MIT Concurrent VLSI Architecture Memo 60

Massachusetts Institute of Technology
Artificial Intelligence Laboratory

M-Machine Microarchitecture v1.1[1]

William J. Dally, Stephen W. Keckler, Nick Carter,
Andrew Chang, Marco Fillo, Whay S. Lee

August 19, 1994

Abstract

This document describes the microarchitecture of the MIT M-Machine. It details the machine's organization in terms of arithmetic units, switches, buses, memories, and control units. The function of each logic block is described, as well as the coarse interfaces between the different blocks. The programmer's view of the M-Machine is described more clearly in *M-Machine Architecture v1.0*.

[1]The research described in this paper was supported in part by the Defense Advance Research Projects Agency of the Department of Defense under contracts N00014-88K-0738 and F19628-92C-0045 and in part by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation and IBM Corporation.

Contents

Chapter 1

Introduction

This document describes the microarchitecture of the MIT M-Machine. It details the machine's organization in terms of arithmetic units, switches, buses, memories, and control units. The function of each logic block is described, as well as the coarse interfaces between the different blocks. The programmer's view of the M-Machine is described more clearly in *M-Machine Architecture v1.0*.

In the rest of this chapter an overviev of the MAP chip is presented and the 6 major functional modules of the MAP are identified: cluster, cache bank, EMI, Cluster-Memory Switch, Intercluster Switch and network interface. Chapter 2 presents the general architecture of the MAP cluster. and serves introduction to the following chapters 3, 4, 5, 6, 7, 8, 9, each one describing the microarchitecture of a cluster submodule.

Chapters 10 and 11 describes the two interconnections modules of the MAP chip: C-switch and M-switch. Chapters 12and 13 describe the memory system; the first discusses the design of the cache banks and the second the organization of the external memory interface.

Chapter 15 focus on the MAP network interface, chapter 17 deals with the config space controller, and 18 with the diagnostic interface. The last two chapters 19 and 20 describe issues global to the map chip namely the clock strategy and the overall system design.

1.1 Overview

The M-Machine consists of a number of nodes connected by a 3-D mesh interconnection network. Each node is made up of a multi-ALU processor (MAP) chip and five high-bandwidth dynamic memory chips.

A block diagram of the MAP chip is shown in Figure 1.1. It consists of the following subsystems:

- Arithmetic clusters (4).

- Cluster switch (C-Switch).
- Global control unit (coordinates operation of the clusters and the C-Switch).
- Cache memory banks (4).
- External memory controller with local translation lookaside buffer (LTLB).
- Memory switch (M-Switch).
- Network input controller.
- Network output controller with global translation lookaside buffer (GTLB).
- Network router.

In addition, 4 hardware *thread slots* are provided to allow up to 4 user level threads to run simultaneously. Two additional thread slots are provided for exception and event handling. The following is a brief description of the function of each of the components of the MAP chip. Each subsystem will be described in more detail in the subsequent sections.

Cluster: Each cluster on the MAP chip executes instructions independently and threads running on them can communicate with one another via memory or directly through registers. An M-Machine *instruction* consists of 3 *operation* fields, one for each operation unit within a cluster. A cluster module contains the arithmetic processing elements which issue operations. Each cluster contains an integer operation unit, a floating-point operation unit, a memory interface unit, an integer register file, and a floating-point register file. Data may be transferred from one cluster to another by executing an instruction that has a remote destination register. The result is written directly into a remote register file via the Cluster Switch (C-Switch). The memory unit issues load and store requests to the memory system via the Memory Switch (M-Switch) which routes requests to the appropriate cache bank. Messages can be transferred between M-Machine nodes using the *send* instruction which transmits the data using the interconnection network. A message is composed in registers and is sent across the C-Switch to the Network Output module which is responsible for depositing it into the network.

The exception thread slot is reserved for handling exceptions that can be detected when an instruction is in the EX stage of the pipeline. This include protection violations and arithmetic errors. The event thread slot is used for handling asynchronous events that cannot be detected immediately. Each event thread has an event queue which hold events that are waiting to be handled. The clusters are dedicated to particular functions in the following way:

- Cluster 0: local TLB misses.
- Cluster 1: block status faults, synchronization misses, thread timeout

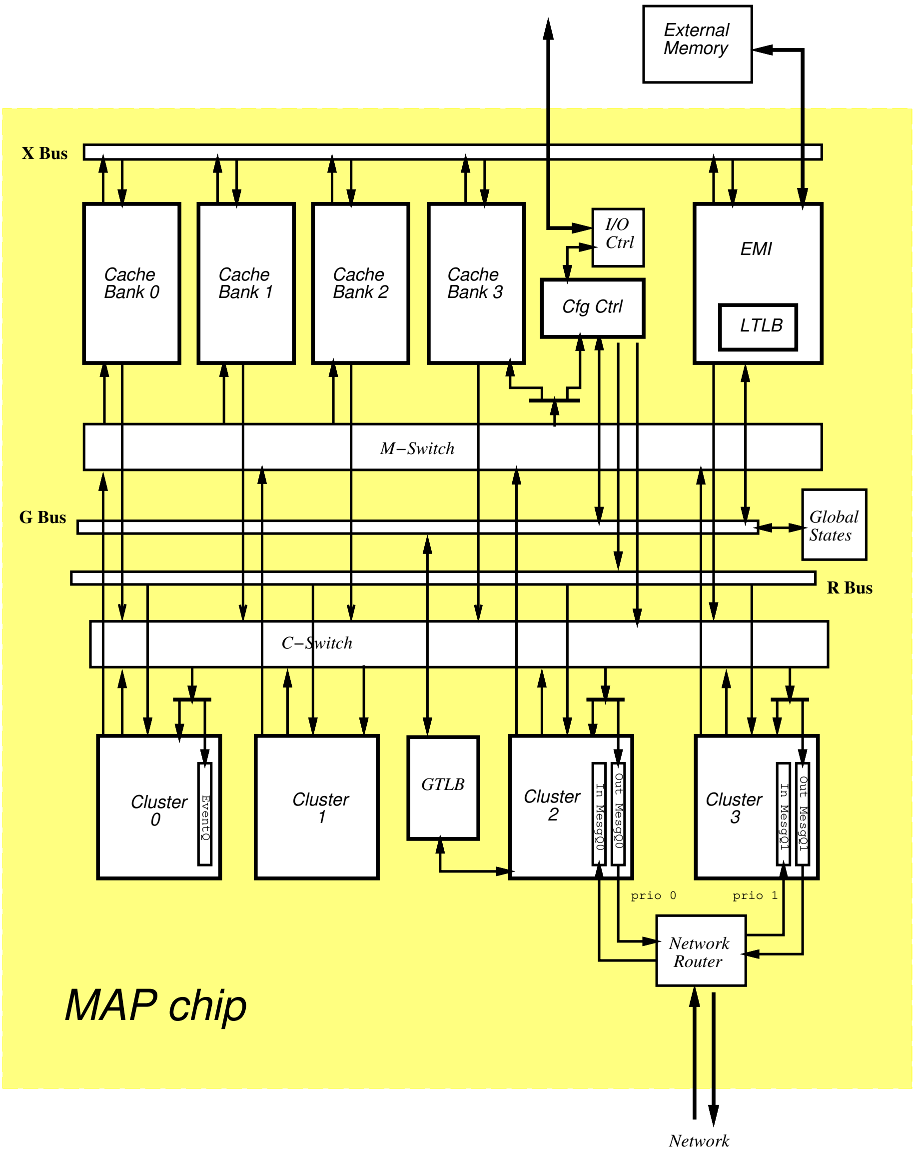


Figure 1.1: Main components and functional blocks of the Multi-ALU Processor (MAP) chip.

- Cluster 2: priority 0 message arrival
- Cluster 3: priority 1 message arrival

Dynamic scheduling, similar to Processor Coupling [1], is employed to exploit instruction level and inter-thread parallelism within a cluster. Threads are interleaved in time over the cluster execution resources on a cycle by cycle basis, depending on data availability.

Global Control: The Global Control Unit (GCU) is responsible for transmitting globally shared control data, such as the global condition code register values, to all of the clusters. In addition, it houses the configuration space controller which provides memory mapped access to internal state registers of the MAP.

C-Switch: The Cluster Switch (C-Switch) is a crossbar switch with 4 buses. Each bus is connected directly to both of the register files in a cluster. In addition, a fault buffer is attached to one of the C-Switch busses. There are 13 possible sources that may drive a bus: 4 integer arithmetic units (one per cluster), 4 floating-point arithmetic units (one per cluster), 4 cache banks, and the external memory interface. Data requested by load instructions are transmitted to register files via the C-Switch. The C-Switch includes arbitration logic for these buses. The priority is fixed in hardware with the cache banks having the highest priority. Each C-Switch bus includes 66 bits for data, 1 bit to specify a destination register file (integer or floating-point) within a cluster, and 5 bits to specify the destination register.

M-Switch: The M-Switch routes memory system requests from the processor clusters to the Cache Banks and the External Memory Interface. The configuration space controller sits on one of the cache bank M-Switch busses as well. The M-Switch is constructed in much the same manner as the C-Switch, containing five buses, one connected to each of the cache banks, and one connected to the External Memory Interface. Each M-Switch bus consists of 61 bits of addressing and control and 66 bits of data. The M-Switch performs arbitration, only allowing one of the input units to drive each of the buses on a given cycle. The arbitration scheme is fixed in hardware, with round robin priority given to the different clusters.

Cache Bank: The four Cache Banks are used as a cache memory by the MAP. Each bank contains 4KW (32KB)of memory as well as sufficient memory to store the cache tag and presence bits. Addresses are interleaved across the Cache Banks in order to reduce bank conflicts due to multiple processor clusters attempting to access the same bank during the same cycle. Each Cache Bank may also access the TLB and external memory interface (EMI) so that cache misses may be handled in hardware. Arbitration for the TLB and EMI is also performed in a round robin fashion.

In addition, each cluster has an instruction cache which holds those instructions that are to be executed in the cluster. The global instruction cache size, which is the sum of all of the cluster instruction caches, is 4KW (32KB).

External Memory Interface: The External Memory Interface handles communication between the MAP chip and the local off-chip memory. An M-Machine node contains 1 MWord of off-chip memory, but has sufficient address lines to support up to 256MWords per MAP chip. The M-Machine EMI supports error correction coding (ECC with 1 bit correction and 2 bit detection) and thus requires 5 16Mbit high-bandwidth DRAM chips for 1MWord of off-chip memory. The bandwidth between the EMI and the off-chip memory is 1 word/cycle.

Local Translation Lookaside Buffer (LTLB): The LTLB caches virtual to physical address translations for data that is stored in the local off-chip memory. Page sizes in the LTLB are fixed at 512words, and the LTLB maintains 2 status bits for each cache line within a page, so that cache lines within a page may be shared across nodes. The four states include invalid, read-only, read/write, and dirty.

Global Translation Lookaside Buffer (GTLB): The GTLB caches translations between virtual addresses and node identifiers. This allows relocation of data across different nodes of the M-Machine. Page sizes in the GTLB are variable, and may be any power of 2 multiple of the LTLB page size, including the full address space of the machine. The GTLB also provides support for mapping a single GTLB page across multiple nodes, allowing data structures to be distributed across the entire machine or certain subsets of the nodes.

Network Output: Messages are assembled in *message composition* registers which are distributed across the clusters. *Send* operations transmit the assembled message into the network output unit (NetOut). There the message is partitioned into flits, and the necessary control information is attached. The NetOut unit accesses the GTLB to obtain a translation from a virtual address to physical node address needed to route the message. A small NetOut buffer (nominally 32 words) is provided to match transmission rates with the network.

Network Input: The network input unit (NetIn) includes a port on the network and 2 fixed sized hardware message queues, one for each priority in the network. When a message arrives, it is deposited in the appropriate message queue; a message handling thread running in the corresponding thread slot is responsible for removing the message from the queue and handling it.

Router: Dedicated routers are employed in the M-Machine. There are 4 virtual channels in each of the X, Y, and Z directions. Two channels are reserved for priority 0 messages and 2 channels are for priority 1 messages. Ownership of the physical channels is arbitrated among the virtual channels on a flit-by-flit basis, except the NetIn (Extract) and NetOut (Inject) physical channels, which are granted on a message-by-message basis.

Additional hardware controllers are provided for:

- Fault detection and queueing of operands so that faults may be handled in software.
- Access to internal and special registers via *configuration space*.
- A diagnostic port to help boot and debug the machine.
- A peripheral I/O interface.

Chapter 2

Overview of the MAP cluster

The four clusters in the MAP chip contain instruction execution units. They are responsible for the fetching of instructions and their execution. Instruction execution may involve communication to the memory or to other clusters. As shown in fig. 2.1 the cluster contains the following units:

- **The instruction fetch module** (`IF`)
- **The Register Read module** (`RR`)
- **The Synchronization module** (`SZ`)
- **The integer execution unit**(`IALU`)
- **The FP execution unit** (`FALU`)
- **The memory execution unit** (`MEMU`)
- **The exception logic module** (`XCPT`)
- **The C-Switch output interface** (`CSWOUT`)
- **The C-Switch input interface** (`CSWIN`)
- **The M-Switch output interface** (`MSWOUT`)
- **The Configuration Space Controller** (`CFGCNTRL`)
- **The message-in/event queues** (`MEQ`)

The cluster is organized around a pipeline of macro-stages. The `IF` requests instructions to the memory banks through the `MSWOUT`. The instructions are supplied via the C-switch and are cached in the `IF` stage. At every cycle the `IF` passes to the `RR` the next instruction corresponding to the V-Thread that just issued.

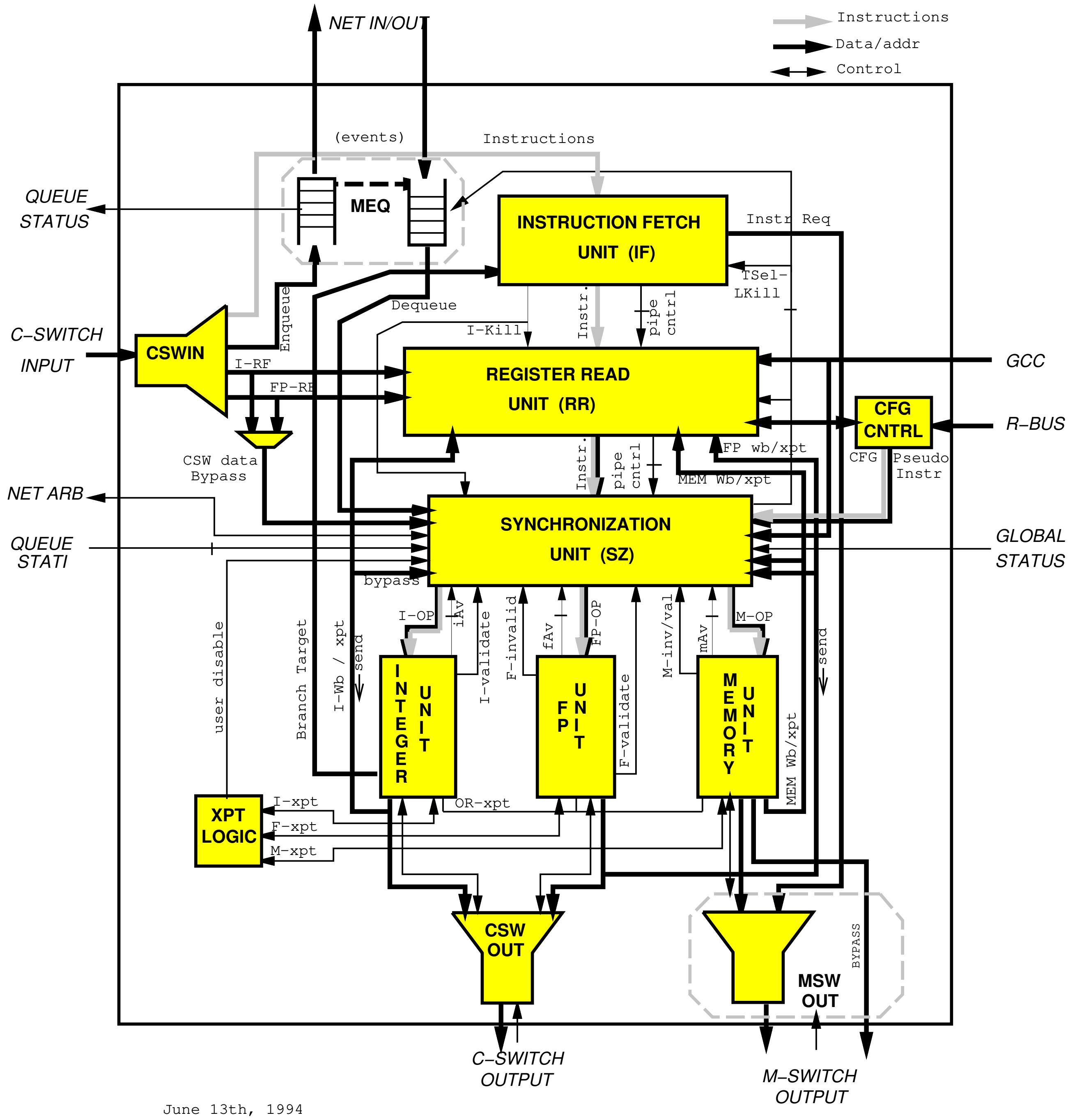


Figure 2.1: Block diagram of MAP cluster

The RR contains the integer and floating point register files. Its function is to read the operands values required by the three operations contained in the instruction, and forward them to the SZ together with the operations. It also holds a copy of the global CC registers, which may be written remotely via the GCC lines but are emptied locally.

The SZ decides from which stream to issue the next instruction out of a maximum of 6 possible V-Threads. Notifies the RR and IF of its selection so that the corresponding stream can be advanced. In the RR and SZ, hardware resources are duplicated once for every V-Thread, while the execution units are time shared among V-Thread streams. In addition to the register presence information provided by the RR, the SZ requires access to global status, such as V-Thread active bits, in order to select the instruction to execute. It also requires status information on the status of the message queues (QUEUE STATI) to ensure that a SEND operation can be successfully executed and performs arbitration (NETARB) to ensure that only one SEND operation per node is issued.

The instruction consisting of the operation triplet [int,mem,float] is split into individual operations as it passes down from the SZ to the three execution units: IALU, FALUand MEMU. The results of the executed operations are either written back to the local register file or forwarded to other clusters via the CSWOUT. A local bypass path forwards results to the subsequent operations in the stream.

The MEMU executes memory operations by generating requests to the memory banks via the MSWOUT. The result of the load operations return through the CSWIN. Clusters 0, 2 and 3 have a single M-Switch output port shared between the IF and the MEMU. Cluster one has an additional port dedicated to bypass memory requests issued by the TLB miss handler running on the system V-thread in that cluster.

A wired-OR exception line, global to the 3 execution units, provides the real-time execution stall required when an exception is encountered. The exception logic XCPT, when invoked, fills the registers of the V-thread X with enough information on the excepted operation to enable the software exception handler to resolve it. It also disables the execution of user threads (later enabled by the software exception handler).

The input from the C-Switch is demultiplexed among 4 destinations in the CSWIN. The IF receives instructions, the data from the memory banks or from another cluster is directed to either the integer or the floating point register files in the RR and also bypassed to the SZ for those operations waiting at that stage. In cluster 2 and 3 the fourth incoming port is mapped to the input of the outgoing message queue, for priority 0 and priority 1 respectively, which is the target of a SEND operation. In cluster 0 the fourth CSWIN provides enqueueing access to the event queue. On cluster 1, this demultiplexed output is not used. The two input queues from the network are mapped to register addresses of the S V-thread slot, in cluster 2 and 3. The event queue dequeueing port is mapped to a register address in the S V-thread slot, in cluster 0.

Finally the CFGCNTRL provides support to access to the machine state through config space load and stores, which are routed via the R-bus to the appropriate cluster. Writes to the cluster state is done directly by the CFGCNTRL. Because the MAP architecture does not support remote reads, reads of registers or internal counters is emulated through the generation of pseudo-instructions which are inserted in the instruction stream and cause a write of the local value to a remote location.

The individual modules that constitute the MAP cluster are described in the appropriate chapters of this document.

Chapter 3

The Cluster Instruction Fetch Unit

3.1 Block Diagram

Figure 3.1, attached, shows a top-level block diagram of the instruction fetch unit (IF). The unit consists of the instruction cache data and tag arrays, operation buffers, IP logic, and FIP logic.

3.2 External Interfaces

The IF unit provides four main interfaces: instruction output, thread select, branch control, and miss handling.

The signals involved are:

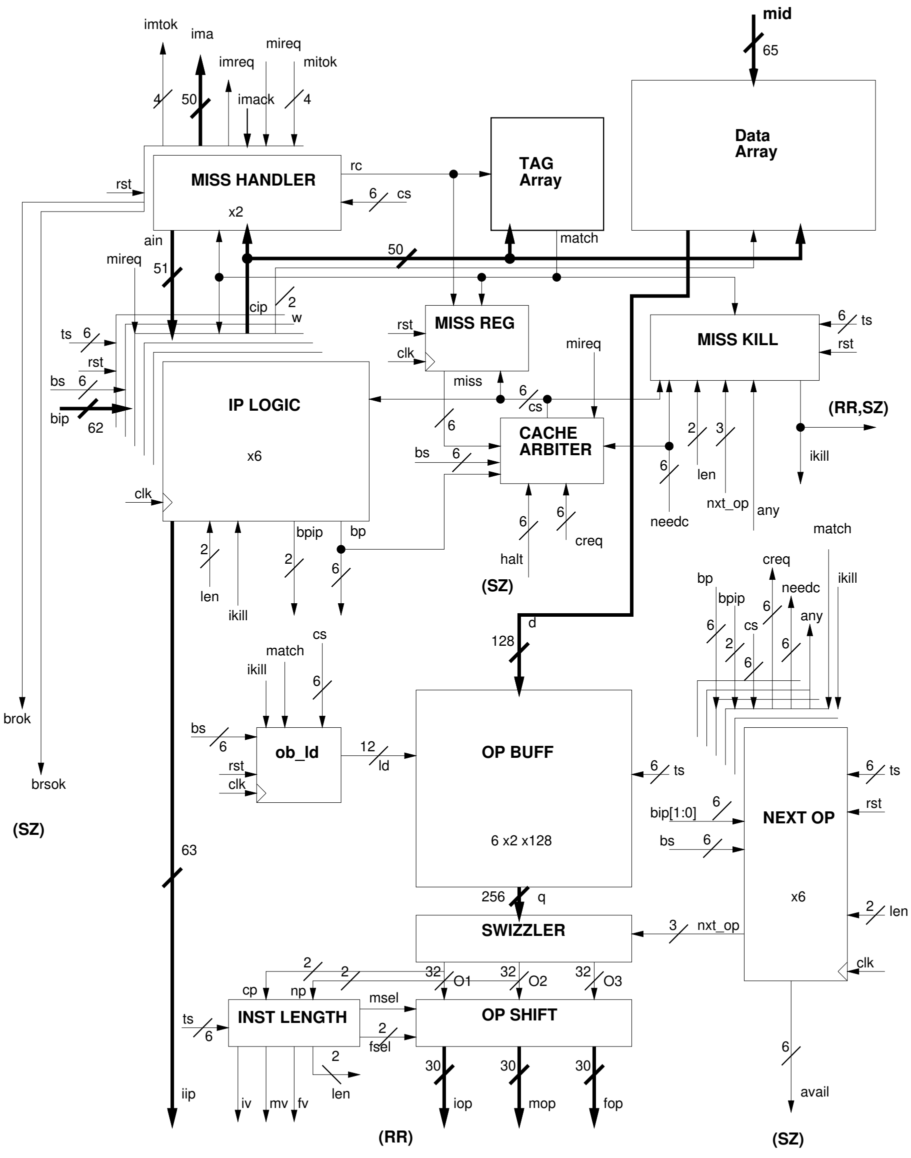


Figure 3.1: *IF Stage Overview*

Thread Select:

EMPTY[0:5]	thread operation buffer empty signals. A high bit indicates that the corresponding thread attempted to issue an instruction and failed because there were insufficient operations (not necessarily zero operations) in the operation buffer. A bit is cleared when a new line is loaded into the buffer. Empty is an output from the I-stage to the SZ-stage.
HALF_FULL[0:5]	thread operation buffer status signals. A high bit indicates that there are four or more operations in the corresponding thread's buffer and that the thread is guaranteed to be able to issue. HALF_FULL is an output from the I-stage to the SZ-stage.
TRUN[0:5]	thread run enable from the sync stage to the I-stage. When a bit is set, the corresponding thread is enabled to fetch instructions.
TSEL[0:5]	decoded (one-hot) thread select from the sync stage to the IF and reg stages. If one bit is high, that thread will be selected to output an instruction. If all bits are low, no instruction will be output.

Instruction Output:

IOP,MOP,FOP[0:29]	Integer, Memory Unit, and Floating operations, respectively. From the IF to the register stage.
IV,MV,FV	Integer, memory, and floating operation valid signals. If the signal is true, the corresponding OP signal carries a valid operation
INST_VALID	Instruction valid, if true, the OP lines carry a valid instruction.
INST_ILLEGAL	Illegal instruction,
IIP[0:61]	From IF to reg stage. Instruction pointer of the current instruction.

Branch Control:	
BOP[0:1]	Branch opcode (one of NOP, JUMP, FORK, HALT). The opcode is always applied to the thread selected by TSELD.
BIP[0:61]	IP for branch operations.

Miss Handling:	
IMREQ	IF to memory request
IMACK	IF ack from memory (request accepted)
IMADD[0:49]	Address for memory request (bits 4-53)
IMTOK[0:3]	Four bit token to track memory requests from this I-cache.
MIREQ	C-switch to IF request to return requested data.
MITOK[0:3]	Token passed through memory system.
MIDATA[0:63]	Data from C-switch.

The synchronization stage requests an instruction by providing a valid code on the 6-bit TSELD (decoded thread select signal). If the operation buffer contains an instruction for the selected thread, the instruction and its IP are presented on the IOP, MOP, FOP, and IIP signals. Each of the op signals contains a 30-bit operation (e.g. MOP) accompanied by a 1-bit valid signal (e.g., MV). The IIP contains the entire 62-bit IP (less pointer tag and low two bits). A one bit instruction valid signal qualifies all of the OP, V, and IIP signals.

On a branch or jump operation, the execution unit provides a 62-bit branch IP (BIP) (low two bits 0), and a 2-bit branch opcode. If the JUMP opcode is sent, the BIP replaces the IP for the selected thread and the operation buffer for that thread is cleared. If the branch signals are valid at the end of one cycle, an I-cache fetch will be attempted and if the address hits in the I-cache, the instruction at the branch target will be available for thread selection at the end of the next cycle. The FORK and HALT branch opcodes are used to change the thread states. [Question: should FORK and HALT use the same interface as JUMP?]

The miss handling interface arbitrates for the M-switch with the memory unit and makes memory requests to handle I-cache misses. The outgoing request interface includes signals IMREQ and IMACK that are used to arbitrate for the M-switch. The miss address is provided on IMADD. Also, a 4-bit token to identify the return data is provided on IMTOK. Data returning from the memory arrives on signals MIREQ, MIDATA, and MITOK. Data from memory always has priority, so there is no need for an MIACK signal.

3.3 Operation Buffers

The operation buffer (OpBuf) is a section of the MAP Instruction Fetch Stage (IF). The function of the operation buffer is to store and supply instructions to the downstream stages of the MAP pipeline, detect and unpack instruction from the I-Cache using the two bit format detailed in the *MAP Instruction Set Reference Manual v1.2*. The OpBuf contains logic to manage the 2 128-bit buffers reserved for each of the 6 hardware thread thread slots and interacts with the I-Cache request and control logic, Register Read (RR), Synchronization (SZ), and Integer Execution (INTU) pipeline stages.

The Operation Buffer block is diagramed in five figures: fig 3.2, fig 3.3, fig 3.5, fig 3.6, and fig 3.4. The Operation buffer contains six 256-bit operation buffers, one for each thread along with associated status and logic. Each operation buffer contains two 128-bit halves that are loaded as a unit from the instruction cache. Each 128-bit half-buffer contains four bit-interleaved 32-bit operations. Thus, the MSB (bit 31) of all four operations are next to each other at the left end of the operation buffer. This interleaving greatly simplifies the multiplexing required to delimit instructions and select operations.

In addition to the buffers themselves, the Operation Buffer block also contains state to manage the operation of the buffers. For each buffer, the following bits are kept:

Name	Bits	Description
ld	1	Which 128-bit half should be loaded next
nxt_op	3	Which of the eight ops is next in sequence
nr_avail	2	Number of valid instructions in the buffer

Figure 3.2 diagrams the operation buffer control logic.

Figure 3.3 of the Operation buffer drawings shows the basic buffer cell. This cell contains two static latches (one bit for each of the two half buffers) and two NFET pass gates. As shown on Figure 3.4, these cells are composed in a 128x6 array. The inputs of the array are the 128 data lines from the cache while the outputs are 256 lines representing the entire state of the buffer for the thread selected by *ts*

The 256 bits out of the array of cells connects to the inputs of 32 3→8 selectors (*swizzler*) as shown on Figure 3.4. The schematic for these selectors is shown on Figure 3.5. The 0,4,1,5,...,3,7 ordering of inputs due to the buffer arrangement may complicate layout slightly. These selectors select the next three 32-bit operations (if they are in the buffer) from the selected thread's instruction stream. To this end, they are driven by the NOPD[0:7] signal, the decoded *nxt_op* register of the selected thread.

The upper two bits of the operation buffer hold the "PACK" bits that specify how the operations are packed into instructions. The pack bits for the current and next operation out of the two most significant 8→3 selectors are called CUR_PACK and NEXT_PACK. The third output from these two selectors is unuused. The "PACK" bits are used to calculate: instruction length (LEN), whether the instruction is valid (do we have enough ops in the buffer), if this instruction includes I, M, and/or F ops, and if so where the I, M, and F ops

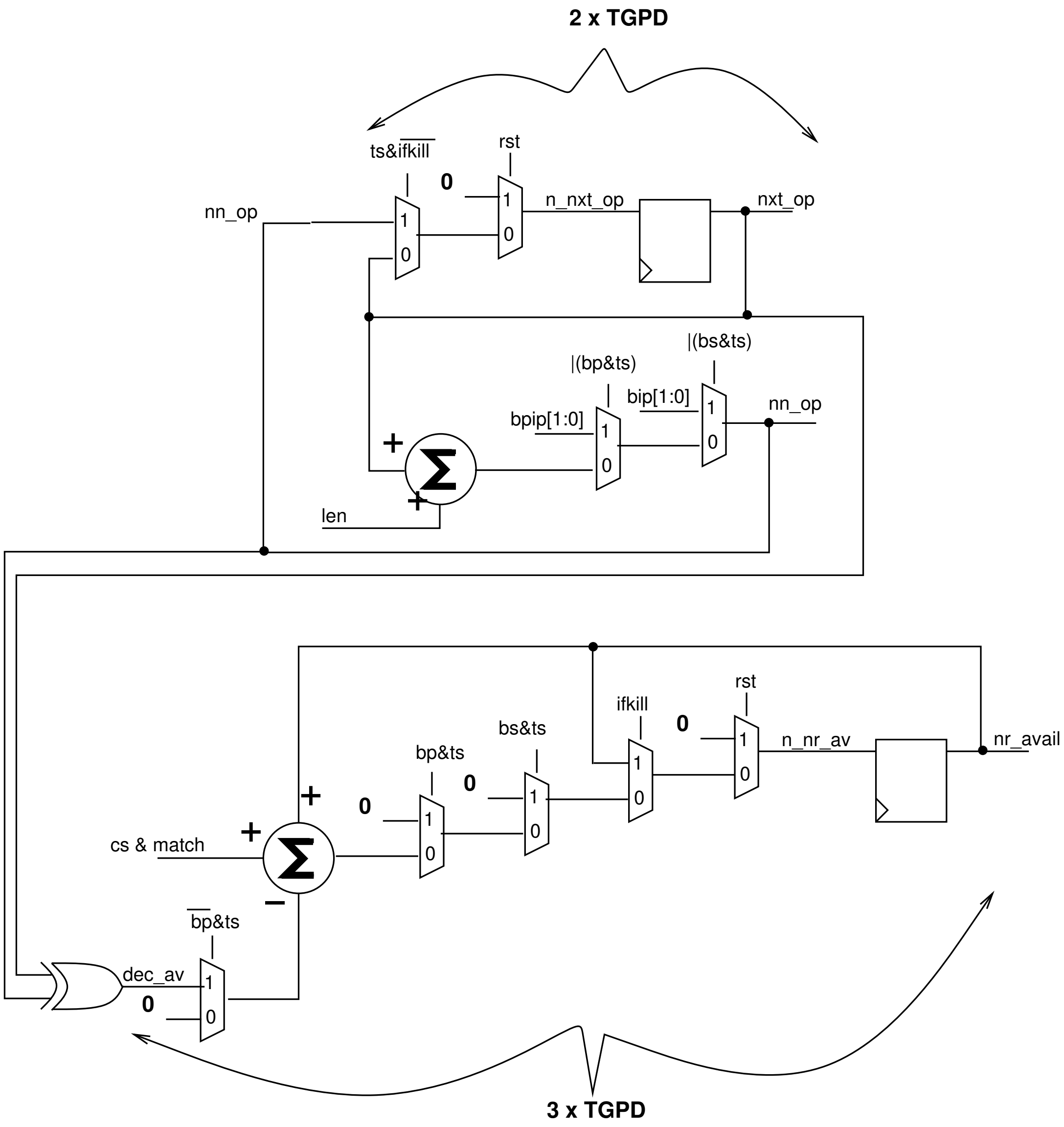


Figure 3.2: *Operation Buffer Control Logic*

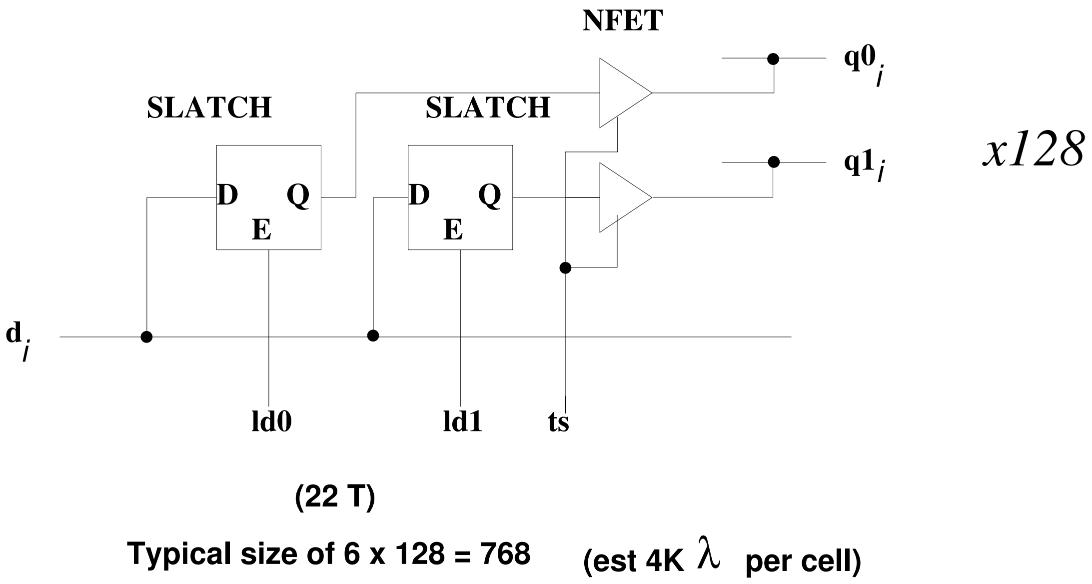


Figure 3.3: *Operation Buffer Cell*

are.

