file format in stage 3 of the pipeline and sent to the interface on the WR bus. This data is written to a register (WR latch) to enable reading the result in the same cycle as the register file write.

The floating point register file has 65 bits designated as RF<64:0>. The msb, RF<64>, is a z-bit that is set when the RF<62:0> field is all zeros. The z-bit does not cover the sign bit, RF<63>. The register file data format is shown in Figure 3–4.

Figure 3–4: Register File Data Format



3.4.2.2 MUL data path

The bits in the MUL fraction data path are numbered as follows:

```
Weight:   4  2  1   .5 .25
Bits:    A2 A1 A0 . B0 B1 ........   B55
                  ^ ^
                  | |_______ HIDDEN BIT FOR FLOATING POINT NUMBERS
                  \_________ BINARY POINT
```

BITS<A2:A1> are only significant during the multiply operation.

The exponent data path, E<12:0> is 13 bits wide, with E<12> representing the exponent sign bit. The pipeline also maintains a sign (N) and a zero (Z) bit for each operand.

Register file data is formatted onto the fraction and exponent datapaths depending on the datatype as follows:

BIT	F/S/G/T
A0	0
B0	NOT RF<64>
B1:B52	RF<51:0>
B53:B56	0
E<12:11>	0
E<10:0>	RF<62:52>

The sign and z-bits are taken directly from RF<64:63>.

3.4.2.3 Nomenclature

E — Exponent field.

F — Fraction field, including the hidden bit.

LXD — Vector of one out of 64 bits, MSB bit set indicates zero left shift, LSB set indicates left shift of 63.

ELXD — Encoded LXD in 13 bits; only the least significant 6 bits to encode 0 to 63 are generated in FDP, other bits are forced to zero in EDP.

AIN,BIN — Inputs to the FRACTION adders.

<r> — The round bit is a function of the floating point operation.

STKY — Logical OR of all bits shifted out of destination datapath width.

3.4.3 INSTRUCTION FLOWS

Following is an analysis of the pipeline flow. This analysis demonstrates what is done in each stage of the pipeline with emphasis on fraction and exponent computation. See Figure 3–5 for a block diagram of the multiplier pipe.

3.4.3.1 Floating Point Multiply

The multiplier array in stage 2 utilizes a radix-8 modified Booth algorithm, recoding three bits of the multiplier at a time. The computation is performed on two threads in parallel and the results are combined at the output of the array.

Figure 3–5: Multipy Pipe Block Diagram

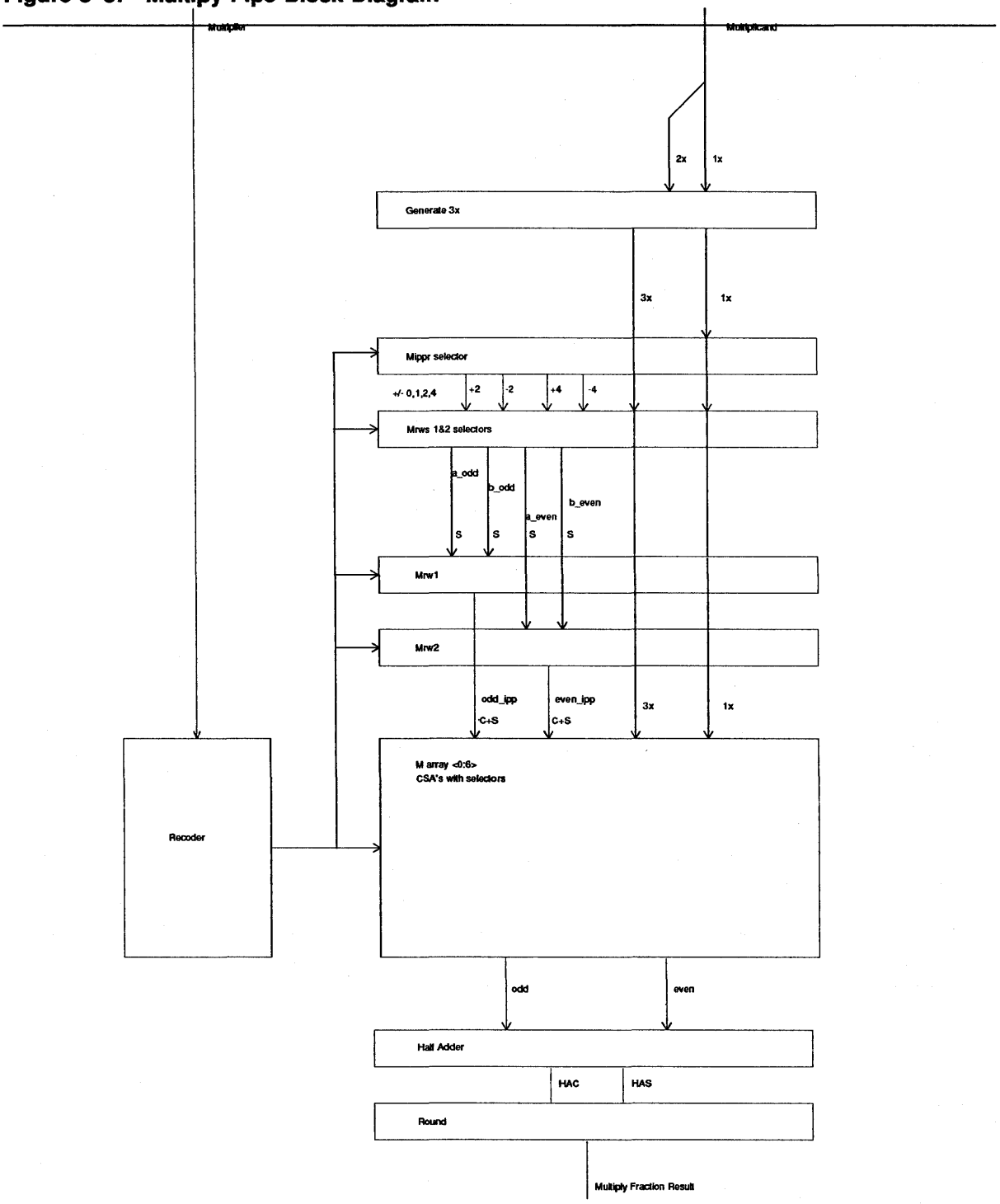


Table 3–9: Booth Algorithm

Mi+2	Mi+1	Mi	Mi-1	Operation	MRECODE +01234
0	0	0	0	+0 x Mcand	010000
0	0	1	0	+1 x Mcand	001000
0	1	0	0	+2 x Mcand	000100
0	1	1	0	+3 x Mcand	000010
1	0	0	0	-4 x Mcand	100001
1	0	1	0	-3 x Mcand	100010
1	1	0	0	-2 x Mcand	100100
1	1	1	0	-1 x Mcand	101000
0	0	0	1	+1 x Mcand	001000
0	0	1	1	+2 x Mcand	000100
0	1	0	1	+3 x Mcand	000010
0	1	1	1	+4 x Mcand	000001
1	0	0	1	-3 x Mcand	100010
1	0	1	1	-2 x Mcand	100100
1	1	0	1	-1 x Mcand	101000
1	1	1	1	-0 x Mcand	110000

Precomputation of three times the multiplicand and partial recode of the multiplier is performed in stage 1. The stage 1 recoder retires 12 bits of the muliplier and sets up the initial partial products for the first double row in the multiplier array. The trailing zero detection used in the sticky bit calculation for IEEE rounding is also performed in stage 1 by detecting the trailing 1 position.

The multiplier array is composed of double rows of carry sum adders that perform the addition of weighted multiplicands in parallel on odd and even threads of the array. The result of each computation is represented in sum and carry form. The product is obtained by using two additional carry sum adders to reduce the vector pairs to a single pair of sum and carry vectors. In order to facilitate parallel rounding in stage 3, the sum and carry vectors are passed through a half adder to generate a sum and carry vector pair with a place created for injecting the round bit.

The sticky bit calculation is done with a trailing zeros summing circuit. The sum of the number of trailing zeros in the multiplier and multplicand is compared with a fixed value (depending on data type) to determine the sticky bit. If the total number of trailing zeros in the muliplier and multiplicand span the sticky bit range of the product then the sticky bit is set to zero.

The carry out from the lower half of the stage 2 double precision multiply is added to the sum and carry vectors to generate a single precision result in stage 3.

3.4.4 Mul Pipe Stage 1

The first stage of the pipeline consists of fraction, exponent, and control sections as shown in Fig. Figure 3–6.

The fraction datapath consists of a 56b adder, trailing zero detect circuits for both the multiplier and multiplicand operands, and a set of MIPPR muxes for driving the initial partial products to row 1 (even and odd) of the multiply array. The operands do not require formatting since the MUL Pipe does not execute any conversions to or from integer format and are loaded directly from the output of the MUL Pipe bypasses. The recoder is located adjacent to the fraction datapath and is distributed throughout the first two stages of the MUL pipeline. The 56 bit fraction adder is used to precompute 3X the multiplicand. TR_PLIER and TR_CAND vectors which set a bit a the first trailing 1 are calculated. The sum of the trailing zeros in the product eTR_SUM is determined by adding eTR_PLIER and eTR_CAND (6 bit encodes of the position of the trailing one in the multiplier and the multiplicand). The eTR_PLIER and eTR_CAND values also serve to determine if the fraction sections of the operands are zero. This information is used to detect a dirty zero input for a multiply. The MIPPR muxes use the recoded multiplier data to set up several initial partial products for the multiplier array during multiply operations. The exponent datapath contains a 13-bit adder, and an output mux. The exponent sum is driven to Mul Stage 2 for multiplies, and the multiplicand exponent is output for CPYS. Note the stage 1 exponent result includes (2 * Bias). This is corrected in stage 2.

3.4.5 Mul Pipe Stage 2

A block diagram of stage 2 is shown in the Fig. Figure 3–7.

The fraction datapath of Mul Pipe stage 2 consists of a multiplier array. The multiplier consists of 8 double rows of multiplicand selectors and CSAs organized as odd and even threads. The multiplier recoder in stage 2 uses the multiplier from stage 1 to perform Booth recoding. Each of the two threads produces the product in two halves in sum and carry form. These are combined using two additional CSAs to produce a pair of sum and carry vectors (FS, FC). The two operand buses that carry 1x the multiplicand and the 3x the multiplicand through the datapath are used to set up the input addends for the CSA's. In addition, a half adder to enable parallel rounding is included at the output of the multiplier array. The Mul Sticky bit is determined by comparing eTR_SUM to a constant which is determined by the data type of the multiplication (52 for T/G, 81 for F/S). The exponent datapath consists of a 13-bit adder which subtracts the bias from the exponent sum calculated in Mul stage 1, and an output mux to provide the original exponent of the multiplicand operand for CPYS.

Figure 3–6: STAGE 1

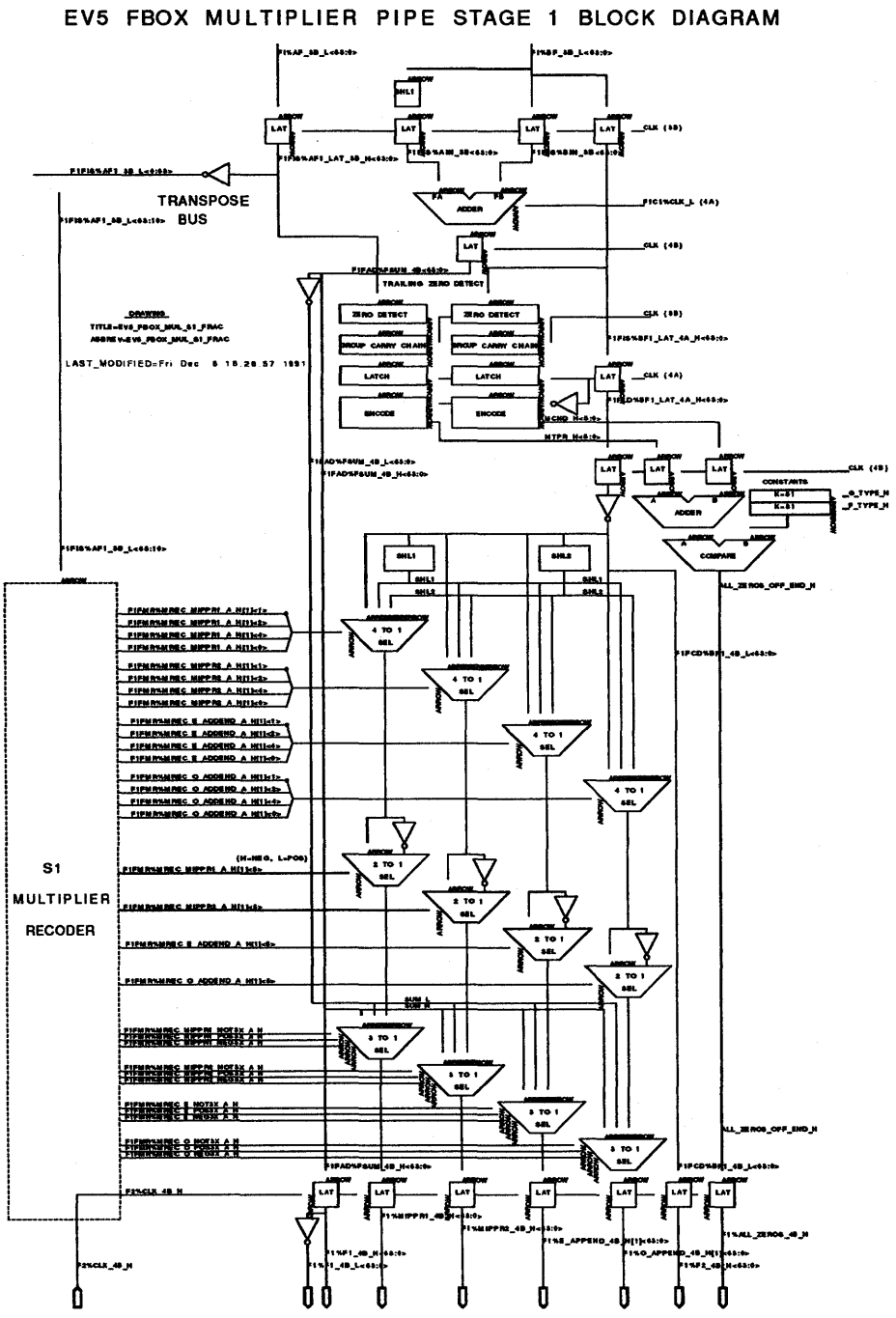
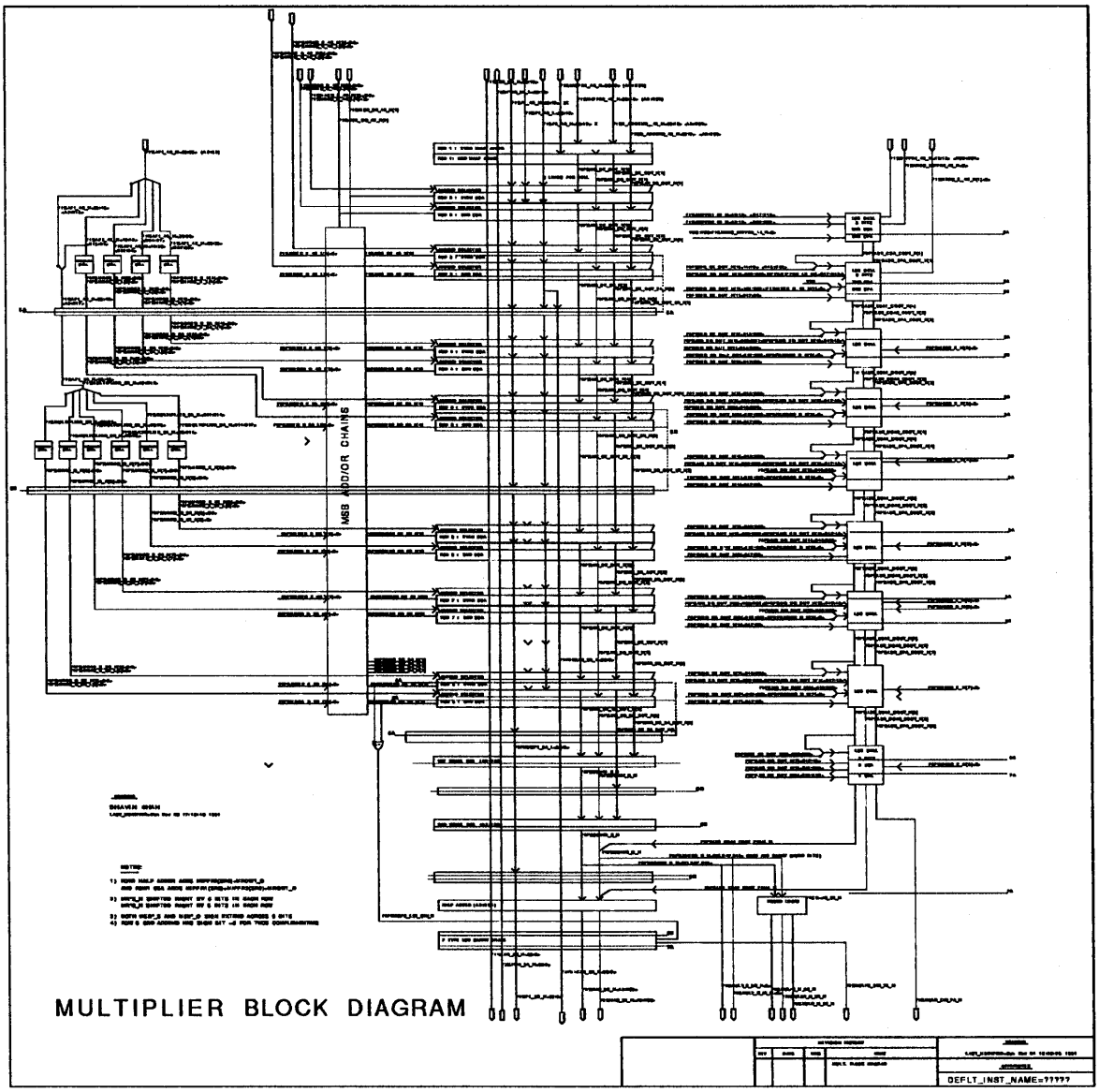


Figure 3–7: STAGE 2



3.4.6 Mul Pipe Stage 3

As the final stage of the pipeline, stage 3 computes the final result including the various roundings, calculates the z-bit result, detects exceptions and formats the result back to the register file format. Fig. Figure 3–8 shows the block diagram of stage 3 of the pipeline.

The fraction datapath of MUL PIPE stage 3 contains, an input selector, a 56 bit double adder, and an output mux. The input selector chooses the operands from the multiplier(FS, FC) or zero and the multiplicand for CPYS. The 56-bit double adder produces results either of the form A+B and A+B+1 for VAX rounding and IEEE round to nearest, or of the form A+B and A+B+2 for IEEE round to infinity. The final output mux chooses the results from the two adder results and performs a 1 bit normalization. The mux control is generated using the MSBs of the adders (MSB logic), and detection of fraction equal to exactly 1/2 in the higher order adder. The exponent datapath contains a 13-bit adder which is used to compute the Er-1 corresponding to the potential fraction outputs, floating overflow and underflow detection logic and an output selector. The control section of the MUL Pipe stage 3, performs exception detection, final sign computation, and Z bit calculation. The fraction, exponent, and sign (are already in register file format) are driven to the Floating Register file and bypasses.

3.4.7 Copy Sign

There are three instructions in this group, only CPSY is implemented in the multiply pipe in order to allow the compiler to generate a multiply pipe NOP. Instruction requires copying of the sign and exponent fields. The fraction field is unchanged. Exception checking is disabled for the copy sign instruction.

For CPYS (copy sign), the sign bit of register Fa is concatenated with the exponent and fraction bits from register Fb. The result is stored in register Fc.

3.4.7.1 Copy Sign - STAGE 1

The appropriate data is passed to stage 2 in both the fraction and exponent datapaths.

```
1%F1 = 0
1%F2 = FB

1%E1 = EA
```

The sign and z-bits associated with the input operands are piped to the next stage.

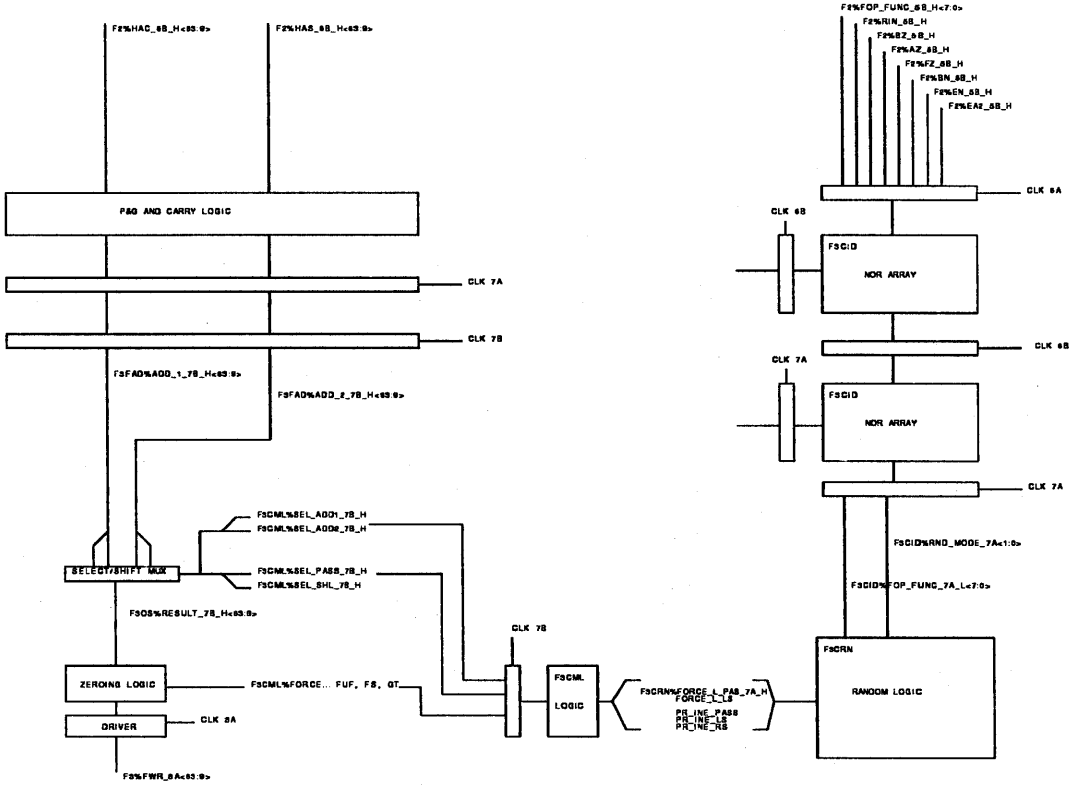
3.4.7.2 Copy Sign - STAGE 2

The operands are passed directly to stage 3.

```
2%F1 = 1%F1
2%F2 = 1%F2
2%E = 1%E
```

The sign and z-bits from the previous stage are piped to the next stage as well.

Figure 3–8: STAGE 3



EV5 MULTIPLIER FRACTION - STAGE 3

3.4.7.3 Copy Sign - STAGE 3

Stage 3 passes the stage 2 fraction and exponent outputs to the result. The sign bit is passed from stage 2.

```
F = 2%F2
E = 2%E
N = 2%BN
```

Since the z-bit calculation does not include the sign bit, the z-bit resulting from the CPYS instruction is equal to the z-bit associated with the Fb register operand;

3.4.8 Rounding

Table 3–10: Chop Rounding

Add1 <B0>	Add2 <B0>	Cin <r>	Output select	Shift	Actual Value	Add1= A+B+	Add2= A+B+
0	X	0	Add1	L	A+B	0	1
0	0	1	Add2	L	A+B+1	0	1
0	1	1	Add2	no	A+B+1	0	1
1	X	0	Add1	no	A+B	0	1
1	X	1	Add2	no	A+B+1	0	1

note: 1) Cin<r> is used to select Add1 or Add2

Table 3–11: Normal Rounding

Add1 <B0>	Add2 <B0>	Cin <g>	S and C <g>	S or C <g>	Output select	Shift	Actual Value	Add1= A+B+	Add2= A+B+
0	X	0	X	X	Add1	L	A+B+M	M	M+1
0	0	1	0	0	Add2	L	A+B+1	M	M+1
0	0	1	0	1	Add1	L	A+B+1	M	M+1
0	0	1	1	X	Add2	L	A+B+2	M	M+1
0	1	1	0	1	Add1	L	A+B+1	M	M+1
0	1	1	X	0	Add2	no	A+B+1	M	M+1
0	1	1	1	X	Add2	no	A+B+2	M	M+1
1	X	0	0	1	Add1	no	A+B+1	M	M+1
1	X	X	X	0	Add2	no	A+B+1	M	M+1
1	X	X	1	X	Add2	no	A+B+2	M	M+1
1	X	1	X	X	Add2	no	A+B+M+1	M	M+1

note: 1) M = S<g> + C<g>
2) Add1 and Add2 add all bits down to and including <r>
3) additional logic will detect if the first 1 is at the <r> or <g> position and if so may conditionally force the LSB of the rounded result to zero if in IEEE rounding mode

Table 3–12: Rounding to Infinity

Add1 <B0>	Add2 <B0>	STKY bit	Cin <r>	Add1 <r>	Output select	Shift	Force <LSB>	Actual Value	Add1= A+B+	Add2= A+B+
0	0	0	0	1	Add1	L	no	A+B	STKY	2+STKY
0	X	0	0	0	Add1	L	no	A+B	STKY	2+STKY
0	X	0	1	0	Add1	L	1	A+B+1	STKY	2+STKY
0	X	0	1	1	Add2	L	0	A+B+1	STKY	2+STKY
0	1	0	0	1	Add2	no	no	A+B+2	STKY	2+STKY
0	X	1	0	X	Add1	L	no	A+B+1	STKY	2+STKY
0	0	1	1	0	Add1	L	1	A+B+2	STKY	2+STKY
0	0	1	1	1	Add2	L	0	A+B+2	STKY	2+STKY
0	1	1	1	0	Add1	L	1	A+B+2	STKY	2+STKY
0	1	1	1	1	Add2	no	no	A+B+3	STKY	2+STKY
1	X	0	0	0	Add1	no	no	A+B	STKY	2+STKY
1	X	0	0	1	Add2	no	no	A+B+1	STKY	2+STKY
1	X	0	1	0	Add1	no	no	A+B+1	STKY	2+STKY
1	X	0	1	1	Add2	no	no	A+B+2	STKY	2+STKY
1	X	1	0	0	Add1	no	no	A+B+2	STKY	2+STKY
1	X	1	0	1	Add2	no	no	A+B+2	STKY	2+STKY
1	X	1	1	X	Add2	no	no	A+B+3	STKY	2+STKY

note: 1) STKY = 0 if all of the bits to the right of <r> are zero
2) Add1 and Add2 add all bits down to and including <r>

3.5 Reset and Initialization

3.6 Error Handling and Recording

3.7 Test Aspects

3.8 Performance Monitoring Features

3.9 Issues

3.10 Revision History

Table 3–13: Revision History

Who	When	Description of change
your name	date	description

August 1994

Chapter 1

The Mbox

1.1 Functional Description

The primary function of the Mbox is to process loads and stores issued by the IBOX by performing the following operations:

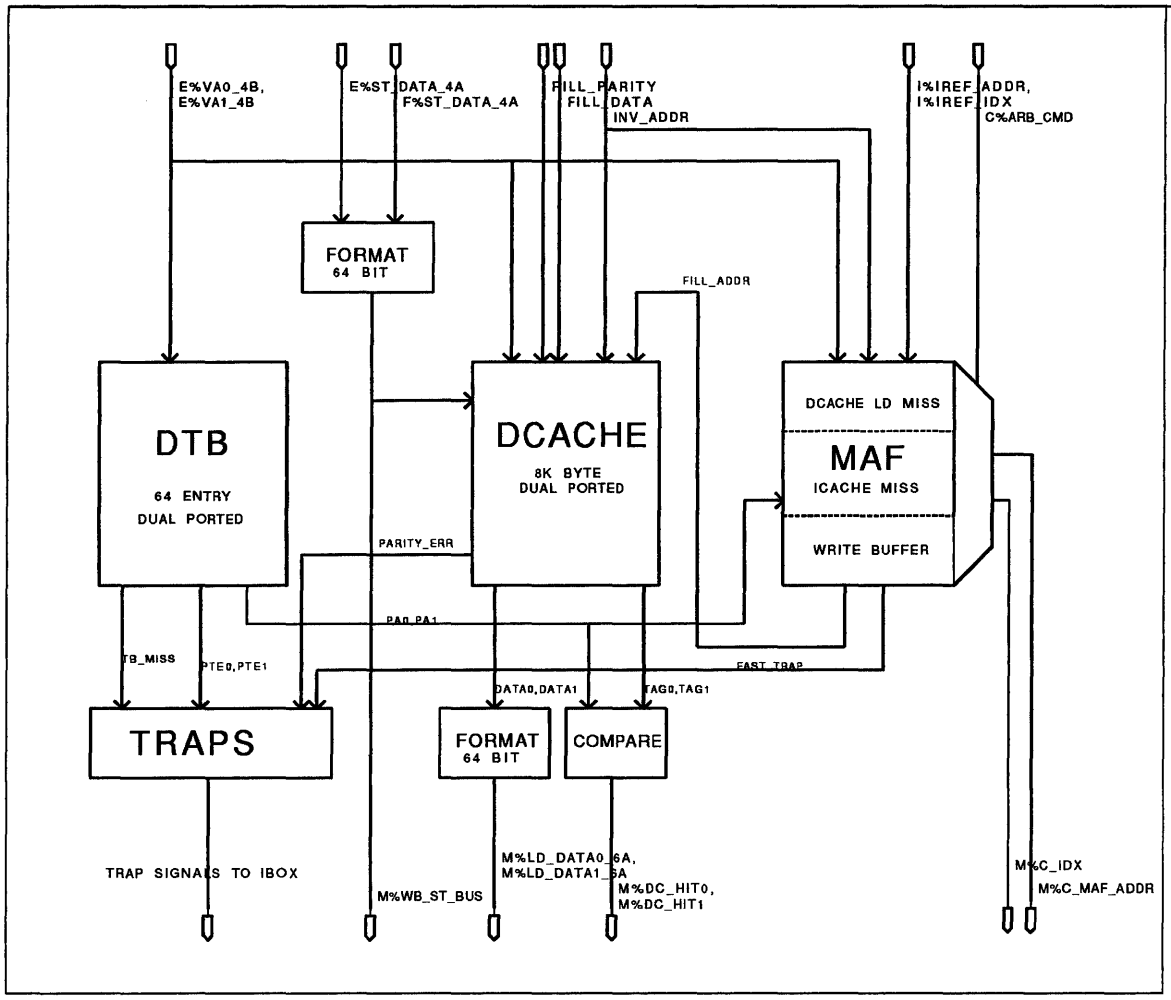
- Lookup the virtual to physical address translation in the Data Translation Buffer (DTB),
- Calculate Dcache hit or miss,
- Format and return Dcache hit data to the EBOX on loads,
- Queue up and merge loads that miss in the Dcache or Icache for issuing to the Scache,
- Format and return Dcache fill data to the EBOX register file,
- Control return of Dcache hit and fill data to the FBOX register file,
- Control stores to the Dcache,
- Queue up and merge stores in a write buffer for issuing to the Scache,
- Ensure strict ordering between reads and writes to the same address,
- Detect and report Dstream faults and Dcache parity errors.

The virtual addresses are calculated by the EBOX in either pipe0, pipe1, or both. LD's may come down either pipe, but ST's are only processed by pipe0. A ST and a LD will not be issued simultaneously.

The major sections of the Mbox include a dual-ported 64-entry Data Translation Buffer (DTB), fault and trap logic for each pipe, interface logic to support the control of a dual-ported 8-K Dcache (implemented as two 8K-byte single-ported Dcaches - one per pipe), and a Miss Address File (MAF), which holds addresses for outstanding Dcache read misses, Icache read misses and prefetches, and Dstream writes waiting to be processed by the CBOX. In addition, the MBOX has various control and status IPRs, the Processor Cycle Counter (PCC), and an instruction decode section which controls the rest of the MBOX.

The MBOX begins action in Stage 4 of the EV5 pipe. During this stage, the Dcache does a tag lookup for LDs and STs and a data lookup on LDs. Meanwhile, the virtual to physical address translation is being performed by the DTB.

Figure 1–1: Mbox



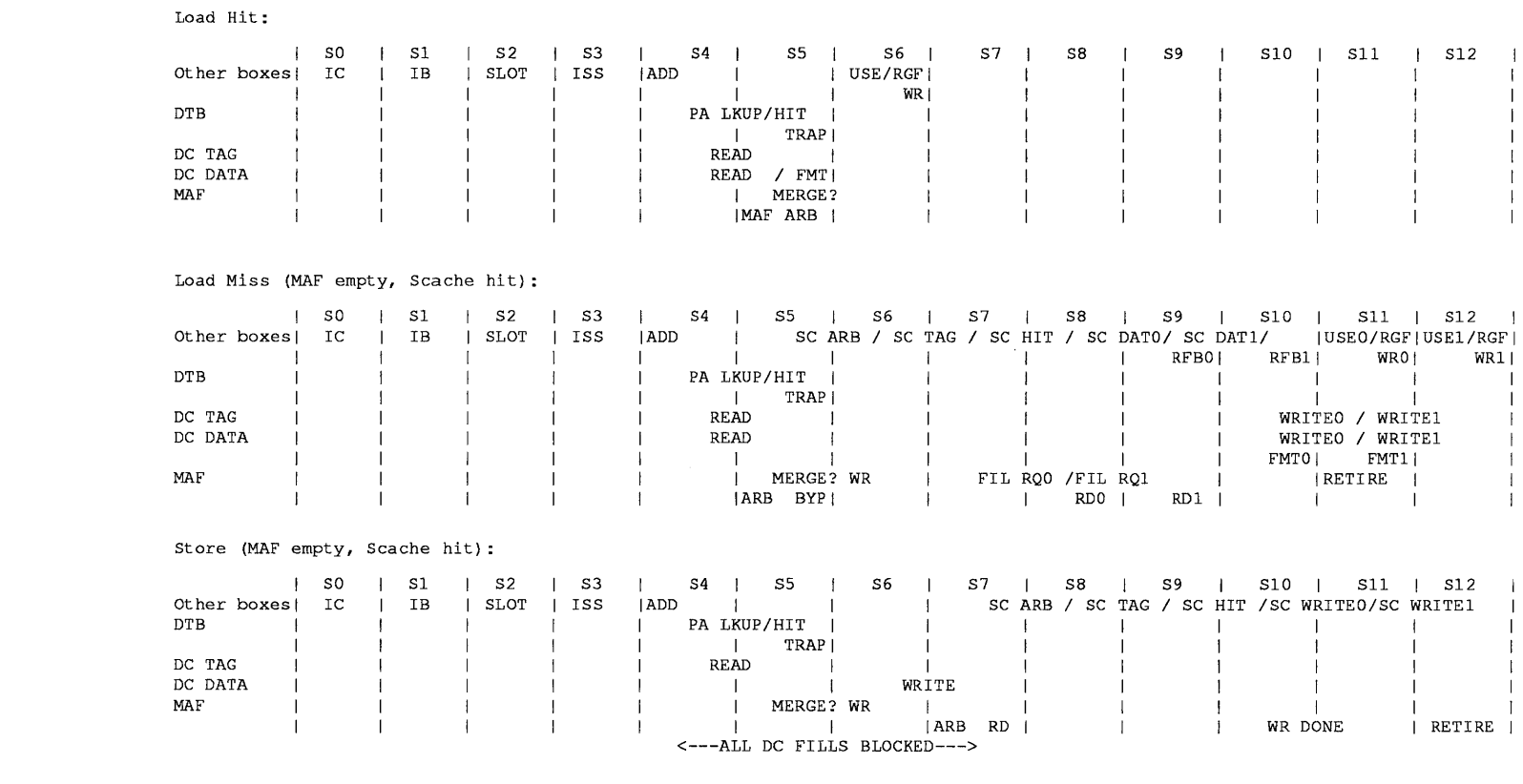
During Stage 5, LD data is formatted and driven to the EBOX (the FBOX handles floating point formatting) while the Dcache tag and the physical address from the DTB are compared to determine whether the address hit or missed in the Dcache. Also in Stage 5, memory management faults and parity errors are calculated. The MAF uses the physical address to determine if an incoming instruction merges with an existing MAF entry. It also does read-write conflict checking between the new address and addresses already in the MAF. Meanwhile, the MAF arbitrates between all outstanding memory references and the new reference to determine which shall be issued next to the CBOX for processing.

In stage 6, the MAF is updated based on the hit, merge and conflict results. For LDs that hit in the Dcache, the data is bypassed into the next instruction and also written into the register file. No entry is made in the Miss Address File. For LDs that missed in the Dcache, the reference either makes a new DREAD entry in the MAF, or is merged with an existing MAF entry. If the MAF has no memory references already waiting to be issued to the Cbox for an Scache lookup, then the new reference may be issued to the Cbox for a Stage 6 Scache tag lookup. For STs that

hit in the Dcache, the data is written to the Dcache in Stage 6. Regardless of whether the ST hit or missed in the Dcache, the ST address is placed in the Write Buffer (WB) section of the MAF, either as a new entry or merged with an existing entry. The ST data is sent to the CBOX along with an MAF index for entry into the CBOX data write buffer.

Figure 1–2 shows how the MBOX fits into the overall pipe.

Figure 1–2: MBOX Pipe



1.1.1 Instruction Descriptions

Table 1–1, Instructions Handled by the MBOX, contains the list of instructions the MBOX needs to handle and on which pipe they may be issued.

Table 1–1: Instructions Handled by the MBOX

Instruction Name	Pipe 0	Pipe 1
LDx - (LDL, LDQ, LDF, LDG, LDS, LDT)	X	X
LDQ_U	X	X
LDx_L - (LDL_L , LDQ_L)	X	
STx - (STL, STQ, STF, STG, STS, STT)	X	
STx_C - (STL_C , STQ_C)	X	
STQ_U	X	
FETCHx - (FETCH , FETCH_M)	X	
MB	X	
WMB	X	
RPCC	X	
HW_LD	X	X
HW_ST	X	
HW_MTPR	X	
HW_MFPR	X	

1.1.1.1 LDx - (LDL, LDQ, LDF, LDG, LDS, LDT)

The MBOX will accept up to two LDx instructions per cycle. These instructions may be issued in either pipe0 or pipe1. A STx may not be issued in the same cycle as a LDx. When two loads occur in the same cycle, the load in pipe0 will always be considered "earlier" than pipe1. A load cannot issue in the same cycle as a HW_MxPR instruction. These rules are important with regards to traps and read ordering.

When the MBOX receives the load address it is checked for the proper quadword or longword alignment, as appropriate. An ALIGN_ERR trap is generated if the address is not aligned properly.

The Dcache reads the tag and data arrays for each pipe in stage 4, using the index from **VA0** and **VA1**. In stage 5 the Dcache returns data to the FBOX and MBOX on the **DATA0** and **DATA1** busses, and the tag to the MBOX on **TAG0** and **TAG1**. The FBOX will format the data according to floating point type for floating point instructions. The MBOX will provide format controls to the FBOX. The MBOX will perform longword shifting and sign extension for integer loads and drive the formatted integer data to the EBOX on the **LD_DATA0** and **LD_DATA1** busses. Also in stage 5, the Dcache tag is compared against the physical address of the instruction (read from the Data Translation Buffer, or DTB) to determine whether the load hit in the Dcache. **DC_HIT_EX** must be asserted to the IBOX on a Dcache hit and deasserted on a miss. If the load hit, the actual write to the FBOX and EBOX register files will occur in stage 6.

The physical address of an incoming load is compared against every location in the Miss Address File (MAF), and on a Dcache miss the load is either merged with an existing request or allocated a new entry. The MAF will merge entries that are in the same 32 byte block but to different quadwords. Loads to the same quadword will trap (discussed later). The MAF will not merge floating point and integer requests. The MAF will also not merge longword load requests with

quadword load requests, nor longword load requests to odd addresses with longword load requests to even addresses. These restrictions were put in place to ease the implementation of the MAF.

There are 6 entries in the MAF. When there are 5 entries already allocated in the MAF, any new load in pipe1 will be forced to trap by asserting **MBOX_UNAVAIL** to the IBOX. A load in pipe0 will be allowed to allocate a new entry. If there are 6 entries already allocated in the MAF, then any new load will be trapped. Therefore, once a sixth entry has been allocated, no loads may be processed by the Mbox until an entry is retired from the MAF.

The addresses of all incoming loads are compared against all addresses in the MAF. If there is an exact match (to the exact longword/quadword), then the incoming load will be forced to trap, regardless of whether it would have hit or missed in the Dcache. This is to guarantee precise ordering of reads from the same address (as required by litmus test #1 in the ALPHA SRM). This trap is called the LD-MAF Conflict trap, and is discussed in Section 1.1.3.

If the index of an incoming load matches the index of a store in the immediately preceding cycle, and the store hit in the Dcache, then the load will be forced to trap. If the load would have hit in the Dcache, then the store will have updated the Dcache location by the time the load is issued the second time around. If the load would have missed, then it will miss again the second time around and will then be processed by the MAF. This trap is referred to as the Ld-ST Silo trap.

All load miss addresses are checked against every entry in the Write Buffer portion of the MAF, or WB. If a conflict is detected, the load instruction that conflicts will be stored in the MAF along with conflict bits indicating which entry(ies) in the WB caused the conflict. The WB will be forced to flush - that is, issue to the CBOX all outstanding writes up to and including the conflicting entry. When the conflicting WB entry is retired by the Cbox, the corresponding conflict bit(s) in the Dread portion of the MAF will be cleared. An entry in the MAF with any conflict bits set will be blocked from issuing to the Cbox until all conflict bits for that entry are clear. The corresponding Write Buffer entry will be set to NOMERGE to keep subsequent writes from passing the read.

Load requests will arbitrate with Ireads, writes and BIU requests for the Scache (see Table 1–22, MAF Issue Priority). If there are no outstanding requests pending, the MAF is bypassed. Pipe0 is the primary bypass of the MAF. Pipe1 may bypass the MAF if there is not a load in pipe0.

1.1.1.1.1 Dcache FILLs

When the load is passed to the CBOX, the index into the MAF is also sent. The fill data is returned with this index, an octaword address bit, and whether this is the first or second part of the fill. The index and octaword address are used to read out the register destination numbers, formatting information, status bits, and physical address out of the MAF. The register destination numbers are sent to the IBOX to use for register file fills. The physical address is sent to the Dcache on **DC_ADDR**<38:4> along with the FILL command. The octaword address and first/second fill information are used to determine which octaword valid bits should be set and these valid bits are driven to the Dcache along with a NOFILL indicator for each pipe. The Dcache is written the cycle after the fill data shows up on the fill bus (**RFB**) from the CBOX. The Dcache will drive the fill data on the **DATA0** and **DATA1** busses to the FBOX and MBOX. For integer fills, the MBOX will perform a longword shift and sign extend and drive the formatted data to the EBOX on the **LD_DATA0** and **LD_DATA1** busses. For floating point fills, the FBOX will do the formatting under MBOX control.

For a fill destined for the EBOX register file, the CBOX will notify the IBOX (and MBOX) to insert idle bubbles in the EBOX pipe to ensure slots for the fill to write the Dcache and the register file. For each octaword of fill data, the IBOX will insert one "no-op" bubble to ensure the EBOX register file write port is free, followed by one "no-MBOX" bubble to ensure the Dcache is free.

For fills destined to the FBOX, the MBOX will detect any loads or stores that want to use the Dcache or register file at the same time. If an incoming load coincides with either the cycle when the fill is writing the register file or the cycle when the fill is writing the Dcache, then it will be forced to miss and the fill will complete normally. If either the tag lookup, hit, or data write cycles of a store coincides with the cycle when the fill is writing the Dcache, then the fill will not be written to the Dcache but the data will still be forwarded and written to the register file.

Every FILL writes identical data to both 8K banks of the Dcache.

If an ECC error is detected on fill data from the Bcache or the external memory system, the CBOX will assert **M_RFB_ECC_ERR** to the MBOX. The MAF locks the register number and formatter control in a special holding register called the ECC error register and flushes the Dcache. When the ECC error occurs on the first part of a fill, the MAF sets the NOFILL bit for that DREAD entry to block the second half of the fill from writing the Dcache. For correctable ECC errors, the CBOX returns the corrected fill data with an ECC_FILL return status. The register number and formatter control are read from the ECC error register and the corrected fill data is forwarded and written to the register file. The Dcache is not written with the corrected data. Refer to Section 1.1.3.0.5 for a further description of ECC errors and the ECC_FILL operation.

LD's from "non-cacheable" memory (PA<39> = 1), will be forced to miss the Dcache and will be loaded into the MAF with the NOFILL bit set. On the returning fill, the NOFILL bit will be read from the MAF and force a bypass of the Dcache. Load merging for IO space addresses is done the same as load merging for non-IO space addresses. After an IO space read is issued to the CBOX, the CBOX will send the MBOX a FILL request at the time normally reserved for fills due to Scache hits. The MAF will read out the register numbers, formatting information, quadword request bits, and physical address in anticipation of the fill. The fill will be aborted (since I/O addresses will miss in the Scache), but the quadword request bits will be sent to the CBOX. The CBOX will forward these bits to the pins when sending out the IO read request to the system to identify the requested quadwords within the block. Once the FILL request is received from the CBOX, the MAF will disallow further merging to that entry. The actual fill will be completed later when the data is returned from the system.

IMPLEMENTATION NOTE

Bit 39 of the address is not part of the Dcache tag. Only data with PA<39> = 0 may reside in the Dcache. Therefore, the condition, PA<39> = 1, will force a Dcache miss and set the fill to NOFILL.

If MCSR<DC_FHIT> is set, all loads will be forced to hit in the Dcache, regardless of the value of PA<39>.

The CBOX will notify the IBOX to insert to insert idle bubbles in the pipeline when the initial FILL request is asserted to activate the quadword request bits.

EV5 provides minimal support for a "big endian" mode. When this mode is enabled, bit 2 of the address is inverted for all longword loads (LDL, LDS, LDF).

1.1.1.2 LDQ_U

The LDQ_U instruction is handled in the same manner as the LDQ instruction except that alignment traps are disabled. Note that address bits <2:0> are not actually cleared, but are ignored.

1.1.1.3 STx - (STL, STQ, STF, STG, STS, STT)

The MBOX will accept at most one STx instruction per cycle. The STx can be either a floating point or integer type store. The addresses for both types of stores will be processed by EBOX pipe0. The actual store data will come from the FBOX on floating point stores and from EBOX pipe0 on integer stores. Data will be driven to the Dcache on the **WR_DATA** bus by either the FBOX or MBOX as appropriate. The Dcache will forward this data on to the CBOX write data buffer. Parity is calculated on the data by the FBOX on floating point stores and by the MBOX on integer stores.

The Dcache tag lookup for the store will occur in Stage 4. **DC_HIT** is calculated in Stage 5, and the data is stored in the Dcache in Stage 6 if DC_HIT is true.

DC_HIT is calculated using the tag from the Dcache bank associated with pipe0. If DC_HIT is true, then both banks are written with the ST data. Otherwise, neither bank is written.

In the 2nd cycle following the issue of the store, the IBOX will not issue any LDs to the Dcache. In the cycle immediately following a store, the Ibox may issue a LD, (as long as there were not 2 consecutive stores). The address of this load is checked against the address of the preceding store. If there is an exact index match (down to the longword level), and the store hit in the Dcache, then the load will be trapped. Otherwise, the data will be read from the Dcache as normal. The trapped load will then be replayed by the IBOX, this time presumably hitting in the Dcache, since the store will have had time to complete. If a store is present in Stages 4, 5, or 6, any FILLs coming in from the CBOX (at stage 5) will not fill the Dcache.

Stores are checked for quadword or longword alignment, as appropriate. Improper alignment will cause an ALIGN_ERR trap.

Regardless of whether the store hits or misses in the Dcache, the write is entered into the Write Buffer (WB) section of the MAF, where it is queued up for issuing to the Scache and eventually the system. Subsequent stores may merge with a previous store in the WB if they are in the same 32 byte block. Entries in the WB will eventually be flushed to the Scache interface. The WB will send entries to the Scache when a 2nd entry is made to the WB, when a load conflicts with a WB entry, or every 64 cycles (when the 6-bit WB counter overflows). In the conflict case, the MAF will set a conflict bit and flush the WB until the conflict is resolved. The FETCHx, STx_C, MB and WMB instructions also inititate a flush of the write buffer.

When the store conflicts with a LD-miss entry already in the MAF, a conflict bit will be set in the WB entry corresponding to each MAF index that conflicts. As fills retire entries from the MAF, the corresponding conflict bits will be cleared in the WB entries. A WB entry will not be issued as long as any conflict bit is set in that entry. Conflict checking is not performed against Istream addresses.

All store addresses are checked against every entry in the WB. If the ST matches a WB entry but for some reason can not merge with the previous store, the new store instruction will be allocated an entry in the WB. A conflict bit is set in the WB entry to indicate the address of the newly allocated store conflicts with the address of a previous store. The conflicting store instruction

will not be allowed to issue until all previous stores have been retired. This ensures ordering of store/store operations is maintained in the system.

IMPLEMENTATION NOTE

The ST/ST conflict is implemented as a virtual WMB instruction by the Mbox. (The WMB bit is used to order ST/ST conflicts in the WB).

Once a write is issued to the Scache, the CBOX takes over that entry. No further merging is allowed to the WB entry after it is issued. The entry remains in the WB until the actual write is complete for conflict checking purposes. After the CBOX has written the entry into the Scache, the CBOX will return the index of the WB entry with a **WR_DONE** command. The MBOX will retire the entry from the WB and clear the corresponding conflict bits in the DREAD file. The CBOX has the means for requesting the MBOX to reissue a store after it was initially issued by the MBOX and accepted by the CBOX.

A store to "non-cacheable" memory space (**PA**<39>=1) will always miss the Dcache, unless the MCSR<DC_FHIT> mode bit is set. Otherwise, writes to "non-cacheable" memory space are treated the same as "cacheable" memory space by the MBOX. The CBOX handles any other differences in EV5 behavior for the two spaces.

EV5 provides minimal support for a "big endian" mode. When this mode is enabled, bit 2 of the address is inverted for all longword stores (STL, STS, STF).

When there have been 6 entries allocated in the WB, a new store will be forced to trap by asserting **MBOX_UNAVAIL** to the IBOX.

1.1.1.4 STQ_U

The STQ_U instruction is handled in the same manner as the STQ instruction except that alignment traps are disabled. It will only be issued to EBOX pipe0. Note that address bits <2:0> are not actually cleared, but are ignored.

1.1.1.5 MB

When the IBOX detects a memory barrier (MB), it stops issuing any MBOX instructions until the MBOX tells it to restart. The IBOX will issue the MB in pipe0 and will not issue any other MBOX instructions in the same cycle.

When the MBOX detects the MB instruction, it will flush all WB requests to the Cbox and allow all DRD requests to issue to the CBOX. The MBOX will wait until all DREAD and WB entries have been retired before issuing the MB command to the CBOX. This is to ensure proper ordering between the outstanding fills and writes with respect to new loads and stores issued once the MB finishes and restarts the IBOX. Note that the MB instruction does not affect IREADs, so there may be outstanding IREADs in the system.

The MBOX will receive acknowledgment from the CBOX (MB_DONE return status) and send the **MB_CLEAR** signal to the IBOX to restart instruction issue.

1.1.1.6 WMB

The IBOX will issue a WMB in pipe0. A store will never be issued in the same cycle with a WMB.

Unlike the normal MB, the WMB does not stop the IBOX from issuing further instructions. Its purpose is to ensure that all writes issued before the WMB finish before any writes issued after the WMB. Writes issued after the WMB may not merge with writes issued before.

When a WMB is received by the MBOX, the MAF sets all existing WB entries to be non-mergeable, so subsequent writes will make new entries. Since the Cbox cannot guarantee that it will complete all writes in the order they were issued to the CBOX, the MAF will stall the issuing of all writes received after the WMB until all previous writes have been retired from the WB.

The WMB command will not be issued to the CBOX or the pin interface.

1.1.1.7 RPCC

The RPCC instruction will return the value in the Processor Cycle Counter register on the **LD_DATA0** bus with the same timing as a LD instruction that hit in the Dcache. The IBOX will only issue the RPCC instruction down EBOX pipe0.

The signal **DC_HIT_E0** must be asserted to the IBOX to emulate a Dcache hit.

Refer to Section 1.1.4, Processor Cycle Counter, for more details on the Processor Cycle Counter.

1.1.1.8 LDx_L - (LDL_L , LDQ_L)

When the IBOX detects a LDx_L instruction, it stops issuing any MBOX instructions until the MBOX tells it to restart. The IBOX will issue the LDx_L in pipe0 and will not issue any other MBOX instructions in the same cycle.

LDx_L instructions are forced to miss in the Dcache. (**DC_HIT_E0** to the IBOX must be cleared).

LDx_L commands will always allocate a new MAF entry regardless of whether it could have merged with previous entries. This is to prevent it from merging with an entry that has already been issued with the DREAD command instead of the LDx_L command. In order to prevent subsequent LDx instructions from matching multiple entries in the DREAD file, merging to the LDx_L entry is not allowed. The LDx_L instruction will cause the top entry of the write buffer to arbitrate at low priority. If there is one entry pending in the WB when the LDx_L is executed, this feature improves the performance of the LDx_L/STx_C sequence by clearing the WB in preparation for the expected STx_C instruction.

When the LDx_L command is issued to the CBOX, the CBOX will take care of locking the associated address and setting the lock flag. The MBOX will send the MB_CLEAR instruction to the IBOX to restart instruction issue after the LDx_L has been issued and accepted by the CBOX (the command is past the RETRY point). The CBOX will return the LDx_L data to the MBOX like an ordinary FILL.

All the same conflict checking and traps that apply to normal LDx's apply to the LDx_L as well.

EV5 provides minimal support for a "big endian" mode. When this mode is enabled, bit 2 of the address is inverted for all LDL_L instructions.

NOTE

There is another version of the LDx_L instruction implemented by the HW_LD instruction (HW_LD_L). This can be LW, QW, virtual or physical and can be issued down either pipe0 or pipe1. PALcode will guarantee the HW_LD_L can not be dual-issued with any other MBOX instruction. (The IBOX does not support simultaneous issue of "MB-Class" instructions).

1.1.1.9 STx_C - (STL_C, STQ_C)

A STx_C instruction will only be issued by the IBOX on EBOX pipe0. The IBOX will stop issuing further MBOX commands until it is told to restart by the MBOX. The Dcache interface will process the STx_C instruction like a normal store. Fills that coincide with the previous, current and following cycle of the STx_C will be blocked from writing the dcache.

STx_C addresses are stored in the WB section of the MAF. The address will always be allocated a new entry and no merging will be allowed to that entry. STx_C will set the FLUSH bit to force the WB to empty.

The STx_C register number is stored in a special latch in the control section of the Miss Address File, so that when notification of the STx_C passing or failing arrives from the CBOX, the appropriate register may be written with the status of the STx_C.

Just like a normal store, the STx_C address will be checked against the MAF for conflicts and store the appropriate conflict bits in the WB to ensure all prior Dstream read and write requests to the same block are processed by the CBOX first.

The STx_C is issued to the CBOX using the STx_C command so that the CBOX will condition the write with the lock bit value. When the STx_C completes, the CBOX will send back the value of the lock bit on a dedicated wire along with the STx_C_DONE command. The result bit will be returned to the EBOX register file by driving it onto the **LD_DATA0** bus in the MBOX, and the register number will be sent to the IBOX. The CBOX will assert the **RFB_DATA_VALID** signal on a STx_C_DONE so that the **FILL_VALID** signal will be asserted by the MBOX when it sends the register number to the IBOX. The MBOX will restart the IBOX at this time by asserting **MB_CLEAR**. The CBOX will notify the IBOX (and MBOX) to insert an idle bubble in the EBOX pipe to ensure there is a slot to write the register file with the STx_C result.

EV5 provides minimal support for a "big endian" mode. When this mode is enabled, bit 2 of the address is inverted for all STL_C instructions.

The lock bit is required to be cleared by PALcode on certain exceptions. This may be accomplished by issuing a HW_ST with the PHYS and COND bits set. (The HW_ST_C is to a bit bucket address). This is guaranteed to clear the lock bit since any previous LDx_L instructions will be issued to the CBOX ahead of the HW_ST_C.

NOTE

There is another version of the STx_C instruction implemented by the HW_ST instruction. This can be LW, QW, virtual or physical and can only be issued down pipe0.

1.1.1.10 HW_MFPR

IPR reads from MBOX registers will have timing similar to a load hit. Data is returned to the EBOX register file on the **LD_DATA0** bus and **DC_HIT_E0** is asserted to the IBOX.

1.1.1.11 HW_MTPR

An IPR write to an MBOX register will be treated like a store except it will not be entered into the WB and will not modify the Dcache (unless the destination is a Dcache IPR). As long as the Ra and Rb fields of the HW_MTPR are the same, IPR write data will be driven from the EBOX to the MBOX on both the **ST_DATA** and the **VA0** buses.

IMPLEMENTATION NOTE

For implementation reasons, the write data for the MBOX IPRs is taken from the **VA0** bus. The **ST_DATA** is ignored by the MBOX on HW_MTPR instructions.

1.1.1.12 FETCHx - (FETCH , FETCH_M)

The IBOX will issue a FETCHx instruction in pipe0. FETCHx instructions will be loaded into the WB and will be issued to the CBOX with the FETCH or FETCH_M command. A FETCHx instruction will inititate a flush of the write buffer at high priority. Merges will not be allowed to FETCHx entries. The FETCHx will be removed from the WB only after receiving a FETCH_ DONE command from the CBOX. No data is returned to the MBOX on a FETCHx.

TB_Misses and ACC_VIO's are generated like a LDx for the FETCH and like a STx for the FETCH_M. BAD_VA traps will be generated, but alignment checks will be disabled.

NOTE

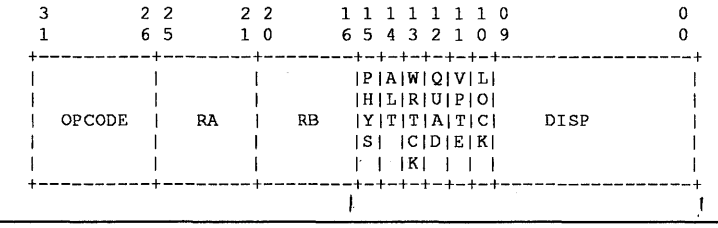
A side effect of putting the FETCHx instructions in the write buffer is that normal conflict checking will be performed between the incoming FETCH and the MAF and also between incoming loads and the FETCH in the write buffer. This will have the consequence of ordering FETCH and LD instructions, except in the case of a LD and FETCH issued in the same cycle. In this case, no conflict checking is performed.

1.1.1.13 HW_LD

The HW_LD is an implementation-specific instruction that is used to implement some load variations not accounted for in the Alpha SRM. It is an unaligned load, which means that the alignment trap is disabled.

The following diagram shows the instruction decode for the HW_LD instruction.

Figure 1–3: HW_LD instruction



The HW_LD instruction takes on different behavior depending on which bits in the instruction are set. These bits and their effect on behavior are described below.

Table 1–2: HW_LD Format

Field	Load Behavioral Description
PHYS	0 - The address from the Ebox is virtual.
	1 - The address from the Ebox is physical. The DTB is bypassed and memory management access checks are inhibited.
ALT	0 - Memory management checks use PS current mode bits.
	1 - Memory management checks use ALT_MODE IPR.
WRTCK	0 - Memory management checks FOR and read access violations.
	1 - Memory management checks FOR, FOW, read and write access violations.
QUAD	0 - Length is longword.
	1 - Length is quadword.
VPTE	1 - Flags a virtual PTE fetch. Used by trap logic to distinguish single TBmiss from double TBmiss. Memory management checks are done against KMODE.
LOCK	1 - Load_lock version of HW_LD. Will be issued on Pipe0 only, as ensured by PALcode.

The HW_LD instruction is treated just like a normal LDx (LOCK=0) or LDx_L (LOCK=1), except for the address translation and access checks as specified in the decode of the opcode. PALcode can not dual-issue a HW_LD_L instruction with any other MBOX instruction.

The QUAD bit distinguishes between longword and quadword versions of this instruction.

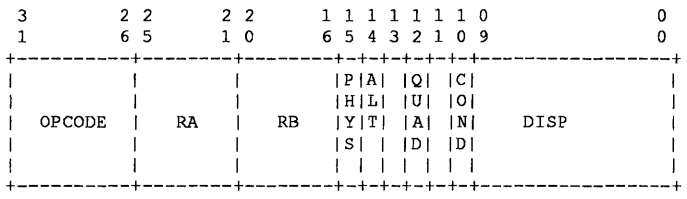
Memory management traps are generated according to the setting of the PHYS, ALT,WRTCHK, and VPTE bits. If the PHYS bit is set, then all memory management traps are disabled and the DTB is bypassed. If the ALT bit is set, then the read and write access checks are performed against the Current Mode bits from the ALT_MODE register rather than those from the Processor Status. If WRTCK is set, then write access checks are performed in addition to the read access checks. If VPTE is set, access checks are done against KMODE rather than using the Processor Status mode. If a DTB_MISS is detected on a HW_LD with the VPTE bit set, then the signal **IN_TB_FLOW** will be asserted to the IBOX along with the normal trap indicators.

1.1.1.14 HW_ST

The HW_ST is an implementation-specific instruction that implements some store variations not accounted for in the SRM. It is an unaligned store, which means the alignment trap is disabled.

The following diagram shows the instruction decode for the HW_ST instruction.

Figure 1–4: HW_ST instruction



The HW_ST instruction takes on different behavior depending on which bits in the instruction are set. These bits and their effect on behavior are described below.

Table 1–3: HW_ST Format

HW_ST Field	Store Behavioral Description
PHYS	0 - The address from the Ebox is virtual.
	1 - The address from the Ebox is physical. The DTB is bypassed and memory management access checks are inhibited.
ALT	0 - Memory management checks use PS current mode bits.
	1 - Memory management checks use ALT_MODE IPR.
QUAD	0 - Length is longword.
	1 - Length is quadword.
COND	1 - Store_conditional version of HW_ST. In this case, Ra will be written with the value of LOCK_FLAG.

HW_ST addresses are stored in the WB section of the MAF. The address can be physical or virtual depending on the PHYS bit of the opcode. All HW_ST's, except the version with the COND bit set, will be handled like any other STx, except for the address translation and access checks. If the COND bit is set, the HW_ST will treated as a STx_C.

The QUAD bit distinguishes between longword and quadword versions of this instruction.

If the PHYS bit is set then the address received from the EBOX is physical, so the DTB is bypassed and memory management traps are disabled. If the ALT bit is set, the trap logic will use the bits from the ALT_MODE register instead of the PS Current Mode when doing read/write access checks.

1.1.2 Memory Management

1.1.2.1 Data Translation Buffer

EV5 contains a 64-entry fully associative dual ported translation buffer. The DTB caches recently used data stream page table entries for 8Kbyte pages. Two addresses can be translated simultaneously for loads and one address for stores. In addition, each of the entries supports all four granularity hint options, i.e. 1, 8, 64, or 512 pages as described in section 6.5 of the ALPHA SRM V4.0. The operating system via PALcode is responsible for insuring that translation buffer entries, including super page regions, do not map overlapping virtual address regions at the same time.

In addition, EV5 provides an extension referred to as the super page, which can be enabled via bits in the MCSR register. Super page mappings provide virtual to physical address translation for two regions of the virtual address space. The first mode (SP0) maps a 30-bit region of the total physical address space to a single corresponding region of virtual space defined by VA<42:30> = 1FFE(Hex). In this mode, if VA<42:30> = 1FFE(Hex), then PA<39:30> is forced to 0 and VA<29:13> is copied to PA<29:13>. The second mode (SP1) enables superpage mapping when the virtual address bits <42:41> = 2. The entire physical address space is mapped multiple times over to one quadrant of the virtual address space defined by VA<42:41> = 2. Address translation in this mode is done by copying VA<39:13> to PA<39:13>. No DTB miss traps are generated during a superpage translation.

Super page translation is only allowed in kernel mode. The translation will fault if the super page translation is attempted while not in kernel mode. This is accomplished by forcing all the protection bits except the KRE and KWE bits to "0" when the super page translation is attempted. The KRE and KWE bits are forced to "1". The DTB is bypassed during a superpage translation.

For load and store instructions, the effective 43 bit virtual address is presented to the DTB, one address for each pipe. If the PTE's of the supplied virtual addresses are cached in the DTB, the PFN and protection bits for the page associated with that address are used by the MBOX to complete the address translation and access checks. If either of the addresses misses in the DTB, then a trap to PALcode is generated. PALcode is responsible for filling the DTB.

Each of the 64 DTB entries can support all 4 granularity hint bit (GH) page size options. At the time when the PTE is written to the DTB, the GH bits are decoded and used to disable the compare on a subset of Virtual Address bits <21:13>. The number of address bits disabled corresponds to the size of the page indicated by the GH bits. The GH bits themselves are stored in the DTB array. When a load or store instruction is presented to the DTB, only those address bits that are enabled by the GH bits of each entry are compared with the Virtual Address of the instruction. If a match occurs on these bits, then the corresponding PFN and Protection bits are read out of the DTB. A subset of the tranlated physical address, bits <21:13>, are replaced with the corresponding bits of the Virtual Address according to the page size specified by the granularity hint bits.