Figure 3.6 diagrams the packing/unpacking control logic.

The instruction length is calculated from:

$$inst\_length[1] = cur\_pack == \text{`}P1\_COMPOUND$$
$$inst\_length[0] = \; inst\_length[1] \mid next\_pack == \text{`}P2\_IMF$$

With our current assignments, both immediates are 0x3, so this logic consists of a NAND gate, an inverter, and a 3-input AND-OR invert.

Then the presence of each of the three operations (IV, MV, and FV) is determined from the PACK fields and their positions are calculated. The positions are encoded in a 5-bit OPSH (operation shift) signal that is used to drive the 30 operation shifters at the end of the data path. The output of the shifter is the IOP, MOP, and FOP signals. These signals are buffered and sent to the register (RR) stage of the pipeline.

When *nr_avail* falls to four or less, the Operation Buffer signals the Fetch Logic that buffer space is available by asserting creq (this is just the inversion of the high bit of *nr_avail*). An I-cache access may then be attempted. A fetch to a particular thread is signaled by the Fetch Logic with signal *cs*. If it is successful, as signaled by the tag array on signal *match*, a 128-bit block is loaded into the buffer indicated by *ld*, *ld* is complemented, and *nr_avail* is incremented by four (MSB toggled).

On a branch, if the branch cache access succeeds, *nr_avail* is set to four and *nxt_op* is set to zero (branch targets are always loaded to half buffer zero). If the cache access fails, *nr_avail* is

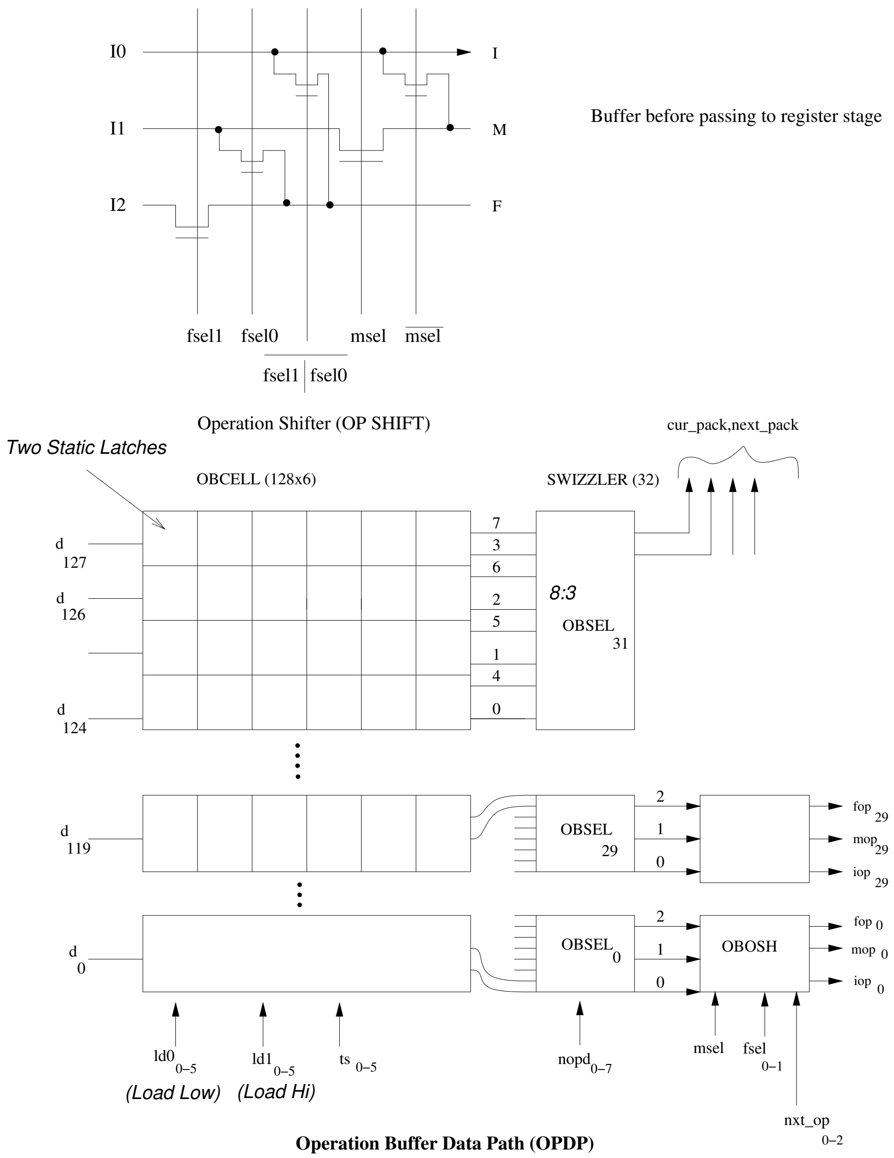


Figure 3.4: *Operation Datapath*

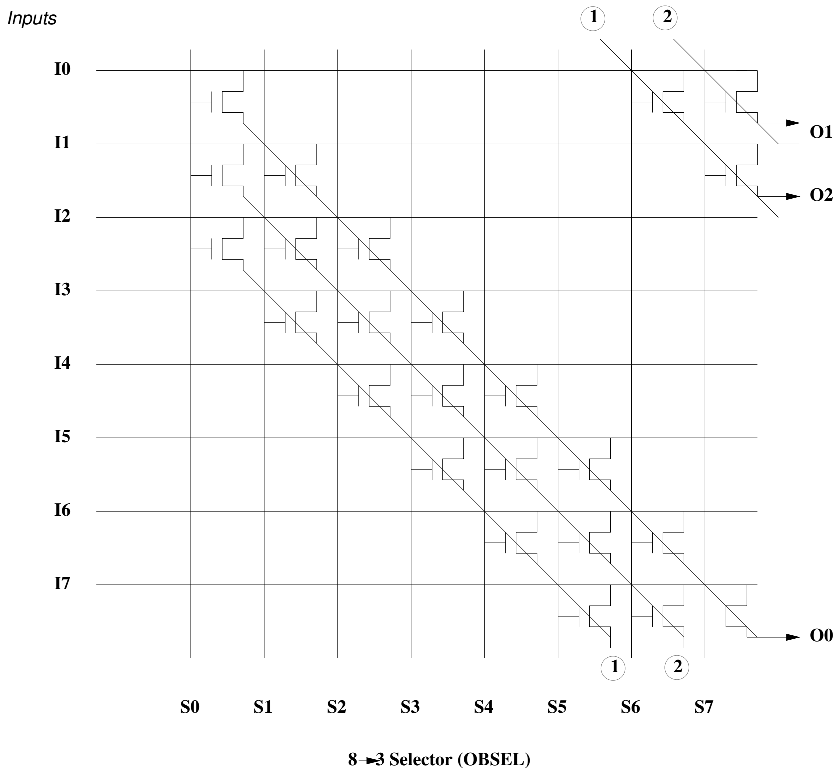


Figure 3.5: *Operation Buffer Selector (Swizzler)*

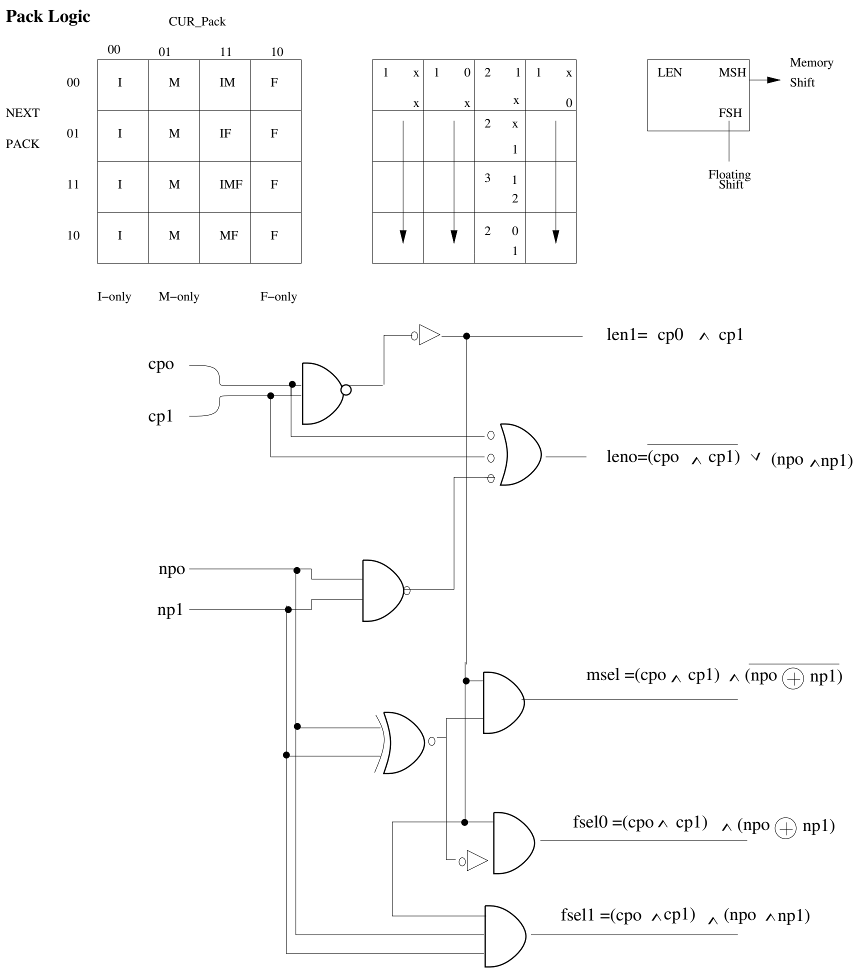


Figure 3.6: *Packing/Unpacking Logic*

set to zero to stop the instruction stream until the branch target can be fetched. The branch is signalled by input *bs* from the Integer Execution stage (INTU). Cache access success is determined by *cs* and *match*.

The *nr_avail* and *nxt_op* logic is shown in figure 3.2.

For now, no attempt is made to handle a short branch in the operation buffer.

3.4 IP Logic

The IP logic contains six IP registers of 62-bits each. The pointer bit and low two bits are dropped.

When the synchronization stage signals a valid TSEL, the IP Logic outputs the selected IP on the IIP lines so that it flows down the pipeline with the instruction. The IP Logic passes the TSEL to the Operation Buffer and increments the selected IP by the INST_LENGTH value returned from the Operation Buffer.

On a branch, the BIP overwrites the IP selected by BTSEL. This overwrite takes priority over an increment.

3.5 Fetch Logic/Miss Handler

The fetch logic attempts to keep the operation buffers full by fetching (128-bit) double words from the instruction cache.

The fetch logic maintains three state bits for each of the six threads to encode the following four states:

run	no unsatisfied i-cache miss
miss	i-cache miss has occurred, but refill has not yet been requested.
pend	a refill operation is pending on this thread.
inactive	no thread running in this slot

To keep track of instruction locations, the Fetch Logic maintains six FIPs. Only 50bits are required per FIP as the type and length fields are not used and the low order four bits (byte in 128-bit half buffer) are not needed.

Also, the fetch logic maintains two three-bit sequence/token counters that are used to sequence addresses and count replies when handling misses. Two three-bit miss-thread and two two-bit miss state latches are also used. The miss-thread latches match up miss data returning from memory. The miss-state latches encode the states idle, requesting, and receiving. Each sequence counter, thread, and state latch set is called a miss handler.

A fetch is initiated for an active (state == run) thread that has buffer space available (as signalled by the Operation Buffers.). A simple fixed two-priority arbitration is used to select among eligible threads. Priority is given to threads with LENGTH_OK false. The selected thread is signaled to the Operation buffer with a three-bit FTSEL signal. I-cache writes (see below) take priority over fetches.

On a fetch, the low 9 bits of the FIP (bits 4-12) are hashed (by XORing with some high bits) and then used to access the data array. The high seven of the 9 bits are used to access the tag array. The remaining 41 bits of FIP (bits 13-53) are sent to the tag array for comparison. The tag array returns a hit or miss signal to the Operation Buffer and Fetch Logic.

On an i-cache miss, a thread enters the miss state, or pend state depending on need and miss handler availability. If the thread still has valid operations in its buffer (LENGTH_OK) or if the miss handler is already busy, it enters the miss state. Whenever the miss handler is available and the thread needs the instructions, the thread enters the PEND state and a refill is initiated.

When a refill operation starts, an idle miss handler (one of two) is allocated by writing the thread's number into the miss-thread latch and setting the miss-state to requesting. Only one of the two miss handlers can be requesting at a time. The requesting miss handler asserts IMREQ and places drives IMADDR with the concatenation of FIP (minus two low bits) and the sequence counter. The sequence counter with the miss-handler number appended is used to drive IMTOK. Each time arbitration is won as signaled by IMACK, the sequence counter is incremented until it reaches seven.

When an IMACK is received while the sequence counter is seven, the sequence counter is cleared and the miss-state is set to receiving. In the receiving state, the miss handler monitors the MIREQ, MIDATA, and MITOK lines. Data returns over the C-switch with the same token as the request that went out over the M-switch. When MIREQ is asserted the high bit of MITOK selects the miss handler to handle the returning data. The selected miss handler initiates a 64-bit I-Cache write using the FIP selected by the miss-thread latch and the MITOK field for the address. The MIDATA field is written to the selected half of the selected data array line. The high 41 bits of the FIP are written to the tag array, and the valid bit is cleared unless the sequence counter equals seven in which case it is set. On each MIREQ to a given miss handler, the sequence counter is incremented. When the final MIREQ arrives with the sequence counter at seven, the sequence counter is cleared, the miss-state is set to idle, and the thread state is set to run.

3.6 Open Issues

How do we set a thread state to RUN from INACTIVE? Need a way to do this on HFORK and return on EXIT.

Probably the best way is to provide another interface for C-Switch operations.

Chapter 4

The Cluster Register Read Stage

4.1 Functional Overview

The RR stage of the MAP pipeline performs instruction predecode, scoreboard updates and register reads. The RR stage is responsible for providing the source operands for each operation. For each of the three execution units, the RR stage performs the same functions. However, the implementation of of the RR stage varies to accommodate the differences between execution units.

Figure. 4.1 shows an overview of the MAP chip pipeline. The pipeline for the MAP chip is composed of five stage:

1. Instruction Fetch (IF) - Responsible for fetching and unpacking instructions, managing the ICache and managing the Instruction Pointers (IPs).
2. Register Read (RR) - Responsible for decoding instructions, performing sign extension on immediates, providing source operands, and managing the Register Files and Scoreboards.
3. Synchronization (SZ) - Responsible for selecting instructions to issue, arbitrating for the Network resource, and generating the thread select (TSEL) control signals.
4. Execution (EX) - Responsible for executing operations, detecting exceptions, arbitrating for CSW and MSW resources, stalling the pipeline, validating and invalidating registers, and creating Errval's.
5. Writeback (WB) - Responsible for writing results to the local register file, for writing results to the CSW for remote destinations and updating the scoreboard.

Figure 4.2 shows the high level logical block diagram of the RR stage. Logically, the MAP pipeline contains one shared RR function for all the execution units. The actual implementation will be distributed. The INTU RR stage will have some added functionality to handle global issues.

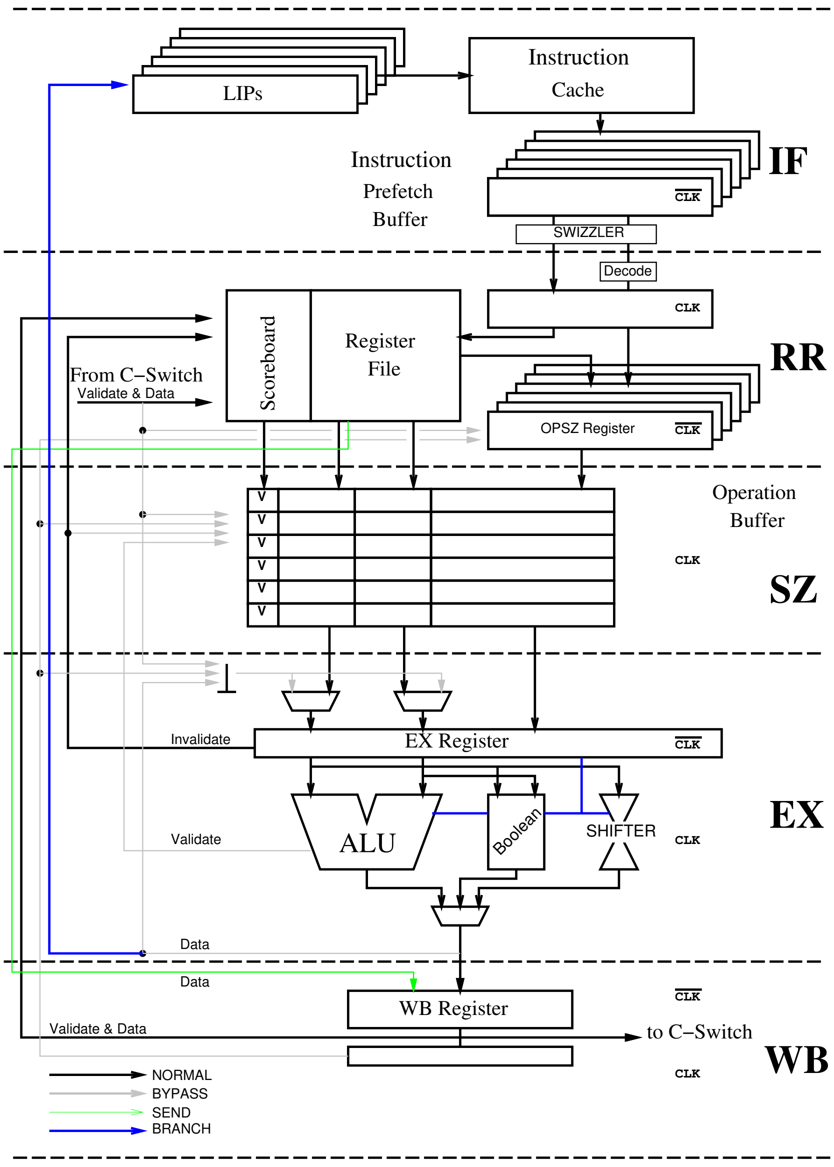


Figure 4.1: *The MAP Pipeline*

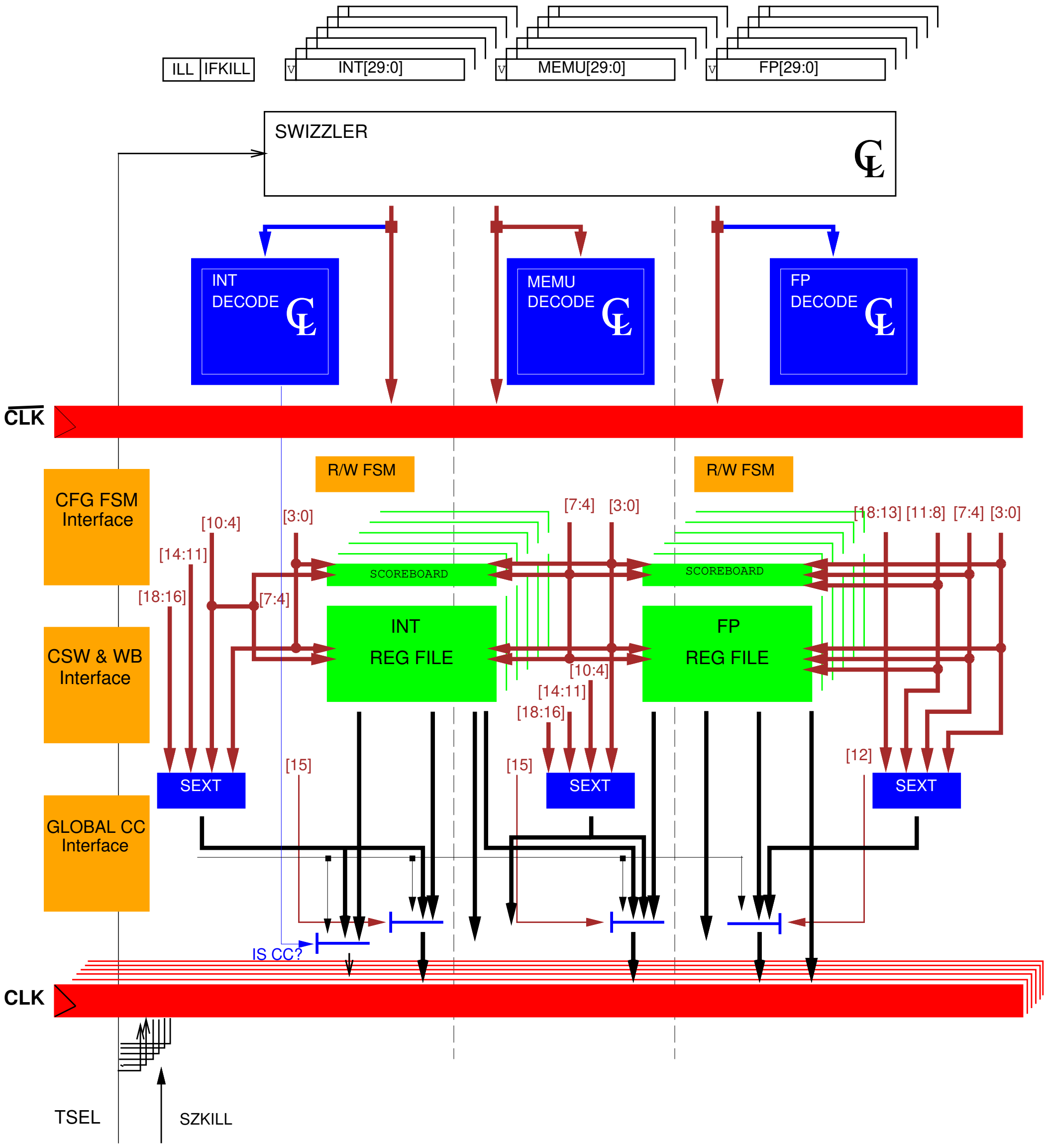


Figure 4.2: *RR Logical Overview*

4.1.1 Function

Specifically, the RR stage is responsible for the following:

- Read the appropriate Register Files Based on Operation
- Multiplex Input Sources (including Bypass paths) to OPSZ Register
- Perform "Predecode" of operation field
- Set Valid Bits in OPSZ Register
- Perform Sign Extension on Immediates
- Destination for validation/invalidation of Scoreboard Entries
- Destination for updates to Register File Data
- Provide message data from the Register Files to the WB stage during SEND operations.

4.2 Interfaces

The RR stage has six interfaces which are itemized below and detailed in the subsequent figures.

1. Instruction Fetch (IF/RR) - Operation Fields for INTU, MEMU FPU

   160 bits x 6 Registers = 960 bits

2. Synchronization (RR/SZ) - OPSZ Pipeline Register for Operations

   752 bits x 4 Registers + 756 x 2 Registers

3. Execution Stage (RR/EX) - Invalidation and Register addresses for SENDs

   29 bits

4. Write Back Stage (RR/WB) - Source of Validation/Invalidation and Data during normal operations. Destination for register data during SENDs

   294 bits

5. Cluster Switch (RR/CSW) - Validation and Data

   TBD

6. Configuration Space FSM (CFG/RR) - Pseudo-Operations

   TBD

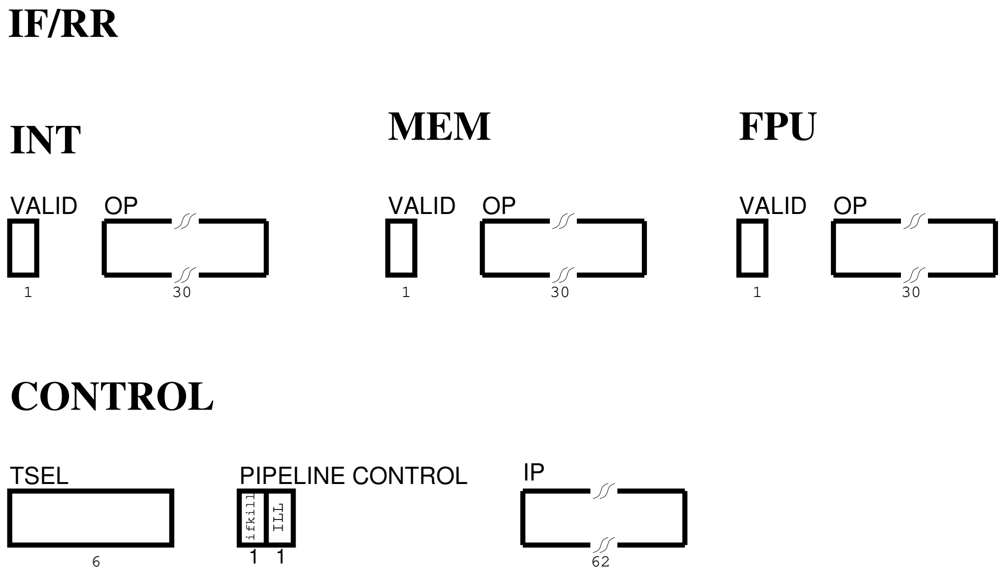


Figure 4.3: *IF/RR Interface*

4.2.1 IF/RR Interface

Figure 4.3 presents the bit fields of the IF/RR Interface. The IF stage passes the 30-bit operation for each execution unit to the RR stage (32 bit operation less the 2 packing bits). In addition, the IF stage generates a valid bit (VAL) for each operation as well as a late kill line (IFKILL). and an Illegal Bit (ILL). The TSEL lines are bussed to all the pipeline stages. Finally, the IF stage provides the corresponding IP with the instruction.

The following combinations have meaning:

- VAL & $\overline{IFKILL}$ = Non-NOP operation for the function unit.
- $\overline{VAL}$ & $\overline{IFKILL}$ = NOP operation for the function unit.
- ILL = Illegal operation caused by branch to ErrVal

4.2.2 RR/SZ Interface

Figure 4.4 presents the bit fields of the RR/SZ Interface. The interface drawing details the OPSZ pipeline register format. There are five 752-bit registers, one each for the four user threads and the exception thread. The system thread has a 756-bit since it has 2 additional MC-Present lines from the INT register file and the FP register file. The detailed breakdown and count for each function unit is detailed in the figure.

The OPSZ pipeline register is composed of the following fields:

- Decoded Operation
- Source Operand Addresses, Sync status, and Values
- Destination Register Addresses
- Send Length and accompanying MC register status
- MBAR flag indicating that a CFG pseudo operation should decrement the MBAR counter

4.2.3 RR/EX Interface

Figure 4.5 presents the bit fields of the RR/EX Interface. The RR/EX interface is used to invalidate destination registers in the scoreboard and to bypass invalidations to the OPSZ buffer for EMPTY and ISND in the INTU and for cycle operations in the MEMU and FPU.

4.2.4 RR/WB Interface

Figure 4.6 presents the bit fields of the RR/WB Interface. The RR/WB interface is used to write data and validate of destination registers in the register file and the scoreboard, to bypass data and validation to the OPSZ buffer.

The READ bit signals the direction of data travel over the interface: to the Register Files during normal operation, from the Register Files during SENDs.

4.2.5 RR/CSW Interface

The RR/CSW interface remains to be defined.

4.2.6 RR/CFG Interface

The RR/CFG interface remains to be defined.

4.3 Operation

The RR stage performs "predecode" for each instruction, multiplexes the required inputs sources, manages the register file and the scoreboard. The integer unit RR stage is detailed in figure 4.7.

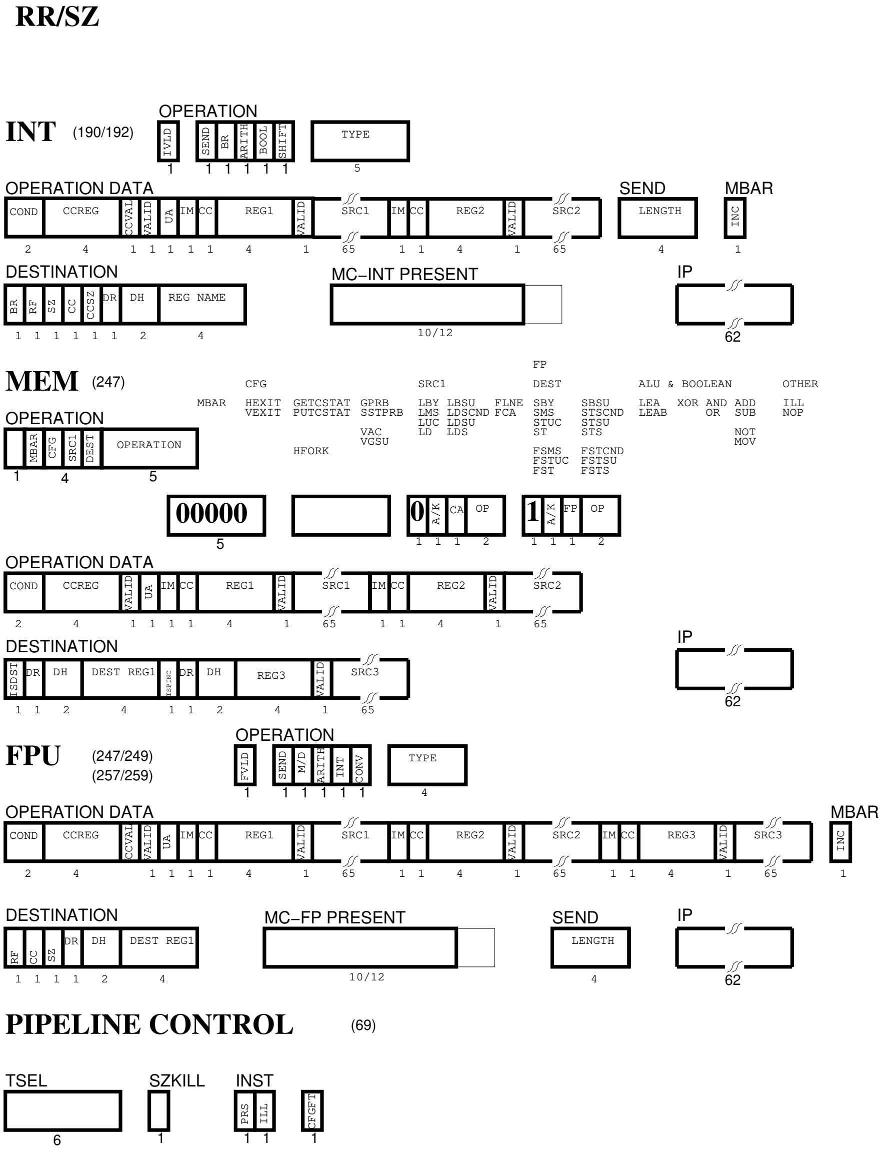


Figure 4.4: *RR/SZ Interface*

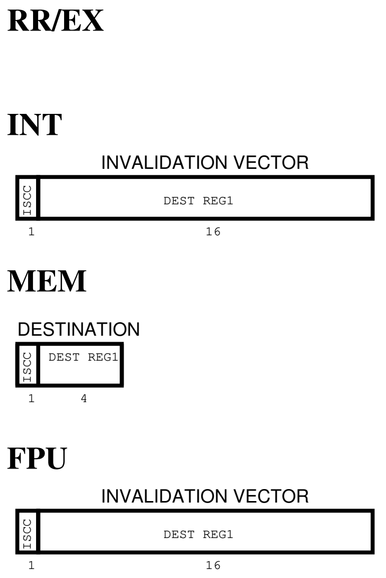


Figure 4.5: *RR/EX Interface*

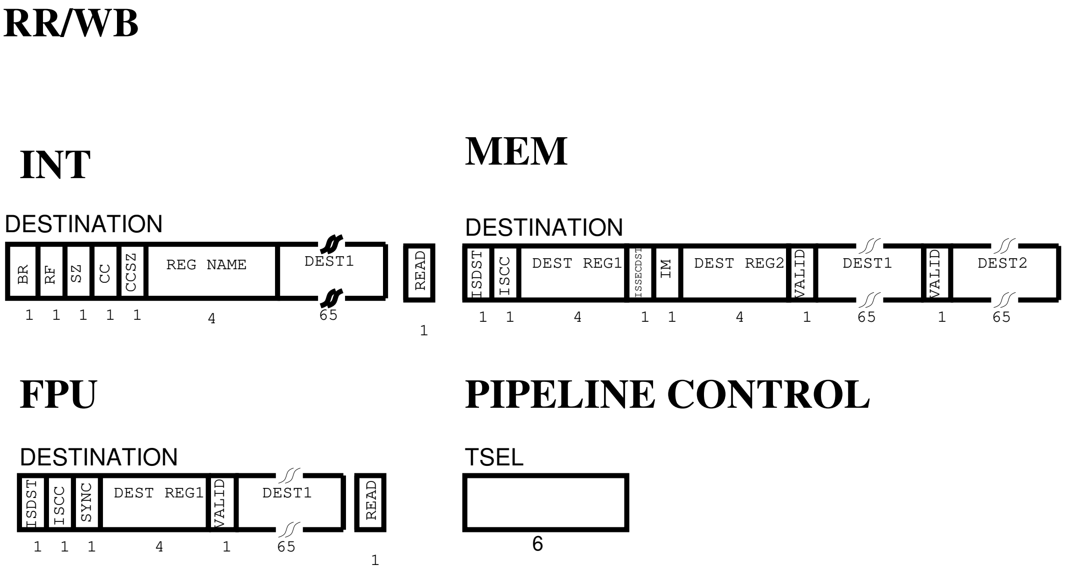


Figure 4.6: *RR/WB Interface*

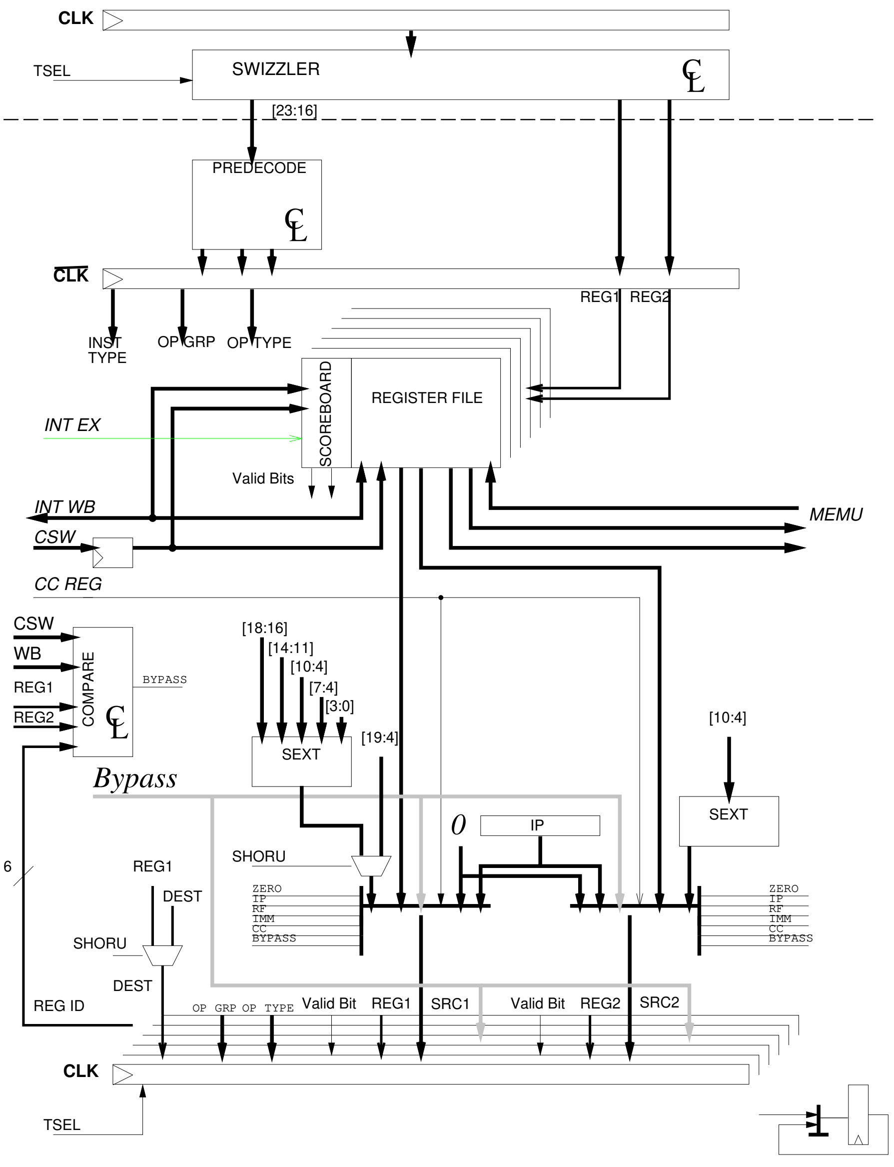


Figure 4.7: *IU RR Logical Block Diagram*

SRC1

Zero =	i0 + IMM
IP =	i1
RF =	i[15:2]
IM SEXT =	b[15] • (TYPE B + TYPE G)
CC =	TYPE E
BYPASS =	BYPASS

SRC2

Zero =	i0
IP =	i1 + BR
RF =	i[15:2]
IM SEXT =	b[15] • (TYPE A1 + TYPE C + TYPE P)
IM =	SHORU
CC =	TYPE E
BYPASS =	BYPASS

Figure 4.8: *Input Sources*

Figure 4.8 details the possible sources for operands in each integer unit operation.

M-Machine instructions follow the standard three address format. The instruction bit encodings ensure that SRC1, SRC2, and the Immediate indicator always reside in the same bit positions. DEST is in the same location for all operations except SHORU, where it is actually in SRC1.

Therefore, the RR stage *always* attempts to read the register addresses indicated by the SRC1 and SRC2 bit fields. In the case of single source instructions or instructions using immediates, the proper operand is multiplexed in before being latched into the OPSZ Register and the values resulting from the read are discarded. This policy was chosen because the Register File access time is in the critical path and any logic added to determine whether to perform a register read would directly add to the stage delay.

During normal operation, the RR stage receives an instruction from the IF stage based on the value of the thread select lines (TSEL) asserted by the SZ stage. Each instruction is composed of the following:

- Three 30 bit operations (INT, MEM, FPU).
- Three (3) valid bits (one for operation) indicating whether there is an actual operation for the relevant execution unit.
- Two (2) control bits indicating Illegal instruction and IF late kill
- Associated Instruction Pointer IP

The RR stage *always* performs each of the following tasks to gather the required operands for an instruction:

- Reading the Register File
- Performing Sign Extension on the bits in the immediate field of the instruction.

The RR stage decodes the 5-8 Bit operation field to determine whether to use the Register File Values, sign-extended immediate, unsigned immediate, IP, Condition Code Register Values, or Zero as input operand value. In addition, bypassed data and validation information from the CSW, EX, or WB stages are also multiplexed into the OPSZ register.

The RR stage maintains 6 OPSZ Buffers, one for each thread. The OPSZ buffers are bit interleaved. The SZ stage asserts TSEL to select operations for the desired thread.

Timing: The RR stage employs a *clk* transparent latch between the output of the IF Swizzler and the Register File. Most of the Instruction Decode is performed before this latch. The results from the Register File are latched into the OPSZ buffer by a $\overline{clk}$ transparent latch.

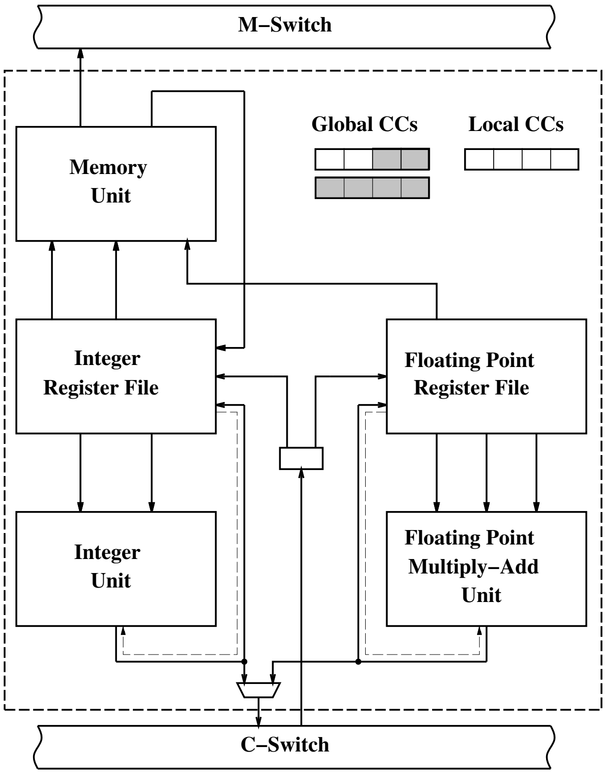


Figure 4.9: *Register File Overview*

4.3.1 Register File

The M-Machine architecture provides each VThread with a unique integer register file and a unique floating point register file. Thus, each cluster of the M-Machine contains six (6) integer register files and six (6) floating point register files. The four user threads (A,B,C,D) and the system thread (S) each have 14 integer registers and 15 floating point registers. The exception handling thread (X) has only 5 integer and 4 floating point registers. As shown in figure 4.9, each integer register file has 4 read ports and 3 write ports and each floating point register file has 4 read ports and 2 write ports.

Registers

Note: This section is reproduced from M-Machine Architecture v1.0 .

The register state accessible within a cluster consists of:

- General registers
- Special registers

- Condition registers

General Registers

Each H-Thread sees 14 integer registers and 15 floating point registers in its own cluster. The integer register file has 4 read ports and 3 write ports. Within a single cycle, the integer unit and the memory unit can each read two operands. In addition, both units and the C-Switch may write a result. The floating-point register file has 4 read and 2 write ports The floating point unit may read 3 operands and write a result. The memory unit may read 1 operand and write a result.

The general registers are divided into two portions *public* and *private*. The private registers reside in the lower half of the register file [i7:i0], [f7:f0] , while the public registers reside in the upper half. Private registers may only be written by operations executed within the cluster. Any communication between clusters must use the public registers. The hardware enforces this separation; an attempt to writes to a remote private register results in a write to the corresponding remote public register.

Special Registers

The special registers reside in the same namespace as the general registers. Table 4.1 specifies the special register names and meanings. Explanations are as follows:

`Zero` : Integer register 0 (*i0*) in all clusters is mapped to zero. Reading *i0* gets a zero value, while writing *i0* has no affect on the state of the register file. Register *f0* holds floating point zero, but otherwise has the same behavior as *i0*.

`IP` : The Instruction Pointer register points to the instruction that the cluster is executing. This register may not be written directly, but is modified via *branch*, or *jump*. Writing directly to *i1* has no affect on the state of the register file.

`HEAD` : The dedicated system slot on each cluster has an associated *event* or *message* queue. The first word of the next message is accessed by reading integer register 14. This flushes any remaining words from the current event or message.

`BODY` : This is mapped to integer register 15 and accesses the next word in the message or event queue. If the next word is the first word of the next message or event, *Errval* is returned. User threads may access *i14* and *i15* as a general registers.

`MC` : Each thread has 10 special registers used for message composition. Outgoing messages are composed in these registers and then launched using a *send* instruction. These registers may also be used by the programmer as general registers.

Cluster	Register	Meaning
all	i0	Always zero. Writes to i0 disappear.
all	f0	Always fp zero. Writes to f0 disappear.
all	i1	Instruction Pointer. Writes to i1 disappear
all	i14	Head (System V-thread, Event Cluster only).
all	i15	Body (System V-thread, Event Cluster only).
all	i4–i13	Message composition (MC).

Table 4.1: Special registers.

4.3.2 Scoreboard

The integer unit scoreboard is physically located in the RR stage. The scoreboard maintains the presence bits for each register in the integer register file.

The Scoreboard maintains the value of the presence bits for each register and is updated in the following manner:

- Normal Operation: the WB stage of the Pipeline writes data and validates the presence bits during the first half cycle (Clock High) of the master clock *clk*. The data is latched at clock low and the values that were written can be read in the second half of the same clock cycle.
- Normal Operation: the EX stage of the Pipeline invalidates the presence bits during the first half cycle (Clock High) of the master clock *clk*. The data is latched at clock low and the values that were written can be read in the second half of the same clock cycle.
- Normal Operation (MOV): the WB stage of the remote cluster writes the presence bits via the CSW during the first half cycle of the master clock. Data follows in the second half cycle (Clock LOW) of the master clock *clk*. The data is latched at clock high of the subsequent period.
- Bypass (EX invalidation to OPSZ buffer): the EX writes presence bits during the first half cycle of the master clock *clk*. The OPSZ buffer latches these values with all other validation/invalidation requests.
- Bypass (LD returning over CSW): the CSW writes presence presence bits during the first half cycle of the master clock *clk*. These values are latched during the second half cycle. The data over the CSW is provided during the second half cycle and is bypassed directly to the input of the OPSZ and EX pipeline stages if the appropriate register address matches.

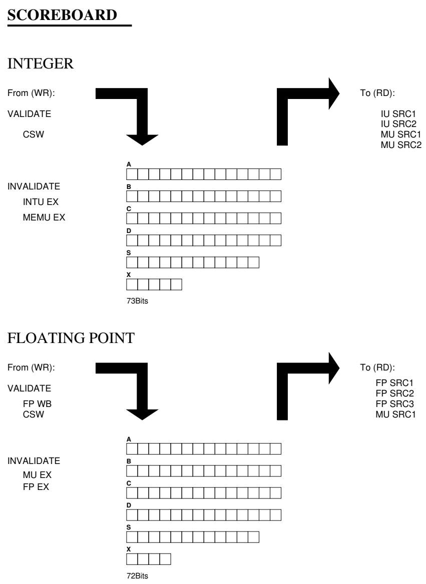


Figure 4.10: *Scoreboard Logical Overview*

4.3.3 Operation Predecode

Integer Unit Operations The integer unit performs 41 different operations. The RR stage decodes the 5-8bit operation field for each integer operation to determine which input sources are needed for each instruction. Also, the RR stage generates a five (5) bit group identifier and a five (5) bit type identifier for use by the execution unit.

4.3.4 SEND

When a SEND operation is issued, the Integer EX stage stalls the pipeline until the SEND completes. This decision was made to reduce design complexity and eliminate the need to add an additional Read Port on each register file. In both the INTU and FPU the SEND FSM in the EX stage generates the SEND control signal. Each WB stage then asserts READ and provides the required register addresses to the corresponding register file to access the Message Composition Registers (MC) using the existing RR/WB wires. The data values are transmitted to the corresponding WB stage by indicating "READ" and "turning around" and reusing the data wires of the RR/WB interface.

Figure 4.12 shows the SEND FSM.

Figure 4.13 shows the SEND timing.

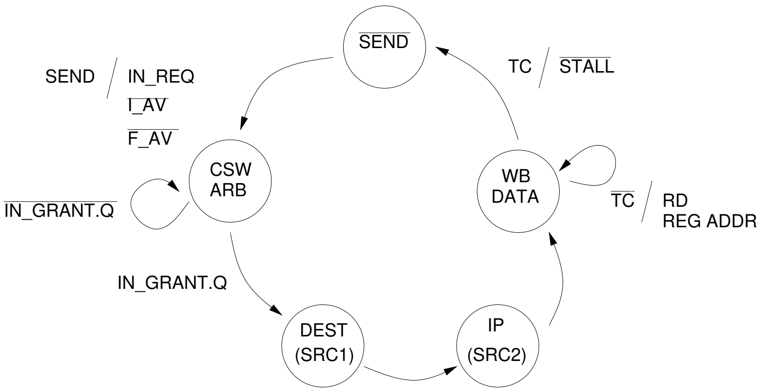
INT

GROUP					TYPE				
SEND	BR	ARITH	BOOL	SHIFT					

SEND	X	X	[Pri]	[Phys]	[~INC]			5
BR	X	X	X	X	[JMP]			2
ARITH	PTR	NEQ	SIGN	EQ	$\overline{\text{SRC2}}$			11
	0	0	1	0	0	ADD		
	0	0	0	0	0	ADDU		
	0	0	1	0	1	SUB		
	0	0	0	0	1	SUBU		
	0	0	1	0	1	ILT	**CC DEST**	
	0	1	1	1	1	ILE	**CC DEST**	
	0	0	0	0	1	ULT	**CC DEST**	
	0	1	0	1	1	ULE	**CC DEST**	
	0	0	1	1	1	IEQ	**CC DEST**	
	0	1	1	0	1	INE	**CC DEST**	
	1	0	0	0	0	LEA/LEAB		
BOOL	MOV	PTR	XOR	NOT	AND/$\overline{\text{OR}}$			12
	0	0	0	0	0	OR		
	0	0	0	0	1	AND		
	0	0	0	1	0	NOT		
	0	0	1	0	0	XOR		
	0	0	0	0	0	CCOR	**CC DEST**	
	0	0	0	0	1	CCAND	**CC DEST**	
	0	0	0	1	1	CCNAND	**CC DEST**	
	1	0	0	0	0	MOV, EMPTY, CCEMPTY		
	0	1	0	0	0	SETPTR		
	0	1	0	0	1	ISPTR	**CC DEST**	
SHIFT	OR	H/$\overline{\text{B}}$	W	ROT	LG/AR			9
	0	0	1	0	0	ASH		
	0	0	1	0	1	LSH		
	0	0	1	1	1	ROT		
	0	0	0	0	1	EXTB		
	0	1	0	0	1	EXTH		
	1	0	0	0	1	INSB		
	1	1	0	0	1	INSH		
	1	0	1	0	1	SHORU		
	1	0	1	0	1	IMM		

Figure 4.11: *Integer Operation Predecode*

LOGICAL



IMPLEMENTATION

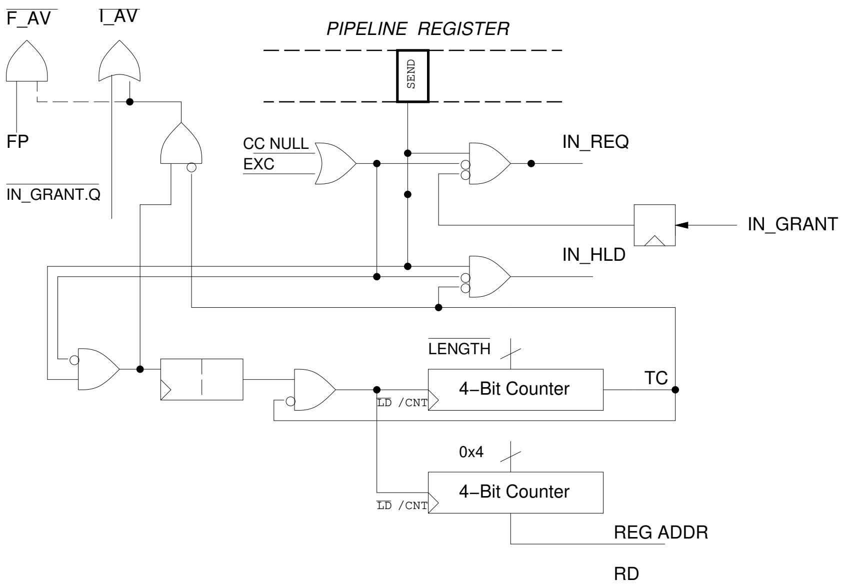


Figure 4.12: *EX SEND State Diagram and Implementation*

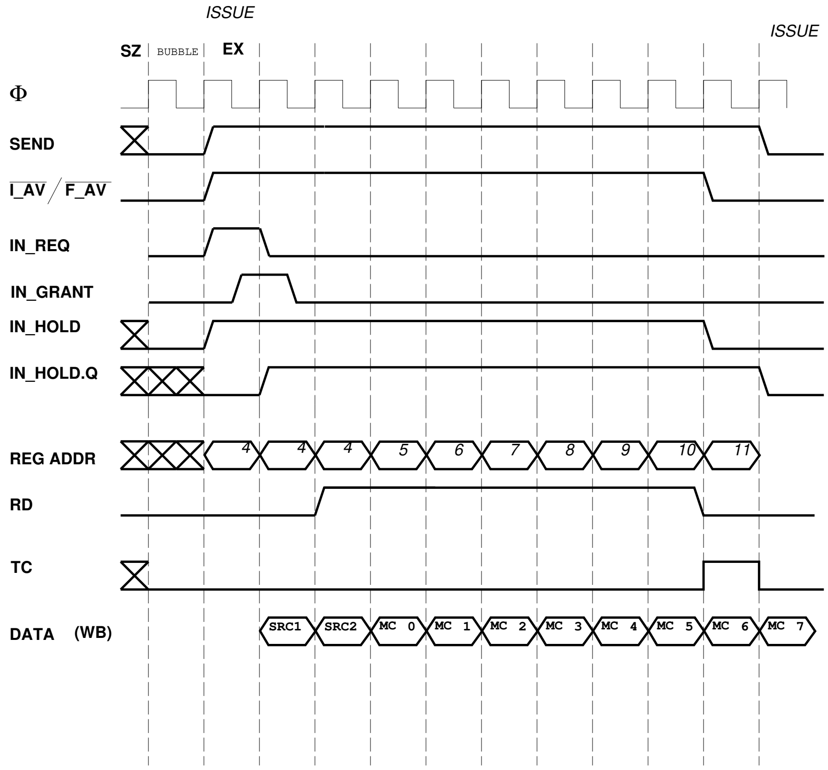


Figure 4.13: *SEND Timing*

4.4 RTL

A RTL module view of the INTU RR Stage is provided in figure 4.14.

The INTU RR stage is composed of the following 15 modules:

- Control Logic
  - IDEC Operation decode
  - ISEL Operand select
  - ISXT Sign extension and immediate generation
  - IDST Destination decode and selection
  - IVAL Validation/invalidation
  - CC Conditon code control
  - IBYP Bypass control and select
  - ICFG Config Space Interface
  - ICSW C-Switch Interface
  - ISND Send control
- Datapath
  - IRF Integer Register File (7-ports, 4 read, 3 write)
  - ISC Integer scoreboard
  - CCRF Condition code register file and valid bits
  - IMUX Operand Multiplexers
  - IOPSZ Integer portion of the OPSZ Pipeline Registers

OPDEC The *opdec* module is a combinational logic block which takes operation bits (op[23:16]) from the output of the IF stage and decodes the instruction type as defined in the instruction encoding appendix. The instruction type information (9 bits) is one hot encoded and is used subsequently by the OPSEL, SEXT, DEST modules. The *opdec* module also translate the operand validity (iv, mv, fv) and late kill (ifkill) signal from the IF stage into valid and present bits for the downstream pipeline stages.

OPSEL The *opsel* module is a combinational logic block which takes the operation bits op[23:0] from the output of the IF stage, the instruction type vector from *opdec* and generates a "one hot" encoded operand select vector for each source operand.

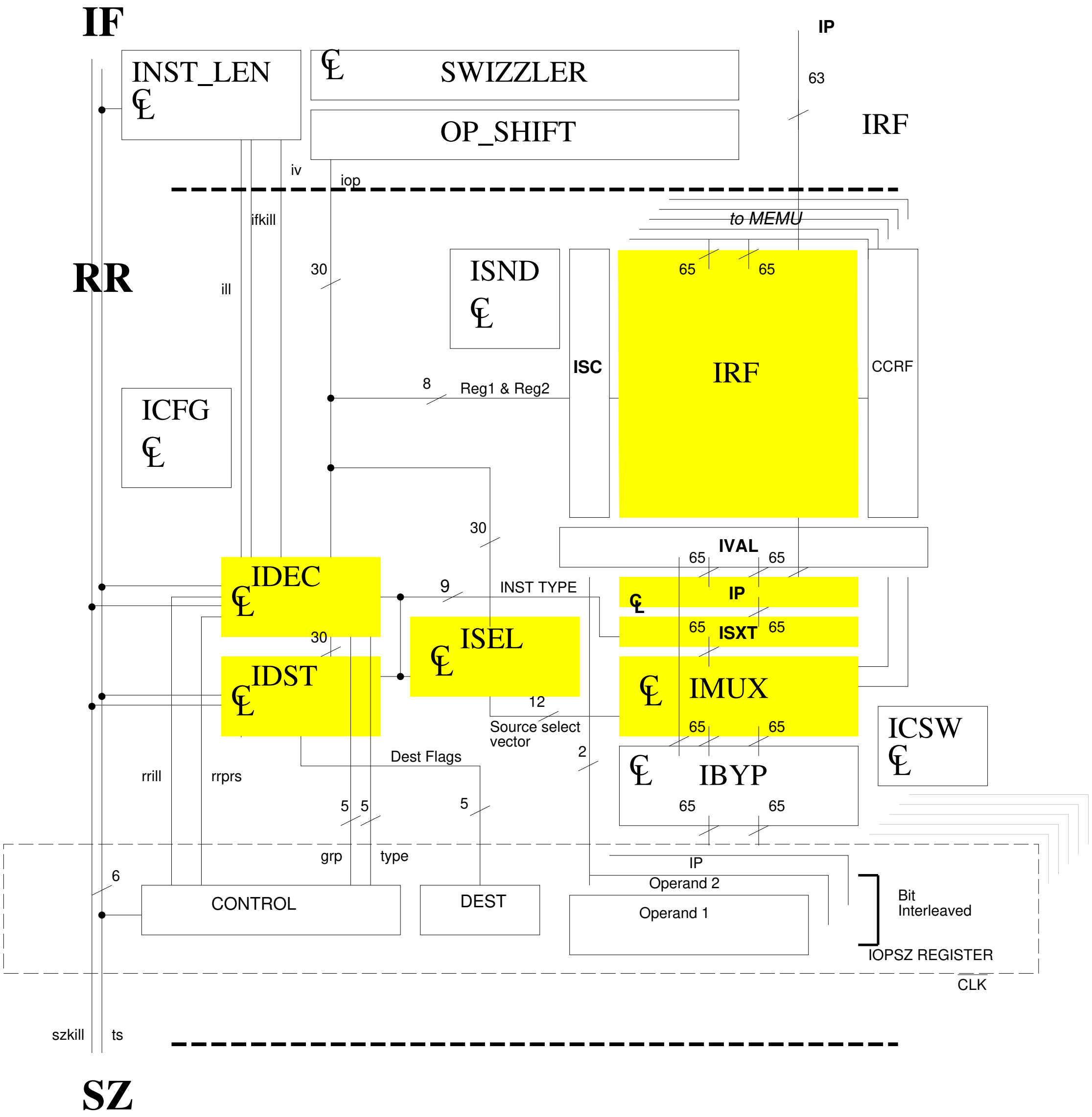


Figure 4.14: *INTU RR Module Diagram*

SEXT The *sext* module is a combinational logic block which takes the operation bits op[23:0] from the output of the IF stage, the instruction type vector from *opdec*, the operand select vector from *opsel* and generates the requested 65-bit constant: unsigned immediate, sign-extended immediate, IP or zero.

DEST The *dest* module is a combinational logic block which takes the operation bits op[23:11] from the output of the IF stage, and the instruction type vector from *opdec* and generates a "one hot" encoded destination vector indicating: branch, register file, scoreboard, cc register file, cc scoreboard. The *dest* module *deasserts* bits in the vector for *NO* destination. No destination is asserted when the operation has no destination (ILL,NOP) or when the destination is suppressed (writes to i0, i1, f0).

OPMUX The *opmux* module is a datapath module containing the multiplexors to select between register file data, *sext* generated immediate and CC values. The *opmux* module is driven by the source selection vector generated by the *opsel* module.

4.5 Timing

Figure 4.15 shows the RR stage critical timing for a normal integer add operation.

Figure 4.16 shows the RR stage critical bypass timing.

4.6 Area Estimate

4.6.1 Register File

The schematic for a single 7-port register cell is shown in figure 4.17. The preliminary area estimate for this cell is 7.6Kλ^2 (473μm^2). This estimate is based on a 60μm^2 cell size and 8λ wire pitch (2μm).

The anticipated area for each 14 register integer file is 6.92Mλ^2. The anticipated total area required for all integer register files is 37.1Mλ^2.

4.6.2 Scoreboard

The anticipated area for each scoreboard is 0.22Mλ^2. The anticipated total area required for all integer scoreboards is 1.2Mλ^2.

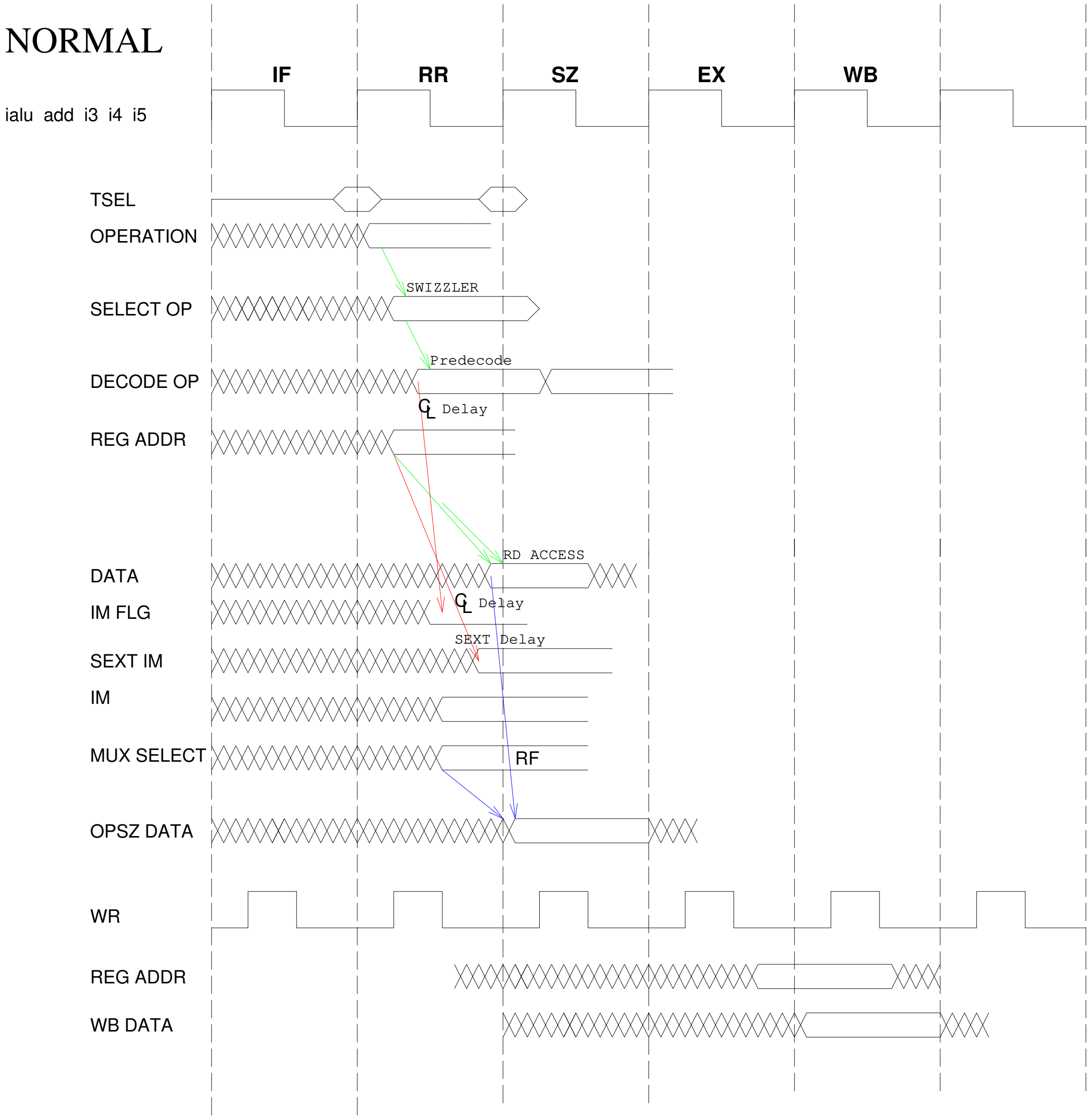


Figure 4.15: *Normal Operation Timing for RR Stage*

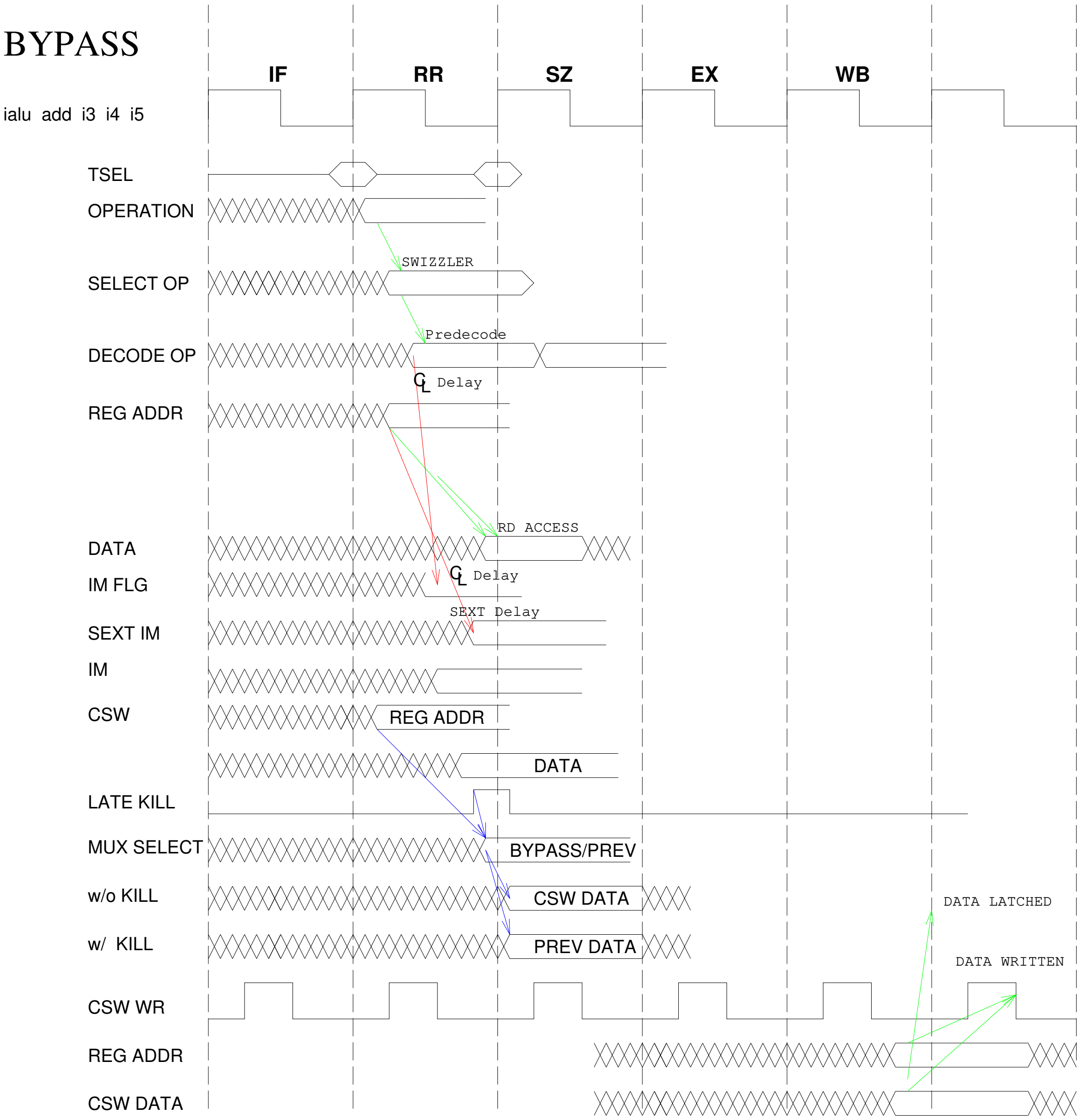


Figure 4.16: *Bypass Operation Timing for the RR Stage*

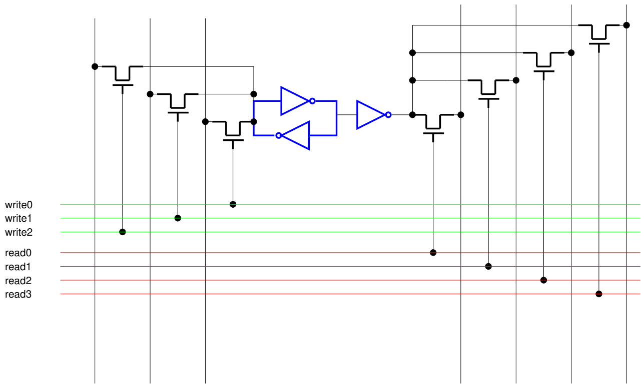


Figure 4.17: *Register Cell Schematic: 7 Ports*

4.6.3 OPSZ Registers

The anticipated area for the 6 registers is 7.296Mλ^2. The anticipated area required for bit line input multiplexors is 0.97MMλ^2. The anticipated area required for bit line output multiplexors is 0.73MMλ^2.