Table 1–4: Granularity Hint Bit Mapping

GH<1>	GH<0>	Page Size	Physical Address of Page	Address within Page
0	0	8K bytes	PA<39:13>	PA<12:0>
0	1	64K bytes	PA<39:16>	PA<15:0>

Granularity Hint Bit Mapping

Physical Address of Page	Address within Page
19>	PA<18:0>
22>	PA<21:0>

is that have the PHYS bit set, the DTB is bypassed altogether. ss (VA) lines is the actual physical address and is transferred s (PA) lines. No DTB miss or memory management traps are

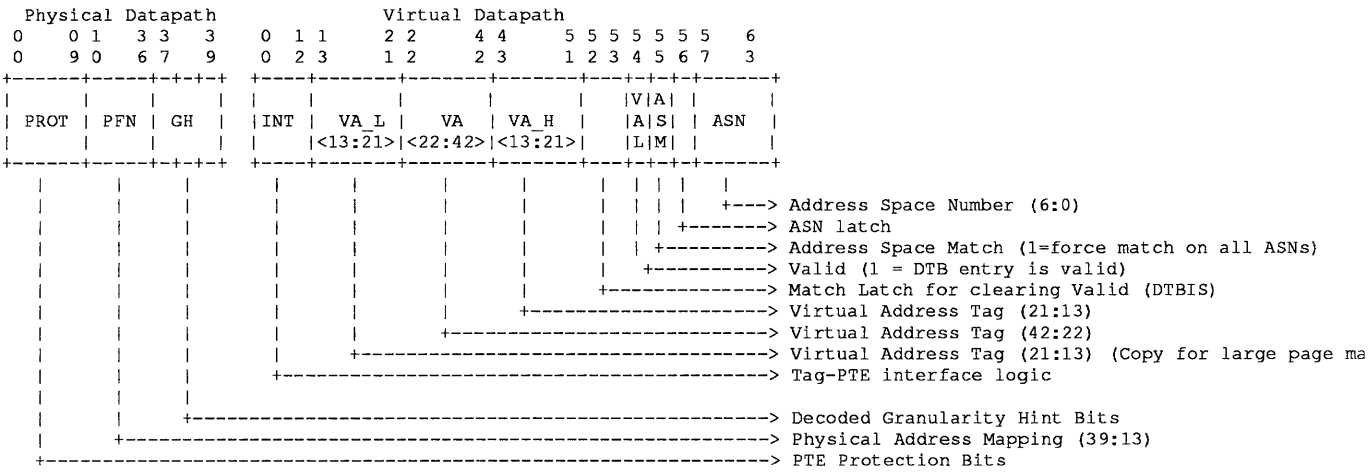
NOTE

instructions take precedence over superpage translation. to PA<39:13> on a HW_LD/ST instruction even in SP0 mode.

Number (ASN) bits. When a tag is loaded into the DTB on a TB to a field in the same entry. The tag for a DTB entry may only if the ASN bits in the DTB match the current ASN. Each DTB ich, when set, forces the ASN stored in the DTB to always match t is a part of the PTE written to the DTB on a TB fill.

are physically arranged in the DTB datapaths as follows:

Figure



The translation buffer uses a not_last_used replacement algorithm. This is implemented via a round-robin pointer which is initialized to the 1st DTB entry on RESET or a DTBIA. This pointer is bumped to the next entry whenever PAL code finishes a DTB fill for an entry, or when a DTB hit is detected for the entry currently being pointed to. The pointer always points to the next entry to be filled upon detection of a DTB_MISS.

NOTE

For load/store operations, the pointer bump is done before the Ibox trap point. When the normal conditions for bumping the pointer are met, this has the effect of bumping the NLU pointer on a trapping virtual address instruction or on a virtual address instruction in the cycle following a trapping instruction.

The DTB is filled and maintained by PALcode. The chapter on PALcode in the external spec details the DTB miss flows. The DTB is filled via the HW_MTPR instruction to the DTB_TAG and DTB_PTE IPRs. The virtual address bus for pipe0 is used to carry the data read from the EBOX register specified in the Ra field of the HW_MTPR instruction, to the DTB. A HW_MTPR to the DTB_PTE is first executed, which loads the PTE into a temporary register. When a HW_MTPR to the DTB_TAG is executed, the data from the temporary PTE register is loaded into the PTE side of the DTB array at the same time as the tag is written. The valid bit for this entry will be set by hardware at this time. The HW_MTPR to the DTB_TAG will cause the pointer to be bumped.

There are 3 types of invalidates for the DTB. They are DTB-Invalidate-All (DTBIA), DTB-Invalidate-All-Process (DTBIAP) and DTB-Invalidate-Single (DTBIS). Each of these commands is issued via the HW_MTPR instruction; each has its own IPR encoding. When DTBIA is detected (HW_MTPR DTBIA), all the valid bits in the DTB are cleared. When DTBIAP is detected (HW_MTPR DTBIAP), any entry whose ASM bit is clear will have its valid bit cleared. When DTBIS is detected (HW_MTPR DTBIS), the EBOX will drive the contents of the register addressed in the Rb field of the instruction onto the pipe0 VA lines. If any entry in the DTB matches the value on the VA lines, then that entry's valid bit is cleared. The DTBIS invalidate occurs in Stage 5A of the HW_MTPR instruction. The DTBIAP and DTBIA invalidates occur in Stage 7A of the HW_MTPR instruction. Because the DTBIS is executed prior to the IBOX trap point, a special IBOX kill signal, **KILL_DTBIS**, is used to abort the HW_MTPR DTBIS. The DTBIS will be aborted when the IBOX detects a PC mispredict or an ITB Miss trap, or when the HW_MTPR is issued during user mode.

The valid bits in the DTB array are not cleared by hardware on RESET. PALcode will clear the DTB using the HW_MTPR_DTBIA instruction.

1.1.3 Traps

The following traps will be detected by the MBOX.

NOTE

These are not the actual trap entry points recognized by PALcode. For a complete description of EV5 traps, PAL entry points and priorities, please see <REFERENCE>(ev5_IBOX_trap_section\full).

Table 1–5: Traps Detected by the MBOX

Trap Name	Trap Descriptions
FOR	Fault on Read: LD or FETCH from an address whose PTE has the FOR bit set.
FOW	Fault on Write: ST, FETCH_M, or LDQ/AW to/from an address whose PTE has the FOW bit set

Table 1–5 (Cont.): Traps Detected by the MBOX

Trap Name	Trap Descriptions
ACCVIO	Access Violation: An access occurred to an address whose protection bits were set up in such a way as to forbid that access.
BAD_VA	Bad Virtual Address: Bits <63:43> of the Virtual Address are not a sign-extension of bit <42>.
ALIGN_ERR	Bits <2:0> of the address of a quadword LD or ST are not all 0, or bits <1:0> of a longword LD or ST are not all 0.
DTB Miss Single	The VA of a Dstream access does not have a valid translation in the DTB and the instruction occurred outside the TB miss PALcode flow.
DTB Miss Double	The VA of a Dstream access does not have a valid translation in the DTB and the instruction was in a TB miss PALcode flow. This trap will only occur on the HW_LD instruction with the VPTE bit set.
MAF Full	The Dstream read portion of the Miss Address File is full.
WB Full	The Write Buffer section of the Miss Address File is full.
LD-MAF Conflict	The physical address of a LD instruction already has an outstanding LD miss in the Dread section of the Miss Address File. The fill must complete before the new LD can be processed. This is to guarantee exact ordering for reads from the same address - (litmus test #1).
LD-ST Silo Conflict	The Dcache index of a load matches that of a store that hit in the immediately preceding cycle.
Dcache Parity Error	The Dcache detected a parity error on either the tag or data read out of the Dcache on a Dcache reference.

All traps generated by the MBOX, IBOX, or FBOX, except the Dcache Parity Error, must inhibit the MBOX from updating any state. This includes, but is not limited to: writing the MAF in cycle 6A, writing the Dcache in cycle 6B on stores and issuing commands to the CBOX in cycle 6A. Any instruction in the same stage of the pipe as the trapping instruction must be aborted if it is later in time. If it is earlier in time, then it must be allowed to complete. The imprecise traps (certain FBOX traps, CBOX Fill errors, Dcache parity errors) will be resolved by the IBOX and will appear to the MBOX as IBOX traps. Any trap on an instruction in Stage 6 of the pipeline must abort any instructions (that are later in time) in Stages 3 through 6 of the pipe. Since the MBOX pipe begins at Stage 4, this will require aborting Stage 4 for 2 cycles.

Table 1–6 shows the traps detected by the MBOX and the resulting signals sent to the IBOX. The actions taken by the IBOX when one or more of these signals is asserted is described in <REFERENCE>(ev5_IBOX_trap_section\full).

Table 1–6: Trap Signals to IBOX (One pipe shown)

Traps detected by MBOX	CYCLE	dmm_err	align_err	in_tb_flow	dtb_miss	MBOX_unavail	perr
FOR[2]	5b	1	-	-	0	-	-
FOW[2]	5b	1	-	-	0	-	-
ACCVIO[2]	5b	1	-	-	0	-	-
BAD_VA[1,2]	5b	1	-	-	0	-	-
ALIGN_ERR	5b	1	1	-	-	-	-
DTB Miss Single	5b	1	-	0	1	-	-
DTB Miss Double	5b	1	-	1	1	-	-
MAF Full	5b	-	-	-	-	1	-
WB Full	5b	-	-	-	-	1	-
LD-MAF Conflict	5b	-	-	-	-	1	-
LD-ST Silo Conflict	5b	-	-	-	-	1	-
Dcache Parity Error	6a	-	-	-	-	-	1

[1]A BAD_VA trap will disable dtb_miss signal to the IBOX.

[2]FOR, FOW, ACCVIO, and BAD_VA are recognized at the IBOX when align_err and dtb_miss are deasserted and dmm_err is asserted. The FOR, FOW, and ACCVIO bits in the MM_STAT register will only be set if there was a DTB hit. The ACCVIO bit will also be set by BAD_VA.

- means signal is unaffected by this trap condition.

1.1.3.0.1 Memory Management Traps

The Dstream Memory Management traps include FOR, FOW, ACCVIO, BAD_VA, ALIGN_ERR, and DTB Miss traps. These are briefly described in Table 1–5. When any one of these traps is detected, the IBOX is notified by the **DMM_ERR** signal. A group of encoded signals is sent to the IBOX at the same time so the IBOX may generate the appropriate trap vector. These signals are specified in Table 1–6. There is an identical set of signals for each of the 2 issue pipes.

For the HW_LD and HW_ST instructions with the PHYS bit set, the DTB is bypassed and memory management traps are not generated (although MBOX_UNAVAIL and Dcache Parity Error traps may be).

When **DMM_ERR** has been asserted, the following MBOX IPRs are loaded and locked:

- MM_STAT - This register is loaded with the opcode and Ra field of the trapping instruction. It also stores the nature of the fault (FOR, FOW, ACCVIO, BAD_VA, DTB_MISS).
- VA - The complete Virtual Address (bits <63:0>) of the faulting memory reference is loaded into the VA register.
- VA_FORM - The VA_FORM is not actually a register that is loaded and locked. Instead, when a HW_MFPR VA_FORM is issued, it returns shifted and truncated bits of the Virtual Address from the VA register along with bits from the MVPTBR register.

These registers may also be loaded and locked when a Dcache parity error is detected as discussed in Section 1.1.3.0.3.

These registers are all unlocked when the VA register is read by PALcode. Their contents will remain unchanged until another Memory Management trap or Dcache parity error is detected.

1.1.3.0.2 Miss Address File Full and Conflict Traps

The Miss Address File generates an **MBOX_UNAVAIL** trap when either the write buffer or the Dstream read portion of the MAF is full, or when the LD-MAF Conflict or LD-ST Silo Conflict conditions are detected. There is one **MBOX_UNAVAIL** signal per pipe, and it signals to the IBOX that the instruction being trapped needs to be replayed. These types of traps do not trap to PALcode. The IBOX loads the PC of the trapping instruction and restarts issuing immediately.

The **MBOX_UNAVAIL** trap is a precise trap and must inhibit the MBOX from updating any state. The same considerations apply as are mentioned in Section 1.1.3.0.1, Memory Management Traps. No IPR registers are loaded or locked as a result of this trap.

The sources of the **MBOX_UNAVAIL** trap are talked about in more detail in Section 1.1.9.9, Mbox Unavailable Traps.

1.1.3.0.3 Dcache Parity Errors

Parity will be checked on the data read from the Dcache on LDs and on the tag read from the Dcache on both LDs and STs. Parity will not be checked on either data or tag if the data sub-block valid bits are not set. If either data sub-block valid bit is set, then tag parity will be checked. Dcache parity checking will be enabled unless the **DC_PERR_DIS** bit in the DC_MODE register is set.

When a Dcache Parity error is detected, **PERR** is asserted to the IBOX. The following registers are loaded and locked: MM_STAT, VA, VA_FORM and DC_PERR_STAT. DC_PERR_STAT contains information as to which of the Dcache banks produced the error, whether the faulting instruction wrote the register file, and whether a second error occurred after the register was locked. If any of the error bits is set, then the register is locked.

MM_STAT, VA, VA_FORM are all unlocked by reading the VA register. DC_PERR_STAT is unlocked and cleared by writing a "1" to the lock bit.

NOTE

If Machine Checks are disabled for some reason, a Dcache parity error will still load and lock these registers, even though the machine check will never be recognized.

The pipes are not aborted when a parity error is detected. Instead, the IBOX will detect the parity error as a Machine Check and will assert the IBOX trap signal(s) at a later time. This is to ensure that the instructions aborted match up to the exception PC reported to PALcode.

1.1.3.0.4 Traps from the IBOX

The IBOX will send 2 signals to the MBOX indicating which pipe(s) to abort on any trap originating from the IBOX (included also are all EBOX and FBOX imprecise traps, interrupts, CBOX fill errors, etc.) These signals are **KILL_E0** and **KILL_E1**. Each signal will abort the instruction in the current stage of the pipe specified by the signal name and all instructions already issued in both pipes back through Stage 3. When the IBOX aborts a particular instruction, the MBOX must ensure that any MBOX-generated traps for that instruction or following instructions will

not result in the locking of the VA and MM_STAT registers. The trap signals themselves, except the Dcache parity error, will be ignored by the IBOX while in the trap shadow.

NOTE

It is possible for **KILL_E0** to be asserted, but not **KILL_E1**. This means that the instruction in pipe1 is logically earlier than the instruction in pipe0. When this condition occurs, the instruction in pipe0 is aborted. The instruction in pipe1 is not aborted unless there is an MBOX generated trap in pipe1. All following stages must still be aborted.

If there is an IBOX trap in pipe0 and the instruction in pipe0 is logically ahead of the instruction in pipe1, then the IBOX is guaranteed to assert the **KILL_E1** signal.

The same actions are inhibited on IBOX traps as mentioned in Section 1.1.3.0.1, Memory Management Traps. No IPR registers are loaded or locked as a result of this trap.

1.1.3.0.5 CBOX fill errors

When the CBOX detects an ECC error on a Dcache fill, it will assert the signal **M_RFB_ECC_ERR**. The MBOX will load the register number and format control into a special ECC_FILL holding register. This register will be locked against further updates until the CBOX returns corrected data. (A second ECC error can not occur before the corrected data is returned). When the corrected data is returned on an ECC_FILL from the CBOX, the fill data is not written to the Dcache. The IBOX register number and the E/FBOX format control are read from the ECC_FILL register and the fill data is forwarded to the integer/floating register file.

The Dcache is flushed in the cycle following the assertion of **M_RFB_ECC_ERR**. The flush takes effect 2 cycles after the Dcache write of the fill in error. Due to restrictions placed on the timing of ST/FILL and LD/FILL operations, a load or store will never access the bad Dcache data in the cycle before the Dcache is flushed.

Because both sub-block data valid bits are set when the second half of a fill is written to the Dcache, care must be taken to maintain coherence in the Dcache. The MAF will set the NOFILL bit if the ECC error happens on a FIRST_FILL. This prevents the second half of the fill from being written to the Dcache after the first half is flushed.

NOTE

There is potential for the first half of a fill from the Scache to be written to the Dcache in the cycle following the fill with the ECC error (11B). The first half of the fill will be flushed from the Dcache in 12A. The second half of the fill will be written to the Dcache in 12B, and both sub-block data valid bits will be set. Since the Dcache flush operation clears valid bits but does not affect the Dcache data, Dcache coherence is maintained as long as the two parts of the fill operation happen in back-to-back cycles.

No aborting of the pipes will be done when the ECC error is first signaled to the MBOX. Instead, the IBOX will detect it as a Machine Check and will assert the IBOX trap signal(s) at a later time. This is to ensure that the instructions aborted match up to the exception PC reported to PALcode.

1.1.3.0.6 Multiple Traps

If multiple traps are detected in the same cycle, then multiple trap signals may be asserted to the IBOX in that same cycle. However, special actions must be taken in regards to aborting the pipes and loading and locking the VA and MM_STAT registers depending on which traps were asserted by which pipe.

Dcache parity error traps take precedence over any other traps that occur in the same cycle (either pipe). For all but the Dcache parity error trap, if traps occur on both pipes in the same cycle, the trap that is associated with the logically earlier of the two instructions takes precedence over the other. Within a given instruction, the precedence of traps is this:

1. Dcache Parity Error
2. IBOX
3. Memory Management
4. MBOX_UNAVAIL

Within a given cycle, the actions taken for aborting pipes and loading and locking registers are as follows:

- Dcache parity errors by themselves do not abort any pipes. If there is a Dcache parity error, then the VA, MM_STAT, VA_FORM and DC_PERR_STAT registers will be loaded and locked by the pipe generating the Dcache parity error even if the instruction in the other pipe is logically ahead of the instruction with the Dcache parity error.
- An IBOX trap will abort the instruction in the pipe that generated the trap. If one of the instructions IBOX traps and that instruction is logically the earlier of the two instructions, then the IBOX will abort both pipes by asserting **KILL_E0** and **KILL_E1**. IBOX traps do not load and lock any registers in the MBOX. An IBOX trap will abort any attempt to load and lock the VA, VA_FORM and MM_STAT registers by any MBOX generated traps that happen in the IBOX trap shadow.
- If there is not an IBOX trap in pipe0, then MBOX generated traps in pipe0 take precedence over any traps in pipe1. The instructions in both pipes are aborted and the MM_STAT, VA and VA_FORM are loaded and locked with information from pipe0. If two loads occur in the same cycle, the instruction in pipe0 is guaranteed to be logically ahead of the instruction in pipe1. If the instruction in pipe1 is a non-MBOX instruction that is logically ahead of the instruction in pipe0, then the abort of pipe1 has no effect.
- MBOX traps in pipe1 take precedence over IBOX traps in pipe0 unless **KILL_P1** is asserted. This is because the IBOX will abort the instruction in pipe1 if it is logically after the instruction in pipe0 and the instruction in pipe0 IBOX traps. If there is an IBOX trap in pipe0 and **KILL_P1** is not asserted, and there is an MBOX generated trap in pipe1, then both pipes are aborted and the MM_STAT, VA and VA_FORM registers are loaded and locked with information from pipe1.
- If there is a trap from any source in pipe1 and there are no traps in pipe0 then the instruction in pipe0 is allowed to complete and the instruction in pipe1 is aborted. The MM_STAT, VA and VA_FORM are loaded and locked with information from pipe1. Note the instruction in pipe0 may be logically ahead of the instruction in pipe1. If pipe1 has an IBOX trap, then **KILL_E0** will be asserted by the IBOX and pipe0 will be aborted. If the instruction in pipe1 MBOX traps and the instruction in pipe0 is a non-MBOX instruction, then allowing pipe0 to complete is innocuous since the instruction can not modify any MBOX state. If the instruction in pipe1 MBOX traps and the instruction in pipe0 is an MBOX instruction then the pipe0 instruction

is guaranteed to be logically ahead of the pipe1 instruction, and the pipe0 instruction should be allowed to complete.

- Memory management traps take precedence over MBOX_UNAVAIL traps. The VA, VA and VA_FORM are loaded and locked with information from the trapping pipe when the memory management trap takes precedence.
- MBOX_UNAVAIL traps do not load and lock any registers.

Summary of priority of loading and locking the VA/MM_STAT/VA_FORM registers:

1. Dcache parity error pipe0
2. Dcache parity error pipe1
3. IBOX trap pipe0, pipe0 is logically ahead of pipe1
4. IBOX trap pipe1, pipe1 logically ahead of pipe0
5. MBOX MM trap pipe0
6. MBOX unavailable trap pipe0
7. IBOX trap pipe1, pipe0 logically ahead of pipe1
8. MBOX MM trap pipe1
9. MBOX unavailable trap pipe1

Note that the VA,VA_FORM and MM_STAT registers are not loaded or locked when an IBOX or MBOX_UNAVAIL trap has precedence. Attempts to load and lock the registers by lower priority traps are aborted.

Table 1–7: Table of Multiple Trap Effects

PERR0	PERR1	I0	MM0	M_UNAVL0	I1	MM1	M_UNAVL1	Action
0	0	0	0	0	0	0	0	VA, MM_STAT unlocked; noabort
0	0	x	0	0	0	0	1	VA, MM_STAT unlocked; abort
0	0	x	0	0	0	1	x	VA, MM_STAT loaded and locked from pipe1; abort
0	0	x	0	0	1	x	x	VA, MM_STAT unlocked; abort
0	0	0	0	1	x	x	x	VA, MM_STAT unlocked; abort
0	0	0	1	x	x	x	x	VA, MM_STAT loaded and locked from pipe0; abort
0	0	1	x	x	0	0	0	VA, MM_STAT unlocked; abort
0	1	0	0	0	0	0	0	VA, MM_STAT, DC_PERR_STAT loaded and locked from pipe1; noabc
1	0	0	0	0	0	0	0	VA, MM_STAT, DC_PERR_STAT loaded and locked from pipe0; noabc

- Ix—ALL non-MBOX traps in PIPEx
- MMx—Dstream MM faults and DTB_Miss in PIPEx
- M_UNAVLx—MAF full and conflict traps in PIPEx
- PERRx—Dcache parity error in PIPEx
- abort—abort all instructions following an instruction that traps due to IBOX, MM, or MBOX_UNAVAIL traps

Table 1–7 (Cont.): Table of Multiple Trap Effects

PERR0	PERR1	I0	MM0	M_UNAVL0	I1	MM1	M_UNAVL1	Action
1	1	0	0	0	0	0	0	VA, MM_STAT, DC_PERR_STAT loaded and locked from pipe0; set SEO; noabort
0	1	x	x	x	x	x	x	VA, MM_STAT, DC_PERR_STAT loaded and locked from pipe1; abort if other trap
1	0	x	x	x	x	x	x	VA, MM_STAT, DC_PERR_STAT loaded and locked from pipe0; abort if other trap
1	1	x	x	x	x	x	x	VA, MM_STAT, DC_PERR_STAT loaded and locked from pipe0; set SEO; abort if other trap

- Ix—ALL non-MBOX traps in PIPEx
- MMx—Dstream MM faults and DTB_Miss in PIPEx
- M_UNAVLx—MAF full and conflict traps in PIPEx
- PERRx—Dcache parity error in PIPEx
- abort—abort all instructions following an instruction that traps due to IBOX, MM, or MBOX_UNAVAIL traps

Here are the simplified equations for loading and locking the VA/VA_FORM/MM_STAT registers and selecting **VA0/VA1**:

- LD_LK = (MM0 && !I0) | | (MM1 && !I1 && !M_UNAVL0) | | PERR0 | | PERR1
- SEL_VA0 = PERR0 | | (MM0 && !PERR1)

1.1.4 Processor Cycle Counter

The Processor Cycle Counter is a 32-bit counter which is written via the CC and CC_CTL IPR registers and read via the RPCC instruction. RPCC returns a 64 bit value. The lower 32 bits are the cycle counter and the upper 32 bits are an offset which may be written to by writing the CC IPR. The CC_CTL IPR is used to write the lower 32 bits and to enable or disable the counter. The counter is disabled on chip reset.

Implementation Note: The counter is partitioned into two parts - a 4-bit part and a 28-bit part. The carry input to the 4-bit part is tied to the counter's enable signal and this portion increments every cycle. The 28-bit incrementer is only updated whenever the carry out of the lower 4-bit incrementer is active. This gives the 28-bit incrementer 16 cycles to work.

The upper 32 bits form a simple register which is written via MTPR to CC and read via RPCC, but never incremented.

The SRM specifies that there must be a mode where 0 is always returned when the PCC is read. This is accomplished by writing all 0's to CC and all 0's to CC_CTL. This has the effect of initializing the counter to 0 and turning it off, so subsequent RPCCs will return a value of 0.

1.1.5 Big Endian Support

EV5 will provide minimal support for big endian systems (EV5 is "little endian"). When the big endian mode bit in the MCSR register is set, address bit <2> will be inverted for all longword Dstream accesses.

"Big endian" refers to the addressing of longwords within a quadword. EV5 is "little endian" and addresses the low order longword of a quadword with a lower address than the high order longword. In "big endian" systems, this is the opposite (the low order longword has a higher address than the high order longword).

For longword accesses, EV5 will perform a rotate and sign extend based on address bit 2. Therefore, when in big endian mode, an inversion of bit 2 of the address is necessary for all longword Dstream references. This inversion happens upon entering the MBOX - the formatting logic in the Dcache interface and the MAF PA control both need the big endian version of bit 2. The virtual address stored in the VA register will not reflect this inversion.

1.1.6 Interface requirements with FBOX, EBOX, IBOX for Dstream Instruction Execution

The Icache interface is discussed in Section 1.1.9.4, Icache Interface.

1.1.6.1 Instruction Opcode

For each EBOX pipe, the IBOX will send a subset of the 32-bit instruction on **E0_INST**<31:0> and **E1_INST**<31:0>. The MBOX will use these to decode the type of instruction, the destination register number for LDs, the IPR number for HW_MxPRs, and the instruction type for HW_LDs and HW_STs.

E0_INST<31:0> and **E1_INST**<31:0> will arrive in Stage 2 of the pipe and will be decoded, piped along, and aborted, as needed, by the MBOX. The IBOX will also send a stage 3 stall signal (**STALL**), which will be used to stall the stage 3 opcode latches for both pipes.

The IBOX also sends the signal **PAL_SHADOW_EN** which indicates that the destination register on integer loads is really a PAL shadow register. This information is stored in the MAF with the register number and returned to the IBOX on fills. This signal is a stage 3 signal which is already stalled appropriately by the IBOX.

The instruction in pipe0 is valid if the signal **E0_ISSUE** is asserted, and the instruction in pipe1 is valid if the signal **E1_ISSUE** is asserted. **EX_ISSUE** are asserted in stage 4. Since this is too late for the MBOX to setup the Dcache commands and address based on an "issued" instruction, the IBOX will also send a signal that will give the hint that the instruction in pipe0 is not "junk" (such as for an Icache or ITB miss). This signal, **E0_VALID**, is asserted in stage 2. The **E0_VALID** signal is stalled using the stage 3 **STALL** signal, and is cleared after **E0_ISSUE** is asserted while the pipeline is still stalled. The MBOX does this to reduce the chance of blocking fills on a "false" store (this is the case where the store has issued and its valid bit is stalled).

NOTE

If a store is stalled and has not issued yet, incoming fills to the Dcache will be blocked by the stalled store.

For integer fills (including STx_C_DONE), the CBOX sends the IBOX notification to allocate a bubble in the EBOX pipe by asserting **ALLOC_CYCLE** in stage 2. The IBOX uses this signal to disable the **EX_ISSUE** signals, which are too late for the MBOX to know that there is an idle bubble in the pipe. The MBOX will need to use **ALLOC_CYCLE** directly in order to setup the Dcache properly for a fill (in the case when a fill is coming in and it looks like there is a store in the EBOX pipe).

1.1.6.2 Restarting the IBOX After MB, LDx_L and STx_C Instructions

The IBOX will stop issuing MBOX instructions whenever it encounters an MB, STx_C, a HW_ST with the COND bit set, a LDx_L, or a HW_LD with the LOCK bit set. For the MB, STx_C and HW_ST_C cases, the CBOX will notify the MBOX upon completion of the command via **RETURN_STATUS**<3:0> and the MBOX will then restart the IBOX by asserting **MB_CLEAR**. For the LDx_L and HW_LD_L instructions, the MBOX will restart the IBOX after the CBOX accepts the load locked command.

1.1.6.3 Virtual Address from EBOX

For each EBOX pipe, the EBOX will send a clocked 64-bit virtual address from the output of the Stage 4 adder, **VA0**<63:0> and **VA1**<63:0>. There are both active high and active low versions of each of these busses. The addresses on **VA0** and **VA1** will be valid for all LDx (including LDx_L and HW_LD), ST (including STx_C and HW_ST), FETCHx and HW_MTPR TBIS instructions. The MBOX uses these signals primarily for address translation and the bad VA check. They are also piped and saved in the VA register on traps.

The EBOX has a special "fast" index adder which will send the index bits of the virtual address directly to the Dcache.

On HW_MTPR instructions the data on the **VA0** bus will be a copy of the data on the **ST_DATA** bus. (The Ra/Rb field of the HW_MTPR instruction must be the same). The **VA0**<63:0> is used on HW_MTPR instructions to load the Mbox/Dcache IPR registers.

1.1.6.4 LD bus

The MBOX drives two 64-bit busses into the EBOX, **LD_DATA0**<63:0> and **LD_DATA1**<63:0>, one for EBOX pipe0 and the other for pipe1. These busses are muxed together with other EBOX sources and written into the register file. They can also be bypassed directly into the current EBOX operation. The MBOX will return data on one or both of these busses for several operations (LDx, HW_LD, STx_C, HW_ST with the COND bit set, RPCC, HW_MFPR, and fills). During these operations, unused bits will be driven with "0" values, such as in the STx_C instruction which only returns one bit of data. The MBOX will also drive a "0" value on the **LD_DATA0** bus for any HW_MFPR from a write-only MBOX IPR or from an unassigned IPR in the range MBOX IPR addresses, 2xx(hex). This is done to support the EV5 verification effort.

The Dcache also returns data to the MBOX on two busses, **DATA0**<63:0> and **DATA1**<63:0>. During LD and HW_LD instructions, **DATA0**<63:0> and **DATA1**<63:0> will contain the data read from the Dcache arrays. For fill operations, these busses will carry the contiguous octaword of data supplied by the CBOX, with the lower quadword on **DATA0**<63:0> and the upper quadword on **DATA1**<63:0>. For both load and fill operations, the MBOX will drive properly formatted versions of the **DATA0**<63:0> and **DATA1**<63:0> busses onto the **LD_DATA0**<63:0> and **LD_DATA1**<63:0> busses, respectively.

There are 3 possible formats for integer LD and fill operations: quadword, longword from the upper longword of a quadword, and longword from the lower longword of a quadword. No formatting is necessary in the quadword case. For a longword load from the upper longword, the upper longword is shifted into the lower longword position and the MSB of the longword is sign-extended across the upper longword of the new quadword. For a longword load from the lower longword, no shifting is necessary, but the upper longword of the new quadword is replaced with a sign-extension of the MSB of the lower longword.

On floating point LDs and FILLs, the Dcache will send the data directly to the FBOX on **DATA0**<63:0> and **DATA1**<63:0>. The FBOX will perform floating point data formatting under MBOX control.

1.1.6.5 ST Bus Sources and Destinations

The EBOX sends the MBOX a 64-bit store bus, **ST_DATA**<63:0>, for integer STx, HW_ST, STx_C and HW_MTPR instructions. The store bus data is ignored by the MBOX on a HW_MTPR instruction; the IPR write data is taken from the **VA0** bus instead.

For store instructions, the MBOX will pass the data along to the Dcache for quadword integer STxs and will copy the lower longword of data onto the upper longword for longword integer operations. Longword parity is calculated on this "formatted" data. The MBOX drives the formatted data and the two longword parity bits to the Dcache on **WR_DATA**<63:0> and **WR_LW_PARITY**<1:0>, which are tristate busses that also carry floating point store data and parity from the FBOX to the Dcache. All floating point formatting and parity generation is done by the FBOX, but MBOX will control the tri-state **WR_DATA** bus drivers for both integer and floating point stores. For longword stores, the Dcache will select which longword to write into the Dcache based on bit 2 of the address. The Dcache will forward the store data and parity on to the CBOX write data buffer on separate busses.

1.1.6.6 Register Numbers and Controls to FBOX and IBOX for Dstream FILLs and LDs

For Dstream FILL data coming from the CBOX, the MBOX sends the destination register numbers and valid bits to the IBOX. The IBOX controls both the integer and floating point register file writes. There are 2 sets of register numbers and valid bits, one for each quadword of the octaword of data being returned from the CBOX. These bits are read from the MAF when a Dstream FILL request comes in from the CBOX. **FILL_RNUM0**<6:0> and **FILL_RNUM1**<6:0> contain the 5-bit register number, whether it is a floating point or integer register, and whether it is a PAL shadow register (integer only). These signals go to the IBOX and are qualified by the signals **FILL_VALID0** and **FILL_VALID1** to select which pipe(s) contain data to be written to the register file. The signal **FILL_COMING** is issued at the same time as the register numbers to indicate that there is a potential fill. This allows the IBOX to setup the FBOX register number muxes appropriately. **FILL_COMING** is needed on floating point fills, but not integer fills, because the CBOX will allocate bubbles in the EBOX pipe for integer fills, but not for floating point fills. Therefore, a load could be issued at the same time as a floating point fill, requiring the load to be force missed, and the fill register numbers to be selected for writing the floating point register file. Separate **FILL_COMING** signals for the 2 FBOX ports are not necessary since a FILL reserves both ports regardless of whether data is being written to both ports of the register file (a Dcache resource requirement).

For floating point LDs and FILLs, the MBOX sends formatting information to the FBOX to control the floating point formatters on the input to the FBOX register file. These signals are **LD_FORMAT0**<2:0> and **LD_FORMAT1**<2:0>, one for each pipe. These specify IEEE or VAX Floating Point format, whether it was a longword or quadword operation, and bit <2> of the address for selecting which longword within a quadword.

1.1.7 Dcache Hit and Load Miss Conditions

For LD and HW_LD commands, the IBOX needs to know whether the LD hit or missed the Dcache for the purposes of scheduling bypasses, freeing up "dirty" registers, and controlling register file writes. To determine if the LD hit or missed, the PFN read out of the DTB is compared with the tag read out of the Dcache. If there is a match, and the data is valid in the Dcache, then the hit signal is generated. This information is determined on a per pipe basis and is sent to the IBOX on **DC_HIT_E0** and **DC_HIT_E1**.

This is true for the "normal" case. However, there are cases when the LD may be "forced" to hit or miss regardless of the results of the tag match, such as in reads from I/O space, or for the LDx_L instruction. In addition, the same signal is used by the IBOX for other instructions to accomplish the same objective. For instance, the HW_MFPR and RPCC instructions "look" like a LD to the IBOX, so the MBOX must force the DC Hit signal active when returning data for these. The cases are listed in Table 1–8. This same signal is used by the MAF for loading Dcache LD misses, and by the Dcache on stores to enable the update of the data array.

Table 1–8: DC Hit Conditions, (prioritized)

Condition	Action
RPCC	HIT
HW_MFPR from any MBOX or DCACHE IPR (2xx,hex)	HIT
LDx_L or HW_LD with LOCK bit set	MISS
HW_ST, STx, STx_C, LDx or HW_LD, AND NOT MCSR<DC_ENA>	MISS
LD or HW_LD simultaneous with a DC FILL[1]	MISS
HW_ST, HW_LD with LOCK bit not set, STx, STx_C or LDx, AND MCSR<DC_FHIT> AND MCSR<DC_ENA>	HIT
HW_ST, STx, STx_C, LDx or HW_LD, AND PA<39> = 1	MISS
HW_ST, STx, STx_C, LDx or HW_LD, AND tag nomatch OR data not valid	MISS
HW_ST, HW_LD with LOCK bit not set,STx, STx_C or LDx, AND tag match AND data valid	HIT

[1]This will only occur on floating point fills.

The **ST_VALID** signal indicates to the Dcache the need to update its data array for a store. The Dcache command interface will be setup for a write, but the actual write will be qualified by **ST_VALID**. **ST_VALID** is asserted if the Dcache hit conditions in Table 1–8 are true and the store does not trap.

1.1.8 Dcache Interface

The Dcache is organized as 2 separate 8KB arrays, one for pipe0 and one for pipe1 (see <REFERENCE>(Dcache\full)). Each array has a separate tag store associated with it. The tag and data arrays for each pipe are accessed through separate word line decoders. This allows different accesses to be happening to the tag and data arrays in any given cycle (necessary for STs).

The MBOX is responsible for controlling the operation of the Dcache each cycle. Requests for access to the Dcache can come from either the LD/ST pipes or the CBOX (on Dstream fills). The Dcache interface is responsible for deciding which request is granted to the Dcache, driving the proper commands and status, and processing the results of the Dcache operation.

Every cycle the MBOX initiates a Dcache access by driving a TAG_CMD and DATA_CMD to both Dcaches. Only one set of commands are needed for both pipes since both Dcaches will always be performing the same basic operation (Fill, LD, ST). The table below shows the 2 commands sent and the operation done for all combinations of Dcache requests/cycle. The priority order for Dcache access is 1) STORES, 2) FILLS, 3) LOADS. Any ST in the silo during a given cycle will cause any FILL to bypass (not write) the Dcache. The IBOX guarantees that no LDs will be issued in the 2nd cycle after a ST. Also, a LD in the same and/or next cycle as an incoming FILL will be forced to miss. The LD is never sent to the Dcache in this case. This will only happen on floating point fills. On integer fills, there will never be another instruction requiring Dcache resources (LD or ST) since the CBOX will notify the IBOX to insert idle cycles into the EBOX pipes whenever an integer fill is coming. If the index of a LD matches exactly the index of a ST in the immediately preceding cycle, and the store hit in the Dcache, then the LD will be trapped.

The following table shows the Commands and Index select controls that will be driven to the Dcache. All command information will be sent to the Dcache in the B phase of the cycle before the actual DC operation takes place. It is the responsibility of the MBOX to properly prioritize and align the ST/FILL/LD operations for the DC.

Assumptions for Stores:

1. The tag lookup for Store operations takes place in 4B/5A of the store instruction.
2. DC hit will be calculated for the store operation in 5B of the store.
3. If the Store hits, then the data array will be updated in 6B/7A of the store.
4. The Dcache processes the STx_C as a normal ST instruction. The CBOX will always invalidate the DC at the index of a STx_C that fails, and the Ibox will not issue any MBOX instructions after the STx_C until the operation is complete.

Assumptions for Fills:

1. Fill requests are aligned to cycle 2B at the MBOX.
2. MAF is read in 3B of the re-aligned fill cycle.
3. The Fill data is on the RFB in re-aligned cycle 4B/5A.
4. Fill data is formatted by the MBOX in 5B for use in the EBOX at 6A.
5. The actual Dcache write of Fill data is in 5B/6A. Only FIRST_FILL and LAST_FILL data are written to the Dcache.
6. ECC_FILLs are not written to the Dcache, but are forwarded to the integer/floating register files through the Dcache NOP command.

Assumptions for Loads:

1. The Dcache tag and data arrays are both read in 4B/5A.
2. Dcache hit is calculated in 5B.
3. The load data is formatted and returned to the EBOX in 5B for use in 6A.

Table 1–9: Dcache Commands

Operation	tag_cmd	tag_idx_sel	data_cmd	dat_idx_sel	d%z_data src	wr_data src	Comment
NOP	NOP	-	NOP	-	RFB[1],[11]	-	Must select RFB to d%z_data
FILL[1],[12]	FILL	MBOX	FILL	MBOX	RFB[2],[12]	RFB[7]	Conditional Dcache write in 5B/6A[3]
LD	RD	EBOX	RD	EBOX	DC[4]	-	Dcache tag/data read in 4B /5A
ST4[9]	RD	EBOX	()[8]	()[8]	RFB[5]	-	Dcache tag read in 4B/5A
ST6[10]	()[8]	()[8]	WR	MBOX	RFB[5]	ST_DATA	Conditional Dcache write in 6B/7A[6]
IPR_RD	RD	MBOX	RD	MBOX	DC[4]	-	IPR read of Dcache tag array and data parity
IPR_WR	WR	MBOX	NOP	-	-	-	Conditional tag write for Dcache test, see DC_TEST_CTL register

[1]Fill data is returned to the Mbox in the cycle before the actual DC write operation. A DC data command of "NOP" will be used to steer the Fill data on the RFB to the MBOX in the proper cycle.

[2]During the write operation of a fill the d%z_data bus source must be the RFB in anticipation of a following fill (ie, back-to-back fills) needing to steer data to the Mbox.

[3]The DC write operation for fills is conditioned with RFB_DATA_VALID from the Cbox.

[4] The only time the d%z_data needs to be sourced from the Dcache is when the data_cmd is RD.

[5] During Store operations it is still necessary to return any coincident fill data to the Mbox.

[6] The DC write operation for stores is conditioned with ST_VALID.

[7] The RFB is piped by the DC for one cycle before being written into the Dcache.

[8] It is possible for several piped stores to be executing simultaneously in the DC. When this happens, the tag for a new store instruction can be read coincidentally with the data write for a previous store. Valid data commands during these cycles are NOP and WR, valid tag commands are NOP and RD.

[9]ST4 — Store in stage 4 of the pipe (1st store silo stage).

[10]ST6 — Store in stage 6 of the pipe (3rd store silo stage).

[11]To save power, the d%z_data bus is not updated unless there is valid fill request from the CBOX. The MBOX will send the dcache a separate signal to indicate whether to update the bus on a NOP or FILL command.

[12]Includes FIRST_FILL and LAST_FILL but not ECC_FILL.

Table 1–10: Dcache Command Encodings

Command	Encoding	MUX Select	Encoding
TAG_CMD:		**TAG_IDX_SEL:**	
NOP (RD)	00	EBOX	0
RD	01	MBOX	1
FILL	10		
WR	11		
DATA_CMD:		**DAT_IDX_SEL:**	
NOP (RD)	00	EBOX	0
RD	01	MBOX	1
FILL	10		
WR	11		

1.1.8.1 Dcache LDs

The MBOX initiates a LD operation by sending **TAG_CMD**<1:0> = RD and **DATA_CMD**<1:0> = RD. The EBOX sends the indices for the loads directly to the Dcache on **VA0**<12:3> and **VA1**<12:3> from a special "fast" adder. The MBOX notifies the Dcache to select the EBOX source as the address via the **TAG_IDX_SEL** and **DAT_IDX_SEL** signals. The tag, valid, and parity bits are returned for each pipe to the MBOX to be used for DC_HIT and parity error checking. These signals are: **TAG0**<38:13>, **TAG1**<38:13>, **VALID0**<1:0>, **VALID1**<1:0>, **TAG_PAR0**, and **TAG_PAR1**. A quadword of data for each LD is sent to the MBOX and FBOX on **DATA0**<63:0> and **DATA1**<63:0>. For integer LDs, the MBOX formats the data and forwards it to the EBOX on **LD_DATA0**<63:0> and **LD_DATA1**<63:0>. For floating point LDs, the FBOX handles all formatting. Longword parity for each quadword is sent by the Dcache to the MBOX on **DATA_PAR0**<1:0> and **DATA_PAR1**<1:0>. For all LDs, the MBOX will check for data and tag parity errors.

A load that arrives at the Dcache at the same time as a fill will be forced to miss.

1.1.8.2 Dcache STs

Dcache STs are a 3 cycle operation. The MBOX begins a ST sequence by sending **TAG_CMD**<1:0> = RD and **TAG_IDX_SEL** = EBOX. During the first cycle, the Dcache reads the tag, valid and parity bits and returns them to the MBOX. DC_HIT and tag parity error checking is done in the next cycle. For the 3rd cycle, the MBOX sends **DATA_CMD** = ST, **DAT_IDX_SEL** = MBOX (to select **DC_ADDR**<12:3> as the data index), and a siloed version of **VA0** on **DC_ADDR**<12:3> and **ST_ADR**<2>to the Dcache. If there was a DC_HIT on pipe0 and no traps were detected, then the MBOX will assert **ST_VALID**. Each bank writes the data simultaneously in the 3rd cycle if **ST_VALID** is asserted. The MBOX will also send **WR_TYPE** to indicate whether the operation is a quadword or longword write.

Data and data parity are sent to the Dcache on the **WR_DATA**<63:0> and **WR_LW_PARITY**<1:0> buses. The FBOX drives these busses on floating point STs while the MBOX drives them on integer STs. For longword stores, the lower longword of data will be duplicated on both the upper and lower longwords of the data bus. (The Dcache will use bit **ST_ADR**<2> and **WR_TYPE** to determine which longword(s) to write).

During the 3 cycles where there is a ST in the silo, all FILLs will be blocked from accessing the Dcache. The Dcache will bypass any FILLs that arrive during these 3 cycles by defaulting the Dcache read muxes to the **RFB**<127:0>. The IBOX guarantees that no LDs will be issued in the 2nd cycle following a ST. However, LDs may be issued in the cycle immediately following a ST. The address for this LD will be compared with the address of the preceding store. If there is an exact match between address bits <12:3>, they access the same longword, and the store hit in the Dcache, then the LD will be trapped. If there is not a match, then the LD will read the Dcache as normal.

For STx_C instructions, the data will be written to the Dcache as for a normal ST instruction. If the STx_C fails, the CBOX will invalidate the Dcache block via the INVAL port to the Dcache. Since the IBOX will not issue any more MBOX instructions until after the STx_C is complete, there is no danger of a subsequent LD hitting on the updated data of a failing STx_C.

The MBOX will invert the data parity bits that are sent to the Dcache on a store when the **FORCE_BAD_PAR** is set. The MBOX will send the mode bit signal, **FORCE_BAD_PAR**, to the Dcache, so that the Dcache can restore data parity at the input to the CBOX Write Data Buffer. This mode bit may be found in the DC_MODE register and is for test/diagnostic purposes.

1.1.8.3 Dcache FILLs

The MBOX receives Dcache fill requests from the CBOX on **RETURN_STATUS**<2:0> for each octaword of fill data on the **RFB**<127:0>. The tag and index are supplied to the Dcache by the MAF on the **DC_ADDR**<38:3> bus. The data and data parity are supplied to the Dcache by the CBOX on **RFB**<127:0> and **FILL_PAR**<3:0>, respectively. When a FIRST_FILL or LAST_FILL request is received, the Dcache interface drives **DATA_CMD** = Fill, **TAG_CMD** = Fill, **TAG_IDX_SEL** = MBOX and **DAT_IDX_SEL** = MBOX along with the tag parity (**TAG_PAR**) and the valid bits (**VALID**<1:0>) to be written to the array. For the first fill operation to a Dcache block, the MBOX will assert only one of the valid bits, the other valid bit will be deasserted. Which valid bit to assert is based on **OW_VALID** received from the CBOX. **OW_VALID** is effectively address bit <4> for the fill data. During the last fill (second) operation to a Dcache block, the MBOX will assert both of the valid bits. Both valid bits are always written at the same time. **OW_VALID** is muxed onto **DC_ADDR**<4> to select the appropriate octaword in the Dcache data array to write. If, for some reason, the first octaword did not get written to the Dcache, or was written to the Dcache with an ECC error, then the second octaword will be set to NOFILL to ensure the valid bits are correct.

The data will only be written if

1. **DATA_CMD**<1:0> = FILL,
2. **NOFILL** is deasserted and
3. **RFB_DATA_VALID** (driven by the CBOX) is asserted.

Corrected ECC data fills (ECC_FILLs) are not written to the Dcache.

There are separate **NOFILL** signals for each pipe for testability reasons. In normal operations, they have identical values and are set and cleared in the MAF based on certain error and conflict conditions (see Table 1–16, Dread Control Bits).

On Fills, all addresses and commands to the Dcache are identical for the 2 different Dcache banks. The octaword of fill data, the tag, tag parity, and valid bits are all written in the Dcache simultaneously, regardless of whether it is the first fill or second fill. Fill data for a first, last or ECC fill is forwarded to the MBOX/FBOX on the **DATA0**<63:0> and **DATA1**<63:0> busses. Fill data is always returned to the MBOX/FBOX, even if a ST is accessing the Dcache during that cycle (Fill bypass). Fills will take precedence over any LDs that arrive at the Dcache at the same time. The LD will be blocked and forced to miss.

NOTE

It is assumed that the CBOX will guarantee that no fills containing "old" data will be sent to the MBOX/Dcache after an invalidate for the same address has been sent to the Dcache.

1.1.8.4 Dcache Invalidates

The Dcache may be invalidated (valid bits cleared for a block or group of blocks) in one of the following ways:

1. The CBOX directly sends an invalidate command and index to the Dcache. The Dcache will clear all the valid bits for 2 Dcache blocks (1 Scache block).
2. PALcode can issue a HW_MTPR to the DC_FLUSH IPR. The Mbox will assert **DC_FLUSH** and the Dcache will clear all the valid bits in the array. It is assumed that PALcode will flush the Dcache in this way after reset.
3. PALcode may use the DC_TEST_TAG and DC_TEST_CTL IPRs to write to the tag and valid bit fields in the Dcache arrays.
4. When the CBOX signals an ECC error, the MBOX will assert **DC_FLUSH** and the Dcache will clear all the valid bits in the array.

The MBOX will not assert **DC_FLUSH** to the Dcache under reset.

1.1.8.5 Parity Generation and Checking

The MBOX will do even longword parity generation on the integer store bus on its way to the Dcache and write buffer. The FBOX will generate even longword parity on the floating point store bus. The data parity bits generated are a tristate bus driven by the FBOX on floating point stores and by the MBOX on integer stores. The MBOX generates even parity on the tag address on its way to the Dcache on fills. Data parity is generated by the Cbox on fills.

On all loads from the Dcache (both integer and floating point), the MBOX will do even longword parity checking on the data and even parity checking on the tag address bits, and generate appropriate parity errors if the data valid bits are set. The data valid bits themselves are not covered by parity. On stores, only tag parity is checked if the data valid bits are set.

Parity checking will be disabled during IPR accesses of the Dcache or if the Dcache data valid bits are not set. The **DC_PERR_DIS** bit in the DC_MODE IPR can be used to disable Dcache parity errors when the Scache is being tested, or when the Dcache is disabled or is in force hit mode.

If a parity error is detected, the information for the instruction causing the error is saved in the IPR registers: VA, MM_STAT, and DC_PERR_STAT. The instruction is not aborted (note: store data will still be written to the Dcache, load data to the register file). The IBOX will recognize it as a machine check and abort the instruction stream at a later time. This is discussed in more detail in Section 1.1.3.0.3, Dcache Parity Errors.

1.1.8.6 Operation Modes for the Dcaches

1.1.8.6.1 Dcache Force Bad Parity and Disable Parity

The DC_MODE register contains control bits for forcing bad parity to be written to the Dcache and for disabling Dcache parity errors. These modes are used for testability and diagnostic purposes.

When the **DC_BAD_PARITY** bit in the DC_MODE register is set, the Mbox will invert the data parity bits that are written to the Dcache on a store. The MBOX will assert **FORCE_BAD_PAR** to the Dcache, causing the data parity bits written to the CBOX write data buffer to be restored to the value calculated by the data parity generators. The value of the data parity bits sent to the Dcache on fills will be unaffected by the setting of the **DC_BAD_PARITY** bit.

The **DC_PERR_DIS** signal specifies whether Dcache parity error checking is enabled. When this bit is set the Dcache interface will disable Dcache parity error reporting.

1.1.8.6.2 Dcache Enable and Force Hit Modes

The DC_MODE register contains control bits for enabling the Dcaches, for disabling Dcache parity errors, and for putting the Dcache into a FORCE HIT mode for testability and diagnostic purposes. These modes are implemented via the normal MBOX/Dcache command interface, and so are transparent to the Dcache itself.

The **DC_ENA** bit in the DC_MODE IPR is cleared on reset and, when clear, forces all LDs to miss the Dcache and all STs and FILLs to not write the Dcache.

The **DC_FHIT** bit in the DC_MODE register specifies Dcache Force Hit mode. When **DC_FHIT** is set, then all STs will write to the Dcache regardless of the outcome of **DC_HIT**. A LD issued in pipe1 will have the effect of forcing **DC_HIT1** active and data will be read and used from the pipe1 Dcache, regardless of the actual outcome of the **DC_HIT** calculation. This same scenario works with the pipe0 Dcache. The **DC_ENA** bit takes precedence over **DC_FHIT**.

The tag, tag parity, and valid bits will remain unaffected by stores during **DC_FHIT** mode. (This is true for non-**DC_FHIT** mode as well).

NOTES

The **DC_HIT** and **DC_BAD_PARITY** bits are used exclusively for diagnostics and test.

Outstanding Dstream fills may return and write the Dcache, even in force hit mode. Also, floating point fills arriving at the same time as a LD will write to the Dcache, forcing the LD to miss, even though in force hit mode. A memory barrier inserted before code that tests the Dcaches will ensure all outstanding fills have been completed before the start of the test code. Disabling the Dcache will force loads to miss, causing fill requests to be generated to the CBOX.

A LD in the cycle following a ST to the same index while in force hit mode will result in a replay trap.

A LD(ST) from(to) a "non-cacheable" memory address (PA<39>=1) while the DC_MODE<DC_FHIT> bit is set, will result in the data being read from(written to) the Dcache.

1.1.8.6.3 Dcache Flush

A write to the DC_FLUSH IPR will cause the MBOX to assert **DC_FLUSH** to the Dcache. The Dcache will flush (clear) all the valid bits in both Dcaches regardless of the settings of **DC_ENA** and **DC_FHIT**.

1.1.8.7 Reading/writing Dcache Tags for Testability

There are three IPRs for the support of reading and writing the Dcache tags for testability. These are DC_TEST_CTL, DC_TEST_TAG, and DC_TEST_TAG_TEMP.

The DC_TEST_CTL register contains the row and bank in the Dcache that will be affected by reading/writing the DC_TEST_TAG register.

A write to the DC_TEST_TAG register will cause the MBOX to take the data from **VA0**<63:0> from the EBOX, and drive it onto **DC_ADDR**<38:13>, **TAG_PAR**, and **VALID**<1:0> to the Dcache. At the same time, the contents of the DC_TEST_CTL register will be driven onto **DC_ADDR**<12:5>, **NOFILL0**, and **NOFILL1**. When the HW_MTPR DC_TEST_TAG instruction is decoded, the **TAG_CMD** will be set to WR, **TAG_IDX_SEL** will be set to select the MBOX address, and the **DATA_CMD** will be set to NOP. This will enable the Dcache to write only the tag array. (See Table 1–9).

When a HW_MFPR DC_TEST_TAG instruction is decoded, the contents of the DC_TEST_CTL register will be driven onto **DC_ADDR**<12:3>, **TAG_CMD** and **DATA_CMD** are set to RD, and **TAG_IDX_SEL** and **DAT_IDX_SEL** will be set to select the MBOX address. The Dcache will read out the tag, tag parity, valid and longword data parity bits onto **TAG0**<38:13>, **TAG1**<38:13>, **TAG_PAR0**, **TAG_PAR1**, **VALID0**<1:0>, **VALID1**<1:0>, **DATA_PAR0**<1:0>, and **DATA_PAR1**<1:0>, respectively. Note that 4 HW_MFPR's are required to read all of the data parity bits associated with a block of dcache data. The MBOX will latch the tag, tag parity, valid and data parity bits from one of the banks into a temporary register (according to the bank select bits in the DC_TEST_CTL register). A subsequent HW_MFPR from DC_TEST_TAG_TEMP will enable this temporary register to drive onto the **LD_DATA0** bus to the EBOX.

NOTE

The Dcache data parity bits may only be written to the Dcache via a store or a fill. There is no IPR access for writing, only reading.

1.1.9 Miss Address File

1.1.9.1 Overview

The Miss Address File (MAF) has 3 basic functions:

1. Store all address and instruction data for memory references which will eventually be issued to the Scache. These include LD misses, all STs, LDx_L, STx_C, FETCHx, MB, WMB, and IREF requests.

2. Issue addresses to the Scache and BIU. This includes logic to guarantee proper ordering of LD's and ST's and prioritizing of Scache requests originating in the MBOX, including MB sequencing and IREF requests.
3. Supply addresses and instruction data to the Dcache, IBOX and formatters for incoming fills from the BIU/Scache.

The MAF is divided into 3 sections:

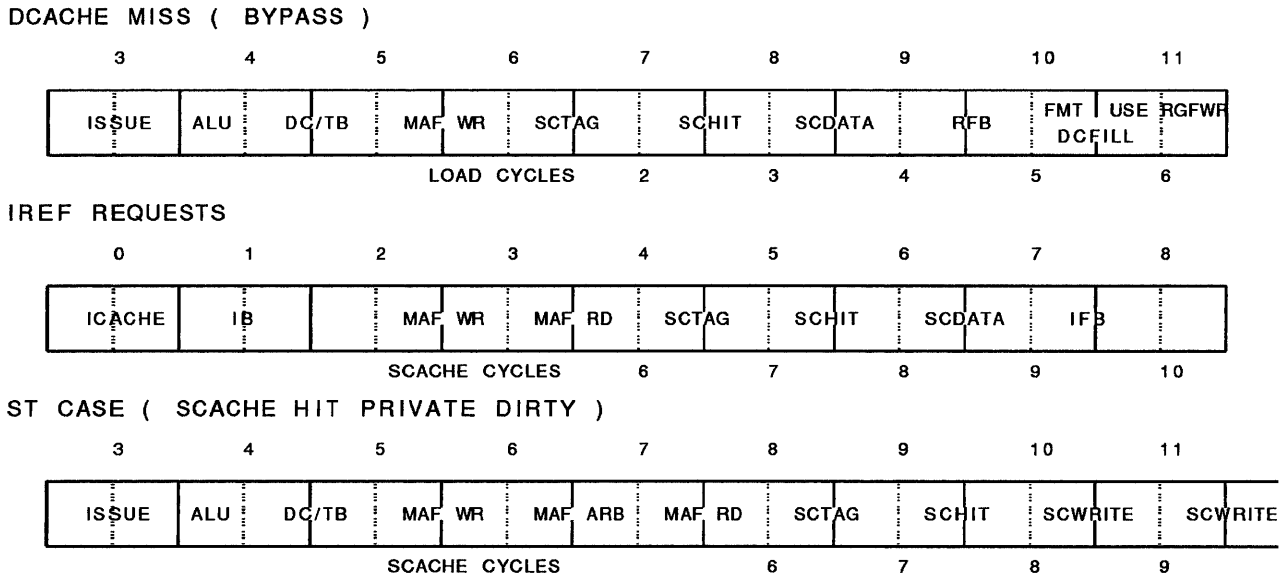
1. DREAD - handles all HW_LD and LDx misses, and all LDx_L's.
2. WB - handles all HW_ST, STx, STx_C, WMB, and FETCHx.
3. IREF - handles all IBOX prefetch queue requests.

1.1.9.2 Basic Timing

The MAF is the convergence zone between the end of the normal EV5 LD/ST pipes through DC_HIT, the Icache prefetch queue requests, the beginning of the address sequencing for the Scache, and the return of FILL data to the Dcache. Figure 1–6 shows how each of these timings lines up to the MAF timing. Throughout the text, the following timing assumptions will be used.

1. Incoming LD/ST - referenced to the EBOX pipes (Register file write cycle 6).
2. Incoming IREF Requests - referenced to Icache cycle 0.
3. Addresses to the Scache - referenced to Scache Tag cycle 6.
4. Return Status (Fills) - referenced to Register file write cycle 6.

Figure 1–6: MAF Timing Definition



1.1.9.3 CBOX Interface

The MAF in general has three communication links with the CBOX. (1) It supplies addresses for memory references to the Scache. (2) It interfaces with the data portion of the WB for any instructions stored in the WB and (3) the CBOX informs it when data is returning or operations initiated by the MAF are completing.

1.1.9.3.1 Command/Address Issue Interface

For issuing memory references to the Scache/CBOX, the MAF sends a command (**MAF_CMD**<3:0>) informing the CBOX what operation is being done, the address (**MAF_ADDR**<39:4>), the MAF entry of this address (**MAF_INDEX**<4:0>) which is used as an ID for the operation, and the type of reference being issued to the CBOX (**MAF_TYPE**: integer=0/floating=1).

When the MAF receives a Dread down one of the pipes, there is a 2 cycle window when it may be bypassed to the CBOX before the results of the merge or the Dcache hit logic are known. Arbitration proceeds during this time as if the Dread will ultimately want to be issued to the CBOX. If the Dread wins the arb in the first cycle, but merges with an existing entry or hits in the Dcache, then the command to the CBOX is aborted via the **MAF_ABORT** signal. If the Dread wins the arb only in the second cycle, but merges with an existing entry or hits in the Dcache, it will be effectively aborted by issuing a NOP command to the CBOX. Once this 2 cycle window has passed, if the Dread was not bypassed, didn't merge and missed the Dcache, it is loaded into the pending queue and begins normal arbitration. Once in the pending queue, Dreads are not aborted.

For a pair of loads issued to the MBOX by the IBOX in a given cycle, pipe0 has the higher priority for bypassing to the CBOX. Pipe1 will automatically begin arbing the next cycle assuming that the pipe0 bypass will be aborted. If the load in pipe0 is aborted due to a hit or merging, then the load in pipe1 proceeds to issue in the next cycle. This too is before the merge result is known, so it may be aborted by issuing a NOP command on **MAF_CMD**<3:0>. Meanwhile, 2 new loads may have come down the pipe. The new load in pipe0 will not have immediately been bypassed to the CBOX because the pipe1 load from the previous cycle speculatively won the arb. However, the new pipe0 load will be setup to issue in the next cycle, again assuming the previous cycle's pipe1 load will be aborted. As before, this is before the results of the merge are known, so it, too, may be NOP'd.

The CBOX indicates backpressure by two mechanisms. 1) If the Scache is busy the CBOX requests the MAF to hold off sending addresses by asserting a busy signal (**SC_BUSY**) 2 cycles before the CBOX needs the Scache. This signal must be asserted for each cycle that the Scache is busy. Any 2 consecutive cycles where **SC_BUSY** is not asserted will potentially be used by the MAF. 2) The CBOX indicates that the BIU cannot accept an address just issued by the MAF by asserting **RETRY** 2 cycles after the address is sent to the Scache. The MAF will place the rejected command in a replay queue. The command and the Scache address may be issued again to the CBOX 2 cycles later if the CBOX is not busy and there is nothing already pending in the replay queue. (Four cycles is the quickest turnaround time between command issue and reissue). The replay queue will always arb if any entry is valid in the queue.

For IO space reads (**PA**<39> = 1), if the command is not retried, then the CBOX will give the MBOX an indication that a fill is coming along with the index of the command. This fill command will always come at the same time as if the read had hit in the Scache. The fill will be marked invalid by the CBOX (using signal **RFB_DATA_VALID**), but the MAF will use the fill notification to read the quadword request bits out of that IO space read entry in the MAF, stop further merging to that entry, and send the quadword request information to the CBOX on **DRD_MASK**<3:0>. This takes place in cycle 8. **DRD_MASK**<3:0> will be sent whenever a fill request is generated by the CBOX, but it is only meaningful on IO space reads.

Table 1–11: Commands From MBOX MAF to CBOX Arbiter

Commands	MAF_CMD<3:0>	Description
NOP	0000	No operation
	0001	Reserved
	0010	Reserved
	0011	Reserved
DREAD	0100	DREAD Request
LDx_L	0101	Load Memory Data into Integer Register Locked
IREAD	0110	IREF Request
	0111	Reserved
FETCH	1000	Prefetch Data
FETCH_M	1001	Prefetch Data, Modify Intent
MB	1010	Memory Barrier
	1011	Reserved

Table 1–11 (Cont.): Commands From MBOX MAF to CBOX Arbiter

Commands	MAF_CMD<3:0>	Description
WR32	1100	32B Write Request
STx_C	1101	Store Conditional
	1110	Reserved
	1111	Reserved

1.1.9.3.2 Write Buffer Interface

When a ST is presented to the MAF, the MAF informs the write data buffer in the CBOX whether the ST was LW (**WR_TYPE**=1) or QW (**WR_TYPE**=0), the LW address of the ST (**WR_LW_ADDR**<4:2>), and which entry to write the data to in the WB data store (**WR_ENABLE**<5:0>). The write data and longword parity (**WR_DATA**<63:0> and **WR_LW_PARITY**<1:0>) are sent to the Dcache where they are latched and forwarded on to the CBOX write data buffer on **WB_DATA**<63:0> and **WB_LW_PARITY**<1:0>.

The data portion of the WB can ask the MAF to reissue write addresses by asserting **WR_NOW** and sending the MAF index of the entry on **WR_MAF_INDEX**<4:0>. If the CBOX is doing a 64B write operation, it will ask the MAF to reissue the same address twice, once with **WR_64B_REQ** deasserted, and once with **WR_64B_REQ** asserted. When **WR_64B_REQ** is asserted, the MBOX will invert **MAF_ADDR**<5> when reissuing the address. This second transaction is used to read the other 32B block from the Scache to form a complete 64B block to be written to the system.

1.1.9.3.3 Return Status

Upon completion of a requested operation the CBOX sends the MAF the index (**RETURN_INDEX**<4:0>) and the status (**RETURN_STATUS**<2:0>) which describes the operation being completed. This allows the MAF to clear entries from the file and forward necessary controls to the Dcache and register files for DREAD fills. Fills also require the **OW_VALID** (1=upper , 0=lower) to indicate whether the lower or upper octaword of the block is being filled. On all fills, the CBOX indicates that there is valid data on **RFB**<127:0> with **RFB_DATA_VALID**. The Mbox uses **RFB_DATA_VALID** to verify that a Fill is actually valid (Fills from the Scache and/or the Bcache may be speculative) and complete the fill of the Dcache and register files. (The CBOX also asserts **RFB_DATA_VALID** on a STx_C_DONE so that the MBOX will assert the **FILL_VALID** signal to the IBOX along with the STx_C register number). **SCACHE_HIT** is sent to the MBOX and used to disable merging when a load hits in the Scache. If the load did not hit, then merging is allowed to continue until the first fill from the Bcache is signalled (this is speculative).

If an ECC error is detected on fill data destined for the Dcache, the CBOX will assert the signal **M_RFB_ECC_ERR**. This signal is too late to hold off the register file write or the Dcache fill operations. On the second half of a fill the MAF DREAD entry will be cleared before the ECC error is known. The **ECC_ERR** signal is used by the MAF to conditionally lock the ECC error register that holds the register number and format control associated with the most recently returned fill data. (If there is no ecc error, the holding register will be overwritten on the next fill request from the CBOX). The MBOX will flush the dcache and the MAF will set the NOFILL bit for that DREAD entry of the error occured on the first half of a fill. When an ECC error is correctable in hardware, the CBOX will return the corrected fill data a minimum of three cycles later along with an ECC_

FILL return status. The ECC error register supplies the IBOX register number and the formatter control on an ECC_FILL. A read from the ECC error register unlocks the register for updates.

NOTES

The MAF needs to see a LAST_FILL in order to retire an entry from the DREAD file. The CBOX guarantees that a fill operation will complete even when a hard error (Scache parity error or uncorrectable ecc error) occurs. This means the MBOX will see the LAST_FILL request for all DREAD entries, even though the fill data may be garbage. This ensures the MAF is left in a predictable state.

The CBOX guarantees that the MBOX will not see multiple ECC errors. When the ECC holding register is loaded and locked, the MBOX will not see another ecc error until after the corrected fill data has been returned on an ECC_FILL.

LAST_FILLs from the CBOX are always non-speculative and are used by the MBOX to retire the MAF entry. If the CBOX generates a LAST_FILL request before an ECC error is detected on a FIRST_FILL, the CBOX will send a correction signal, **BOGUS_LF**, to indicate the LAST_FILL request was not valid. The MBOX will use this signal to abort any operations that may affect state as a result of the erroneous LAST_FILL request.