The total anticipated area required for the OPSZ pipeline registers is 9Mλ^2. A preliminary floorplan for the RR stage is presented in figure 4.18

4.6.4 Decode Logic

4.7 Power Consumption

4.8 Open Issues

The following issues remain to be resolved:

4.8.1 Logical Design

1. Physical location of Local and Global CC Registers
2. Signal distribution for Local and Global CC Registers
3. The definition of the RR/CFG interface

RR

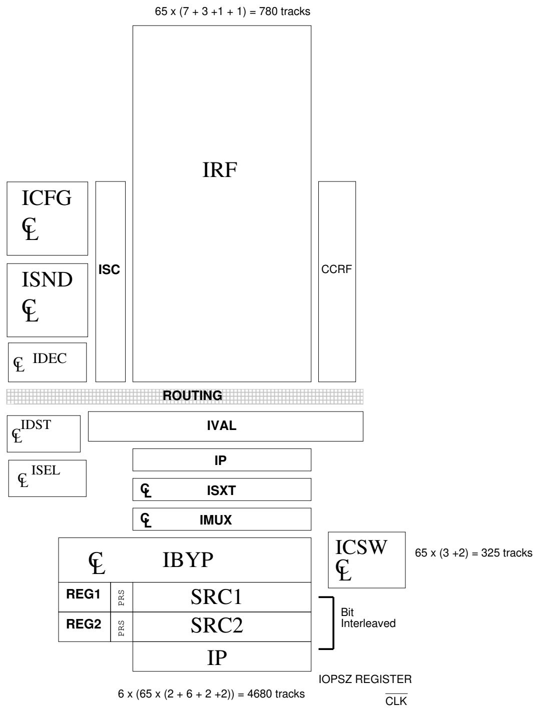


Figure 4.18: *Preliminary OPSZ Floorplan*

4.8.2 RTL Implementation

1. Finish MEMU and FPU sections of opdec.v, opsel.v, sext.v, dest.v and opmux.v
2. Finish remaining modules: byp.v, opsz.v, val.v, send.v, csw.v cfg.v.
3. Integrate and test completed RR stage

4.9 Logic Design Review Action Item List

The following action items resulted from the logic design review:

1. Explicitly work out bypass paths and timing to determine number and sense of required latches. Verify that "it all works".

   STATUS: **Remains to be done**

2. Decide pipeline clocking convention

   STATUS: **Done**, clk = pass on high, $\overline{clk}$ = pass on low.

3. Nuke RD line

   STATUS: **Done**

4. Nuke TSEL for bottom part of pipeline

   STATUS: **TSEL needed to for INTU to generate bs for IF**

5. Nuke MEMU WB validate for INT Scoreboard

   STATUS: **Done**

6. Added FSND instruction

   STATUS: **Done**

7. Added RR logic to coerce ISND, FSND combination into ISND only.

   STATUS: **Done**

8. Add separate IP pointers back into EACH EX stage

   STATUS: **Done**

9. Multiplex access of SRC1 and SRC2 in INTU/FPU SEND controller

   STATUS: **Done**

10. Move bypass mux's to front of EX stage

    STATUS: **Done**

11. Show clocking of REG FILE

    STATUS: **Remains to be done**

12. Show clocking of REG FILE

    STATUS: **Remains to be done**

13. Add path from EX to IF for Branch

    STATUS: **Done**

4.10 RTL Design Review Action Item List

The following action items remain to be completed in the RTL implementation:

1. Use more realistic timing delays (n*‘TGPD, where n = number of levels) instead of just ‘TGPD.

   STATUS:

2. Examine RR bypass logic to see if it can be centrally grouped with SZ Bypass logic - specifically CSW interface.

   STATUS

3. Replace all “?” multiplexors with library MUX version.

   STATUS:

4. Decide on Register File prototype.

   STATUS:

5. Zero out SRC1 base for LEAB in opmux.

   STATUS:

6. Change RR-SZ interface drawing to reflect three IP’s.

   STATUS:

7. Use “OR” instead of “MUX” for final type field select. in opdec.v

   STATUS:

8. Add tDEC and tAC (decoder and access time) constants to map_consts.vh

   STATUS:

9. Separate out the description and count of interface wires from register bits in Section 2 for RR/SZ.

   STATUS:

10. Fix extra comma in IDST inputs list.

    STATUS:

Chapter 5

The Cluster Synchronization Stage

5.1 Overview

The synchronization stage of the pipeline is responsible for selecting valid instructions to issue to the execution stage of pipeline. Figure 5.1 shows a high level view of the synchronization stage. The *Instruction Buffer* structure has a slot for an instruction from each V-Thread. Instructions wait in the instruction buffer until they issue; they may wait for pending data dependencies or for pipeline or other resources to become available. Bypass signals are used to update operand fields of instructions already in the instruction buffer. The instruction buffer provides to the *resource checker* which indicate which instructions have all of their data dependencies satisfied, and what resources they need to use.

The resource checker compares the resources available (e.g. integer pipeline or send buffer), to the resources required by the instructions. It forwards those that are available to the arbiter. In addition, the resource checker has inputs from the cluster control module which holds the thread state. These inputs indicate thread priority, which threads are enabled to run, and which threads threads have been waiting to issue for a long time. The thread state controller contains stall cycle counter and the counter threshold registers used to prevent starvation and deadlock. Threads of lower priority are masked off by the resource checker and are not forwarded to the arbiter.

The arbiter performs a round-robin arbitration among all threads forwarded to it by the resource checker to select one to issue. The *tSel* lines indicate which thread is selected. This value is used locally to write to the execute register and forwarded to the instruction fetch stage, register read stage, and thread controller for use on the next cycle. Two late kill signals may arrive which invalidate the thread select. The *kill* signal is asserted by the C-Switch indicating that promised data will not be delivered. The ifKill signal is asserted by the if the instruction fetch unit cannot deliver a promised instruction from the instruction cache. If either kill signal is asserted matches the thread chosen by the thread selector, then the pipeline is bubbled and tSel is ignored. The signals tSel, kill, and ifKill are distributed throughout the pipeline so that kills may occur locally.

If the instruction contains a send, *thread selector* initiates arbitration for the send buffer. The *sTid* register keeps track of a send operation that is waiting to find out if it has won arbitration among other send operations for access to the send buffer. This is discussed further in Section 5.7.

5.1.1 Criteria for issue

An instruction may only be issued if the following conditions are satisfied:

- The H-Thread is in the RUN state
- If the operations belongs to a user thread, user thread issue is enabled
- All operands are valid
- For SEND operations:
  - Specified message composition registers are valid
  - Outgoing message buffering is available
  - The operation wins arbitration for netout
- For LOAD operations
  - Memory barrier counter < max
- For MBAR operation
  - Memory barrier counter = 0
- Pipeline resources for all of the non-NOP operations are available

5.1.2 Outline

This document describes the following:

- Synchronization stage interface signals
- Instruction buffer design
- Thread state machine
- Resource requirements
- Arbitration policy
- Thread selection

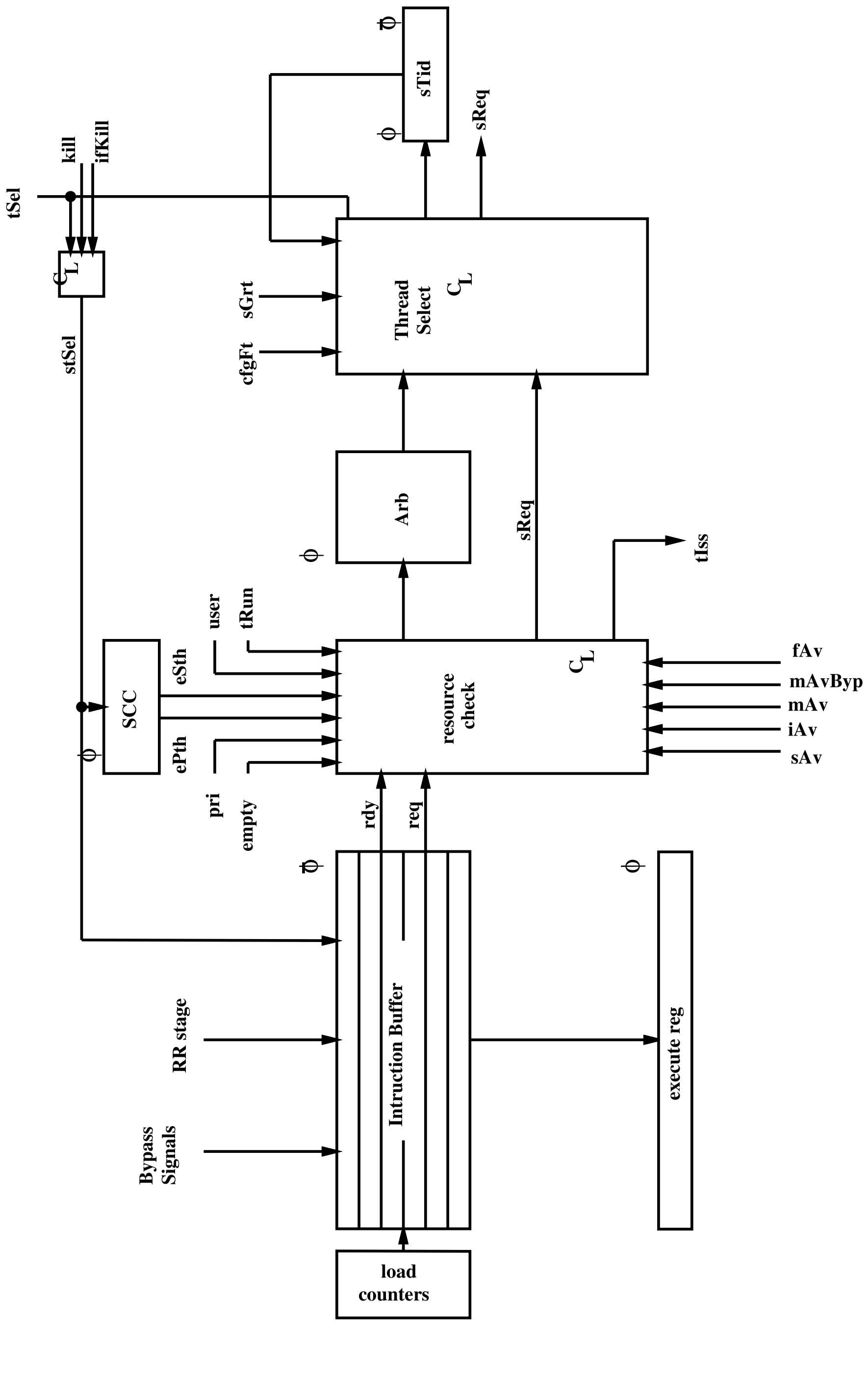


Figure 5.1: SZ stage top level diagram

- Timing

Area and power estimates are in progress.

5.2 Interface Signals

- Cluster Control
  - vAct[0:5] - activate V-Thread
  - vDact[0:5] - disactivate V-Thread
  - hAct[0:5] - activate V-Thread
  - hDact[0:5] - disactivate V-Thread
  - tIss[0:5] - wired-or line, global to all clusters, which is released by a cluster when the stall cycle counter (SCC) exceeds its threshold. When this line is high, the global controller generates a thread timeout event.
  - exc - indicates that an exception is taking place and no user threads may issue. An exception handler unsets this when it has completed.
  - user - indicates that user threads are allowed to issue instructions.
- RR stage: The RR stage decodes the operations and reads the source operand registers before passing the data to the SZ stage. Figure 5.2 shows the data in this interface.[1]
- EX stage: The SZ stage places the instruction selected for issue in the EX register for execution during the subsequent cycle. This data includes all of the information required to execute the instruction, such as operations, operands, destinations, etc. Figure 5.3 show the data in this interface.[2]
- IFETCH
  - empty[0:5] - (input) indicates that an attempt to issue an instruction was made but failed due to insufficient number of operands in ifetch buffer. This indicates that tSel must be asserted later to get the requested instruction into the pipeline.
  - tRun[0:5] - (output) indicates that the corresponding threads are running.
  - tSel[0:5] - (output) indicates a request to issue an instruction down the pipeline from the IFETCH unit.
  - ifKill - (input) one line from the I-Fetch unit which indicates that the next instruction from the thread indicated by tSel on the previous cycle was not delivered. If the next instruction for this thread was selected by the SZ stage, the pipeline is bubbled.

[1]This figure is replicated from Andrew Chang's register read stage design document.
[2]This figure is replicated from Andrew Chang's execute stage design document.

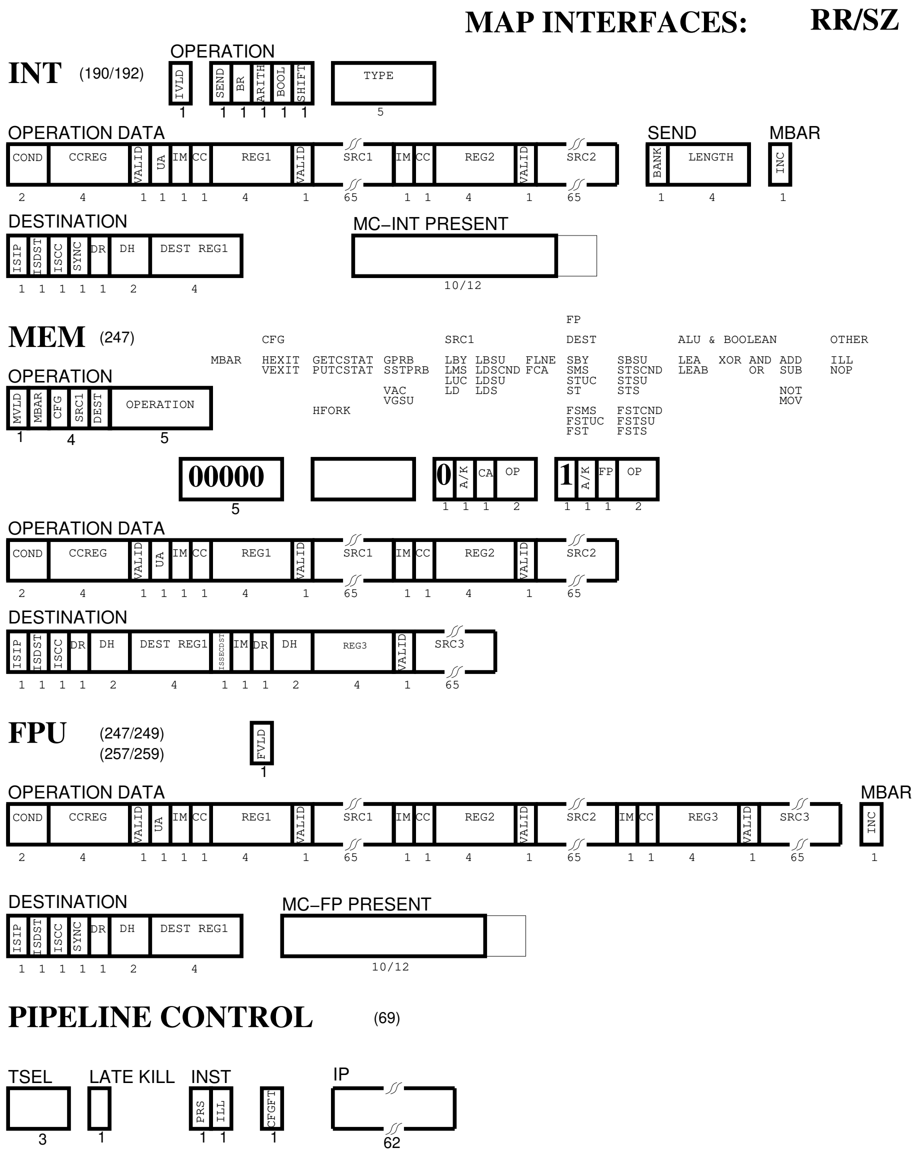


Figure 5.2: The RR-SZ Interface

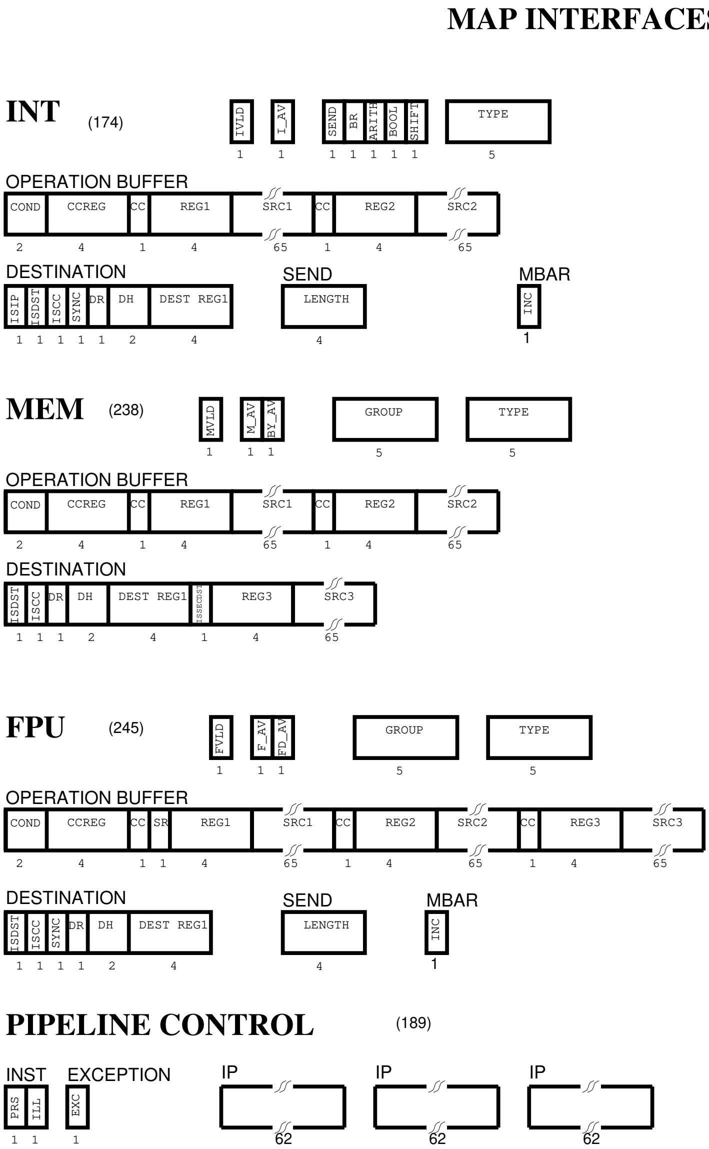


Figure 5.3: The SZ-EX Interface

- Bypassed values There are in excess of 389 bypass wires for the entire synchronization stage including thread and register specifiers.
  - CC Registers (72 wires for bypass for IIB, MIB, FIB):
    * local
      · data: 4 per hthread (need not be bypassed to SZ stage)
      · valid: 4 per thread
    * global
    * data: 8 per thread (need not be bypassed to SZ stage)
    * valid: 8 per thread
  - Register Validation (36 wires for bypass for IIB, MIB, FIB if we allow automatic invalidation between register files in same cluster)
    * Validation from event queues for event threads (1)
    * Invalid from M-EX (3:tid + 4:rid)
    * Valid from CSW (3:tid + 4:rid)
    * Invalid from F-EX (3:tid + 4:rid)
    * Valid from F-EX (3:tid + 4:rid)
    * invalid from CSW late kill (3:tid + 4:rid)
  - Register Data (219 for IIB, 292 for MIB, 146 for FIB):
    * I-EX (3:tid + 4:rid + 66:data)
    * M-EX (3:tid + 4:rid + 66:data)
    * F-EX (3:tid + 4:rid + 66:data)
    * CSW (3:tid + 4:rid + 66:data)

  Thus the totals are:
  - IIB: 327 bypass wires
  - MIB: 400 bypass wires
  - FIB: 254 bypass wires
- Pipeline signals
  - CFGFT - (input) configuration space feed through. This suspends arbitration and allows a configurations space register read request to go directly from the register read stage to the execute stage.
  - I_AV - (input) integer unit is available to accept an operation. This is false when the integer unit has lost C-Switch arbitration.
  - M_AV - (input) memory unit is available to accept an operation. This is false when the memory unit has lost M-Switch arbitration.

- M_AVBYP - (input) memory unit is available to accept a bypass operation. This is false when the bypass path of the TLB miss handling memory unit has lost M-Switch arbitration.
- F_AV - (input) floating point unit is available to accept an operation. This is false if the FP unit is executing a long latency operations (such a FDIV) or if the FP unit has lost C-Switch arbitration.
- KILL[0:5] - (input) kill signal from the C-Switch to squash instruction issue for a selected thread if promised data cannot be delivered. This is one-hot encoded.

- Send resource arbitration
  - S0_AV - (input) priority 0 send buffer can accept a message
  - S0_REQ - (output) request for access priority 0 send buffer
  - S0_GRT - (input) grant line for access to priority 0 send buffer
  - S1_AV - (input) priority 0 send buffer can accept a message
  - S1_REQ - (output) request for access priority 0 send buffer
  - S1_GRT - (input) grant line for access to priority 0 send buffer

5.3 Instruction Buffer

5.3.1 Overview

The instruction buffer is distributed across the 3 function units in the cluster. All of the updates to the state in the instruction buffer takes place there, as does local checking of operation validity. The operation valid signals are combined at a central location to generate the instruction valid signals for the thread state machine. The instruction's IP is also kept at a central location so that it may be passed to the execution stage and used when generating error values (actually, we might need the IP for all operations since invalid operations–nops–do not use pipeline resources). In addition, two wires to signal that send arbitration needs to be performed are provided for the thread selector. All of the fields of the instruction buffer are bit interleaved. Figure 5.4 shows a diagram of the instruction buffer module.

5.3.2 Bypass

Figure 11.5 shows a simple instruction buffer data cell from the IIB, with bypass paths. The register cell has a single read and write port. All possible writers must be multiplexed in. The logic at the bottom of the diagram shows the control signals for the data cell. In addition to data cells, valid and control cells are required. The valid cells hold the status of the data and is used to determine if all of the operands are valid. Thus the valid bits need not be routed to the EX stage, but additional local logic is required (a local AND of

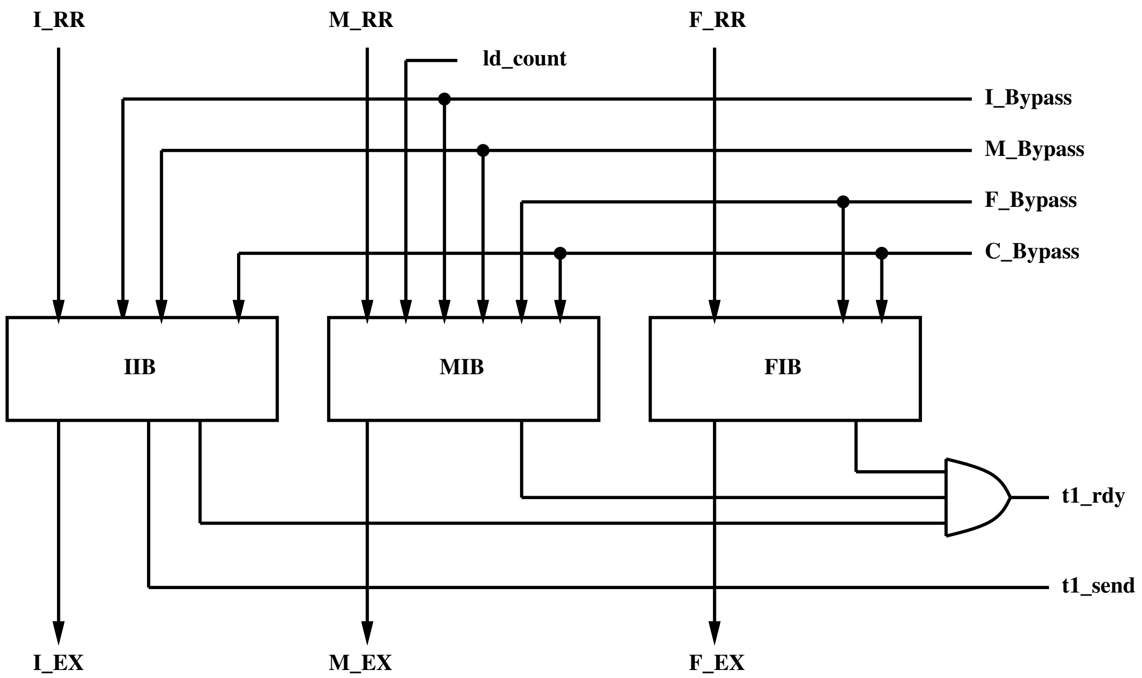


Figure 5.4: Instruction buffer.

all valid signals) to determine if the operation is valid. The control cells hold the operation specification (opcode, operand specifiers, etc.) and are used to drive the data and valid cell control lines.

For the IIB we need the following cells:

- cond: 1 input
- ccreg: 1 input
- valid: 12 inputs (1 for each CC reg)
- ua, im, cc, reg1: 1 input
- valid1: (12:ccreg, 1:eq, 1:m-ex, 2:csw) - 17 inputs
- src1: 4 inputs (times 66)
- im, cc, reg2: 1 input
- valid2: (12:ccreg, 1:eq, 1:m-ex, 2:csw) - 17 inputs
- src2: 4 inputs (times 66)
- bank, length, mbar, dest: 1 input

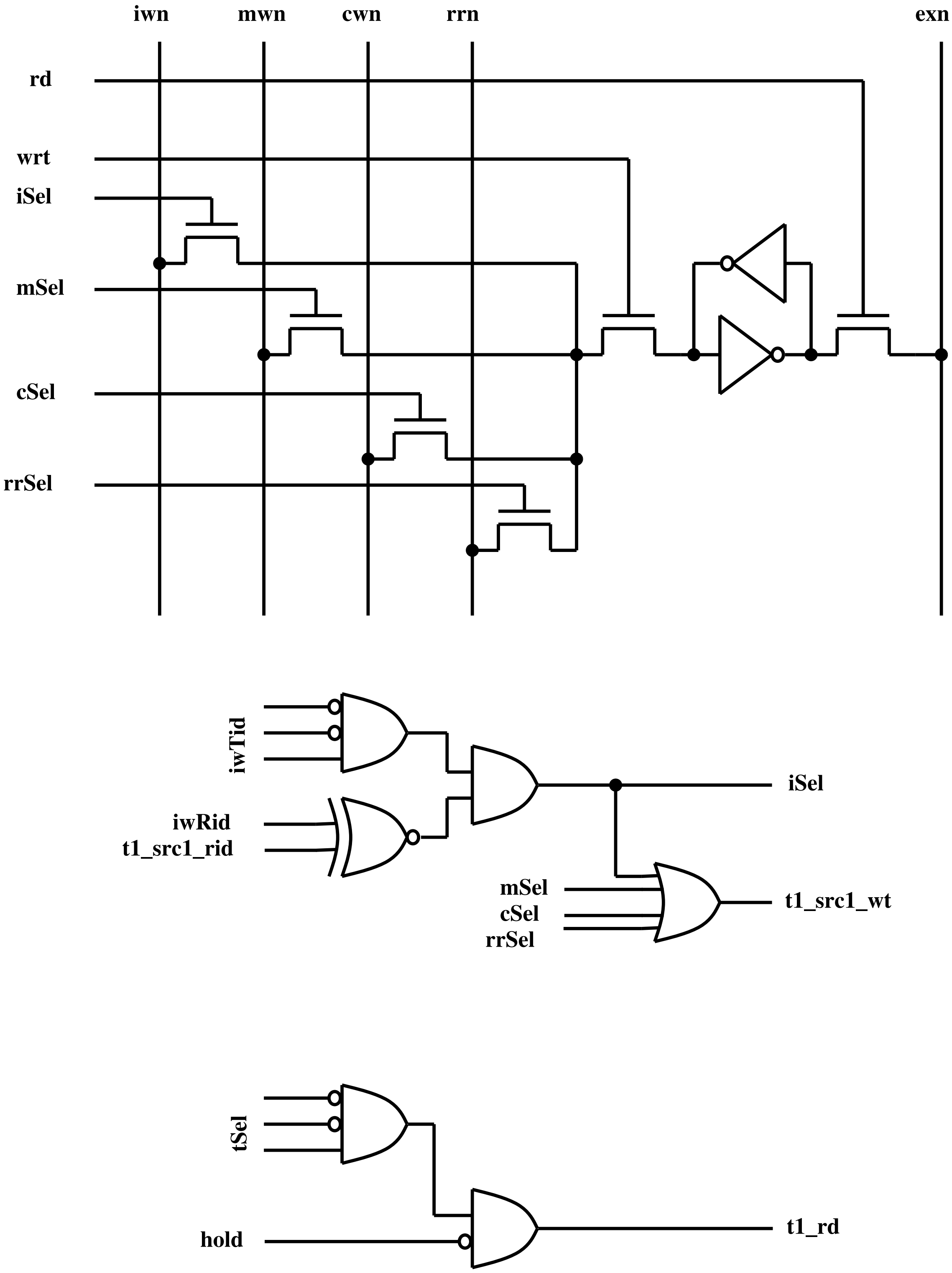


Figure 5.5: Instruction buffer cell and update logic.

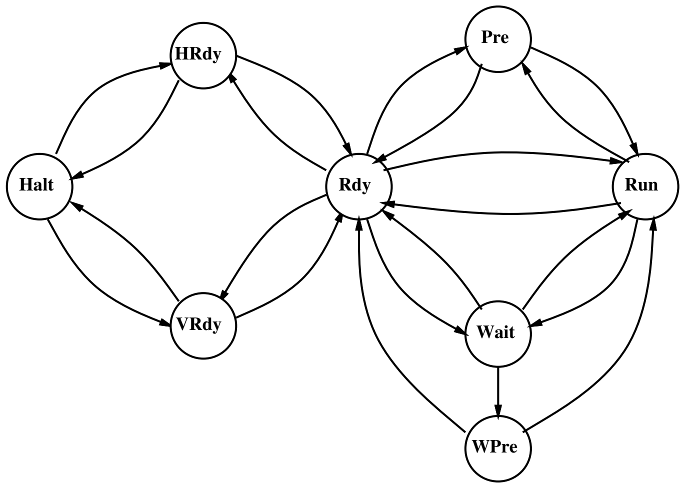


Figure 5.6: H-Thread States

- mc-int present: 5 inputs

The control portion for each data field will have the same number of inputs as the field itself.

5.3.3 Validity Check

The validity check is essentially a giant AND of all operand valid bits. This includes all of the source registers, condition registers, message composition registers for send operations, and a check on the memory barrier counter for load and MBAR operations. For MBAR to issue the memory barrier counter must be zero. For a load to issue the memory barrier counter must be less than the max value. This logic will be placed under the wires in the instruction buffer. Additional wire tracks will be required to concentrate all of the individual valid signals.

5.4 Thread State Machine

A thread may be in one of the following states:

- Halt - H-Thread is not enabled to run. The thread is in this state if the H-Thread is vacant or if the V-Thread is halted
- V-Act - The V-Thread is active but the H-Thread is not allowed to issue instructions.
- H-Act - The H-Thread will be allowed to issue instructions as soon as its V-Thread becomes active.

- Pnd - (pending) Thread may execute operations but not all operands are valid
- Rdy - Thread may execute operations and all operands are valid
- Pre - Thread has higher priority to issue for a single cycle.
- Wait - Thread is waiting for IF unit to deliver another instruction down the pipe.
- WPre - Thread is has a higher priority for a single cycle to eliminate a bubble in the top half of the pipe.

Figure 5.6 shows the transition diagram and Table 5.1 shows the state transition table for these states. A thread moves from *Halt* to *Rdy* when its V-Thread and H-Thread active signals are asserted. The thread only returns to Halt when the V-Thread and H-Thread are disactivated. If the user threads are prohibited from issue (such as during an exception), the thread remains in the running state but the resource checking logic will prevent instruction issue.

A thread moves to the *Wait* state when the IFETCH unit raises the empty bit for the thread, indicating that a prior tSel has been unable to be satisfied. While empty is true, the thread will not issue instructions. When empty becomes false, the thread remains in the wait state but becomes eligible for instruction issue (and has, by definition, all of its data dependencies satisfied). A thread may move from the Wait state to the WPre state when empty is no longer asserted and the counter exceeds the PTH. When in the Wait state, if empty is true, then the counter may trigger a STH event. When empty becomes false, the SCC is reset and later compared with the PTH.

When the thread is selected for issue, no operation is issued to the EX stage, but tSel is asserted. tSel tells the IFETCH and RR units to deliver an operation. Locally, a HOLD signal tells the instruction buffer to keep the current state and throw away whatever the RR stage gives it. The thread then moves back to the PND or RDY state. All state transitions are latched at the end of phi1 (rising clock).

In addition, the thread state machine includes the stall cycle counter (SCC), and two threshold registers: the preempt threshold register (PTH) and the stall threshold register (STH). The SCC is incremented while the thread is in the PND or RDY states and tSel is not asserted. The SCC is cleared whenever tSel is asserted or when the thread transitions from PND to RDY. When the SCC exceeds the STH and the thread is in the PND state, tIss is deasserted, which may signal an event in the global control unit. When the SCC exceeds the PTH and the thread is in the RDY state, the thread transitions to the PRE state, gaining higher priority until it issues.