Table 1–12: CBOX Return Status

Status	RETURN_STATUS<2:0>	Description
NOP	000	No Operation
FIRST_FILL	001	This is the 1st Dcache fill cycle of 2
LAST_FILL	010	This is the last Dcache fill cycle of 2
WR_DONE	011	Write Operation Done
FETCH_DONE	100	FETCH_x Operation Done
MB_DONE	101	Memory Barrier Operation Done
ECC_FILL	110	This is corrected ECC fill data
STXC_DONE	111	The Result of the STx_C Operation is Returning

1.1.9.3.4 Invalidates - CBOX Guarantee

The MBOX is unaware of invalidates to the Dcache. The CBOX sends these to the Dcache directly. The MBOX and Dcache depend on the CBOX to guarantee that there will never be a 1ST_FILL-INVAL-2nd_FILL sequence on the same address, nor any INVAL-FILL sequence on the same address where the FILL represents "older" data than any updates from the system causing the INVAL, or where the FILL would cause the Dcache to load valid data for an address that has been removed from the Scache due to the INVAL.

1.1.9.4 Icache Interface

The MAF interfaces with the IBOX prefetch queue to either demand fetch 32B blocks of instructions on Icache misses or prefetch blocks of instructions during prefetch sequences. The Icache requests an IREF by sending the physical address (**IREF_ADDR**<39:4>) and prefetch queue index (**IREF_IDX**<1:0>) to the MAF. The MAF will load the address at the location specified by the prefetch index in 3A. When the MAF receives the **IREF_REQ** signal from the Icache, the entry is validated and normal arbitration for the Scache begins. Once the MAF receives a valid request, the operation will complete (ie. no aborting is done).

When the IREF is issued to the CBOX, the entry is retired by the MAF. The default (precharged) state for the QW request bits is sent to the CBOX on Istream IO space reads as **DRD_MASK**<3:0> = (F)HEX. The CBOX controls the filling of the Icache directly.

1.1.9.5 Loading the MAF

The MAF has 3 separate sources of addresses to be stored for issue to the Scache. All DREAD addresses are received on both pipes 0 and 1, at most 2 per cycle. All WB addresses are received on pipe 0 only, at most 1 per cycle. Dreads and writes may not occur in the same cycle. All IREF addresses are received on a dedicated IREF address bus, at most 1 per cycle. All DREAD and WB requests are loaded into the MAF during 6A. All IREF requests are loaded during 3A.

The DREAD and WB sections can either merge the incoming address with previous requests or allocate a new entry for the address. These sections each have FREE LIST fifos and PENDING queues. The FREE list points to the next MAF entry to be allocated for each type of reference. When an address is loaded, the MAF index is shifted off the FREE list and onto the PENDING queue for issuing. When the memory reference is complete and the entry deallocated, the MAF index is loaded back onto the respective FREE LIST to be reallocated to another address. IREF references receive the index directly from the prefetch queue, so no FREE LIST is needed in the IREF section.

Each entry in the free list will be initialized on reset to a unique index and all pending queue entries will be invalidated.

The following shows the bit fields of the pending queues for each section of the MAF.

Figure 1–7: Pending Queue Bit Fields

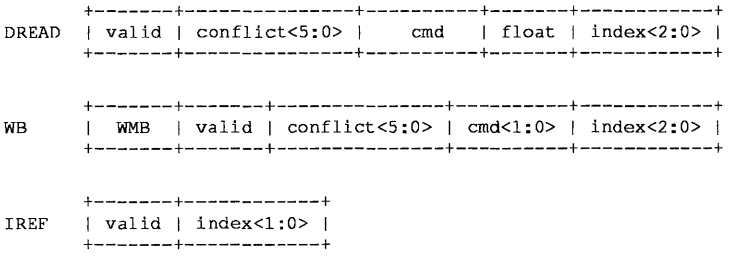


Table 1–13: Pending Queue Bit Fields

Field Name	Description
VALID	Pending queue entry is valid
INDEX<4:0>	Entry in the MAF corresponding to pending entry
CMD<3:0>	Command to be issued to CBOX
FLOAT	1 = Floating Point DREAD, 0 = Integer DREAD
WMB	Marks insertion of a WMB or a ST/ST conflict. All previous stores finish before subsequent stores can be issued
DREAD Conflict<5:0>	The WB entries that this DREAD entry has a conflict with. All conflicts must be cleared before an entry can be issued. clear<5:0> = WB_DONE<5:0>, where WB_DONE = WR_DONE | | FETCH_DONE | | STx_C_DONE set<5:0> = DREAD_ALLOCATE && WB_MATCH<5:0>
WB Conflict<5:0>	The DREAD entries that this WB entry has a conflict with. All conflicts must be cleared before an entry can be issued. clear<5:0> = LAST_FILL<5:0> && !BOGUS_LF set<5:0> = WB_ALLOCATE && DREAD_MATCH<5:0>

1.1.9.5.1 Dcache Read Misses

The DREAD section holds addresses for LD misses and LDx_L's. It consists of 6 entries of 32B block physical addresses. Each entry also has 4 slots which represent 1 of 4 quadwords within the 32B block. When a LD miss requests a certain quadword, the block address is written into the entry and the requested quadword slot is loaded with the register number and format information for this request. When the LD data is returned, the physical address is read out of the MAF and driven to the Dcache interface for FILLS. The register number and format information will be returned to the IBOX and floating point/integer formatters. Dcache write enables are derived at the Dcache interface from the NOFILL status bit and the **OW_VALID** control.

If a subsequent LD miss requests the same 32B block but a different quadword within the block, the address and pipe data can be merged into the same DREAD entry with register number and format information loaded into its respective quadword slot. Otherwise, it is allocated a new entry.

All DREAD merging is done on physical addresses. No merging is allowed between floating and integer requests (this is because integer fills allocate bubbles in the pipe and floating point fills do not). Also, quadword load requests are not merged with longword load requests, and longword requests to even address are not merged with longword requests to odd addresses (this was done as a simplification of the merge logic implementation). LDx_L requests are always allocated a new entry in the MAF (to ensure the LDx_L request is issued to the CBOX with the LDx_L command), and merging to the LDx_L entry is not allowed (this prevents subsequent LDx requests from matching multiple entries in the DREAD file). If merging needs to be disabled for a given entry in general, the NOMERGE bit is set for that entry. (See Table 1–16.)

If an incoming load address matches that of an address already in the MAF (down to the longword level), then the incoming load is forced to trap. This is the LD-MAF Conflict Trap and occurs whether or not the load would have hit in the Dcache. This is to ensure that loads to the exact same address do not finish out of order (Litmus test #1 in the ALPHA SRM).

The MAF also detects ST/LD conflicts by comparing incoming LD PA's (Physical Address) with the PA of previous writes in the WB. If a match is detected then a new entry is made in the MAF for that load and the new entry in the pending queue is made with the conflict bits corresponding to the conflicting WB indices set. No entry in the DREAD pending queue may be issued until all the conflict bits are cleared. All WB entries up to and including the conflicting write will be flushed to the CBOX interface at an elevated priority. When a conflicting write is retired, all conflict bits in the DREAD pending queue corresponding to that WB index will be cleared.

Figure 1–8: Dread Address Datapath

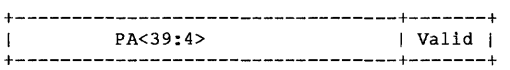


Table 1–14: Dread Physical Address Datapath bits

Field Name	Description
PA<39:4>	Physical Address of 32B Block Loaded on DREAD_ALLOCATE.
Valid	This address is a valid entry. set = DREAD_ALLOCATE clear = (LAST_FILL && !BOGUS_LF) | | RESET

Figure 1–9: Dread Register Formatting Bits

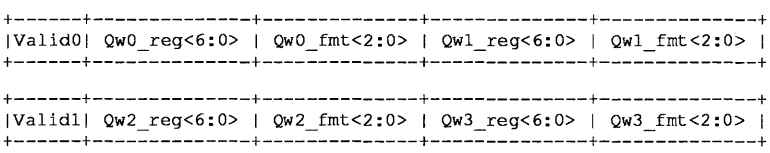


Table 1–15: Dread Register Formatting Bits

Field Name	Description
Qwx_reg<6:0>	QWx <6>-pal shadow, <5>-(1=floating,0=integer),<4:0>-register number
Qwx_fmt<2:0>	QWx format: <2> - (1=vax_fp,0=ieee), <1> - (1=LW,0=QW), <0> - (1=upper,0=lower LW)

Table 1–15 (Cont.): Dread Register Formatting Bits

Field Name	Description
Validx	This is a valid entry. set = (DREAD_ALLOCATE | | DREAD_MERGE) && (decode<4:3>) clear = (LAST_FILL && !BOGUS_LF) | | RESET NOTE - The register number and format control for the last entry cleared are locked in a special register when an ECC error is detected on a LAST_FILL. The ECC register supplies the register number and format control when the corrected data is returned from the CBOX.

Figure 1–10: Dread Control Bits

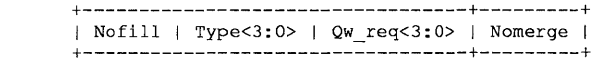


Table 1–16: Dread Control Bits

Field Name	Description
Nofill	Don't fill this entry into the Dcache. clear = (LAST_FILL && !BOGUS_LF) | | RESET set = (STx | | STx_C | | FETCHx) && DREAD_MATCH && !TRAP | | FIRST_FILL && FILL_BLOCKED (from DC interface) && RFB_DATA_VALID | | DREAD_ALLOCATE && PA<39>=1 | | FIRST_FILL && ECC_ERR
Type<3:0>	<3> = Quadword <2> = Longword Even <1> = Longword Odd <0> = Floating Point Loaded on DREAD_ALLOCATE.
Qw_req<3:0>	Quadwords requested within this block. Loaded on (DREAD_ALLOCATE | | DREAD_MERGE) && decoded PA<4:3>
Nomerge	Disable merging to this entry. clear = (LAST_FILL && !BOGUS_LF) | | RESET set = (STx | | STx_C | | FETCHx) && (DRD_MATCH | | MCSR<MAF_NMERGE>) && !TRAP | | LDx_L && !TRAP | | FIRST_FILL && SC_HIT && Scache operation | | FIRST_FILL && !Scache operation | | FIRST_FILL && IO_SPACE

Table 1–17: Dread Merge and Allocate Conditions

MAF Action	Description
DREAD_MERGE	entry_VALID && entry TYPE same as incoming LD (F/I, LWE/LWO/QW) && PA_MATCH && !NOMERGE && !MCSR<MAF_NMERGE> && DC_MISS && Valid LD instruction && !LDx_L && LD doesn't TRAP (MAF Full, LD-ST Silo Trap, LD-MAF Conflict Trap, DMM_ERR, IBOX Trap)
DREAD_ALLOCATE	(LD && DC_MISS && CANT_MERGE | | LDx_L) && LD doesn't TRAP

1.1.9.5.2 Dstream Writes WMB, FETCHx

The Write Buffer (WB) section of the MAF stores addresses and commands for all ST, STx_C, WMB, and FETCHx instructions. The data portion of the write buffer is in the CBOX. The WB section of the MAF consists of 6 entries of 32B block PAs. When a ST PA is presented to the MAF, it will be compared with all other PA's in the WB section. If it matches an address already stored in the WB, it will be merged with that entry. Otherwise, it will be allocated a new entry. Merging can be disabled to any entry by setting the NOMERGE bit. Unlike the DREAD section, ST's can overwrite previous ST's which have not yet been issued to the Scache. Therefore, no quadword slot data needs to be stored or checked for quadword conflicts.

When any ST is presented to the MAF, the MAF index in which the ST was loaded will be sent to the WB data section in the CBOX along with the LW address and an indication as to whether the opcode was a longword or quadword store. LW valid bits are kept by the CBOX for each entry of the WB.

LD/ST conflicts are detected by comparing all incoming ST's with previous LD's in the Dread section. A ST to the same 32B block as an outstanding LD will set the CONFLICT bit corresponding to the Dread entry with the same PA. The NOFILL bit is also set for the outstanding LD entry to ensure the Dcache does not end up with "stale" data. (If the previous LD was forced to miss, the ST may hit in the Dcache). If multiple Dread entries match the store, then multiple conflict bits will be set for the store's entry in the WB. A WB entry with conflict bits set will be blocked from issuing until the conflicting LD has completed. When the fill for a LD completes, it clears any conflict bits related to it in the WB.

ST/ST conflicts are detected by comparing all incoming ST's with previous WB entries. A ST that has the same 32B block address as a previous WB entry but cannot merge with the previous store will set the WMB bit associated with the entry allocated in the WB. The store will not be issued until all previous WB entries have completed.

The MAF does not do anything special for stores to "non-cacheable" memory. These are handled by the CBOX.

Each WB entry also has a FLUSH bit to indicate a HI-Priority arbitration for the entry to the MAF arbiter. Setting and clearing the FLUSH bit are shown in Table 1–19.

FETCHx instructions are allocated a new entry in the WB and are loaded with the NOMERGE bit set to disable subsequent WB entries from merging with it. A FETCHx will set the FLUSH bit to force the WB to empty.

The WMB instruction will set the WMB bit in the next available entry in the pending queue. This keeps any subsequent WB entries from issuing until all previous WB requests have been retired. It will also set the FLUSH and NOMERGE bits on all valid entries.

The STx_C instruction is allocated a new entry in the WB and is loaded with the NOMERGE bit set to disable subsequent WB entries from merging with it. A STx_C will cause the WB to flush all non-issued entries to the CBOX.

The WB will begin normal low-priority arbitration whenever a second entry is made to the buffer. The top entry of the WB arbitrates at low priority every 64 cycles or when a LDx_L instruction is executed. These events do not affect the state of the WB FLUSH bits.

Figure 1–11: WB PA Datapath

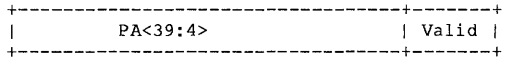


Table 1–18: WB PA Datapath

Field Name	Description
PA<39:4>	Physical address of 32B block. Loaded on WB_ALLOCATE. NOTE - PA<4:2> are sent directly to the CBOX WB as the LW address of the ST.
Valid	This address is a valid entry. clear = WB_DONE | | RESET set = WB_ALLOCATE

Figure 1–12: WB Control Bits

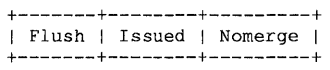


Table 1–19: WB Control Bits

Field Name	Description
Flush	A LD matched this WB entry. This entry should be issued to the CBOX at high priority. clear = entry is issued | | RESET set = !ISSUED && !TRAP && ((LD | | LDx_L) && WB_PA_MATCH && DC_MISS | | STx_C | | FETCHx | | VALID && (MB | | WMB)) NOTE: MCSR<WB_FLUSH> doesn't set the flush bits explicitly. It only forces a high priority request (overriding the state of the stored flush bits). When cleared, all requests return to using stored state.
Issued	The WB entry has been issued to the CBOX, but not completed. clear = WB_DONE | | RESET set = entry is issued
Nomerge	Disable merging to this entry. clear = WB_DONE | | RESET set = ((STx_C | | FETCHx | | (MB | | WMB) && VALID | | WB_ALLOCATE && MCSR<WB_NMERGE> | | (LD | | LDx_L) && WB_PA_MATCH && DC_MISS) && !TRAP) | | entry has won the ARB (WB_GNT) NOTE - WB_DONE = WR_DONE | | FETCH_DONE | | STxC_DONE

Table 1–20: WB Merge and Allocate Conditions

MAF Action	Description
WB_MERGE	ST && (!STx_C && !WMB && !FETCHx) && entry_VALID && PA_match && !NOMERGE && !MCSR<WB_NOMERGE> && !TRAP
WB_ALLOCATE && !TRAP)	((ST && CANT_MERGE) | | STx_C | | FETCHx)

1.1.9.5.3 Memory Barriers (MB)

On Memory Barriers (MB), the MB command will not be issued to the CBOX until all DREADs in the MAF have filled, and all WB entries in the MAF have been retired (this is detected when both free lists are "full"). The Memory Barrier sets the FLUSH bit for all valid entries in the WB. When MB_DONE is received from the CBOX on **RETURN_STATUS**, the MBOX will assert **MB_CLEAR** to restart the IBOX.

1.1.9.5.4 Write Memory Barriers (WMB)

On Write Memory Barriers (WMB), all writes issued before the WMB must finish before any writes issued after the WMB. The WMB does not get allocated an entry in the WB since it does not get issued to the CBOX. On a WMB instruction, the MAF sets the WMB bit in the next available entry in the WB pending queue and disables merging to all valid WB entries. This prevents subsequent WB entries from being issued until all previous WB entries have been retired. The WMB instruction sets the FLUSH bits for all valid entries in the WB.

1.1.9.5.5 Icache Read Misses

The IREF section holds PA's for all IBOX prefetch queue requests. It consists of 4 entries of 32B block PA's. When the IBOX prefetch queue requests a new block, it will supply the PA and prefetch queue index to the MAF. The PA will be loaded into the IREF section at the entry specified by the index (direct mapped to the Icache prefetch queue).

Figure 1–13: IREF PA Datapath

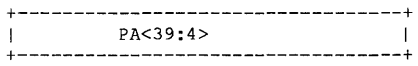


Table 1–21: IREF PA Datapath

Field Name	Description
PA<39:4>	Physical address of 32B block with OW order. Load on IREF_REQ.

1.1.9.6 MAF Issue to Scache

Whenever memory requests (DREAD, WB or IREF) are allocated a new entry in the MAF (i.e. not merged), the corresponding MAF index is loaded into a FIFO pending queue. Separate pending queues exist for each type of request because they have different arbitration criteria. Entries logged in the DREAD pending queue immediately begin arbitrating for Scache cycles and may bypass directly to the Scache. An entry logged in the WB pending queue waits for either 2 entries to be pending, a 64 cycle counter to overflow, a LDx_L instruction, or WB flush condition before it arbitrates for the Scache. Entries logged in the IREF pending queue immediately begin arbitrating.

Incoming loads may be bypassed directly to the Scache if there are no other pending MAF requests of any kind. If there is a load in pipe0, it will be issued to the Scache before it is known whether it hit in the Dcache or merged with a previous MAF entry. If either of these conditions turn out to be true, then the issued command is aborted and a load in pipe1 may be issued the following cycle if there are no other pending requests. This command is speculative also, and may be aborted if it merged with a previous MAF entry (in this case by issuing a NOP to the CBOX). If there were no LD in pipe1 of the previous cycle, then a new load from the current cycle may be bypassed instead. If there are no loads in pipe0, then pipe1 will be issued to the CBOX in the bypass cycle.

All MAF pending requests compete with requests from the BIU for free SC tag cycles. The priority levels are (from highest to lowest):

Table 1–22: MAF Issue Priority

Name	Description
SC_BUSY	BIU is using the SCtag this cycle.
WB_REISSUE	Reissue a previous WB address (WR_NOW cmd from the CBOX).
REPLAY	Previously issued address was not accepted by the BIU. It must be replayed.
WB_HI	Elevated WB request whenever any flush bits in WB are set
DREAD_PEND	DREAD requests basic arb.
MB	Memory Barrier issue request
IREF_PEND	IREF requests basic arb.
WB_LO	WB requests basic arb.
PREVIOUS BYPASS	No other pending requests. A load from a previous cycle has not yet been loaded into the MAF, but may be issued straight to the Scache.
BYPASS	No other pending requests. Load misses can be issued straight to the Scache in parallel with loading into the MAF.

Once a given request wins the arb for the Scache, the index is used to read the address out of the MAF to be driven to the Scache/BIU. If no other requests are pending in the MAF, LD addresses can be bypassed to the Scache in parallel with loading the MAF. The MAF index is shifted out of the pending queue and shipped as an ID for the address. The index is stored in a REPLAY queue and kept until the address has passed the point where it can be rejected by the BIU. Once an address is issued from any pending queue, that queue is incremented to start arbitrating at the next pending request (if any are pending). This process is independent for each pending queue (DREAD, WB, IREF). The MAF will only issue addresses when SC_BUSY has been deasserted for 2 consecutive cycles. This guarantees a minimum number of free cycles for the MAF request to complete in the Scache Tag store.

1.1.9.6.1 Reissuing WB addresses

The MAF has the capability of reissuing WB addresses which needed to get system permission before actually writing the data to the Scache. The data portion of the WB (in the CBOX) returns the MAF index, the **WR_NOW** command, and a bit indicating whether to issue the original 32B address or the other 32B address in the 64B block. The index is used to read the entry out of the WB section and reissue it to the Scache. If **WR_64B_REQ** is asserted, then **MAF_ADDR**<5> is inverted when the write is reissued.

When the **WR_NOW** command is received by the MBOX, the command is latched in the wb_reissue latch. The command waits there until it wins arbitration for issue to the CBOX (the CBOX may be busy). Once the command is issued, the latch is cleared.

1.1.9.6.2 Replaying an Address

The MAF has the capability of replaying addresses which have been issued to but not accepted by the CBOX. This is the mechanism the CBOX uses to back pressure the Mbox when it detects certain address or resource conflicts. When an address is issued to the Scache, the MAF index of that address is latched for 2 cycles. If the CBOX rejects the address by asserting **RETRY**, the CBOX packet (MAF command, type, and index) is placed into the replay pending queue. Once loaded, the replay queue will arb at high priority. Upon winning the arb, the replay index is used to read the address back out of the MAF. Replays have higher priority than DREAD, WB and IREF requests, but lower priority than any CBOX request. There can be up to 2 replays in the queue at any time.

Commands that are issued to the CBOX have already been checked for LD/LD, ST/ST LD/ST and ST/LD conflicts with previous MAF entries. For this reason, there is no need to maintain ordering of commands in the replay queue. With the exception of WB_REISSUE commands, addresses that are retried by the CBOX are placed into the bottom of the replay queue. This has the effect of round-robin issue from the replay queue when multiple retries occur.

The WB_REISSUE command is never placed in the replay queue. After the WB_REISSUE command is issued to the CBOX, it is piped along for a couple of cycles until it hits the "retry" point. If, at this time, the CBOX decides to "retry" it, then the WB_REISSUE command is re-latched into the wb_reissue latch instead of the replay queue.

NOTE

It is possible that a second MAF command will already have been issued to the CBOX when **RETRY** is asserted for the first command. The MAF will not automatically abort a command that is in the shadow of the retry; the CBOX is responsible for accepting or retrying each MAF command on its own.

1.1.9.7 Retiring MAF entries

The final stage in the life of a MAF entry is retiring the entry. For LD's, this occurs when the data has been returned to the Ebox/Fbox and the block filled into the Dcache (if the fill wasn't blocked for some reason). For ST's, the CBOX informs the MAF that the write has completed. Once the given operation has completed, the entry in the MAF is cleared and the MAF index for the entry is loaded into the corresponding FREE list to be reallocated later. For IREF's, the entry is retired upon issuing to the CBOX.

For Dcache fills, the CBOX returns the MAF index which was sent with the address when it was issued. The MAF reads the address, register number and format information out of the DREAD portion of the MAF. The address is driven to the Dcache interface, the register number to the IBOX and the format control to the respective formatter (E or F). This register number and format information is also piped along to the ECC error register, where it is loaded and locked if there is an ECC error associated with the fill from the CBOX. The MAF receives an MAF index for each octaword returned by the Scache/BIU and follows the same basic procedure for each. When the last octaword is sent to the MBOX, the CBOX also informs the MAF that this is the last fill. This allows the MAF to clear the entry and place the index back on the free list.

For ECC fills, the CBOX returns corrected data for the fill associated with the previous **M_RFB_ECC_ERR**. When the MAF receives the ECC_FILL return status, the register number and format information are read out of the ECC fill register to be sent to the IBOX and formatter along with

the corrected data. The ECC fill register is unlocked on read. The Dcache is not filled with the corrected ECC_FILL data.

For the WB, the BIU simply sends the MAF index and informs the MAF that the write corresponding to that entry has completed. The MAF clears the entry and places the index in the WB free list.

For IREF's, the CBOX controls the Icache fills directly. The MAF clears the entry when it is issued to the CBOX.

1.1.9.8 Loads from IO SPACE

IO space addresses will be handled like any other addresses in the MAF. They will be allowed to merge and issue as described above. When the entry is made to the MAF, the NOFILL bit is set if PA<39> is set (indicating IO space). The NOFILL bit will be read out of the MAF on fills and will disable the writing of the Dcache.

When an IO space DREAD is issued to the CBOX, merging may continue to that entry until approximately 2 cycles later, when the CBOX returns a FIRST_FILL command and the index of the issued command. The fill request will be aborted by the Scache miss, but the MAF uses the FIRST_FILL command to read out the quadword valid bits and send them to the CBOX as **DRD_MASK**<3:0>. These bits are sent out to the system as a quadword mask, indicating which quadwords in IO space were actually requested by the CPU. The NOMERGE bit for that entry is set at this time as well. For Istream IO loads the quadword mask is set to all ones.

1.1.9.9 Mbox Unavailable Traps

Trap conditions caused by the MAF are calculated during 5B based on available entries, DC_HIT and PA conflict results. All traps will be reported on the instruction causing the trap. If pipe1 causes the trap, pipe0 will continue and pipe1 will abort; if pipe0 causes the trap, both pipe0 and pipe1 will be aborted.

Table 1–23: Mbox Unavailable Traps

Trap Condition	Description
DREAD FULL	Any Load && DREAD_FREE_LIST_EMPTY | | Any Load in Pipe1 && DREAD_FREE_LIST_ONE_LEFT.
WB FULL	(Any Store | | FETCH | | WMB) && WB_FREE_LIST_EMPTY
LD-MAF CONFLICT TRAP	Any Load && DREAD_PA_MATCH && same quadword or longword
LD-ST SILO TRAP	Any Load && Index match immediately preceding Store && ST hit in Dcache

1.1.9.10 MAF Boundary Conditions

1.1.9.10.1 Dread Merge Cutoff Point

Incoming LD's may merge to existing LD MAF entries until one of the following conditions are encountered.

- FIRST_FILL and Scache lookup and Scache hit
- FIRST_FILL and IO space
- FIRST_FILL and not Scache lookup for existing entry
- Incoming Store to same address sets NOMERGE bit.

In the worst case, for a load that bypasses the MAF and hits in the Scache, there are 2 cycles in addition to the current cycle in which subsequently issued loads may merge to the original load.

LDx_L instructions will always allocate a new DREAD entry; the MAF will disable merging to LDx_L entries.

1.1.9.10.2 WB Merge Cutoff Point

Merging to the WB is constrained by how fast the data portion of the WB can supply the data to the Scache. For this reason merging is only allowed to entries that have not won the arb for the Scache. Merging is stopped during the cycle the WB wins the arb. Note, merging is cutoff before it is known whether the write is actually issued to the Scache.

1.1.10 Mbox and Dcache IPR's

NOTE

Traps are factored into MBOX IPR write operations unless noted otherwise.

Unless explicitly stated, IPRs are not cleared or set by hardware on chip or on timeout reset.

1.1.10.1 DTB_ASN, Dstream TB Address Space Number

The DTB_ASN register is a write-only register which, when not in PALmode, must be written with an exact duplicate of the ITB_ASN register's ASN field.

Figure 1–14: DTB_ASN

1.1.10.2 DTB_CM, Dstream TB Current Mode

The DTB_CM register is a write-only register which, when not in PALmode, must be written with an exact duplicate of the Ibox Processor Status (IPS) register's CM field. These bits indicate the Current Mode of the machine.

Figure 1–15: DTB_CM

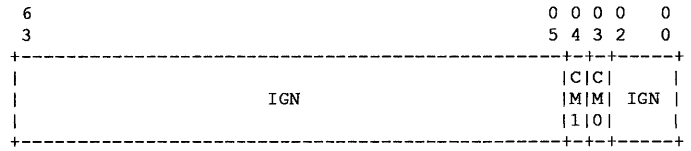


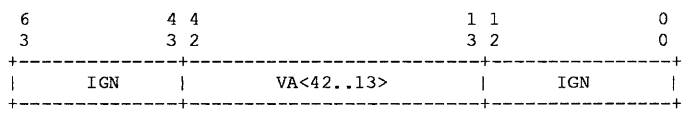
Table 1–24: DTB_CM Mode Bits

CM<1>	CM<0>	Current Mode
0	0	Kernel Mode
0	1	Executive Mode
1	0	Supervisor Mode
1	1	User Mode

1.1.10.3 DTB_TAG, Dstream TB TAG

The DTB_TAG register is a write-only register which writes the DTB tag and the contents of the DTB_PTE register to the DTB. To insure the integrity of the DTBs, the DTB's PTE array is updated simultaneously from the internal DTB_PTE register when the DTB_TAG register is written. The entry to be written is chosen at the time of the DTB_TAG write operation by a not-last-used algorithm implemented in hardware. A write to the DTB_TAG register increments the TB entry pointer of the DTB which allows writing the entire set of DTB PTE and TAG entries. The TB entry pointer is initialized to entry zero and the TB valid bits are cleared on chip reset but not on timeout reset.

Figure 1–16: DTB_TAG, Dstream TB Tag



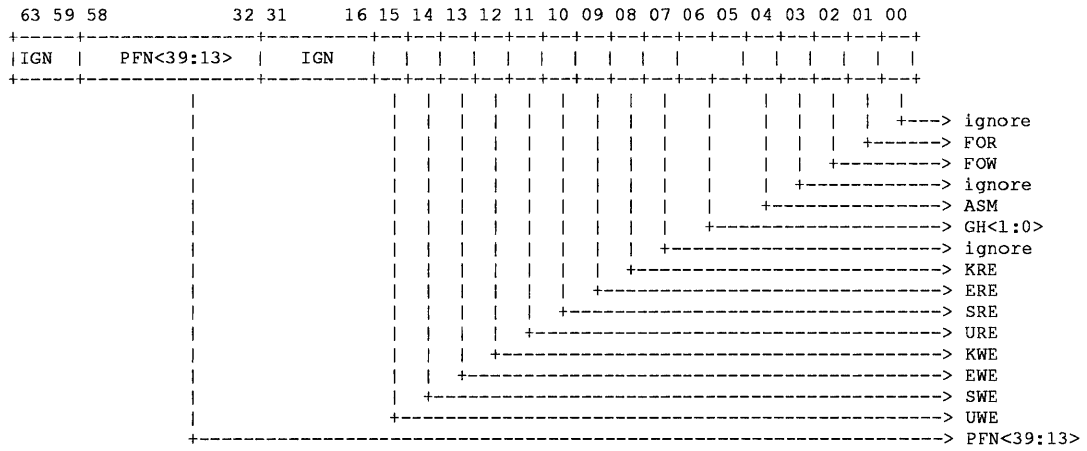
1.1.10.4 Dstream TB PTE, DTB_PTE

The DTB_PTE register is a read/write register representing the 64-entry DTB page table entries. The entry to be written is chosen by a not-last-used algorithm implemented in hardware. Writes to the DTB_PTE use the memory format bit positions as described in the Alpha SRM with the exception that some fields are ignored. In particular the PFN valid bit is not stored in the DTB.

To ensure the integrity of the DTB, the PTE is actually written to a temporary register and not transferred to the DTB until the DTB_TAG register is written. As a result, writing the DTB_PTE and then reading without an intervening DTB_TAG write will not return the data previously written to the DTB_PTE register.

Reads of the DTB_PTE require two instructions. First, a read from the DTB_PTE sends the PTE data to the DTB_PTE_TEMP register. A zero value is returned to the integer register file on a DTB_PTE read. A second instruction reading from the DTB_PTE_TEMP register returns the PTE entry to the register file. Reading the DTB_PTE register increments the TB entry pointer of the DTB which allows reading the entire set of DTB PTE entries.

Figure 1–17: DTB_PTE, Dstream TB PTE

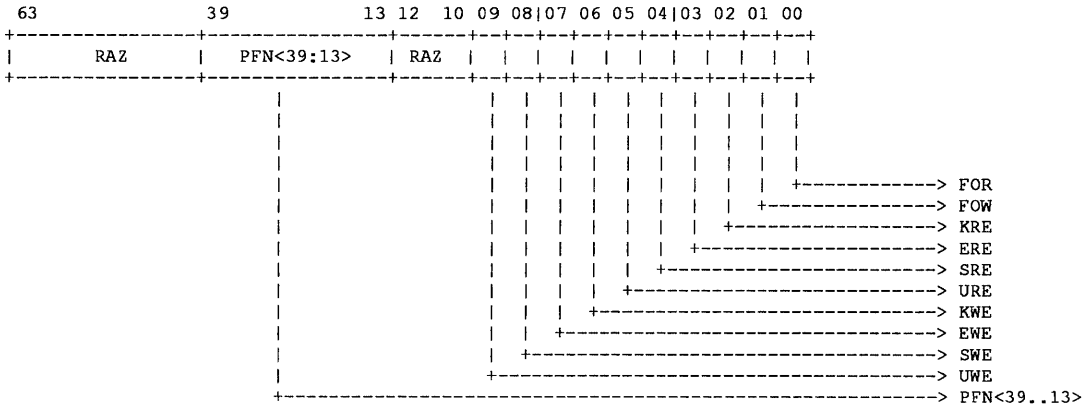


Note: The fields of the Page Table Entry are described in the ALPHA SRM.

1.1.10.5 DTB_PTE_TEMP

The DTB_PTE_TEMP register is a read-only holding register for DTB_PTE read data. Reads of the DTB_PTE require two instructions to return the PTE data to the register file. The first reads the DTB_PTE register to the DTB_PTE_TEMP register and returns zero to the register file. The second returns the DTB_PTE_TEMP register to the integer register file.

Figure 1–18: DTB_PTE_TEMP



1.1.10.6 MM_STAT, Dstream MM Fault Status Register

When D-stream faults or Dcache parity errors occur the information about the fault is latched and saved in the MM_STAT register. The VA, VA_FORM and MM_STAT registers are locked against further updates until software reads the VA register. MM_STAT bits are only modified by hardware when the register is not locked and a memory management error, DTB miss, or Dcache parity error occurs. The MM_STAT is not unlocked or cleared on reset.

Figure 1–19: MM_STAT, Dstream MM Fault Register

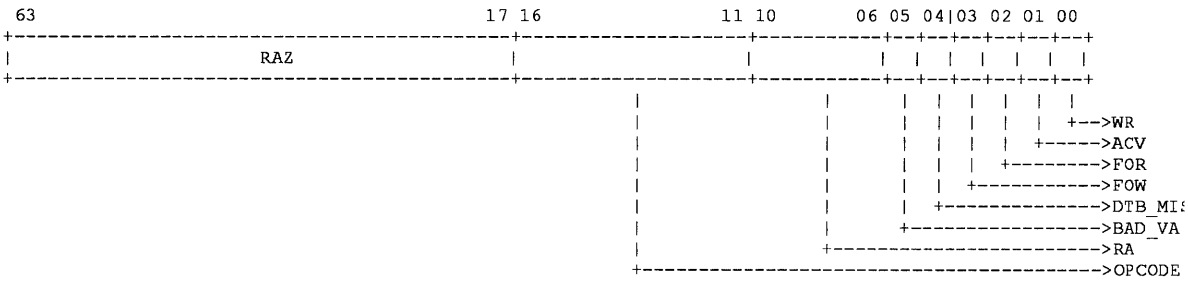


Table 1–25: MM_STAT Field Descriptions

Name	Extent	Type	Description
WR	0	RO	Set if reference which caused error was a write.
ACV	1	RO	Set if reference caused an access violation. Includes bad VA.
FOR	2	RO	Set if reference was a read and the PTE's FOR bit was set.
FOW	3	RO	Set if reference was a write and the PTE's FOW bit was set.
DTB_MISS	4	RO	Set if reference resulted in a DTB miss.

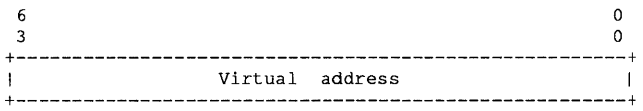
Table 1–25 (Cont.): MM_STAT Field Descriptions

Name	Extent	Type	Description
BAD_VA	5	RO	Set if reference had a bad virtual address.
RA	10:6	RO	Ra field of the faulting instruction.
OPCODE	16:11	RO	Opcode field of the faulting instruction.

1.1.10.7 VA, Faulting Virtual Address

When D-stream faults, DTB misses, or Dcache parity errors occur the effective virtual address associated with the fault, miss, or error is latched in the read-only VA register. The VA, VA_FORM, and MM_STAT registers are locked against further updates until software reads the VA register. The VA IPR is not unlocked on reset.

Figure 1–20: VA, Faulting VA Register



1.1.10.8 VA_FORM, Formatted Virtual Address

VA_FORM contains the virtual page table entry address calculated as a function of the faulting VA and the Virtual Page Table Base (VA and MVPTBR registers). This is done as a performance enhancement to the Dstream TBmiss PALflow. The VA is formatted as a 32-bit PTE when the NT_Mode bit, MCSR<SP0>, is set. VA_FORM is a read-only IPR, and is locked on any D-stream fault, DTB miss, or Dcache parity error. The VA, VA_FORM, and MM_STAT registers are locked against further updates until software reads the VA register. The VA_FORM IPR is not unlocked on reset. Figure 1–21 describes VA_FORM when MCSR<SP0> is clear. Figure 1–22 describes VA_FORM when MCSR<SP0> is set.

Figure 1–21: VA_FORM, Formatted VA Register for NT_Mode=0

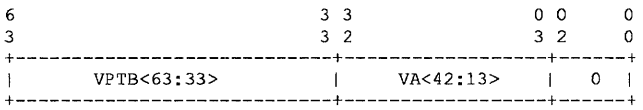


Figure 1–22: VA_FORM, Formatted VA Register, NT_Mode=1

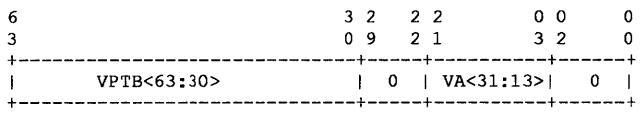


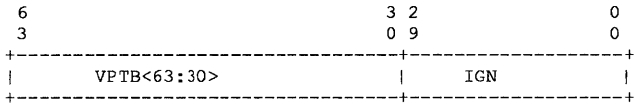
Table 1–26: VA_FORM Field Descriptions

Name	Extent	Type	Description
VA<42:13>	32:03	RO	Subset of the original faulting Virtual Address,NT_Mode=0.
VPTB	63:33	RO	Virtual Page Table Base address as stored in MVPTBR,NT_Mode=0.
VA<31:13>	21:03	RO	Subset of the original faulting Virtual Address,NT_Mode=1.
VPTB	63:30	RO	Virtual Page Table Base address as stored in MVPTBR,NT_Mode=1.

1.1.10.9 MVPTBR, Mbox Virtual Page Table Base Register

MVPTBR contains the virtual address of the base of the page table structure. It is stored in the Mbox to be used in calculating the VA_FORM IPR for the Dstream TBmiss PAL flow. Unlike the VA register, the MVPTBR is not locked against further updates when a Dstream fault, DTB Miss or Dcache parity error occurs. The MVPTBR is a write-only IPR that looks like this:

Figure 1–23: MVPTBR



1.1.10.10 DC_PERR_STAT, Dcache Parity Error Status

When a Dcache parity error occurs, the error status is latched and saved in the DC_PERR_STAT register. The VA, VA_FORM and MM_STAT registers are locked against further updates until software reads the VA register. If a Dcache parity error is detected while the Dcache parity error status register is unlocked, the error status is loaded into DC_PERR_STAT<5:2>. The LOCK bit is set and the register is locked against further updates (except for the SEO bit) until software writes a "one" to clear the LOCK bit. The SEO bit is set when a Dcache parity error occurs while the Dcache parity error status register is locked. Once the SEO bit is set it is locked against further updates until the software writes a "one" to DC_PERR_STAT<0> to unlock and clear the bit. Note the SEO bit does not get set when Dcache parity errors are detected on both pipes within the same cycle. For this particular situation, the pipe0/pipe1 Dcache parity error status

bits will indicate the existence of a second parity error. The DC_PERR_STAT is not unlocked or cleared on reset.

Figure 1–24: DC_PERR_STAT, Dcache Parity Error Status

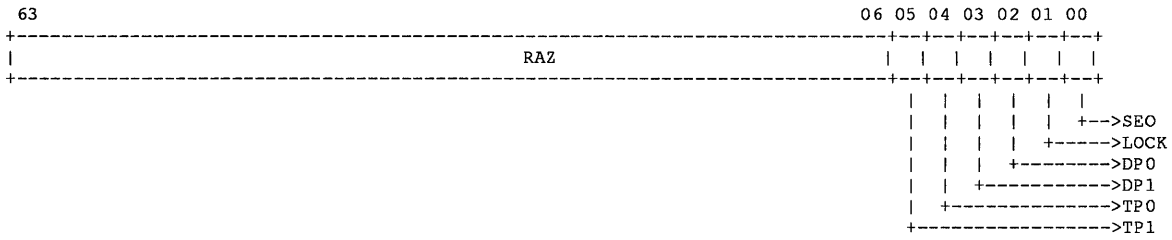


Table 1–27: DC_PERR_STAT Field Descriptions

Name	Extent	Type	Description
SEO	0	W1C	Set if second Dcache parity error occurred in a cycle after the register was locked. The SEO bit will not be set as a result of a second parity error that occurs within the same cycle as the first.
LOCK	1	W1C	Set if parity error detected in Dcache. Bits <5:2> are locked against further updates when this bit is set. Bits <5:2> are cleared when the LOCK bit is cleared.
DP0	2	RO	Set on data parity error in Dcache bank 0.
DP1	3	RO	Set on data parity error in Dcache bank1.
TP0	4	RO	Set on tag parity error in Dcache bank 0.
TP1	5	RO	Set on tag parity error in Dcache bank 1.

1.1.10.11 Dstream TB Invalidate All Process, DTBIAP

This is a write-only register. Any write to this register invalidates all DTB entries in which the ASM bit is equal to zero.

1.1.10.12 Dstream TB Invalidate All, DTBIA

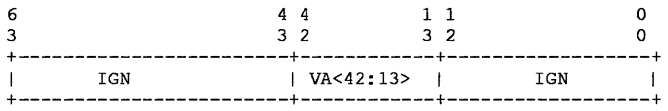
This is a write-only register. Any write to this register invalidates all 64 DTB entries, and resets the DTB NLU pointer to its initial state.

1.1.10.13 DTBIS, Dstream TB Invalidate Single

This is a write-only register. Writing a virtual address to this IPR invalidates the DTB entry that meets any one of the following criteria:

- A DTB entry whose VA field matches DTBIS<42:13> and whose ASN field matches DTB_ASN<63:57>.
- A DTB entry whose VA field matches DTBIS<42:13> and whose ASM bit is set.

Figure 1–25: DTBIS



NOTE

The DTBIS is written before the normal IBOX trap point. The DTB invalidate single operation will be aborted by the IBOX only for the following trap conditions: ITB miss, PC mispredict, or when the HW_MTPR DTBIS is executed in user mode.

1.1.10.14 MCSR, Mbox Control Register

The MCSR register is a read/write register that controls features and records status in the Mbox. This register is cleared on chip reset but not on timeout reset.

Figure 1–26: MCSR, Mbox Control Register

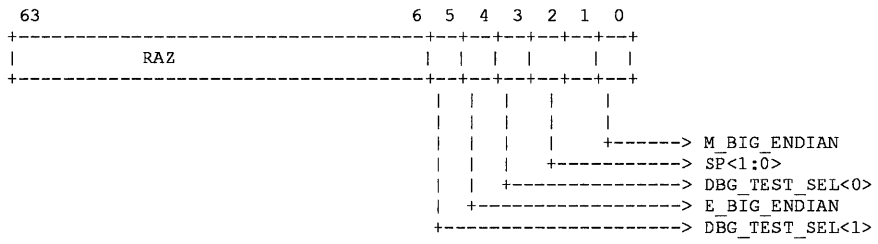


Table 1–28: MCSR Field Descriptions

Name	Extent	Type	Description
M_BIG_ENDIAN	0	RW,0	Mbox Big Endian mode enable. When set, bit 2 of the physical address is inverted for all longword Dstream references.

Table 1–28 (Cont.): MCSR Field Descriptions

Name	Extent	Type	Description
SP<1:0>	2:1	RW,0	Super page mode enables. SP<1> enables superpage mapping when VA<42:41> = 2. In this mode, virtual addresses VA<39:13> are mapped directly to physical addresses PA<39:13>. Virtual address bit VA<40> is ignored in this translation. SP<0> enables one-to-one super page mapping of D-stream virtual addresses with VA<42:30> = 1FFE(Hex). In this mode, virtual addresses VA<29:13> are mapped directly to physical addresses PA<29:13>, with bits <39:30> of physical address set to 0. SP<0> is the NT_Mode bit that is used to control VA formatting on a read from the VA_FORM IPR. Superpage access is only allowed in kernel mode.
DBG_TEST_SEL<0>	3	RW,0	Debug Test Select. The DBG_TEST_SEL<1:0> bits are used to control the Mbox/Cbox DECchip 21164-AA parallel test port mux selection. When DBG_TEST_SEL<1:0> = (00), the Cbox DBG_DATA<7:0> is selected. When DBG_TEST_SEL<1:0> = (01), the Mbox DCI debug packet is selected. When DBG_TEST_SEL<1:0> = (10), the Mbox MAF_OUT debug packet is selected. When DBG_TEST_SEL<1:0> = (11), the debug packet selection is dynamically controlled by the state of the RFB_DATA_VALID signal from the Cbox. (**Need a reference to the Mbox test packet signal description.**) These bits are used for diagnostic and test purposes only.
E_BIG_ENDIAN	4	RW,0	Ebox Big Endian mode enable. This bit is sent to the Ebox to enable Big Endian support for the EXTxx, MSKxx and INSxx byte instructions. This bit causes the shift amount to be inverted (ones-complemented) prior to the shifter operation.
DBG_TEST_SEL<1>	5	RW,0	Mbox debug packet select. See DBG_TEST_SEL<0>.

1.1.10.15 DC_MODE, Dcache Mode Register

The DC_MODE register is a read/write register that controls diagnostic and test modes in the Dcache. This register is cleared on chip reset but not on timeout reset.

Figure 1–27: DC_MODE, Dcache Mode Register

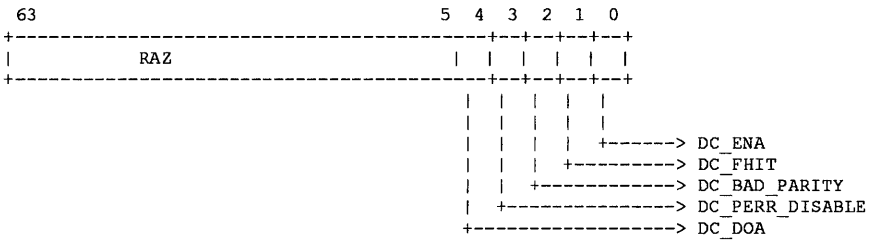


Table 1–29: DC_MODE Field Descriptions

Name	Extent	Type	Description
DC_ENA	0	RW,0	Software Dcache enable. Unless the Dcache has been disabled in hardware (DC_DOA is set), the DC_ENA bit enables the Dcache. (The Dcache is enabled if DC_ENA=1 AND DC_DOA=0). When clear, the Dcache command will not be updated by ST's or FILLs, and all LD's will be forced to miss in the Dcache.
DC_FHIT	1	RW,0	Dcache force hit. When set, this bit forces all D-stream references to hit in the Dcache.
DC_BAD_PARITY	2	RW,0	When set, this bit inverts the data parity inputs to the Dcache on integer stores. This will have the effect of putting bad data parity into the Dcache on integer stores that hit in the Dcache. This bit will have no effect on the tag parity written to the Dcache during fills or the data parity written to the CBOX Write Data Buffer on integer stores. Note: Floating point stores should NOT be issued when this bit is set because it may result in bad parity being written to the CBOX Write Data Buffer.
DC_PERR_DISABLE	3	RW,0	When set, this bit disables Dcache parity error reporting. When clear, this bit enables all Dcache tag and data parity errors. Parity error reporting is enabled during all other Dcache test modes unless this bit is explicitly set.
DC_DOA	4	RO	Hardware Dcache Disable. When set, the Dcache is faulty and has been disabled under hardware control (a programmable /readable fuse resides in the MBOX). All D-stream references will be forced to miss in the Dcache, and outstanding fills will be blocked from filling the Dcache. When DC_DOA is clear, the Dcache can be enabled under software control (DC_ENA=1). Note the DC_MODE register must be written under software control at least once before the state of the DC_DOA fuse is readable.

NOTE

The DC_MODE bits are only used for diagnostics and test. For normal operation, they will only be supported in the following configuration:

DC_ENA = 1
DC_FHIT = 0
DC_BAD_PARITY = 0
DC_PERR_DISABLE = 0

1.1.10.16 MAF_MODE, MAF Mode Register

The MAF_MODE register is a read/write register that controls diagnostic and test modes in the Mbox Miss Address File. This register is cleared on chip reset. Bit<5> is also cleared on timeout reset.

Figure 1–28: MAF_MODE, MAF Mode Register

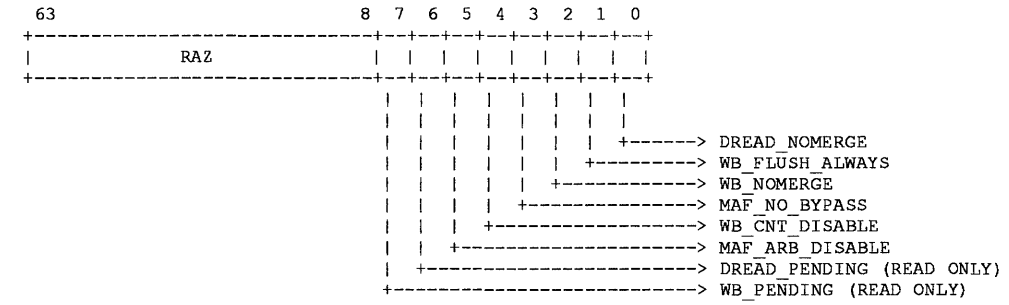


Table 1–30: MAF_MODE Field Descriptions

Name	Extent	Type	Description
DREAD_NOMERGE	0	RW,0	Miss Address File DREAD Merge Disable. When set, this bit disables all merging in the DREAD portion of the miss address file. Any load that is issued when DREAD_NOMERGE is set will be forced to allocate a new entry. Subsequent merging to that entry is not allowed (even if DREAD_NOMERGE is cleared).
WB_FLUSH_ALWAYS	1	RW,0	When set, this bit forces the write buffer to flush whenever there is a valid WB entry.
WB_NOMERGE	2	RW,0	When set, this bit disables all merging in the write buffer. Any store that is issued when WB_NOMERGE is set will be forced to allocate a new entry. Subsequent merging to that entry is not allowed (even if WB_NOMERGE is cleared.)

Table 1–30 (Cont.): MAF_MODE Field Descriptions

Name	Extent	Type	Description
MAF_NO_BYPASS	3	RW,0	When set, this bit disables Dread bypass requests in the MAF arbiter. All Dread requests will be loaded into the MAF pending queue before arbitration takes place.
WB_CNT_DISABLE	4	RW,0	When set, this bit disables the 64-cycle WB counter in the MAF arbiter. The top entry of the WB will arb at low priority only when a LDx_L is issued or a second WB entry is made.
MAF_ARB_DISABLE	5	RW,0	When set, this bit disables all Dread and WB requests in the MAF arbiter. WB_Reissue, Replay, Iref and MB requests are not blocked from arbitrating for the Scache. This bit is cleared on both timeout and chip reset.
DREAD_PENDING	6	R,0	This bit indicates the status of the MAF Dread file. When set, there are one or more outstanding Dread requests in the MAF file. When clear, there are no outstanding Dread requests.
WB_PENDING	7	R,0	This bit indicates the status of the MAF WB file. When set, there are one or more outstanding WB requests in the MAF file. When clear, there are no outstanding WB requests.

NOTE

Bits <5:0> of the MAF_MODE register are only used for diagnostics and test. For normal operation, they are supported in the following configuration:

DREAD_NOMERGE = 0
WB_FLUSH_ALWAYS = 0
WB_NOMERGE = 0
MAF_NO_BYPASS = 0
DREAD_WB_ARB_DISABLE=0
WB_CNT_DISABLE=0

1.1.10.17 DC_FLUSH, Dcache Flush Register

A write to this register clears all the valid bits in both banks of the Dcache.

1.1.10.18 ALT_MODE, Alternate mode

ALT_MODE is a write-only IPR. The AM field specifies the alternate processor mode used by HW_LD and HW_ST instructions.

Figure 1–29: ALT_MODE

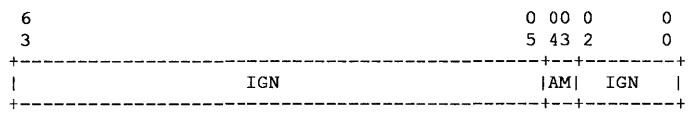


Table 1–31: ALT Mode

ALT_MODE<4:3>	Mode
0 0	Kernel
0 1	Executive
1 0	Supervisor
1 1	User

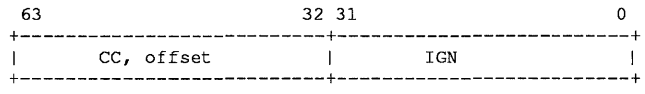
1.1.10.19 CC, Cycle Counter

DECchip 21164-AA supports a cycle counter as described in the Alpha SRM. The low half of the counter, when enabled, increments once each CPU cycle. The upper half of the CC register is the counter offset. CC<63:32> is written on a HW_MTPR to the CC IPR; bits <31:0> are unchanged. CC_CTL<32> is used to enable or disable the cycle counter. The lower half of the cycle counter is written on a HW_MTPR to the CC_CTL IPR.

The CC register is read by the RPCC instruction as defined in the Alpha SRM (The RPCC instruction returns a 64-bit value). The cycle counter is enabled to increment only 3 cycles after the MTPR CC_CTL (with CC_CTL<32> set) is issued. This means that an RPCC instruction issued 4 cycles after an MTPR CC_CTL that enables the counter will read a value that is 1 greater than the initial count. The cycle counter is disabled on chip reset.

The write-only CC Register looks like this:

Figure 1–30: CC, Cycle Counter Register



1.1.10.20 CC_CTL, Cycle Counter Control

The CC_CTL register is a write-only register that is used to write the low 32 bits of the cycle counter and to enable or disable the counter. Bits CC<31:4> are written with the value CC_CTL<31:4> on a HW_MTPR to the CC_CTL register. Bits CC<3:0> are written with zero; bits CC<63:32> are not changed. If CC_CTL<32> is set then the counter is enabled, otherwise the counter is disabled.

Figure 1–31: CC_CTL, Cycle Counter Control Register

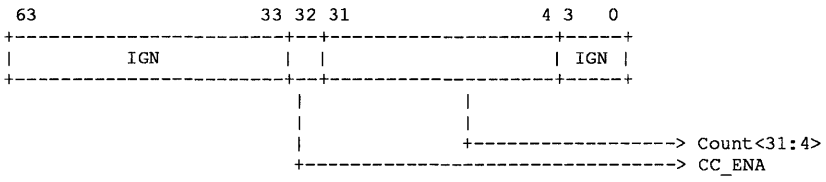


Table 1–32: CC_CTL Field Descriptions

Name	Extent	Type	Description
Count<31:4>	31:4	W0	Cycle count. This value is loaded into bits <31:4> of the CC register.
CC_ENA	32	WO	Cycle Counter enable. When set, this bit enables the CC register to begin incrementing 3 cycles later. An RPCC issued 4 cycles after CC_CTL<32> is written will see the initial count incremented by 1.

1.1.10.21 DC_TEST_CTL, Dcache Test TAG Control Register

The DC_TEST_CTL register is a read/write IPR used exclusively for test and diagnostics.

An address written to this register will be used to index into the Dcache array when reading or writing the DC_TEST_TAG register. See Section 1.1.10.22 for a description of how this register is used.

Figure 1–32: DC_TEST_CTL, Dcache Test TAG Control Register

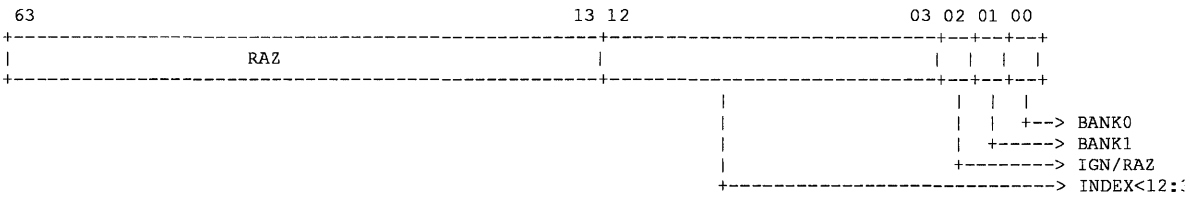


Table 1–33: DC_TEST_CTL Field Descriptions

Name	Extent	Type	Description
BANK0	0	RW	Dcache Bank0 enable. When set, reads from DC_TEST_TAG will return the tag from Dcache bank0 and writes to DC_TEST_TAG will write to Dcache bank0. When clear, reads from DC_TEST_TAG will return the tag from Dcache bank1.
BANK1	1	RW	Dcache Bank1 enable. When set, writes to DC_TEST_TAG will write to Dcache bank1. This bit has no effect on reads.
INDEX	12:3	RW	Dcache tag index. This field is used on reads/writes from /to the DC_TEST_TAG register to index into the Dcache tag array.

1.1.10.22 DC_TEST_TAG, Dcache Test TAG Register

The DC_TEST_TAG register is a read/write IPR used exclusively for test and diagnostics.

When DC_TEST_TAG is read, the value in the DC_TEST_CTL register is used to index into the Dcache and the value in the tag, tag parity, valid and data parity bits for that index are read out of the Dcache and loaded into the DC_TEST_TAG_TEMP IPR register. A zero value is returned to the integer register file. If BANK0 is set, the read is from Dcache bank0. Otherwise it is from Dcache bank1.

When DC_TEST_TAG is written, the value written to DC_TEST_TAG is written to the Dcache index referenced by the value in the DC_TEST_CTL register. The tag, tag parity, and valid bits are affected by this write. Data parity bits are not affected by this write (use DC_MODE<DC_BAD_PARITY> and force hit modes). If BANK0 is set, the write is to Dcache bank0. If BANK1 is set, the write is to Dcache bank1. If both are set, the write will occur to both banks.

Figure 1–33: DC_TEST_TAG, Dcache Test TAG Register

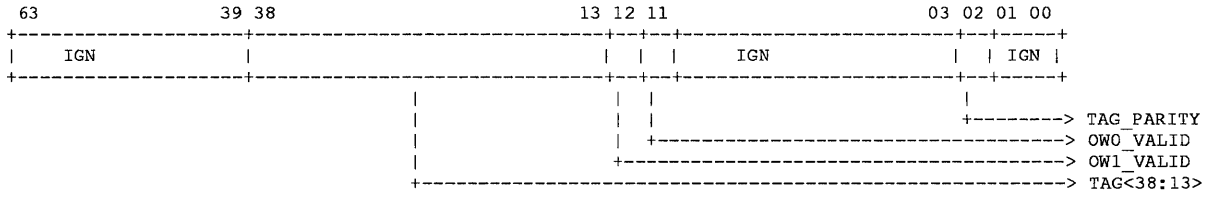


Table 1–34: DC_TEST_TAG Field Descriptions

Name	Extent	Type	Description
TAG_PARITY	2	WO	Tag Parity. This bit refers to the Dcache tag parity bit which covers tag bits 38 through 13 (valid bits not covered).

Table 1–34 (Cont.): DC_TEST_TAG Field Descriptions

Name	Extent	Type	Description
OW0_VALID	11	WO	Octaword valid bit 0. This bit refers to the Dcache valid bit for the low order octaword within a Dcache 32B block.
OW1_VALID	12	WO	Octaword valid bit 1. This bit refers to the Dcache valid bit for the high order octaword within a Dcache 32B block.
TAG	38:13	WO	Tag<38:13>. This refers to the tag field in the Dcache array. (Note: Bit 39 is not stored in the array)

1.1.10.23 DC_TEST_TAG_TEMP, Dcache Test TAG Temp Register

The DC_TEST_TAG_TEMP register is a read-only IPR used exclusively for test and diagnostics.

Reading the Dcache tag array requires a 2 step process. First, a read from DC_TEST_TAG reads the tag array and data parity bits and loads them into the DC_TEST_TAG_TEMP register. An undefined value is returned to the integer register file. A second read of the DC_TEST_TAG_TEMP register will return the Dcache test data to the register file.

Figure 1–34: DC_TEST_TAG_TEMP, Dcache Test TAG Temp Register

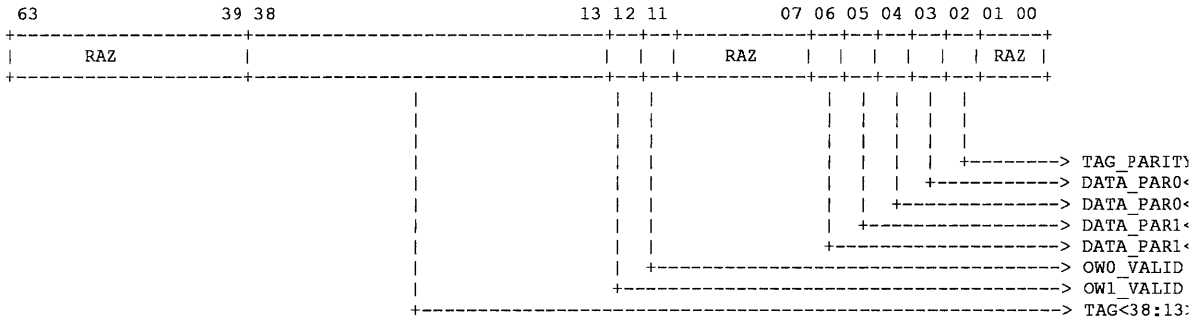


Table 1–35: DC_TEST_TAG_TEMP Field Descriptions

Name	Extent	Type	Description
TAG_PARITY	2	RO	Tag Parity. This bit refers to the Dcache tag parity bit which covers tag bits 38 through 13 (valid bits not covered).
DATA_PAR0<0>	3	RO	Data Parity. This bit refers to the Bank0 Dcache data parity bit which covers the lower longword of data indexed by dc_test_ctl<INDEX>.
DATA_PAR0<1>	4	RO	Data Parity. This bit refers to the Bank0 Dcache data parity bit which covers the upper longword of data indexed by DC_TEST_CTL<INDEX>.

Table 1–35 (Cont.): DC_TEST_TAG_TEMP Field Descriptions

Name	Extent	Type	Description
DATA_PAR1<0>	5	RO	Data Parity. This bit refers to the Bank1 Dcache data parity bit which covers the lower longword of data indexed by DC_TEST_CTL<INDEX>.
DATA_PAR1<1>	6	RO	Data Parity. This bit refers to the Bank1 Dcache data parity bit which covers the upper longword of data indexed by DC_TEST_CTL<INDEX>.
OW0_VALID	11	RO	Octaword valid bit 0. This bit refers to the Dcache valid bit for the low order octaword within a Dcache 32B block.
OW1_VALID	12	RO	Octaword valid bit 1. This bit refers to the Dcache valid bit for the high order octaword within a Dcache 32B block.
TAG	38:13	RO	Tag<38:13>. This refers to the tag field in the Dcache array. (Note: Bit 39 is not stored in the array)

1.2 Reset and Initialization

The MCSR, DC_MODE and MAF_MODE IPRs are cleared on chip reset; all other IPRs must be reset by PALcode.

On both chip and timeout reset, the MAF operating state will be reset. This includes clearing all status bits in the MAF file, clearing all pending queues, setting the free list indices, clearing the WB counter and MB request flip-flops, clearing the replay and wb_reissue valid bits, and clearing the valid bits in the register number array.

The cycle counter IPR, CC, is disabled on chip reset.

The DTB pointer will be initialized to point to the bottom entry and the DTB valid bits will be cleared on chip reset but not on timeout reset. The valid bits in the Dcache will not be cleared on either reset.

Palcode is expected to read the VA register to unlock the VA, VA_FORM and MM_STAT registers, and to write to the DC_PERR_STAT register to unlock and clear the status bits. (DC_PERR_STAT<SEO> is unlocked and cleared under separate control from the remaining status bits).

1.3 Error Handling and Recording

The MM_STAT, VA, and VA_FORM registers record the status of an instruction causing a memory management fault or Dcache parity error. These registers are locked against further updates until PALcode reads the VA register.

The DC_PERR_STAT register records the tag and data parity status for the instruction causing a Dcache parity error. The DC_PERR_STAT register is locked against further updates (except for the SEO bit) until software writes a "1" to the LOCK bit. A W1C on the LOCK bit will unlock and clear the tag and data parity status bits. The DC_PERR_STAT<SEO> bit is set if a Dcache parity error occurs when the DC_PERR_STAT<LOCK> bit is set. The SEO bit is locked against further updates until software writes a "1" to unlock and clear the bit.

The Dcache is flushed when an ECC error occurs on a fill. The register number and format information for the associated DREAD entry are loaded into the MAF ECC error register and the register is locked against further updates. The MAF sets the NOFILL bit when the ECC error occurs on the first half of a fill. When the CBOX returns the corrected data, it is forwarded to the EBOX/FBOX register file but is not written to the Dcache. The MAF inititates a read of the ECC error register to supply the register number and format control on an ECC_FILL and unlocks the ECC error register for future updates.

1.4 Test Aspects

The Mbox is equipped with the standard LFSR chains used for chip testability and the parallel port used for debug. Detailed information on the specification of these can be found in the EV5 external spec.

1.5 Performance Monitoring Features

The performance monitoring hardware is located in the Ibox. The normal Mbox trap and Dcache hit signals will be used to count DC misses, DTB misses, memory management errors, Dcache parity errors, and replay (MAF_UNAVAIL) traps. The Mbox is sending special signals to the Ibox to indicate whether a load in pipe0 or pipe1 got allocated a new entry in the MAF (used in conjunction with traps and DC_HIT to count load merging). The Mbox also sends signals that indicate the WB_FULL or MAF_FULL condition has occured for stores in pipe0 and loads in pipe0 or pipe1.

1.6 Revision History

Table 1–36: Revision History

Who	When	Description of change
J.Meyer, L.Noack, B.Benschneider	13-Dec-91	Initial spec.
J.Meyer	02-Mar-92	Updates after 2nd Mbox review.
J.Meyer	21-May-92	Updates for new Mbox timing.
S.Britton	20-November-92	Updates for architecture and implementation changes.
S.Britton	17-February-93	Updates for architecture and implementation changes.
S.Britton	01-April-93	Updates for Mbox IPRs
B.Benschneider	27-January-94	Update to Pass1 implementation

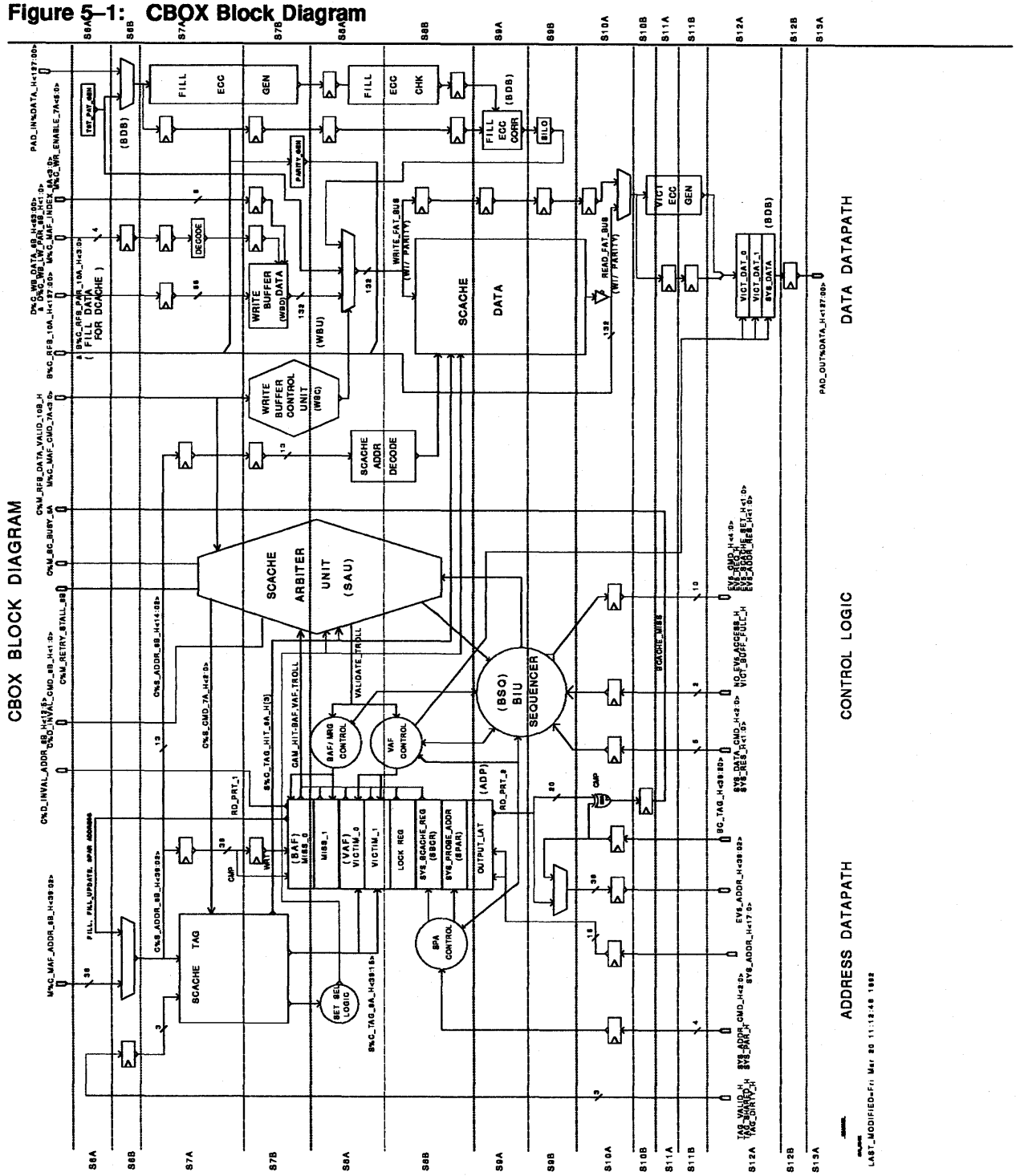
Chapter 5

The Cbox

5.1 Overview & Block Diagrams

The EV5 CBOX is responsible for providing data to/from the Scache and the System data stores. The CBOX consists of three major sections: the SCache Arbiter Unit (SAU), the Write Buffer Data Unit (WBU) and the Bus Interface Unit (BIU). The SAU prioritizes access requests from the MBOX and the BIU to the Scache. WBU provides the storage for write data and is responsible for the successful completion of store requests. The BIU controls the interface to the EV5 pin bus. The block diagram of the CBOX is shown in Figure 5–1.

Figure 5–1: CBOX Block Diagram



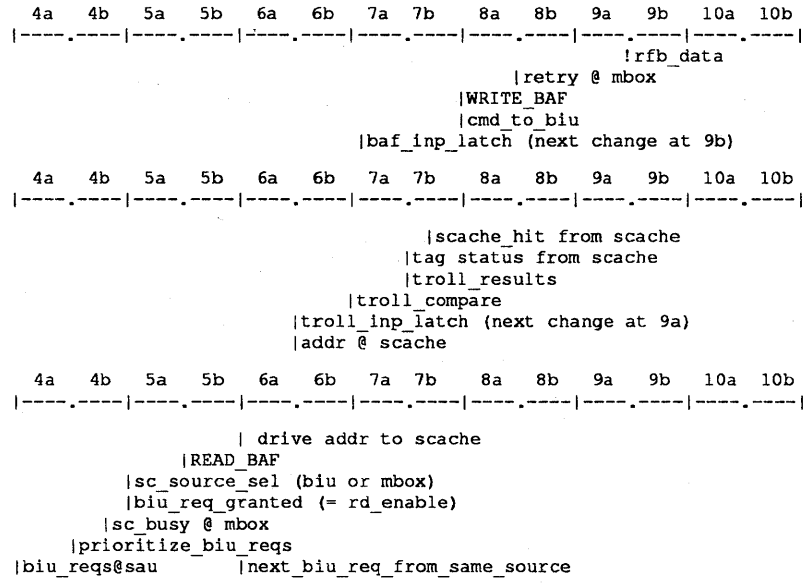
5.2 Functional Description

5.2.1 Scache Arbiter Unit

The Scache Arbiter Unit (SAU) arbitrates access to the SCache. Primary sources of requests for the SCache access are the Mbox and the Bus Interface Unit (BIU). Arbitration requests from the BIU have higher priority over the requests from the Mbox. In the idle mode, the Mbox has access to SCache if no BIU requests are pending. The Cbox asserts an early SCache busy signal (SC_BUSY) to the Mbox if the BIU needs access to the SCache.

Figure Figure 5–2 shows the general flow of the SCache Arbiter Unit (SAU) pipe stages. Requests arrive from the BIU Address File (BAF), the Victim Address File (VAF) and the System Probe Arbiter (SPA) in 3b. These requests are presented to the arbiter in 4a, Arbiter runs in 4a-4b and generates the SC_BUSY signal to Mbox if a valid BIU request for SCache is found. A grant signal is sent to the BIU controller that requested access. The address register is read out in 5b and driven to SCache Tags in 6a. This address is also latched into the TROLL register input latch in 6b. TROLL results are driven in 7a. The SCache ships all block tag status bits to the CBox in 7a. The hit signal arrives in 7b at the Cbox from the SCache. Based on these signals the SAU generates appropriate merge, retry, set number, victim and Ibox allocate cycle signals.

Figure 5–2: SAU Pipe Stages



5.2.1.1 Mbox Requests

Requests from the Mbox are primarily DCache or ICache Load Misses and Stores. Each Mbox request is accompanied by a miss address file index physical address and a command. Physical address bits and the command are driven to both the SAU and the SCache by the Mbox. Mbox miss address file index information is sent directly to the SAU.

The Mbox is guaranteed access to SCache Tags in cycles s6b and s7b ONLY if it sees the SC_BUSY signal de-asserted during both cycles s4b and s5b. In the idle mode, Mbox must drive a NOP command if it has no valid Scache requests pending. See Figure 5–3.

Figure 5–3: SC_BUSY and Mbox Command Issue

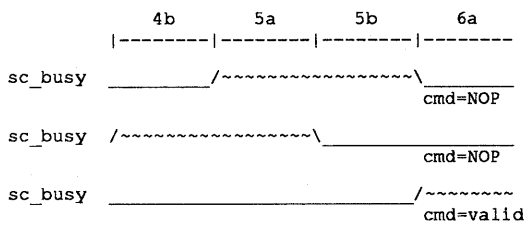


Table 5–1: Commands from Mbox

Command	Description
NOP	No Operation
DREAD	Dref Read
IREAD	Iref Read
LDX_L	Load Locked from Memory
FETCH	Fetch from Memory
FETCH_M	Fetch from Memory with Modify Intent
MB	Memory Barrier
WR	Write to Scache
STX_C	Store Conditional to Memory

Table 5–2: Encoded Cbox Return Status to Mbox (7a and 8a signals)

Command	Description
NOP	No Operation
FIRST_FILL	First Octaword of the I/D Cache fill cycle
LAST_FILL	Last (second) Octaword of the I/D Cache fill cycle
WR_DONE	Write Operation Done
STXC_DONE	Store Conditional Done Successfully
FETCH_DONE	Fetch Done
MB_DONE	Memory Barrier Done

Figure 5–4: Possible FIRST_FILL/LAST_FILL sequences from Cbox to Mbox

```
1.  FA   FB  LB  LA
2.  FA   FB  -   LA              ! FB is speculative (if B is Scmiss)
3.  FAe  FB  LB  ECCA  LA
4.  FAe  FB  LB  LA    EFA  LA   ! Possible sequence from Cbox
                 ^
                 | Mbox can not handle this LAST_FILL
                 ! Cbox will send this in a 3cycle Bcache config
5.  FA   FB  LB  LAe   ECCA
6.  FA   FB  LA  LB    LB        ! Will not occur

Key:
A   = Scmiss and Bchit
B   = Schit
FA  = FIRST_FILL
LA  = LAST_FILL
FAe = FIRST_FILL has ecc error
LAe = LAST_FILL has ecc error
EFA = ECC Corrected FIRST_FILL
ELA = ECC Corrected LAST_FILL
```

Table 5–3: Cbox Special Signals to Mbox

Signal[1]	Description
SC_BUSY	Cbox is accessing the Scache 4a signal
RETRY	Mbox request denied. Mbox must retry. 8a signal
RETURN_INDEX	Mbox MAF index of the returning fill data. 7a and 8a signal
FILL_OW	Which OW is this fill. bit 4 of the address
WRITE_NOW	High priority write for broadcast data
WR_MAF_INDEX	Mbox MAF index for writes
WRITE_32B_REQ	Distinguish between 32B and 64B write request

[1]See global signal list for actual signal names and widths

5.2.1.1.1 Requests from Mbox

5.2.1.1.1.1 Load requests

All reads are looked up in the Scache for a hit or miss. On a SCache hit the requested 32B are sent in two consecutive cycles (s9b, s10b) on the Read Fat Bus (RFB). The Mbox MAF index and return status is driven back to the Mbox in s7b before each 16B data transfer. The Ibox Allocate Cycle information is dispatched two cycles before each RFB cycle. Both Ibox bubble information and the FIRST_FILL return status are speculative for the first OW fill to Dcache and non-speculative for second OW fill.

On a SCache miss, the address is forwarded to the BIU for a lookup in the external memory system. Mbox reads are merged at the 64B level at the BIU when in the 64B mode of operation. Only accesses to different 32B within the same 64B are merged. All other requests are re-tried if it can not be merged or if BIU resources are not available. No merging is done in the 32B mode of operation.

IO space reads are treated as SCache misses and are forwarded to the BIU. If the address maps to an IO space reserved for EV5 then it is treated as a read to the Cbox IPR's.

Loads to IPR space are processed by SAU. These addresses do not get loaded in the BAF register. Requested IPR data is returned in the upper Quadword of the lower Octaword of a 32B block which is sent to the Mbox in the first of two RFB cycles. RFB data valid is driven ONLY for the first Octaword of the fill. Appropriate allocate cycle signal is also driven for the first Octaword returned.

Speculative allocate cycle signal is asserted by Cbox for integer Dreads and reads to Cbox IPR space. Allocate cycle for the second Octaword of the fill from Scache is non-speculative.

5.2.1.1.1.2 Load Locked requests

If a LDxL request hits in the SCache, the Lock flag is set and the requested 32B are returned to the Mbox. A Lock command is also sent to the BIU. On a SCache miss, the command is forwarded to the BIU which does a lookup in the Bcache. The Lock flag is set only when the fill returns from external memory.

On a BCache hit, the Lock flag is set and the data is returned to both the Mbox and the SCache. On a BCache miss the request is further forwarded to the SI.

LDxL to IO space are treated as misses in the SCache and the request is forwarded to the BIU. The address is loaded into the BAF register.

LDxL to IPR space is not permitted.

5.2.1.1.1.3 Store requests

Stores requests are looked up in the SCache to determine the state of the block. Stores are accepted to a private/dirty block. Stores to a private/clean block require permission from the SI. These requests are re-tried if BIU Address File (BAF) resources are full. Upon receipt of the required permission from the SI, these stores are re-initiated by the Cbox. Cbox provides write permission to the SCache to mark the block as dirty.

Stores to shared blocks require permission from the SI. These requests are re-tried if either the BAF or the VAF resources are unavailable. Upon receipt of permission from the SI, these writes are re-initiated by the Cbox by requesting Mbox to reissue the command. Cbox provides write permission to the SCache.

Store to IO space are forwarded to the SI. The SAU dispatches WRITE_DONE (WD) return status to the Mbox in 8a for all IO writes. Allocate cycle is sent to the Ibox along with the STXC_DONE return status ONLY for STX_C commands to non-IPR IO space.

STXC to IPR space is NOT permitted.

5.2.1.1.1.4 Store Conditionals

A STxC to a private/dirty block in SCache succeeds if the Lock flag is found set and the Lock register matches the store address. Cbox returns a STXC_DONE status to the Mbox. If the Lock flag is found cleared, a STXC_FAIL status is returned. The RFB data valid signal is also driven to the Mbox to help load the register file with the status of the lock flag.

A STxC to an IO space is treated as an SCache miss and forwarded to the system. The lock flag is cleared. The BIU completes the transaction.

STX_C to IPR space is not permitted.

The Cbox sends an invalidate to the DCache when it receives a STxC from the Mbox in all instances except when the STXC hits a private and dirty block in the Scache and the the Lock flag is found set. The Cbox also provides appropriate one cycle allocate information to the Ibox on completion of a StxC. STxC data is dropped if it is not to I/O space and the STxC fails.

Allocate_cycle signal, RFB data valid, return index and the return status bits are driven back to the Mbox when the second Octaword is written to the Scache.

5.2.1.1.1.5 Fetch, FetchM and MB

These commands do not access the SCache and are forwarded directly to the BIU. A MB_DONE, FETCH_DONE return status is dispatched back to the Mbox upon receipt of the command from Mbox in cycle 8a. Mbox will ensure appropriate instruction ordering around MB.

5.2.1.1.1.6 Commands to BIU

The Scache receives commands from the Mbox, the Bcache Sequencer (BSQ), the VAF controller and the SPA controller. The SAU forwards all Mbox commands and the SH_UPDATE and SC_INVAL to the BIU. These get loaded in to BAF register. Mbox commands that are re-tried by the SAU are not loaded in to the BAF.

Table 5–4: Mbox Commands and Scache Arbiter Actions

From Mbox	To Scache	Type	Hit	Status	Action	To BIU/BSQ
I/DREAD	READ	-	hit	-		NOP
	READ		miss	-	-	READ
	READ	IO	-	-	-	READ
	READ	EV5_IO	-	-	-	NOP
LDxL	READ	-	hit	-	set(L)	LOCK
	READ	-	miss	-		LDXL,LOCK
	READ	IO	-	-	-	LDXL,LOCK
WRITE	WRITE	-	hit	priv/dirty	set(M,D)	NOP
	WRITE	-	hit	priv/clean	-	WRITE
	WRITE	-	hit	shared	-	WRITE
	WRITE	-	miss	-	-	WRITE
	WRITE	IO	-	-	-	WRITE
	WRITE	IPR	-	-	-	NOP
	WRITE	permission [2]	-	priv/clean	set(M,D)	NOP
	WRITE	permission	-	shared	clr(D,M)	Error. Will not occur.
STxC	WRITE	Lock=1	hit	priv/dirty	set(M) STxC DONE	CLR_LOCK
	WRITE	Lock=1	hit	priv/clean	-	WDTY
	WRITE	Lock=1	hit	shared	-	WBDCST_LOCK
	WRITE	-	miss	-	-	WRITE_LOCK
	WRITE	IO	-	-	-	WRITE_LOCK
	WRITE	permission Lock=1	-	priv/clean	set(M) STxC DONE	NOP
	WRITE	permission Lock=1	-	shared	clr(D,M) STxC pass	NOP
	WRITE	permission Lock=0	-	priv/clean	STxC fail	NOP
	WRITE	permission Lock=0	-	shared	STxC fail	NOP
FETCH	NOP	-	-	-	FETCH_DONE	FETCH

[2] permission to write a block in the scache if its shared or its clean

Table 5–4 (Cont.): Mbox Commands and Scache Arbiter Actions

From Mbox	To Scache	Type	Hit	Status	Action	To BIU/BSQ
FETCHM	NOP	-	-	-	FETCH_DONE	FETCHM
MB	NOP	-	-	-	MB_DONE	MB

5.2.1.1.2 Invalidates to DCache

In general, the Cbox invalidates a DCache block whenever that block is displaced from the SCache. A DCache block is also invalidated on all STxC commands from the Mbox except one which hits a private dirty block in the Scache and the Lock flag is found set.

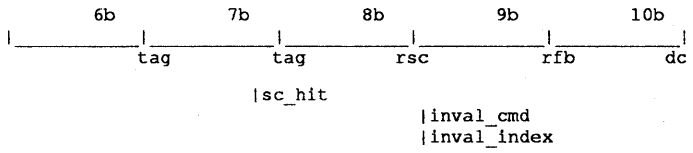
On an Update command from the system, the Cbox sends an invalidate to the Scache and DCache.

DCache invalidates are also sent by the Cbox on a FLUSH command from the system.

Invalidate command and index bits to the DCache will follow the same timings as a fill cycle from the Cbox. This helps avoid potential fill-inval-fill sequences to the same block.

Invalidates are sent to the Dcache on both WRITE misses and IREAD misses in the Scache.

Figure 5–5: Invalidate Timing



5.2.1.1.3 Retries and Merging of Mbox requests

Mbox requests that result in misses in the SCache are re-tried, merged with already pending misses or queued at the BIU for a fill from the external memory. In general, BIU requests have higher priority over Mbox requests and will cause a retry of the Mbox request until the BIU request is completed. BIU accesses can occur at any time and are indicated by asserting the SC_BUSY signal. Mbox should not issue any commands to Cbox if it sees SC_BUSY asserted. See Figure 5–3.

The MBOX has to replay instructions if they are blocked by the TROLL registers (which contain DCache indices of outstanding requests to the external system) or if certain BIU resources are not available. A RETRY signal is asserted in s8a to the Mbox. See Figure 5–6. If BIU address and data resources are full, all Mbox requests that result in non-mergeable SCache misses are re-tried.

A maximum of 2 non-mergeable Mbox requests that miss in the SCache can be queued in the Cbox for external memory access. Additional Mbox requests that result in SCache misses can be accepted if it merges with any of the previous misses. Only accesses to the same 64B and different 32B block are merged. Accesses to the same 64B and same 32B block of a pending miss are re-tried to avoid double pumping of the same octaword to two different Mbox MAF indices. Merging is stopped as soon as the fill to that address starts.

Stores that miss in the SCache are re-tried if BAF resources are full. Stores to a private/clean block in SCache that need permission from the System Interface (SI) are re-tried if BAF resources are full. Stores to a shared block in the SCache are re-tried if BAF or VAF resources are full. A WR command is always merged to the existing WR address in the BAF if it is to a private/clean or shared block awaiting permission from the SI.

Instructions from the Mbox in the shadow of a retry are accepted by the Cbox. These may be re-tried if any of the retry conditions are true. However if the Mbox command issued is a re-issued WRITE (initiated by the WBU), all Mbox instructions in the shadow of this write will be aborted if the reissued write is re-tried by the Cbox.

Instructions from the Mbox are re-tried if the WBU asserts STOP_WRITE or STOP_READ signals to the SAU.

Table 5–5: Mbox Retry Conditions

Request	Action	Condition
Read	Retry	(sc_miss) && !(victim) && (baf_full)
	Retry	(sc_miss) && (victim) && (vaf_full + baf_full)
	Retry	(sc_miss) && (troll_match) && (fill in progress)
	Retry	(sc_miss) && (troll match) && !(fill in progress) && (same 64B) && (same 32B)
	Merge	(sc_miss) && (troll_match) && !(fill in progress) && (same 64B) && (diff 32B)
	Retry	(sc_hit) && (troll match)
	OK	(sc_hit) && !(troll match)
Write	Retry	(troll_match) && (input_cmd == wr) && (diff WB entry)
	Merge	(troll_match) && (input_cmd == wr) && (same WB entry)
	Retry	(sc_miss) && (victim) && (vaf_full + baf_full) && ((!64B mode) or (64B mode && !populate))
	OK	(sc_hit) && (priv/dirty)
	Retry	(sc_hit) && (priv/clean) & (baf_full)
	Retry	(sc_hit) && (shared) & (baf_full + vaf_full) ((!64B mode) or (64B mode && !populate))
	Retry	(IO mode) && (vaf_full)

Figure 5–6: Mbox Retry on Miss

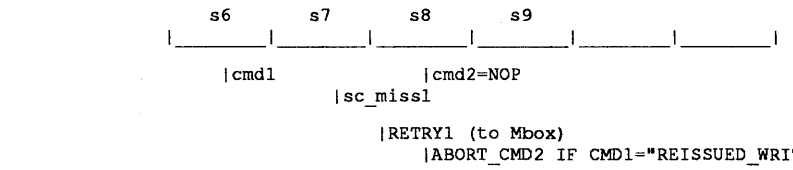
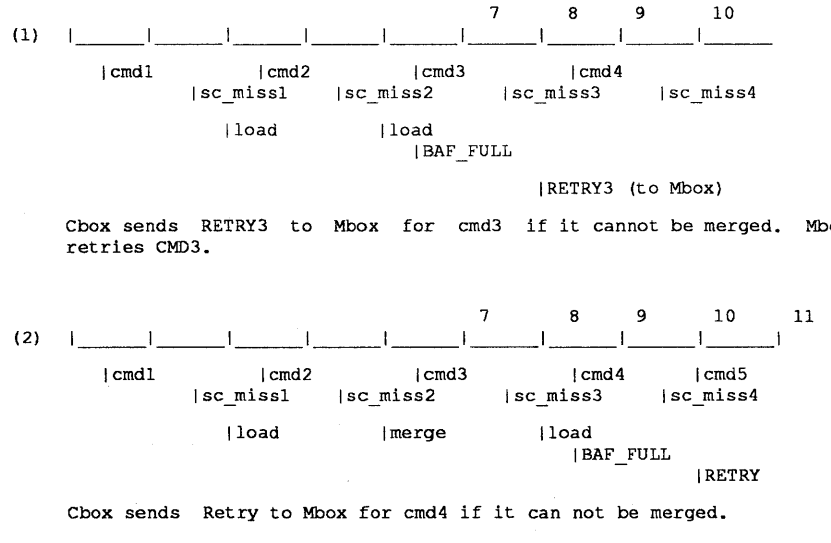


Figure 5–7: Retry on BIU resources full



5.2.1.1.4 Read/Write Ordering from Mbox

Read/Write conflicts are resolved at the Mbox. Read-Read conflicts to same 32B block are resolved at the Cbox by issuing a retry to Mbox. Following table lists some actions under SCache miss conditions.

Table 5–6: Mbox Read/Write Ordering

Request	Conditions	Action @ Mbox (Why)	Action @ Cbox on sc_miss
Read then Read	To same 32B and same 64B	Could Issue (lw/qw, nomerge)	Retry second Read
	To diff 32B and same 64B	Could Issue (only check 32B)	Process normally. Merge if possible
Read then Write	To same 32B and same 64B	Will not issue Write (ld/st conflict)	Flag error if issued
	To diff 32B and same 64B	Will issue (only check 32B)	Flag error if issued
Write then Read	To same 32B and same 64B	Will not issue (st/ld conflict)	No action
	To diff 32B and same 64B	Could Issue (only check 32B)	No action
Write then Write	To same 32B and same 64B	Merged at Mbox (Mbox issues only one command)	Process Normally
		Not merged (stop merge on issue)	Retry second if first not done. Process in order
	To diff 32B and same 64B	Issue (merge only within 32B)	Process Normally
		Not merged	Retry first if first not done. Process in order.

5.2.1.2 TROLLing of Scache Access Requests

The Scache arbiter trolls load and store accesses to DCache indices that are already in the read/write miss pending or write permission pending state. All incoming requests are compared against physical address bits <12:5> in the BIU Address File (BAF) and the System Bcache Register (SBR).

SCache load misses are forwarded to the BIU for look-up in the Bcache. Bits <12:6> of the physical address are entered into the TROLL register. The miss address file index for the loads, obtained from the Mbox, is sent to the BIU. The BIU returns this index information back to the Mbox when the fill arrives.

In the fastest Bcache implementation, fill data (one octaword at a time) arrives at a rate of 12ns or 3 EV5 clock cycles. The tag and status bits of the filling block are written into the SCache with every octaword of fill data. This leaves the 64B block in the SCache in a partially valid state until the fill is complete. To avoid read/write accesses to the partially valid block between octaword fills, such accesses are filtered by the TROLL register and a retry is sent to the requester. The SCache is indexed using bits <14:5> of the physical address. However the TROLL compare is done on a smaller field of bits <12:5> equivalent to the DCache index in order to prevent interleaved fills into the same DCache index. This is explained below.

Let us assume that the TROLL compare is performed on the SCache index. If there is a LD miss in the DCache and in the SCache, the associated SCache index is loaded into the TROLL register. Before the fill for this LD completes, another LD miss to the same DCache index (phy_address<12:5>) as the previous LD is issued to the SCache. This LD may not be prevented from accessing the SCache if it has a different SCache index (phy_address<14:5>) and may hit in the SCache. This data is returned to the DCache and register file. If the fill data for the first LD miss arrives at the same time, we would have a situation where the DCache block is filled with data from two different blocks. To avoid this, the TROLL compare is performed on the DCache index. To ease implementation when addresses hit the BAF in 64B or 32B modes, trolling is done on bits <14:6>.

A write request from the Mbox to a private-clean or shared block in the SCache needs permission from the System Interface unit. The DCache index of this write request is also entered in the TROLL register so that subsequent reads and writes to the same index are blocked until after the permission for the first write has been granted by the system.

5.2.1.3 BIU requests

BIU requests have highest priority for SCache access. For the data movement commands (Fills or Updates), BIU alerts the Scache arbiter in s4a of the pipe for the corresponding RFB cycle in 9b. This helps the Arbiter to assert the SC_BUSY signal to the Mbox to free up the SCache and the other buses in time for the BIU requests.

Table 5–7: Commands from BIU for SCache access

Command	Description
NOP	No Operation
FILL_TAG	Fills from memory. Update Tag and Status in Scache
FILL_TAG_STATUS	Fills from memory. Update only Tag Status
FILL_NOP	Reserve an RFB slot for fills to Dcache
SH_UPDATE	Set/Clear Shared bit in Scache
INVAL	Invalidate block if present in Scache
RD_DIRTY	Read Dirty or Flush from System
READ_VICT	Read Victim

5.2.1.3.1 BIU request Prioritization at SAU

BIU provides 3 request lines to the SAU in s4a. The SAU prioritizes these requests and arbitrates for the SCache. A grant signal is sent to the request that won the arbitration. SC_BUSY is appropriately asserted in s5 and s6 to hold off SCache tag accesses from the Mbox. The grant signal also enables the address read from the appropriate address register of the request source (BAF, VAF or the SPA). This address is latched and driven to the SCache tags in s6b by the SAU.

VAF and SPA grants are aborted if the requests are followed by a FILL in the next cycle.

1. Request from BAF (Fills and Fill updates from Memory)
2. Request from VAF (Victim Reads)

3. Request from SPA (System Probe Requests)

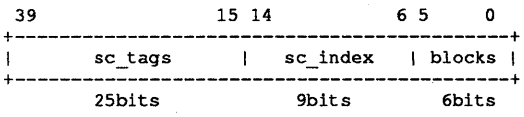
Requests for fills originate from the BCache Sequencer (BSQ) and BAF controllers. Addresses to SCache lookup are driven from the BAF or the SBR register. For updates from the system, the fill address is stored in the System BCache Register (SBR). Addresses for fills from the BCache are stored in the BAF registers. The grant signal from SAU is used to read the appropriate address from these registers.

Requests for victim reads are sent from the VAF controller. Addresses for the victim reads are stored in the VAF register. The SAU provides the appropriate set number to the SCache and forces a hit. The SCache drives out the tags and data which are accumulated in the VAF and Victim Data Buffer by the VAF controller.

System probe requests are initiated by the SPA (System Probe Arbiter) controller. The SAU provides appropriate SCache block status bits and the SCache hit information to the SPA. Information on VAF address hits are forwarded to SPA by the VAF controller.

5.2.1.4 SCache Set Allocation

The SCache is a 96KB, 3-way set associative, write-back on-chip secondary cache. The tag, index and block bits of the physical address are as follows.



The SCache Tag Store sends the following block status signal to the Set Allocation Section.

- Block Valid Bit from all sets
- Block Dirty Bit from all sets
- Block Shared Bit from all sets
- Bcache Index Match from all sets
- Tag parity from all sets
- Hit signal from all sets
- 2 Sub-block Modified Bits
- 25 bits of Tag from the set that was hit

A modified round robin scheme is used for set allocation on an SCache Miss.

Figure 5–8: Set Allocation Algorithm

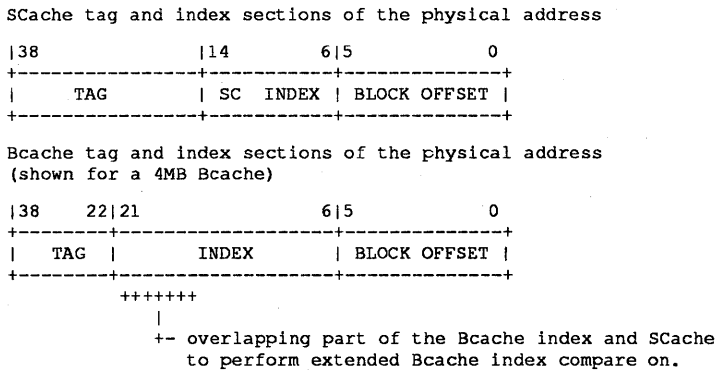
```
if (bcache index match) {
   set allocated = bcache index matched set.
}
else {
   set allocated = round-robin
}
```

5.2.1.4.1 Bcache Index Match

The SCache is a 3-way set associative 96KB cache and the Bcache is a direct mapped cache. If a miss occurs in the SCache, the Bcache is read . This read may miss in the Bcache and the Bcache miss may produce a victim. (i.e. the evicted block is dirty and has to be written out to main memory) Because the SCache is a write back cache, a "dirtier" version of this victim may be in any of the 3 sets in the SCache. The SCache copy of the victim is most up to date and it is this copy of the block that we should write to memory, not the Bcache copy.

We can prevent this from happening by doing some extra work when we read the SCache. Since we know the size of the Bcache, we can create a "Bcache victim might be old" signal by comparing the part of the Bcache index that overlaps with the SCache tag. If there is a match we know that one of the three blocks in the SCache maps to the block in the Bcache that we are going to read to fix the SCache miss. Further more, since we know that this is the block that we will read from the Bcache and it's not the block the SCache wants, we know the Bcache will miss. (If there was a Bcache index match, and it hit in the Bcache, it would mean that the tag portion also matched. This would imply that it would have hit in the SCache to begin with. We are hence guaranteed a miss in the Bcache.) Thus, if there is an extended Bcache index match and the block is dirty, the thing to do is to force the block in the SCache into the Bcache and then do the read. The Bcache extended index matching is shown in Figure 5–9

Figure 5–9: Bcache Index match



Thus, if the Bcache index matched set is dirty, then a victim read must be scheduled for the set, BEFORE any off-chip transaction for the miss can be dispatched. The BIU initiates this Read Victim transaction by driving the SCache index bits and victim set number to the SCache tag store through the SCache Arbiter. One victim read is mandatory to clear the dirty bit in the SCache. The SCache tag store drives the two INT32 Modified bits during this first victim read.

Since a miss in the SCache, on an extended Bcache index match, is guaranteed to miss in the Bcache, the fill data returned will overwrite the indexed location in the Bcache and in the SCache. Therefore, even if the data is not dirty in the SCache, it has to be invalidated to "make room" for the fill data. Thus the Set Allocation Logic checks for a Bcache index match in any one of the 3 sets. If there is a match, that set is allocated for refills.

Mode bits will be required to perform the Bcache index compare on the appropriate range of bits for different Bcache sizes.

5.2.1.4.2 Fills from Scache to I/DCache

Figure 5–10: I/DREAD hits in the SCache

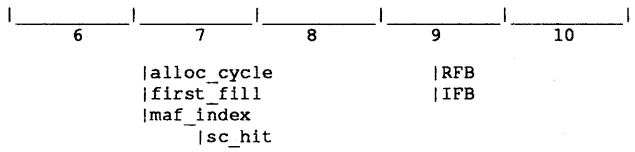


Figure 5–11: DREAD fills from external memory (Non-error mode)

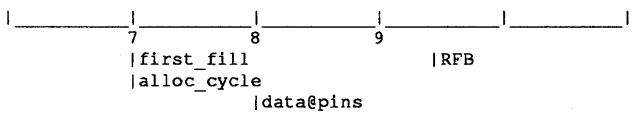


Figure 5–12: IREAD fills from external memory

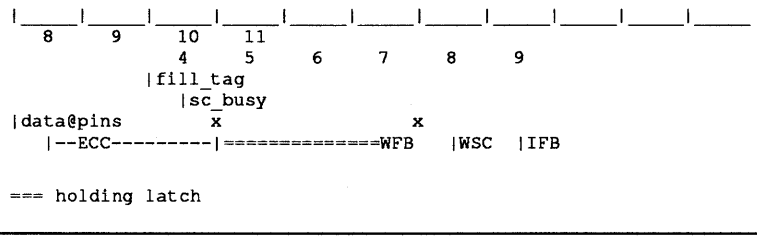


Figure 5–13: SCache Arbitration under fills

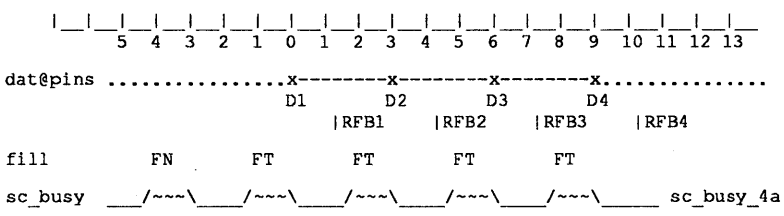
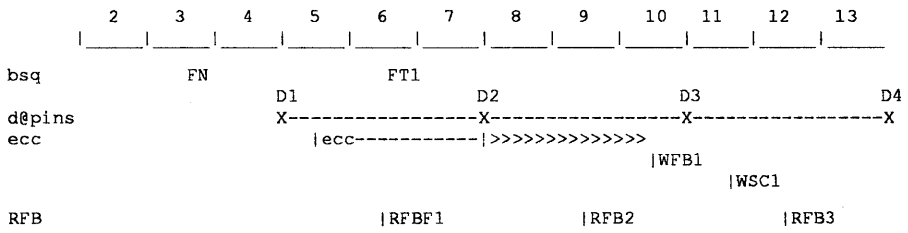
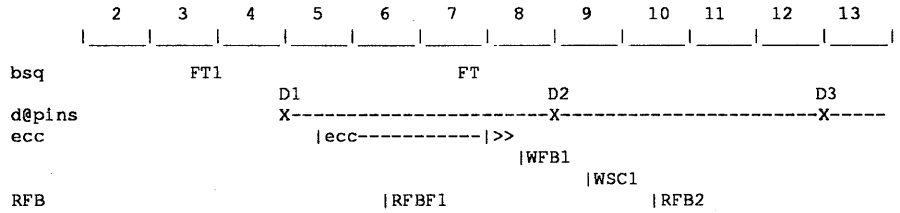


Figure 5–14: SCache Dstream (non-error mode) Fill Flow (3 cycle sysclock)



key: WSC1 = D1 gets written into Scache
RFB1 = D1 gets sent to Dcache
FT1 = fill request from BSQ to write D1 into Scache
FN = fill request from BSQ to free up RFB2 slot
>>>> = Holding latch

Figure 5–15: SCache Dstream (non-error mode) Fill Flow (4 cycle sysclock)



key: WSC1 = D1 gets written into Scache
RFB1 = D1 gets sent to Dcache
FT1 = fill request from BSQ to write D1 into Scache
>>>> = Holding latch

Figure 5–16: SCache Dstream (non-error mode) Fill Flow (5 cycle sysclock)

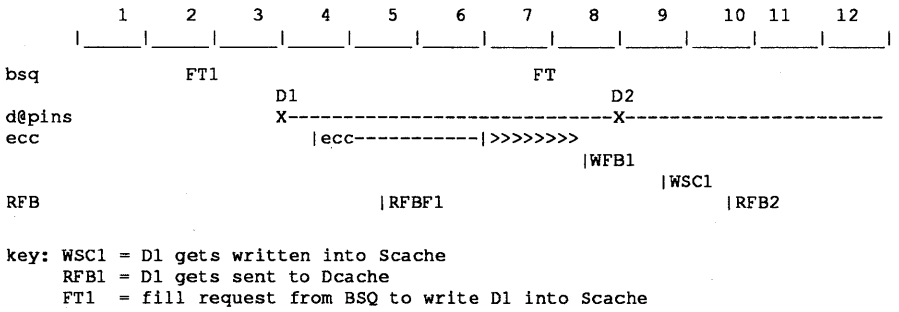


Figure 5–17: SCache Dstream (Error mode) Fill Flow (5 cycle sysclock)

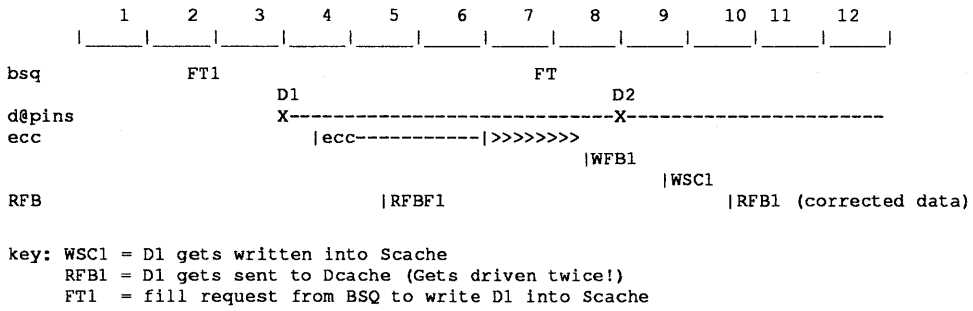


Figure 5–18: SCache Istream (non-error mode) Fill Flow (5 cycle sysclock)

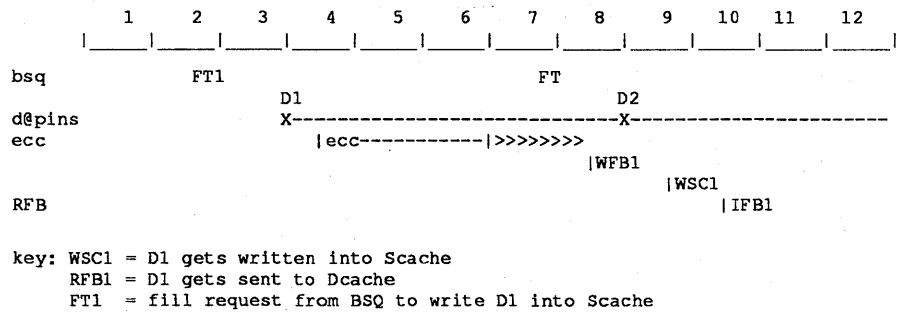


Figure 5–19: Scache Read Hits Under Fills (3 cycle sysclock)

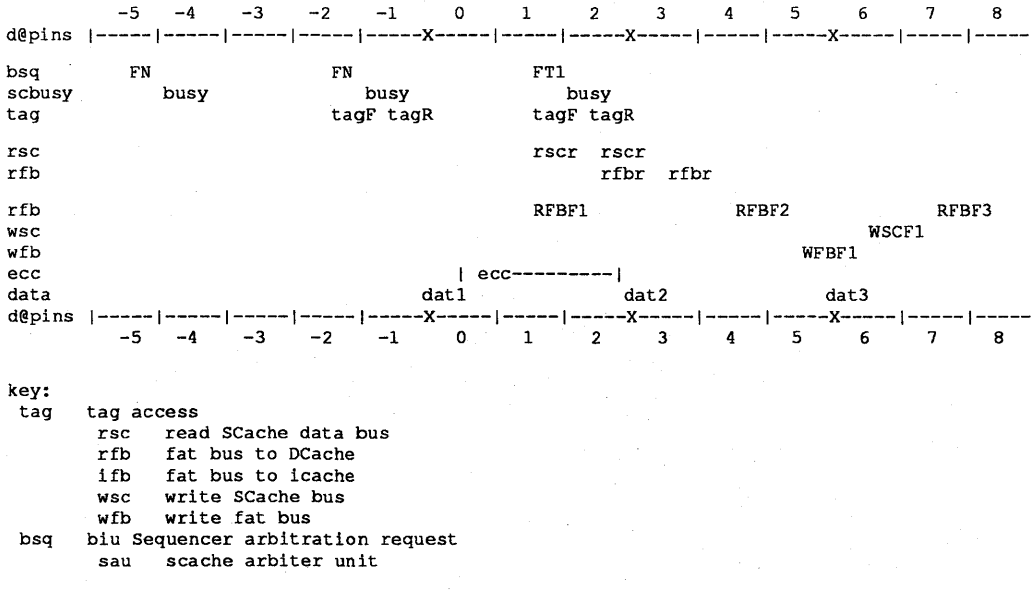
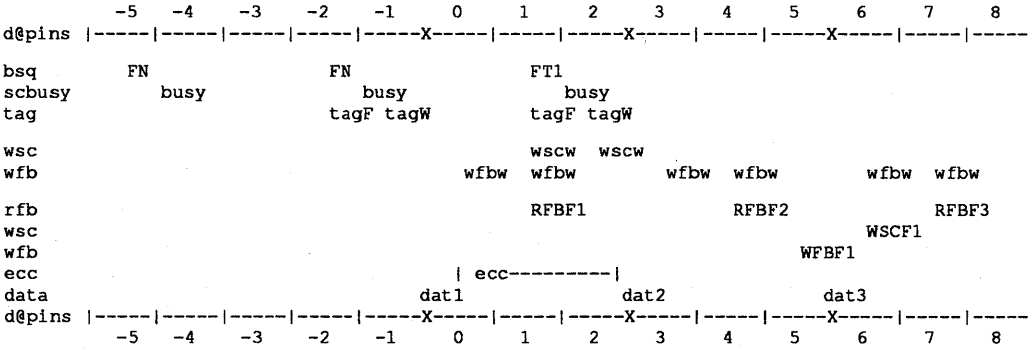


Figure 5–20: Scache Write Hits Under Fills (3 cycle sysclock)




```
key:
 tag   tag access
        rsc   read SCache data bus
        rfb   fat bus to DCache
        ifb   fat bus to icache
        wsc   write SCache bus
        wfb   write fat bus
 bsq   biu Sequencer arbitration request
        sau   scache arbiter unit
```

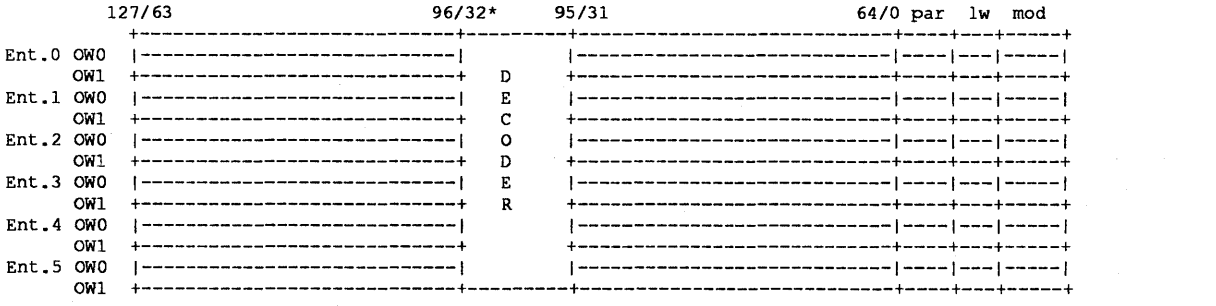
5.2.2 Write Buffer Unit

The Write Buffer Unit (**WBU**) contains 6 entries, each of 32B blocks. The Write Buffer Data Store (**WBD**) is in the CBOX and the write address file (**WAF**) is in the MBOX. Stores are allocated a new entry in the **WBU** unless they can merge to an existing entry. The **WAF** is responsible for merging stores into the write buffer and the issue of the writes to the CBOX. The write buffer completion control (**WCC**) in the CBOX is responsible for the completion of the writes. The Write reissue control (**WRC**) is responsible for the reissue of writes by the MBOX. Every write transaction issued to the CBOX is a 32B write. If EV5 is in 64B mode, writes to the system are performed by first populating the 64B block by performing a second "wr_for_populate".

5.2.2.1 Write Buffer Data Store: WBD

Each entry in the **WBD** is organized as 2 rows of octawords (16B) for a total height of 12 rows as seen in Figure 5–21. The store data is valid from the EBOX/FBOX in cycle 6 and the data is piped to be written into the **WBD** in cycle 7. The delay is required in order to allow the **WAF** to perform merge/conflict calculation on the incoming store addresses and for TRAP calculation. It is assumed that the longword parity bits are provided along with the store data by the MBOX. LW valid bits for each octaword and octaword modified bits for each entry are also stored in the **WBD**.

Figure 5–21: Write Buffer Data Store



* In the layout, bit <127>/<63> thro <96>/<32> are mirrored so that <127>/<63> sits closer to the center.

5.2.2.2 Storing Data in write buffer

When a STx is received by the MBOX, they write the DCache (if the block is present) and load the appropriate entry in the **WAF**. Data is written into the **WBD** in cycle 7a. Physical address bits **M%C_WR_LW_ADDR_5B_H**<4:2> are sent to the MBOX in 5b along with the signal **M%C_WR_TYPE_5B_H** (0=QW or 1=LW). The address and type are decoded as shown in Table 5–8. Write enables are sent in 6a to enable the write in 7a. *Data being written cannot be bypassed and issued to the CBOX.*

Table 5–8: Wr Decode

wr_lw_addr<4:2>	wr_type=0 (qw)	wr_type=1 (lw)
000	wr <63:0>	wr <31:0>
001	wr <63:0>	wr <63:32>
010	wr <127:64>	wr <95:64>
011	wr <127:64>	wr <127:96>
100	wr <191:128>	wr <159:128>
101	wr <191:128>	wr <191:160>
110	wr <255:192>	wr <223:192>
111	wr <255:192>	wr <255:224>

5.2.2.3 Issue of Writes

The MBOX will determine when to issue a write to the CBOX. (For further detail please see the MBOX chapter of the spec.) When the **WAF** issues a write it stops merging to that entry. The MBOX sends the signals **M%C_MAF_INDEX_5B_H**<4:0>, (<4> is asserted for WB indices with <2:0> indicating which entry), **M%C_MAF_CMD_5B_H**<3:0> and address in 5b/6a. It is assumed that the MBOX asserts the maf_cmd wires only in the first cycle of the 2 cycle write. The **WBD** is read in 6b and in 7b and driven onto the **WFB** (fat bus) in 7B/8A and in 8b/9a. *The lower 16B is ALWAYS read out in 6b followed by the upper 16B,regardless of which half of the 32B of data has valid longwords written*

Bad LW parity for the WB data can be forced by asserting the appropriate bit in the SC_CTL ipr. If a write completes successfully and updates data in the SCache, the MBOX is sent a "wr_done" or "stxc_done" return status in 8a accompanied by the maf_index of the write. The lw_valid bits and the ow_modified bits of the corresponding entry in the **WBD** are cleared in 9a. Data and parity are not cleared.It is assumed that **WBD** will not get write enables for writes that have already been issued by the MBOX. (ie. we will get a case where we are clearing an entry while we are writing into it)

NOTE

M%C_MAF_ABORT_6A_H will never be asserted for writes since writes are never bypassed by the MBOX.

The basic write transaction is shown in Figure 5–22 and the data write and data issue timing diagrams are shown in Figure 5–23 and Figure 5–24.

Figure 5–22: Write Flow

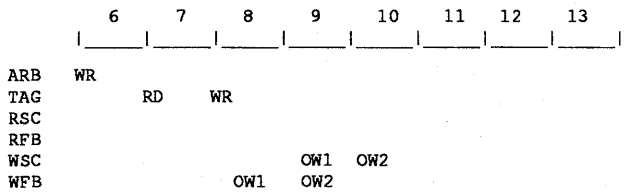
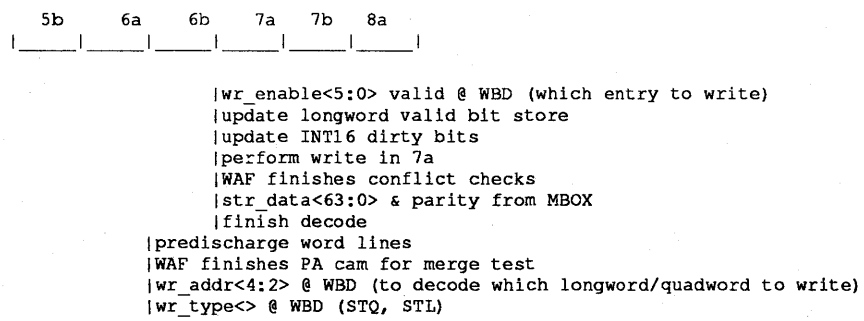


Figure 5–23: Write buffer data write timing diagram

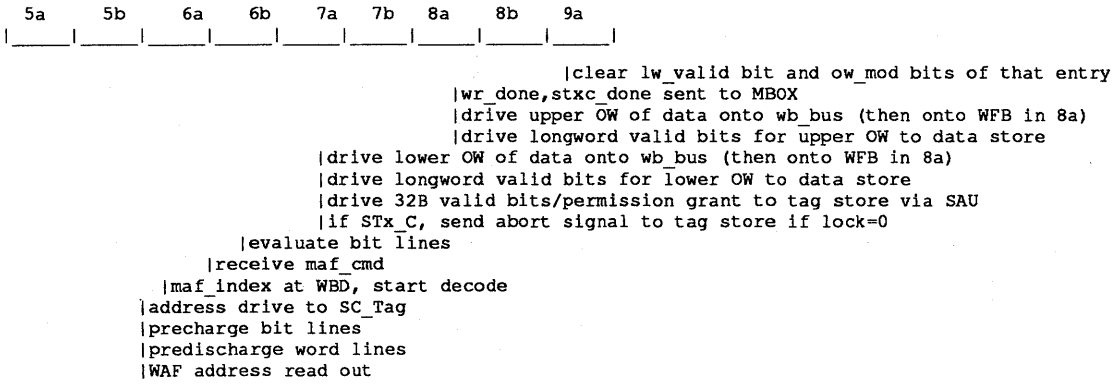


5.2.2.4 Write Buffer Completion Control:WCC

The Write buffer Completion Control (WCC) is responsible for determining whether a write can successfully complete. Writes can successfully complete if they satisfy the conditions shown below:

- Cacheable References
  - hits the SCache with private/dirty status
  - hits the SCache with private/clean status and permission to set the dirty bit
  - hits the SCache with shared status and has a shared permission grant
  - the SCache is operating on force hit mode

Figure 5–24: Write buffer data issue timing diagram



- Non-Cacheable References
  - the BIU has enough resources (one BAF & one VAF entry) to store the data

If any of the above conditions are satisfied, "wr_done" is returned as status to the MBOX. If any of the above conditions are not satisfied, the write is loaded into a reissue queue and reissued later to the MBOX. This is discussed in Section 5.2.2.5.

STx_C's can successfully complete if they satisfy the conditions shown below:

- Cacheable References
  - lock is clear & hits the SCache regardless of status (fails)
  - lock is set & hits the SCache with private dirty status (succeeds)
  - lock is set & hits the SCache with private/clean status and permission to set the dirty bit (succeeds)
  - lock is set & hits the SCache with shared status and has a shared permission grant (succeeds)
  - lock is clear but system CACK'd a shared write. (succeeds)
- Non-Cacheable References
  - Ignores local lock flag. Completes after systems has CACK'd/CFAIL'd the STx_C.

If any of the above conditions are not satisfied, the STx_C is loaded into the reissue queue and reissued later to the MBOX. A more detailed table of the STx_C flow is shown in Table 5–10 and Table 5–11.

NOTE

STx_C should not be issued under force hit mode.

5.2.2.5 Write Reissue Queue and Control : WRQ,WRC

Writes from the MBOX are stored in a reissue queue if they do not complete successfully. There are 2 reissue queues, one for 32B/64B mode (called the write queue) and another used specifically for 64B mode (called the read queue). The write queue is a 2 deep linear queue that holds:

- cacheable writes that miss in the SCache
- writes that hit private/clean data which requires system permission to set the dirty bit
- STx_C's that hit private/clean data with the lock flag set
- writes that hit shared data which has to be broadcast on the bus before it can be written into the SCache
- STx_C's that hit shared data with the lock flag set
- STx_C's to I/O space which ignore local lock flag and have to get acknowledged from the system before proceeding.

Each entry in the reissue queue consists of a valid bit (set when it is loaded), a reissue bit(set when the write is ready to reissue) and a permission bit(set when the write has permission to update the SCache). It also holds other information such as the maf_index, shared,dirty,cmd and address bit<39>. Requests are sent to the system by the BIU to process the writes in the reissue queue (SET_DIRTY,WRITE_BLOCK,WRITE_BLOCK_LOCK & READ_MISS_MODIFY). The reissue bit for an entry is set when the following conditions are satisfied:

- For misses, when the fill completes , the BIU asserts **C_BAF%FILL_DONE_4A_H** along with **C_BAF%SC_MAF_IDX_6A_H** which sets the reissue bit for that entry.
- When an CACK is received, for private/clean writes ,shared writes and STx_C to I/O space, the BIU will assert **C_BIU%ACK_9A_H** along with **C_BAF%SC_MAF_IDX_6A_H** which sets the reissue bit and permission bit for that entry.
- When a CFAIL is received, for shared writes, and STx_C to I/O space, the BIU asserts **C_BIU%NOACK_9A_H** along with **C_BAF%SC_MAF_IDX_6A_H** which sets the reissue bit for that entry.
- If a pending write got invalidated, the BIU asserts **C_BIU%WR_INVAL_9A_H** along with **C_BAF%SC_MAF_IDX_6A_H** which sets the reissue bit for that entry.

If the entry that got acknowledged (reissue bit set) is at the top of the reissue queue, the write buffer reissues the write to the MBOX by asserting **C%M_WR_NOW_4A_H** and **C%M_WR_MAF_INDEX_4A_H**<2:0>. Once this signal is asserted, in the best case, the MBOX reissues the write in the very next cycle (cycle 5). In the worst case, if they had already arbed for an instruction in cycle 4, then the reissued write will issue only in cycle 6. Since the write queue is a linear queue, writes are reissued to the MBOX in the original order that they were issued. Only one reissue can be performed at a time. If a write has been reissued to the MBOX, the next reissue will not be sent until the first one does not get retried. Reissued writes take highest priority in the MBOX to ensure forward progress of writes. Writes are not guaranteed to complete in the order that they were originally issued by the MBOX.

In 64B mode ONLY, writes that hit shared data are loaded into the read queue in order to accumulate the second half of the 64B data to broadcast onto the bus. This write will be reissued to the MBOX , along with a request to send the the address of the other half of the 64B block to the SCache by asserting the **C%M_WR_64B_REQ_4A_H** signal.

The write and read queues are loaded in cycle 9 for a tag cycle access in cycle 6. This implies that any acknowledgment from the BIU must not arrive earlier than cycle 10. Entries in the read queue take higher priority over those in the write queue. When a write is reissued to the MBOX, (either from the read queue or from the write queue), the maf_index and other pertinent information are loaded into a reissue_register. If an entry in this register is valid, no other write can reissue until this register is cleared. The register is cleared in 8B when the reissued write is issued by the MBOX and is not retried.

NOTE

- Reissued STx_C's are issued by the MBOX as a WR cmd. The reissue logic keeps track of the STx_C's and asserts **C_WBU%STXC_CMD_6B_H** to indicate that the instruction that is being issued is a STx_C and not a write command.
- STx_C is **NOT** allowed to a CBOX IPR.

5.2.2.5.1 Stopping Writes

To prevent a "deadlock" situation from arising, the CBOX will retry all new writes from the MBOX when it gets into a situation when it cannot reissue a write even though it may be ready to reissue. This situation will arise when

- entries in the write reissue queue are acknowledged out of order
- There is a pending reissue when another reissue queue entry has been acknowledged.

In other words , the situations where this can occur are

- when we have 2 valid entries in the write reissue queue, both missing in the SCache. If the first one misses in the Bcache and the second one hits in the Bcache, the second entry in the reissue queue is ready to reissue before the first one. The second entry cannot issue until the first entry has its fill completed from memory.
- Whenever a fill completes/ACK arrives just as we are are issuing a write to populate a shared write in 64B mode. The write cannot issue until the write_for_populate completes.

Whenever we get into the situations described above, all subsequent (new) writes from the MBOX will be retried until the "stop_write" scenario is resolved. Both the scenarios are likely to happen very infrequently and its impact to chip performance is expected to be negligible. The stop_write scenario resolves the deadlock situation described below:

Currently, in the CBOX , writes that miss in the SCache are loaded into the WB reissue queue and into the BIU Address File (BAF). The BAF entry acts as the "troll" register and retrys any subsequent SCache access that has the same Dcache index to prevent interleaved accesses to the same cache location. Take for example that a STR, STR A, missed in the SCache and is loaded into the BAF. Another STR , STR B, to the same SCache index as STR A, gets retried until the fill for STR A completes.

In order to facilitate streaming writes on the pins, the BAF entry is cleared a few cycles after the fill actually completes. For the most part, the clearing of the BAF entry is synchronized with the reissue of STR A, (for which the fill completed) by the MBOX. In certain situations,as explained above the clearing of the BAF may occur before STR A can be reissued to the MBOX. As a result, the BAF entry is cleared prematurely in these cases and TROLLing on STR A's index is stopped.

STR B, to the same SCache index as STR A, could then sneak in before the original write is reissued by the MBOX.

A deadlock situation can arise if STR A and STR B have the same Bcache index and both miss in the SCache. If STR A and STR B have the same Bcache index and the Bcache is direct mapped, STR A and STR B cannot simultaneously exist in the SCache. So in the above example, if STR B sneaks in before STR A is reissued , STR B would miss in the SCache and evict STR A from the SCache. The reissued STR A, meanwhile is continuously being retried since it trolls on the STR B BAF entry. Eventually when the fill for STR B completes, the BAF entry is cleared and STR A proceeds. STR A misses in the SCache and evicts STR B from the SCache. The process continues infinitely with both the STR's swapping each other out indefinitely.

Part of the deadlock problem arises from the implementation of the reissue queue in the CBOX as a linear queue where writes are reissued to the MBOX only in the order that they were originally received. Considering the fact that the above cases occur infrequently, changing the reissue queue structure from a linear queue structure where only the top of the queue can reissue, to a structure where any entry can reissue may not be a big win. In fact changing the queue structure does not solve all cases, unless the MBOX redesigns their logic to accept multiple reissues. Making the reissue queue non-linear also complicates,the write buffer reissue control. The safest solution would be to clear the BAF entry only when the reissued write actually completes. However , this solution has a big impact when we are streaming writes off chip. Bus bandwidth would drop from 4.26GB/sec (64B mode) to 3.55GB/sec. In 32B mode, the bandwidth would drop from 3.55GB to 3.01GB/sec. Hence this solution was abandoned and the stop_write widget was introduced.

5.2.2.5.2 Stopping Reads

Whenever an CACK is received from the system for either a shared write or a SET DIRTY, and the write cannot immediately reissue (because of a pending reissue), MBOX reads are stopped. This is done in order to prevent the block ,for which an CACK was rceived from the system , from being swapped out of the SCache until the block status has been updated.
To prevent a deadlock situation from occurring, we also stop stop reads whenever any reissued writes which possibly hits shared data is followed by another write which is ready to issue (for which a fill completed) and needs a BAF entry. The deadlock is shown below

- Reissue WR 1 : hits shared : allocate BAF entry, wait for wr for populate
- MBOX read miss : allocate other BAF entry
- Reissue WR 2 : misses or hits shared or p/c => needs BAF entry , but none available, retries
- BAF unload ptr, still waiting for wr_for populate before servicing shared write. It cannot serice the read_miss since it services requests in order
- => Deadlock : BAF waits for write for populate which cant issue until Reissue WR 2 completes which cant until a BAF entry clears which cant until the Shared write is serviced which cant until BAF receives write for populate etc.

The deadlock was fixed by detecting if there is a pending reissue to a block that originally missed followed by another write, ready to issue, that originally missed or got invalidated. If such a case occurs, then MBOX reads will be retried.

5.2.2.6 Write flows

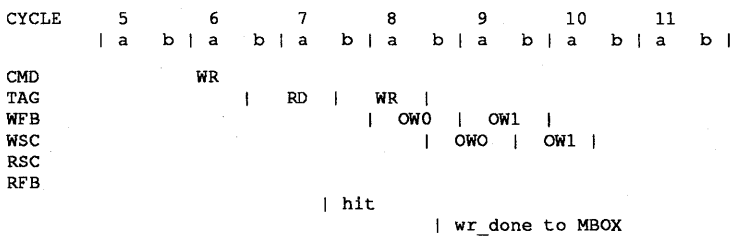
Data in the SCache can exist in one of four possible states

- private/clean
- private/dirty
- shared/clean
- shared/dirty
- invalid

5.2.2.6.1 Private & Dirty

Writes can proceed without system intervention only if the write is to a block that is in the private and dirty state. This is depicted in Figure 5–25.

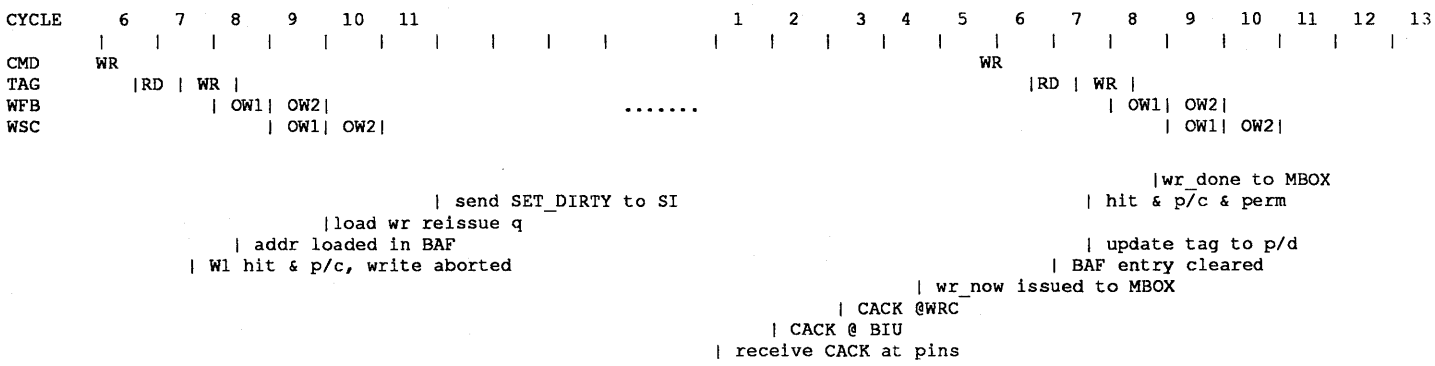
Figure 5–25: Write hit private/dirty



5.2.2.6.2 Private & Clean

If the block is private/clean, before the SCache and Bcache tag status can be changed to private/dirty, the duplicate tag store in the System Interface (SI) must be updated. So the write is loaded into the "write" reissue queue and a SET_DIRTY request is sent to the SI to request permission to write this block. The SI processes the request by setting the dirty bit, for the block, in its duplicate tag store and CACK's the request. (On a uniprocessor system, EV5 works in auto-ACK mode where an external CACK will not be required) On receipt of the CACK, the WRC reissues the write at high priority by asserting the C%M_WR_NOW_4A_H signal along with the C%M_WR_MAF_INDEX_4A_H<2:0>. The write is then re-issued but this time the WRC asserts the signal C%S_WR_DIRTY_PERM_6B_H to the SCache tag store,which sets the dirty bit and proceeds with the write. The flow is depicted in Figure 5–26.

Figure 5–26: Write hit private/clean



In the boundary case, EV5 could receive a SET_SHARED(INVALIDATE,FLUSH) to the same block, before the BIU has had a chance to issue the SET_DIRTY cmd. The shared bit of the block would then be set in the SCache tag store. The BIU will NOT issue the SET_DIRTY command. It will service the SET_SHARED(INVALIDATE,FLUSH) request and then assert **C_BIU->WR_INVAL_9A_H** along with **C_BAF%SC_MAF_IDX_6A_H** to the **WRC**. The write is then reissued which will hit shared data (miss)in the SCache and the shared write (write miss) flow will be initiated. If EV5 had just issued the SET_DIRTY command as it received the SET_SHARED (INVALIDATE,FLUSH), it is assumed that the SI will ignore the SET_DIRTY, since it was to the same block that the SET_SHARED was sent for. The BIU will service the request and proceed as described above.