5.5 Resource Check

The resource checking logic is responsible for filtering out those threads whose resources are not available, thus preventing them from arbitrating for issue. In addition, the priority of

State	Inputs	Next State	Outputs
Halt	*hAct*	HRdy	*tRun*
	vAct	VRdy	
HRdy	*vAct*	Rdy	*tRun*, *pClear*
	hDact	Halt	
VRdy	*hAct*	Rdy	*tRun*, *pClear*
	vDact	Halt	
Rdy	*Ops Valid*, *tSel*	Rdy	*tRun*, *arb*, *sccReset*
	Ops Invalid	Pnd	*tRun*
	Ops Valid, *pth*	Pre	*tRun*, *TPre*
	empty	Wait	*tRun*, *sccReset*
	vDact	HRdy	
	hDact	VRdy	
Pnd	*Ops Valid*	Rdy	*tRun*, *Arb*, *sccReset*
	Ops Invalid	Pnd	*tRun*
	Ops Invalid, *sth*	Pnd	$\overline{tIss}$
	empty	Wait	*tRun*, *sccReset*
	vDact	HRdy	
	hDact	VRdy	
Pre	*tSel*, *Ops Valid*	Rdy	*tRun*, *sccReset*
	tSel, *Ops Invalid*	Pnd	*tRun*, *sccReset*
	vDact	HRdy	
	hDact	VRdy	
Wait	$\overline{empty}$	Wait	*tRun*, *sccReset*
	$\overline{empty}$, *tSel*, *Ops Valid*	Rdy	*tRun*
	$\overline{empty}$, *tSel*, *Ops Invalid*	Pnd	*tRun*
	$\overline{empty}$, *pth*	WPre	*tRun*
	empty, *sth*	Wait	$\overline{tIss}$
	vDact	HRdy	
	hDact	VRdy	
WPre	*tSel*, *Ops Valid*	Rdy	*tRun*
	tSel, *Ops Invalid*	Pnd	*tRun*
	vDact	HRdy	
	hDact	VRdy	

Table 5.1: Thread State Table

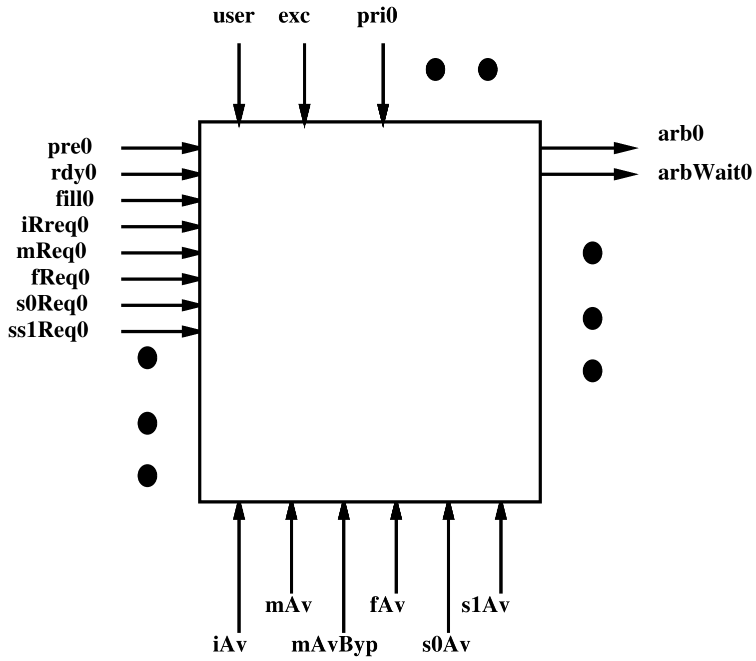


Figure 5.7: Inputs and Outputs for the resource control module.

threads is checked, and only threads of the highest equal priority are allowed to arbitrate. Figure 5.7 shows the inputs and output to the resource control module.

5.5.1 Thread Prioritization

The synchronization stage selects based on a software controlled priority bit. One bit is provided for each user thread; a single bit controls both user and exception threads. The priority bit may be set through configuration space.

The partial order on priorities is shown in Figure 5.8. If all user threads are are at the same priority then there is no scheduling bias among them. However, user threads at *U1* will be granted issue slots over threads at *U0*, given that resources are available for both.

The event thread may set its priority to *S1*, in which it has the highest priority, always issuing if it's resources are available. When the event thread is at *S0*, it is equivalent in priority to the highest priority user thread *U0* or *U1*. The exception thread is always equivalent in priority to a user thread (say *U1*). Thus the event thread may choose to be at the same or higher priority.

Under certain situations, a thread may be ready to issue (all of its operands are present), but is starved due to other threads gaining issue slots and consuming pipeline resources. Livelock can occur with three threads: two threads repeatedly consume resources, while the third loses out every cycle. In order to guarantee no starvation, a preempt threshold (PTH)

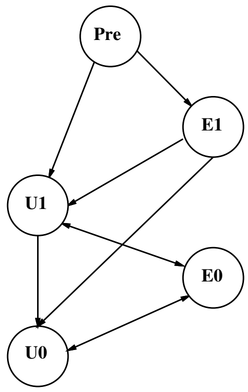


Figure 5.8: Thread Priority

register is used for each thread. When a thread becomes ready to issue, the stall cycle counter (SCC) is set to zero, and is subsequently incremented every time the thread is denied issue. If the SCC exceeds the PTH, then the thread enters the *preempt* state. When any thread is in the preempt state, threads in the run state are prohibited from issue, thus guaranteeing that the preempt thread will issue an instruction. As soon as the preempt thread issues an instruction, it returns to the run state.

5.5.2 Resource Checking Protocol

The gross functionality of the resource control module is described below:

- Always, all threads of equal highest priority with RDY and FILL are released to ArbWait.
- If no threads have PRE, then those threads of equal highest priority with RDY (U0 and E0, U1 and E0, or E1) and resources available are released to ArbAll.
- If any thread has RDY and PRE, then priority is ignored and only those threads with PRE and resources available and released to ArbAll. If no PRE threads have resources available, then no thread is released to ArbAll.

5.6 Arbitration

Arbitration in each of the arbiters (ArbAll and ArbWait) is performed using a simple round robin scheme, with a rotating token and possibly using lookahead. The exact design of the arbiters will influenced heavily by the C-Switch and M-Switch designs. If necessary the ArbWait arbiter could use a fixed priority scheme.

The second arbiter (ArbWait) is used to fill bubbles in the pipeline in the event a late kill signal arrives and aborts the thread that won arbitration for the execution stage. Instead of bubbling the entire pipeline, this opportunity can be used to fill a bubble in the top part of the pipeline incurred at a previous time.

5.7 Thread Selection

The thread selection module must be extremely simple. It makes the final decision as to whether tSel is to be asserted and for what thread. If a late kill comes in, the selector can switch in the ArbWait winner instead of the ArbAll winner. In addition, it arbitrates for the send resources and hold the send request for issue on the next cycle.

5.7.1 Send Resources

In order for a send operations to arbitrate for issue, space in the right priority send buffer must be available (sAv). If a send instruction wins arbitration in ArbAll, it must then arbitrate for the connection to the send buffer with send instructions from other clusters. The send instruction's identifier is placed in the sTid for use on the next cycle. In addition, sReq is asserted to arbitrate for send resources during the next cycle. At this point, the ArbWait winner, if any, may assert tSel.

On the next cycle, the entire arbitration procedure is repeated and the tread selection logic must choose between the thread in sTid, and the ArbAll and ArbWait winners. If sGrt is asserted, the send operation is issued. Otherwise, the send is returned to the arbitration pool (sTid) is nullified, and the winner from ArbAll is issued. If that operation is late killed, then the ArbWait winner asserts tSel.

5.7.2 Late Kill

Two late kill signals may affect the decision of the thread selector:

kill: The kill signal comes from the C-Switch and indicates that the data promised on this cycle will not be delivered. The thread selector compares this signal with the thread id of the ArbAll winner. If it is the same, then the ArbWait winner asserts tSel. The aborted operation must arbitrate again on a later cycle. The late may mark an operand invalid in the instruction buffer, rendering a previously ready operation pending.

ifKill: The ifKill signal arrives from the IF stage and indicates that the instruction that issued on the previous cycle has caused an instruction cache fault. Thus it must not issue an operation on this cycle. If ifKill is asserted, then if lastTsel matches the ArbAll winner then the ArbWait winner asserts tSel. Otherwise, the ArbAll winner issues.

5.8 Timing

Figure 5.9 is a timing diagram of the synchronization stage, showing the critical timing paths.

5.9 Area Estimate

5.10 Power Consumption

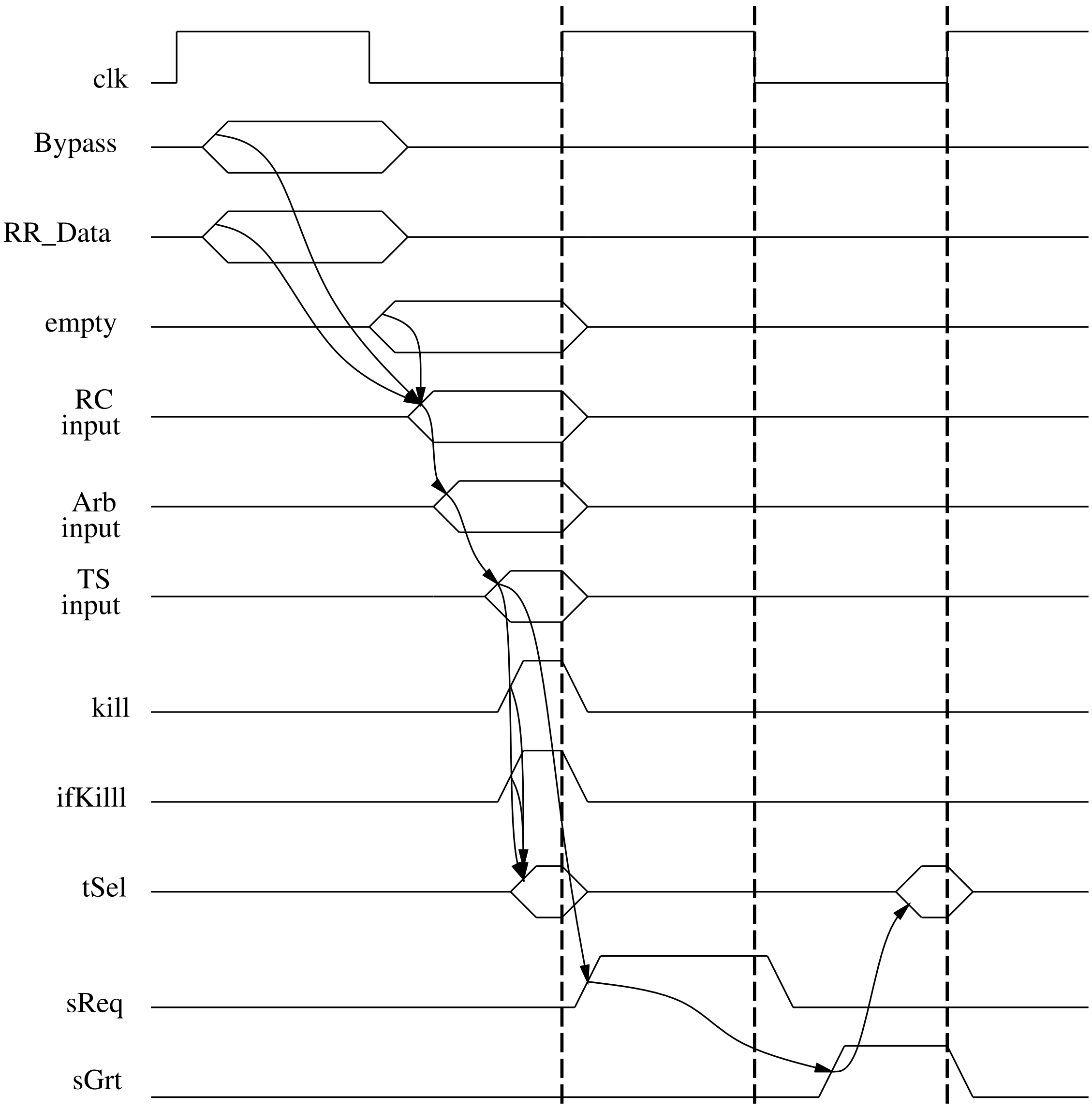


Figure 5.9: SZ stage timing

Chapter 6

The Cluster Integer Execution Unit

6.1 Functional Overview

The Integer EX stage of the MAP pipeline performs all integer operations except multiplication and division. Figure 6.1 shows an overview of the MAP chip pipeline.

The each cluster in the MAP chip possess three seperate EX pipelines: Integer, Memory, and Floating Point. All of the execution units are responsible for performing the following operations:

- Control multiplexing between normal and Bypass sources.
- Detect Conditional Nullification.
- Initiate C-Switch Arbitration.
- Generate register invalidation to SZ and RR stages.
- Generate register validation to SZ and RR states for Bypass results (ie. not in Register File).
- Bypass data to OPSYNC register, Instruction Buffers, and EX register.
- Provide new Instruction Pointer to IF stage during branches or jumps
- Detect Exceptions and initiates handling
- Create Error Values.
- Signal "Not Available" when arbitration is lost for either the CSW or MSW.

The MAP performs *in-order issue* and allows *out-of-order completion*. Out-of-order completion improves performance in the presence of long latency operations. No operation is

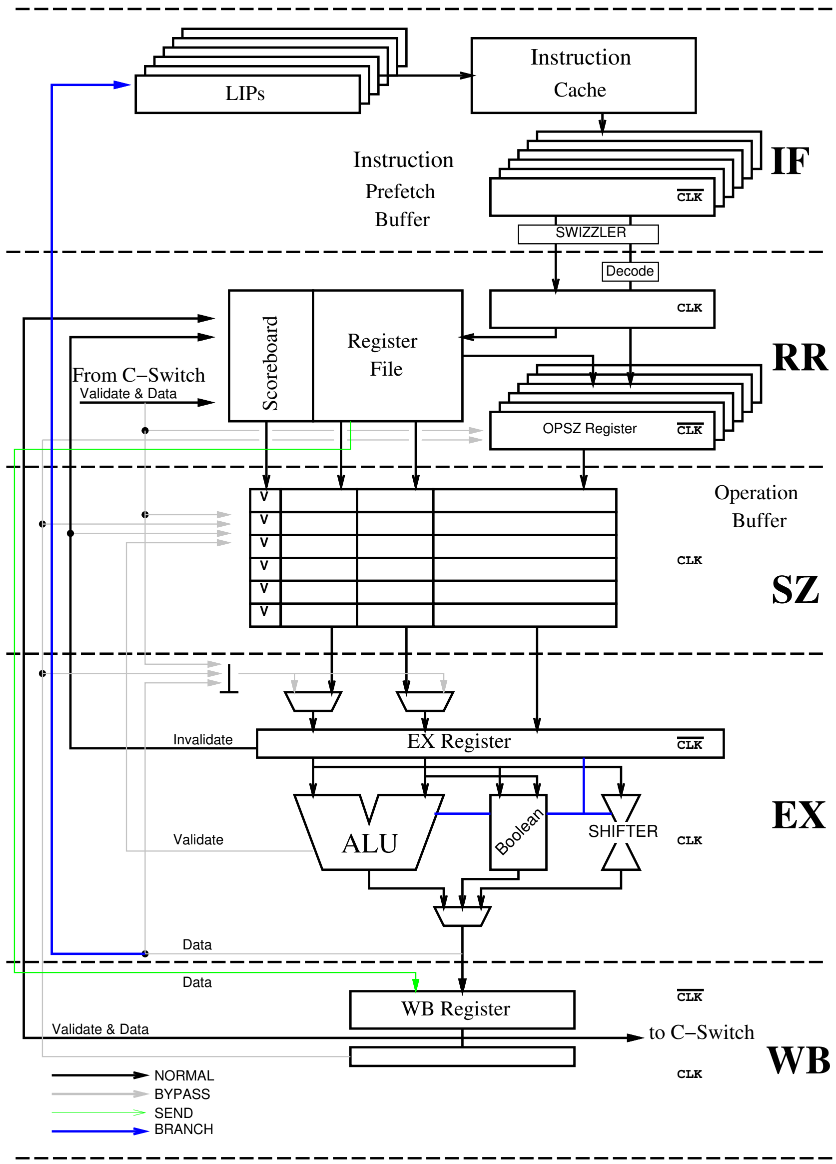


Figure 6.1: *The MAP Pipeline*

issued if it has an unresolved data dependency. Several bypass pathes are provided to allow back-to-back operations to use the destination of the first operation as a source for the second.

6.1.1 Preliminaries

All data is *internally* represented in the MAP chip in *two's complement* format.

The byte ordering in the MAP chip defaults to *Little-endian* format. Alternately, an additional bit can be added to the MAP global state to control byte-ordering on Loads and Stores.

6.1.2 Integer Unit

Figure 6.2 shows the high level logical block diagram of the Integer EX stage.

Functions

The integer unit (IU) of the MAP performs 43 operations. All operations besides the five ISENDs, complete in one (1) cycle. Please refer to *The MAP Instruction Set Reference Manual v1.1* for details on the specific integer operations supported.

In general, the MAP integer unit:

- Execute all integer operations except for IMUL and IDIV
- Execute ISEND

During operation execution the following tasks are performed *in parallel* by the EX stage:

- Test for conditional nullification
- Begin execution of requested operation
- Arbitrate for CSW if needed
- Perform pointer bounds check for LEA, LEAB, BR, and JMP
- Perform permission check for JMP
- Check for Operands for Errval's

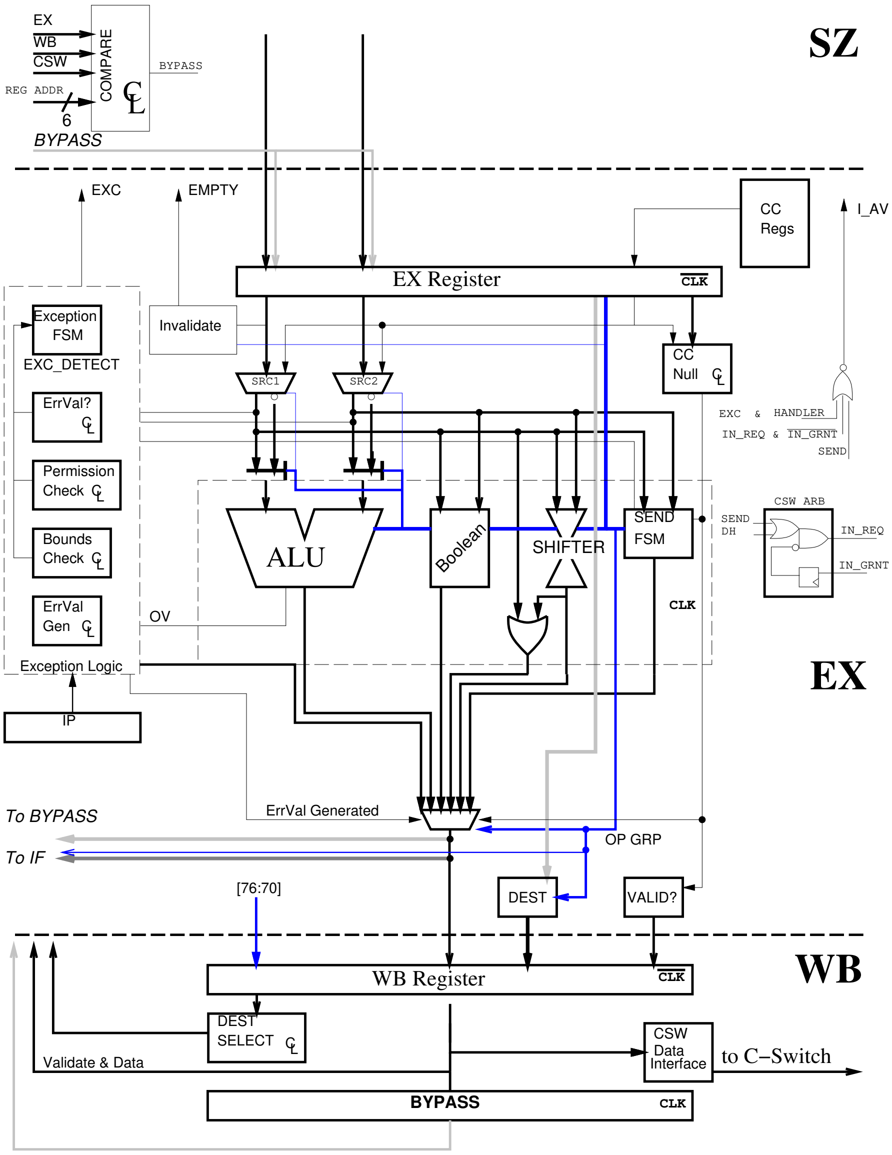


Figure 6.2: *IU Logical Block Diagram*

6.2 Interfaces

The Integer EX stage has five interfaces which are itemized below and detailed in the subsequent figures.

1. Instruction Fetch Stage (IF/EX) - IP for Branches and Jumps
   63-bits
2. Synchronization Stage (SZ/EX) - Operation issue
   236-bits
3. Write Back Stage EX/WB - Data value
   75-bits
4. Cluster Switch (EX/CSW) - Request CSW
   TBD bits
5. Configuration Space FSM (CFG/EX) -
   TBD bits

6.2.1 IF/EX Interface

Figure 6.3 presents the bit fields for the IF/EX interface. During taken branches and jumps the EX stage signals the IF stage that an IP is present and drives the 62 bits of the new IP directly to the IF stage.

6.2.2 SZ/EX Interface

Figure 6.4 presents the bit fields for the SZ/EX interface. The EX stage receives seven types of input from the SZ stage:

1. Condition Code and designated CC register
2. Instruction Pre-decode - Group and Type
3. Source Operand Values
4. Destination Register Name
5. Send Length
6. Current IP
7. Pipeline Control including Present, Valid and Illegal Operation Flags

The EX stage generates I_AV and EXC for the SZ stage.

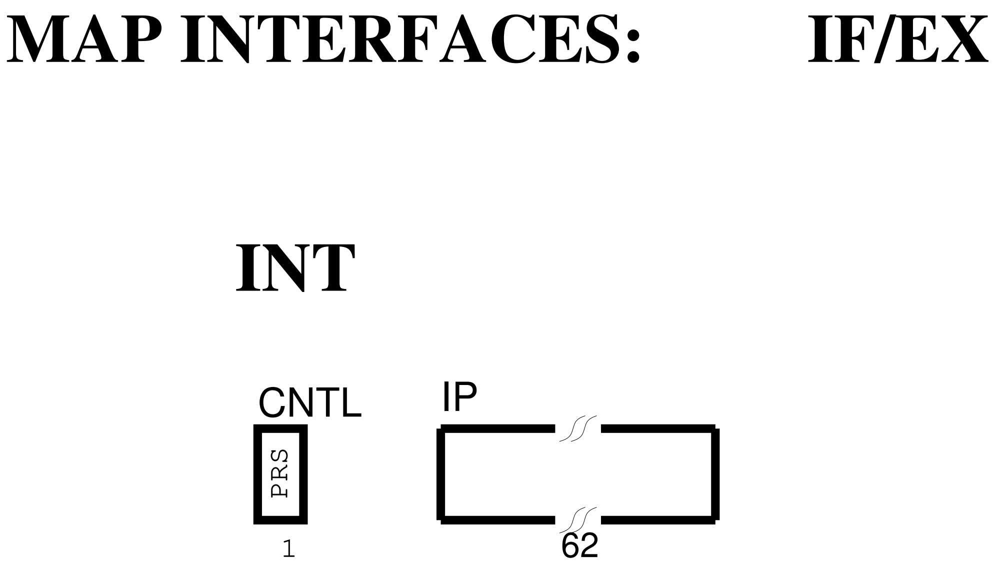


Figure 6.3: *IF/EX Interface*

6.2.3 EX/WB Interface

Figure 6.5 presents the bit fields for the EX/WB interface. The EX stage provides the destination type, the register name and the result data to the WB stage.

6.2.4 EX/CSW Interface

The bit fields for the EX/CSW interface remain *to be defined.*

6.2.5 CFG/EX Interface

The bit fields for the CFG/EX interface remain *to be defined.*

6.3 Operation

This section describes the operation of the MAP pipeline and explores the following functions in detail: pipeline control and sequencing, synchronization, operation execution, exception detection, operation pre-decode, pipeline hazards, exception handling, and error value generation.

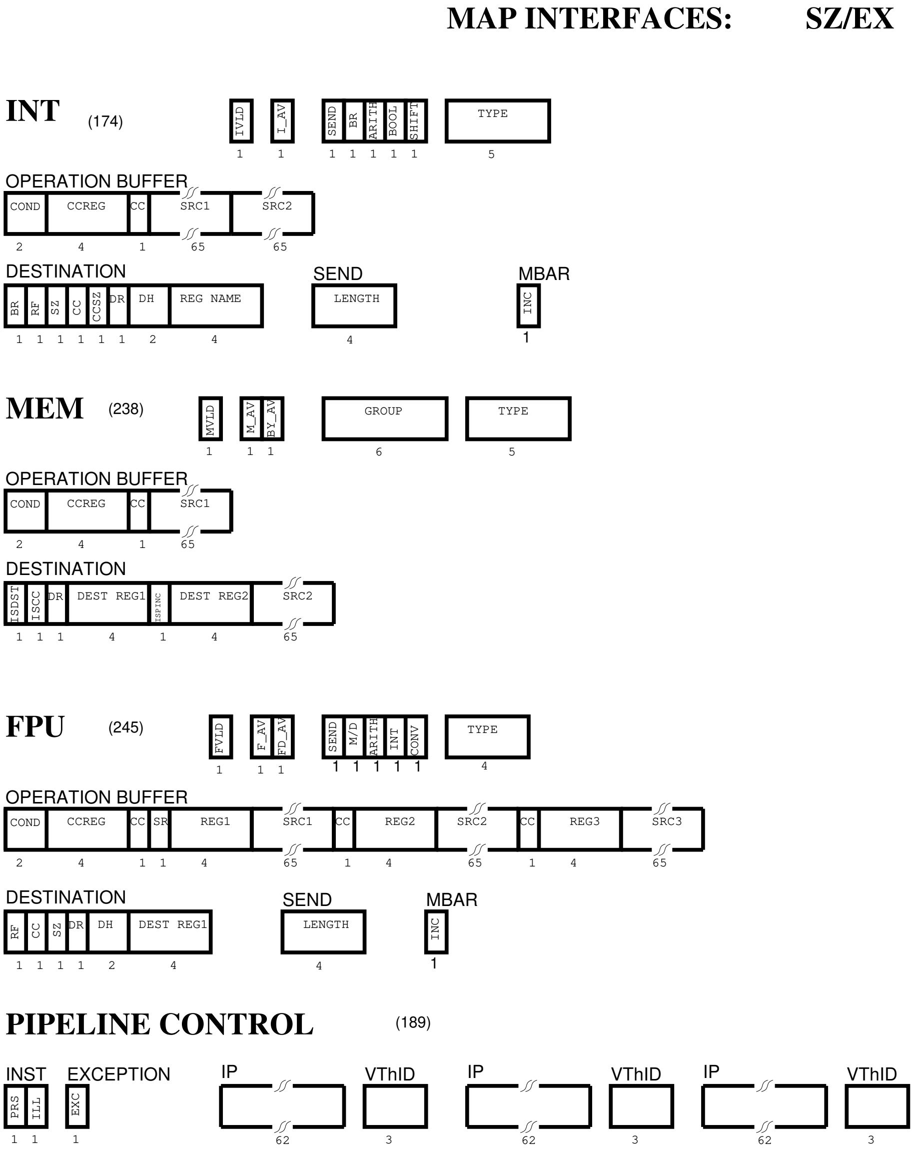


Figure 6.4: *SZ/EX Interface*

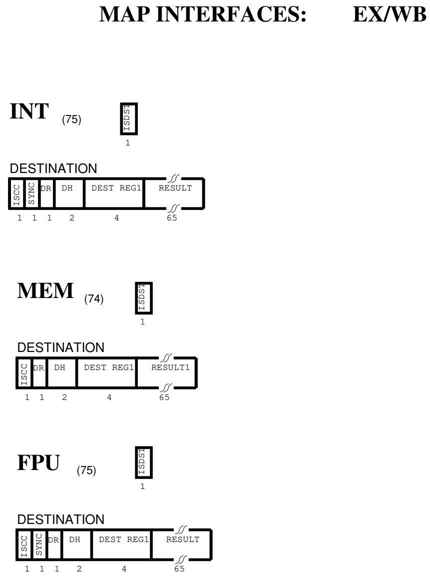


Figure 6.5: *EX/WB Interface*

6.3.1 Pipeline Control

Stalls: The MAP Pipeline stalls under the following conditions:

- Loss of CSW Arbitration stalls *both* the Integer and Floating Point execution pipelines
- Loss of MSW Arbitration stalls the memory unit execution pipeline.
- ICache Miss *ONLY* stalls instruction issue for the relevant thread.
- Exception handling stalls EX and WB stages for *USER* threads only.

6.3.2 Synchronization

All register invalidation/validation must complete in the *first half* of the cycle. Since all integer operations (except ISENDs) complete in one cycle, *no* explicit invalidation is performed.

6.3.3 Operation Execution

Conditional Nullification All operations are conditionally executed. The four conditions are:

- *UA* Execute Unconditionally Always

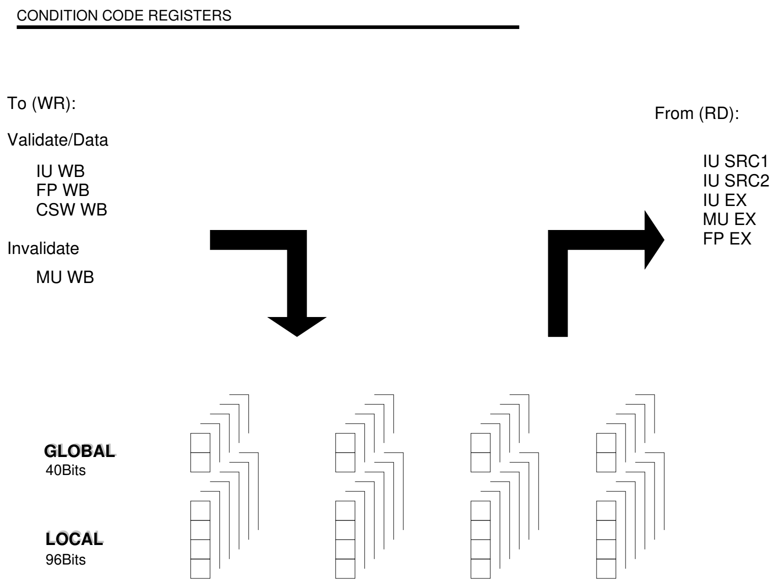


Figure 6.6: *Condition Code Registers*

- *UN* Execute Unconditionally Never
- *CF* Execute conditionally if False
- *CT* Execute conditionally if True

In parallel with the start of execution, the EX stage tests the condition specified with the instruction against the value of the indicated condition code register and potentially nullifies the result of the operation. Note, in all cases except ISEND, the operation is allowed complete in the execution unit. The results are prevented from changing state up setting the "valid" bit on the input to the WB stage to False. Therefore, an operation issue slot is wasted for a conditionally nullifed instruction but no state is modified.

In the case of ISEND, the CSW request is aborted and I_AV is re-asserted after one cycle.

The values of the condition code registers are bussed across the EX unit and are, therefore, readily available for testing.

Figure 6.7 presents the logic to perform CC nullification checking.

6.3.4 ISENDs

The integer unit is responsible for executing an ISEND. Figure 6.8 shows both the logical state transition diagram and a physical implementation requiring only the SEND bit from the pipeline register, a 2-bit shift register, and two 4-bit counters.

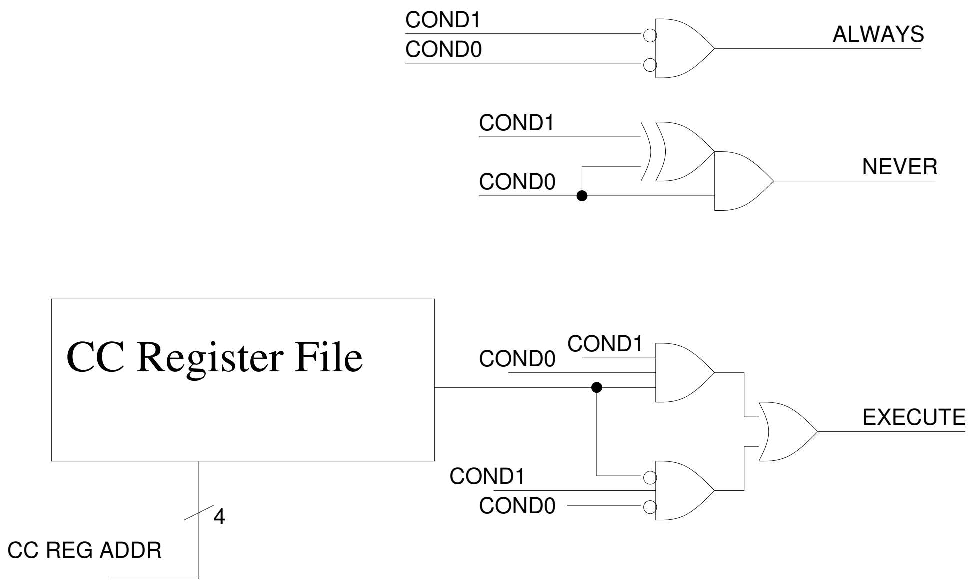


Figure 6.7: *Condition Nullification Check Logic*

The 2-bit shift register is used to sequence the transmission of the SRC1 and SRC2 operands before accessing the integer register file for Message composition register file values.

6.3.5 Operation Predecode

The RR stage "predecodes" the operation code into group and type to expedite execution in the EX stage.

Figure 6.10 presents the group and type fields "predecoded" by the RR stage for integer operations.

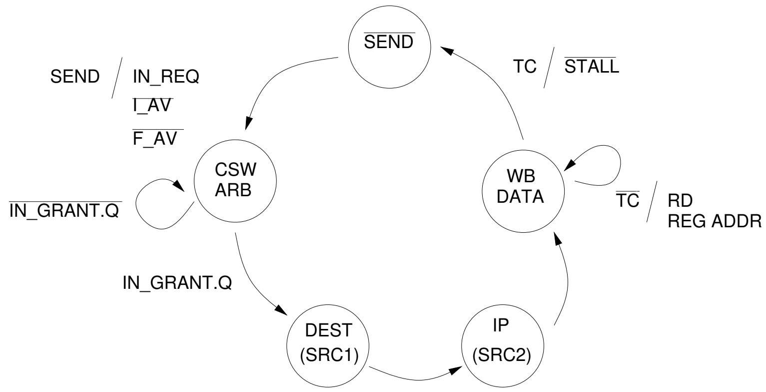
6.3.6 CFG Pseudo Operations

The CFG space interface and the implementation of CFG space pseudo operations remains *to be defined.*

6.3.7 Bypassing Operands

All operations in the integer unit, except for ISEND, are single cycle operations and thus, do not invalidate/validate their destination registers. However, several data bypass pathes must be provided:

LOGICAL



IMPLEMENTATION

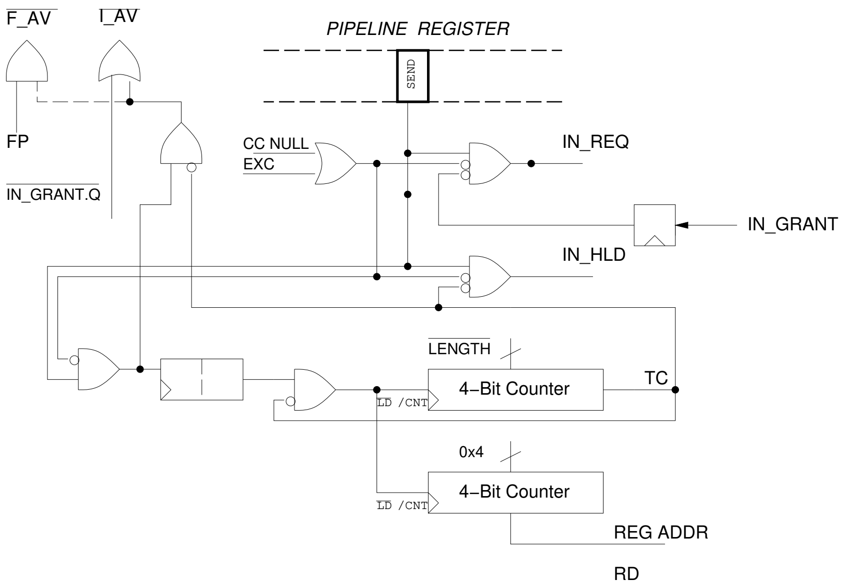


Figure 6.8: *Send State Diagram*

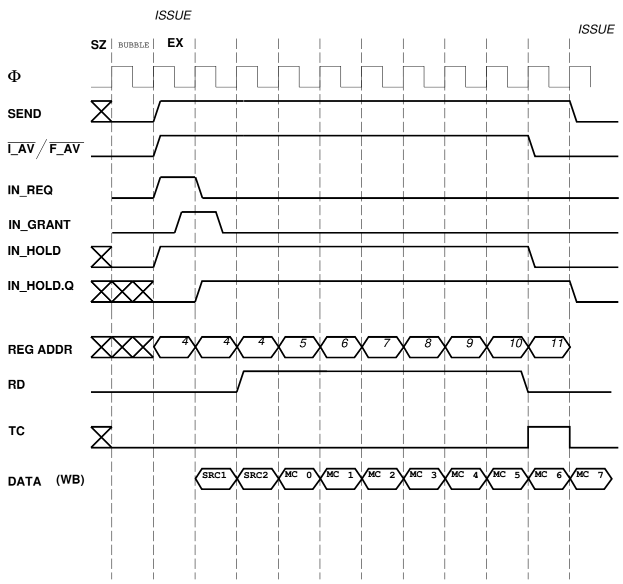


Figure 6.9: *8 Word Send Timing*

1. Output of the EX stage to input of EX stage (latched $\overline{clk}$)
2. Output of WB stage to input of EX stage (latched $\overline{clk}$)
3. Output of WB stage to Instruction Buffers (latched clk)
4. Output of WB stage to OPSYNC Buffers (latched $\overline{clk}$)

Figure 6.11 shows an example using the described bypass pathes.