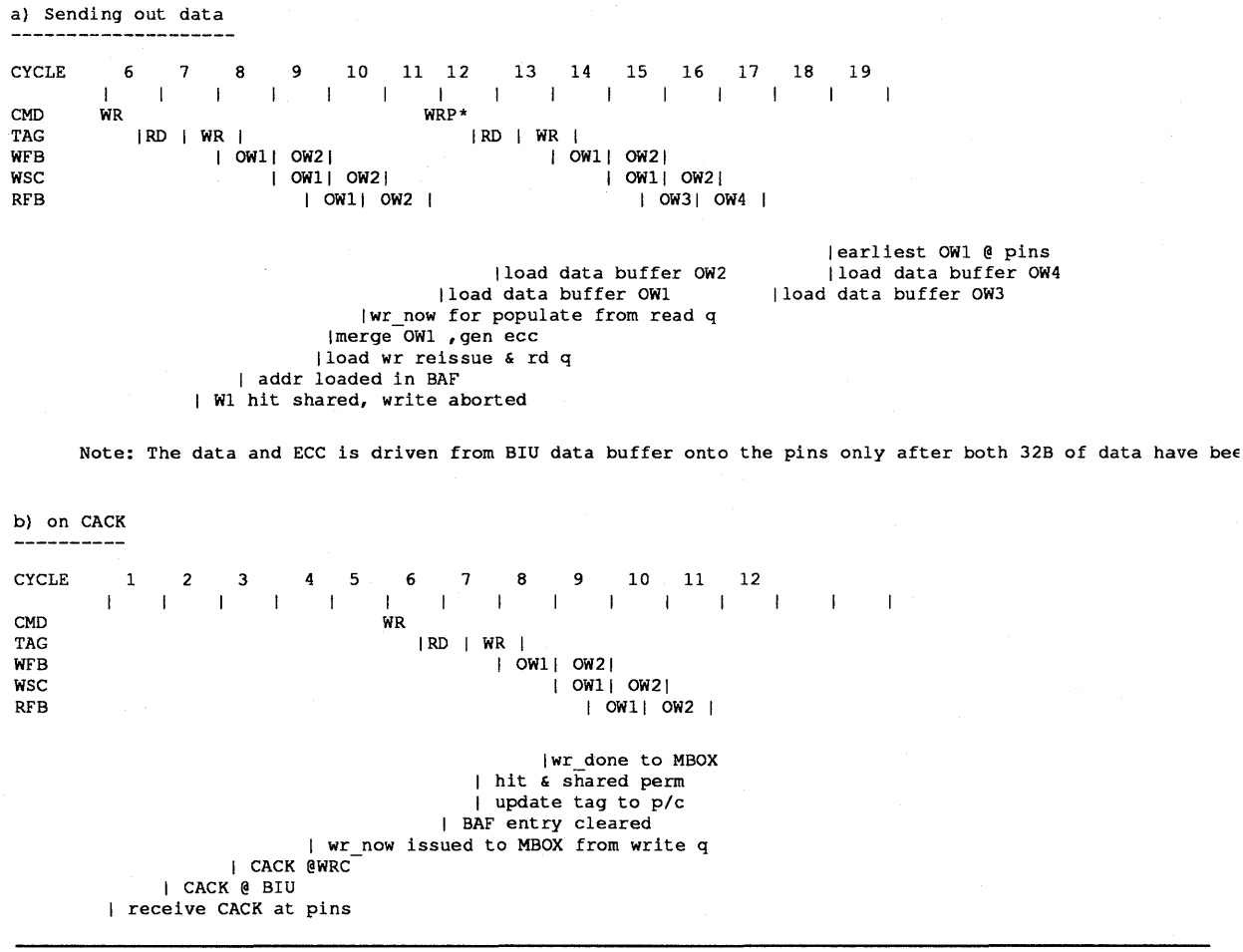
The other boundary case is when EV5 receives a READ_DIRTY request to a block that has just been CACK'd for a SET_DIRTY request. The READ_DIRTY could sneak in and set the shared bit of the block before the write has been re-issued with permission to set the dirty bit. This would cause incoherency in the tag stores. To prevent this from happening, every time EV5 receives an CACK, it will not process further system requests until the dirty bit has been set. Therefore on an CACK, the signal **C_WBU%STOP_SPA_4A_H** is asserted to stop servicing system requests until the reissued write sets the dirty bit. If the reissued write can not go out immediately, MBOX requests are then retried by asserting **C_WBU%STOP_WRITES_6B_H** and **C_WBU%STOP_READS_6B_H**to ensure that the block for which the CACK arrived does not get evicted from the SCache until the dirty bit is set.

5.2.2.6.3 Shared & Clean

Shared writes or "broadcasts" onto the bus require a fully populated block. Therefore, in 32B mode, if the block is shared, the 32B block is first populated by reading the SCache and merging it with the written longwords from the write buffer. (the SCache is not written at this stage) ECC is then generated on the merged data and loaded into the BIU system buffer. The write is also loaded into the "write" reissue queue. Then a WRITE_BLOCK request is sent to the SI, along with the two octawords of data. The Bcache is written with the data as it is passed to the SI. When an ACK is received, the **WRC** reissues the write. The **C%S_WR_SHARED_PERM_6B_H** signal is asserted to the SCache which then updates the SCache tag and data stores. The state of the block transitions to private/clean since the WRITE_BLOCK invalidates every other copy of the block in the system.

If EV5 is operating in 64B mode, the first 32B is populated as explained above and the data loaded into the BIU data buffer. The write is also loaded into both the read reissue queue and write reissue queue. The entry in the read reissue queue issues to the MBOX asap. The **C%M_WR_NOW_4A_H** signal is asserted to the MBOX along with a **C%M_WR_64B_REQ_4A_H** signal since the reissue is coming from the read queue. This forces the MBOX to flip bit<5> of the address to access the other 32B half of the 64B block. Typically it will take about 6 cycles from when we see the first write from the MBOX to when the MBOX will send the second write to populate the 64B block. The second write is referred to as the "write_for_populate" The data for this half is read out from the SCache and is loaded into the BIU data buffer. Having accumulated all the data, the BIU then processes the transaction to the SI as explained for the 32B flow. On receipt of the CACK, the **WRC** reissues the write and the SCache is updated. The broadcast transaction for 64B mode is shown. is shown in Figure 5–27

Figure 5–27: Write broadcast



The boundary cases are similar to the private/clean case, If EV5 receives an INVALIDATE(FLUSH) to the same block, before the BIU has had a chance to issue the WRITE_BLOCK cmd, the BIU services the INVALIDATE and then asserts C_BIU%WR_INVAL_9A_H signal along with C_BAF%SC_MAF_IDX_6A_H to the WRC. The write is then reissued which will miss the SCache and the write miss flow will be initiated. In 64B mode, the INVALIDATE could occur between the original write and the write_for_populate. In this case, the BIU will not service the INVALIDATE until the wr_for_populate has gone through. After it has, it will service the INVALIDATE and then ,like described above,assert C_BIU%WR_INVAL_9A_H (Previously the INVALIDATE would be serviced first and then the write_for_populate will miss in the SCache and be ignored, since it is not a "real" write)

Again, to prevent incoherency in the system, every time EV5 receives an CACK, it will not process further system requests until the SCache has been updated and the state of the tag store changed to private/clean. The signal C_WBU%STOP_SPA_4A_H is asserted until the reissued write updates the tag store. If the reissued write can not go out immediately, MBOX requests are then retried by asserting C_WBU%STOP_WRITES_6B_H and C_WBU%STOP_READS_6B_H. This is done to ensure that the block for which the CACK arrived does not get evicted from the SCache.

As can be seen in Figure 5–27, in 32B mode, the earliest EV5 can provide data & ecc at the pins for a tag access beginning in cycle 6, is at cycle 13. In 64B mode, the earliest EV5 can provide data & ecc at the pins is cycle 19.

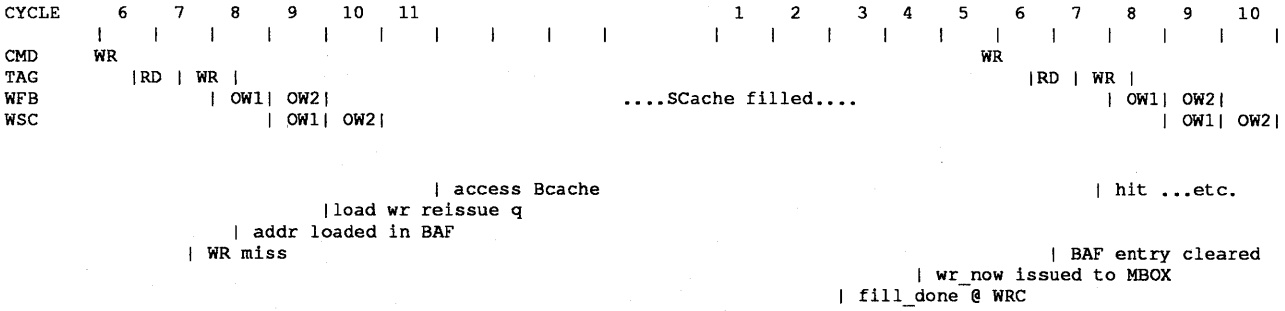
5.2.2.6.4 Shared & Dirty

For a shared dirty block, the transaction is identical to the shared/clean case except that the dirty bit is also cleared. It is assumed that the SI will clear the dirty bit in the backmap on receipt of the WRITE_BLOCK request from EV5.

5.2.2.6.5 Write misses/Invalid

If the write misses in the SCache or the write hits an invalid block in the SCache, the write miss flow is initiated. The block is fetched from the Bcache if it is present there, else it is fetched from memory if it is not present in the Bcache using the READ_MISS_MODIFY command. The write is loaded into the "write" reissue queue Once the fill completes into the SCache, the BIU asserts the C_BAF%FILL_DONE_4A_H signal and the WRC reissues the write at high priority by asserting the C%M_WR_NOW_4A_H signal along with the C%M_WR_MAF_INDEX_4A_H<2:0>. The write is then re-issued and depending on the status of the fill data, the appropriate flow is initiated. The write miss flow is depicted in Figure 5–28.

Figure 5–28: Write miss



5.2.2.6.6 I/O writes & non-cacheable writes

I/O writes and non-cacheable writes are writes to addresses with bit<39> of the address set. The SCache is not written and data is driven from the write fat bus directly into the BIU system buffer. The longword valid bits are provided to the BIU to indicate which longwords are valid in each octaword. (for longword granularity in I/O space). Regardless of which mode EV5 is in, I/O writes are always only 32B writes. Table 5–9 shows the cases when writes succeed for all combinations of permission grants/tag status.

Table 5–9: Writes with Permission grant

c%s_wr_shared_perm_6b_h	c%s_dirty_perm_6b_h	shared bit	dirty bit	write SCache
0	0	0	0	No
		0	1	Yes
		1	0	No
		1	1	No
0	1	0	0->1	Yes
		0	1	Yes
		1	0	No, rd SC
		1	1	No, rd SC
1	0	0	0	Yes
		0	1->0	Yes
		1->0	0	Yes
		1->0	1->0	Yes
1	1		ERROR	

5.2.2.7 General considerations for writes

- Up to two write requests can be queued up at the BIU but only one request can be pending at any time at the BIU to the system. Only after the first permission request (WRITE_BLOCK, WRITE_BLOCK_LOCK,SET_DIRTY) to the system is CACK'd will the second entry issue its request to the Bcache/system. The only exception is if the first request is a READ_MISS_ MODIFY cmd to the SI to fetch data from memory. In this case, the second entry can still access the Bcache underneath the first fill request.
- In general, write hits on DCache index matches are not allowed to be issued, and are replayed by the MBOX until the TROLL entry has been cleared. (If they were allowed, then we could have a situation where we get write hits to a set that has been allocated for a fill. If the write requires system permission, the fill data could return before write permission was granted thus changing the status of the tag.) The only exception to this rule is if the DCache index match is to the same Write buffer entry. This is to allow the second half of a 64B write to pass and to be merged at the BAF.
- Write buffer should not get write enables for MB's, WMB's and FETCH's that are store in the write buffer.
- It is assumed that if a reissued write gets retried, no further MBOX loads and stores will be serviced until the reissued write is allowed to pass.
- In general, if an instruction gets retried, the instruction in its shadow is treated indepedently. However, if the retried instruction is a reissued write, then the shadow is aborted and unconditionally retried. This is to ensure that no instructions in the shadow can access/displace the same block that the reissued write accesses.
- Write misses are *not* merged in the BIU.
- Writes are aborted on
  - failed STx_C instructions
  - troll matches except for write_for_populate
  - whenever the stop_write scenario occurs.

5.2.2.8 STx_C

STx_C are not merged with any existing entry in the **WBU** and are allocated separate entries in the **WBU** by the **WAF**. The **WBU** is flushed when it receives a STx_C. The lock flag is loaded on a fill with the logical AND of the local lock flag and the system lock register. When a write is issued the **WRC** checks the lock flag before issuing the data. If the lock flag is clear, an abort signal is sent to the SCache. If the lock flag is set, the success of the STx_C is dependent on the V,S,D tag status bits and the permission grant signals. The SAU is sent a **C_WBU%STXC_DONE_8A_H** signal which then returns STx_C_DONE as as return status to the MBOX. The lock flag is then cleared. A signal **C%M_STXC_ FAIL_7A_H** is returned along with the status, to the MBOX to be written into the appropriate register file location. Invalidates to the locked address that arrive at **EV5** clear the lock flag. As shown in tables Table 5–10 and Table 5–11, the following cases are possible.

Table 5–10: STx_C cases: Cacheable References

tag status	lock flag on first issue	action	lock flag on re-issue	action
private/dirty	1	Proceed with write. STx_C succeeds,clear lock flag	*	
	0	The write into the SCache is aborted. STx_C fails	*	
private/clean	0	STx_C fails	*	
	1	Send SET_DIRTY to SI. On CACK	1	Reissue STX_C. STx_C succeeds,clear lock flag.
			0	The write into the SCache is aborted. STx_C fails
		on CFAIL	*	SET_DIRTYs should not be CFAIL'd
shared	0	STx_C fails	*	
	1	Send WRITE_BLOCK to SI		Accumulate and send data out
		on CACK	*	Reissue STX_C. Local lock flag ignored since system accepted data. The STx_C succeeds. Clear lock flag
		on CFAIL	*	Should not occur. If it occurs,Reissue STX_C. STx_C fails. clear lock flag
miss/invalid	*			Fetch Data from memory and restart STX_C. Ignore local lock flag on the miss

Table 5–11: STx_C cases: Non-Cacheable References

tag status	lock flag on first issue	action	lock flag on reis-sue	action
-	*	Send WRITE_BLOCK_LOCK to SI		Accumulate and send data out
		on CACK	*	Reissue STX_C. STx_C succeeds. Clear lock flag
		on CFAIL	*	Reissue STX_C. STx_C fails

5.2.3 Bus Interface Unit

The cache system is made up of the Scache, the Scache victim buffers, and the Bcache. The Bcache is optional. The BIU will support only non-pipelined Bcaches of various sizes and access speeds. Some address support is provided for Bcache victims, but a Bcache victim buffer is not directly support by the BIU. The block size of Scache and Bcache can be either 32 bytes or 64 bytes, which is controlled by a mode bit located in the SC_CTL IPR register.

The BIU will service read miss, write miss, shared write, interlock, and victim requests from the Scache. It will manage the state of the cache system for the System, allowing the system to invalidate, flush, and read blocks in the cache system.

Figure 5–29 shows a simple picture of the system as seen by the BIU.

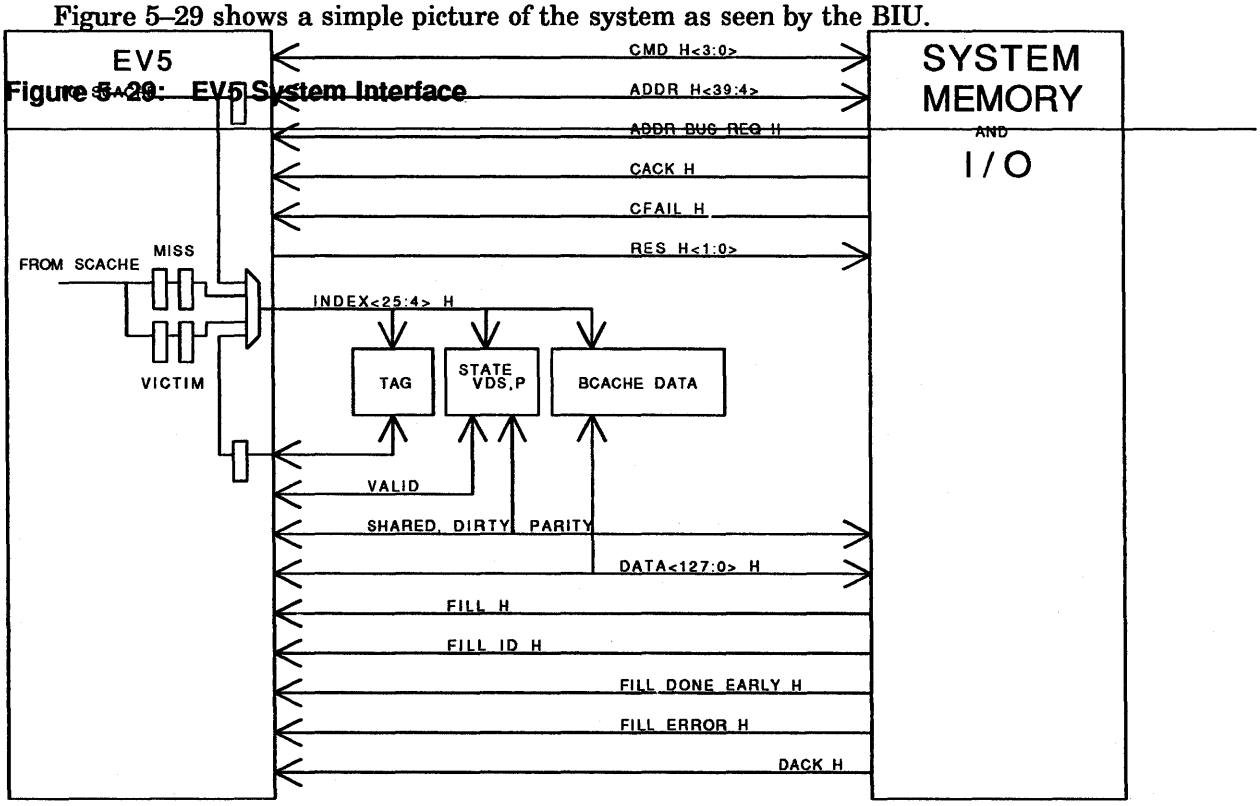


Figure 5–29: EV5 System Interface

5.2.3.1 BIU Functions

The BIU is made up of five parts and their control. Each is tightly integrated into the Scache.

The Lock Register and its control is used to maintain the state of the lock flag.

The System Probe Address Register (SPR) holds the probe address and command from the system. The SPR interacts with the Scache to perform the Scache Probe, and then interacts with the Bcache, provided one exists in the system.

The BIU miss Address File (BAF) holds the state and address for two requests from the CPU. The BIU Sequencer (BSQ) will access the Bcache and/or the system to satisfy these requests.

The Victim Address File (VAF) holds Scache victim addresses and state. It interacts with the Scache to remove the victims in a timely manner, with the WBU to buffer write data, and with the BSQ to write the victim data into the Bcache or System.

5.2.3.2 Lock Register

The lock register is loaded with the Scache address each time a LDx_L command is issued in the Scache. The lock flag is also set.

Each cycle the address in the lock register is compared to the system addresses that arrive. If an INVALIDATE or FLUSH is received to the cache block that is locked, the lock bit is cleared.

PAL code clears the lock flag by initiating a STx_C to the address in the lock register.

These signals are required for this function:

- Scache address
- lock flag

5.2.3.3 Scache Requests

This section outlines the interface between the Scache and the bus interface.

5.2.3.3.1 Loading the BAF and VAF

The address going into the Scache is compared to the addresses that are already in the BAF. If the cache system is in 64 byte mode, the address is the same 64B block, but a different 32B within that block, the command, the type (integer or floating), and the stream type (I or D) are the same, then the requests merge. Only read misses can merge. Merges can only occur until the first octaword of fill data arrives for that BAF entry.

If the request is a miss and it merges, the second MAF_idx location is validated and loaded into the existing BAF entry.

If the request is a miss and does not merge, a new BAF location is allocated and written with the command, address, set allocation, and MAF_idx. The command loaded into the BAF is a function of the Scache command and the status from the Scache tags. Table 5–12 for the full story.

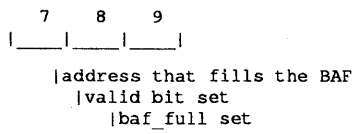
Table 5–12: Loading of BAF and VAF

Command	Hit	Shared	Dirty	Modified	BAF Command	VAF command
READ	HIT	*	*	*	NOP	NOP
READ	MISS	*	*	no	READ	NOP
READ	MISS	*	yes	yes	READ	VICTIM
WRITE	HIT	no	no	no	TAG UPDATE	NOP
WRITE	HIT	no	yes	no	NOP	NOP
WRITE	HIT	yes	no	no	WRITE BRDCST	WRITE_DATA
WRITE	HIT	yes	yes	no	WRITE BRDCST	WRITE_DATA
WRITE	MISS	*	*	no	READ_FOR_WRITE	NOP
WRITE	MISS	*	*	yes	READ_FOR_WRITE	VICTIM
LDx_L	HIT	*	*	*	LOCK	NOP
LDx_L	MISS	*	*	no	READ_LOCK	NOP
LDx_L	MISS	*	*	yes	READ_LOCK	VICTIM
STx_C	HIT	no	no	no	TAG UPDATE	NOP
STx_C	HIT	no	yes	no	NOP	NOP
STx_C	HIT	yes	no	no	STx_C BRDCST	WRITE_DATA
STx_C	HIT	yes	yes	no	STx_C BRDCST	WRITE_DATA
STx_C	MISS	*	*	no	READ_FOR_WRITE	NOP
STx_C	MISS	*	yes	yes	READ_FOR_WRITE	VICTIM
FETCH	*	*	*	*	FETCH	NOP
FETCH_M	*	*	*	*	FETCH_M	NOP
MB	*	*	*	*	MB	NOP

If the miss caused an Scache victim to be created, the VAF will be loaded with the Scache index, the set number of the victim, and the modify bits for that block. At least one modify bit must be set for a victim to be created. Note that the block must be dirty to have modify bits set. Having modify bits set in a clean block is an error condition.

If the allocation of the BAF fills the BAF, the SAU must be informed. The SAU will use this information to prevent the overflow of the BAF. The baf_full timing is shown in Figure 5–30.

Figure 5–30: BAF full timing



Some commands do not access the Scache, but are passed through to the BIU. FETCH, FETCH_M, and MB will allocate a new BAF entry every time.

A BAF entry is cleared after a fill is completed, a FETCH, FETCH_M, or MB is completed, or a shared or IO write is completed. The BAF entry is cleared 5 cycles after a fill for a write miss completes to allow time for the WBU to reissue the write. The valid bit for the entry is cleared the cycle after the clear condition occurs.

If a FLUSH, INVALIDATE, or SET_SHARED hit on an entry in the BAF, the entry is a write command, and a VAF entry is allocated for it (shared or IO write), then the BAF_inval bit will be set for that BAF entry. When all the data for the write is collected, the entry will be shown to the BSQ. If the BAF_inval bit is set, the BSQ will forward the correct MAF_idx to the WBU with a fail notification. The WBU will then reissue the write if necessary.

5.2.3.3.2 Loading the BAF and VAF

The following bits of information must be stored in the BAF file for each entry:

- Address<39:4>
- BAF CMD<3:0>
- SC_SET<1:0>
- Victim Hit or Bcache Index Match
- MAF1_Valid
- MAF1_idx<4:0>
- MAF2_Valid
- MAF2_idx<4:0>
- ARB
- Bcache miss
- Stop merging
- BAF_inval

The following list of functions will be performed on the BAF entries:

- Write Address, CMD, SC_SET, Vic./BIM, MAF1_valid, MAF1_idx,
- Set ARB
- Set BC_MISS
- Set BAF_inval
- Set Stop merging
- Set MAF2_valid, Write MAF2_idx
- TROLL(cam on Address<12:5>), in 32-byte mode
- TROLL(cam on Address<12:6>), in 64-byte mode
- HIT(cam on Address<39:6>)
- MERGE(cam on Address<39:6>, XOR Address<5>)

5.2.3.3.3 Victims

Victims are only generated by dirty and modified blocks which are deallocated from the Scache.

The Victim Address File (VAF) performs actions when it detects any of the following conditions:

1. Victim generated from a miss
2. WRITE BROADCAST command from the WBU
3. WRITE BROADCAST LOCK command from the WBU
4. WRITE FULL BLOCK command from the WBU
5. READ DIRTY which hits in the VAF
6. SC_INVALIDATE which hits in the VAF on a victim

The following is a brief description of the actions taken by the VAF for each condition.

- Victim generated from a miss

  When a victim is produced, the VAF is loaded with the index, set number, and status bits (modify, shared, and dirty) of the Scache victim. The SAU generates a victim request on behalf of the VAF and arbs for the Scache. The VAF sends a request to the SAU for a second Scache access for the victim. When the SAU grants the VAF access to the Scache, the VAF sends the index, set number, and subblock for the victim to the Scache. The SAU sends FORCE_HIT to the Scache for this access. The Scache returns the data for that subblock and the tag for the block. The VAF sets the corresponding data valid bit for the block. In the meantime, the second Scache access request for the victim is arbing in the SAU. (Note that the tag is stored in the VAF only as the first subblock returns from the Scache. A fill to the block may change the tag before the victim is completely read out of the Scache). The second data valid bit for the victim is set when the data arrives in the BDP (Biu DataPath). Once both data entries are collected the VAF sends a request to the BSQ to read the victim out of the chip.
- WRITE BROADCAST, WRITE BROADCAST LOCK, and WRITE FULL BLOCK commands from the WBU

  These commands are treated in the same way by the VAF. After the permission grant arrives back from the system, the WBU signals the Mbox to start the write transaction. The VAF captures the address and data arriving from the Scache and sets the WB bit for that entry. When the data for the specific type of write is collected, the VAF signals the Biu Sequencer (BSQ) to begin the process of reading the data off chip. Data arriving from the Scache is not wrapped for these commands (ie the INT16 corresponding to the octaword 0 arrives first from the Scache, then the INT16 corresponding to to octaword 1, etc) For more information about these commands see Section 5.2.2.6.3 and <REFERENCE>(wr_full_blk).
- READ DIRTY which hits in the VAF

  When the address of a READ DIRTY command hits in the VAF, the shared bit is set for the entry. When the BSQ sends the data off chip, the Entry_Valid bit for that entry is cleared.
- SC_INVALIDATE which hits in the VAF

  When the address of an Scache invalidate command hits in the VAF on a victim, the Entry_Valid bit for that entry is cleared if it is not currently being sent to the pins by the BSQ. An INVALIDATE to should not occur to the address of one of these WRITES after EV5 has been ACKED for them. This case should be checked with an assertion checker in the behavioral model.

The subblock entry (bit A<5>) is generated using the fill order to the Scache block when a victim is read out. The data arrives from the Scache in the same order in which it will be filled. While the VAF is processing one of the three WRITE commands listed above, the subblock entry (bit A<5>) starts at 0 for the first victim read and is a 1 for the second victim read.

The commands SET_SHARED and CLR_SHARED should not hit in the VAF due to the operations which must precede them. SET_SHARED should only be sent to clean blocks which, by definition, won't be in the VAF. A CLR_SHARED to a block is processed after a WRITE BROADCAST to that block. The VAF would invalidate its entry after the WRITE BROADCAST thus the CLR_SHARED would not hit in the VAF.

Quadword ECC is generated for the data entries in the VAF as they arrive from the Scache.

The VAF is required to accumulate the entire block from the Scache. It is also required to send the entire block out to the system on any of the following cases:

- A Bcache is NOT present.
- The operation is a WRITE BROADCAST, WRITE BROADCAST LOCK, or WRITE FULL BLOCK.
- All four modify bits are set for the Victim entry.

However, if only one modify bit is set for the block and none of the other conditions listed above are met, then the VAF can write out only the octawords that have modify bits set. This reduces the bus traffic on the pins.

The VAF can only process 1 victim at a time. So any instruction which is issued by the Mbox which requires VAF resources (IO write, shared write, or victim) will be retried until the victim is processed. IO writes or shared writes can be processed by the VAF as quickly as they are issued. The VAF processes an entry by collecting all the data required for the entry based on type of entry and the CBOX SC_BLK_SIZE ipr bit. Victims always collect 64b of data regardless of the value of the Scache block size. IO writes always are 32b regardless of the assertion of the Scache block size. Once all the required data for a victim is collected, the VAF sends a request to the BSQ to send the victim data to the pins. When processing a WRITE_BROADCAST of WRITE_BLK, the VAF informs the BAF once all the data is collected for the entry and then the BAF issues the request to the BSQ to service the WRITE_BROADCAST or WRITE_BLK.

The following bits of information must be stored in the VAF file for each entry:

- Address<39:15>
- Address<14:6>
- SC_SET<1:0>
- Modify<1:0>
- Data Valid<1:0>
- Entry Valid
- Shared
- Dirty
- WB

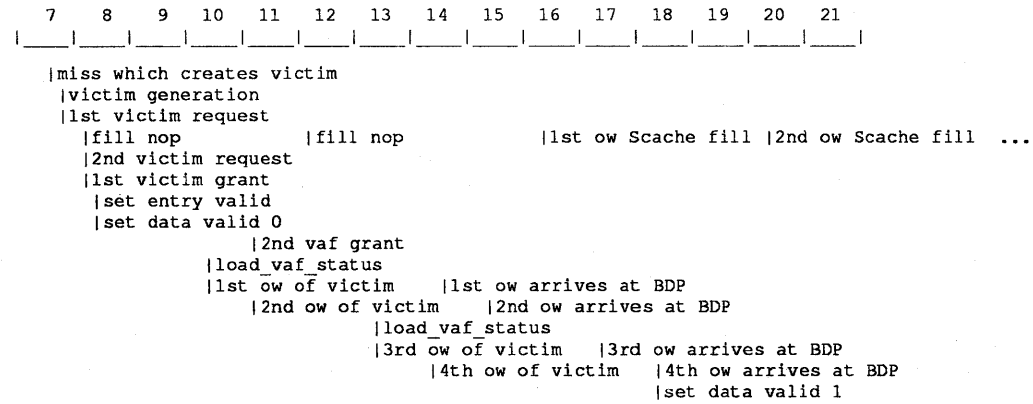
The following list of functions will be performed on the VAF entries:

- Write Address<14:6>, SC_SET, Modify, Set Entry_Valid, Clear Data Valid

- Set Data Valid
- Write Address<39:15>
- VICTIM_HIT(Cam on Address<39:6>)
- Clear Entry Valid, WB
- Write Address<39:6>, SC_SET, Modify, Set Entry_Valid, WB, Clear Data Valid
- Set Shared bit
- Write Dirty (possibly could be optimized away)

The timing for a sample victim flow is shown in Figure 5–31. This sample hits in the Bcache. The Bcache access is 4 cpu cycles. Two octawords of the victim are read in the first Scache access. The third and fourth octawords of data are read in the second Scache access.

Figure 5–31: Victim data flow



The loading of the first subblock of data in the VAF for a shared or IO write is shown in Figure 5–32. The WB bit signifies that this entry is a write transaction rather than a victim. Longword valid bits are used to select between the **WFB** (asserted) or the **RFB** (deasserted) for the data arriving to the BDP for the write.

The timing for a populate write in 64b mode is shown in Figure 5–33. The populate write has all its longword valid bits deasserted, thus it is read entirely from the Scache via the **RFB**.

Figure 5–32: Data collection of first subblock in VAF

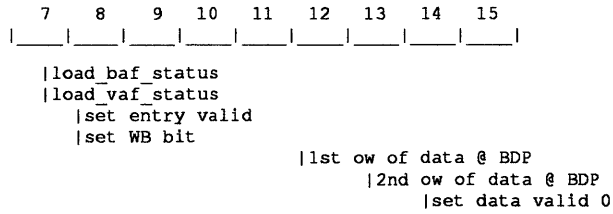
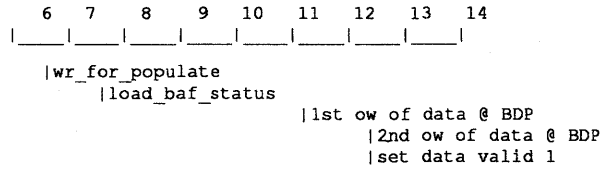


Figure 5–33: Data collection of second subblock in VAF



5.2.3.4 System Probe Address Requests

EV5 receives system probe commands and addresses on dedicated bidirectional pins from the system interface. Cache block address<39:04> and the four-bit system probe command are latched on separate buses in one system clock cycle following the assertion of Addr_Bus_Req_H.

The following commands can be received at any time from the system. These commands require the Scache to be probed and possibly modified. For systems with a Bcache, access to that cache may also be necessary to complete the transaction.

- INVALIDATE
- SET SHARED
- READ
- READ DIRTY
- FLUSH

Once a valid address and command are received from the system, indicated by the assertion of Addr_Bus_Req_H, the address is written to the System Probe Register (SPR) in the Cbox Address Data Path (ADP), and the command is fed into the System Probe Arbiter (SPA), which processes all system requests. In general, the command will be completed by a two step procedure.

1. Arbitration to access the Scache

   Access to the Scache is requested from the Scache Arbiter Unit (SAU). Once access is granted by the SAU, the Scache is probed and/or updated as required by the probe command. If there is a fill or victim operation pending, there will be some delay in completing the Scache access.

2. Arbitration to access the Bcache

If a Bcache is present in the system, arbitration is requested from the BIU Sequencer (BSQ) for access to modify the Bcache. The BSQ then notifies the SPA once the Bcache access is complete, or has started in the case of probe command requiring data transmission. Only then is the command acknowledged by EV5, and the SPA can process the next probe command.

Once the system asserts Addr_Bus_Req_h to transmit probe commands, it is allowed to send two probe commands without waiting for for a response from EV5. Then the system must wait for a response before dispatching another probe command, and if Addr_Bus_Req_h is left asserted to process a packet of commands, the system must transmit a NOP as a command until it receives a response, when it can dispatch the next command. Otherwise, Addr_Bus_Req_h must be deasserted until a response for a previous command is received.

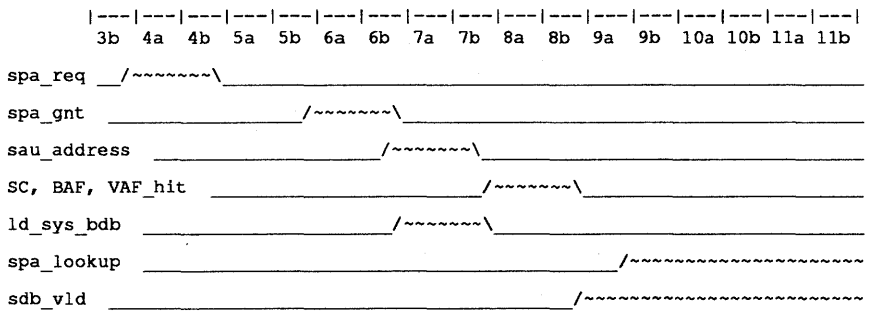
This is necessary to prevent the System Probe Arbiter (SPA) from processing the same command multiple times, since the probe command is held by a latch in the pad ring as long as Addr_Bus_Req_h is asserted.

If a parity error occurs as a result of either a bad probe command or address, the SPA begins processing the command in the normal manner, in order to get access to the Scache and Address Data Path (ADP) from the Scache Arbiter Unit (SAU). This is the only means writing the system probe address to the proper Cbox IPR in the ADP, i.e. EI_ADDR. Once the SAU has granted access, the SPA terminates the command and transmits NOACK as a system response.

In processing system probe commands that require transmission of a cache block from the Scache, such as a READ or READ_DIRTY, the data is unloaded from Scache and stored in the sys_data buffer of the data datapath explained below. The SPA sends signals to both the Victim Address File (VAF) control and the BIU Sequencer (BSQ) to facilitate the transfer of data to the system.

- c_biu_spa%ld_sys_bdb_6b_h : notifies VAF control that data is arriving to be loaded in sys_data buffer.
- c_biu_spa%spa_lookup_a_h : a strobe signal that notifies VAF control a probe command is in progress that will use one of the three data buffer entries in the data datapath.
- C_BIU_SPA%SYS_BDB_ENTRY_8B_H<1:0> : two-bit field indicates to VAF control which data buffer entry to use for the probe command.
- c_biu_spa%sdb_vld_8b_h : notifies BSQ whether to use a data buffer entry or Bcache for probe command; 1-data buffer, 0-Bcache.

Figure 5–34: Timing for System Probe Address Logic



Below is a brief description of the procedure followed by the System Probe Arbiter (SPA), for each system probe command issued to EV5. For more details about the interaction of the SPA with the Scache Arbiter Unit, see <REFERENCE>(biu_cmds).

- INVALIDATE

  For INVALIDATE, SPA requests access to the Scache by sending an SC_INVAL command and asserting Spa_Req_3b_h to the SAU. Regardless of the result of address compares in the BIU miss Address File (BAF) and Victim Address File (VAF), or whether the command hits or misses in the Scache, once the SAU grants access to the Scache by asserting Spa_Granted_5b_h, the probe command is sent to the BSQ to update the Bcache, providing a Bcache is present. Otherwise the SPA returns to the idle state, ready to process the next probe command. When the SPA is finished with the probe command, EV5 responds by sending ACK/Shared/Scache if no Bcache is present, or sending ACK/Shared/Bcache if there is one.

  If there is a victim pending in the VAF, or a shared write pending in both the BAF and VAF, these entries will be invalidated; the victim will be cleared from the cache system, and the shared write will be restarted. If there is a fill pending to the Scache from Bcache that matches the cache block being invalidated by the system, the BAF entry will be invalidated.

- SET SHARE

  For SET SHARE, SPA requests access to the Scache to perform a SC_SET_SHARED command. Regardless of the result of address compares in the BAF and VAF, or whether the command hits or misses in the Scache, once the SAU grants access to Scache, the probe command is sent to the BSQ to update the Bcache, providing a Bcache is present. Otherwise the SPA returns to the idle state, ready to process the next probe command. When processing of the SET SHARE command is complete, EV5 responds with ACK/Share/Scache if no Bcache is present, or sending ACK/Shared/Bcache if there is one.

- READ

  For a READ, SPA requests access to the Scache for an SC_READ command. If the EV5 is in 32-byte mode only one SC_READ is requested, but in 64-byte mode two scache accesses are requested. The resulting value driven on the read fat bus (RFB) is loaded into the sys_data buffer by the VAF controller on assertion of ld_sys_bdb_6b_h by SPA.

Next the SPA checks the results of address compares in the ADP and tag compares in the Scache. This is done after the first Scache access in 32-byte mode, and after the second access in 64-byte mode. If the probe command hits in the Scache (Sc_hit), and there is not a fill or victim pending, the SPA generates the following signals: Spa_Lookup_a_h is asserted for the rest of the transaction, Sys_Bdp_Entry_8b_h<1:0> is written with the entry number of the sys_data buffer (0x2), sdb_vld_8b_h is also asserted for the rest of the transaction, and the probe command is sent to the BSQ. When the BSQ processes the command, the cache block in the sys_data buffer is transmitted from the EV5, a signal is returned to SPA, c_biu_bsq%spa_bc_dne_9b_h, which clears the above logic, and an ACK/Shared/Scache response is transmitted.

If there is a hit in the address compare with a Victim Address File (VAF) entry, but not with any BIU miss Address File (BAF) entries, this means a pending victim matches the cache block the system intends to read. In this case, Sys_Bdp_Entry_8b_h<1:0> is written with the entry number of the matching VAF entry, either (0x0) or (0x1), and either the vic0 or vic1 data buffer entries is used both to process a victim and the system read. The ACK/Shared/Scache response is sent to the system.

If there is a hit in the BAF, but not the VAF, this indicates a pending fill to Scache. The probe command is sent to the BIU Sequencer (BSQ) without asserting sdb_vld_8b_h, providing a Bcache is present in the system. The BSQ then looks in Bcache to complete the probe command. If the cache block is found in Bcache an ACK/Shared/Bcache is sent, otherwise NOACK is dispatched.

If the cache block is not found in Scache, the probe command is sent to the BSQ to complete, provided a Bcache is present, and an ACK/Shared/Bcache is sent. If a Bcache is not present in the system, NOACK is sent as a response and SPA returns to the idle state in order to process the next command.

- READ_DIRTY

  For a READ_DIRTY, SPA requests access to the Scache for an SC_READ_DIRTY command, one access in 32-byte mode and two Scache accesses in 64-byte mode. The resulting value driven on the read fat bus (RFB) is loaded into the sys_data buffer by the VAF controller on assertion of ld_sys_bdb_6b_h by SPA.

  Next the SPA checks the address compares in the ADP and tag compares in the Scache, this being done after the first Scache access in 32-byte mode, and after the second access in 64-byte mode. If an Sc_hit results and the cache block is dirty, and there is not a fill or victim pending, the SPA drives the following: Spa_Lookup_a_h is asserted for the rest of the transaction, Sys_Bdp_Entry_8b_h<1:0> is written with the entry number of the sys_data buffer (0x2), sdb_vld_8b_h is also asserted for the rest of the transaction, and the probe command is sent to the BSQ. When the BSQ processes the command, the cache block in the sys_data buffer is transmitted from the EV5, a signal is returned to SPA, c_biu_bsq%spa_bc_dne_9b_h, which clears the above logic, and an ACK/Shared/Scache response is transmitted.

  If there is a hit in VAF but not the BAF, meaning a pending victim that matches the READ_DIRTY cache block, Sys_Bdp_Entry_8b_h<1:0> is written with the entry number of the matching VAF entry, and either the vic0 or vic1 data buffer entries is used both to process a victim and the READ_DIRTY probe command. The ACK/Shared/Scache response is sent to the system.

  If there is a victim pending in the VAF, the VAF controller is responsible for setting the share status bit for that entry.

If there is a hit in the BAF, but not the VAF, this indicates a pending fill to Scache. The probe command is sent to the BIU Sequencer (BSQ) without asserting sdb_vld_8b_h, providing a Bcache is present in the system. The BSQ then reads the data out of the Bcache to complete the READ_DIRTY command. Since this probe command should always be found in the EV5 cache system, an ACK/Shared/Bcache is sent.

If the cache block is not found in Scache, the probe command is sent to the BSQ to complete, provided a Bcache is present, and an ACK/Shared/Bcache is sent. If a Bcache is not present in the system, NOACK is sent as a response and SPA returns to the idle state in order to process the next command.

- FLUSH

  For FLUSH, SPA requests access to the Scache for an SC_READ command, one access in 32-byte mode and two Scache accesses in 64-byte mode, in order to accumulate a victim in the sys_data buffer before invalidating the cache block.

  Next the SPA checks the address compares in the ADP and tag compares in the Scache, this being done after the first Scache access in 32-byte mode, and after the second access in 64-byte mode. If an Sc_hit results and the cache block is dirty, and there is not a fill or victim pending, the SPA drives the following: Spa_Lookup_a_h is asserted for the rest of the transaction, Sys_Bdp_Entry_8b_h<1:0> is written with the entry number of the sys_data buffer (0x2), sdb_vld_8b_h is also asserted for the rest of the transaction, and the probe command is sent to the BSQ.

  As the BSQ processes the command, however, the SPA requests access of the Scache for an SC_INVAL command, to flush the cache block from the scache. The BSQ transmits the cache block in the sys_data buffer from the EV5, and sends a signal back to SPA, c_biu_bsq%spa_bc_dne_9b_h, which clears the above logic, and an ACK/Shared/Scache response is transmitted.

  If there is a hit in VAF but not the BAF, meaning a pending victim that matches the cache block to be flushed, Sys_Bdp_Entry_8b_h<1:0> is written with the entry number of the matching VAF entry, and either the vic0 or vic1 data buffer entries is used both to process a victim and the FLUSH probe command. The SPA requests access to Scache for an SC_INVAL command, and an ACK/Shared/Scache response is sent to the system.

  If there is a hit in the BAF, but not the VAF, this indicates a pending fill to Scache. In this case the SPA requests access to Scache for an SC_READ in order to recirculate the FLUSH probe command to allow the fill to complete. This has the side effect of reloading the sys_data buffer entry in the data datapath.

  Address compares in the Scache, BAF, and VAF indicate a pending write to the sytem waiting for permission to start. The SPA will attempt to process the FLUSH command, but its request to the Scache Arbiter Unit (SAU) will be disabled in this instance by a signal from the Write Buffer Unit (WBU), c_wbu%stop_spa_4a_h. The SPA request should be disabled until the pending write completes its Scache access, and then allowed through.

  If the cache block is not found in Scache, the probe command is sent to the BSQ to complete, provided a Bcache is present, and an ACK/Shared/Bcache is sent. If a Bcache is not present in the system, NOACK is sent as a response and SPA returns to the idle state in order to process the next command.

Table 5–13: System Probe Commands and Related Actions if Address match

Command	Where	Present	Action
INVALIDATE	Scache	Maybe	clear valid bit on Sc_Hit
	BAF	Maybe	clear valid bit on entry
	VAF	Maybe	clear valid bit on entry
	Bcache	Maybe	clear valid bit on Bc_Hit
SET SHARE	Scache	Maybe	set share bit on Sc_Hit
	BAF	Maybe	No action taken
	VAF	Maybe	set share bit on entry
	Bcache	Yes	set share bit
READ	Scache	Maybe	load sys_data buffer and use if Sc_Hit
	BAF	Maybe	if fill, Read done from Bcache, else NOACK
	VAF	Maybe	vaf entry used for victim and read
	Bcache	Maybe	Read from Bcache if Bc_Hit, elde NOACK
READ DIRTY	Scache	Maybe	Do Rd_Drty if Sc_Hit and Dirty
	BAF	Maybe	if fill, Rd_Drty from Bcache, else NOACK
	VAF	Maybe	vaf entry used for victim and Rd_Drty
	Bcache	Yes	Rd_Drty from Bcache if not Sc_Hit
FLUSH	Scache	Maybe	If Sc_Hit & Drty, do Read, else invalidate
	BAF	Maybe	if fill, wait then invalidate
	VAF	Maybe	vaf used for victim & flush, then invalidate
	Bcache	Maybe	deallocate block if dirty, then invalidate
			if write_broadcast, wbu will disable

5.2.3.5 System Data Requests

These commands are used by the system to move data in and out of the EV5 cache system.

- SEND BRDCST DATA
- SEND DIRTY DATA
- READ VICTIM DATA
- TAG WRITE
- FILL0
- FILL1
- FILL0 SHARED
- FILL1 SHARED
- FILL0 NO CHECK

- FILL1 NO CHECK

The SEND BRDCST DATA command will select the address and data of the next write broadcast that is to take place.

The SEND DIRTY DATA and TAG WRITE commands will use the address in the System Probe Register and the data in the system data buffer if data is required.

READ VICTIM DATA will use the address and data in the next victim buffer to be written to memory.

The FILLn commands will use the address in the nth BAF register.

5.2.3.5.1 BIU Sequencer

The BIU sequencer (BSQ) creates the runs the EV5 command and selects the address that is used to control the Bcache and request service from the System. It also produces read addresses for the data buffers.

Inputs to BSQ include:

- Sysclock-2
- Last EV5 CMD
- next BAF request
- next VAF request
- System Bcache Request
- NO_EV5_ACCESS
- VICTIM_BUFFERS_FULL
- NEXT_EV5_REQUEST
- SYSTEM_DATA_CMD
- Bcache Hit
- Configuration Data

Outputs from BSQ include:

- New BAF state
- New VAF state
- New EV5 CMD
- Data Buffer Read Address
- Clear System Bcache Request

The rough ARB priority for BSQ is this

1. System Data request
2. System Bcache request
3. BAF request
4. VAF request

The VAF will have priority over the BAF if the next request from the BAF has victim hit/Bcache index match asserted.

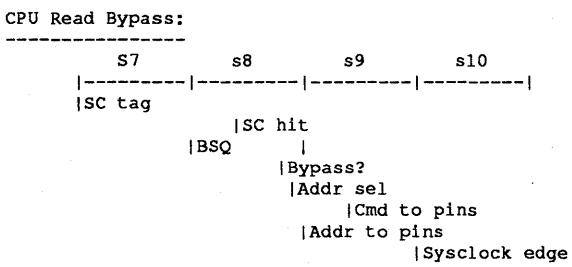
These commands will be used by the BIU to service all requests from the Scache and System. Refer to the EV5 Functional Specification for a full description of these commands.

- READ
- WRITE INT32
- READ MISS
- READ MISS MOD
- VICTIM
- DATA FROM BCACHE
- MEMORY BARRIER
- FETCH
- FETCH_M
- TAG UPDATE DIRTY
- TAG CHANGE
- WRITE DATA
- WRITEBACK
- WRITE BROADCAST
- WRITE BROADCAST LOCK

There are four basic sequences in the BSQ; CPU read bypass, CPU read, CPU write, and system cycle. They are outlined below.

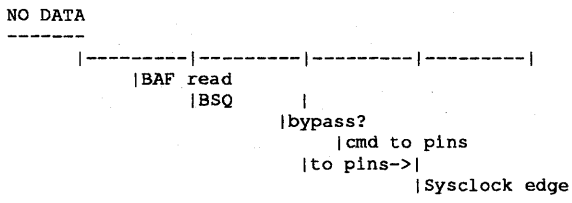
An Scache tag access starts in S7 of the pipe. If the access results in a miss that requires a read, of Bcache or memory, BSQ will attempt to drive the command and address off chip at the rising edge of S10. The miss signal arrives from the Scache during S8a. If a fill is required and there is no Bcache Index Match, the command can be bypassed. BSQ will have started at S8a and not finish until the end of S8b. It will assert bypass possible if the command could be driven at S10. This will only be true if there was nothing else to do and the needed clock edge will be there at S10. If the bypass is possible, BSQ will select the bypass address and send it to the pins. During S9a we will decide on the command to send and drive it to the pins. If there is no clock at S10, a second cycle of bypass is possible at S11. The miss will be written into the BAF at S9A and be read by the normal BSQ arbitration during S9B. This would result in a read starting in S12 or later.

Figure 5–35: BSQ Bypass Flow



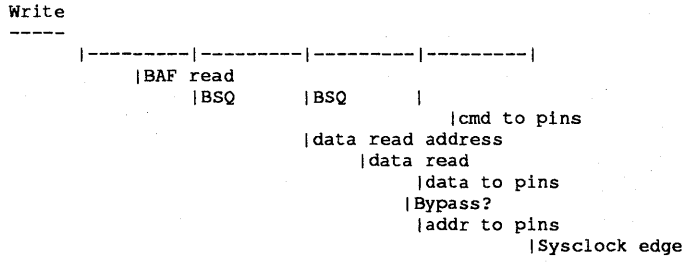
If there are reads to do in the BAF, the command will be read from the BAF during a B phase. The next cycle the BSQ will evaluate to produce the needed command. This will be driven along with the address to the pins.

Figure 5–36: BSQ No Data Flow



If the BAF or the VAF contains a command that requires data to be sent to the pins, this flow will be used. The command will be read in the B phase of the first cycle. During the second cycle BSQ will produce the read address for the data buffer and send it out. During the third cycle the command will be created and the address read out. During the four cycle the command, address, and data will be driven to the pins.

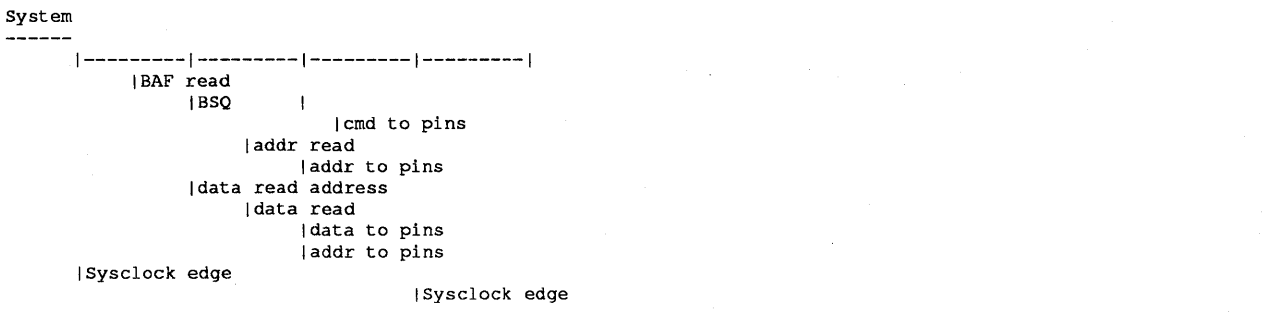
Figure 5–37: BSQ Data Flow



When a system data command is received we have one Sysclock to respond. The minimum Sysclock rate is 3 CPU cycles. In order to respond in time, most of the normal BSQ control must be bypassed. The data read address must be computed in the B phase of the first cycle. The data read will start in the B phase of the second cycle, allowing for one cycle to drive the address across the bottom of the chip. During the second cycle BSQ will produce the correct command.

The address is read during the second cycle from the BAF. During the third cycle the command, address, and data are driven to the pins.

Figure 5–38: BSQ System Flow



5.2.3.5.2 Bcache Data Cycle Timer

This control will time each Bcache read or write cycle. BSQ will start the timer at the beginning of each read or write. The timer will assert a done signal to BSQ at the end of each cycle.

5.2.3.5.3 Bcache Data Valid

This control will most likely be a timing chain that will provide a data valid signal at the end of each Bcache read. The first data valid of each read or write sequence will also be used to trigger the Bcache tag check.

5.2.3.6 Data Datapath:ECC generation/check

The CBOX provides parity bits for data, tag and status bits in the SCache. If EV5 is operating in ECC mode, Quadword ECC is provided for all off-chip data transactions and parity for all off-chip address transactions. Otherwise byte parity is generated on all data. The mode is determined by a bit in the BC_CONFIG IPR sitting in the address datapath.

Store data is written 2 octawords per transaction into the SCache in 2 consecutive cycles.

The data datapath consists of 3 sections.

- The outgoing data section which generates ECC on outgoing data
- The data buffer section
- The incoming data section which checks ECC on incoming data
- The IPR section

Physically the data datapath is split up into two halves, each half for each quadword of data, sitting on opposite sides of the chip.

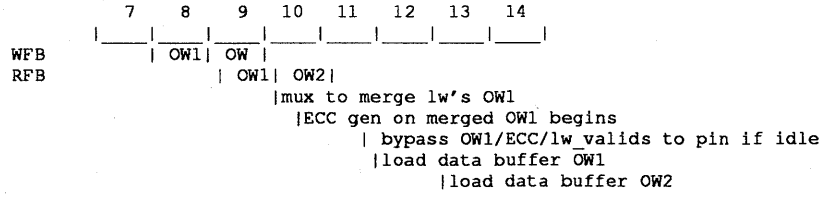
5.2.3.6.1 Outgoing Data section

For shared writes, valid longwords from the write buffer are merged in the BIU data datapath with the remaining longwords from the SCache coming from the **RFB** in 9b. If EV5 is operating in ECC (byte parity) mode, ECC (byte parity) is then generated and the data, LW valid bits along with the ECC (byte parity) are written into the BIU data buffer in 12a. The longword valid bits are driven onto the data valid pins through a mux which selects between these longword valid bits and QW valid bits (for reads from I/O space).

For I/O writes, all data is taken directly from the write buffer. No merging is done.

For victims,write for populates and system reads, all data entering the data datapath comes directly from the SCache (**RFB**). All lw_valid bits are set for victims. The timing diagram for the ECC generation for a shared write is shown in Figure 5–39.

Figure 5–39: Outgoing Data flow



5.2.3.6.2 Data buffer section

There are three data buffers sitting in the data datapath. Two for victims, shared writes and I/O writes, namely vic0 and vic1 and one to hold system data requests, namely sys_data. Each data buffer is capable of holding up to 64B of data along with ECC and longword valid bits for each quadword of data. The data buffer is written in cycle 12a for a write/victim read tag access beginning in cycle 6. The data buffer is read in cycle 11b to get data at the pins in cycle 13.

5.2.3.6.3 Incoming Data section & Error Signals

This section checks incoming data from the pins for ECC errors, corrects them if possible and returns the raw data to the DCache and returns the corrected data to the DCache and ICache.

Fill data is valid at the pins at the beginning of cycle 8 and is driven to the data datapath to be driven directly to the Dcache in 9b via the **RFB**. Longword parity is generated for the fill data and also driven to the DCache. (The data datapaths have a 3:1 mux that drives either fill data, corrected fill data or IPR data onto the **RFB**)

If EV5 is operating in ECC mode, the syndrome is first calculated for each of the 64 bits of fill data. If the syndrome is non-zero, that implies that an error occurred and **C_BDP%RAW_ECC_ERR_10B_H**<1:0> is asserted(one bit for each quadword). The syndrome is then decoded to correct the data if possible . If it is a single bit error, it is a correctable error and **C_BDP%CORR_ERR_11A_H**<1:0> is asserted.

NOTE

C_BDP%CORR_ERR_11A_H<1:0> should be examined only if the corresponding bit in **C_BDP%RAW_ECC_ERR_10B_H**<1:0> is asserted.

The data datapaths also have a longword parity checker to check the parity of data read from the SCache. This parity checker is used for BOTH arrowheads and Dreads. To summarize, the internal CBOX errors:

- **C_BDP%RAW_ECC_ERR_10B_H**<1:0> : one bit for ECC error for each quadword of valid fill data for both I & D streams. Not asserted if the FILL_NOCHECK pin is asserted for that octaword of fill data.
- **C_BDP%CORR_ERR_11A_H**<1:0> : Correctable error on each quadword of fill data for both I & D stream. Not qualified. Should be examined only when **C_BDP%RAW_ECC_ERR_10B_H**<1:0> is asserted.
- **C_BDP%RFB_PAR_ERR_10B_H**<3:0> : **RFB** LW parity error on SCache read hits and shared writes without permission, for which the Scache is read. To be examined twice for each SCache access for both octawords on the **RFB**.
- **S%C_TAG_PERR_7B_H**<2:0> : tag parity error, asserted for each set in the Scache for Scache reads and writes (even if it misses)

The local signals are used to generate the global signals shown below:

- **C%M_RFB_ECC_ERR_10B_H**: This signal is asserted only on Stream ECC errors.
- **C%I_HARD_ERR_TRAP_11B_H**: Uncorrectable ECC error (I or D stream) OR Scache fill parity (Data or Tag)error if Scache hit OR BCache tag parity error. EV5 goes into machine check. Asserted in 11b and 12b for data parity errors on both octawords from Scache.
- **C%I_CORR_ERR_TRAP_11B_H**: Correctable ECC error on Stream fills only. This signal is a flip flop that is set when a correctable error occurs and is cleared only when the corrected data is written back into the register file.
- **C%I_CORR_ERR_INTR_11B_H** : Correctable ECC error (both I/D streams)

The data is corrected and written into a silo in 11a to be written into the SCache and some later time. If the error is double bit or more, the data is not corrected.

On an ECC error, the CBOX enters error mode. In error mode, data is no longer driven directly to the Dcache from the pins but data is always corrected (if possible), written into the Scache and *then* returned to the DCache. This is so that corrected data can always be returned to the DCache/register file once an error occurs. Therefore on the first ECC error, the DCache and register file get the same data returned twice. First the raw data and then again in corrected form. After this, all data is returned via the SCache. The CBOX leaves error mode only after the BIU's address file is emptied.

In byte parity mode, the byte parity of the incoming data is generated and compared against the byte parity at the check pins. If there is a difference, the error is flagged as an uncorrectable ECC error and the CBOX enters error mode. The following points should be noted about the CBOX error mode:

- **C%M_RFB_ECC_ERR_10B_H** will be asserted on any ECC error. CBOX will enter error mode. Any further ECC errors while the CBOX is in error mode will NOT cause this signal to be asserted since data is being returned via the correctable path.

- C%I_CORR_ERR_TRAP_11B_H will be set on the first Stream correctable error received. This puts CBOX into error mode. If another correctable error occurs while this signal is set, this signal will not change (ie.it will remain set). This signal is de-asserted when the corrected data for the first error is written in to the register file It does NOT have to remain set until the corrected data for the second error is returned, because the CBOX had already entered error mode on the first error, and in this mode only corrected data , not raw fill data is returned to the register files. C%I_CORR_ERR_INTR_11B_H will be asserted twice however for each of the correctable errors.
- It is possible for hard errors to occur while C%I_CORR_ERR_TRAP_11B_H is set. (hard errors in the shadow of correctable errors).
- For Scache accesses, it is possible for both octawords of the 32B block being read to have parity errors. If so, the data read in 9b onto the RFB will have C%I_HARD_ERR_11B_H asserted in cycle 11b and the data read in 10b onto the RFB will have C%I_HARD_ERR_11B_H asserted in cycle 12b.
- If a hard error occurs, C%I_HARD_ERR_11B_H is asserted. This forces CBOX to enter error mode. Currently CBOX will do the following.
  - On the first hard error, flag C%I_HARD_ERR_11B_H (corresponds to raw fill data on the RFB in 9b) and enter error mode.
  - load error information into IPR's and lock them.
  - On subsequent hard error, set second error bit. Do not flag C%I_HARD_ERR_11B_H at this time (although this is easier for the CBOX)
  - Later, assert I_HARD_ERR_11B_H in cycle 11b when the fill data for the second error is being returned to the register files via the Scache in 9b

A summary of CBOX behavior with respect to the error signals while one error is pending is shown in Table 5–14

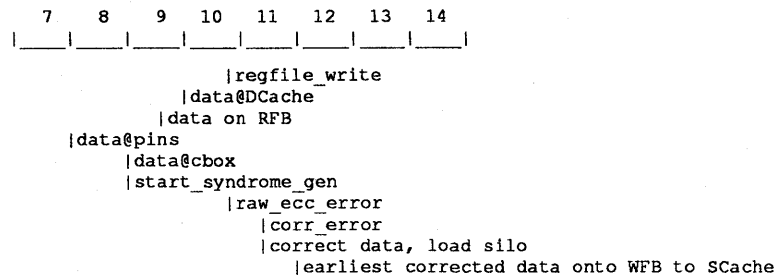
Table 5–14: Behavior of CBOX of errors in shadow of other errors

First error	Second	C%M_RFB_ECC_ERR_10B_H	C%I_CORR_ERR_TRAP_11B_H	C%I_HARD_ERR_TRAP_11B_H	C%I_CORR_ERR_INTR_11B_H
-	-	0	0	0	0
corr	-	1 (if Stream)	SET (if Stream)	0	1
corr	corr	0	No change. De-asserted when first corrected data is returned.	0	1
corr	hard	0	No change. De-asserted when first corrected data is returned.	assert only when the SCache is updated	0
hard	-	1 (if Stream)	0	1	0
hard	corr	0	0	0	1
hard	hard	0	0	assert only when the Scache is updated	0

The timing diagram for the data flow on fills is shown in Figure 5–40

The ECC code is defined in the EV5 CPU Chip Functional Specification. It provides single bit error detect & correct, double bit error detect, quad-bit nibble error detect and all ones and all zeros failure detect.

Figure 5–40: Incoming data flow



5.2.3.7 IPR's

There are 10 IPR locations in the CBOX, namely :

- SC_CTL (Phy. Addr:FFFFF000A8)
- SC_STAT (Phy. Addr:FFFFF000E8)
- SC_ADDR (Phy. Addr:FFFFF00188)
- BC_CTL (Phy. Addr:FFFFF00128)
- BC_CONFIG (Phy. Addr:FFFFF001C8)
- BC_TAG_ADDR (Phy. Addr:FFFFF00108)
- EI_STAT (Phy. Addr:FFFFF00168)
- EI_ADDR (Phy. Addr:FFFFF00148)
- FILL_SYN (Phy. Addr:FFFFF00068)
- LOCK (Phy. Addr:FFFFF001E8)

Of these IPR's the SC_CTL, SC_STAT and FILL_SYN IPR's sit in the upper quadword section of the data datapath. The remaining IPR's sit in the Address datapath. Apart from SC_CTL, BC_CTL and BC_CONFIG IPR's, all IPR's are readable. IPR's are driven onto the upper quadword of the **RFB** in 9b for a MBOX command issued in cycle 5. Some details about the IPR's are described below. For a more detailed discussion on the IPR's, please refer to the external functional specification.

5.2.3.7.1 SC_STAT

The SC_STAT IPR is written in cycle 13A if a parity error occurred on the access. This is because the worst case data parity error for the second OW, is available only in early 12a. It is locked in 13A on a parity error for any Scache Read or shared write. Writes to SC_STAT clear the IPR but *do not* unlock it. Only reads to SC_ADDR can unlock this IPR. Unlocking is also done in cycle 13. In order to obtain the status of an SCache read, a restriction placed is that any read to SC_STAT must be at least 5 cycles after the last SCache read. Similarly, any read to SC_ADDR must be at least 5 cycles after the last SCache access.

5.2.3.7.2 SC_ADDR

The SC_ADDR IPR is written in cycle 12A following every Scache access. It is locked from further writes if a tag or data parity error for an Scache access will cause SC_STAT to be written. Reading SC_ADDR will unlock the IPR and allow writes to occur.

5.2.3.7.3 SC_CTL

SC_CTL is written in cycle 8b from the first OW (lower 16B of 32B address) from the WB. It is written in cycle 8b and latched in 9a and driven to the SCache/CBOX/BIU. It can also be read.

5.2.3.7.4 FILL_SYNDROME

The syndrome of the fill data is is currently written (on ECC error)into this IPR in 12a corresponding to raw fill data on RFB in 9b. This is because ECC error for both qw's is valid only in mid 11a. If an ECC error is uncorrectable , this IPR is locked in cycle 12a. Correctable ECC errors *do not* lock this IPR. This IPR is unlocked by reads to EI_ADDR. Reads to FILL_SYNDROME can be made only after all fill data has been loaded into the SCache.

5.2.3.7.5 EI_STAT

The EI_STAT IPR is written in cycle 13A and locked for any of the following errors:

- ECC or byte parity error on fill data from Bcache or Memory
- Tag parity error on fill from Bcache
- Tag Status (Valid, Shared, Dirty) parity error on fill from Bcache
- Address and Command parity error on System Probe Command

Writes to EI_STAT clear the IPR but *do not* unlock it. Only reads to EI_ADDR can unlock this IPR. Unlocking is also done in cycle 13.

5.2.3.7.6 EI_ADDR

The EI_ADDR IPR is written in cycle 12A following every Scache access. It is locked from further writes if an ECC or parity error for a Fill or System Probe Command will cause EI_STAT to be written. Reading EI_ADDR will unlock the IPR and allow writes to occur.

5.2.3.7.7 BC_TAG_ADDR

The BC_TAG_ADDR IPR is written in cycle 9A following every Bcache access, with the exception of BC_HIT. It is locked from further writes if a Bcache tag or tag status parity error will cause EI_STAT to be written. Since BC_HIT cannot be computed as fast as the incoming tag and status bits, it is written and locked one cycle after the other bits in the IPR. Reading EI_ADDR will unlock the IPR and allow writes to occur.

5.2.3.7.8 BC_CTL

BC_CTL is written in cycle 8a with the first OW (lower 16B of 32B address) from the WB. It is latched in cycle 8a driven to the Bcache/CBOX/BIU. This IPR is write only.

5.2.3.7.9 BC_CONFIG

BC_CONFIG is written in cycle 8a with the first OW (lower 16B of 32B address) from the WB. It is latched in cycle 8a driven to the Bcache/CBOX/BIU. This IPR is write only.

5.2.3.7.10 LOCK

The LOCK register file entry can also be read as an IPR location in the address data path. It is read in the same manner as any other Cbox IPR.

5.3 Reset and Initialization

5.4 Error Handling and Recording

5.5 Test Aspects

5.6 Performance Monitoring Features

5.7 Issues

5.8 Revision History

Table 5–15: Revision History

Who	When	Description of change
Chandra Somanathan	12-08-1991	Cbox Arbiter, Set Allocation, Transaction Flows
Sribalan Santhanam	12-15-1991	adding wbuffer,iprs and block diagram
Cbox team	2-20-1991	lots more stuff added especially to BIU

Chapter 6

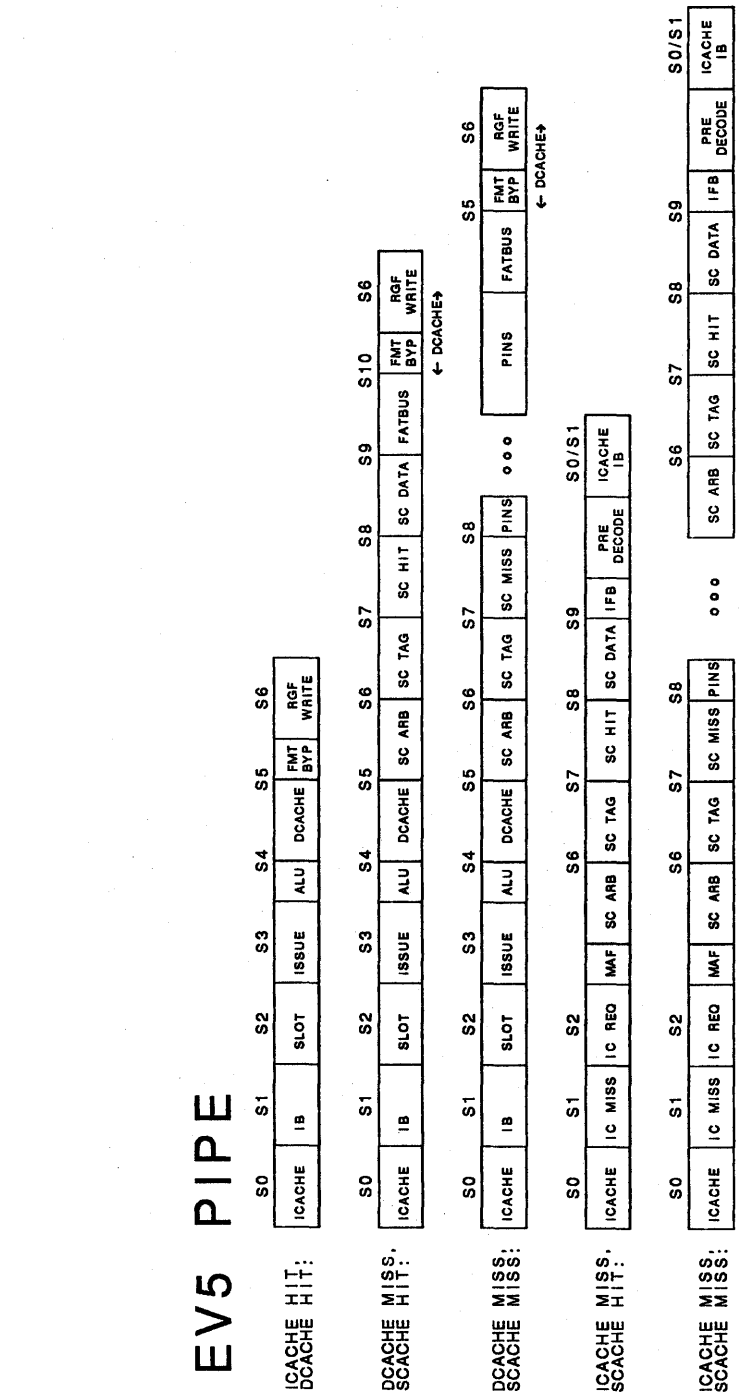
The Caches

6.1 Overview

The EV5 on-chip memory is structured into two levels. The first level consists of separate instruction and data caches; the second level is a unified instruction and data cache. The instruction cache (ICache) is an 8 KByte direct-mapped virtual cache, accessed from the Ibox. The data cache (DCache) is a 16 Kbyte direct-mapped write-through physical cache, accessed from the Mbox. The second-level cache (SCache) is a 96 Kbyte, 3-way set associative, write-back, physical cache, accessed from the Cbox.

The ICache resides in S0 of the EV5 pipeline. The DCache resides in S4B/S5A. The SCache resides in S6B through S11A of the pipeline.

Figure 6–1: Cache Positions in EV5 Pipeline



6.2 ICache and Refill Buffer Functional Description

IBOX instruction data is stored in the Icache and the Refill Buffer (RFB) (see Figure 6–2). Instructions are processed in the IBOX from the Instruction Buffer (IB); data is loaded into the IB from two sources:

- From the Refill Buffer during FILLs or during a Read when (**I_HIT%RFB_HIT_1A_H** AND NOT **I_HIT%IC_HIT_1A_H**).
- From the Icache on **I_HIT%IC_HIT_1A_H**.

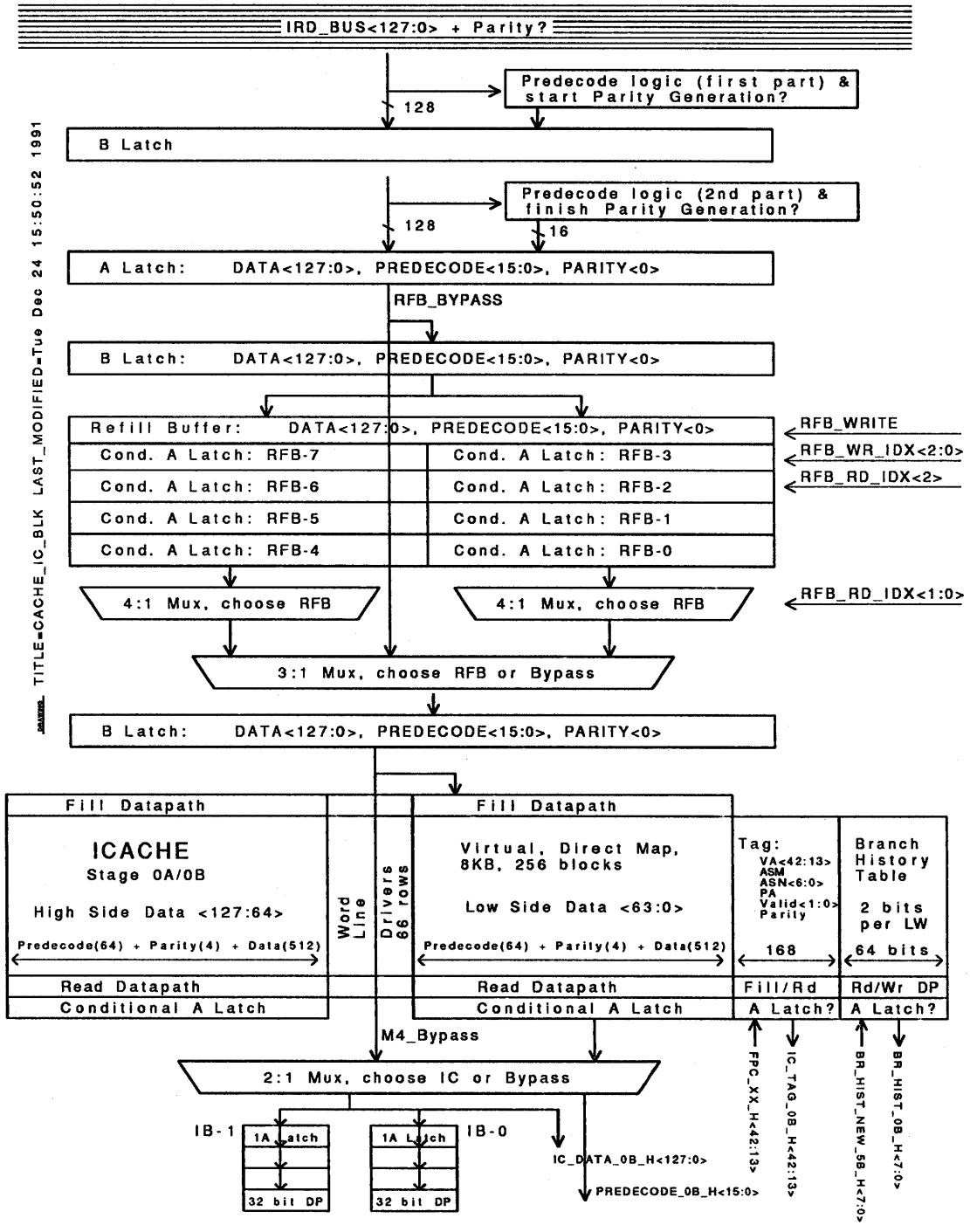
The IBOX sends the control signals that determine the data flows to the Icache and Refill Buffer; these flows are detailed in Section 6.2.4.

The Icache is an 8Kbyte, direct-mapped, virtual address cache that holds 256 32-byte blocks of instruction stream data. The Icache has a one cycle access and a one cycle repetition rate for both FILLs and READs. A cache block is filled in two octaword FILL transactions, and READs to the cache read an octaword of data (four instructions).

The Refill Buffer is an 8-entry prefetch buffer holding 8 octawords of instruction data in the same format as the Icache, see Section 1.2.2, Instruction Fetch. The data portion of the buffer is in the Icache datapath while tags and control are in the IBOX. The IBOX directs the filling of the Refill Buffer by sending the FILL enable, **I%J_RFB_WRITE_A_H**, and the FILL index, **I%J_RFB_WR_IDX_A_H<2:0>**. The IBOX directs the reading of the Refill Buffer via **I%J_RFB_RD_IDX_B_H<(6:4)>**. Data is written into the Buffer with conditional A-latches and read using a mux.

Both the Refill Buffer and the Icache hold predecoded data bits, 5 bits per instruction. A cycle is allocated to decode these bits from the FILL data on **S%J_IFB_DATA_9B_H<127:0>**; the advance decoding and storing of this data saves time in the slotting logic and the branch logic when the actual instruction is read and processed, see Section 1.2.8, Instruction Slotting.

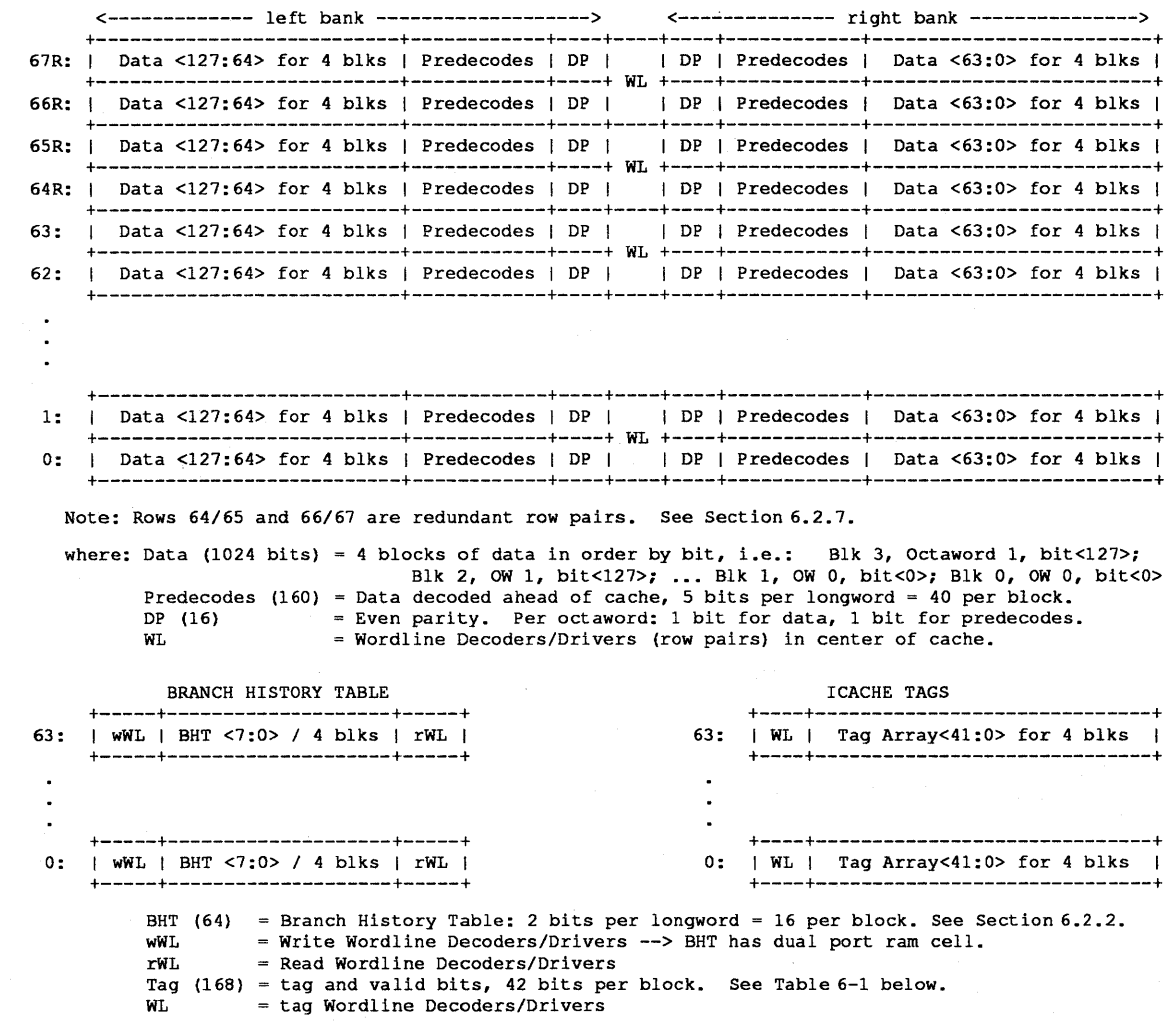
Figure 6–2: Instruction Data Flow through Refill Buffer and Icache



6.2.1 Icache Details

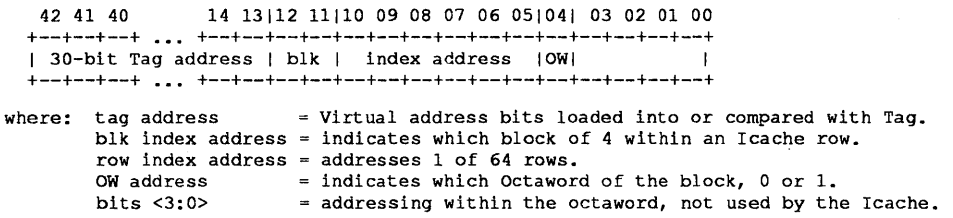
The logical organization of the Icache is shown below:

Figure 6–3: Logical Icache Organization



As can be seen from the diagram, the Icache is organized into 64 direct mapped indexes, where each index consists of four blocks. The breakdown of the Virtual address bits for Icache decoding is shown below:

Figure 6–4: Icache Address Breakdown



The 42-bit Icache tag field holds the tag and the following information:

Table 6–1: Icache Tag

Name	Extent	Description
Tag	41:12	30 bit tag, Virtual Address <42:13>
ASM	11	Address Space Map
ASN	10:4	Address Space Number <6:0>
PA	3	Indicates Icache address is a Physical Address
Valid	2:1	Valid bits, 2 per block = 1 per octaword
Parity	0	Even tag parity for VA<42:13>, ASM, ASN<6:0>, PA

The ASM and ASN bits allow implementation of process tags; see the ALPHA SRM for more information. The Physical Address bit specifies that the Tag is physical, not virtual; it prevents address translation by the ITB. There are 2 valid bits per block because a FILL to the Icache occurs as two separate octaword FILL transactions.

6.2.1.1 Icache SROM Interface

The Icache supports a Serial ROM interface for diagnostics to allow the Icache (data, tags, Branch History Table) to be written and read in a serial fashion from the pins. An IBOX counter sequences the index during serial reads and writes. See Section 6.2.4.1.1.

6.2.2 Branch History Table

The Branch History Table (BHT) is physically separate from the Icache, but its timing and design are very similar to the Icache, so it is part of the Icache block. The control for updating the Branch History Table is in the IBOX, see Section 1.2.4, Branch History Table. Each longword of data has 2 branch history bits to implement a 2-bit branch prediction scheme. These bits are not initialized on FILLs, but the SROM interface may be used to initialize the table in a serial fashion. If the table is not fully loaded using the SROM, the first time a location is read, these bits are UNPREDICTABLE. The Branch History Table is read with the Icache data in stage 0 and updated in stage 6 once the branch results are known. Like the Icache, Branch History reads and updates correspond to four instructions, i.e. 8 bits at a time.

When the Instruction Buffer is loaded from the Refill Buffer instead of the Icache, the IBOX accesses the Branch History Table using the Refill Buffer address instead of the Icache address. This is handled in the IBOX using a 2:1 mux to set the BHT index, **J_BHT_IDX_ZB_H<12:4>**. There is a good chance that the prediction bits delivered on a Refill Buffer access are correct, being left from the last time that particular branch instruction was stored in the Icache. This is true because:

- The BHT is not initialized on FILLs, and
- Instructions are swapped in and out of the Icache, and
- Not all instructions are branches.

These factors combined give the effect of a BHT that holds history bits for more branches than are actually stored at any one time in the Icache.

6.2.3 Icache and Refill Buffer Initialization and Test

All valid bits are cleared using **I%J_FLUSH_A_H** from the IC_FLUSH_CTL IPR, see Section 1.2.10.11. There are no external invalidates for the Icache.

On reset, after the BiST logic completes, all valid bits should be cleared using **I%J_FLUSH_A_H** or another signal which has the same effect.

It is planned that the full Icache will be flushed using **I%J_FLUSH_A_H** if data with a parity error or uncorrectable ECC data has been written into the Icache. For a description of ECC and parity error handling, see the Error Handling Chapter in the EV5 CPU Chip/Functional Specification.

The IBOX is responsible for initializing and maintaining the valid bits for the Refill Buffer, see Section 1.2.2, Instruction Fetch.

Bad parity may be forced for both the Icache tag and data parity by writing bits in the ICSR IPR, see Section 1.2.10.17, Ibox Control/Status Register, ICSR. The IBOX handles asserting bad parity for tags; if the IBOX asserts **I%J_FORCE_BAD_DP_A_H**, the parity destined for the Icache will be inverted ahead of the Refill Buffer.

6.2.4 Icache & Refill Buffer Transactions

The IBOX and CBOX send the control signals necessary for reading and writing the Icache, the Refill Buffer, and the Branch History Table. Several read/write scenarios exist depending on the control signals in Table 6–2; the basic flows will be outlined in this section.

Table 6–2: Icache and Refill Buffer Control Signals

Control Signal	Source	Operation/Notes
Icache Control:		
I%J_IC_CMD_A_H:	IBOX	Indicates Icache transaction
0 READ		Read the Icache data, tags, and the BHT
1 FILL		Fill the Icache data and tags

Table 6–2 (Cont.): Icache and Refill Buffer Control Signals

Control Signal	Source	Operation/Notes
Icache Control:		
I%J_IB_STALL_A_H	IBOX	Hold data at output of RFB and Icache data/tags/BHT
Refill Buffer Control:		
I%J_RFB_RD_IDX_B_H<6:4>	IBOX	RFB Data Index; read this RFB data entry
I%J_RFB_WRITE_A_H	IBOX	Refill Buffer write enable
I%J_RFB_WR_IDX_A_H<2:0>	IBOX	If RFB_WRITE_A_H asserted, write this entry
Control for the Instruction Buffer fill mux:		
I%J_BYPASS_IC_B_H:	IBOX	Instruction Buffer fill mux
0		IB is loaded with data from the Icache
1		IB is loaded with data from the Refill Buffer
Branch History Table Control:		
I%J_IC_CMD_A_H = 0	IBOX	Read the BHT
I%J_HUP_EN_5B_H	IBOX	Write the BHT with updated prediction bits
I%J_BHT_SILO_SEL_B_H:	IBOX	Delay BHT history by 1 cycle for RFB reads
0		Send IBOX history bits just read from BHT
1		Send IBOX history bits piped for one cycle

6.2.4.1 Icache & Refill Buffer Fill Operations

FILL data with longword parity is always received from the Scache on **S%J_IFB_DATA_9B_H<127:0>** and **S%J_IFB_PARITY_9B_H<3:0>**; FILLs from off-chip are written through the Scache to drive the data and parity. The FILL data is piped into an A/B-latch pair and then into the appropriate Refill Buffer Entry, see Figure 6–2. A Refill Buffer entry is written when **I%J_RFB_WRITE_A_H** is received from the IBOX; this signal has been conditioned with **C%I_IFB_DATA_VALID_9A_H** in the IBOX. This allows a cycle to calculate the Predecoded bits and their parity which become part of the data datapath.

Two types of FILLs exist, those returning requested data (Demand FILLs) and those returning the other octaword of the requested block, (non-Demand FILLs). For non-Demand FILLs, the data is only written into the Refill Buffer; meanwhile the IBOX can be probing/reading the Icache and the Refill Buffer and writing the IB (see Section 6.2.4.2).

For Demand FILLs, the FILL data is written into both the Icache and the Instruction Buffer in Stage 0A; this is the cycle after the Refill Buffer is written. To allow the parallel write of the Icache and the IB, the 2:1 Mux ahead of the IB is set to choose Refill Buffer Data: **I%J_BYPASS_IC_B_H** = 1.

If the Instruction Buffer is full, the FILL data is held in the B-latch that follows the Refill Buffer until the IB is ready to receive data. This is accomplished by preventing loads of the B-latch when I%J_IB_STALL_A_H is asserted.

The Icache is also written from the Refill Buffer on reads which miss in the Icache and hit in the Refill Buffer. Since this is initiated by a read transaction, it is described in Section 6.2.4.2. The SROM interface may also be used to write the Icache and the Branch History Table, see Section 6.2.4.1.1.

The Icache tag field is written whenever the data is written; the IBOX provides the tag address, tag parity, both valid bits, and other qualifiers. Since FILLs occur as two separate octaword transactions, the IBOX determines the valid bits using the octaword address and information from the CBOX as to which FILL this is (first or second).

The Refill Buffer tags are maintained in the IBOX; they are also written on FILLs, see Section 1.2.2, Instruction Fetch.

S%J_IFB_DATA_9B_H<127:0> is driven to the Refill Buffer and the Icache before parity checking and ECC error correction are complete. Once the CBOX detects that bad data was written, there will be a machine check. The full error sequence has not been defined, but it is expected that the full Icache will be flushed using I%J_FLUSH_A_H. Handling of bad data in the Refill Buffer is TBD by the IBOX. For a description of ECC and parity error handling, see the Error Handling Chapter in the EV5 CPU Chip/Functional Specification.

6.2.4.1.1 Writing the Icache and Branch History Table with the SROM

The SROM is another source of Icache and Branch History data and tags. Two types of SROM data may be loaded: shifted serial data or serial data that is being held and recirculated at the inputs to the Icache and BHT. The IBOX controls the SROM operation; when in SROM mode, one of the two types of SROM inputs will load the cache tag, data, and the Branch History Table, over-riding the FILL datapath described in Section 6.2.4.1.

6.2.4.2 Icache & Refill Buffer Read Operations

Reads of the Icache are initiated when I%J_IC_CMD_A_H is a READ. The Refill Buffer is always being read via the muxes at its output. If the requested data is in either the Icache or the Refill Buffer, the IB will be written and validated, otherwise a fill request will be issued by the IBOX. The basic read flow is outlined below, see also Section 1.2.2, Instruction Fetch.

On a "new" Read, assume the Icache will hit:

xA: I%J_IC_CMD_A_H is a READ

xB: Icache index received from IBOX (critical path).

I%J_BYPASS_IC_B_H = 0 to choose data from Icache latch.

0A: Icache index decode, wordline drive, and ram cell read.

Refill Buffer tag read in the IBOX.

0B: Icache data, tag, and branch prediction bits read and latched at output of Icache and Branch History Table.

1A: If I%J_IB_STALL_A_H is low then IB-0 or IB-1 is written with the Icache data.

If I%J_IB_STALL_A_H is high then the data is held at the output of the Icache until the IB is not full.

Refill Buffer data read (one cycle behind the Icache).

In the IBOX, I_HIT%RFB_HIT_1A_H and I_HIT%IC_HIT_1A_H calculated.

1B: Refill Buffer data captured in B-latch.

I_HIT%RFB_HIT_1A_H and I_HIT%IC_HIT_1A_H available:

a. On Icache hit: IB has correct data. Proceed with next transaction, if a read, assume the Icache will hit again, i.e. keep I%J_BYPASS_IC_B_H = 0.

b. On Icache miss and Refill Buffer hit:
   — IB has incorrect data, invalidate IB entry.
   — Move I%J_BYPASS_IC_B_H to 1 to choose the IC bypass.
   — 2A: Overwrite the IB with the Refill Buffer data.

     Write the Icache with the Refill Buffer data and tag.
   — Proceed with the next transaction. If a read, assume the Refill Buffer will hit, i.e. keep I%J_BYPASS_IC_B_H = 1. Icache hit will no longer be checked; reads will be taken from the Refill Buffer (IB and Icache are written in 1A) until a Refill Buffer miss occurs and the IBOX requests a FILL. Once the FILL has been requested, a "new" read sequence may be initiated and data will once again be loaded from the Icache assuming Icache hit, i.e. I%J_BYPASS_IC_B_H is changed to a 0. When the fill data comes back, it is processed according to the FILL flows in Section 6.2.4.1.

c. On Icache miss and Refill Buffer miss: IBOX requests a FILL; subsequent reads are "new" reads and will assume Icache hit.

During reads, the IBOX checks the parity read from the Icache data and tag. Parity checking is done in stages 1 and 2. On parity error, the IBOX traps in stage 7.

6.2.4.3 Branch History Table Reads and Writes

The BHT is read during Icache and Refill Buffer reads. Like the Icache data and tag, reads are initiated in phase A and complete in phase B. The IBOX provides the index, I%J_BHT_IDX_ZB_H<12:4>, which corresponds to an Icache or Refill Buffer index, see Section 6.2.2, and the BHT returns 8 history bits to the IBOX. A piped version of the BHT data read in the previous cycle is available for reads of the BHT that correspond to a new Refill Buffer access. (As noted in Section 6.2.4.2, Refill Buffer data is available one cycle after Icache data on Icache miss and RFB hit.) The IBOX asserts I%J_BHT_SILO_SEL_B_H when they want the piped data instead of the new data, see Figure 6–5.

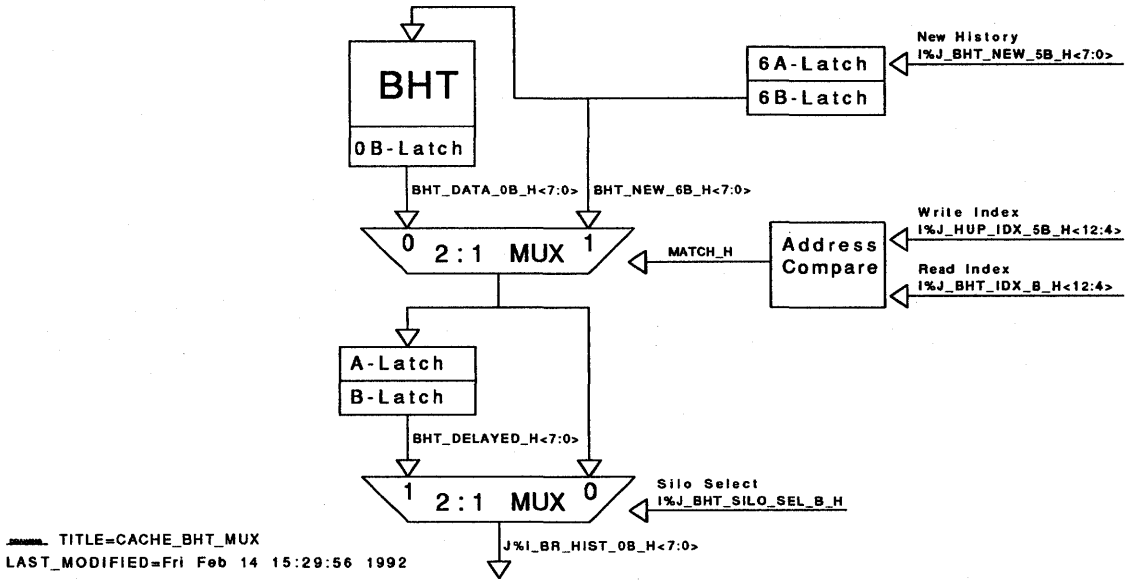
The Branch History Table is not written on FILLs. Once the IBOX finishes the processing of a branch, it recalculates the prediction bits. In stage 5B, the 8 new prediction bits, their index, and the write enable, I%J_HUP_EN_5B_H, are sent to the Branch History Table. In 6B, the Branch History Table entry is written.

The read and write ports of the Branch History Table are separate. Thus during a write, a read may be occurring. If a read and write access the same index; two cases are possible.

- Case 1: A read initiated in phase A follows a write in the previous cycle; the read will get the newly written data.
- Case 2: A read initiated in phase A coincides with a pending write for the following B; the read will get the new "pending" data, not the data currently stored in the BHT.

A Case 1 read is handled with the normal sequencing of BHT reads and writes, i.e. the RAM cells will be written by the time the read starts. Case 2 is satisfied by providing a BHT Bypass using a mux at the BHT output; the new write data is sent back to the IBOX. Figure 6–5 shows the Branch History read path with the bypass mux and the mux to choose the delayed data on Refill Buffer reads.

Figure 6–5: Branch History Table Datapath



6.2.5 Icache Test Operations

Built-in-Self Test (BiST) will be incorporated into the Icache. BiST will provide read and write access (with test patterns) to the Icache data and tag arrays. The SROM interface may also be used to facilitate reads and writes of Icache data, tags, and branch history data.

Note that BiST will probably run while the chip is in reset. This requires certain IBOX/Icache functions to be operable during reset. BiST should clear all the Icache valid bits at the end of the BiST testing.

6.2.6 Icache States Resulting in UNPREDICTABLE operation

- Reading a Branch History Table entry before that entry has been updated or initialized using the SROM will give UNPREDICTABLE history bits.

6.2.7 Icache Redundancy Logic

To increase yield of the Icache array, two extra row pairs are included in the Icache data array. These rows will be programmable, but the specific programming scheme has not been determined. There will be no column redundancy in the Icache data array.

There is no redundancy in the Branch History Table or in the Icache Tag Array.

6.3 DCache Functional Description

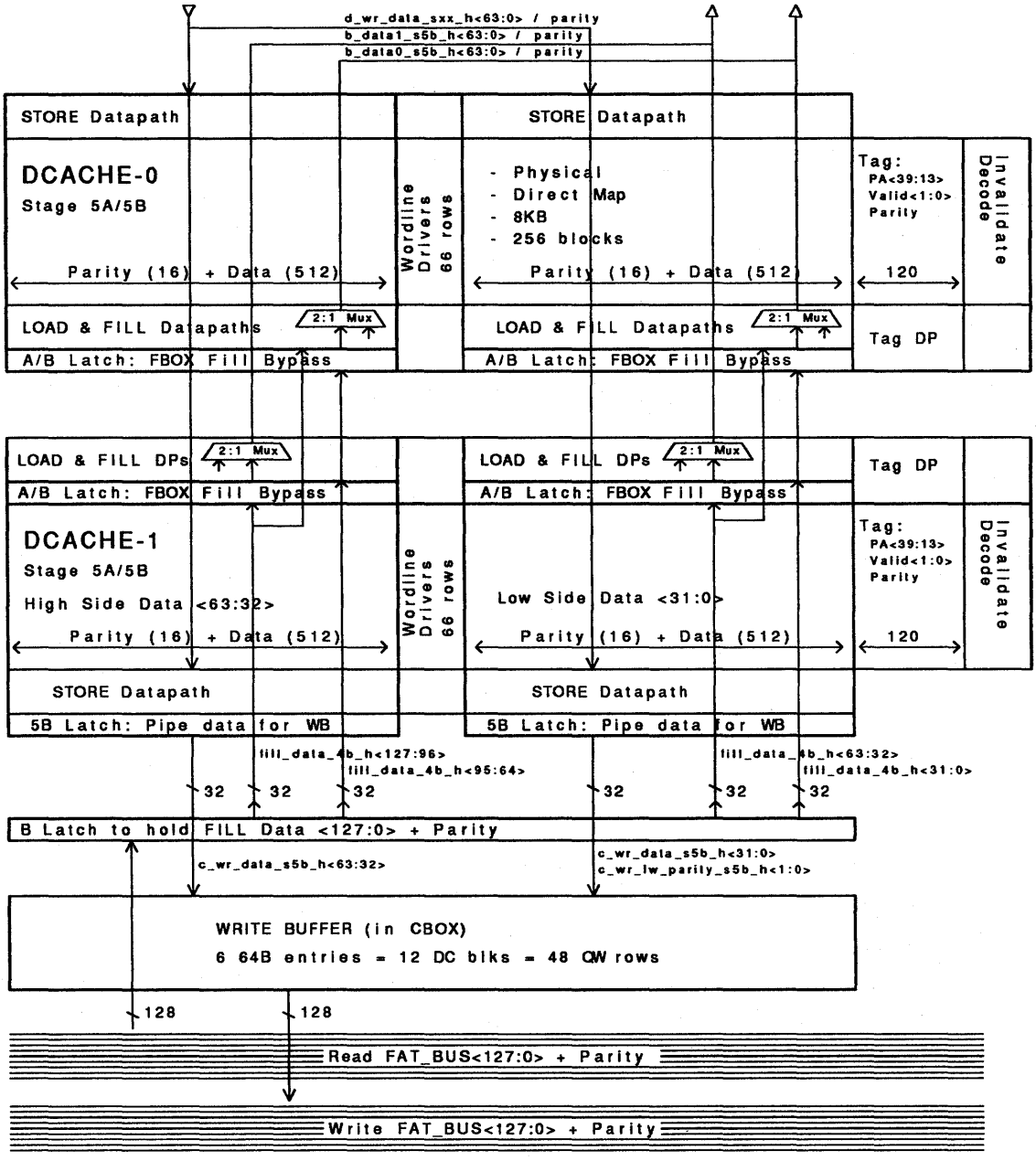
The Dcache is a direct-mapped, write through, physical address cache of D-stream data. It has a one cycle access and a one cycle repetition rate for both reads and writes. The Dcache is comprised of two 8KByte caches, Dcache-0 and Dcache-1; each holds 256 32-byte blocks of data. The two caches allow two read accesses at a time, or a single STORE or FILL access (FILL data may also be returned to the EBOX and FBOX without writing the Dcache, see Section 6.3.2.3). They may be thought of as a single 8K cache which is dual-ported for reads (allows two concurrent LOADs) and single-ported for FILLs or STOREs with the two caches being exact copies of each other. A cache block is filled in two octaword FILL transactions; LOADs to the cache access a quadword of data; STOREs may write a longword or quadword of data. Even longword parity is maintained for the data (8 bits per block), and one bit of even tag parity is maintained for tag bits <38:13> (valid bits are not covered).

The Dcache is maintained as a subset of the Scache. When the Scache replaces a block, an invalidate is sent to the Dcache. The invalidate is done based upon address bits <12:6>. These invalidates correspond to one Scache block (64 bytes) and clear two 32-byte Dcache blocks.

A diagram of the Dcache is shown in Figure 6–6.

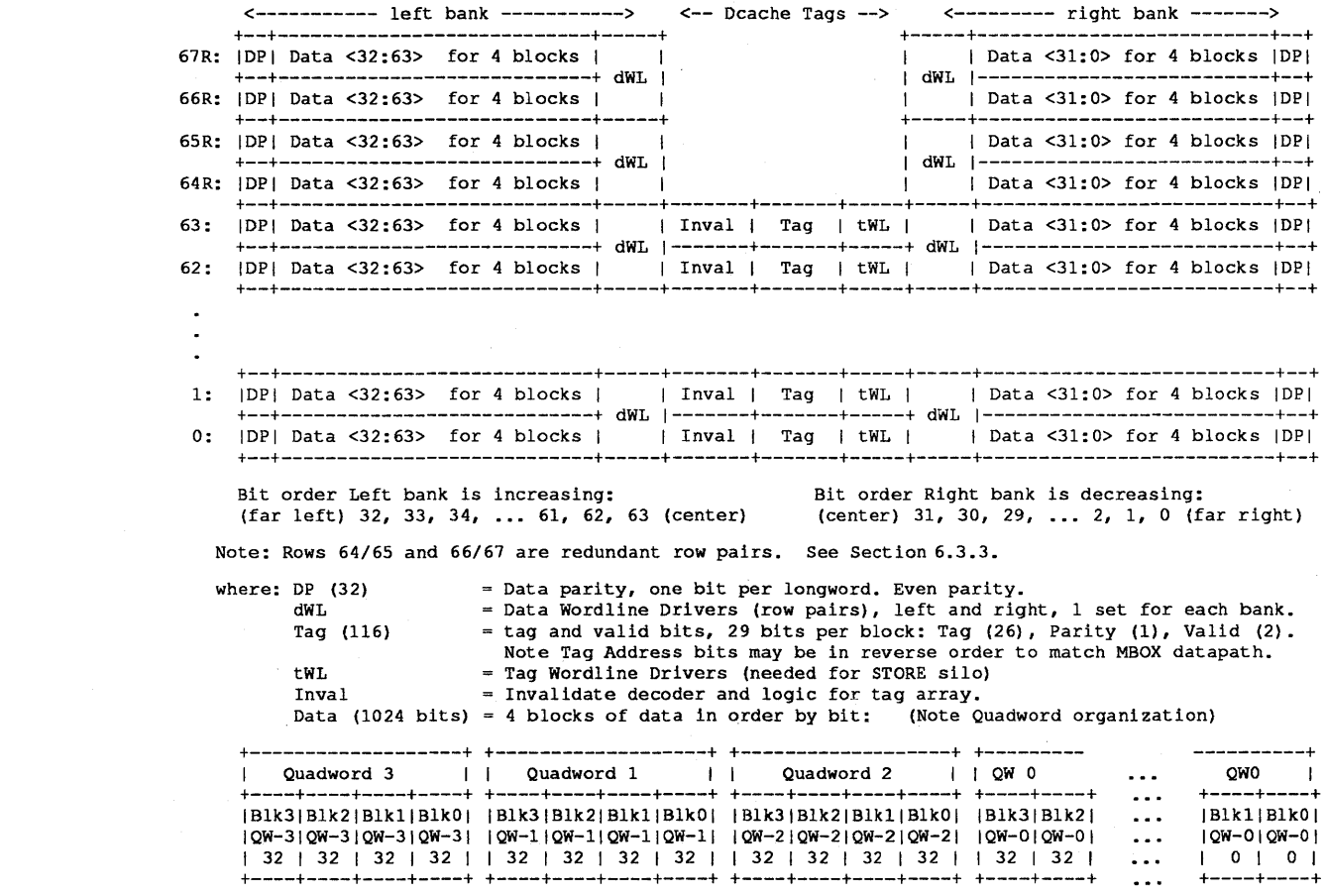
Figure 6–6: Dcache-0 and Dcache-1

NOTE: No SILO or Write-Miss-Invalidate specifics included.
Waiting for decision on which one to implement.



In the diagram below, the logical organization of each Dcache is shown.

Figure 6–7: Logical Dcache Organization



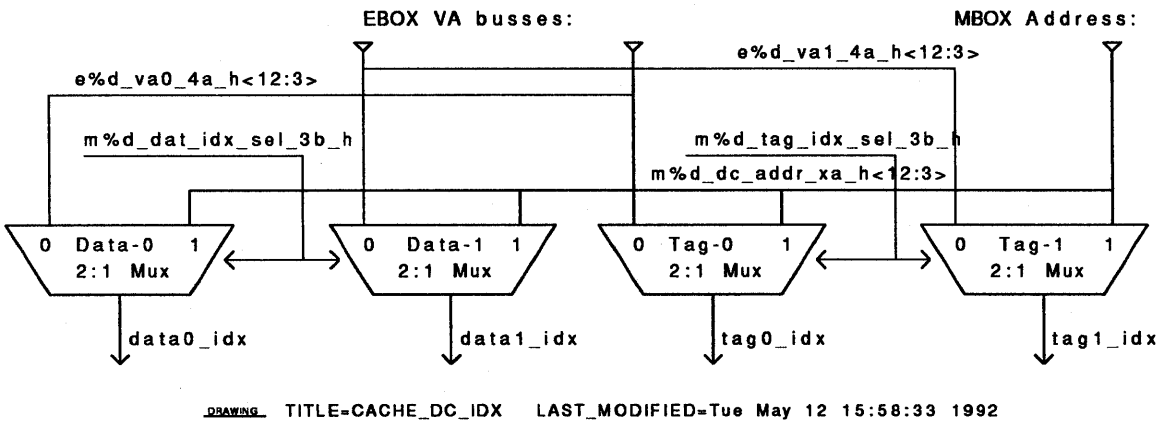
As can be seen from the diagram, each Dcache is organized into 64 direct mapped indexes, where each index consists of four Dcache blocks (or two Scache/Bcache blocks). The breakdown of the Physical address bits for Dcache decoding is show below:

Figure 6–8: Dcache Address Breakdown

```
 38 37 36       14 13|12 11 10 09 08 07|06 05|04 03 02 01 00
+--+--+--+ ... +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 26-bit Tag address | index address    | blk | QW  |LW|     |
+--+--+--+ ... +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

where:  tag address       = Physical address bits loaded into or compared with Tag.
        row index address = addresses 1 of 64 rows:
                              - Index<12:8> decodes one of 32 M3 wordlines (row pairs)
                              - Index<7> drives 1 of 2 M1 wordlines from the M3 wordline.
        blk index address = indicates which block of 4 within an Dcache row.
        QW address        = indicates which Quadword of the block.
        LW address        = indicates which Longword within the QW, used for longword STOREs.
        bits <1:0>        = addressing within the longword, not used by the Dcache.
```

Data in the Dcache is accessed using an address from the MBOX, **M%D_DC_ADDR_XA_H**, or one of the EBOX Virtual Addresses (VA) depending on the type of operation. The MBOX sends the control signals, **M%D_TAG_IDX_SEL_3B_H** and **M%D_DAT_IDX_SEL_3B_H**, telling the Dcache tag array and Dcache data array which address to pick. The address muxing is illustrated in Figure 6–9.

Figure 6–9: Dcache Index Muxing for Data and Tag Arrays



6.3.1 Dcache Initialization and Test

All valid bits in a Dcache are clear when **M%D_DC_FLUSH_A_H** is asserted via the DC_FLUSH IPR, see Section 4.1.10, Mbox and Dcache IPR's. Thus on powerup, **M%D_DC_FLUSH_A_H** should be asserted in order to clear the valid bits.

There are enable and force_hit signals for the Dcache; these are described in Section 4.1.8.6.2.

The Dcache tags are written by FILL operations. They may also be written and read using IPR access, see Section 4.1.10, Mbox and Dcache IPR's.

Bad parity may be written for both the Dcache tag and data parity, see Section 4.1.8.2. Bad parity is written to the Dcache tags using the IPR tag access. Bad parity may be written to the Dcache data parity bits on STOREs; if the MBOX asserts **M%D_FORCE_BAD_PAR_5B_H**, the longword parity bits will be inverted as they are written to the Dcaches. This does not affect the parity sent to the Write Buffer; it will still be correct.

6.3.2 Dcache Transactions

The MBOX sends the Dcache a Tag Command, Data Command, Tag Index Select, and Data Index Select every cycle. The two separate command busses help to facilitate the Write-Silo used for STOREs, Section 6.3.2.2.

The MBOX also sends the Dcache **M%D_UPDATE_DCOUT_3B_H** which the Dcache uses on non-READ commands to decide whether to update the data busses, **D%Z_DATA0_5A_H<63:0>** and **D%Z_DATA1_5A_H<63:0>**. This is a power-savings feature, which prevents the large data busses from changing value when they are not needed, see Section 6.3.2.3.

The following Tables show the results of each tag and data transaction based upon the commands and index selects received. For LOADs and FILLs, the Data and Tag Command are the same; for STOREs which occur as a three cycle operation, the commands may be different.

Table 6–3: Dcache Tag Command and Transactions

Transaction: TAG_CMD_3B_H	Address Select: TAG_IDX_SEL_3B	MBOX Action		Tag Bus: D%M_TAG_5A	Write Tag?
0 0 NOP[1]	Default (DON'T CARE)	NOP	—>	Default[2]	No
0 1 Read	EBOX VA	LOAD	—>	LOAD Tag	No
0 1 Read	EBOX VA	STORE	—>	STORE Tag	No
0 1 Read	MBOX (IPR)	IPR RD	—>	Read Tag	No
0 1 Read	MBOX (BiST)	BiST RD	—>	Read Tag	No
1 0 FILL	MBOX (Fill)	FILL	—>	FILL Tag	w/ FILL[2]
1 1 Write	MBOX (IPR)	IPR WR	—>	DC_TEST_TAG	w/ IPR data[3]
1 1 Write	MBOX (BiST)	BiST WR	—>	BiST pattern	w/ BiST data

[1]NOP may default to Read or may be used to save power.

[2]Tag Bus depends upon what TAG_IDX_SEL_3B the MBOX sends and/or any power-saving logic.

[3]Write with FILL data if (RFB_DATA_VALID_9A * ^NOFILL_5A)

Table 6–4: Dcache Data Command and Transactions

Transaction: DATA_CMD_3B_H	Address Select: DAT_IDX_SEL_3B	MBOX Action	DC Tag Action[1]		Data Bus: D%Z_DATA_5A	Write Dcache?
0 0 NOP	Don't Care	Note[2]	Read	—>	RFB Data[3]	No
0 1 Read	EBOX VA	LOAD	Read	—>	Dcache Data	No
0 1 Read	MBOX (BiST)	BiST RD	BiST RD	—>	Dcache Data	No
1 0 FILL	MBOX (FILL)	FILL	FILL	—>	RFB Data[3]	w/ FILL[4]
1 0 FILL	MBOX (BiST)	BiST WR	BiST WR	—>	RFB Data[3]	w/ BiST patte
1 1 Write	MBOX (Silo)	Note[2]	Read	—>	RFB Data[3]	w/ Silo[6]

[1]Data and Tag commands are independent. This column indicates what the tag action would be.

[2]Possible MBOX actions: NOP, STORE, FILL bypass

[3]RFB data is driven if M%D_UPDATE_DCOUT_3B_H=1, otherwise "old" data remains on the bus, see Section 6.3.2.3. feature as it prevents the data bus from switching when it is unnecessary.

[4]Write with FILL data if (RFB_DATA_VALID_9A * ^NOFILL_5A)

[5]Unconditional write, BiST over-rides RFB_DATA_VALID_9A and NOFILL_5A

[6]Write with Silo data if (D_ST_VALID_6A)

6.3.2.1 Dcache Load Operation

A read of the Dcache occurs whenever the MBOX sends a Read command requested by the EBOX and/or FBOX, to the Dcaches in stage 3B of the pipe, and sets the index selects to choose the EBOX Virtual Address. Late in 4A, the EBOX will send the Virtual address outputs from the EBOX fast adders to each Dcache for the data requested in Pipe-0 and the data requested in Pipe-1. (Note, if a LOAD is requested for one pipe only, the other pipe will be driven (by default) with the data corresponding to its index.)

In stage 4B, the data and tag decoders at each Dcache decode the indices and drive their wordlines enabling the data and tag ram cells to be read. The sense-amps are fired in 5A and a quadword of data with its parity is driven into each data pipe from a 5A latch. **D%Z_DATA0_5A_H<63:0>** is driven from Dcache-0 into Pipe-0 and **D%Z_DATA1_5A_H<63:0>** is driven from Dcache-1 into Pipe-1. With the same timing, **D%M_TAG0_5A_H<38:13>**, **D%M_TAG1_5A_H<38:13>**, tag parity, and valid bits are read from the tag array and driven to the MBOX Dcache Hit logic which is calculated in 5B.

6.3.2.2 Dcache Store Operation

STOREs occur as a three cycle operation using a Write Silo, see Section 4.1.8.2, Dcache STs:

- Stage 4: Read the tag from Dcache-0 that corresponds to the index on EBOX VA-0.
- Stage 5: The MBOX calculates Dcache hit for Dcache-0.
- Stage 6: If the STORE hit in the cache, both Dcache data arrays are written with the STORE data.

To allow back-to-back STOREs, the Dcaches' tag arrays are accessed separately from the data arrays (each cache has a tag index decoder and a data index decoder). This allows for reading of the tag index for the hit calculation while data from the previous STORE is being written into the Dcache, see Table 6–5.

Here's the full STORE sequence by cycle:

3B: MBOX sends a READ command to the Dcache tags.

MBOX sets **M%D_TAG_IDX_SEL_3B_H**=0 to choose EBOX VA-0.

4A: EBOX sends STORE index on VA-0 to Dcache Tag-0.

4B: Dcache tag read.

5A: **D%M_TAG0_5A_H<38:13>** with parity is driven to the MBOX Dcache hit logic.

5B: MBOX calculates Dcache hit for Dcache-0 only.

MBOX sends a write command to Dcache data arrays in preparation for a possible data write in stage 6.

MBOX sets **M%D_DAT_IDX_SEL_3B_H**=1 to choose MBOX silo'd index.

6A: MBOX sends the piped STORE index to both Dcache data decoders.

STORE data with parity from the EBOX or FBOX is sent to the Dcaches.

MBOX enables/disables the data STORE by sending **M%D_ST_VALID_6A_H** based on the results of **M%DC_HIT_E0_5B_H** and IBOX traps.

Dcache-1 buffers the STORE data and forwards it with parity to the CBOX Write Buffer on **D%C_WB_DATA_6A_H<63:0>** and **D%C_WB_LW_PARITY_6A_H<1:0>**.

6B: If **M%D_ST_VALID_6A_H** is high then the STORE data is written into both Dcaches.

Table 6–5 shows an example of three back-to-back STOREs. Note that in this example, the second STORE misses in the Dcache and the data is not written.

Table 6–5: Dcache STORE Silo, Example of 3 back-to-back STOREs at one Dcache

	S4	S5	S6	S7
Tag Command:	Read	Read	Read	X
Tag Index:	Store-1 (EBOX)	Store-2 (EBOX)	Store-3 (EBOX)	X
Data Command:	X	NOP	Write	Write
Data Index:	X	X	Store-1 (MBOX)	Store-2 (MBOX)
D_ST_VALID:	X	X	1 (hit)	0 (miss)
Tag-1:	DC Lookup	DC Hit		
Data-1:	@ EBOX/FBOX	@ EBOX/FBOX	Write DC	
Tag-2:		DC Lookup	DC Miss	
Data-2:		@ EBOX/FBOX	@ EBOX/FBOX	No Write
Tag-3:			DC Lookup	DC Hit
Data-3:			@ EBOX/FBOX	@ EBOX/FBOX

STOREs may write a longword or quadword of data. The Dcache will write a quadword of data when **M%D_WR_TYPE_5B_H** = 1; otherwise a longword of data is written using **M%D_ST_ADR_5B_H<2>** to indicate which longword within the quadword to write. The upper longword within a quadword is in the left bank of each Dcache; the lower longword is in the right bank.

This Write Silo places restrictions on other commands. For instance, FILLs which arrive during the 3-cycle STORE sequence will be bypassed over the cache and written to the register files only. See Section 4.1.8.2 for other command restrictions during the processing of a STORE.

6.3.2.3 Dcache Fill Operation

FILL data is received from the CBOX on **B%Z_RFB_9B_H<127:0>**, the parity arrives a phase later on **C%D_FILL_PAR_10A_H<3:0>**. FILL data is driven to the FBOX and MBOX/EBOX as soon as it arrives at the Dcache. The fill-bypass mux in the Dcache Data LOAD/FILL Datapath, see Figure 6–6, muxes **B%Z_RFB_9B_H<127:0>** with the Dcache read data, latches it, and drives it onto the data bus (if **M%D_UPDATE_DCOUT_3B_H** is asserted, see below). The FILL data is returned during Stage 5A without waiting for the Dcache write. (Note, timing is normalized such that 9B/10A in the Scache/CBOX pipe is equivalent to cycle 4B/5A in the Dcache LOAD pipe). Both data busses are driven with the FILL data even if this is not the requested data. Furthermore, the upper quadword is always driven on Pipe-1 and the lower quadword on Pipe-0:

D%Z_DATA1_5A_H<63:0> = Pipe-1 Data Bus, driven with Fill data <127:64>

D%Z_DATA0_5A_H<63:0> = Pipe-0 Data Bus, driven with Fill data <63:0>.

The FILL parity is not driven to the FBOX, EBOX and MBOX.

The actual write of the Dcache with the FILL data occurs in 5B after the data has already been returned to the FBOX, EBOX and MBOX. **B%Z_RFB_9B_H<127:0>** is piped at the Dcache and the MBOX sends the tag, tag parity, and two valid bits, in stage 5A for the 5B write.

For the FILL to write either Dcache, its NOFILL bit must be inactive and the CBOX must indicate that the FILL data is valid with **C%Z_RFB_DATA_VALID_9A_H**. Since FILLs occur as two separate octaword transactions, the MBOX determines the valid bits using the octaword address and information from the CBOX as to which FILL this is (first or second). Both valid bits are updated on every FILL.

If the above conditions are met, the data and tag are written into the Dcache in stage 5B.

There are restrictions around FILLs occurring with other operations, i.e. LOADs, STOREs. These are detailed in the MBOX specification. Some conflict cases are handled by returning the FILL data to the FBOX, EBOX, and MBOX, and not writing the FILL data into the Dcache. For instance, a FILL can occur with a STORE as long as the FILL data does not write the cache—the fill-bypass mux passes the FILL data onto the data busses, and the STORE data is written into the Dcaches.

The fill-bypass mux is used to pass the RFB data onto the data busses. The data busses may be driven with the RFB data anytime except during Dcache Data Read (i.e. LOAD) operations, but to save power, the data busses, **D%Z_DATA0_5A_H<63:0>** and **D%Z_DATA1_5A_H<63:0>**, are not always updated. If one of the following is true, the data busses will be driven with new data:

1. The Dcache Data command is a READ (drive data busses with Dcache data).
2. M%D_UPDATE_DCOUT_H is a 1 (drive data busses with RFB data).

Otherwise, the data busses will driven with old data (no switching of large devices and loads saves power).

Data from the CBOX is filled in the Dcache(s) before the ECC checking or Scache parity checking has completed. ECC or parity errors discovered on data that has been written to the Dcache are reported by the CBOX and will cause a machine check. The error flow has not been fully determined, but it is likely that the parity error or ECC error will cause the full Dcache to be flushed via the DC_FLUSH IPR. For a description of ECC and parity error handling, see the Error Handling Chapter in the EV5 CPU Chip/Functional Specification.

6.3.2.4 Dcache Invalidate Operation

The types of Dcache invalidates that use the Dcache invalidate port addressed by the CBOX are shown in Table 6–6. All invalidates are based on an index only and clear four valid bits corresponding to a 64-byte Scache block (2 Dcache blocks). For increased performance, invalidates have their own index decoder and occur during the precharge phase of the Dcache; this means they do not consume a cycle and can happen asynchronous to other Dcache operations.

The invalidate command and invalidate address are received from the CBOX in stage 9A: **C%D_INVAL_CMD_9A_H** and **C%D_INVAL_ADDR_9A_H<12:6>**. The invalidate will occur at the Dcache one cycle later in Stage 10A.

Table 6–6: CBOX Initiated Dcache Invalidates

Type	Index Bits	Octawords Cleared	Notes
CBOX Scache	<12:6>	4	Clears 64 bytes.[1]
Any STxC	<12:6>	4	Don't wait for STxC Pass/Fail

[1]Must clear Scache/Bcache block size to keep Dcache subset of Scache.

In addition to the CBOX initiated invalidates, individual Dcache entries may be invalidated by using IPR access to the Dcache tags to clear the valid bit(s) (however, at this point there are no plans for using this mechanism). See the Dcache tag IPRs in Section 4.1.10, Mbox and Dcache IPR's. For a description of ECC and parity error handling, see the Error Handling Chapter in the EV5 CPU Chip/Functional Specification.

As mentioned in Section 6.3.1, the full Dcache may also be invalidated by a write to the DC_FLUSH IPR.

6.3.2.5 Dcache Test Operations

Referring to Table 6–3, the Dcache tag array is accessible via IPR reads and writes processed by the MBOX. If BiST is implemented for the Dcache, the Dcache tags will also be readable and writable using the BiST logic; the BiST tag operation works by using existing MBOX-Dcache signals and busses.

There is no IPR access to the Dcache data arrays. If BiST is implemented for the Dcache, the data arrays may be written using the FILL path as indicated in Table 6–4. In order for the write to occur, a signal indicating that BiST is running will be needed to over-ride the values of **C%Z_RFB_DATA_VALID_9A_H**, **M%D_NOFILL0_5A_H**, and **M%D_NOFILL0_5A_H**. During BiST, the data arrays may be read using the normal data read path.

6.3.3 Dcache Redundancy Logic

To increase yield of the Dcache array, two extra row pairs are included in the Dcache data array. These rows will be programmable, but the specific programming scheme has not been determined. There will be no column redundancy in the Dcache data array.

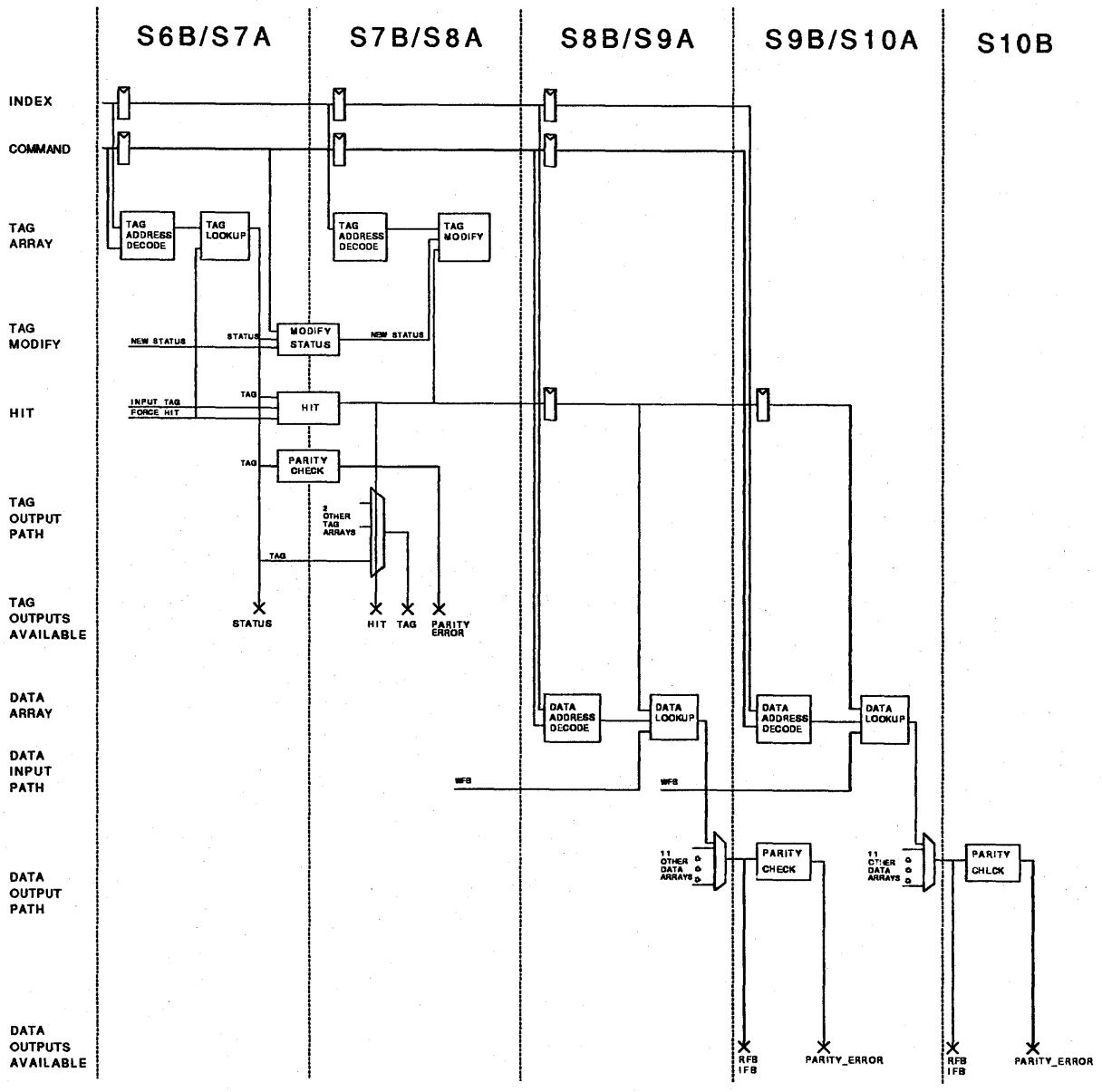
There is no redundancy in the Dcache Tag Array.

6.4 SCache Functional Description

The second-level cache, or SCache, is a 96 KByte cache. It is 3-way set associative and physically addressed. Accesses to the SCache are controlled by the SCache Arbiter, which is part of the Cbox.

The SCache consists of a tag array and a data array. It operates in a pipelined manner.

Figure 6–10: SCache



6.4.1 SCache Tag Array

The SCache Tag Array actually consists of three small arrays, one for each set. Each small array contains 512 tags. Each tag is made up of the upper bits of the physical address, plus some status bits which are used by the Cbox.

Table 6–7: SCache Tag

Name	Extent	Description
Tag<38:15>	34:11	Physical Address
Valid<1:0>	10:9	Valid bits for each 32 bytes of data
Shared<1:0>	8:7	Block is in Shared state: it is also present in another CPU's SCache or BCache. One Shared bit per 32B.
Dirty<1:0>	6:5	Block is in Dirty state: this CPU's copy of this block is more up-to-date than the copy in main memory. One Dirty bit per 32B.
Modified<3:0>	4:1	Block is modified. One Modified bit per octaword.
Parity	0	Even parity over the Tag portion only

Note that status bits are maintained for each 32 bytes of data. This is done to support a block size of 32 bytes, in addition to the native block size of 64 bytes.

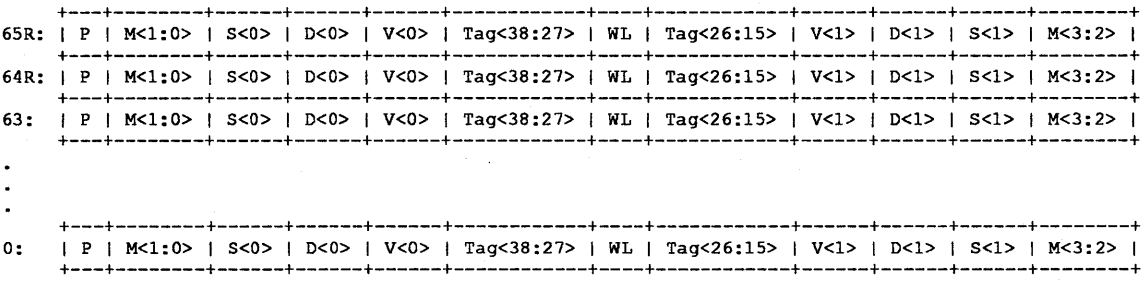
6.4.1.1 Block Size

The SCache's native block size is 64 bytes. A 32 byte block size is also supported, with separate tag status bits for each 32 byte block. However, there is only one tag for each 64 byte block. This tag is shared for two 32 byte blocks: the two 32B blocks must have identical values for physical address bits <38:15>. In the best case, where all blocks have adjacent addresses, the SCache can hold twice as many different blocks in 32B mode as in 64B mode. In the worst case, it holds the same number of blocks in either mode, and therefore half as much actual data in the 32B mode. In general, tag operations look at only the status bits for the addressed 32B. During 64B mode, the Cbox must keep identical both of the Valid, Shared, and Dirty bits for each half of the block.

6.4.1.2 Physical Organization

The physical organization of one tag subarray is described below. The array is made up of 64 rows, plus two redundant rows. Each row contains 8 tags. The eight tags are interleaved on a bit-by-bit basis, i.e. the 'P' field described in Figure 6–11 actually consists of eight 'P' bits, one for each tag in that row.

Figure 6–11: SCache Tag Physical Organization

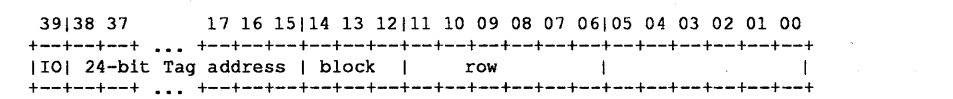


WL = Wordline Decoders/Drivers in center of cache.

Note: Rows 64 and 65 are redundant. See Section 6.4.5.

The following diagram shows how the 40-bit physical address is broken down and used within the SCache Tag Array.

Figure 6–12: SCache Tag Address Breakdown



where: I/O = Indicates whether block is cacheable (=0)
tag address = Physical address bits
block index = Indicates which block of 8 within a row
row index = Addresses 1 of 64 rows
bits <5:0> = Addressing within the block, not used by the Scache tags

Each reference to the SCache begins with parallel lookups of all three SCache Tag sections. The SCache Tag Array is looked up in EV5 pipeline stages S6B/S7A. Each tag section calculates SET_HIT, and drives its SET_HIT signal to its SCache data banks. The SET_HIT signals are also driven to the Cbox. Any modifications to the tag are written back into the array in S7B/S8A.

SET_HIT is asserted if the tag read out of the array matches the input tag driven by the Cbox, and the valid bit for the 32B being addressed is set. The SCache tag section also derives a TAG_MATCH signal for each set, which is asserted if the two tags match, and EITHER valid bit is set. TAG_MATCH is used by the Cbox when in 32B mode, to assist in determining which set to select for a FILL.

In addition to SET_HIT, each tag section calculates BCACHE_INDEX_MATCH. These signals are used by the SCache Allocation logic (see <REFERENCE>(cbox_??)). They indicate when a miss in the SCache will also cause a miss in the BCache (board-level cache). BCACHE_INDEX_MATCH is simply a hit calculation over fewer tag bits. The number of bits compared depends on the BCache size; if there is no BCache, BCACHE_INDEX_MATCH is not calculated. Either of the two valid bits must be set in order to assert BCACHE_INDEX_MATCH.

Table 6–8: BCache Index Match

BCache Size (Megabytes)	Tag Bits Compared
1	Physical Address <38:20>
4	Physical Address <38:22>
8	Physical Address <38:23>
16	Physical Address <38:24>
32	Physical Address <38:25>
64	Physical Address <38:26>

6.4.1.3 Force Hit/Force Miss Conditions

There are several transactions for which the Cbox can force the SET_HITs using a set selection signal. (For more detail on the transactions, see Section 6.4.4). Force hit is also used in testing the SCache.