Data Hazards

All multi-cycle latency operations mark their destination registers invalid upon successful issue (ie no conditional nullification or exception).

The destination registers of all single cycle operations are always valid.

RAW The most common data dependency is "read after write" in which the result of the previous instruction is used as a source operand in the current instruction

Example instruction sequences:

Single cycle operation

INT

GROUP						TYPE				
SEND	BR	ARITH	BOOL	SHIFT						

SEND	X	X	[Pri]	[Phys]	[~INC]			5
BR	X	X	X	X	[JMP]			2
ARITH	PTR	NEQ	SIGN	EQ	$\overline{\text{SRC2}}$			11
	0	0	1	0	0	ADD		
	0	0	0	0	0	ADDU		
	0	0	1	0	1	SUB		
	0	0	0	0	1	SUBU		
	0	0	1	0	1	ILT		
	0	1	1	1	1	ILE		
	0	0	0	0	1	ULT	**CC DEST**	
	0	1	0	1	1	ULE		
	0	0	1	1	1	IEQ		
	0	1	1	0	1	INE		
	1	0	0	0	0	LEA/LEAB		
BOOL	MOV	PTR	XOR	NOT	AND/$\overline{\text{OR}}$			12
	0	0	0	0	0	OR		
	0	0	0	0	1	AND		
	0	0	0	1	0	NOT		
	0	0	1	0	0	XOR		
	0	0	0	0	0	CCOR	**CC DEST**	
	0	0	0	0	1	CCAND		
	0	0	0	1	1	CCNAND		
	1	0	0	0	0	MOV, EMPTY, CCEMPTY		
	0	1	0	0	0	SETPTR		
	0	1	0	0	1	ISPTR	**CC DEST**	
SHIFT	OR	H/$\overline{\text{B}}$	W	ROT	LG/AR			9
	0	0	1	0	0	ASH		
	0	0	1	0	1	LSH		
	0	0	1	1	1	ROT		
	0	0	0	0	1	EXTB		
	0	1	0	0	1	EXTH		
	1	0	0	0	1	INSB		
	1	1	0	0	1	INSH		
	1	0	1	0	1	SHORU		
	1	0	1	0	1	IMM		

Figure 6.10: *Integer Unit Operation Predecode*

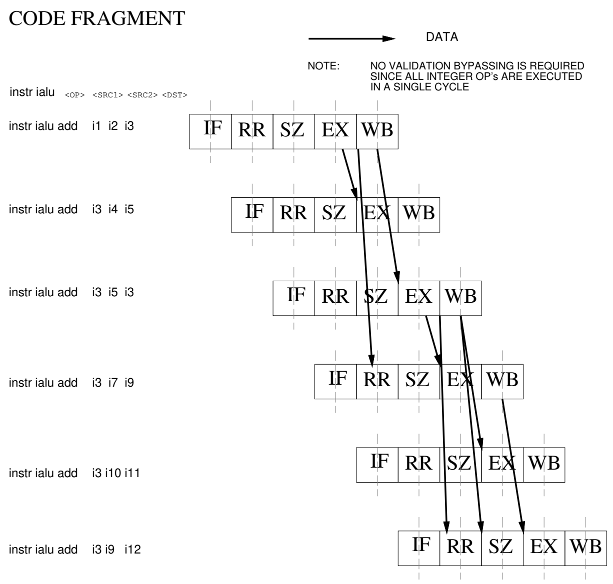


Figure 6.11: *Bypassing Example*

```
ADD   i5 i6 i7
SUB   i7 i8 i7
```

The data result of ADD is *bypassed* to the beginning of the EX stage to "meet up" with the SUB operation. Since ADD is a single cycle operation its destination register is always valid.

Multi-cycle operation - in Memory Unit or Floating Point Unit

```
LD    i5 i6 i7
ADD   i7 i8 i7

FMUL   f5 f6 f7
FADD   f7 f8 f7
```

After successfully issuing, the IMUL marks i7 invalid. The ADD is prevented from issuing until the IMUL completes. The IMUL validation, over the CSW, is bypassed to the SZ stage Instruction Buffer (latched on clk) and the data result is bypassed to both the Instruction Buffer and the beginning of the EX stage (latched on clk).

WAR The M-Machine architecture guarantees *In-Order Issue* and thus prevents the results of subsequent operations from overwriting source values of previous operations.

```
LD    i5 i6 i7
SUB   i4 i8 i5
```

WAW The M-Machine architecture supports *Out-of-order* operation completion, and therefore provides *NO* guarantees regarding the order of writes.

The following instruction sequence is a race condition and is defined to be a programming error:

```
LD   i5 i6 i5
ADD  i6 #1 i5
LD   i5 i7 i8
```

The value of i5 is non-deterministic. If a programmer is concerned about the exact ordering of writes, the following sequence may be used:

```
LD   i5 i6 i5
MOV  i5 i5
ADD  i6 #1 i5
LD   i5 i7 i8
```

The additional MOVs guarantee that the IDIV will complete before the ADD is issued.

Control Hazards

Branches The IF stage double buffers the IP to correctly support the case of a branch during an ICACHE miss for the instruction in the third delay slot.

For either a taken branch or a jump, the integer execution stage supplies the new IP directly to the IF stage. It is the IF stage's responsibility to check for ErrVal in the provided pointer.

The M-Machine architecture has *three (3)* branch delay slots.

6.3.8 Exception Handling

The M-Machine architecture categorizes interrupts into two clases: exceptions and events.

Exceptions are synchronous and local to an individual cluster. Exceptions are handled by the dedicated X thread in each cluster. Each cluster may only have one exception active at any single instant.

Events are asynchronous and occur outside the clusters and require system software intervention. Memory synchronization faults and block status faults are examples of events. Events are queued in a hardware event queue and resolved by software event handlers.

The integer execution unit can generate an exception under the following four conditions:

1. ErrVal is a source operand

- For compare operation
- For BR or JMP
- For CCMOV or MOVCC
- For ISEND
- For SHIFT operation
- For SHIFT operation

2. Permission violation
   - For JMP
   - For ISEND
   - For LEA and LEAB

3. Non-Pointer
   - For SRC1 $\neq$ Pointer in ISEND

4. Illegal instruction (ILL)

Figure 6.12 shows the state transition diagram for the exception finite state machine (FSM) . The integer unit exception FSM performs the following four tasks after an exception is detected.

1. Set I_AV, M_AV, and F_AV to false
2. Write the IP into i4
3. Create the exception vector and write it to i5
4. Assert I_AV, M_AV, and F_AV to true

6.3.9 Arithmetic Operations

By convention, comparison operations are the *only* integer arithmetic operations that can generates an exception.

ADD and SUB Overflow during integer ADD, SUB return the low-order bits of the true two's complement result.

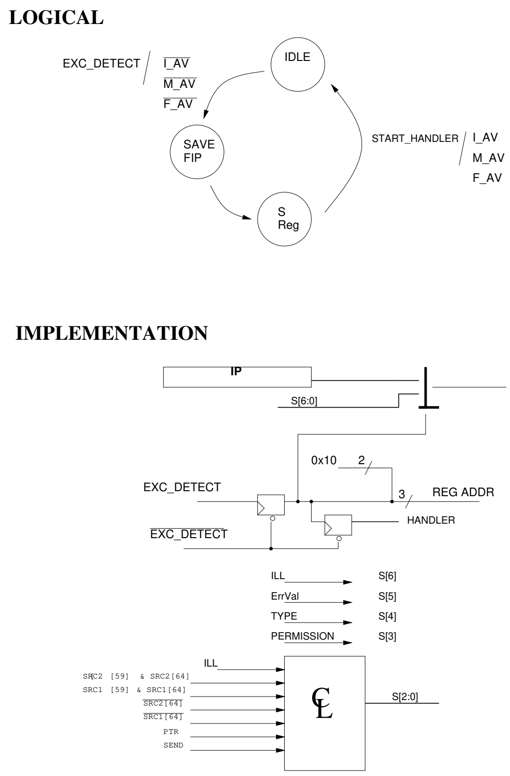


Figure 6.12: *Exception Handler FSM*

IMUL Note, IMUL and IMULU are performed by the Floating Point execution pipeline.

Overflow during signed integer multiplication (IMUL) generates the lower bits of the true two's complement result.

Overflow during unsigned integer multiplication (IMULU) also produces the lower bits of the true two's complement result.

Integer Division Note, IDIV and IDIVU are performed by the Floating Point execution pipeline.

Overflow during signed integer division (IDIV) produces an error value (ErrVal).

Unsigned integer division (IDIVU) can not overflow.

Divide by zero for either type of integer division produces an error value.

Conversion Note, FTOI and FTOIU are performed by the Floating Point execution pipeline.

Overflow during signed floating-point to integer conversion (FTOI) produces an error value.

Overflow during unsigned floating-point to integer conversion (FTOIU) also produces an error value.

6.3.10 Error Value Creation

The creation and usage of ErrVal's allows the MAP processor to schedule and perform speculative operations and improve performance. ErrVals do not raise exceptions unless they are used in a ST operation. Arithmetic operations on Errvals return the SRC1 Errval. Figure 6.13 shows the logical implementation of ErrVal creation hardware for the integer execution unit.

In the Integer execution pipeline error values can only created by under two conditions:

1. BR, LEA, LEAB, beyond segment
2. ErrVal used as one of the source operands for non-compare operation

6.4 Implementation

This section describes and discusses the implementation issues for the described logical design.

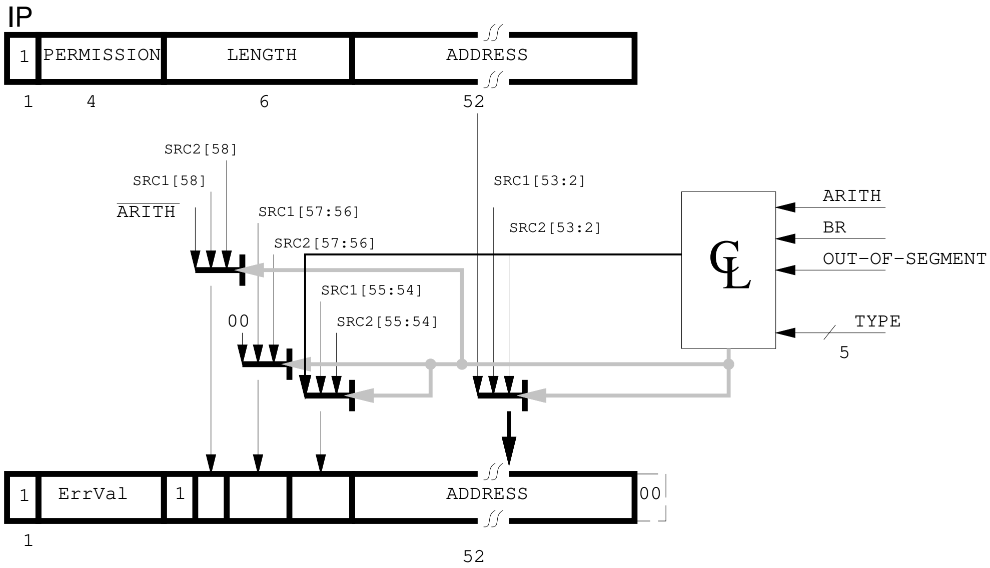


Figure 6.13: *Integer Unit Errval Creation*

6.4.1 Timing

Figure 6.14 shows the expected timing for a normal operation. All protection, exception, and conditional nullification checks are performed in parallel with operation execution. A final multiplexor selects the output supplied to the WB stage.

6.4.2 Circuits

A circuit study is currently in progress.

6.4.3 Area Estimate

An estimate of chip area is currently in progress.

The anticipated area for the EX register is $2M\lambda^2$. The current area estimate for the complete Integer EX stage including pipeline registers, 64-bit adder, shifter and boolean array is $20M\lambda^2$. Figure 6.15 shows the preliminary floorplan.

6.4.4 Power Consumption

Also, an estimate of chip power is currently in progress.

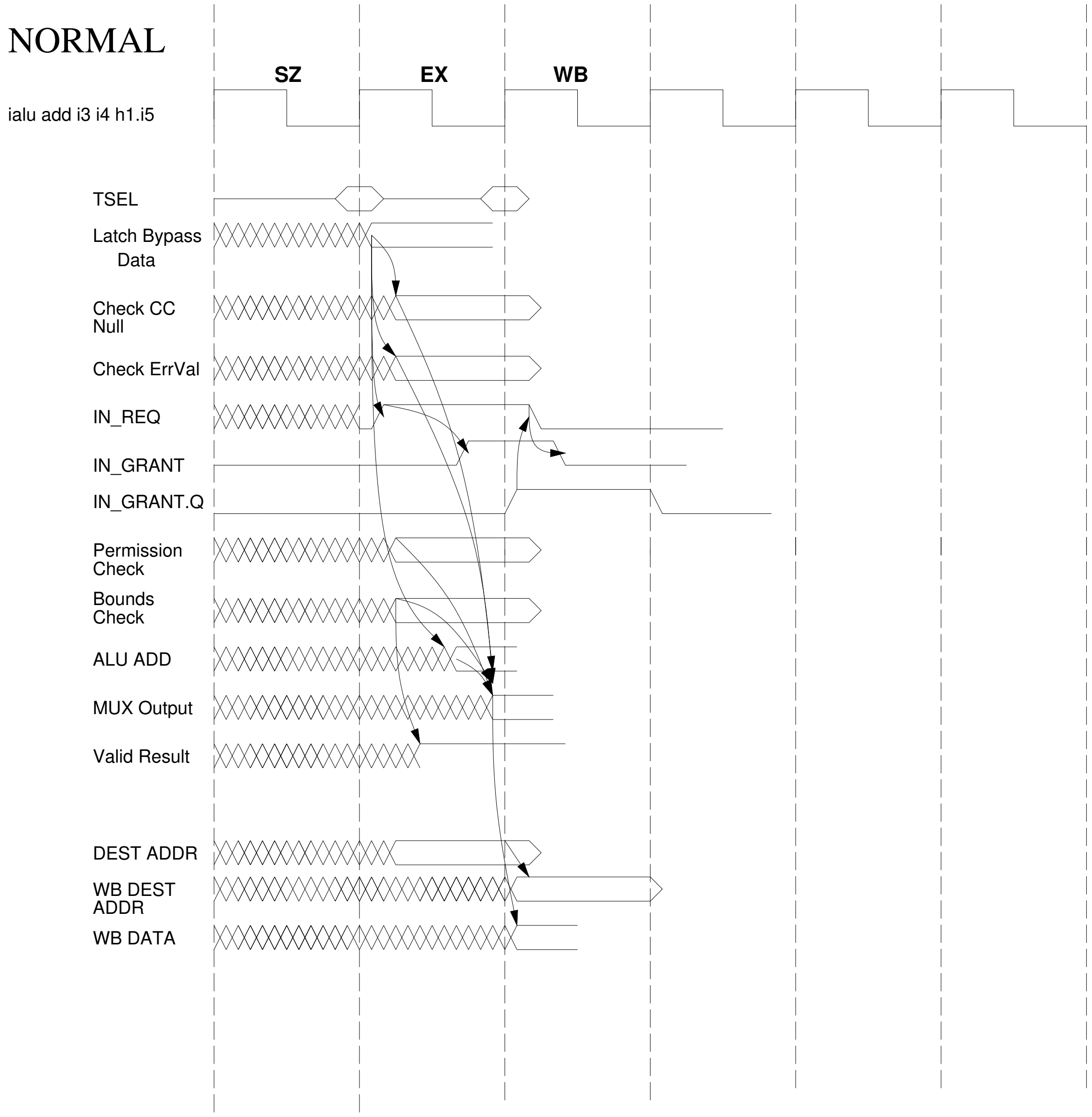


Figure 6.14: *Integer Unit Timing Diagram*

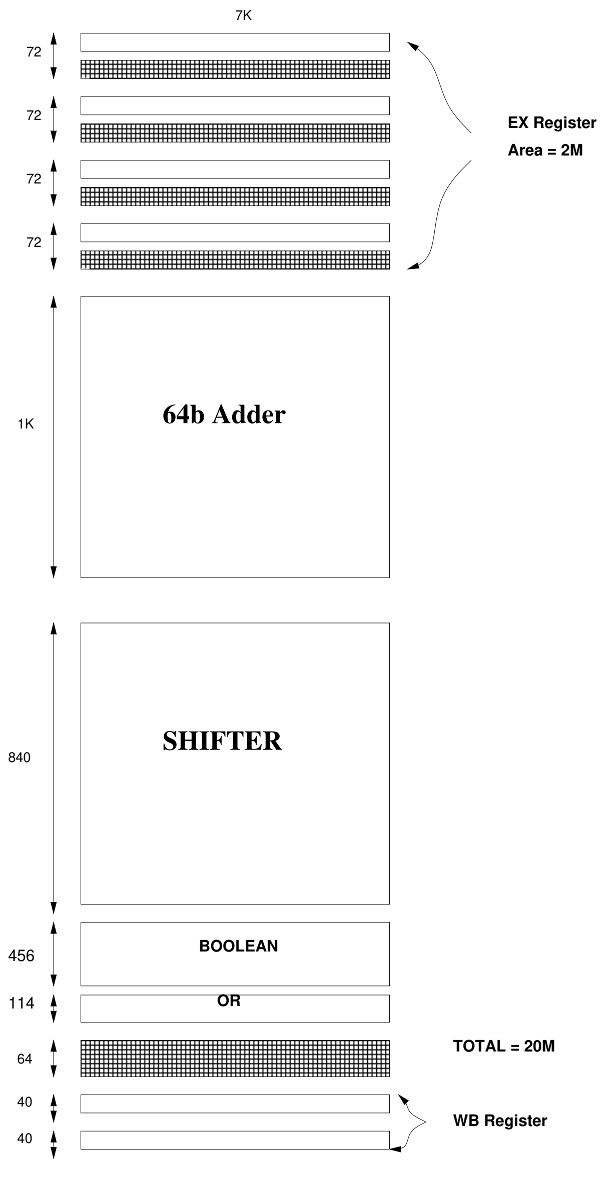


Figure 6.15: *Integer Execution Stage Floorplan*

6.5 OPEN ISSUES

The following issues remain to be resolved:

1. The implementation of CFG Space Pseudo Operations
2. Physical location of Local and Global CC Registers
3. Signal distribution for Local and Global CC Registers
4. The definition of the EX/CSW interface
5. The definition of the CFG/EX interface

Chapter 7

The Cluster Memory Execution Unit

7.1 Functional Overview

The memory unit (`MEMU`) of the MAP cluster it is one of the three units constituting the execution stage of the cluster pipeline. It executes all memory operations and a subset of logical/arithmetic operations, as specified in the *The MAP Instruction Set Reference Manual v1.1.* It executes the predecoded operations from the `SZ` unit either by computing the result in the case of logical/boolean operations, or by synthesizing the appropriate M-Switch request for load/store operations. It also incorporates *errVal* detection and generation circuitry and segment boundary checks for pointers.

7.2 Interface with other modules

The `MEMU` is one of three parallel execution and writeback stages of the pipeline. It interfaces to other modules with the following paths:

- with the `SZ` unit:
  - Predecoded operations to be executed.
  - Pipeline control: handshake/status signals to the `SZ` unit.
  - Result bypass to op buffer (for I results).
  - Invalidation to operation buffer (for I and FP and CC dest).
  - Validation to operation buffer (for I result).
- with the `RR` unit:
  - Invalidation to opsync buffer (for I and FP destination).
  - Validation to opsync buffer (for I result).

  - Invalidation to Integer and FP scoreboards (for loads).
  - Invalidation to CC scoreboards (for load/store cnds).
  - Writeback Validation to Integer scoreboard (for I result).
  - Result writeback to Integer register file (for I result).

- with the M-Switch output control
  - Load/store to memory
  - Load/store bypass to memory
  - Handshake Req./Grant lines

- with `IALU` and `FALU`: wired-OR exception line to signal an exception

- with exception logic (`XCPT`): exception control signals.

Fig. 7.1 shows an overview of the interfaces of the `MEMU` to other modules in the cluster.

7.2.1 Interface with Synchronization stage

The `MEMU` interfaces to `SZ` through a feed forward (operation flow) path and two feedback paths: validate/invalidate and writeback. Since these paths are independent from each other they are described separately.

The input `SZ`→`MEMU` interface

The feedforward path brings predecoded operations and operands from the `SZ`. It consist of a pipeline latch (ex input latch), a TLB-event shadow latch, and two flow control signals from the `MEMU` back to the `SZ`.

- **Flow control signals** The two flow control signals `mAvl` and `shAvl` signal to the `SZ` unit whether or not the `MEMU` is capable of accepting operations on the next cycle on the normal or shadow paths. *The TLB-handler shadow latch is only present in cluster 1* and is used exclusively by the TLB miss handler running on the system slot on that cluster. In this cluster, the `MEMU` decides in which of the two latches to latch the incoming operation based on the value of the `vslotId` field. If this field indicates `V-Thread S`, the bypass latch is used, otherwise the normal latch is used. On all other clusters the `shAvl` signal is always deasserted.
  - `mAvl`: memory unit available. Output from the `MEMU` back to the `SZ` to signal whether the unit is capable of accepting an operation in the next cycle (backpressure to stalls).

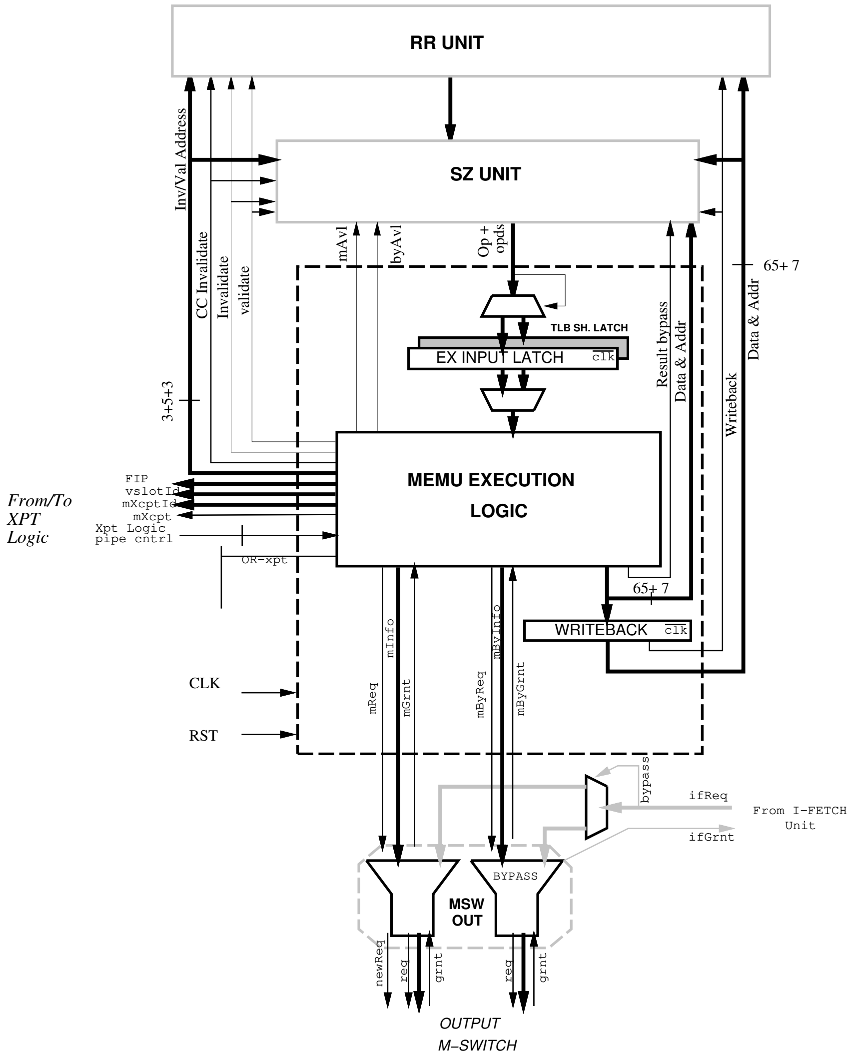


Figure 7.1: *Interface of the* MEMU *to other logic blocks.*

– shAvl: memory unit TLB-handler path available. Output from the MEMU back to the SZ to signal whether the unit is capable of accepting a TLB-event handler operation in the next cycle (this signal is alway deasserted except in cluster 1 where it provides handshake for the TLB-event handler path).

Fig. 7.2 shows the fields that are latched into the input latch (either ex-input or TLB-shadow latch) of the MEMU.

- **Thread information**

  – prs: instruction present bit. It signals whether or not there is an instruction at the input of the latch (all other fields should be ignored if this signal is not asserted).

Input Latch (most signif. portion)										
Thread info					*Opcode*		*Conditional*		*Sources*	
`prs`	`ill`	`mVld`	`vslotId`	`mOpIP`	`mGrp`	`mType`	`mOpCond`	`mOpCCV`	`mSrc1V`	`mSrc2V`
1	1	1	6	65	6	5	2	1	66	65 bits

MSB

Input Latch (least signif. portion)							
Destination							
`mDRF`	`mDh`	`mDregId`	`mDestV`	`mPincId`	`preC`	`postC`	`mCCdestId`
1	2	4	65	4	2	1	3 bits

LSB

Figure 7.2: *Organization of the* `MEMU` → `SZ` *interface input latch*

- `ill`: illegal operation. Input to `MEMU` from `SZ` stage to signal that the operation is an illegal operation. The `MEMU` will generate an exception whenever this signal is asserted.
- `mVld`: valid memory operation. Input to `MEMU` from `SZ` stage to assert that there is a valid memory operation. The table below summarizes the meaning of these 3 bits:

`prs`	`ill`	`mVld`	*Operation type*
0	X	X	op not present
1	0	1	valid op
1	0	0	`nop` (wired)
1	1	X	Illegal operation

- `vslotId`: one hot encoded V-Thread slot identifier (6 bits). It specifies the thread slot to which the operation belongs. It is coded as {X,S,D,C,B,A}
- `mOpIP`: instruction pointer value for that operation (65 bits).

• **Opcode** predecoded opcodes

- `mGrp`: operation group (6 bits). The bits in this group specify a set of characteristics of the operation class:

Group field					
`in_mb`	`cfg`	`load`	`store`	`arith`	`bool`

with the following meaning:

* `in_mb`: the operation has incremented the memory barrier counter.
* `cfg`: configuration space access (the pointer is of configuration type or the operation is a configuration operation)
* `load`: load type operation (src1 contains address)

* `store`: store type operation (destination contains address)
* `arith`: arithmetic operations
* `bool`: boolean operations

- `mType`: operation code within the group (5 bits). Selects an operation within a group. The coding of the boolean and arithmetic operations are the same as on the `IALU`.

  (*****USE SAME CODINGS AS `IALU` *****)

Operation	**Type field**	**Privil.**
	Boolean	
	(XOR NOT AND/OR)	
OR		
AND		
NOT		
XOR		

`mType` definition cont.:(*****USE SAME CODINGS AS `IALU` . Waiting for UADD, USUB decision*****)

Operation	**Type field**	**Privil.**
Arithmetic		
	(PTR EQ SIGN $\overline{\text{SRC1}}$ $\overline{\text{SRC2}}$)	
ADD (NOP)		
ADDU		
SUB		
SUBU		
ILL		
MOV		
	(PTR BASE $\overline{\text{SRC1}}$ $\overline{\text{SRC2}}$)	
LEA LEAB		

`mType` definition cont.: (****IGNORE THESE CODINGS FOR NOW*****)

Operation	**Type field**					**Privil.**
Store						
	(SYN	PINC	CCDEST	FP-RF	I-RF)	
ST	0	0/1	0	0	1	
FST	0	0/1	0	1	0	
STS	1	0	0	0	1	
STSU	1	0	0	1	0	
STSCND	1	0	1	0	1	
FSTS	1	0	0	1	0	
FSTSU	1	0	0	1	0	
FSTSCND	1	0	1	1	0	
STNA	0	0	0	0	1	
FSTNA	0	0	0	0	1	
SMS	0	0	0			P
FSMS	0	0	0			P
FLNE	0	0	0			
FCA	0	0	0			P
Load						
LD	0	0/1				
LDS	1	0				
LDSU	1	0				
LDSCND	1	0				
LUC	0	0				
LMS	0	0				P

`mType` definition cont.:

Operation	Type field	Privil.
config		
GETCSTAT		P
PUTCSTAT		P
GPRB		P
HFORK		
HEXIT		
VGSU		P
VAC		P
SSTPRB		P

- **Conditional** condition for execution of operation
  - `mOpCond`: condition for execution of operation (2 bits encode : UA, UN, CT , CF)
  - `mOpCCV`: CC value that must match condition for execution (1 bit: true or false)
- **Sources**
  - `mSrc1V`: source 1 operand value (66 bits, presence bit also required)
  - `mSrc2V`: source 2 operand value (65 bits)
- **Destinations**
  - `mDRF`: destination register file: integer or float (1 bit).
  - `mDh`: destination H-Thread (2 bits)
  - `mDregId`: identifier of destination register (4 bits)
  - `mDestV`: destination value (65 bits)
  - `mPincId`: identifier of postincrement register (4 bits)
  - `preC`: precondition field specifier (for synchronizing memory ops).
  - `postC`: postcond field specifier (for synchronizing memory ops).
  - `mCCdestId`: destination condition code reg identifier.

`MEMU`→`SZ` invalidate/validate Interface

The invalidate/validate interface is required to:

- invalidate the destination of load operations (the validation in done when the data comes back through the C-Switch)
- invalidate CC dest (2nd destination of conditional sync load operations)
- validate destination of boolean operations

Therefore a maximum of one Int or FP register, plus one CC register must be addressable in any one cycle. The interface derives the addresses from the destination fields in the ex-input latch. It outputs the following signals during the first half of the clock cycle:

mIVSlotId	mIVRegId	mVal	mInv	mInvCCId	mInvCC
3	5	1	1	3	1

- `mIVSlotId`: thread identifier part of the invalidate/validate address(es)
- `mIVRegId`: invalidate/validate register Identifier (including int/FP reg file).
- `mVal`: validate register. When asserted the register addressed by `mIVSlotId` plus `mIVRegId` is validated.
- `mInv`: invalidate register. When asserted the register addressed by `mIVSlotId` plus `mIVRegId` is invalidated.
- `mInvCCId`: cc register identifier to invalidate.
- `mInvCC`: invalidate cc register. When asserted the cc register addressed by `mIVSlotId` plus `mInvCCId` is invalidated.

MEMU→SZ result bypass Interface

This interface conveys the result of the operation to the SZ unit corresponding to previously validated registers to the SZ and RR units. It outputs the following signals during the second half of the clock cycle:

mBypVslotid	mBypRegId	mBypRes	mBypss
3	4	65	1

- `mBypVslotid`: thread identifier part of bypass address
- `mBypRegId`: bypass register identifier (for Int RF only, 4 bits only)
- `mBypRes`: bypass result data
- `mBypss`: if asserted the bypass path has a valid value.

7.2.2 MEMU→RR Writeback Interface

This interface conveys the result of the operation to the RR for the integer register file. It outputs the following signals during the complete clock cycle from a short delay after the rising edge:

mWbThid	mWbRegId	mWbRes	mWback
3	4	65	1

- `mWbThid`: thread identifier part of writeback address
- `mWbRegId`: writeback register identifier (for Int RF only, 4 bits only)
- `mWbRes`: writeback result data
- `mWback`: if asserted the writeback path has a valid value.

7.2.3 Interface to M-switch control block

The memu has two ports on the M-Switch one for the normal path and one for the shadow path. Each interface port to the M-Switch consists of 4 handshake signals and a data vector, called `mswInfo` (`mswShInfo`) which contains both the operation to be performed and the return id for the destination. This data follows a path parallel to the writeback one.

	Handshake			
	`mReq`	`mReqCfg`	`mbank`	`mGrnt`
Width (bits)	1	1	2	1
Dir	↓	↓	⇓	↑

M-Switch Out (most signif. portion)				
Transfer			*Address*	
`mAccType`	`mpreC`	`mpostC`	`mphys`	`maddr`
7	2	1	1	64 bits

MSB

M-Switch Out (least signif. port.)						
Return					*Data*	
`mRtnClst`	`mTslotId`	`mRtnRf`	`mRtnRegId`	`mRtnCC`	`msync`	`mdata`
2	3	1	4	3	1	65 bits

LSB

The signals have the following meaning (see also M-Switch description)

- **Handshake**
  - `mReq`: request signal to M-Switch
  - `mGrnt`: grant signal from M-Switch
  - `mReqCfg`: destination is config space

– `mbank`: destination memory bank

- **Transfer**

  – `mAccType`: specifies memory operation to be performed (7 bits) coded as follows:

MSB | | | | | | LSB

type	r/$\overline{\text{w}}$	sz	ft	cdnl	$\overline{\text{cach}}$	byps
if *type* = "1"	read/$\overline{\text{write}}$ 1 bit	synchronize 1 bit	faulting 1 bit	conditional 1 bit	$\overline{\text{caching}}$ 1 bit	shadow 1 bit

if *type* = "0"	00000ζ	*Abort Transaction*
	00010ζ	FLNE
	00011ζ	FCA
	00100ζ	PUTCSTAT
	00101ζ	GETCSTAT
	01000ζ	SMS
	01001ζ	LMS
	01000ζ	FSMS
	10100ζ	GPRB
	10000ζ	HFORK
	10001ζ	HEXIT
	10010ζ	VGSU
	10011ζ	VAC
	10101ζ	SSTPRB
	11111ζ	*errVal* flow through

ζ: shadow (bypass) path bit

  – `mpreC`: precondition (reflect `preC` in input latch)

  – `mpostC`: postcondition (reflect `postC` in input latch)

- **Address**

  – `mphys`: Physical address (reflects the pointer type in the address)

  – `maddr`: destination address (64 bits)

- **Return**

  – `mRtnClst`: return cluster. The return cluster identifier is computed as `mRtnClst` = $abs.clusterid + rel.H - Tid \bmod 4$. Where the absolute cluster id. is hardwired in the cluster, and the relative H-Thread id. is given by `mDh`.

  – `mTslotId`: return V-Thread slot encoded into 3 bits.

  – `mRtnRf`: return register file

  – `mRtnRegId`: return register id

  – `mRtnCC`: return CC reg

- **Data**

  – `msync`: synchronization bit

  – `mdata`: data

7.2.4 Interface `MEMU` ↔`XCPT` unit

The interface to the exception logic consists of the following signals:

- `mXcpt` handshake signal
- `mXcptId` a 7-bit exception identifier
- `FIP` faulting IP
- `vslotId` the V-Thread slot number of the faulting operation

`mXcpt` is used by the `MEMU` to signal the occurrence of an exception, while `mXcptId` specifies what type of exception has occurred.

The `mXcpt` signal also drives a global OR line `ORXcpt` common to `MEMU`, `FALU`, `IALU` and `XCPT`. Locally in the `MEMU` the assertion of the `mXcpt` line also cause the deassertion of the `mAvl` and `shAvl` lines which signals the `SZ`stage the inability to accept any more instructions. The `mAvl` line is also deasserted whenever the `ORXcpt` line is asserted signalling an exception in another unit. The assertion of the `ORXcpt` signal causes in the same cycle:

- detection by the `IALU` and `FALU` of the execption condition and thus deassertion of the corresponding `iAvl` and `fAvl` lines. No operation can advance while an exception is being posted in hardware (note however that the writeback stage is free to complete the current operation).
- detection by the `XCPT` unit, which asserts its input to the `ORXcpt` line, thus stalling all execution pipelines until the exception is successfully posted.

The `XCPT` unit (external to the MEMU) should perform the following sequence of steps:

- when `mXcpt` is detected, it should assert the `ORXcpt` line within the same cycle
- it reads the V-Thread id., the IP, and the `mXcptId` from the `MEMU` and saves them in the *FIP* and *S*registers of the exception thread slot.
- *(if multiple exceptions result from operations in the same instruction the exception identifier of all exceptions is saved)*
- it sets the Exception Bit in the cluster (which is later cleared in software through config space).
- If the exception is caused by an Event or an Exception H-Thread (identifiable by the V-Thread slot id) it signals a catastrophic exception.
- It deasserts the `ORXcpt` line so that the pipeline can restart. (However, from now until the service of the exception in software is completed only the exception H-trhread will be able to issue instructions).

7.2.5 Signal timing

Fig. 7.3 shows the expected signal timing for a typical memory operation: `ld i3, #8, f2;`

7.3 Internal architecture

The internal architecture of the `MEMU` is shown in Fig 7.4. The execution logic consists of the following blocks:

- validate/invalidate logic (VINV)
- operation validity check logic (OPCHK)
- operand validity check logic(OPDCHK)
- operation decoding logic(OPDEC)
- 64 bits integer adder/boolean (2's complement)(MPLUS)
- M-Switch request formatter(FRMT)
- *errVal* generation logic(EVGEN)
- Segment bound checks (for lea, leab and all postinc ops)(BNDCHK)
- Pipeline control logic (generation of `mAvl` and `shAvl`)(MPCNTRL)

7.3.1 Validate/Invalidate logic

The Validate/Invalidate logic module, VINV, receives the `mDregId` and `mCCdestId` and operation information from the input latch and under control of the operation decoder asserts the validate or invalidate (`mDregId`or `mCCdestId`or both). Three control lines `mVal`, `mInv` and `mInvCC` from the OPDEC control the operation of the VINV.

7.3.2 Operation validity check logic

The operation validity check logic (OPCHK) has to perform the following checks on the operation during the first half cycle of execution:

- check the presence bit to ensure that an operation is present, otherwise signal the pipeline control to ignore the contents of the operation and execute a NOP instead.

```
(V=latch data)|                   |      EXECUTE      |     WRITEBACK     |
              |            latch IN new op       latch WB res             |
              |                   |                   |                   |
clk         __/~~~~~~~~~\_________/~~~~~~~~~\_________/~~~~~~~~~\_________/
              |         |         |         |         |         |         |
prs        xxxxxxxxxxxxxxxxxxx~~~~~x        |         |         |         |
vslotid    xxxxxxxxxxxxxxxxxxx~~~~~x        |         |         |         |
INop+opd   xxxxxxxxxxxxxxxxxxx~~~~~x        |         |         |         |
              |         |         |         |         |         |         |
mAvl         _____________/~~~~~~~~\_______ |         |         |         |
              |         |         |         |         |         |         |
mByAvl      ______________/~~~~~~~~\_______ |         |         |         |
              |         |         V         |         |         |         |
EXINP LTCH(-) |         |<load&post load&post>        |         |         |
              |         |         |         |         |         |         |
mInv         ___________________/~~~~~~~~~~~~~\______ |         |         |
              |         |         |         |         |         |         |
mIVRegId     ======================<f2f2f2f2f2>=======|         |         |
              |         |         |         V         |         |         |
INTER LTCH(+) |         |         | <load&post load&post>       |         |
              |         |         |         |         |         |         |
mReq         _________________________/~~~~~~~~~~~~~~~~\________|         |
              |         |         |         |         |         |         |
mReqCfg      =======================\__________________=======  |         |
              |         |         |         |         |         |         |
mbank        =======================<   i3Value[4:3]    >=======|         |
              |         |         |         |         |         |         |
mGrnt       _______________________________/~~~~~~~~~~\________ |         |
              |         |         |         |         |         |         |
mswInfo       ======================<opcod,addr,dest=f2>======= |         |
              |         |         |         |         |         |         |
mBypss        _________________________________/~~~~~~~\________          |
              |         |         |         |         |         |         |
mBypRegId    ================================<i3i3i3i3i3>________         |
              |         |         |         |         |         |         |
mBypRes      =======================xxxxxxxxxxxxx<i3Val+8>xxxxxxx|
              |         |         |         |         V         |         |
WRBACK LATCH  |         |         |         | <  postinc result  >        |
              |         |         |         |         |         |         |
mWback        ________________________________________/~~~~~~~~~~~~~~~~~~\___
              |         |         |         |         |         |         |
mWbRegId      ==========================================<i3i3i3i3i3i3i3i3>=
              |         |         |         |         |         |         |
mWbRes        =========================================< i3Val+8  i3Val+8>=
              |         |         |         |         |         |         |
```

Figure 7.3: Expected signal timing for the interface signals between `MEMU` and other modules for load with postinc: `ld i3, #8, f2`.

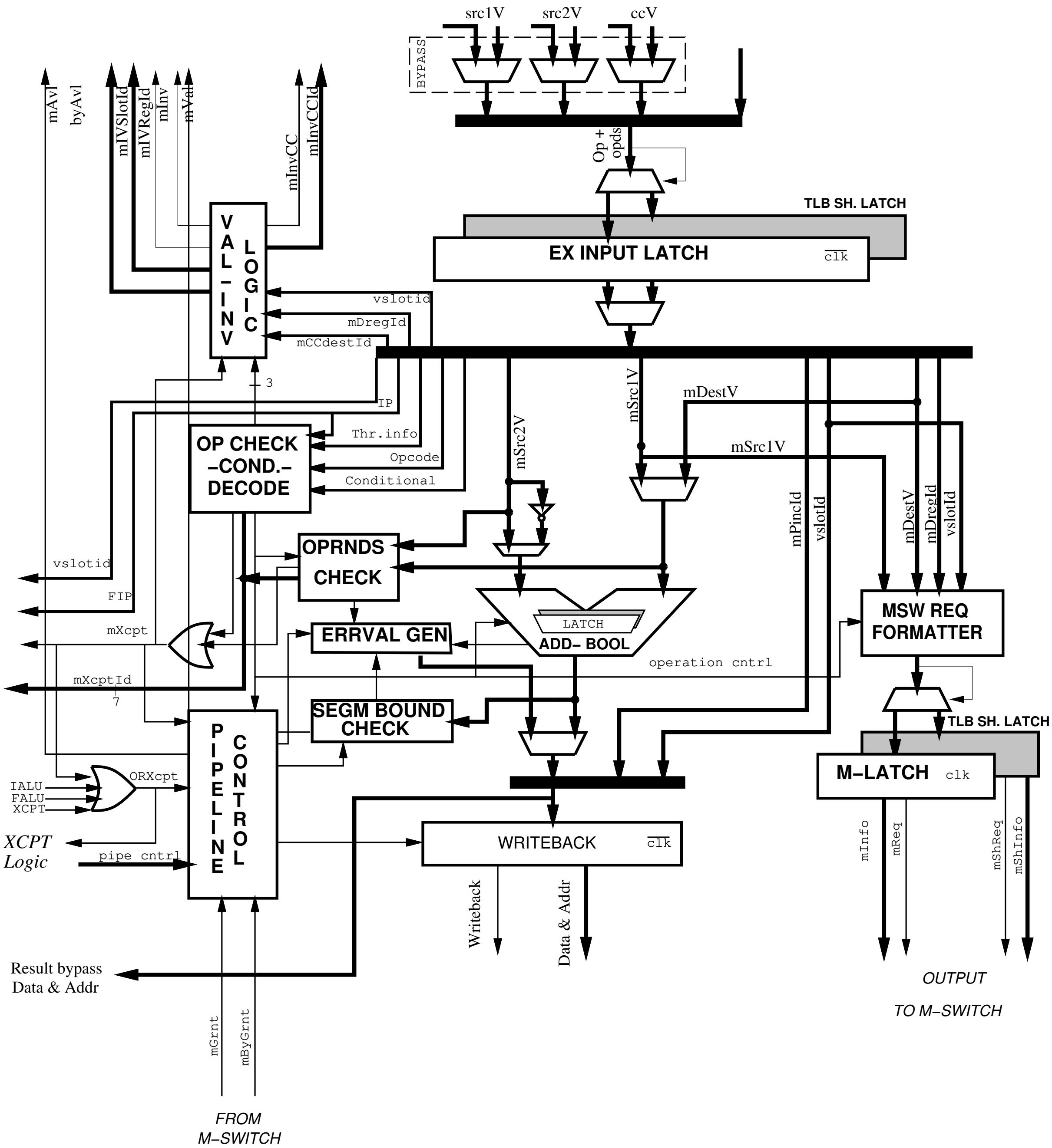


Figure 7.4: Internal architecture of the MEMU

- determine whether the condition for execution is satisfied. Compare the value `mOpCCV` against the condition in `mOpCond`, and signal a NOP if the condition is false.
- detect a "wired" nop by decoding the `prs`, `ill` and `mVld` signals.
- detect a "wired" illegal operation by checking the `ill` signal. Signal it to the exception logic.

The signalling of exceptional conditions takes places by asserting the `mXcpt` and outputting a `mXcptId` corresponding to the detected exception.

7.3.3 Operand validity check logic

The operand validity check logic (OPDCHK) ensures that operands are of the correct type for the current operation. It always checks for *errVal* and pointer type on both operands and signals the result of the check to the PCNTRL. It outputs 6 permission signals (`prmss`), three per operand, to the PCNTRL: `rdPrmss` `wrPrmss` and `errvaldet` according to the following table:

	Outputs		
Operand	Read	Write	errval
Integer	0	0	0
Pointer Read Only	1	0	0
Pointer Read Write	1	1	0
Pointer Execute User	1	0	0
Pointer Execute Priv	1	0	0
Pointer Execute Phys	1	1	0
Pointer Enter User	0	0	0
Pointer Enter Priv	0	0	0
Pointer Enter Msg	0	0	0
Pointer Config Space	1	1	0
Pointer Key Only	0	0	0
Pointer Physical	1	1	0
Pointer Errval	0	0	1

7.3.4 Operation Decoding Logic

The operation decoding logic (OPDEC) provide control signal for the data processing unit (MPLUS, and FRMT), so that they perform the correct function. It should:

- decode the opcode and generate appropriate control signals for the MPLUS and FRMT.
- select the appropriate check to be performed by the OPCHK.

- select the appropriate validation/invalidation to be performed by the VINV.
- decide whether or not to ignore the result of the bound check logic in the BNDCHK
- Exception handling: assert the `mXcpt` signal and the appropriate `mXcptId` if one or more of the following occurs:
  - a privileged operation is encountered and the instruction pointer capabilities do not allow the execution of such operation
  - opcode not recognized
  - no write permission in `mDestV` during STORE
- ErrVal handling:
  - *errVal* detected on either operands for any arithmetic operation: signal PCNTRL to set MPLUS to to "forward" the *errVal* input operand unchanged to its output
  - *errVal*detected in `mDestV` during a STORE operation: it asserts `mXcpt` and corresponding `mXcptId` For STORE with postincrement an *errVal*must also be written in the `mPincId`register.
  - no read permission on operand `mSrc1V` (LEA, LEAB, LOAD): request EVGEN for *errVal* and signal PCNTRL to set data path accordingly. For LOAD with postincrement two *errVal* must be generated one for the incremented pointer and one for the destination value. (this may require an additional EVGEN module).

7.3.5 Integer Adder - Boolean network

The integer adder/boolean network (MPLUS) is the computation unit within the `MEMU`. It computes the following subset of arithmetic/boolean operations:

- signed add: $(s)c = (s)a + (s)b$
- unsigned add: $c = a + b$
- signed sub: $(s)c = (s)a - (s)b$
- unsigned sub: $c = a - b$
- signed pointer increment: $p = p + (s)inc$
- bitwise OR: $c = a|b$
- bitwise AND: $c = a\&b$
- bitwise NOT: $c = \tilde{}a$
- bitwise XOR: $c = a \oplus b$

- move (either pointer or integer): $c = a$ (also used for pass-through `mSrc1V`)
- pass through `mSrc2V` (for *errVal* propagation): $c = b$
- request *errVal* generation on overflow

7.3.6 Errval generation

The *errVal* generation logic (EVGEN) synthesizes an *errVal* that replace the actual result in those case where an exception cannot be detected in the first half cycle. The EVGEN always formats the "address" and the "permission" fields of the *errVal* in parallel with any operation execution. The error type (encoded in the length field) is generated later according to the error that occurs. The conditions under which the *errVal* is generated are:

- bound check fail: during an LEA, LEAB or any operation that uses a postincrement
- load operations with illegal pointer types. If a load is attempted with a `mSrc1V` which is an integer, entry pointer or key pointer, the `mDregId` of the load is filled with an *errVal*. In addition, if the operation specified a postincrement, the `mPincId`register also gets an *errVal* with the same IP but different *errVal* type identifier.
- arithmetic overflow for signed arithmetic operations

7.3.7 Segment bound checks

The segment bound check logic (BNDCHK) verifies that the newly generated address is not out of the segment boundary. The BNDCHK module is always active, but its output is ignored by the PCNTRLfor all operations other than LEA, LEAB and all postincrement LOAD and STORE. If also requests the EVGEN to generate the appropriate *errVal*.

7.3.8 M-switch request formatter

It formats load and store operations in the format required by the M-Switch. It synthesize config space operations and formats them in a store or load understandable by the config space controller. The following are the memory operations supported by this module:

- ST
- FST
- STS
- STSU

- STSCND
- FSTS
- FSTSU
- FSTSCND
- STNA
- FSTNA
- SMS
- FSMS
- FLNE
- FCA
- LD
- LDS
- LDSU
- LDSCND
- LUC
- LMS

The following are config space operation supported by this modules.

- GETCSTAT
- PUTCSTAT
- GPRB
- HFORK
- HEXIT
- VGSU
- VAC
- SSTPRB

ζ= shadow path

	M-Switch Out													
OP	Handshake		*Transfer*			*Address*		*Return*					*Data*	
	`mReqCfg`	`mbank`	`mAccType`	`mpreC`	`mpostC`	`mphys`	`maddr`	`mRtnClst`	`vslotId`	`mRtnRf`	`mRtnRegId`	`mRtnCC`	`msync`	`mdata`
ST	`mDestV`=cfg?	`mDestV`[4:3]	100000ζ	XX	X	`mDestV`=phys?	`mDestV`	XX	XX	XXXX	XXXX	XXXX	`mSrc1V`	
FST	`mDestV`=cfg?	`mDestV`[4:3]	100000ζ	XX	X	`mDestV`=phys?	`mDestV`	XX	XX	XXXX	XXXX	XXXX	`mSrc1V`	
STS	`mDestV`=cfg?	`mDestV`[4:3]	101000ζ	`preC`	`postC`	`mDestV`=phys?	`mDestV`	XX	XX	XXXX	XXXX	XXXX	`mSrc1V`	
STSU	`mDestV`=cfg?	`mDestV`[4:3]	101100ζ	`preC`	`postC`	`mDestV`=phys?	`mDestV`	clst + `mDh`%4	`vslotId`	XXXX	XXXX	`mCCdestId`	`mSrc1V`	
STSCND	`mDestV`=cfg?	`mDestV`[4:3]	101010ζ	`preC`	`postC`	`mDestV`=phys?	`mDestV`	clst + `mDh`%4	`vslotId`	XXXX	XXXX	`mCCdestId`	`mSrc1V`	
FSTS	`mDestV`=cfg?	`mDestV`[4:3]	101000ζ	`preC`	`postC`	`mDestV`=phys?	`mDestV`	XX	XX	XXXX	XXXX	XXXX	`mSrc1V`	
FSTSU	`mDestV`=cfg?	`mDestV`[4:3]	101100ζ	`preC`	`postC`	`mDestV`=phys?	`mDestV`	clst + `mDh`%4	`vslotId`	XXXX	XXXX	`mCCdestId`	`mSrc1V`	
FSTSCND	`mDestV`=cfg?	`mDestV`[4:3]	101010ζ	`preC`	`postC`	`mDestV`=phys?	`mDestV`	clst + `mDh`%4	`vslotId`	XXXX	XXXX	`mCCdestId`	`mSrc1V`	
STNA	`mDestV`=cfg?	`mDestV`[4:3]	100001ζ	XX	X	`mDestV`=phys?	`mDestV`	XX	XX	XXXX	XXXX	XXXX	`mSrc1V`	
FSTNA	`mDestV`=cfg?	`mDestV`[4:3]	100001ζ	XX	X	`mDestV`=phys?	`mDestV`	XX	XX	XXXX	XXXX	XXXX	`mSrc1V`	
SMS	`mDestV`=cfg?	`mDestV`[4:3]	001000ζ	XX	X	`mDestV`=phys?	`mDestV`	XX	XX	XXXX	XXXX	XXXX	`mSrc1V`	
FSMS	`mDestV`=cfg?	`mDestV`[4:3]	001000ζ	XX	X	`mDestV`=phys?	`mDestV`	XX	XX	XXXX	XXXX	XXXX	`mSrc1V`	
FLNE	0	`mSrc1V`[4:3]	000010ζ	XX	X	X	`mSrc1V`	XX	XX	XXXX	XXXX	XXXX	X	XXXX
FCA	0	00	000011ζ	XX	X	X	XX	XX	XX	XXXX	XXXX	XXXX	X	XXXX
LD	`mSrc1V`=cfg?	`mSrc1V`[4:3]	110000ζ	XX	X	`mSrc1V`=phys?	`mSrc1V`	clst + `mDh`%4	`vslotId`	`mDRF`	`mDregId`	XXXX	X	XX
LD from *errVal*	`mSrc1V`=cfg?	`mSrc1V`[4:3]	111111ζ	XX	X	X	XX	clst + `mDh`%4	`vslotId`	`mDRF`	`mDregId`	XXXX	X	*errVal*
LDS	`mSrc1V`=cfg?	`mSrc1V`[4:3]	111000ζ	`preC`	`postC`	`mSrc1V`=phys?	`mSrc1V`	clst + `mDh`%4	`vslotId`	`mDRF`	`mDregId`	XXX	X	XX
LDSU	`mSrc1V`=cfg?	`mSrc1V`[4:3]	111100ζ	`preC`	`postC`	`mSrc1V`=phys?	`mSrc1V`	clst + `mDh`%4	`vslotId`	`mDRF`	`mDregId`	`mCCdestId`	X	XX
LDSCND	`mSrc1V`=cfg?	`mSrc1V`[4:3]	111010ζ	`preC`	`postC`	`mSrc1V`=phys?	`mSrc1V`	clst + `mDh`%4	`vslotId`	`mDRF`	`mDregId`	`mCCdestId`	X	XX
LUC	`mSrc1V`=cfg?	`mSrc1V`[4:3]	110001ζ	XX	X	`mSrc1V`=phys?	`mSrc1V`	clst + `mDh`%4	`vslotId`	`mDRF`	`mDregId`	XXXX	X	XX
LMS	`mSrc1V`=cfg?	`mSrc1V`[4:3]	001001ζ	XX	X	`mSrc1V`=phys?	`mSrc1V`	clst + `mDh`%4	`vslotId`	`mDRF`	`mDregId`	XXXX	X	XX
GETCSTAT	0	XX	000101ζ	XX	X	`mSrc1V`=phys?	`mSrc1V`	clst + `mDh`%4	`vslotId`	`mDRF`	`mDregId`	XXXX	X	XX
PUTCSTAT	0	XX	000100ζ	XX	X	`mDestV`=phys?	`mDestV`	XX	XX	XXXX	XXXX	XXXX	`mSrc1V`	
CONFIG SPACE														
GPRB	1	XX	010100ζ	XX	X	X	XX	clst + `mDh`%4	`vslotId`	`mDRF`	`mDregId`	`mCCdestId`	`mSrc1V`	
HFORK	1	XX	010000ζ	XX	X	X	`mSrc1V`	clst + `mDh`%4	`vslotId`	XXXX	XXXX	`mCCdestId`	`mSrc2V`	
HEXIT	1	XX	010001ζ	XX	X	X	XX	XX	XX	XXXX	XXXX	XXXX	X	XXXX
VGSU	1	XX	010010ζ	XX	X	X	XX	clst + `mDh`%4	`vslotId`	`mDRF`	`mDregId`	`mCCdestId`	X	XX
VAC	1	XX	010011ζ	XX	X	X	`mSrc1V`	clst + `mDh`%4	`vslotId`	`mDRF`	`mDregId`	`mCCdestId`	X	XX
SSTPRB	1	XX	010101ζ	XX	X	XX	XX	clst + `mDh`%4	`vslotId`	`mDRF`	`mDregId`	`mCCdestId`	`mSrc1V`	
	157	155	148	146	145	144	79	77	74	73	69	66	65	0
	MSB													LSB

Handling of *errVal*s

When a Load is attempted with an illegal pointer (incorrect pointer type or *errVal*) the EVGEN generates an appropriate *errVal* that must be forwarded to the `mDregId`. In this case the FRMTmust still request a normal memory access through the M-switch but the `mAccType` field must specify an *errVal* forwarding. When such message arrives at the memory system the *errVal* contained in the `mDestV`field (usually ignored during loads) must be sent back through the C-Switch to reach the specified `mDregId`. The solution of forwarding *errVal* through the Memory system that may seem rather cumbersome, but it is necessary because:

- `mDregId` may not be reachable from the `MEMU` (if it is a floating point reg. or a register on another cluster), and
- if the operation requires postincrement the register file write port is already used by the postincrement and cannot be used to locally write the *errVal*.

7.3.9 Pipeline control logic

The pipeline control logic (PCNTRL) generates control signals for the pipeline latches to select whether they should latch in the next operation (normal function), maintain their value (pipeline stall) or reset. It also arbitrates for those pipeline resources (data processing units and writeback register) that are shared between the normal path and the TLB shadow path. Finally, it generates the interface flow control signals `mAvl` and `shAvl` for the `SZ` unit of the cluster.

- assert of `mAvl` under the following conditions: (****TO BE CONFIRMED BY STEVE****)
  - if `prs` bit on input latch is cleared (empty latch), or
  - if `prs` bit on input latch is set and no exception is detected, and
    * in the case of an operation that accesses the M-Switch `mGrnt` has been asserted, or
    * in the case of an arithmetic or boolean operation, the `prs` bit in the TLB shadow input register is cleared. Operation in TLB shadow latches have priority over normal operations and they will always win the writeback latch, therefore the operation in the execution input latch cannot proceed.
- assert of `shAvl` under the following conditions: (****TO BE CONFIRMED BY STEVE****)
  - if `prs` bit on TLB shadow input latch is cleared (empty), or
  - if `prs` bit on TLB shadow input latch is set and no exception is detected, and in the case of an operation that accesses the M-switch `mShGrnt` has been asserted

The PCNTRL also needs to select the data path (i.e. normal result path, *errVal* forwarding, *errVal* gen path etc..) according to the result of the operation and operand checks:

7.3.10 Exception Flagging

To summarize, the following conditions generate an exception in the `MEMU`:

- store to an *errVal* pointer
- illegal opcode
- attempt to execute a privileged operation in a non-privileged code segment

7.4 Floorplan and expected size

TBD

7.5 Open issues

- decide if implement bot signed and unsigned ops
- verify bypassing path to other units
- check on issuing policy bypass vs. normal path
- do we need sync ld and st in cfg space
- verify M-switch out format table to be compatible with memory system

Chapter 8

The Cluster Floating Point Execution Unit

Chapter 9

The Cluster Miscellaneous ciruitry: CFG and XPT logic

Chapter 10

The C-switch

10.1 Functional Overview

The intercluster communication switch (C-Switch) allows communication among the clusters of the multi-ALU processor (MAP), and from the cache banks, the EMI and the configuration space controller to the clusters. Fig. 11.1 shows an overview of the MAP chip with the C-Switch highlighted.

The C-Switch has a total of 10 input ports and 4 output ports. The 10 input ports are connected to EMI, GCSC, 4 cache banks and 4 clusters. Each of the 4 output ports connects to one of the clusters. Up to four concurrent transfers can take place in any clock cycle.

As shown in fig. 10.2, the C-Switch consists of two submodules: a data path switch and a control section. The data path is implemented as a set of four busses. Each bus has 10 input ports and one output port. The cluster's Integer unit, FP unit and Netout controller share a single port out of the module. The module's input interface demultiplexes the output port among various destinations. The C-Switch controller perform the arbitration, manages the handshake with the ports and controls the setting of the data path.

10.1.1 Operation

A transfer via the C-Switch in caused by one of the following events:

- explicit MOVE operations in either user or system threads (only within the same V-Thread),
- any operation that targets a register on a remote register file, even within the same cluster (only within the same V-Thread),
- any SEND operation,

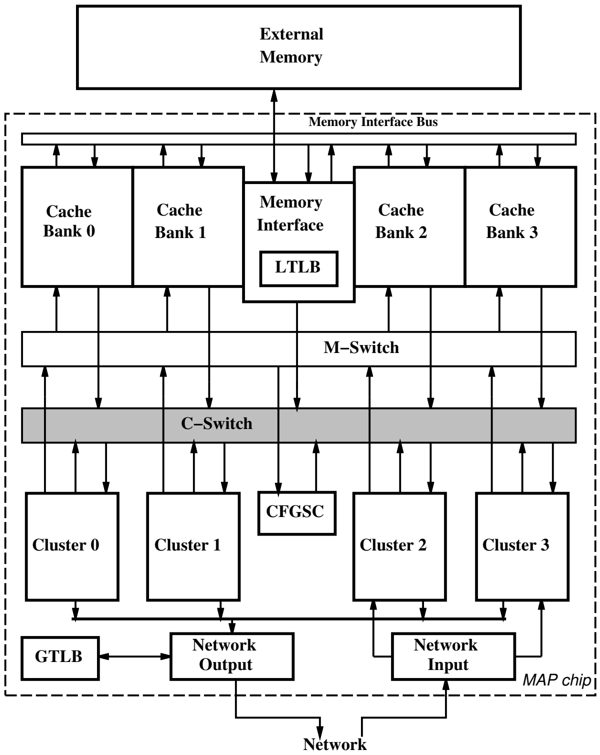


Figure 10.1: *The C-Switch within the MAP chip*

- the system software accesses particular resources, such as event queues or configuration space accesses.

The C-Switch supports two transfer modes: *single* and *burst*. The transfer mode is selected by the source module and it is independent for each of the output ports. Under single transfer mode the C-Switch arbitrates among the possible requesting modules, on a cycle–by–cycle basis, independently for the output port. In burst mode, the source module can transfer multiple words with a single initial arbitration by maintaining exclusive access rights to the output port for the duration of the burst. During a burst transfer, arbitration is disabled, thus all other requests are effectively ignored. To ensure fairness, burst mode transfers cannot be invoked directly by user code, they are specifically targeted to be used by the hardware or system software to efficiently transfer a finite sequence of data words in a non-interruptable manner.

A transfer via the C-Switch is pipelined over two cycles. In the first cycle the module attached to an input port requests a transfer, the C-Switch performs the arbitration and grants the connection to one of the contending requesters. On the following cycle the winner of the arbitration can then proceed to transfer the data across. The arbitration and the set up of the connection is done by the C-Switch control logic based on the destination information provided by the requesting module A transfer across the C-Switch can be requested on every cycle and as long as long as access is granted, data transfer can also take place at every cycle

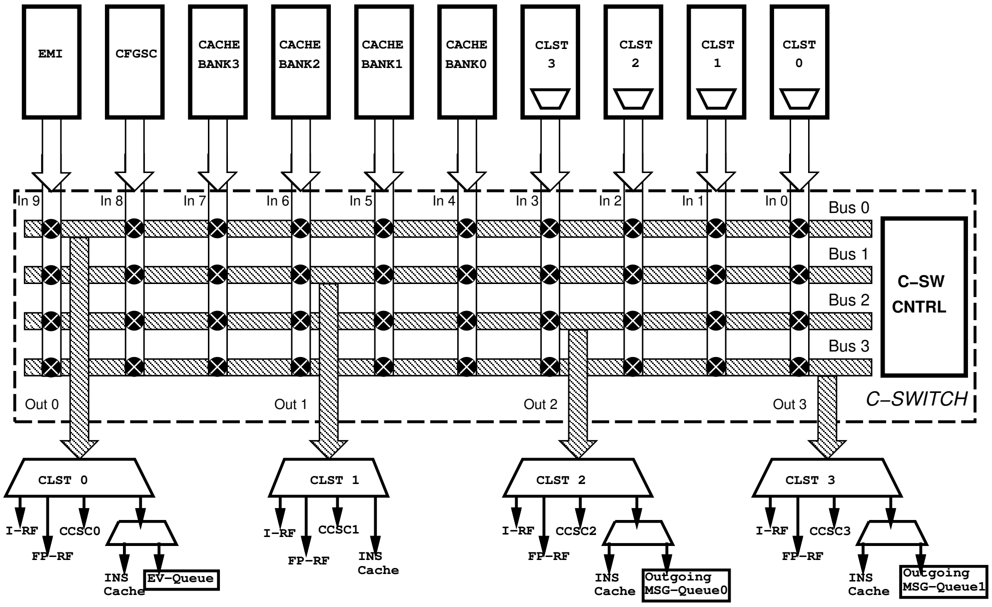


Figure 10.2: *High level view of the C-Switch*

10.1.2 Access priorities

The input ports to the C-Switch are classified in 4 tiers of priority. They are connected to the following modules:

1. Top Priority
   (a) *input port 9*: External Memory Interface (EMI)
2. High Priority
   (a) *input port 8*: Global Configuration Space Controller (GCSC)
3. Medium Priority
   (a) *input port 7*: Cache Bank 3
   (b) *input port 6*: Cache Bank 2
   (c) *input port 5*: Cache Bank 1
   (d) *input port 4*: Cache Bank 0
4. Low Priority
   (a) *input port 3*: Cluster 3 (Integer or FP Units)
   (b) *input port 2*: Cluster 2 (Integer or FP Units)
   (c) *input port 1*: Cluster 1 (Integer or FP Units)

(d) *input port 0*: Cluster 0 (Integer or FP Units)

Across tiers, request with a higher priority level is always guaranteed to win when arbitrating with one of a lower level. Within the Medium and Low priority tiers, requests are serviced in the order of arrival and a fixed priority (higher port number, higher priority) is maintained between requests with equal arrival time. This scheme, even without ensuring starvation, is expected to work well in the C-Switch case because of the traffic characteristics. The higher priority ports are connected to modules (such as cache banks, EMI and GCSC) that generate traffic only in response to a request originated by a module connected to a lower priority port (clusters). This arrangement means that the response of a previous request has priority over a new request, ensuring dynamic equilibrium through back pressure.

Each of the 4 cluster has a single output port connecting to an input port of the C-Switch. Thus, such port must be shared between the Integer Floating Point unit and the Netout controller.

10.1.3 Sharing of output ports

The 4 C-Switch output ports feed to the 4 clusters. Within each cluster the data is demultiplexed to different destinations:

1. Output Port 0
   (a) Cluster 0 Integer Register File (One per V-Thread slot)
   (b) Cluster 0 Floating Point Register File (One per V-Thread slot)
   (c) Cluster 0 Instruction Cache
   (d) Event Queue: enqueueing port
   (e) Cluster 0 Configuration Space Controller (CSCC 0)
2. Output Port 1
   (a) Cluster 1 Integer Register File (One per V-Thread slot)
   (b) Cluster 1 Floating Point Register File (One per V-Thread slot)
   (c) Cluster 1 Instruction Cache
   (d) Cluster 1 Configuration Space Controller (CSCC 1)
3. Output Port 2
   (a) Cluster 2 Integer Register File (One per V-Thread slot)
   (b) Cluster 2 Floating Point Register File (One per V-Thread slot)
   (c) Cluster 2 Instruction Cache
   (d) Output Message Queue 0: enqueueing port

(e) Cluster 2 Configuration Space Controller (CSCC 2)

4. Output Port 3

   (a) Cluster 3 Integer Register File (One per V-Thread slot)
   (b) Cluster 3 Floating Point Register File (One per V-Thread slot)
   (c) Cluster 3 Instruction Cache
   (d) Output Message Queue 1: enqueueing port
   (e) Cluster 3 Configuration Space Controller (CSCC 3)

10.2 Interfacing to the C-Switch

All inputs ports of the C-Switch are identical except for the built–in priority level. Thus, any module can attach to any port. One or more of the information fields can be redundant for particular modules, in which case these fields can be ignored. Similarly, all output port are organized in identical ways, even though the mapping of system resources to clusters makes them structurally different (e.g., cluster 1 doesn't have any type of queue). However, the difference among the clusters are only visible at the system level and it is responsibility of the system code to ensure address correctness while accessing a system resource. No checking is done in the C-Switch for a valid cluster–resource combination.