There is also a force miss mechanism. References to non-cacheable regions of memory must not be stored in the SCache. These blocks all have bit<39> of their physical address equal to 1. Rather than store PA<39> in the SCache tags (since it must always be '0' for cacheable references), the condition of PA<39> equal to 1 is detected and used to force a miss in the SCache.

6.4.1.4 Status Bits

Each tag includes several status bits, described above. These may be modified in S7B/S8A, depending on the command driven by the Cbox. Note that all modifications are done only to the status bits corresponding to the 32B being addressed.

Table 6–9: Tag Modifications

Transaction	Modifies	Hit Condition	Explanation
SC_READ	none		No modifications done on a read.
SC_WRITE	M	SET_HIT . [PRIVATE.DIRTY + WR_ PERMISSION]	Modified bits are set based on which longwords will be written in the SCache.
	D	SET_HIT . S_WR_PERMISSION	Dirty is cleared on every write to Shared data
		SET_HIT . [PRIVATE.DIRTY + P_ WR_PERMISSION]	Dirty is set on every write to Private data
SC_INVAL	V	SET_HIT	Both valid bits cleared
SC_READ_DIRTY	S	SET_HIT	Shared bit set
SC_FILL	all	SET_HIT[1]	Cbox writes new tag entry
SC_SET_SHARED	S	SET_HIT	Shared bit set

[1]Note that SET_HITs may be forced by the SCache Arbiter.

Some status bits are set to fixed values based on the transaction. The following table lists the final value for each portion of the SCache tag at the completion of each transaction type. Modifications are only done to the status bits for the status bits corresponding to the 32B being addressed, unless otherwise noted.

Table 6–10: Final Status Values

Status	Value	Transaction[1]	Note
TAG	-[2]	SC_READ	
	-	SC_WRITE	
	-	SC_INVAL	
	-	SC_READ_DIRTY	
	val[3]	SC_FILL	
	-	SC_SET_SHARED	
V	-	SC_READ	
	-	SC_WRITE	
	0	SC_INVAL	Both valid bits cleared
	-	SC_READ_DIRTY	
	val	SC_FILL	
	-	SC_SET_SHARED	
S	-	SC_READ	
	-	SC_WRITE	Private data
	0	SC_WRITE	Shared data
	-	SC_INVAL	
	1	SC_READ_DIRTY	
	val	SC_FILL	
	1	SC_SET_SHARED	
D	-	SC_READ	
	1	SC_WRITE	Private data
	0	SC_WRITE	Shared data
	-	SC_INVAL	
	-	SC_READ_DIRTY	
	val	SC_FILL	
	-	SC_SET_SHARED	

[1]The transaction is successful: SET_HIT is detected/forced; SC_WRITE performs the write.

[2]- indicates no change was made

[3]"val" is the value driven from the Cbox

Table 6–10 (Cont.): Final Status Values

Status	Value	Transaction[1]	Note
M	-	SC_READ	
	M OR val	SC_WRITE	Private data
	0	SC_WRITE	Shared data
	-	SC_INVAL	
	-	SC_READ_DIRTY	
	val	SC_FILL	
	-	SC_SET_SHARED	
P	-	SC_READ	
	-	SC_WRITE	
	-	SC_INVAL	
	-	SC_READ_DIRTY	
	val	SC_FILL	
	-	SC_SET_SHARED	

[1]The transaction is successful: SET_HIT is detected/forced; SC_WRITE performs the write.

6.4.1.5 Aborting an SCache Reference

Only the SC_WRITE command can be aborted. This can be accomplished in either of two ways: the command can be changed to NOP if there is time to do so, or the abort signal can be asserted. The abort signal is sent in S7A. On an abort no tag modifications are done, and no data array modifications are done.

6.4.1.6 Parity Checking

The SCache tag array also has parity checking logic, which operates over the Tag portion only. Each of the three tag arrays has its own parity checking logic; parity is checked and errors are reported for every access to a valid block (either valid bit set), regardless of hit or miss. See the external spec for details of parity error handling.

6.4.2 SCache Data Array

The SCache is a 96KB, 3-way set associative, physically addressed, write-back cache. It is a unified instruction and data cache. Misses in the ICache and DCache generate accesses into the SCache; misses in the SCache generate accesses off-chip. Transactions generated by the sytem are also processed in the SCache.

The data within the SCache is protected with even longword parity; writes to the SCache can be done to the granularity of a longword.

The SCache data array resides in pipeline stages S8B/S9A and S9B/S10A. Most transactions are done on a 32-byte basis, since that is the block size for the ICache and DCache. These 32-byte transactions are handled as two 16-byte operations. The octaword requested (determined by decoding the address originating in the Mbox, Ibox, or Cbox) is read or written in S8B/S9A; the other half the 32-byte block is processed in S9B/S10A.

The SCache has two major internal buses: the Read SCache bus (RSC), used for reading data from the SCache data array, and the Write SCache bus (WSC), used for writing data into the SCache data array. These internal buses connect to external read and write buses: the RFB, WFB, and IFB. The SET_HIT signals generated in the SCache Tag Array are used to select which set will drive onto/be written from the RSC/WSC.

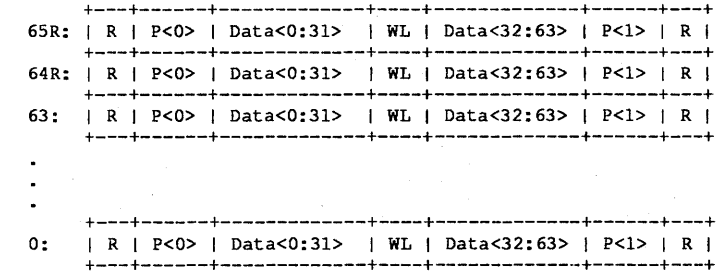
The Mbox and Cbox access the SCache from the Read Fatbus (RFB) and Write Fatbus (WFB), which are separate 128-bit buses. The RFB is used to read data from the SCache; the WFB is used to write data into the SCache. The Ibox accesses the SCache from the ICache Fill Bus (IFB); this bus is used for reading from the SCache.

Control signals from the Cbox are used to enable the SCache to drive onto the RFB and IFB, and to enable the SCache to receive data from the WFB. All of these buses have multiple drivers and/or receivers.

The Cbox checks parity on every read of the SCache data array. See the external spec for details of parity error handling.

The SCache is physically made up of twenty-four 4KB banks. There are 64 rows, plus two redundant rows, in each bank. Each row contains 8 quadwords of data, plus longword parity for those 8 quadwords. To access an entire octaword, two of the 24 banks are looked up, producing one quadword from each of the two banks. The physical organization of one SCache bank is described below. Note that all fields are actually interleaved, as in the tag arrays, i.e. the Data<63> field is actually made up of eight bits: Data<63> for each of the 8 quadwords stored in this row.

Figure 6–13: SCache Physical Organization, Lower Quadword (Right Half of SCache)

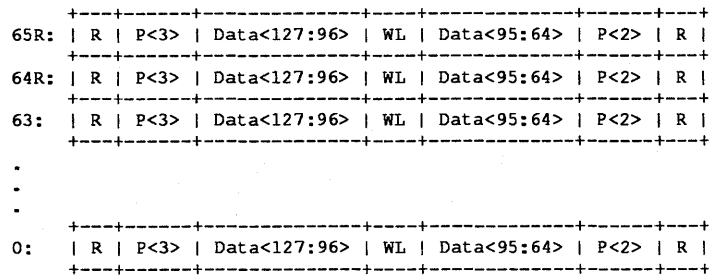


where: R = redundant bitslice (8 columns)
P<1> = longword parity bit for LW1, Data<63:32>
P<0> = longword parity bit for LW0, Data<31:0>
WL = wordline decoders/drivers

Note: Rows 64 and 65 are redundant. See Section 6.4.5.

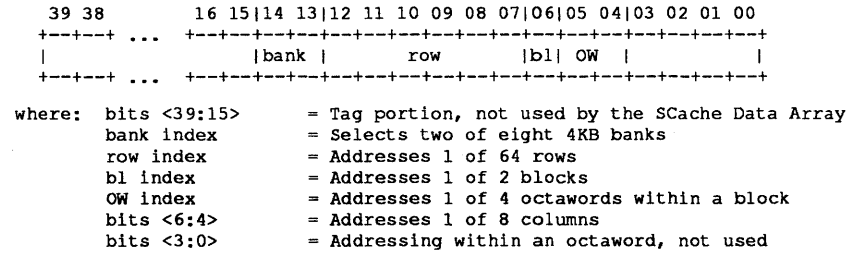
The previous diagram shows storage of the lower half of an octaword. The upper half is stored in another 4KB bank, as shown in the following diagram.

Figure 6–14: SCache Physical Organization, Upper Quadword (Left Half of SCache)



The following diagram shows how the 40-bit physical address is broken down and used within the SCache Data Array.

Figure 6–15: SCache Data Address Breakdown



6.4.3 Pipeline

The SCache is a 4-stage pipeline. Most transactions have a 4-cycle latency: if the tag lookup is done in S6B/S7A, the first octaword of data is output on the bus in S9B/S10A. In general, a new transaction can be started every two cycles. The general SCache flow for a single operation is:

Table 6–11: SCache Pipeline

Stage	Tag Operations	Data Operations
S6B/S7A	Tag Lookup	Index driven to data arrays
	Hit Calculation	
S7B/S8A	Tag Modification	Index decoding
		Hit driven to data arrays
		WRITE data driven to data arrays (1st octaword)
S8B/S9A	[Next Tag Lookup]	Data read/written (1st octaword)
		WRITE data driven to data arrays (2nd octaword)
S9B/S10A		Data read/written (2nd octaword)
		READ data available (1st octaword)
S10B/S11A		READ data available (2nd octaword)

6.4.4 Transactions

The SCache handles six transaction types: SC_READ, SC_WRITE, SC_INVAL, SC_READ_DIRTY, SC_FILL, and SC_SET_SHARED. Read commands are initiated by the Mbox and Ibox; Write commands are initiated by the Mbox; Read Dirty, Fill, Inval, and Set_Shared are initiated by the the system. For more details on interactions between transactions, see <REFERENCE>(cbox_sc_arb).

6.4.4.1 SC_READ

The SCache reads 32 bytes at a time, in two 16-byte transactions. The octaword requested is read first, followed by the other half of the 32-byte block. If there is no SET_HIT detected in the SCache tags, no read is done. For a regular read, SET_HIT is not forced.

This transaction is also used to copy a victim from the SCache into the Cbox. A victim is a block which has been deallocated, but has been modified so it must be written back. In this case, the SCache Arbiter will send the SCache a tag array index plus a set selection signal in order to choose which block is to be removed (SET_HIT will be forced).

An SC_WRITE command which fails becomes an SC_READ command.

No tag status bits are modified for this operation.

Table 6–12: SCache Transactions: SC_READ

	S6B/S7A	S7B/S8A	S8B/S9A	S9B/S10A	S10B/S11A
Tag	Lookup				
Data			Read OW1[1]	Read OW2	
RFB,IFB				OW1	OW2
WFB					

[1]OWx: octaword x

6.4.4.2 SC_WRITE

The SCache writes 32 bytes at a time, in two 16-byte transactions. The octaword requested is written first, followed by the other half of the block. Longword enables are driven to the SCache with each octaword, indicating which longwords are to be written (zero to four enables asserted).

If there is no SET_HIT detected, no write is done. If a SET_HIT is detected, but the status is Private and Not Dirty, no write is done. The Cbox must obtain permission from the system to change the tag status to Dirty before the write can be done. If a SET_HIT is detected, but the status is Shared, no write is done until the Cbox obtains permission from the system.

A write succeeds if:

1. The block status is Private and Dirty.
2. The block status is Private and Clean, and the Cbox asserts a "Set Dirty" permission signal.
3. The block status is Shared, and the Cbox asserts a "Shared Write" permission signal.

Any write which fails is turned into an SC_READ command by the SCache Tag section.

On a write, any longword which is not written (based on the longword enables) is read. This assists the Cbox in accumulating a block which will require an off-chip broadcast.

The SCache tags modify the Shared, Dirty, and Modified status bits only on a successful Write. Shared, Dirty and Modified are cleared on every write to a Shared block. On a write to a Private block, Dirty is set and the new values for the Modified bits are created by OR'ing the previous Modified status bits with the input Modified bits, which are generated from the longword write masks.

Table 6–13: SCache Transactions: SC_WRITE

	S6B/S7A	S7B/S8A	S8B/S9A	S9B/S10A	S10B/S11A
Tag	Lookup	Modify S,M,D			
Data			Write OW1	Write OW2	
RFB,IFB				OW1[1]	OW2[1]
WFB		OW1	OW2		

[1]Unwritten longwords.

6.4.4.3 SC_INVAL

This transaction is used to invalidate a block in the SCache. The only action performed by this transaction is that of clearing both Valid bits corresponding to a 64B block, regardless of actual block size.

This transaction never accesses the SCache data array.

Table 6–14: SCache Transactions: SC_INVAL

	S6B/S7A	S7B/S8A	S8B/S9A	S9B/S10A	S10B/S11A
Tag	Lookup	Modify V			
Data					
RFB,IFB					
WFB					

6.4.4.4 SC_READ_DIRTY

This transaction is used to perform a read that was initiated from off-chip. The status of the block must be changed to Shared. Note that only the Shared bit for the 32 bytes addressed is modified, regardless of actual block size. In 64B mode, the Cbox must eventually do an SC_READ_DIRTY to both halves of the block in order maintain the Shared bit correctly.

SC_READ_DIRTY behaves like a normal Read, in that it operates on a 32-byte piece of data. It returns the requested octaword first, then the other octaword in the 32-byte datum. Two SC_READ_DIRTY commands are required to read an entire 64B block.

Table 6–15: SCache Transactions: SC_READ_DIRTY

	S6B/S7A	S7B/S8A	S8B/S9A	S9B/S10A	S10B/S11A
Tag	Lookup	Modify S			
Data			Read OW1	Read OW2	
RFB,IFB				OW1	OW2
WFB					

6.4.4.5 SC_FILL

Data requested due to a miss in the SCache is written to the SCache using the SC_FILL command. Since this transaction handles data received from the system, the data is handled one octaword at a time. The SCache arbiter prevents CPU access to a block being filled, so no accesses can be done to a partially filled block.

A new tag entry is written on an SC_FILL. The SCache set allocation logic decides which location is to be written. Like the SC_INVAL command, block selection is done by sending the SCache a tag array index and a set selection signal. Four SC_FILL commands are required to write an entire 64B block.

SC_FILL is the only transaction which writes the SCache Tag array in S6B/S7A.

Table 6–16: SCache Transactions: SC_FILL

	S6B/S7A	S7B/S8A	S8B/S9A	S9B/S10A	S10B/S11A
Tag	Write New Tag				
Data			Write OW1		
RFB,IFB					
WFB		OW1			

6.4.4.6 SC_SET_SHARED

This command is used to set the Shared status bit based on commands generated by the system. The tag array is probed, to see if the SCache contains a particular block; on a SET_HIT, the Shared bit is set. This SCache command is used to implement the system command Set_Shared (changing a private block to a shared block).

This transaction never accesses the SCache data array.

Table 6–17: SCache Transactions: SC_TAG_UPDATE

	S6B/S7A	S7B/S8A	S8B/S9A	S9B/S10A	S10B/S11A
Tag	Lookup	Modify S			
Data					
RFB,IFB					
WFB					

6.4.5 SCache Redundancy Logic

The SCache tag array is implemented as 3 banks of approximately 2KB each. Each bank has two extra rows, which are enabled by fuses. The exact fuse usage is TBD.

The SCache data array is implemented as 24 banks of 4KB each. Each bank has two extra rows; each extra row can replace any failing row within the 4KB bank. The extra rows are enabled by fuses: mapping of the extra row to a faulty row is done by programming the failing row's address into the fuses associated with the extra row. Every address driven into the cache bank is compared to the addresses encoded in the mapping fuses; if a match is detected, the extra row is looked up rather than the failing row.

Each 4KB bank also has two sets of 8 redundant columns, one set on each side of the word line drivers. The set of 8 columns can be mapped to any bitslice within its half-bank (any one of 33 other bitslices: 32 data plus one parity). Column redundancy fuses are shared between two 4KB banks, so that one repair can be done within any given half-bank; the same repair will be done in the half-bank immediately below.

The SCache data array can also be disabled a set at a time. If a bank cannot be repaired, its set can be disabled. The SCache will continue to function properly; performance may be reduced. Only one good set is required for correct operation of the SCache.

6.4.6 Cbox Interface

The SCache Arbiter in the Cbox controls the SCache by sending the SCache tag array a command, and with some commands, new data values. Set selection signals are also sent with some commands in order to force SET_HIT. The command and set selection encodings are described below.

Table 6–18: SCache Commands from Cbox

Encoding	Command	Force Hit
000	NOP	11[1]
001	SC_READ	##[3]
010	SC_INVAL	11
011	SC_WRITE	11
100	SC_SET_SHARED	11
101	SC_READ_DIRTY	11
110	Not Used	11
111	SC_FILL	##

[1]No set is caused to "force hit".

[3]For a normal read, Force Hit = 11; for a victim read, Force Hit = ##.

On every tag lookup, status information is driven to the Cbox. The information, and the stage in which it is driven, are listed in the table below.

Table 6–19: Tag Status Driven to Cbox

S7A	S7B
V[2:0]<1:0>[1]	Tag[2]
S[2:0]<1:0>	Parity Error[2:0]
D[2:0]<1:0>	
M[2:0]<3:0>	
SET_HIT[2:0]	
BCACHE_INDEX_MATCH[2:0]	
TAG_MATCH[2:0]	

[1][2:0] indicates this status is driven from all 3 sets

[2] this status is driven from only the set which detected SET_HIT

6.4.7 Ibox Interface

The Ibox interface with the SCache is via the ICache Fill Bus (IFB). For a reference which hits in the SCache, data is driven on the IFB from the RSC. If a reference misses in the SCache, the request is sent off-chip. Fill data returns to the Ibox through the SCache, on the WSC. This fill data is already ECC-corrected. Data from non-cacheable regions of memory are sent through the SCache, but are not written into the SCache.

Fill data to the Ibox must be piped one cycle within the SCache in order to mimic the timing of an SCache hit and prevent collisions on the IFB. This piping is done in the SCache data array, at the Ibox interface.

The SCache uses the longword write signals sent by the Cbox in order to determine whether data to be driven on the IFB is appearing on the RSC or WSC.

6.5 Reset and Initialization

See the External Spec.

6.6 Error Handling and Recording

See the External Spec.

6.7 Test Aspects

The ICache incorporates built-in self-test (BiST). The DCache and SCache are tested via IPRs.

Bad parity may be written to the Dcache tags and data, see Section 6.3.1.

6.7.1 BiST

See the External Spec for a description of BiST in the ICache.

6.7.2 IPR access

The DCache and SCache are accessible for testing via IPRs. Using these access paths, all bits can be tested as desired. These IPRs may also be used in error handling.

6.7.3 Scan Chains

A scan chain is located at the output of the SCache, at its interface to the IFB. There are actually two segments of the scan chain, one over each half of the SCache data array. The signals in order of appearance are:

Position	Description	Comment
	Scan Chain for Right Half of SCache:	
<0>	S%IFB_PAR_H<0>	LW Parity for Data<31:0>
<1>	S%IFB_H<0>	Data<0>
<2>	S%IFB_H<1>	Data<1>
<3>	S%IFB_H<2>	Data<2>
<4>	S%IFB_H<3>	Data<3>
<5>	S%IFB_H<4>	Data<4>
<6>	S%IFB_H<5>	Data<5>
<7>	S%IFB_H<6>	Data<6>
<8>	S%IFB_H<7>	Data<7>
<9>	S%IFB_H<8>	Data<8>
<10>	S%IFB_H<9>	Data<9>
<11>	S%IFB_H<10>	Data<10>
<12>	S%IFB_H<11>	Data<11>
<13>	S%IFB_H<12>	Data<12>
<14>	S%IFB_H<13>	Data<13>
<15>	S%IFB_H<14>	Data<14>
<16>	S%IFB_H<15>	Data<15>
<17>	S%IFB_H<16>	Data<16>
<18>	S%IFB_H<17>	Data<17>
<19>	S%IFB_H<18>	Data<18>
<20>	S%IFB_H<19>	Data<19>
<21>	S%IFB_H<20>	Data<20>
<22>	S%IFB_H<21>	Data<21>
<23>	S%IFB_H<22>	Data<22>
<24>	S%IFB_H<23>	Data<23>
<25>	S%IFB_H<24>	Data<24>
<26>	S%IFB_H<25>	Data<25>
<27>	S%IFB_H<26>	Data<26>
<28>	S%IFB_H<27>	Data<27>
<29>	S%IFB_H<28>	Data<28>
<30>	S%IFB_H<29>	Data<29>
<31>	S%IFB_H<30>	Data<30>
<32>	S%IFB_H<31>	Data<31>
<33>	S_DIR_CTL%LSEL_WSC_H	LW write enable for Data<31:0>
<34>	S_DCR%ADDR_7A_L<14>	Address driven to SCache
<35>	S_DCR%ADDR_7A_L<13>	"

Position	Description	Comment
	Scan Chain for Right Half of SCache:	
<36>	S_DCR%ADDR_7A_L<12>	"
<37>	S_DCR%ADDR_7A_L<11>	"
<38>	S_DCR%ADDR_7A_L<10>	"
<39>	S_DCR%ADDR_7A_L<9>	"
<40>	S_DCR%ADDR_7A_L<8>	"
<41>	S_DCR%ADDR_7A_L<7>	"
<42>	S_DCR%ADDR_7A_L<6>	"
<43>	S_DCR%ADDR_7A_L<5>	"
<44>	S_DCR%ADDR_7A_L<4>	"
<45>	S_DCR%HIT_H<2>	SET_HIT signal, set 2
<46>	S_DCR%HIT_H<1>	SET_HIT signal, set 1
<47>	S_DCR%HIT_H<0>	SET_HIT signal, set 0
<48>	S_DIR_CTL%RSEL_WSC_H	LW write enable for Data<63:32>
<49>	S%IFB_H<32>	Data<32>
<50>	S%IFB_H<33>	Data<33>
<51>	S%IFB_H<34>	Data<34>
<52>	S%IFB_H<35>	Data<35>
<53>	S%IFB_H<36>	Data<36>
<54>	S%IFB_H<37>	Data<37>
<55>	S%IFB_H<38>	Data<38>
<56>	S%IFB_H<39>	Data<39>
<57>	S%IFB_H<40>	Data<40>
<58>	S%IFB_H<41>	Data<41>
<59>	S%IFB_H<42>	Data<42>
<60>	S%IFB_H<43>	Data<43>
<61>	S%IFB_H<44>	Data<44>
<62>	S%IFB_H<45>	Data<45>
<63>	S%IFB_H<46>	Data<46>
<64>	S%IFB_H<47>	Data<47>
<65>	S%IFB_H<48>	Data<48>
<66>	S%IFB_H<49>	Data<49>
<67>	S%IFB_H<50>	Data<50>
<68>	S%IFB_H<51>	Data<51>
<69>	S%IFB_H<52>	Data<52>
<70>	S%IFB_H<53>	Data<53>

Position	Description	Comment
	Scan Chain for Right Half of SCache:	
<71>	S%IFB_H<54>	Data<54>
<72>	S%IFB_H<55>	Data<55>
<73>	S%IFB_H<56>	Data<56>
<74>	S%IFB_H<57>	Data<57>
<75>	S%IFB_H<58>	Data<58>
<76>	S%IFB_H<59>	Data<59>
<77>	S%IFB_H<60>	Data<60>
<78>	S%IFB_H<61>	Data<61>
<79>	S%IFB_H<62>	Data<62>
<80>	S%IFB_H<63>	Data<63>
<81>	S%IFB_PAR_H<1>	LW Parity for Data<63:32>

Position	Description	Comment
	Scan Chain for Left Half of SCache:	
<0>	S%IFB_PAR_H<3>	LW Parity for Data<95:64>
<1>	S%IFB_H<64>	Data<64>
<2>	S%IFB_H<65>	Data<65>
<3>	S%IFB_H<66>	Data<66>
<4>	S%IFB_H<67>	Data<67>
<5>	S%IFB_H<68>	Data<68>
<6>	S%IFB_H<69>	Data<69>
<7>	S%IFB_H<70>	Data<70>
<8>	S%IFB_H<71>	Data<71>
<9>	S%IFB_H<72>	Data<72>
<10>	S%IFB_H<73>	Data<73>
<11>	S%IFB_H<74>	Data<74>
<12>	S%IFB_H<75>	Data<75>
<13>	S%IFB_H<76>	Data<76>
<14>	S%IFB_H<77>	Data<77>
<15>	S%IFB_H<78>	Data<78>
<16>	S%IFB_H<79>	Data<79>
<17>	S%IFB_H<80>	Data<80>
<18>	S%IFB_H<81>	Data<81>
<19>	S%IFB_H<82>	Data<82>
<20>	S%IFB_H<83>	Data<83>
<21>	S%IFB_H<84>	Data<84>

Position	Description	Comment
	Scan Chain for Left Half of SCache:	
<22>	S%IFB_H<85>	Data<85>
<23>	S%IFB_H<86>	Data<86>
<24>	S%IFB_H<87>	Data<87>
<25>	S%IFB_H<88>	Data<88>
<26>	S%IFB_H<89>	Data<89>
<27>	S%IFB_H<90>	Data<90>
<28>	S%IFB_H<91>	Data<91>
<29>	S%IFB_H<92>	Data<92>
<30>	S%IFB_H<93>	Data<93>
<31>	S%IFB_H<94>	Data<94>
<32>	S%IFB_H<95>	Data<95>
<33>	S_DIL_CTL%LSEL_WSC_H	LW write enable for Data<95:64>
<34>	S_DCL%ADDR_7A_L<14>	Address driven to SCache
<35>	S_DCL%ADDR_7A_L<13>	"

Position	Description	Comment
	Scan Chain for Left Half of SCache:	
<36>	S_DCL%ADDR_7A_L<12>	"
<37>	S_DCL%ADDR_7A_L<11>	"
<38>	S_DCL%ADDR_7A_L<10>	"
<39>	S_DCL%ADDR_7A_L<9>	"
<40>	S_DCL%ADDR_7A_L<8>	"
<41>	S_DCL%ADDR_7A_L<7>	"
<42>	S_DCL%ADDR_7A_L<6>	"
<43>	S_DCL%ADDR_7A_L<5>	"
<44>	S_DCL%ADDR_7A_L<4>	"
<45>	S_DCL%HIT_H<2>	SET_HIT signal, set 2
<46>	S_DCL%HIT_H<1>	SET_HIT signal, set 1
<47>	S_DCL%HIT_H<0>	SET_HIT signal, set 0
<48>	S_DIL_CTL%RSEL_WSC_H	LW write enable for Data<127:96>
<49>	S%IFB_H<96>	Data<96>
<50>	S%IFB_H<97>	Data<97>
<51>	S%IFB_H<98>	Data<98>
<52>	S%IFB_H<99>	Data<99>
<53>	S%IFB_H<100>	Data<100>
<54>	S%IFB_H<101>	Data<101>
<55>	S%IFB_H<102>	Data<102>
<56>	S%IFB_H<103>	Data<103>
<57>	S%IFB_H<104>	Data<104>

Position	Description	Comment
	Scan Chain for Left Half of SCache:	
<58>	S%IFB_H<105>	Data<105>
<59>	S%IFB_H<106>	Data<106>
<60>	S%IFB_H<107>	Data<107>
<61>	S%IFB_H<108>	Data<108>
<62>	S%IFB_H<109>	Data<109>
<63>	S%IFB_H<110>	Data<110>
<64>	S%IFB_H<111>	Data<111>
<65>	S%IFB_H<112>	Data<112>
<66>	S%IFB_H<113>	Data<113>
<67>	S%IFB_H<114>	Data<114>
<68>	S%IFB_H<115>	Data<115>
<69>	S%IFB_H<116>	Data<116>
<70>	S%IFB_H<117>	Data<117>
<71>	S%IFB_H<118>	Data<118>
<72>	S%IFB_H<119>	Data<119>
<73>	S%IFB_H<120>	Data<120>
<74>	S%IFB_H<121>	Data<121>
<75>	S%IFB_H<122>	Data<122>
<76>	S%IFB_H<123>	Data<123>
<77>	S%IFB_H<124>	Data<124>
<78>	S%IFB_H<125>	Data<125>
<79>	S%IFB_H<126>	Data<126>
<80>	S%IFB_H<127>	Data<127>
<81>	S%IFB_PAR_H<3>	LW Parity for Data<127:96>

6.8 Performance Monitoring Features

These are TBD. They are likely to include tracking of cache hit rates for various transactions. See the Ibox, Cbox, and Mbox chapters for details.

6.9 Issues

6.9.1 ICache

1. Does loading from the SROM take place while RESET is asserted?
2. Handling of data with parity error or ECC error that gets written into the Icache and/or Refill Buffer. I am assuming that this error flow will be described in the Error Handling Chapter of the EV5 CPU Chip/Functional Specification.

6.9.2 DCache

1. Handling of data with ECC error that gets written into the Dcache. I am assuming that this error flow will be described in the Error Handling Chapter of the EV5 CPU Chip/Functional Specification.

6.9.3 SCache

1. Currently parity is only computed over the address portion of the tag. Should separate parity be computed over the status bits?
2. How are the modified blocks removed during powerfail?
3. Define IPRs.

6.10 Revision History

Table 6–20: Revision History

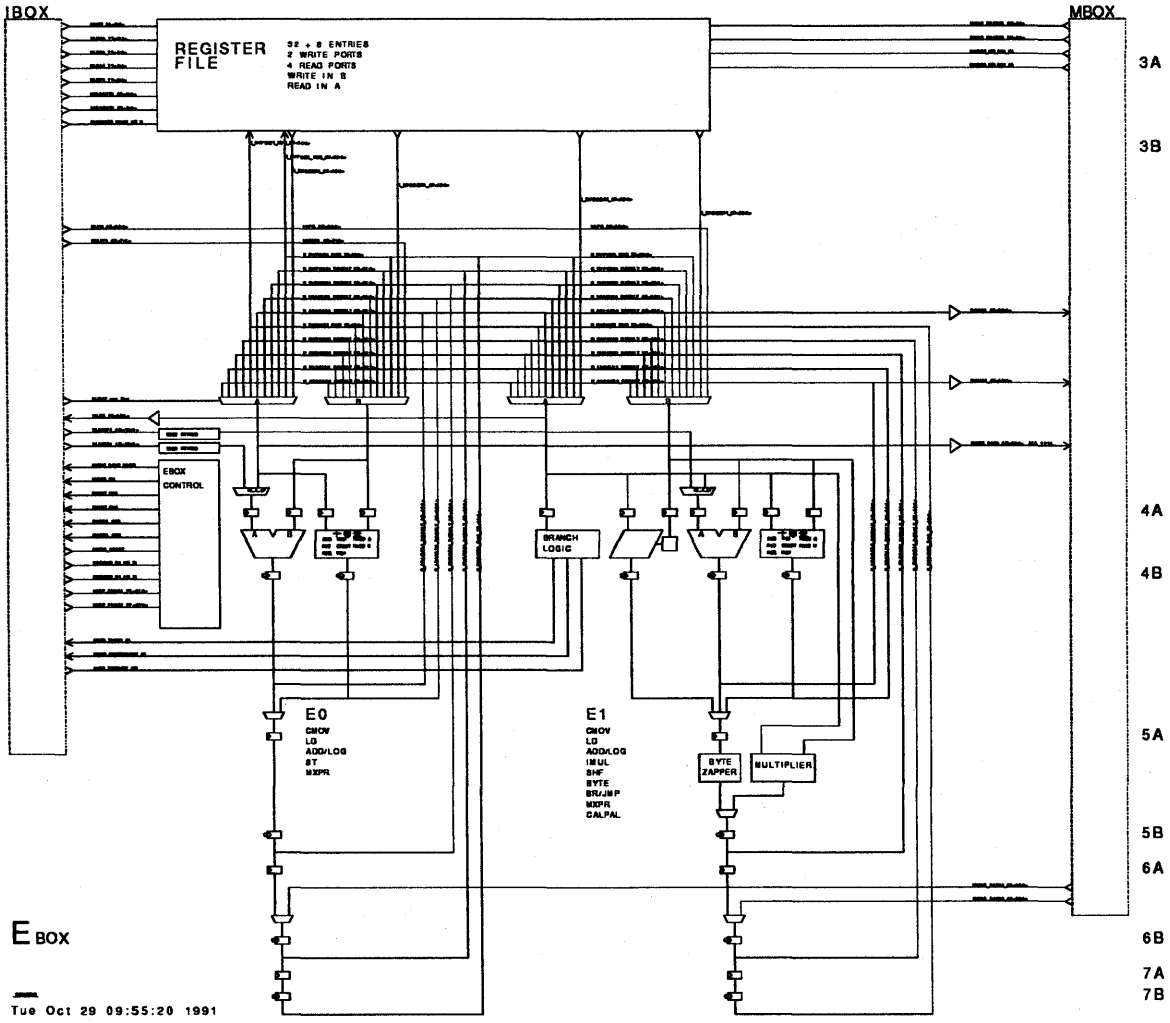
Who	When	Description of change
Elizabeth M. Cooper	2-Dec-1991	Overview, SCache particulars
M. Delaney	17-Dec-1991	Adding to Icache/Dcache sections
M. Delaney	7-Jan-1992	First pass of Icache & Dcache sections done
M. Delaney	10-Jan-1992	Changed Dcache indexing 12:11 –> 6:5 etc.
M. Delaney	13-Jan-1992	Icache section corrections per Mike Smith
M. Delaney	20-Jan-1992	Changed Dcache over to write silo
		Icache: updated for fills coming through Scache
Elizabeth M. Cooper	22-Jan-1992	Scache updates
Elizabeth M. Cooper	3-Feb-1992	Scache command updates
M. Delaney	13-Feb-1992	Icache and Dcache updates
M. Delaney	21-Feb-1992	Minor Dcache updates—Forcing bad parity
M. Delaney	9-Mar-1992	Icache updates: Forcing bad parity; updated RFB logic/timing
M. Delaney	16-Mar-1992	Dcache updates: FILL bypass timing change, DC tag in middle
M. Delaney	12-May-1992	Icache & Dcache updates: new DC timing, no split DC
Elizabeth M. Cooper	26-June-1992	SCache updates

Chapter 7

The Clocks

7.1 Overview-Block Diagram

Figure 7–1: Ebox



7.2 Functional Description

7.3 Reset and Initialization

7.4 Error Handling and Recording

7.5 Test Aspects

7.6 Performance Monitoring Features

7.7 Issues

7.8 Revision History

Table 7–1: Revision History

Who	When	Description of change
your name	date	description

Chapter 8

Test Internals

8.1 Overview

This chapter describes the EV5 CPU chip's testability features and the test port.

8.2 The Testability Strategy

The EV5 CPU chip's testability strategy addresses the broad issue of providing cost-effective and thorough testing during many life cycle testing phases. The strategy specifically implements test features to support

- chip debug
- high fault coverage test at wafer probe and packaged chip test
- support for effective chip burn-in test
- support for efficient embedded RAM testing for laser repair and go/nogo testing.
- support module manufacturing test via IEEE 1149.1 boundary scan architecture
- support for system test via a variety of architectural features.

The strategy uses a combination of a variety of testability techniques and approaches that are best suited to address the specific functional elements in the chip. The cost-effective implementation is realized by the appropriate consideration of the global issues, by unifying the test objectives, by sharing test resources and by exploiting features inherent in the chip. The strategy also relies on leveraging off the design verification patterns in developing production test patterns to meet the fault coverage goals.

The EV5 Testability Micro-Architecture consists of the Test Port and Testability Features. The Test Port implements a comprehensive test access strategy, permiting an efficient access during debug and manufacturing test.

8.3 Test Port

Test Port on EV5 supports a parallel debug port, a serial ROM port, and an IEEE 1149.1 port and a number of miscelleneous test functions through a set of shared test port pins. The test port consists of 13 dedicated test pins. These pins have dual defintions. For normal chip operations, including test operations, the **test_mode_l** pin is connected to ground. All test pins have their normal definitions and functions. When **test_mode_l** is asserted high, the test port becomes an

11-bit wide parallel debug port. This mode is used exclusively for chip/system debugging in chip alone or a prototype system environments.. .
. Table 8–1 summerizes the test pins and their functions.

Table 8–1: EV5 Test Pins

Pin Name	Pin Type	Normal Function	Shared Function
TEST_MODE_H	I, Pull-down	Selects debug port defintion	-
TDI_H	I, Pull-up	IEEE 1149.1 Serial Data Input	**pp_data_h<0>**
TDO_H	O, Tri-state,	IEEE P1149.1 Serial Data Output	**pp_data_h<1>**
TMS_H	I, Pull-up	IEEE 1149.1 Test Mode Select	**pp_data_h<2>**
TCK_H	I, pull-down	IEEE 1149.1 Test Clock	**pp_data_h<3>**
TRST_L	I, pull-up	IEEE 1149.1 Test Reset	**pp_data_h<4>**
TEST_STATUS_H	O	Test status/hand shake for BiST	**pp_data_h<5>**
DISABLE_OUT_l	I, pull-up	Disables all output drivers	**pp_data_h<6>**
SROM_DISABLE_H	I	Serial ROM disable	**pp_data_h<7>**
SROM_CLK_H	O	Serial ROM clock/Tx data	**pp_data_h<8>**
SROM_DATA_H	I	Serial ROM data/Rx data	**pp_data_h<9>**
SROM_OE_L	O	Serial ROM output enable	-
spare test pin	tbd		**pp_data_h<10>**

NOTE

May be the TEST_MODE pin should be renamed DEBUG_MODE pin.

8.4 Parallel Debug Port

This port allows the critical chip nodes to be monitored in parallel. The port consists of 11 output pins and is activated by asserting a high on **TEST_MODE_H** pin.

Signals to be observed on parallel port are selecetd by a *tbd*-bit Debug Port Control register. This register is written by an IPR access. As a back-up, the register may also be set up via the JTAG port. Table 8–2 lists the Parallel Port's configurations.

Table 8–2: Parallel Debug Port Operating Modes

Debug Control Register		Data Pins	
DBG_REG(2:0)	**Port Mode**	**PP_DATA_H<10:0>**	**Signals Observed**
1 1 1	Observe xBOX (Default)	PP_DATA_H(10:i)	internal signal xbox signals
		PP_DATA_H(i-1:0)	more internal x box signals
1 1 0	Observe yBox	PP_DATA_H(10:i)	tbd
		PP_DATA_H(i-1:0)	tbd

Restrictions of parallel debug test port

1. The parallel debug port and the normal test ports are mutually exclusive. That is, neither JTAG nor SROM port could be accessed while the test port is configured as the parallel debug port.
2. The parallel debug port must be activated only after normal power-up and initialiazation of the EV5 chip.
3. When parallel debug port is activated, all inputs corrseponding to normal test input pins are fed with their default values.
4. The test_mode_h pin allows to switch back and forth between the normal test port and the parallel debug port.
5. **Parallel debug port is designed to support chip/system debugging in chip alone or a prototype system environments only.** Some small logic may be required to ensure that there is no interference with other chips connected to the test port.

8.5 SROM Port

The 3-pin SROM port decription

8.6 IEEE 1149.1 (JTAG) Port

The Serial Test Port is a 4-pin test access interface based on IEEE 1149.1 standard. In EV5 this port is used for accessing the internal scan registers, the die identificationr egister, the cache self-test results and *tbd the boundary scan register*. The port supports EXTEST, SAMPLE and BYPASS and a number of *tbd* instructions.

The block diagram of the port logic together with the boundary scan register is shown in Figure 8–1. It consists of the four-wire Test Access Port (TAP), a TAP controller, an instruction register (IR) and a bypass register (BPR).

The five pins in test access port are TDI_H, TDO_H, TMS_H, TCK_H, and TRST_l. These pins conform to all requirements of the standard.

The TAP Controller is a state machine which interprets IEEE 1149.1 protocols received on TMS line and generates appropriate clocks and control signals for the testability features under its jurisdiction.

The Instruction Register resides on a scan path. Its contents are interpreted as test instructions and are used to select the testability modes and features.

The Bypass Register is a 1-bit shift register which provides a single-bit serial connection through the port (chip) when no other test path is selected.

Figure 8–1: IEEE 1149.1 Serial Port (the Basic CTI)

8.6.1 Instruction Register

The JTAG Instruction Register on EV5 CPU consists of *tbd* bits. These bits are interpreted as per Table 8–3 to select and control the operation of EV5 test features. During Capture-IR state, the shift register stage of IR is loaded with data '01'. This automatic load feature is useful for testing the integrety of the JTAG scan chain on module.

Table 8–3: Instruction Register

IR< tbd:0 >	Test Register Selected	Test Instruction/ Operation
tbd	tbd	tbd

More JTAG port description

8.7 Miscellaneous Test Pins

8.7.1 DISABLE_OUT_L

EV5 CPU chip has a dedicated pin **disable_out_l**. When asserted low, the CPU chip tri-states output drivers on all output-only and bidirection pins, except those listed below. When asserted, the pin also forces internally a reset to the EV5 chip.

The only exceptions are the **TDO_H** pin and the clock output pins which are not tristated by the **disable_out_l** pin. Not tristating clock output pins was approved by the stage-1 module test engineers on NVAX.

Leaving out the **TDO_H** pin allows the JTAG circuits to operate while chip tristate is in effect. This affords additonal flexibility for the module manufacturing test. For example, during the interconnection test, the EV5 outputs may be allowed to drive only during the Capture-DR state and kept in tristate in all other states. This can eliminates the effect of shifting patterns, as well as drastically reduces the duration of time for which the drivers may see an interconnect short fault.

The single pin tristate functionality is used only during testing.

8.8 Cache BiST

8.9 Internal Scan Registers

Table 8–4: Internal Scan Register Organization

Scan Chain Name: xyz

Size: b bits

Access Chain Number: ....

Bit #	Signal name	Remarks
0	tbd	tbd
..	..	tbd
31	tbd	tbd

8.10 Internal LFSRs

Table 8–5: Internal LFSR Organization

LFSR Name: xyz
Size: b bits
>left>Feedback polynomial: ....
Access Chain Number: ....

Bit #	Signal name	Remarks
0	tbd	tbd
..	..	tbd
31	tbd	tbd

LFSR Name: xyz
Size: b bits
>left>Feedback polynomial: ....
Access Chain Number: ....

Bit #	Signal name	Remarks
0	tbd	tbd
..	..	tbd
31	tbd	tbd

8.11 Miscellaneous Testability Features

8.12 Issues

1. Should EV5 support SROM disable like EV4 does?

8.13 Revision History

Table 8–6: Revision History

Who	When	Description of change
Dilip Bhavsar	2/13/92	Working draft

Chapter 9

The Interconnect

9.1 EV5CIP.H - the only global interconnect .H file

```
    /*
**
** ev5chip.h
**
** Copyright (c)  1991  by  Digital  Equipment Corporation, Maynard, Mass.  The
** information  in this software is subject to change without notice and should
** not be construed as a commitment by DEC.
**
** @(#) Description
** @(#) ev5chip.h: EV5 Behavior model EV5CHIP header file
**
*/

#define REV_EV5CHIP_H 152

/*
** Revision History:
**
** Rev  Who  Date          Description of Change
** ----------------------------------------------------------------------------------------
** 152 rmf  17-jun-1993   add p->temp_sense
** 151 rmf  07-jun-1993   add signals needed for t_pad.c
** 150 wa   18-may-1993   taking out p->ref_clk_in_l
** 149 wa   17-may-1993   adding clk_mode_h pins
**      took out ev5_addr_h pin definitions, not pins
** 148 rmf  14-may-1993   change i->t_sl_xmit_b_h from bit 31 to a 1 bit signal
** 147  dkb  12-may-1993   Rename t->j_bst_bistdone_b_h to *bist_running_b_l
** 146  wa   11-may-1993   taking out p->vref_h, p->cont_l, p->tristate_l and
**      p->ecl_out_h
** 145  npp  10-may-1993   c->i_perf_mon_in_a_h added
** 144  rom  06-may-1993   adding C->S_WFB_*_7B_H (for moving latch into SCache)
** 143  dkb  03-may-1993   Add OBSA Observe scan macro for use in LFSR lsb.
**                         Removed redundant/unused t->* bist signals.
** 142  vr   27-apr-1993   Add signal i->j_flush_b_h, Raj will remove i->j_flush_a_h later.
** 141  cs   21-apr-1993   c->s_rfb_drive_8b_h
** 140  rwc  15-apr-1993   fix Cbox global signal declarations for Judge
** 139  jem  15-apr-1993   add c->m_wr_64b signal to eventually replace c->m_wr_64B (CHANGO doesn't handle ca
**      deleted m->i_perr_5b_h
** 138  pjb  06-apr-1993   remove fill_done_early, make addr_res_h a 3 bit field
** 137  wa   05-apr-1993   putting in conditional definition of "t" to be "tt"
**      as a work-around to Verilog link conflict
** 136  dkb  01-apr-1993   Add OBLA/V OBLB/V macros. Add t->j*_a_* signals.
** 135  cs   14-mar-1993   new S__SC encodings
** 134 rmf  12-mar-1993   add scache LFSR signals; keep old OBL sigs for now
** 133 rmf  08-mar-1993   change LFSR ctl sigs from A to B, but keep A sigs in model for now
** 132   rpp 07-Mar-93     Moving m->i_fill_valid#_4b_h to 4a (old version won't be deleted until
**    the MBOX updates their code, also adding the new write strobes
    for the LDx ports on the FBOX register file from the IBOX
** 131   cs  07-mar-1993   c->m_return_index is now <2:0>
** 130   vr  25-Feb-1993   Change cbox->ibox error interface signals and adding e->i_mul_ovf_8a_h.
** 129  dkb  23-feb-1993   change t->i sigs to register t->i
```

```
** 128  rmf  17-feb-1993   add multiple ibox reset signals
** 127 rmf  16-feb-1993   add global testability signals
** 126  dkb  16-feb-1993   Correct repair signals from tbox to j box
** 125  rom  15-feb-1993   change two cbox signals from SIGNAL to REGISTER
** 124  npp   9-feb-1993   change ld_merge to ld_alloc
** 123 rmf   3-feb-1993   add support for x_srom and x_ster
** 122  npp   3-feb-1993   add performance counter signals
** 121  pjb   1-feb-1993   move wrt_blk and wrt_blk_lock back to 6 and 7, add read modify lock at E and F
** 120  dkb  28-jan-1993   t->z_jtg_si_to_xxx_a_h changed to si_to_xxx_b_h
** 119  pjb  25-jan-1993   add k->sysclock_in_five_a_h
** 118 rmf  13-jan-1993   observability scan chain signals
** 117  jem  13-jan-1993   Fixing IBOX output declarations for Judge
** 116  jem  11-jan-1993   add m->i_perr_6a_h (m->i_perr_5b_h will be deleted at a future time)
** 115  pjb   8-jan-1993   write_block write_block_lock encoding changed to 14, 15
** 114  rmf  04-jan-1993   change i->t_dbg_data_a_h, remove p_drv->port_mode_drv_ctl
** 113  cs   04-jan-1993   s_set_hit_5b_h
** 112  dkb  28-dec-1992   add jtag global sigbals
** 111 jem  21-dec-1992   add m->e_big_endian_7a_h, change m->i_fill_coming_4a_h from sig to reg
** 111 rmf  17-dec-1992   change p->tristate_l to a sig; eventually get rid of; add i->*_treset_b_l signals
**          change width of test port drive enables
** 110  pjb  07-dec-1992   add c->i_force_time_out_b_h
** 109  rmf  03-dec-1992   add IBOX ICSR test status signals, remove t_pad functionality from c_pad
** 108  pjb  30-nov-1992   add READ_DIRTY_INVAL to the system interface
** 107  rmf  30-nov-1992   update tbox signals
** 106  cs   22-nov-1992   cleanup obsolete c->i sigs
** 105  cs   12-nov-1992   c->s_fill_tag_cmd and s_fill_status are now 1 bit wide
** 104  cs   04-nov-1992   c->m_bogus_lf_8a_h
** 103  dkb  02-nov-1992   Added several TBOX i/f globals in *t, *i. Also s_set_hit_6a_h
** 102 ded  20-oct-1992   Added ebox reset signals and test interface structure, changed timing on integer overflow
** 101  san   8-Oct-1992   Added c->s_flush_b_h
** 100  vr    7-Oct-1992   Added signal i->m_kill_dtbis_4a_h
** 99   mjs   9-sep-1992   change i->j_bypass_ic_b_h to i->j_bypass_ic_a_h.
** 98   vr   24-aug-1992   Replaced m->i_in_tb_flow_e0_5a_h with m->i_in_tb_flow_5a_h
** 97   smb  19-aug-1992   Removed redundant mbox interface signals for global routing
** 97   mkg  19-aug-1992   Removed WMB constant
** 96   ded  10-aug-1992   Added low asserted versions of the e->m_vaX_clk_4b signals for GUIDEWIRE.
** 95   san   6-aug-1992   changed  B__C_RFB_SC_DRV constant to 4 for SCache drive
** 94   mjs   5-aug-1992   Fix missing "end" comment found with guidewire
** 93   rpp  20-jul-1992   Making Judge fixes
** 92   san  17-jul-1992   changed c->m_wr_maf_index from 5 bits to 3 bits
** 91   tb   13-jul-1992   add two pins to p->tag_data for 1MB Bcache
** 90   bbf   8-jul-1992   Add m->i_perr_5b_h for Vidya
** 89   bbf   7-jul-1992   Add drive control states for tri-state pins (p_drv)
** 88   ded   7-jul-1992   Fix Ebox judge warnings
** 87   vr    6-jul-1992   adding interrupt pins from cbox->ibox
** 86   cs    6-jul-1992   slide cbox->ibox timings to 8b. ifb_data to 9b. c->z_alloc_cycle_2a.
** 85   sm    2-jul-1992   swapped the decodings of M__C_DREAD , M__C_LDX_L , M__C_IREF
** 84   pjb   2-jul-1992   judge fixes to the pad signals
** 83 ded   1-jul-1992   Removed old versions of Mbox load data buses.
** 82   rpp  29-jun-1992   Fixing Judge Declarations
** 81   san  28-jun-1992   changed SIGNAL/REGISTER declarations for JUDGE
** 80   rwc  28-jun-1992   add i->c_clr_lock_flg_a_h for clearing lock flag from PAL code
** 79   rwb  21-jun-1992   added m->f_fbox_drv_ena_5a_h, used to control source of b->d_wr_data6a_h
** 78   mkg  17-jun-1992   changed dmm_err from signal to register
** 77   san  04-jun-1992   changed B__C_RFB constants for ADP and BDP drives
** 76 bjb  03-jun-1992   update cbox interface from signal->register
** 75 bjb  28-may-1992   add c->m_sc_hit_7b_h and change bit field of m->c_wr_lw_addr_5b_h
** 74 mkg  28-may-1992   add e->m_va0_clk_4b_h and e->m_val_clk_4b_h
** 73 ded  27-may-1992   Add intr_flag signals for RS/RC instructions
** 72   ded  26-may-1992   Delete obsolete ebox signals
** 71   mkg  26-may-1992   Add new constant, I__WMB
** 70   md   25-may-1992   Added m->d_update_dcout_3b_h.  Used for power savings.
** 69 ded  22-may-1992   Change timing of mul_done_soon to 0a
** 68   md   19-may-1992   Added m->d_st_adr_5b_h<2> (for LW STOREs), removed  m->d_dc_addr_xa_h<2>
** 67   rpp  15-may-1992   Deleting obsolete I/E/F interface signals
** 66   cs   15-may-1992   cbox/dcache interface
** 65   mkg  13-may-1992   add i->m_ex_valid_2b_h and i->m_pal_shadow_en_3a_h
** 64   ded  13-may-1992   ebox updated for new bypass timing
** 63   md   12-may-1992   Added e->d_va0_4a_h<12:3>, e->d_val_4a_h<12:3>, fast adder outputs
** 62   vr   12-may-1992   updated timing on kill signals from ibox to mbox for traps.
** 61   md   11-may-1992   mbox/dcache interface updated for new DC timing
** 60   cs   08-may-1992   cbox/dcache interface, maf_index to/from mbox
** 59   bbf  05-may-1992   add definitions of ev5/system commands, ev5 responses
** 58   vr   04-may-1992   update mbox->ibox trap timings
** 57 tcf  04-may-1992   update load timing to fbox
```

```
     ** 56   cs   01-may-1992   c->s, m->c signals
     ** 55   sch  30-apr-1992   add RFB tristate control
     ** 54   cs   29-apr-1992   cbox->{mbox,scache} interface update
     ** 53   pjb  28-apr-1992   Update external interface signals.
     ** 52   cs   23-apr-1992   New Cbox interface timings to/from Mbox, Scache, Ibox, Dcache
     ** 51   rpp  10-apr-1992   Updated timing on Fill lines between the MBOX and IBOX
     ** 50   tcf  09-apr-1992   changed timing of fbox pipe RF write addresses
     ** 49   BAS  07-apr-1992   changed timing of fbox write strobes
     ** 48   mjs  07-apr-1992   problems with some signal declarations found by running occam.
     ** 47 pjb  07-APR-1992   new chip interface pinout
     ** 46 tcf  07-APR-1992   temporarily fix floating exception signal timing
     ** 45 BAS  02-APR-1992   changed timing of FBOX add pipe exception signals according to new design
     ** 44   rpp  30-Mar-1992   Changing register file timings to match the new design
     ** 43   RPP  27-Mar-1992   Updating a number of IBOX outputs from SIGNAL to REGISTER
     ** 42   BAS  27-mar-1992   add fbox add pipe exception signals
     ** 41 ded  17-mar-1992   Change timing on e->i_mul_done_soon from 2a to 1a
     ** 40   mjs  16-mar-1992   change i->j_rfb_rd_idx_a_h to i->j_rfb_rd_idx_b_h
     ** 39   mjs  12-mar-1992   Fix the fix for m_iref_req_2b_h to m_iref_req_2a_h.
     ** 38   mjs  12-mar-1992   changed/added signals for IBOX/MBOX/CBOX/SCACHE interface.
     ** 37   rpp  12-mar-1992   Fixing syntax error on SIGNAL (sys_fill_end_h, 1);
     ** 36   vr   12-mar-1992   Delete j->i_ic_data_0b_h
     ** 35   ded  12-mar-1992   Change definition of e->i_mul_done_soon_2a_h to REGISTER
     ** 34   md    9-mar-1992   Updated Icache and Dcache interface signals
     ** 33   cs    9-mar-1992   move cbox->mbox retry,index,status to 9a. move cbox->dcache invals to 10a.
     ** 32   jdh   4-mar-1992   Added FBOX div_done_soon signal
     ** 31   ded   2-mar-1992   changed kill_cmov and br signals to REGISTER
     ** 30 dha  27-feb-1992   added pin definitions
     ** 29 cs   24-feb-1992   changed SIGNALW back to BUSW macro (ref: #28 below fixed)
     ** 28 jdh  21-feb-1992   changed BUSW macro to SIGNAL macro [ultrix errors]
     ** 27 tcf  19-feb-1992   changed dcache store data bus and parity to
     **       b->d_wr*; added driver IDs for Fbox and Ebox;
     **       removed old store bus name m->d_wr*
     ** 27 tcf  16-feb-1992   added fbox global signals
     ** 26   cs   14-feb-1992   mbox->cbox commands, cbox->mbox return status
     ** 25   npp  14-feb-1992   Predecodes added
     ** 24   mkg  14-feb-1992   Add mbox signal to abort ebox register writes
     **                         and update other mbox signals.  Also add three
     **                         new ibox trap related signals sent to the mbox.
     **       Remove bit<39> from dcache tag.
     ** 23 md   14-feb-1992   Updated Icache and Dcache global signals
     ** 22 mjs  13-feb-1992   change i->ic_index_xb_h to i->j_ic_index_zb_h.
     ** 21 ded  13-feb-1992   Add the Ebox signals
     ** 20   cs   12-feb-1992   wipe out _s from pipe stage specifications
     ** 19   emc  12-feb-1992   fix scache signals
     ** 18   cs   11-feb-1992   fixed cbox signal names
     ** 17   rp   11-feb-1992   Fixed some problems with the IBOX outputs
     ** 16   jm   07-feb-1992   Changed fill signals from Mbox to Fbox/Ibox
     ** 15   rp   06-feb-1992   Adding Ibox Issue stage outputs
     ** 14   wa   06-feb-1992   Fixing declaration of tag
     ** 13   rpp  02-feb-1992   Adding some useful constants for instruction decoding
     ** 12   wa   21-jan-1992   Changing to single clock
     ** 11   md   20-jan-1992   Update Dcache/MBOX, Dcache/CBOX interfaces for write silo, removed duplicate Scacl
     ** 10 emc  17-jan-1992   Added/corrected scache interface
     ** 09   wa    13-jan-1992  Renamed clocks
     ** 08   jm    9-jan-1992   Changed bitfield widths on Mbox signals, removed d_ and _s from dcache/mbox names
     ** 07   jm    9-jan-1992   Changed Ibox-Mbox trap signal names and timing.  Changed Dcache parity signal tim
     ** 06   md    9-jan-1992   Added mux control signals for ICACHE datapath.  Changed CBOX to DC invalidate add
     **                         d_inval_addr_s8b_h, from full address to index address <12:5>
     ** 05   pb   23-dec-1991   Added the pin bus signals p->
     ** 04   md   20-dec-1991   Added ICACHE interface signals to Ibox, Icache. Changed *sc --> *s, *ic--> *j, *d
-> *d
     ** 03   jm   14-dec-1991   Mbox interface signals to/from Dcache, Ibox, Fbox, Ebox.
     **       added m->c_maf_type_6b_h to cbox-mbox interface
     **       moved m->c_wr_data_s5b_h and parity from mbox to dcache
     **       changed c->i_iref_index_s9a_h to a 2 bit field instead of 1.
     **       added c->i_alloc_cycle_s2b_h to cbox interface
     **       added c->d_fill_data_valid_s4b_h, d_inval_cmd_s8b_h, d_inval_addr_s8b_h to cbox
     ** 02   cs   27-nov-1991   Cbox interface signals to/from Mbox, Scache & Ibox.
     ** 01   wa   25-oct-1991   added two clock pins, moved all pins to ev5sim.h
     ** 00   cs   20-jun-1991   original
     **
     */
```

```
/*
** Hook to load this header file only once.
*/

#ifndef     CCS_EV5CHIP_H_LOADED
#define     CCS_EV5CHIP_H_LOADED


/*
** Hook to allocate storage only once for variables declared in chip.h
*/

#if defined(VMS)
#if defined(CCS_EV5CHIP_ALLOCATE_STORAGE)
#define DECLARE globaldef
#else
#define DECLARE globalref
#endif
#else
#if defined(CCS_EV5CHIP_ALLOCATE_STORAGE)
#define DECLARE
#else
#define DECLARE extern
#endif
#endif

DECL_REV( rev_ev5chip_h, "ev5 chip header", REV_EV5CHIP_H );



/*
** EV5 commands to the system
*/
#define EV5_CMD__NOP                0
#define EV5_CMD__LOCK               1
#define EV5_CMD__FETCH              2
#define EV5_CMD__FETCH_M            3
#define EV5_CMD__MB                 4
#define EV5_CMD__SET_DIRTY          5
#define EV5_CMD__WRITE_BLOCK        6
#define EV5_CMD__WRITE_BLOCK_LOCK   7
#define EV5_CMD__READ_MISS0         8
#define EV5_CMD__READ_MISS1         9
#define EV5_CMD__READ_MISS_MOD0     10
#define EV5_CMD__READ_MISS_MOD1     11
#define EV5_CMD__BCACHE_VICTIM      12
#define EV5_CMD__READ_MS_MOD_LK0    14
#define EV5_CMD__READ_MS_MOD_LK1    15

/*
** System commands to EV5
*/
#define SYS_CMD__NOP                0
#define SYS_CMD__FLUSH              1
#define SYS_CMD__INVALIDATE         2
#define SYS_CMD__SET_SHARED         3
#define SYS_CMD__READ               4
#define SYS_CMD__READ_DIRTY         5
#define SYS_CMD__READ_DIRTY_INV     7

/*
** EV5 responses to system commands
*/
#define EV5_RES__NOP                0
#define EV5_RES__NOACK              1
#define EV5_RES__ACK_SCACHE         2
#define EV5_RES__ACK_BCACHE         3
```

```
/*
** Tri-state pin driver IDs
*/
#define P_DRV__SRC_SYS              1
#define P_DRV__SRC_EV5              2
#define P_DRV__SRC_BC               4
#define P_DRV__SRC_SROM             8
#define P_DRV__SRC_STER             16

/*
**
** Global chipwide (inter-box) constants.
**
*/


/* Clock Inter-box Constants */
/* ------------------------ */


/* Ibox Inter-box Constants */
/* ------------------------ */
/********************************************* ALPHA OPCODES ***********************************************
/*        bits 31:26 of all instruction formats      */
/*****               */

#define I__CALL_PAL 0x00  /* Call_Pal           */
    /* opcodes 01-07 are RESDEC         */
#define I__LDA  0x08 /* LDA           */
#define I__LDAH  0x09 /* LDAH           */
    /* opcode 10 is RESDEC         */
#define I__LDQ_U 0x0b /* LDQ_U          */
    /* opcodes 0c-0e are RESDEC         */
#define I__STQ_U 0x0f /* STQ_U          */
#define I__IARITH 0x10 /* ADDi,SnAddi,SUBi,CMPxx,CMPUxx.CMPBGE       */
#define I__ILOG  0x11 /* AND,BIS,XOR,BIC,ORNOT,EQV,CMOVx       */
#define I__ISHFT 0x12 /* SLL,SRL,SRA,EXTxx,INSxx,MSKxx,ZAP,ZAPNOT      */
#define I__IMUL  0x13 /* MULi,UMULH          */
    /* opcode 14 is RESDEC          */
#define I__VAX_FP 0x15  /* ADDF,ADDG,SUBF,SUBG,CMPGxx,CVTGxx,CVTDG,CVTQF,MULF,MULG,DIVF,DIVG   */
#define I__IEEE_FP 0x16 /* ADDS,ADDT,SUBS,SUBT,CMPTxx,CVTQS,CVTQT,CVTTx,MULS,MULT,DIVS,DIVT   */
#define I__DI_FP 0x17 /* CPYS,CPYSN,CPYSE,FCMOVxx,MT_FPCR,MF_FPCR,CVTQL,CVTLQ     */
#define I__MISC  0x18 /* TRAPB,MB,FETCHx,RPCC,RC,RS        */
#define I__HW_MFPR 0x19 /* HW_MFPR          */
#define I__JSR  0x1a /* JSR           */
#define I__HW_LD 0x1b /* HW_LD          */
    /* opcode 1c is RESDEC         */
#define I__HW_MTPR 0x1d /* HW_MTPR          */
#define I__HW_REI 0x1e /* HW_REI          */
#define I__HW_ST 0x1f /* HW_ST          */
#define I__LDF  0x20 /* LDF           */
#define I__LDG  0x21 /* LDG           */
#define I__LDS  0x22 /* LDS           */
#define I__LDT  0x23 /* LDT           */
#define I__STF  0x24 /* STF           */
#define I__STG  0x25 /* STG           */
#define I__STS  0x26 /* STS           */
#define I__STT  0x27 /* STT           */
#define I__LDL   0x28 /* LDL           */
#define I__LDQ   0x29 /* LDQ           */
#define I__LDL_L 0x2a /* LDL_L          */
#define I__LDQ_L 0x2b /* LDQ_L          */
#define I__STL  0x2c /* STL           */
#define I__STQ  0x2d /* STQ           */
#define I__STL_C 0x2e /* STL_C          */
#define I__STQ_C 0x2f /* STQ_C          */
#define I__BR  0x30 /* BR           */
#define I__FBEQ  0x31 /* FBEQ           */
#define I__FBLT  0x32 /* FBGE           */
#define I__FBLE  0x33 /* FBGT           */
#define I__BSR   0x34 /* BSR           */
#define I__FBNE  0x35    /* FBNE           */
#define I__FBGE  0x36 /* FBGE           */
#define I__FBGT  0x37 /* FBGT           */
#define I__BLBC  0x38 /* BLBC           */
```

```
#define I__BEQ   0x39 /* BEQ            */
#define I__BLT   0x3a /* BLT           */
#define I__BLE   0x3b /* BLE            */
#define I__BLBS  0x3c /* BLBS           */
#define I__BNE   0x3d /* BNE            */
#define I__BGE   0x3e /* BGE            */
#define I__BGT       0x3f /* BGT           */
/****************************** end of Alpha Opcodes ********************************************************

/***************************** MISC (OPC18) Instruction Function Fields ***************************************
/**** Bits 15:0 of the Memory Format MISC Instructions                ****/
/***  I'm only specifying the top 4 bits (all others MBZ              ****/
/****  is this legal with the SRM???                  ****/
#define I__DRAINT 0x0 /* DRAINT          */
    /* 1-3 are not used         */
#define I__MB  0x4 /* MB           */
    /* 6-7 are not used         */
#define I__FETCH 0x8 /* FETCH          */
    /* 9 is not used         */
#define I__FETCHM 0xa /* FETCHM          */
       /* b is not used         */
#define I__RPCC  0xc /* RPCC           */
    /* d is not used         */
#define I__RC  0xe /* RC           */
#define I__RS  0xf /* RS           */
/****************************** end of the MISC Function Codes ************************************************

/********************************** Instruction Formats ***************************************************
/****                          ****/
#define I__OPC_H 0x1f    /* bit position of the top of the opcode field for all formats     */
#define I__OPC_L 0x1a /* bit position of the bottom of the opcode field for all formats     */

#define I__RA_H  0x19 /* bit position of the top of the Ra field, for all applicable formats      */
#define I__RA_L  0x15 /* bit position of the bottom of the Ra field, for all applicable formats      */

#define I__RB_H  0x14 /* bit position of the top of the Rb field, for all applicable formats      */
#define I__RB_L  0x10 /* bit position of the top of the Rb field, for all applicable formats      */

#define I__RC_H  0x04 /* bit position of the top of the Rc field, for all applicable formats      */
#define I__RC_L  0x00 /* bit position of the top of the Rc field, for all applicable formats      */

#define I__MEM_DSP_H 0x0f /* bit position of the top of the displacement field for memory format    */
#define I__MEM_DSP_L  0x00 /* bit position of the bottom of the displacement field for memory format  */

#define I__BRA_DSP_H 0x14 /* bit position of the top of the displacement field for branch format   */
#define I__BRA_DSP_L 0x0f /* bit position of the bottom of the displacment field for branch format  */

#define I__LIT_H 0x14 /* bit position of the top of the literal field for operate format   */
#define I__LIT_L 0x0d /* bit position of the bottom of the literal field for operate format   */

#define I__LIT_BIT 0x0c /* bit determining whether to use the literal or B field in operate format  */

#define I__OP_FCN_H   0x0b /* bit position of the top of the function field for operate format   */
#define I__OP_FCN_L    0x05 /* bit position of the bottom of the function field for operate format   */

#define I__FOP_FCN_H 0x0f /* bit position of the top of the function field for floating operate format  */
#define I__FOP_FCN_L  0x05 /* bit position of the bottom of the function field for floating operate format  */

#define I__PAL_H 0x19 /* bit position of the top of the PAL function field in the pal format    */
#define I__PAL_L  0x00 /* bit position of the top of the PAL function field in the pal format    */
/****                          ****/
/************************************  end of Instruction Formats ***************************************************

/*****************************************************************************************************************
/****                          ****/
/****      Some other useful bit fields for the IBOX decoding            ****/
#define I__LATENCY  0x07 /* Position of the bit defining the latency for opcode 11.  If this bit is set,
*     we have a CMOV which has 2 cycle latency rather than 1    */

#define I__FMUL  0x02 /* Value of the bottom 4 bits of the functioin field for FMULs    */

#define I__FDIV  0x03 /* Value of the bottom 4 bits of the function field for FDIVs    */

#define I__CPYS  0x020 /* Value of the 15 bit function field for CPYS      */
/****************************************   end of useful bit fields ***************************************************

/********************************** 5 bit Instruction Predecodes ***************************************************
```

```
#define I__SP_HW_REI_TYPE 0x00
#define I__BR_TYPE  0x01
#define I__HW_REI_RET_TYPE 0x02
#define I__JMP_TYPE  0x03
#define I__CALL_PAL_TYPE 0x04
#define I__BSR_TYPE  0x05
#define I__JSR_COR_TYPE  0x06
#define I__JSR_TYPE  0x07
#define I__I_COND_BR_TYPE 0x08    /* Actually it is 0x08 and 0x09 */
#define I__F_COND_BR_TYPE 0x18    /* Actually it is 0x18 and 0x19 */

/************************************ end of Predecodes ************************************************

/* Cbox Inter-box Constants */
/* ------------------------ */

/* Cbox read fat bus driver ID */
#define B__C_RFB_BDP_DRV 1     /* constant when BDP is driving its IPR's or FILL data onto RFB */
#define B__C_RFB_ADP_DRV 2     /* constant when ADP is driving its IPR's onto RFB */

/* return status to Mbox */

#define C__M_NOP  0x00 /* nop   */
#define C__M_FIRST_FILL  0x01 /* first fill */
#define C__M_LAST_FILL   0x02 /* last fill  */
#define C__M_WR_DONE  0x03 /* write done  */
#define C__M_FETCH_DONE  0x04 /* fetch */
#define C__M_MB_DONE  0x05 /* memory barrier done  */
#define C__M_ECC_FILL          0x06    /* corrected ecc_fill */
#define C__M_STX_C_DONE  0x07 /* store conditional */


/* Ebox Inter-box Constants */
/* ------------------------ */

/* EBOX store bus driver ID */
#define B__DC_STR_EBOX 2


/* Fbox Inter-box Constants */
/* ------------------------ */

/* FBOX store bus driver ID */
#define B__DC_STR_FBOX 1


/* Mbox Inter-box Constants */
/* ------------------------ */

/* mbox commands to cbox */

#define M__C_NOP 0x00 /* nop */

#define M__C_DREAD 0x04 /* dref read */
#define M__C_LDX_L 0x05 /* load locked from memory */
#define M__C_IREAD 0x06 /* iref read */

#define M__C_FETCH 0x08 /* fetch from memory */
#define M__C_FETCH_M 0x09 /* fetch with modify intent from memory */
#define M__C_MB  0x0a /* memory barrier */

#define M__C_WR  0x0c /* write 32B block */
#define M__C_STX_C 0x0d /* store conditional to memory */

/* Dcache Inter-box Constants */


/* Scache Inter-box Constants */
/* -------------------------- */

/* Scache read fat bus driver ID */
#define B__C_RFB_SC_DRV 4
```

```
/* Scache commands */
#define S__SC_NOP  0x00
#define S__SC_READ  0x01
#define S__SC_FILL  0x02    /* unused */
#define S__SC_WRITE  0x03
#define S__SC_INVAL  0x04
#define S__SC_SET_SHARED 0x06
#define S__SC_READ_DIRTY 0x05
#define S__SC_TEST_WRITE 0x07

#if 0
/* Scache commands OLD SCHEME */
#define S__SC_NOP  0x00
#define S__SC_READ  0x01
#define S__SC_INVAL  0x02
#define S__SC_WRITE  0x03
#define S__SC_SET_SHARED 0x04
#define S__SC_READ_DIRTY 0x05
#define S__SC_TEST_WRITE 0x06
#define S__SC_FILL  0x07
#endif


/* Observability LFSR MACROS
*==============================
**
** The following 4 LFSR Macros may be used for modelling LFSRs implemented with
** the OBLA* and OBLB* cells from the EV5 structure Library.
**
** The first two Macros OBLA and OBLAV model OBLA cells used for capturing
** _B signals. OBLA represents a signle bit of LFSR and can be used when
** LFSR is stiched to capture random scattered isolated signals. OBLAV may
** be used when capturing groups of signals (buses etc).
**
** The OBLB and OBLBV are similar macros used for modelling
** OBLB* cells that observe _A signals.
**
** See examples in t_lfs.c to see how to connect these macros.
*/

/*
** macro:
** OBSA Represents a single Observability Scan Register bit
** Parameters:
** obs_b_h CONTROL INPUT. CONNECT SIGNAL THAT ENABLES CAPTURE ACTION
** pi_b_h PARALLEL INPUT. CONNECT DATA TO BE OBSERVED OR FEEDBACK
** si_h SERIAL INPUT: CONNECT SERIAL OUT FROM PREVIOUS STAGE OR
**  PREVIOUS LFSR.
** lat_a_h _A LATCH. REGISTER DECLARATION IN .H FILE.
** lat_b_h _B LATCH. REGISTER DECLARATION IN .H FILE.
**
** Note: pi_b_h, lat_a_h, lat_b_h are multi bit declarations in .h file
*/

#define OBSA( obs_b_h, pi_b_h, si_h, lat_a_h, lat_b_h )\
{\
    if ( k->clock )\
    {\
 if ( obs_b_h )\
     lat_a_h = pi_b_h ;\
 else\
     lat_a_h = si_h ;\
    }\
\
    if ( !k->clock )\
 lat_b_h = lat_a_h ;\
}
```

```
/*
** macro:
** OBLA Represents a single bit of LFSR
** Parameters:
** obs_b_h CONTROL INPUT. CONNECT SIGNAL THAT ENABLES CAPTURE ACTION
** pi_b_h PARALLEL INPUT. CONNECT DATA TO BE OBSERVED. OR FEEDBACK
** si_h SERIAL INPUT: CONNECT SERIAL OUT FROM PREVIOUS STAGE OR
**  PREVIOUS LFSR.
** lat_a_h _A LATCH. REGISTER DECLARATION IN .H FILE.
** lat_b_h _B LATCH. REGISTER DECLARATION IN .H FILE.
*/
#define OBLA( obs_b_h, pi_b_h, si_h, lat_a_h, lat_b_h )\
{\
\
    if ( k->clock )\
    {\
 if ( obs_b_h )\
     lat_a_h = pi_b_h ^ si_h ;\
 else\
     lat_a_h = si_h ;\
    }\
\
    if ( !k->clock )\
 lat_b_h = lat_a_h ;\
}

/*
** macro:
** OBLB  Represents multiple bits of LFSRs
** Parameters:
** obs_a_h CONTROL INPUT. CONNECT SIGNAL THAT ENABLES CAPTURE ACTION
** pi_a_h PARALLEL INPUT. CONNECT DATA TO BE OBSERVED OR FEEDBACK
** si_h SERIAL INPUT: CONNECT SERIAL OUT FROM PREVIOUS STAGE OR
**  PREVIOUS LFSR.
** lat_a_h _A LATCH. REGISTER DECLARATION IN .H FILE.
** lat_b_h _B LATCH. REGISTER DECLARATION IN .H FILE.
** Note: OBLB* cells should not receive the feedback.
*/
#define OBLB( obs_a_h, pi_a_h, si_h, lat_a_h, lat_b_h )\
{\
    if ( k->clock )\
 lat_a_h = si_h ;\
\
    if ( !k->clock )\
    {\
 if (obs_a_h )\
     lat_b_h = pi_a_h ^ lat_a_h ;\
 else\
     lat_b_h = lat_a_h ;\
    }\
}

/*
** macro:
** OBLAV Represents multiple bits of LFSRs
** Parameters:
** obs_b_h CONTROL INPUT. CONNECT SIGNAL THAT ENABLES CAPTURE ACTION
** pi_b_h PARALLEL INPUT. CONNECT DATA TO BE OBSERVED OR FEEDBACK
** si_h SERIAL INPUT: CONNECT SERIAL OUT FROM PREVIOUS STAGE OR
**  PREVIOUS LFSR.
** lat_a_h _A LATCH. REGISTER DECLARATION IN .H FILE.
** lat_b_h _B LATCH. REGISTER DECLARATION IN .H FILE.
** hbit HIGH BIT
** lbit LOW BIT
**
** Note: pi_b_h, lat_a_h, lat_b_h are multi bit declarations in .h file
*/
```

```
#define OBLAV( obs_b_h, pi_b_h, si_h, lat_a_h, lat_b_h, hbit, lbit )\
{\
    if ( hbit > lbit )\
    {\
 if ( k->clock )\
 {\
     if ( obs_b_h )\
     {\
  if ( lbit == 0 )\
      INSV( lat_a_h, hbit, lbit, pi_b_h ^ ((EXTV( lat_b_h, hbit-1,lbit) << 1) | (si_h & 0)));\
  else\
      INSV( lat_a_h, hbit, lbit, pi_b_h ^ EXTV( lat_b_h, hbit - 1, lbit - 1));\
     }\
     else\
     {\
  if ( lbit == 0 )\
      INSV( lat_a_h, hbit, lbit, ((EXTV( lat_b_h, hbit - 1, lbit) << 1 ) | ( si_h & 1 )));\
  else\
      INSV( lat_a_h, hbit, lbit, EXTV( lat_b_h, hbit-1, lbit - 1));\
     }\
 }\
 if ( !k->clock )\
     INSV( lat_b_h, hbit, lbit, EXTV( lat_a_h, hbit, lbit ));\
    }\
    else if ( hbit == lbit )\
    {\
 if ( k->clock )\
 {\
     if ( obs_b_h )\
     {\
  if ( lbit == 0 )\
      INS( lat_a_h, hbit, pi_b_h ) ;\
  else\
      INS( lat_a_h, hbit, pi_b_h ^ EXT( lat_b_h, hbit - 1)) ;\
     }\
     else\
     {\
  if ( lbit == 0 )\
      INS( lat_a_h, hbit, si_h ) ;\
  else\
      INS( lat_a_h, hbit, EXT( lat_b_h, hbit - 1 )) ;\
     }\
 }\
 if ( !k->clock )\
     INS( lat_b_h, hbit, EXT( lat_a_h, hbit));\
    }\
}
/*      INSV( lat_a_h, hbit, lbit, pi_b_h ^ ((EXTV( lat_b_h, hbit-1,lbit) << 1) | (si_h & 1)));\ */
```

```
/*
** macro:
** OBLBV Represents multiple bits of LFSRs
** Parameters:
** obs_a_h CONTROL INPUT. CONNECT SIGNAL THAT ENABLES CAPTURE ACTION
** pi_a_h PARALLEL INPUT. CONNECT DATA TO BE OBSERVED OR FEEDBACK
** si_h SERIAL INPUT: CONNECT SERIAL OUT FROM PREVIOUS STAGE OR
**  PREVIOUS LFSR.
** lat_a_h _A LATCH. REGISTER DECLARATION IN .H FILE.
** lat_b_h _B LATCH. REGISTER DECLARATION IN .H FILE.
** hbit HIGH BIT
** lbit LOW BIT
**
** Note: pi_a_h, lat_a_h, lat_b_h are multi bit declarations in .h file
**       OBLB* cells should not receive the feedback.
**
*/
#define OBLBV( obs_a_h, pi_a_h, si_h, lat_a_h, lat_b_h, hbit, lbit )\
{\
    if ( hbit > lbit )\
    {\
 if ( k->clock )\
 {\
    if ( lbit == 0 )\
  INSV( lat_a_h, hbit, lbit, ((EXTV( lat_b_h, hbit - 1, lbit) << 1 ) | ( si_h & 1)));\
    else\
  INSV( lat_a_h, hbit, lbit, EXTV( lat_b_h, hbit - 1, lbit - 1) );\
 }\
\
 if ( !k->clock )\
 {\
    if ( obs_a_h )\
  INSV( lat_b_h, hbit, lbit, pi_a_h ^ EXTV( lat_a_h, hbit, lbit));\
    else\
  INSV( lat_b_h, hbit, lbit, EXTV( lat_a_h, hbit, lbit));\
 }\
    }\
\
     else if ( hbit == lbit )\
     {\
 if ( k->clock )\
 {\
    if ( lbit == 0 )\
  INS( lat_a_h, hbit, si_h ) ;\
    else\
  INS( lat_a_h, hbit, EXT( lat_b_h, hbit - 1 )) ;\
 }\
\
 if ( !k->clock )\
 {\
    if ( obs_a_h )\
  INS( lat_b_h, hbit, pi_a_h ^ EXT( lat_a_h, hbit )) ;\
    else\
  INS( lat_b_h, hbit, EXT( lat_a_h, hbit ));\
 }\
    }\
}
/* End of observability LFSR MAcros */


/*
**
** Global chipwide (inter-box) signals
**
*/

/* Clock Interface Signals */

DECLARE struct  k {
 CLOCK    (clock, 1, 50); /* clock */
 SIGNAL   (reset_a_h, 1); /* EVENTUALLY REMOVE THIS SIGNAL !!!!! */
```

```
 SIGNAL    (sys_clk_out1_h, 1);
 SIGNAL    (sys_clk_out2_h, 1);
 SIGNAL    (sysclk_ratio_h, 4);
 REGISTER (sysclk_in_five_a_h, 1, 0);
 REGISTER (sysclk_in_two_a_h, 1, 0);
 REGISTER (sysclk_in_one_a_h, 1, 0);
 REGISTER (c_slip_req_a_h, 1, 0);

 SIGNAL (t_sys_reset_a_l, 1);
 REGISTER (t_dc_ok_h, 1, 0);
 SIGNAL (z_pad_tristate_h, 1);
} *k;


/* Test Box Signals
**------------------------------
*/

/* conditionally redefine "t" to "tt" to solve Verilog link conflict */

#ifdef EXT_USE
#define t tt
#endif

DECLARE struct  t {
 /* title: Test Box Signals */
 SIGNAL(e_reset_l, 1); /* EVENTUALLY REMOVE THIS SIGNAL !!!!! */
 SIGNAL(e_treset_l, 1); /* EVENTUALLY REMOVE THIS SIGNAL !!!!! */

 /* all reset signals asynch assert and synch deassert */
 SIGNAL(i_reset_b_l,1);     /* EVENTUALLY REMOVE THIS SIGNAL !!!!! */
 SIGNAL(i_iss_reset_b_l,1);  /* ibox gets two reset signals */
 SIGNAL(i_idx_reset_b_l,1);  /* ibox gets two reset signals */
 SIGNAL(e_reset_b_l,1);     /* ebox reset */
 SIGNAL(m_reset_b_l,1);     /* mbox reset */
 SIGNAL(c_reset_b_l,1);     /* cbox reset */
 SIGNAL(f_reset_b_l,1);     /* fbox reset */

 SIGNAL(c_pad_reset_b_l,1);  /* reset the pads */
 SIGNAL(c_pad_tristate_l,1);   /* tristate the pads */

 REGISTER(j_clr_tag_a_h,1,0);    /* clear Icache tag durign reset; this signal will probably not be needed */

 /* title: Dispatch to Generic Test Features */
 REGISTER(z_obl_on_a_h,1,0);     /* GET RID OF THIS REAL SOON !!! */
 REGISTER(z_cbl_on_a_h,1,0);     /* GET RID OF THIS REAL SOON !!! */
 REGISTER(s_obs_capture_a_h,1,0);    /* GET RID OF THIS REAL SOON !!! */
 REGISTER(z_obl_on_b_h,1,0); /* turn on observability LFSRs */
 REGISTER(z_cbl_on_b_h,1,0); /* turn on controllability LFSRs; this signal may not get used anywhere */
 REGISTER(s_obs_capture_b_h,1,0);  /* GET RID OF THIS REAL SOON !!! */
 REGISTER(s_l_obl_on_b_h,1,0);  /* turn on lscache LFSRs */
 REGISTER(s_r_obl_on_b_h,1,0);  /* turn on rscache LFSRs */
 REGISTER(i_sl_rcv_a_h,1,0);     /* receive serial data */
 REGISTER(i_icfail_a_h,1,0); /* copy of t_bst->icfail_b_h */

 /* title: Signals to BHT Array */
 SIGNAL(j_bht_new_5b_h, 8);  /* T_BST_FSD outputs to BHT Array */

 /* title: Data Array */
 REGISTERW(j_dat_in_a_h,     128, W4, 0);
 REGISTER(j_dat_in_dcd_a_h,    20, 0);
 REGISTER(j_dat_in_par_a_h,    2, 0);

 /* title: Tag Array */
 REGISTER(j_fpc_par_2a_h,      1,0); /* Tag Parity bit, input to ICACHE Tag*/
 REGISTER(j_valid_2a_h,        2,0); /* (1:0) Tag Valid bits, in to ICACHE Tag */
 REGISTER(j_pa_2a_h,           1,0); /* Phy Addr bit, input to ICACHE Tag */
 REGISTER(j_asn_a_h,           7,0); /* (6:0) ASN bits, input to ICACHE Tag */
 REGISTER(j_asm_a_h,           1,0); /* ASM bit, input to ICACHE Tag */
 REGISTERW(j_fpc_2a_h,     43, W2, 0); /* 30 bits exactly */

 /* title: Other signals to ICache */
 SIGNAL(j_bst_bistdone_b_h, 1); /* Bist done sig to clear TAG Valids */
 /* Delete above once j_tag.c is changed */
 SIGNAL(j_bst_bist_running_b_l, 1); /* Indicates bist is running. */
```

```
 SIGNAL(j_bst_wr_za_h,          1); /* write command for BiSt/SROM */
 SIGNAL(j_bst_do_program_b_l, 2); /* Program enable from FRC to IC */
 SIGNAL(j_bst_red_init_b_l, 1); /* Init redundancy. Disbales spares. Enables regular rows. */

 /* title: Signals to the IBox index dp */
 SIGNAL(i_bst_repair_index_a_h, 13); /* Repair index. Use only 10:6 */
 SIGNAL(i_bst_nextaddress_b_h, 1); /* Increment address */
 REGISTER(i_bst_test_ctl_b_h, 2, 0); /* Test Control to idx counter */

 /* title: Other Signals */
 SIGNAL(z_bst_ictest_b_h, 1); /* IC test mode. Controls muxes etc  */
 SIGNAL(z_bst_bist_init_b_h,     1); /* Init signal from BiSt State m/c UNUSED. DELETE! */

 /* title: JTAG Signals */
 SIGNAL( z_jtg_bsr_capture_b_h,  1); /* capture sig to BSR */
 CLOCK( z_jtg_bsr_update_b_h, 50, 100); /* Update sig to BSR */
 SIGNAL( z_jtg_bsr_drv_pins_b_h, 1); /* Output mux control to pads */
 CLOCK( z_jtg_bsr_clk_a_h, 0, 50); /* Slave clock for bsr */
 CLOCK( z_jtg_bsr_clk_b_h, 50,100); /* Master clock for BSR */
 SIGNAL( z_jtg_si_to_bsr_b_h, 1); /* Ser in to 1st BSR Cell. NOT A GLOBAL */

 REGISTER (k_bsr_so_pm_h, 2, 0); /* si to kbox BSR; only use bit 1 */

} *t;


DECLARE struct p {

 /* title: External interface */

 /* Clocks */
 REGISTER (clk_in_h, 1, 0);
 REGISTER (clk_in_l, 1, 0);
 SIGNAL (cpu_clk_out_h, 1);
 SIGNAL (sys_clk_out1_h, 1);
 SIGNAL (sys_clk_out1_l, 1);
 SIGNAL (sys_clk_out2_h, 1);
 SIGNAL (sys_clk_out2_l, 1);
 REGISTER (ref_clk_in_h, 1, 0);
 REGISTER (sys_reset_l, 1, 0);
 SIGNAL (clk_mode_h, 2);       /* functionality not currently modeled */

 /* System Interface */
 BUSW (addr_h, 40, W2);
 BUS (cmd_h, 4);
 BUS (addr_cmd_par_h, 1);
 REGISTER (victim_pending_h, 1, 0);
 REGISTER (addr_bus_req_h, 1, 0);
 REGISTER (cack_h, 1, 0);
 REGISTER (cfail_h, 1, 0);
 REGISTER (addr_res_h, 3, 0);
 REGISTER (int4_valid_h, 4, 0);
 REGISTER (scache_set_h, 2, 0);
 REGISTER (fill_h, 1, 0);
 REGISTER (fill_id_h, 1, 0);
 REGISTER (dack_h, 1, 0);
 REGISTER (fill_error_h, 1, 0);
 REGISTER (fill_nocheck_h, 1, 0);
 REGISTER (system_lock_flag_h, 1, 0);
 REGISTER (idle_bc_h, 1, 0);
 REGISTER (data_bus_req_h, 1, 0);

 /* Bcache Interface */
 REGISTER (index_h, 26, 0);
 BUSW (data_h, 128, W4);
 BUS (data_check_h, 16);
 BUS (tag_data_h, 19);
 BUS (tag_data_par_h, 1);
 BUS (tag_valid_h, 1);
 BUS (tag_shared_h, 1);
 BUS (tag_dirty_h, 1);
 BUS (tag_ctl_par_h, 1);
 REGISTER (tag_ram_oe_h, 1, 0);
 REGISTER (tag_ram_we_h, 1, 0);
 REGISTER (data_ram_oe_h, 1, 0);
 REGISTER (data_ram_we_h, 1, 0);
```

```
/* Misc stuff */
REGISTER (irq_h, 4, 0);
REGISTER (sys_mch_chk_irq_h, 1, 0);
REGISTER (pwr_fail_irq_h, 1, 0);
REGISTER (mch_hlt_irq_h, 1, 0);
REGISTER (dc_ok_h, 1, 0);

/* test port */
       REGISTER(port_mode_h,2,0); /* selects NORMAL, MANUFACTURING, or DEBUG */
       REGISTER(srom_present_l,1,0);   /* SROM used to load ICache */
       REGISTER(srom_data_h,1,0); /* SROM data, or serial receive */
       REGISTER(srom_clk_h,1,0); /* SROM clock, or serial transmit */
       REGISTER(srom_oe_l,1,0); /* enable either SROM or serial terminal */
       REGISTER(tdi_h,1,0);  /* JTAG data input */
       REGISTER(tdo_h,1,0);  /* JTAG data output */
       REGISTER(tms_h,1,0);  /* JTAG mode select */
       REGISTER(tck_h,1,0);  /* JTAG clock */
       REGISTER(trst_l,1,0);  /* JTAG reset */
       REGISTER(test_sta_h,2,0);     /* information on test status */
REGISTER(temp_sense,1,0);

/* performance counter */

SIGNAL(perf_mon_h, 1);  /* external performance counter input */

} *p;

DECLARE struct p_drv {

/* title: External interface tri-state control variables*/
       VARIABLE(addr_drv_ctl, 32);
       VARIABLE(cmd_drv_ctl, 32);
       VARIABLE(addr_cmd_par_drv_ctl, 32);
       VARIABLE(data_drv_ctl, 32);
       VARIABLE(data_check_drv_ctl, 32);
       VARIABLE(tag_data_drv_ctl, 32);
       VARIABLE(tag_data_par_drv_ctl, 32);
       VARIABLE(tag_valid_drv_ctl, 32);
       VARIABLE(tag_shared_drv_ctl, 32);
       VARIABLE(tag_dirty_drv_ctl, 32);
       VARIABLE(tag_ctl_par_drv_ctl, 32);

/* test port drive enables */
/* VARIABLE(port_mode_drv_ctl,32); */ /* no longer an output */
VARIABLE(srom_present_drv_ctl,32);
VARIABLE(srom_data_drv_ctl,32);
VARIABLE(srom_clk_drv_ctl,32);
VARIABLE(srom_oe_drv_ctl,32);
VARIABLE(tdi_drv_ctl,32);
VARIABLE(tdo_drv_ctl,32);
VARIABLE(tms_drv_ctl,32);
VARIABLE(tck_drv_ctl,32);
VARIABLE(trst_drv_ctl,32);
VARIABLE(test_sta_drv_ctl,32);
} *p_drv;

/* Ibox Interface Signals */
DECLARE struct i {

/* title: Signals to E and MBOXes */

REGISTER(z_e0_inst_2b_h  ,32,0); /* Integer/MBOX Pipe Instructions          */
REGISTER(z_e1_inst_2b_h  ,32,0); /*    includes OPC,Src R#s,LIT,DISP etc. fields  */
REGISTER(z_e0_issue_4a_h ,1,0); /* Instruction Issue Lines     */
REGISTER(z_e1_issue_4a_h ,1,0);
SIGNAL(z_stall_3b_h  ,1); /* Freeze Line       */

/* title: Signals to the EBOX  */
```

```
SIGNAL(e_ra0_addr_2a_h  ,5); /* Register file address for read port RA0  */
SIGNAL(e_ra1_addr_2a_h  ,5); /* Register file address for read port RA1  */
SIGNAL(e_rb0_addr_2a_h  ,5); /* Register file address for read port RB0  */
SIGNAL(e_rb1_addr_2a_h  ,5); /* Register file address for read port RB1  */
REGISTER(e_use_e0a_3b_h  ,1,0); /* use RF as the src operand -- are these needed? */
REGISTER(e_use_e0b_3b_h  ,1,0);
REGISTER(e_use_e1a_3b_h  ,1,0);
REGISTER(e_use_e1b_3b_h  ,1,0);
REGISTER(e_use_e0lit_3b_h ,1,0); /* Use the LIT field instead of RB    */
REGISTER(e_use_e1lit_3b_h ,1,0); /*   are these needed??            */
REGISTER(e_byp_e0s4_e0a_3b_h ,1,0);  /* EBOX RF bypasses      */
REGISTER(e_byp_e0s4_e0b_3b_h ,1,0);
REGISTER(e_byp_e0s4_e1a_3b_h ,1,0);
REGISTER(e_byp_e0s4_e1b_3b_h ,1,0);
REGISTER(e_byp_e1s4_e0a_3b_h ,1,0);
REGISTER(e_byp_e1s4_e0b_3b_h ,1,0);
REGISTER(e_byp_e1s4_e1a_3b_h ,1,0);
REGISTER(e_byp_e1s4_e1b_3b_h ,1,0);
REGISTER(e_byp_e0s5_e0a_3b_h ,1,0);
REGISTER(e_byp_e0s5_e0b_3b_h ,1,0);
REGISTER(e_byp_e0s5_e1a_3b_h ,1,0);
REGISTER(e_byp_e0s5_e1b_3b_h ,1,0);
REGISTER(e_byp_e1s5_e0a_3b_h ,1,0);
REGISTER(e_byp_e1s5_e0b_3b_h ,1,0);
REGISTER(e_byp_e1s5_e1a_3b_h ,1,0);
REGISTER(e_byp_e1s5_e1b_3b_h ,1,0);
REGISTER(e_byp_e0w_e0a_3b_h ,1,0);
REGISTER(e_byp_e0w_e0b_3b_h ,1,0);
REGISTER(e_byp_e0w_e1a_3b_h ,1,0);
REGISTER(e_byp_e0w_e1b_3b_h ,1,0);
REGISTER(e_byp_e1w_e0a_3b_h ,1,0);
REGISTER(e_byp_e1w_e0b_3b_h ,1,0);
REGISTER(e_byp_e1w_e1a_3b_h ,1,0);
REGISTER(e_byp_e1w_e1b_3b_h ,1,0);
REGISTER(e_dual_cmp_3b_h ,1,0); /* Special dual issue widget.   Tells the EBOX that we might */
REGISTER(e_dual_log_3b_h ,1,0); /*   dual issue a CMP-BR or CMP-CMOV and that they should be
     *    ready to use their special widget.  Note that if we don't
     *    actually wiggle both E0 and E1 issue lines in the next phase
     *    then we didn't actually do the dual issue due to some stall
     *    condition      */
REGISTERW(e_pc_4a_h     ,64,W2,0); /* PC bus to the Ebox pipe E1    */
REGISTER(e_use_ld0_5a_h     ,1,0); /* Select the LOAD port rather than the datapath for S6 results */
REGISTER(e_use_ld1_5a_h     ,1,0);
REGISTER(e_mul_abort_h  ,1,0); /* Abort the multiplier     */
REGISTER(e_sel_mul_4b_h  ,1,0);    /* Select the Multiplier result rather than the shifter */
SIGNAL(e_w0_addr_5a_h  ,5); /* Register Write Addresses     */
SIGNAL(e_w1_addr_5a_h  ,5);
SIGNAL(e_w0_en_6a_h  ,1); /* Write Enables      */
SIGNAL(e_w1_en_6a_h  ,1);
SIGNAL(e_rd_pal_shadow_2a_h ,1); /* Use PAL SHADOW regs for read    */
REGISTER(e_w0_pal_shadow_5a_h ,1,0); /* Use PAL SHADOW for W0 write     */
REGISTER(e_w1_pal_shadow_5a_h ,1,0);
REGISTER(e_intr_flag_3a_h ,1,0); /* The intr_flag for RS/RC instructions (must be valid a mux delay ahead of
SIGNAL(e_use_intr_flag_3a_h ,1); /* Selects the intr_flag over the literal(must be valid a mux delay ahead c

/* title: Signals to E and FBOXes */
SIGNAL(z_br_predict_4a_h     ,1); /* Predict that the branch is taken   */

/* title:  Signals to FBOX */
```

```
REGISTER(f_fa_inst_3a_h  ,32,0); /* FA instruction -- all 32 bits
      *   The schematic/layout implementation will not require bits 4:0
      *   they are included in the model in order to retain the correct
            *   bit numbering.     */
REGISTER(f_fm_inst_3a_h  ,32,0); /* FM Instruction  -- all 32 bits
      *    The schematic/layout implementation will only require bits
      * 30,27:26,25:11.  The rest are included in the model in
      *       order to retain the correct bit numbering */
REGISTER(f_st_inst_3a_h  ,32,0); /* E0 Instruction after an S3 latch routed to the FBOX
      *    The schematic/layout implementatation will not require all
      *    32 bits.  They are included in the model in order to retain
      *    the correct bit numbering    */
REGISTER(f_fa_issue_4a_h ,1,0); /* Issue Lines - timing may change, rpp   */
REGISTER(f_fm_issue_4a_h ,1,0);
REGISTER(f_st_issue_4a_h ,1,0);
REGISTER(f_byp_fm_fma_3b_h ,1,0); /* FBOX RF Bypass Lines  -- timing may change, rpp */
REGISTER(f_byp_fm_fmb_3b_h ,1,0);
REGISTER(f_byp_fm_faa_3b_h ,1,0);
REGISTER(f_byp_fm_fab_3b_h ,1,0);
REGISTER(f_byp_fm_st_3b_h ,1,0);
REGISTER(f_byp_fa_fma_3b_h ,1,0);
REGISTER(f_byp_fa_fmb_3b_h ,1,0);
REGISTER(f_byp_fa_faa_3b_h ,1,0);
REGISTER(f_byp_fa_fab_3b_h ,1,0);
REGISTER(f_byp_fa_st_3b_h ,1,0);
REGISTER(f_byp_ld0_fma_3b_h ,1,0);
REGISTER(f_byp_ld0_fmb_3b_h ,1,0);
REGISTER(f_byp_ld0_faa_3b_h ,1,0);
REGISTER(f_byp_ld0_fab_3b_h ,1,0);
REGISTER(f_byp_ld0_st_3b_h ,1,0);
REGISTER(f_byp_ld1_fma_3b_h ,1,0);
REGISTER(f_byp_ld1_fmb_3b_h ,1,0);
REGISTER(f_byp_ld1_faa_3b_h ,1,0);
REGISTER(f_byp_ld1_fab_3b_h ,1,0);
REGISTER(f_byp_ld1_st_3b_h ,1,0);
REGISTER(f_fdiv_abort_h  ,1,0); /* Abort the Floating Point Divider    */
SIGNAL(f_ld0_addr_5a_h ,5);         /* Floating Register File, Load port addresses  */
SIGNAL(f_ld1_addr_5a_h ,5);
SIGNAL(f_we_ld0_6a_h  ,1); /* LOAD port write enables, delete these when the FBOX updates  */
SIGNAL(f_we_ld1_6a_h  ,1);
SIGNAL(f_fill_we_ld0_6a_h       ,1);    /* New Load port write enables for FILLs only, not qualified with aborts */
SIGNAL(f_fill_we_ld1_6a_h       ,1);
SIGNAL(f_hit_we_ld0_6a_h       ,1);    /* New Load port write enables for HITs only, qulaified with MBOX aborts */
SIGNAL(f_hit_we_ld1_6a_h       ,1);
SIGNAL(f_fa_addr_7a_h  ,5); /* operate write port addresses    */
SIGNAL(f_fm_addr_7a_h  ,5);
REGISTER(f_we_fa_8a_h  ,1,0); /* oeprate write port enables     */
REGISTER(f_we_fm_8a_h  ,1,0);

/* title: Signals to Mbox */

SIGNAL (m_pal_shadow_en_3a_h,1);   /* PAL SHADOW Mode bit, MBOX will store in MAF with the
     *  register address         */

REGISTER(m_e0_valid_2b_h,1,0);  /* indicates a valid instruction has been slotted */
REGISTER(m_e1_valid_2b_h,1,0);  /* indicates a valid instruction has been slotted */

REGISTER(m_kill_e0_5b_h,  1,0); /* pipe0 ibox/ebox/fbox traps */
REGISTER(m_kill_e1_5b_h,  1,0); /* pipe1 ibox/ebox/fbox traps */
SIGNAL (m_kill_dtbis_4a_h,  1); /* pipe0 kill for dtbis only */

SIGNAL (m_imaf_req_1b_h,  1); /* load iref PA into MAF */
SIGNAL (m_iref_idx_1b_h,  2); /* iref prefetch queue index*/
SIGNALW(m_iref_addr_2a_h,    40,W2); /* (39:4) iref Physical Address */
SIGNAL (m_iref_req_2a_h,  1); /* iref PA is real, MAF can begin arbing */


/* title: Signals to ICACHE & Refill Buffer & BHT */
```

```
REGISTER(j_ic_index_zb_h,    13,0); /* (12:0) --> ICACHE only uses (12:4): Index (Block, index (row), octaw
SIGNAL (j_ic_index_zb_l,      13); /* (12:0) --> ICACHE only uses (12:4): _L version of above */
SIGNAL (j_asm_b_h,  1); /* ASM bit, input to ICACHE Tag */
SIGNAL (j_asn_b_h,  7); /* (6:0) ASN bits, input to ICACHE Tag */
SIGNAL (j_pa_lb_h,  1); /* Physical Address bit, input to ICACHE Tag */
SIGNAL (j_valid_lb_h,          2);     /* (1:0) Tag Valid bits, input to ICACHE Tag */
SIGNALW (j_fpc_lb_h,     43,W2); /* (42:0) --> ICACHE only uses 30 bits (42:13). Fill Tag, input to ICACHE
SIGNAL (j_fpc_par_lb_h,        1);     /* Tag Parity bit, input to ICACHE Tag */

/** OLD SIGNAL TO BE DELETED */
SIGNAL (j_flush_a_h,  1); /* Clear all IC valid bits.  Flush occurs during B-phase */
/** END OLD SIGNAL TO BE DELETED */
SIGNAL (j_flush_b_h,  1); /* Clear all IC valid bits.  Flush occurs during next B-phase */

SIGNAL (j_ic_cmd_a_h,  1); /* command to Icache: READ (=0), FILL (=1) */
SIGNAL (j_force_bad_dp_a_h, 1); /* Force bad data parity on data going into the Refill Buffer and Icache */
REGISTER(j_ib_stall_a_h,      1,0); /* IB_STALL, Sense Amp disable, mux control */
SIGNAL (j_bypass_ic_a_h, 1); /* Bypass Icache: IB gets FILL data or RFB data */

SIGNAL (j_rfb_rd_idx_b_h,      13); /* (12:0) RFB Read Index, RFB only uses (6:4), Icache latches in A, uses
REGISTER(j_rfb_wr_idx_a_h,    3,0); /* (2:0) Refill Buffer Write Index, arrives one cycle ahead of write */
REGISTER(j_rfb_write_a_h,     1,0); /* Refill Buffer Write Enable, arrives one cycle ahead of write */

   SIGNAL (j_bht_new_5b_h,  8); /* (7:0) Branch History Bits to update */
SIGNAL (j_bht_idx_zb_h,        13); /* (12:0) --> ICACHE uses 9 bits (12:4), Index for BHT read */
SIGNAL (j_bht_idx_zb_l,        13); /* (12:0) --> ICACHE uses 9 bits (12:4), _L of above */
SIGNAL (j_hup_idx_5b_h,        13); /* (12:0) --> ICACHE uses 9 bits (12:4), Index for BHT update */
SIGNAL (j_hup_en_5b_h,  1); /* BHT update enable */
SIGNAL (j_bht_silo_sel_b_h,     1);     /* Use BHT output delayed by one cycle, for use on RFB reads */

      /* title: Signals to Cbox */
      SIGNAL (c_clr_lock_flg_a_h,     1);     /* Signal to clear lock flag when necessary from PAL CODE  */

/* title: Signals to Tbox */
SIGNAL(t_tst_index_za_h, 13); /* Test Indexes. Use 12:4 */
REGISTER(t_lastaddress_a_h, 1,0); /* Test IDX Counter overflow */
REGISTER(t_obl_so_b_h,1,0); /* serial out of observability LFSR chain */
REGISTER(t_sl_xmit_b_h,1,0);   /* transmit serial data */
REGISTER(t_icsr_sle_b_h,1,0);   /* SEL bit (31) of ICSR */
REGISTER(t_icsr_sta_b_h,1,0); /* ICSR can turn on test_sta<1> */
REGISTER(t_dbg_data_a_h,8,0);   /* parallel observability */

/* title: Timeout Reset signals */
SIGNAL(m_treset_b_l,1);
SIGNAL(c_treset_b_l,1);
SIGNAL(e_treset_b_l,1);
SIGNAL(f_treset_b_l,1);

} *i;


/* Ebox Interface Signals */
DECLARE struct e {
 /* title: Signals to Ibox */
 REGISTER (i_mul_done_soon_0a_h,1,0); /* Multiplier will deliver data soon */
 REGISTER (i_kill_cmov0_4b_h,   1,0); /* Do not write or bypass result of CMOV issued to E0 */
 SIGNAL   (i_kill_cmov1_4b_h,   1); /* Do not write or bypass result of CMOV issued to E1 */
 SIGNAL   (i_br_taken_5a_h,     1); /* Branch condition is satisfied */
 REGISTER (i_br_mispredict_5a_h,1,0); /* Branch was mispredicted */
 REGISTER (i_int_ovf0_6b_h,     1,0); /* Overflow from pipe E0 */
 REGISTER (i_int_ovf1_6b_h,     1,0); /* Overflow from pipe E1 */
 REGISTER (i_mul_ovf_8a_h,      1,0); /* Overflow from the multiplier */
 SIGNALW  (i_pc_4ac_h,      64,W2); /* PC bus to Ibox */
 REGISTER(i_obl_so_b_h,1,0); /* serial out of observability LFSR chain */

 SIGNALW  (i_pc_3b_h,      64,W2); /* DELETE ME */

 /* title: Signals to Mbox */
   SIGNALW  (m_va0_4bc_h,  64,W2);  /* Pipe0 Virtual Address  (also data for mtpr tbis */
 SIGNALW  (m_va1_4bc_h,  64,W2);  /* Pipe1 Virtual Address  */

 /* These low asserted versions are not used in the model.  They are here for GUIDEWIRE purposes only. */
   SIGNALW  (m_va0_4bc_l,  64,W2);
 SIGNALW  (m_va1_4bc_l,  64,W2);

 REGISTERW(m_st_data_4a_h,  64,W2,0); /* Integer store  and MTPR data */
```

```
 /* title: Signals to Dcache */
   REGISTER(d_va0_4a_h,        13,0); /* (12:3) Pipe0 Virtual Address from fast adder output */
 REGISTER(d_va1_4a_h,        13,0); /* (12:3) Pipe1 Virtual Address from fast adder output */
} *e;


/* Fbox Interface Signals */
DECLARE struct f {
 REGISTER(i_br_taken_5a_h, 1,0);     /* branch taken   */
 REGISTER(i_br_mispredict_5a_h, 1, 0);  /* branch mispredict  */
 REGISTER(i_kill_cm_5a_h, 1, 0);   /* KILL conditional move */
 REGISTER(div_done_soon_1b_h, 1, 0);  /* Divide done soon  */
 REGISTER(i_fdz_fa_8b_h, 1, 0) ;   /* divide by zero  */
 REGISTER(i_fov_fa_8b_h, 1, 0) ;   /* floating add pipe over flow  */
 REGISTER(i_fun_fa_8b_h, 1, 0) ;   /* floating add pipe under flow */
 REGISTER(i_ine_fa_8b_h, 1, 0) ;   /* floating add pipe inexact */
 REGISTER(i_inv_fa_8b_h, 1, 0) ;   /* floating add pipe invalid operand */
 REGISTER(i_iov_fa_8b_h, 1, 0) ;   /* floating add pipe int overflow    */
 REGISTER(i_swc_fa_8b_h, 1, 0) ;   /* floating add pipe software completion */

 REGISTER(i_fov_fm_8b_h, 1, 0) ;   /* floating mul pipe over flow  */
 REGISTER(i_fun_fm_8b_h, 1, 0) ;   /* floating mul pipe under flow */
 REGISTER(i_ine_fm_8b_h, 1, 0) ;   /* floating mul pipe inexact */
 REGISTER(i_inv_fm_8b_h, 1, 0) ;   /* floating mul pipe invalid operand */
 REGISTER(i_swc_fm_8b_h, 1, 0) ;   /* floating mul pipe software completion */

 REGISTER(c_obl_so_b_h,1,0); /* serial out of observability LFSR chain */

} *f;


/* Mbox Interface Signals */
DECLARE struct m {
 /* title: Signals to Ebox */

/*****************************************************************************************************/
 BUSW(e_ld_data0_5bc_h,   64,W2); /* Pipe0 data returened to Ebox register file for:
        LD, fill, MFPR, RPCC, STxC */
 BUSW(e_ld_data0_5bc_l,   64,W2); /* Pipe0 data returened to Ebox register file for:
        LD, fill, MFPR, RPCC, STxC */
 BUSW(e_ld_data1_5bc_h,   64,W2); /* Pipe1 data returened to Ebox register file for:
        LD, fill  */
 BUSW(e_ld_data1_5bc_l,   64,W2); /* Pipe1 data returened to Ebox register file for:
        LD, fill  */
       SIGNAL (e_big_endian_7a_h, 1);     /* E_BIG_ENDIAN mode bit from MCSR register */

 REGISTER(e_obl_so_b_h,1,0); /* serial out of observability LFSR chain */

/*****************************************************************************************************/

 /* title: Signals to Cbox */

 REGISTERW (c_maf_addr_5b_h,   40,W2,0); /* (39:2) physical address. Valid end-6a at Cbox. */
 SIGNAL (c_maf_cmd_5b_h,  4); /* commands to Cbox. Valid end-6a at Cbox. */
 SIGNAL (c_maf_index_5b_h, 5); /* 1 of 16 maf entries. Valid mid-6a at Cbox */
 SIGNAL (c_maf_type_5b_h, 1); /* integer/floating. Valid end-6a at Cbox. */
 REGISTER (c_wr_type_5b_h, 1, 0); /* LW or QW writes. Valid mid-6a at Cbox. */
 REGISTER (c_wr_lw_addr_5b_h, 5, 0); /* (4:2) LW to write in WB.Valid mid-6a at Cbox. */

 SIGNAL (c_maf_abort_6a_h, 1); /* abort cmd. Valid mid-6b at Cbox. */
 REGISTER (c_wr_enable_6a_h, 6, 0); /* 1 of 6 WB entries. Valid end-6b at Cbox. */
 REGISTER (c_drd_mask_8b_h, 4, 0); /* qw masks for i/o reads. Valid end-9a at Cbox */

 /* title: Signals to Ibox */

  SIGNAL (i_dc_hit_e0_5b_h,  1); /*  pipe0 dc_hit */
 SIGNAL (i_dc_hit_e1_5b_h,  1); /*  pipe1 dc_hit */
 SIGNAL (i_mb_clear_2b_h,  1); /* RS, RC, MB, STxC done, o.k. to restart */
```

```
REGISTER (i_dmm_err_e0_5b_h, 1, 0); /* pipe0 Mbox trap (summary of all MM traps) */
REGISTER (i_dmm_err_e1_5b_h, 1, 0); /* pipe1 Mbox trap (summary of all MM traps) */
SIGNAL (i_align_err_e0_5a_h, 1); /* pipe0 VA alignment error  */
SIGNAL (i_align_err_e1_5a_h, 1); /* pipe1 VA alignment error  */
SIGNAL (i_in_tb_flow_5a_h, 1); /* pipe0 trap happened while in a TB flow */
SIGNAL (i_dtb_miss_e0_5a_h, 1); /* pipe0 dtb miss */
SIGNAL (i_dtb_miss_e1_5a_h, 1); /* pipe1 dtb miss */
/* These two signals to be deleted later */
SIGNAL (i_mbox_unavail_e0_5b_h, 1); /* pipe0 - wb conflict, wb full, maf full trap */
SIGNAL (i_mbox_unavail_e1_5b_h, 1); /* pipe1 - wb conflict, wb full, maf full trap */
SIGNAL (i_mbox_unavail_e0_6ac_h,1); /* pipe0 - wb conflict, wb full, maf full trap, dmm err */
SIGNAL (i_mbox_unavail_e1_6ac_h,1); /* pipe1 - wb conflict, wb full, maf full trap, dmm err */
REGISTER(i_perr_6a_h,          1,0);   /* Dcache tag or data parity error */

SIGNAL (i_fill_rnum0_4a_h, 7); /*  pipe0 register for fill data   NEW TIMING
                                        <6>-pal shadow, <5>-(I=0,F=1),
               <4:0>-register number
                                        Fbox ignores bit 6  */
SIGNAL (i_fill_rnum1_4a_h, 7); /*  pipe1 register for fill data   NEW TIMING
                                        <6>-pal shadow, <5>-(I=0,F=1),
               <4:0>-register number
                                        Fbox ignores bit 6  */
SIGNAL (i_fill_valid0_4b_h, 1); /* fill data coming on pipe0 NEW TIMING   */
SIGNAL (i_fill_valid1_4b_h, 1); /* fill data coming on pipe1   NEW TIMING */

SIGNAL (i_fill_valid0_4a_h,     1);     /* fill data coming on Pipe0, new functionality, is that the MBOX
      will send in 4A, and the IBOX will qualify with the CBOX RFB_DATA_VALID
      signal -- MBOX to remove the 4B signals when the change is implemented */
SIGNAL (i_fill_valid1_4a_h,     1); /* ditto for pipe1 */

REGISTER (i_fill_coming_4a_h, 1, 0); /* Fbox fill is coming, but may not be valid NEW TIMING */

REGISTER(i_ld_alloc_e0_6b_l,  1, 0); /* to performance counter to indicate that missed LD in E0 got allocate
REGISTER(i_ld_alloc_e1_6b_l,  1, 0); /* to performance counter to indicate that missed LD in E1 got allocate
REGISTER(i_wbmaf_full_e0_6a_h,  1, 0); /* to performance counter to indicate that ST has WB full or LD has M
REGISTER(i_maf_full_e1_6a_h, 1, 0); /* to performance counter to indicate that LD in E1 has MAF full */

REGISTER(i_dbg_data_a_h,8,0);   /* parallel observability */

/* title: Signals to Fbox */

SIGNAL (f_ld_format0_4b_h, 3); /* Format info for Fbox pipe0 fills and loads.
               <2> = vax_fp/ieee, <1> = LW/QW,
                                        <0> = lower/upper LW */
SIGNAL (f_ld_format1_4b_h, 3); /* Format info for Fbox pipe1 fills and loads.
               <2> = vax_fp/ieee, <1> = LW/QW,
                                        <0> = lower/upper LW */
SIGNAL(f_fbox_drv_ena_5a_h,     1);     /* asserted at 5a ==> fbox drives b->d_wr_data_6a_h at 6a */


/* title: Signals to Dcache */

SIGNALW(d_dc_addr_xa_h,    39, W2); /* (38:3):  (38:13) = fill tag to be stored in Dcache tags,
                                                (12:8) = address for wordline decode, (7) = M1 w
                                                 (6:3) = column muxing
                                                 Timing is early A (reads, fills, stores occur f
SIGNAL (d_tag_idx_sel_3b_h, 1); /* Address source for DC Tags for 4B operation:  0=EBOX VA, 1=MBOX d_dc_addr
SIGNAL (d_dat_idx_sel_3b_h, 1); /* Address source for DC Data for 4B operation:  0=EBOX VA, 1=MBOX d_dc_addr
SIGNAL (d_nofill0_5a_h,  1); /* nofill Dcache0, from the MAF, we have one for each cache for testability on
SIGNAL (d_nofill1_5a_h,  1); /* nofill Dcache1, used for FILL in 5b,  "                "   */
SIGNAL (d_update_dcout_3b_h, 1); /* For power savings, don't update Z_DATAx_5A_H if this is 0 and cmd is not
SIGNAL (d_tag_cmd_3b_h,  2); /* (1:0) command to DC tag for 4b operation:  nop=00, read=01, fill=10, write=
SIGNAL (d_data_cmd_3b_h, 2); /* (1:0) command to DC data for 4b operation: nop=00, read=01, fill=10, write=
SIGNAL (d_st_adr_5b_h,  3); /* (2) Address bit 2, used for LW Stores, comes early to be used with wr_type_5b
SIGNAL (d_wr_type_5b_h,  1); /* LW or QW Store: 0=LW, 1=QW, used with d_st_adr_5b_h(2) for STORE in 6b */
SIGNAL (d_tag_par_5a_h,     1); /* fill tag parity to be stored in Dcache tags in 5B */
SIGNAL (d_valid_5a_h,          2);     /* fill tag valid bits to be stored in Dcache tags in 5B*/
SIGNAL (d_st_valid_6a_h, 1); /* store data is valid for the Dcache for STORE in 6B */
SIGNAL (d_dc_flush_a_h,  1); /* Dcache flush, if 1, clear all DC valid bits in B-phase */
SIGNAL (d_force_bad_par_5b_h, 1); /* Force bad parity on data parity into the Dcache data array, for STORE

} *m;


DECLARE struct c {
```

```
        /* title: Signals to Mbox */

        SIGNAL (m_sc_busy_4a_h,  1);  /* No scache access. Valid end-4b at Mbox */
        SIGNAL (m_retry_8a_h,  1); /* Retry. Valid mid-8b at Mbox */
            SIGNAL (m_sc_hit_7b_h,          1);     /* sc_hit in one of the 3 banks */
        SIGNAL (m_return_index_7a_h,  3);  /* only <2:0> needed.  1 of 16 maf entries. Valid end-7b at Mbox  cs 3-
7-93 */
        SIGNAL (m_return_status_7a_h, 3);  /* Return status. Valid end-7b at Mbox. */

      SIGNAL (m_wr_now_4a_h,  1);  /* Cbox initiated Writes. Valid mid-4b at Mbox */
      SIGNAL (m_wr_maf_index_4a_h, 3);  /* only <2:0> needed.  1 of 16 maf entries. Valid end-4b at Mbox */
            SIGNAL (m_wr_64B_req_4a_h, 1); /* to be deleted */
            SIGNAL (m_wr_64b_req_4a_h, 1); /* 64B mode Write. Valid end-4b at Mbox */

      SIGNAL (m_stxc_fail_7a_h, 1); /* StxC failed. Valid end-7b at Mbox */

      SIGNAL (z_rfb_data_valid_9a_h, 1); /* rfb. Valid end-9b at Mbox and Dcache */
      SIGNAL (m_rfb_ecc_err_10b_h, 1); /* ecc error. Valid end-11a at Mbox */
      SIGNAL (m_ow_valid_7a_h, 1); /* ow valid (bit4) for fills. Valid end 7b */
      SIGNAL (m_bogus_lf_8a_h, 1); /* bogus last fill. sysclock=3 and ff has error */
                                                  /* valid end 8a @ mbox. may spill into 8b */
      SIGNAL (z_alloc_cycle_2a_h, 1); /* integer fill bubble. Valid early-2b at Mbox */

      REGISTER(m_obl_so_b_h,1,0); /* serial out of observability LFSR chain */
      REGISTER(m_dbg_data_a_h,8,0);   /* parallel observability */

      /* title: Signals to Scache */

      /* to SCache tag array */
      SIGNAL  (s_bcache_size_a_h,    3); /* Bcache size. Valid end-a at Scache */
      REGISTER  (s_set_enable_a_h,     3,0); /* Set enables. Valid end-a at Scache */
      REGISTER (s_32b_mode_a_h,       1,0); /* 32b mode. Valid end-a at Scache    */
      REGISTER (s_flush_b_h,1,0);             /* flush all the valid bits in the Scache */

      SIGNALW (s_addr_6a_h, 40,      W2); /* 39:3 physical address. Valid end-6a at Scache. */
      REGISTER  (s_cmd_6b_h,3,0);      /* Command. Valid end-6b at Scache */

      SIGNAL  (s_fill_tag_cmd_5b_h,  1); /* Fill (on scmiss) command. Valid end-5b at Scache */
      SIGNAL  (s_fill_status_cmd_5b_h,1); /* Fill (32b mode, tag match, not valid) command.
      SIGNAL  (s_set_hit_5b_h,        2);     /* Pick 1 of 3 sets. valid end 5b */

      /* To be Deleted once move to 5b is done [cs] */
      REGISTER  (s_fill_tag_cmd_6a_h,   1,0); /* Fill (on scmiss) command. Valid end-6a at Scache */
      REGISTER  (s_fill_status_cmd_6a_h,1,0); /* Fill (32b mode, tag match, not valid) command.
                                                  Valid end-6a at Scache */
      REGISTER  (s_set_hit_6a_h,2,0);         /* Pick 1 of 3 sets. Valid end-6a at Scache */
      SIGNAL  (s_abort_7a_h,  1); /* Abort Scache operation. Valid end-7a at Scache */

      SIGNAL  (s_wr_shared_perm_6b_h, 1); /* Valid end-7a at Scache */
      SIGNAL  (s_wr_dirty_perm_6b_h, 1); /* Valid end-7a at Scache */


      /* status signals: 6a for fills, 7a for writes */
      SIGNAL (s_tag_v_6a_h,  1); /* Valid bit. Valid mid-6b at Scache */
      SIGNAL (s_tag_s_6a_h,  1); /* Shared bit. Valid mid-6b at Scache */
      SIGNAL (s_tag_d_6a_h,  1); /* Dirty bit. Valid mid-6b at Scache */
      SIGNAL (s_tag_m_6a_h,  2); /* Modified (16B) bit. Valid mid-6b at Scache */
      REGISTER (s_tag_parity_6a_h, 1,0); /* address parity. Valid mid-6b at Scache. */


      /* to SCache data array */
      SIGNAL (s_lw_write_7a_h, 4); /* 4 LW's per OW. Valid mend-7b at Scache. */
      REGISTER (s_wfb_parity_7b_h,4,0); /* 4 LW parity bits */
      REGISTERW(s_wfb_7b_h,128,W4,0);         /* Write Fatbus */
      REGISTER (s_wfb_parity_8a_h,4,0); /* (to be removed) 4 LW parity bits */
      REGISTERW(s_wfb_8a_h,128,W4,0);         /* (to be removed) Write Fatbus */

      REGISTER (s_ifb_drive_9a_h,1,0); /* Iread fill bus select */
      SIGNAL (s_rfb_drive_9a_h, 1); /* Dread fill bus select */
      SIGNAL (s_rfb_drive_8b_h, 1); /* Dread fill bus select */


      /* title: Signals to Ibox */

            /* **** NEW TIMINGS ******* 6-jul-1992 */

      SIGNAL (i_ifb_index_8b_h, 3); /* 1 of 8 iref rfb index */
      SIGNAL (i_ifb_data_valid_8b_h, 1); /* ifb data valid, one cycle before data */
      SIGNAL (i_ifb_last_fill_8b_h, 1); /* ifb data is last of the 2 OW from the fill. */
```

```
 SIGNAL (i_corr_err_trap_11b_h,  1);     /* Correctable error. Dstream. flop */
 REGISTER (i_corr_err_intr_11b_h,1,0);   /* Correctable error. D and I streams */
 SIGNAL (i_err_abt_11a_h,        1);     /* Ecc error D stream, RFB/Scache/Bcache tag/statu parity error */
       REGISTER (i_iecc_hrd_err_11b_h,1,0);    /* Uncorrectable ecc error on istream, fill error, sys cmd er
 /* To be deleted */
       SIGNAL (i_hard_err_11b_h,   1);         /* Uncorrectable ecc error on istream, fill error, sys cmd er

 SIGNAL (i_perf_mon_in_a_h, 1);  /* Input to IBOX perf monitor logic, a cycle wide PULSE */
 SIGNAL (i_irq_a_h, 4);                  /* synchronised interrupt request pins */
 SIGNAL (i_sys_mchk_irq_a_h, 1);         /* system machine check interrupt pin */
 SIGNAL (i_pwr_fail_irq_a_h, 1);         /* powerfail interrupt request pin */
 SIGNAL (i_mch_hlt_irq_a_h, 1);          /* machine halt interrupt request pin */

 SIGNAL (i_force_time_out_b_h, 1); /* force a time out if the system reqests one */

 REGISTER (i_pmc1_in_a_h, 1, 0);  /* input to performance counter 1 */
 REGISTER (i_pmc2_in_a_h, 1, 0);  /* input to performance counter 2 */

 /* title: Signals to Dcache */

 REGISTER (d_inval_cmd_9a_h, 1, 0);  /* invalidate command  */
 REGISTER (d_inval_addr_9a_h, 13, 0); /* 12:6 inval address  */
 REGISTER (d_fill_par_10a_h, 4, 0); /* DC Fill parity, 4 LW parity bits,
        source is A-latch in CBOX */

 REGISTER (k_slip_ok_a_h, 1, 0);  /* wave pipelined bcache access in progress */
 REGISTER (t_bsr_so_addr_h, 5, 0);   /* si to tbox BSR; only use bit 4 */

} *c;


/* Scache Interface Signals */
DECLARE struct s {

 /* title: Signals to Cbox */

 /* signals from tag section */
 SIGNAL (c_tag_v0_7a_h[3],  1);  /* Valid bits. Valid early-7b at Cbox */
 SIGNAL (c_tag_d0_7a_h[3], 1);  /* Dirty bits. Valid early-7b at Cbox */
 SIGNAL (c_tag_s0_7a_h[3], 1);  /* Shared bits. Valid early-7b at Cbox */
 SIGNAL (c_tag_m0_7a_h[3], 2);  /* Modified (OW). Valid early-7b at Cbox */

 SIGNAL (c_tag_v1_7a_h[3],  1);  /* Valid bits. Valid early-7b at Cbox */
 SIGNAL (c_tag_d1_7a_h[3], 1);  /* Dirty bits. Valid early-7b at Cbox */
 SIGNAL (c_tag_s1_7a_h[3], 1);  /* Shared bits. Valid early-7b at Cbox */
 SIGNAL (c_tag_m1_7a_h[3], 2);  /* Modified (OW). Valid early-7b at Cbox */

 SIGNAL (c_tag_parity_7a_h[3], 1);  /* tag parity bits. Valid mid-7b at Cbox */
 SIGNAL (c_tag_bc_index_match_7a_h[3], 1);  /* Bcache idx match. Valid mid-7b at Cbox */

 SIGNAL (c_hit_7a_h[3],      1);  /* Hit signal.  Valid mid-7b at Cbox */
 SIGNAL (c_tag_match_7a_h[3],   1);  /* Match signal. Valid mid-7b at Cbox */

 SIGNALW (c_tag_7b_h,     40,W2);  /* tag bits 39:15. Valid end-7b at Cbox */
 SIGNAL (c_tag_perr_7b_h[3], 1);  /* tag parity error. Valid end-7b at Cbox */


 /* title: Signals to ICache */
       /***** NEW TIMING **** 6-jul-1992 */
 SIGNALW (j_ifb_data_9b_h, 128, W4);  /* ICache fill bus */
 SIGNAL  (j_ifb_parity_9b_h, 4);  /* 4 LW parity bits */


 /****** TO BE DELETED ********* cs 6-jul-1992 */
 /* title: Signals to ICache */
 SIGNALW (j_ifb_data_10a_h, 128, W4);  /* ICache fill bus */
 SIGNAL  (j_ifb_parity_10a_h, 4);  /* 4 LW parity bits */
 /*****************************/

 /* title: Signals to TBOX */
 REGISTER(t_l_obs_so_b_h,1,0); /* GET RID OF THIS REAL SOON !!! */
 REGISTER(t_r_obs_so_b_h,1,0); /* GET RID OF THIS REAL SOON !!! */
 REGISTER(t_l_obl_so_b_h,1,0); /* serial out of lscache LFSR chain */
 REGISTER(t_r_obl_so_b_h,1,0); /* serial out of rscache LFSR chain */

} *s;
```

```
/* Buses */
DECLARE struct b {

 BUSW (z_rfb_9b_h,   128, W4); /* rfb data bus, sources are Scache and CBOX,
                                                  destination CBOX and Dcache */
 BUS  (c_rfb_parity_9b_h,  4); /* 4 LW parity bits, sources are Scache and CBOX,
                                                  destination CBOX */
 VARIABLE(c_rfb_state,        32); /* state variable for driving rfb from (scache or CBOX)*/
 VARIABLE(c_rfb_parity_state,   32); /* state variable for driving rfb parity from (scache or CBOX)
        To be deleted later, rfb_parity is not tristated [san,hale] */

 BUSW     (d_wr_data_6a_h,   64, W2); /* Dcache tristate store data from FBOX/EBOX for 6B STORE */
 BUS      (d_wr_lw_parity_6a_h,  2);     /* Dcache tristate store lw parity from FBOX/EBOX for 6B STORE */

       VARIABLE(d_wr_data_state,      32);   /* dcache store bus data driver state
                                                  (FBOX or EBOX)*/
 VARIABLE(d_wr_lw_parity_state, 32);   /* dcache store bus parity driver state
                                                  (FBOX or EBOX) */

} *b;


/* Dcache Interface Signals */
DECLARE struct d {

 /* title: Signals to Cbox */

 SIGNALW(c_wb_data_6a_h,    64, W2); /* 64 bits of store data to Cbox WB. */
 SIGNAL (c_wb_lw_parity_6a_h, 2); /* 2 parity bits per Quadword of store data */


 /* title: Signals to Mbox */

 SIGNALW (m_tag0_5a_h,      39, W2); /* (38:13)  tag0 for hit logic*/
 SIGNALW (m_tag1_5a_h,      39, W2); /* (38:13)  tag1 for hit logic*/
 SIGNAL (m_tag_par0_5a_h, 1); /* pipe0 tag parity */
 SIGNAL (m_tag_par1_5a_h, 1); /* pipe1 tag parity */
 SIGNAL (m_valid0_5a_h,  2); /* (1:0) pipe0 valid bits for block */
 SIGNAL (m_valid1_5a_h,  2); /* (1:0) pipe1 valid bits for block */
 SIGNAL (m_data_par0_5a_h, 2); /* pipe0 data parity */
 SIGNAL (m_data_par1_5a_h, 2); /* pipe1 data parity */

 /* title: Signals to Mbox and Fbox */

       /* NEW TIMING */
 SIGNALW(z_data0_5a_h,      64,W2); /* (63:0) pipe0 load data bus to mbox and fbox */
 SIGNALW(z_data1_5a_h,      64,W2); /* (63:0) pipe1 load data bus to mbox and fbox */

} *d;


/* Icache Interface Signals */
DECLARE struct j {

 /* title: Signals to Ibox */

 SIGNAL (i_ic_asm_0b_h,  1); /* ASM bit read from ICACHE Tag */
 SIGNAL (i_ic_asn_0b_h,  7); /* (6:0) ASN bits read from ICACHE Tag */
 SIGNAL (i_ic_val_0b_h,  2); /* Block Valid bits read from ICACHE Tag */
 SIGNAL (i_ic_pa_0b_h,  1); /* Physical Address bit read from ICACHE Tag */
 SIGNAL (i_tag_par_0b_h,  1); /* ICACHE tag parity read from ICACHE Tag */
 SIGNALW (i_ic_tag_0b_h,     43,W2); /* (42:0) --> ICACHE only sends 30 bits (42:13). Tag read from ICACHE */
 SIGNAL (i_ic_asm_0b_l,  1); /* ASM bit read from ICACHE Tag */
 SIGNAL (i_ic_asn_0b_l,  7); /* (6:0) ASN bits read from ICACHE Tag */
 SIGNAL (i_ic_val_0b_l,  2); /* Block Valid bits read from ICACHE Tag */
 SIGNAL (i_ic_pa_0b_l,  1); /* Physical Address bit read from ICACHE Tag */
 SIGNAL (i_tag_par_0b_l,  1); /* ICACHE tag parity read from ICACHE Tag */
 SIGNALW (i_ic_tag_0b_l,     43,W2); /* (42:0) --> ICACHE only sends 30 bits (42:13). Tag read from ICACHE */

 SIGNALW(i_istr_data_0b_h,  128,W4); /* (127:0) data bus, IB input from Icache or Refill Buffer */
 SIGNAL (i_predecode_0b_h,      20); /* (19:0) Predecodes from IC/Refill Buffer */
 SIGNAL (i_data_par_0b_h, 2); /* (1:0) data parity, predecode parity */

 SIGNAL (i_br_hist_0b_h,  8); /* (7:0)  Branch History Bits read from BHT */

 SIGNAL (i_ic_srom_out_xx_h, 1); /* Serial SROM output from ICACHE (probably @ BHT end)
                                                  NOTE: This may go to CBOX directly      */
 SIGNAL (j_bht_idx_zb_h,  13); /* Index copy. For BiSt/FRC logic */
} *j;
```

```
/*
** Function prototypes for functions defined in EV5CHIP.C
*/

void ev5chip_init(); /* calls box level init routines */
void ev5chip_main(); /* calls box level main routines */

/* trailer */
#undef DECLARE
#endif
```

9.2 Revision History

Table 9–1: Revision History

Who	When	Description of change
your name	date	description