10.2.1 Structure of the input ports

Each of the 10 input ports of the C-Switch is 93 bits wide, the 3 most significant bits are used for handshake, the remaining carry data and associated information. The input ports are organized as shown in fig. 10.3. All signals are input from the module to the C-Switch

Handshake			*Control*			*Source*		*Destination*				*Condition*		*Data*	
req	hld	grnt	lk	mb	Xfr type	sclstid	sslotid	Dst clst	Dst Tslot	rf	Data reg	CC flag	CC reg	Mem sync	Data word
1	1	1	1	1	3	2	3	2	3	1	4	1	3	1	65 bits
MSB															LSB

Figure 10.3: *Organization of C-Switch input port*

except the `in_grnt` which is a handshake signal going from the C-Switch back to the module. The signals have the following meaning (the naming convention "`in_signalname`" is used to identify the signal as belonging to the *group* "`in`"):

- *Handshake*

- `in_req`: **request** for a C-Switch access
- `in_hld`: **hold** signal used to indicate that the port is requesting a "burst" transfer, i.e., a multi-cycle access to the C-Switch. Once granted the corresponding bus will be assigned to the the input port while hold is maintained asserted (see sec. 10.2.1 for more details).
- `in_grnt`: **grant** signal to indicate that the port has won arbitration for the bus and data can be delivered on the next cycle

• *Control*

- `in_lk`: **late kill signal**. When asserted at any time during the transfer cycle, the receiver ignores the transfer.
- `in_mb`: **memory barrier flag**. When asserted indicates that the receiver should decrement its memory barrier counter as a side result of receiving this transfer.
- `in_xfrType`: **transfer type**. This field indicate the type of transfer in progress. The meaning of the transfer type code varies depending which slot is addressed, as shown in the following table:

XfrType [2:0]	Meaning		
	dstSlot 0-5 (Threads)	dstSlot 6 (IFU-Queue)	dstSlot 7 (CCSC)
000	`abort transfer`		
001	data only transfer	Queue: data	t.b.d.
010	cc only transfer	Queue: t.b.d.	t.b.d.
011	data and cc	Queue: t.b.d.	t.b.d.
100	data only with mem sync		t.b.d.
101	`reserved`	IFU read	t.b.d.
110	`reserved`	IFU write	t.b.d.
111	`Command in data field`		

The "abort transfer" code is used for a late cancellation of a transfer as explained in sec. 10.2.1. The "Command in data field" is used to extend the set of transfer types beyond the 8 provided by the 3 bit encoding. When this code is used, the transfer type will be found in 8 m.s.b of the data field of the transaction. More details to be defined

• *Source*

- `in_sClstId`: (sender cluster id.) identifies the cluster that generated the transfer (2 bits)
- `in_sSlotId`: (sender slot id.)identifies the slot that generated the transfer (3 bits)

• *Destination*

- `in_dstClst`: **destination cluster** indicates to which cluster the transfer is destined. This field must be valid also during the arbitration cycle to ensure that the request are forwarded to the appropriate arbiter and the correct bus is reserved for data transfer in the next cycle.

- `in_dstSlot`: **V-thread slot identifier** within a cluster, this field identifies as the destination either one of the 6 V-thread slots, the instruction cache or the system (event or output message) queue. In addition the Cluster Configuration Space Controller (CCSC) is mapped tom the same slot id as the queue, and it is distinguished by the transfer type field (see `in_xfrType` above). Note that only the enqueueing ports of the system queues can be reached through this path, and depending on the cluster one of three queus is addressed: Cluster 0: Event queue, Cluster 2: Message queue Priority 0, Cluster 3: Message queue Priority 1. The dequeueing side of the system queues is mapped to some register of the integer register file of the corresponding cluster as specified in the *M-Machine Architecture v1.0*. The different destinations are coded as follows:

Tslot [2:0]	Meaning
000	User V-Thread A
001	User V-Thread B
010	User V-Thread C
011	User V-Thread D
100	System V-Thread S
101	Exception V-Thread X
110	Queue (event, message queue 0 or message queue 1) or Instruction Cache, depending on value of transfer type
111	CCSC

- `in_rf`: **register file identifier** indicates either the integer (0) or the FP (1) register file. This field is valid only if the Tslot field is set to indicate one of the V-Thread slots.
- `in_reg`: **register identifier** field indicates one out of the 16 possible registers in the register file. This field is valid only if the Tslot field is set to indicate one of the V-Thread slots.

• *Condition*

- `in_ccflg`: **condition code flag** value to be written to the condition register.
- `in_ccreg`: **condition register** indicates one of 6 possible registers as the destination of the transfer

• *Data*

- `in_memSync`: **memory synchronization bit**. The synchronization bit of the memory system
- `in_data`: **data word**: 65 bits of data including the pointer bit.

The C-Switch ignores the significance of most of the bits fed at the input ports. It simply forwards the data to the appropriate output port. It does however rely on the handshake signals and destination cluster information to arbitrate and then route the incoming data. The collection of fields which are simply forwarded by the C-Switch, i.e. the Data, Condition, Destination (with the exclusion of the cluster id field) and the Control fields are collectively named "inout datapath" (or simply datapath).

Sequence of handshake signals

In this section we present examples of sequences of handshake signals at an input port during various types of transfers. Fig. 10.4 shows a timing diagram for one requester and a single transfer across the C-Switch. The port asserts the `in_req` signal and the corresponding

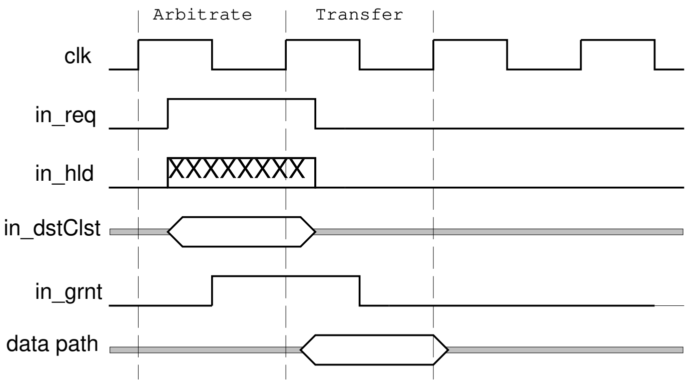


Figure 10.4: *Timing diagram for a single requester and a single transfer across the C-Switch*

`in_dstClst` field to request a C-Switch transfer. The C-Switch responds by asserting the `in_grnt` to indicate that the requested bus is being granted to this port on the following cycle. The Requester deasserts the `in_req` on the next cycle after the `in_grnt` is asserted. Note that the `in_hld` is effectively ignored (XX= don't care) during the arbitration cycle.

For the correct operation of the C-Switch, the `in_req`, `in_dstClst` (and `in_hld`) signals must be valid within the first quarter of the clock cycle. Furthermore the `in_dstClst` field must be held valid whenever the `in_req` or `in_hld` signals are asserted. The `in_grnt` signal is guaranteed valid at the end of that cycle with a reasonable set up time to the rising clock edge[1].

Because the arbitration is always done one cycle ahead of the actual data transfer, consecutive transfers from the same or different requesters can be pipelined, as shown in fig. 10.5.

Fig. 10.6 shows a timing diagram two requesters arbitrating for the same output port of the C-Switch.

[1]Exact timing requirements will be given after circuit simulation.

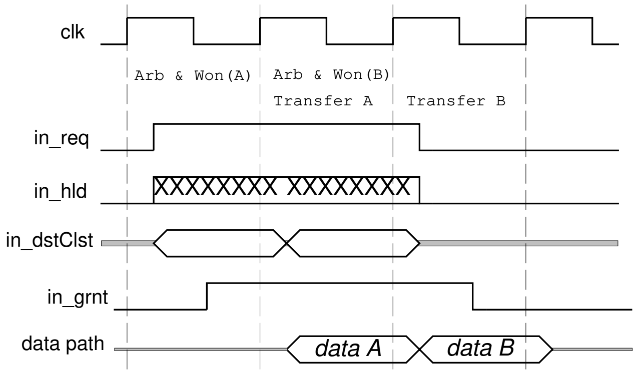


Figure 10.5: *Timing diagram for a single requester executing two consecutive transfers*

Note that requester 1, which lost the arbitration during the first cycle, maintains the `in_req` signal asserted until it obtains access (the corresponding `in_grnt` is asserted), except if the request must be aborted (see sec. 10.2.1).

Burst mode transfers are initiated by asserting the `in_hld` signal and deasserting the `in_req` in the cycle following the one in which arbitration was won. As shown in fig. 10.7 the `in_grnt` signal is kept asserted for the whole length of the burst.

For the duration of the burst, arbitration is disabled, thus other request (even of higher priority) are ignored. Fig. 10.8 shows two requesters contending for the same port with the first one entering burst mode.

The burst is normally terminated by deasserting the `in_hld` signal. The first cycle in which `in_hld` is deasserted is the last data transfer cycle for that burst. In this cycle arbitration is also reactivated and the output port can be granted access for the next cycle.

A burst transfer can also be terminated by reasserting the `in_req` (without dropping `in_hld`). This allows back–to–back burst transfers from a single requester without the need for intermediate cycles, as shown in fig. 10.9.

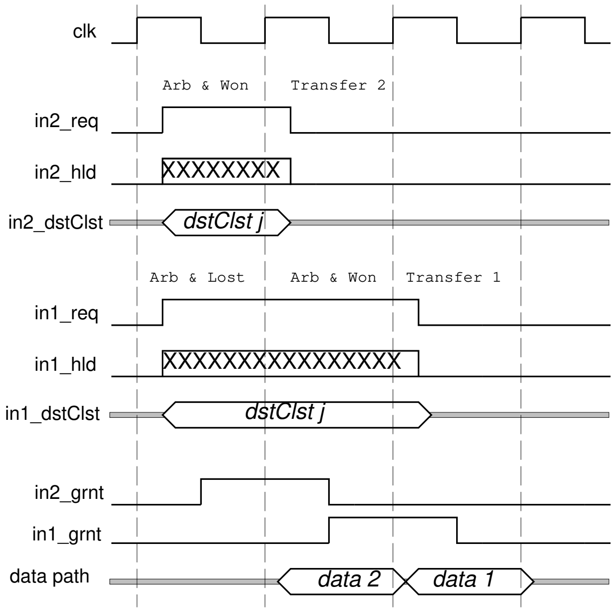


Figure 10.6: *Timing diagram for two requesters arbitrating for an output port*

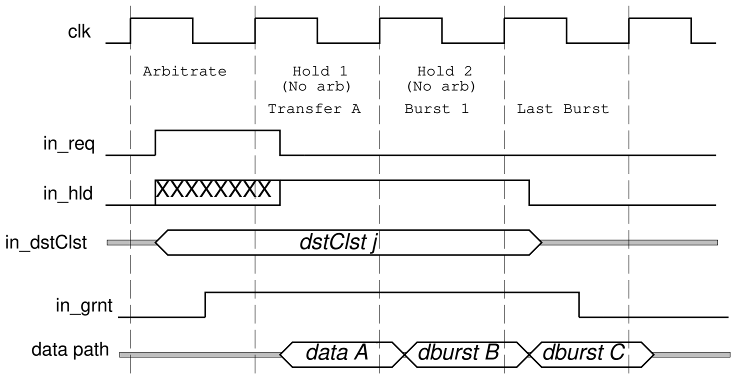


Figure 10.7: *Timing diagram for a requester executing a burst mode transfer*

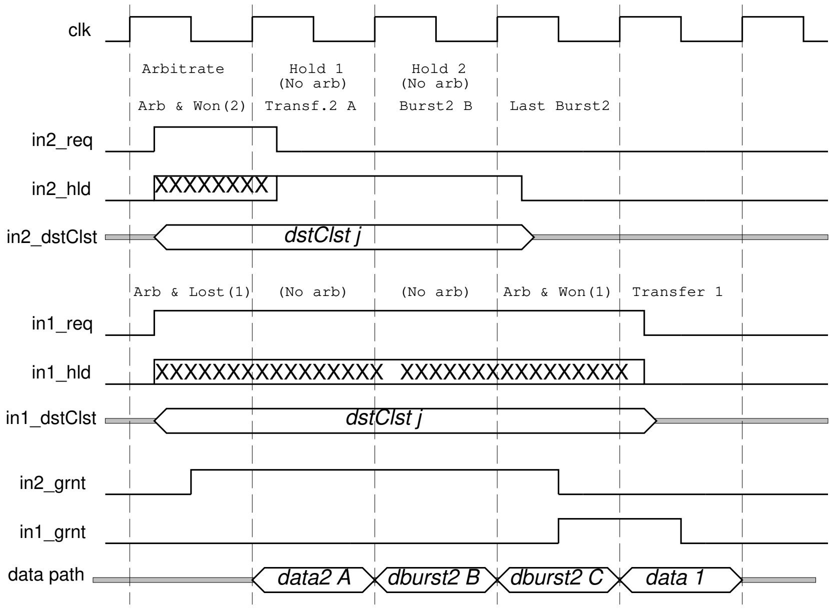


Figure 10.8: *Timing diagram for two arbitrating requesters, one of which in burst mode transfer*

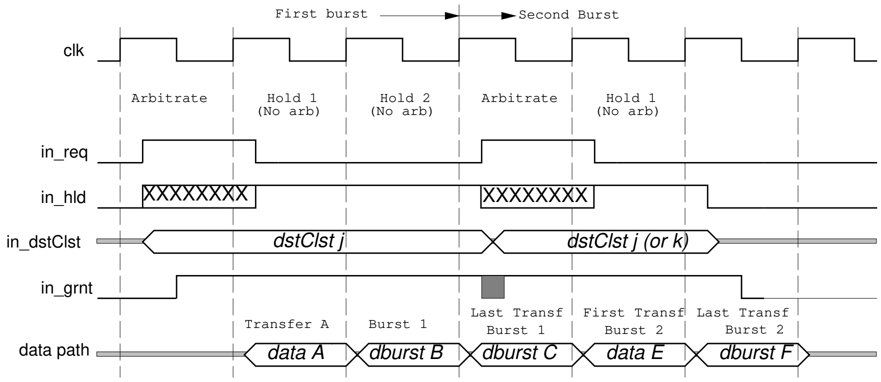


Figure 10.9: *Timing diagram for two back-to-back burst transfers from the same requester*

Aborting C-Switch requests

Because of the built–in pipeline, initiated C-Switch requests can be aborted in only one of two ways, depending on the state of the request.

A typical example of a situation that requires a transfer cancellation is when a cluster requests a C-Switch transfer at the start of an operation's EX cycle and the execution of the operation results in an exception. In this case the C-Switch transfer should be aborted because the data is not present. The cancellation of such transfers should take place in one of the following two ways:

a. **if the cluster had lost arbitration** and should wait for subsequent arbitration cycles, then it can safely abort the request by deasserting the `in_req` signal (and `in_hld` if asserted) during the first quarter of the next clock cycle (deassertion of the `in_req` signal during the end of a clock cycle may result in unrecoverable error).

b. **if the cluster had won arbitration for the next cycle**: then it is too late to abort the transfer of data for the next cycle in the way described above (the C-Switch is already committed). Therefore, the cancellation is performed by proceeding ahead with the transfer as if no abort were required, except that the `in_xfrType` field in the data path will be set to `abort request`, to indicate that the current is a null transfer.

Transfer Protocol Controller

Each of the requesters attached to the input ports should include a finite state machine controller with the functionality shown in fig. 10.10

The signals `start`, `burst`, `last` and `abort` are internally generated by other hardware modules within the cluster.

10.2.2 Structure of the output ports

The output ports are 90-bits wide and carry a subset of the input signals plus an additional control signal generated by the arbiter (`out_dav`). Namely, there is a one to one correspondence between the input and the output datapath *except* for `in_req`, `in_grnt` and `in_dstClst`, which are only required by the C-Switch control logic, and thus are not forwarded to the output. The `in_hld` signal is forwarded both to the arbiter and switched to the appropriate output port as part of the datapath.

The output ports are organized as shown in fig. 10.11 (signal names are preceded by "`out_`").

The `out_dav` signal is used to indicate to the module attached to the output port that there is data available on that bus.

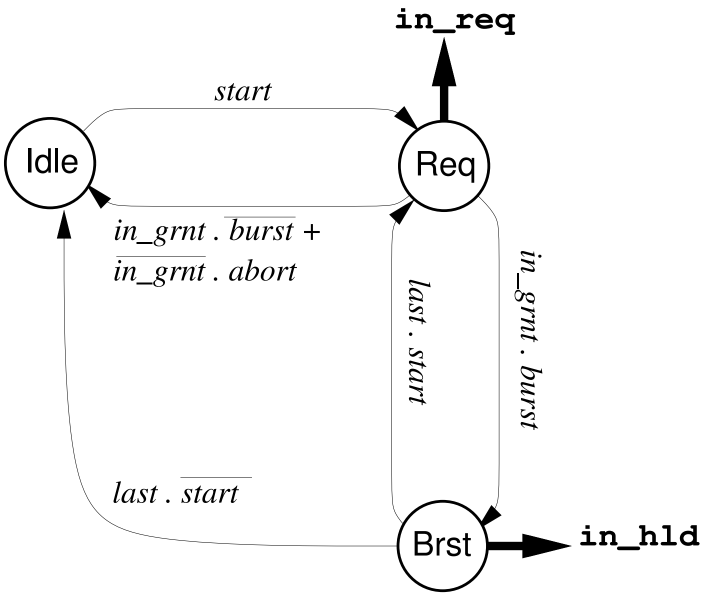


Figure 10.10: *Finite state Machine of requester connected to an input port*

Handshake		*Control*			*Source*		*Destination*			*Condition*		*Data*	
dav	hld	lk	mb	Xfr Type	ssclstid	sslotid	Tslot	rf	Data reg	CC flag	CC reg	Mem sync	Data word
1	1	1	1	3	2	3	3	1	4	1	3	1	65 bits
MSB													LSB

Figure 10.11: *Organization of C-Switch output port*

10.3 Internal architecture

As shown in fig. 10.12, the C-Switch consists of two submodules: a data path module and a control module. The data path module is implemented as a set of four busses. The data portion of the input port feeds directly to the busses while the handshake signals go to the controller. The controller performs arbitration, manages the handshake and controls the operation independently for each of the 4 busses. Four 10-bit wide select busses (`SelBus0[9:0]`, ..., `SelBus3[9:0]`) carry the setting from the controller to the data switches.

10.3.1 The C-Switch data path

Fig. 10.13 shows the internal organization of the busses module of the C-Switch Each bus is 89-bits wide. The access of an input port to a bus is controlled through a set switching

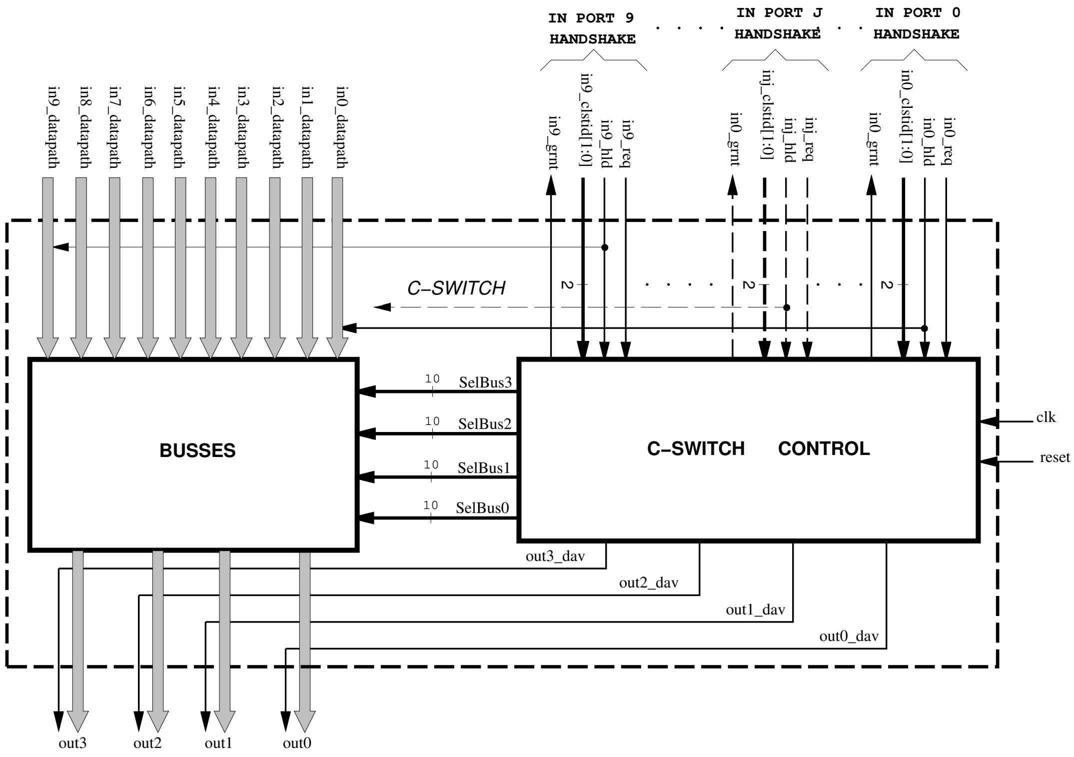


Figure 10.12: *Internal structure of the C-Switch*

points (represented as a black circle with a white cross). Each control bus `SelBus{0-3}` controls the corresponding bus, and each of the 10 bits in the selection bus controls one of the switch points. A one–hot encoding of `SelBus{0-3}` bits ensures that at most one input port is enabled onto each bus during any clock cycle.

Fig. 10.14 shows the internal organization of one of the switch points. Each switch point consists of a single NMOS transistor (ratioed 16:2) per bus line, all controlled by a single bit of the `SelBus`.

In order to reduce power consumption, all busses are low-voltage swing. Special buffer/drivers (TX) and receiver/amplifier (RX) are used to step down and regenerate voltage levels. Fig. 10.15 shows the circuit for a single C-switch line.

10.3.2 The C-Switch controller

The C-Switch controller contains 4 independent arbiters (one per bus), a request decoder and a combiner. Fig. 10.16 shows the internal organization of the control module.

The request decoder routes the requests (`in_req` and `in_hld`) to the appropriate bus arbiter according to the value of the `in_dstClst` field.

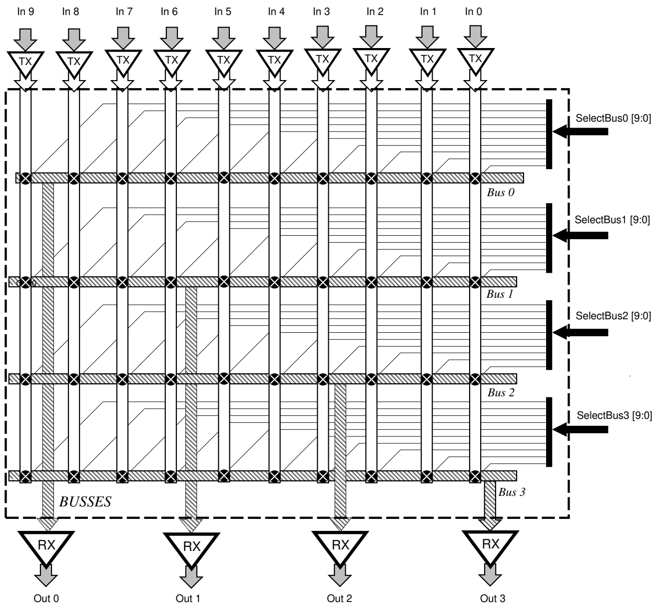


Figure 10.13: *Internal structure of the busses module*

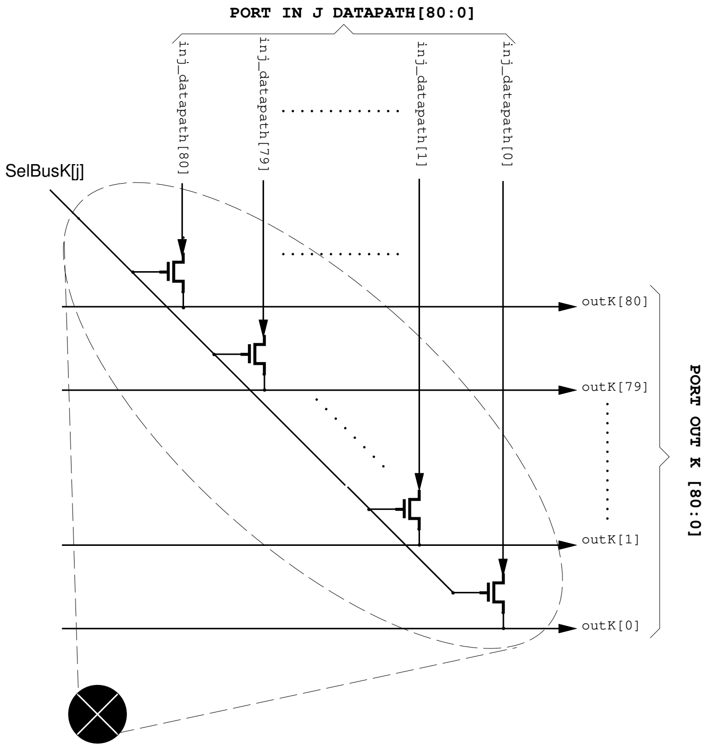


Figure 10.14: *Internal structure of a switch point*

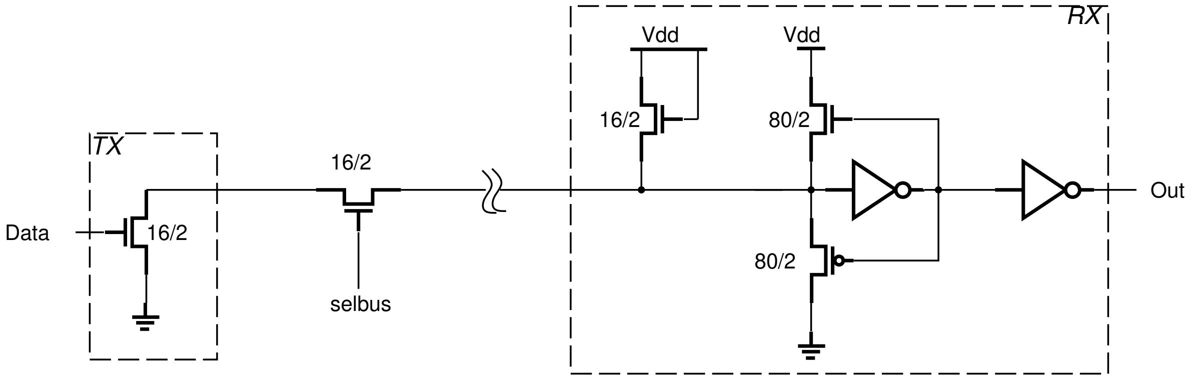


Figure 10.15: *Circuit for one line of the C-Switch bus*

Each bus arbiter accepts 10 `in_req`, `in_hld` pairs and outputs a 10 bit wide vector of `arb`j`Grnts` signals to provide handshake at the input ports. It also generates a `SelBus` vector that controls the switch points of the corresponding bus and `out_dav` signal fed to the corresponding output port.

The combiner performs bit OR function of the four `arb{0-3}Grnts` buses to combine the corresponding signals from each arbiter into one `in_grnt` per input port. It implements the following function `in`j`_grnt = arb0Grnts[`j`] | arb1Grnts[`j`] | arb2Grnts[`j`] | arb3Grnts[`j`]`, $\forall\, j : 0 \leq j \leq 9$.

The bus arbiter

Internally the bus arbiter is organized as shown in fig. 10.17. The heart of the bus arbiter is a set of four cascaded arbitration modules. The two requests `arbReq9` and `arbReq8` (respectively of Top and High priority levels) are handled by two 1 bit prioritizers (1-BIT PRIOR). The two groups of requests of medium and low priority `arbReq{7-4}` and `arbReq{3-0}` are each fed to a rotating priority 4-bit arbiter, which ensure fairness among requests within a group. The four modules are cascaded through an enable signal so that a request of lower priority can only win if there are no requests at a higher priority. The cascaded enable signal is gated with the `hld` signal generated by the hold logic, so that further arbitration is disabled during burst mode.

The arbitration results in a vector of 10 grant signals `grnts` (at most one asserted during any cycle) which are fed back to the corresponding input ports (`arbGrnts`). The GRANTS LOGIC block guarantees the corresponding bit of `arbGrnts` to be asserted for as long as the port has control of the bus (either through request or hold, see fig. 10.7 for exact timing of `in_grnt`). The `arbGrnts` vector is latched and buffered to serve as selector for the bus switch point settings (`SelBus`) on the next cycle. The `out_dav` is generated as a simple OR function of the `ltchGrnts`vector, so that the receiver is informed of the presence of data whenever the bus is granted to one of the input ports.

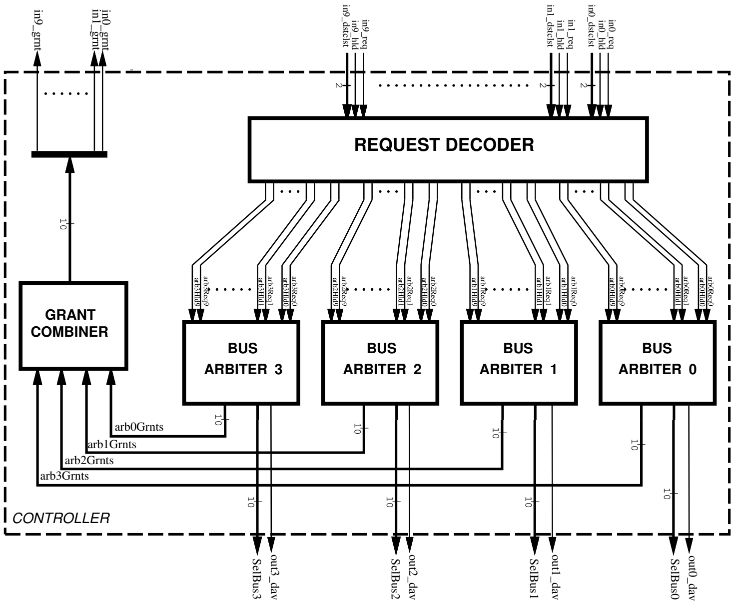


Figure 10.16: *Internal structure of the control module*

The rotating priority arbiter

The rotating arbiter is arranged around a reloadable shift register and a slightly modified fix priority arbiter as shown in fig. 10.18. The incoming requests are gated withe the enable signal coming from the higher priority tier. The resulting signal is then fed to an arbiter constructed on a rotating token. The token insertion point determines the request with the highest priority for that arbitration. The one hot vector of insertion tokens is taken from the output of a register whose are the grant vectors (of the previous cycle) shifted by one position. Thus, the request that has won arbitration on cycle "n" will have the lowest priority on cycle "n+1".

10.3.3 Expected size

The C-Switch size can be estimated by computing the number of wires required by the busses. Each bus is 89-bits wide (for a total of 356 wires) and the wire pitch is $1.8\mu m$, thus the total height of all busses together is about $640\mu m$. Each bus will run for almost the complete length of the chip (say 15mm). The control logic will be distributed and interleaved between busses. Assuming 3×20 wires worth of height dedicated to the control, will give a controller height of $36\mu m$. Total height is estimated to be $750\mu m$. Fig. 10.19 shows a layout of the C-Switch.

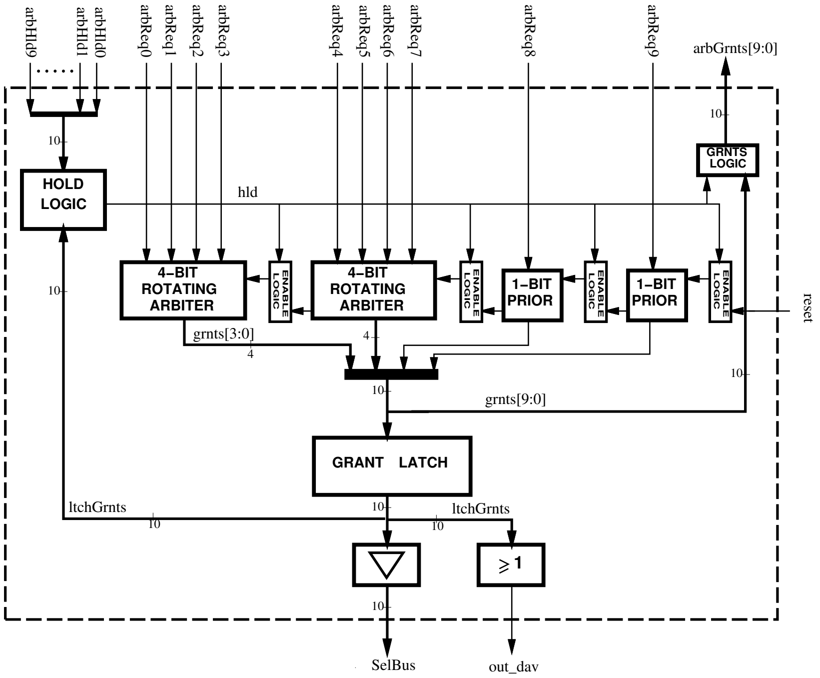


Figure 10.17: *Internal structure of a bus arbiter*

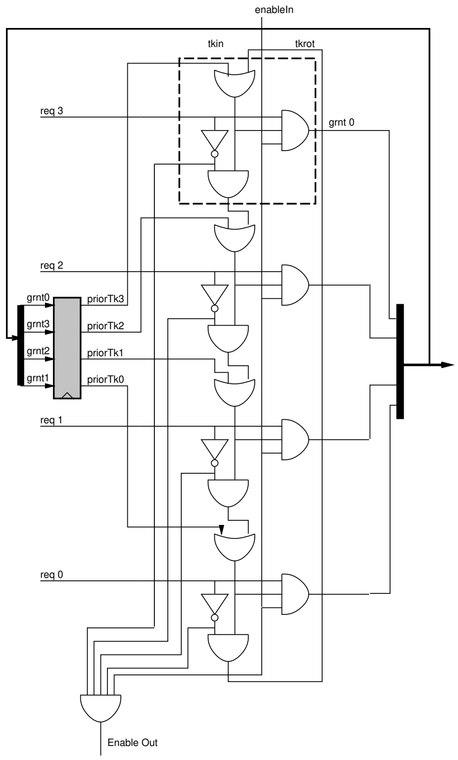


Figure 10.18: *Internal structure of a 4 bit rotating priority arbiter*

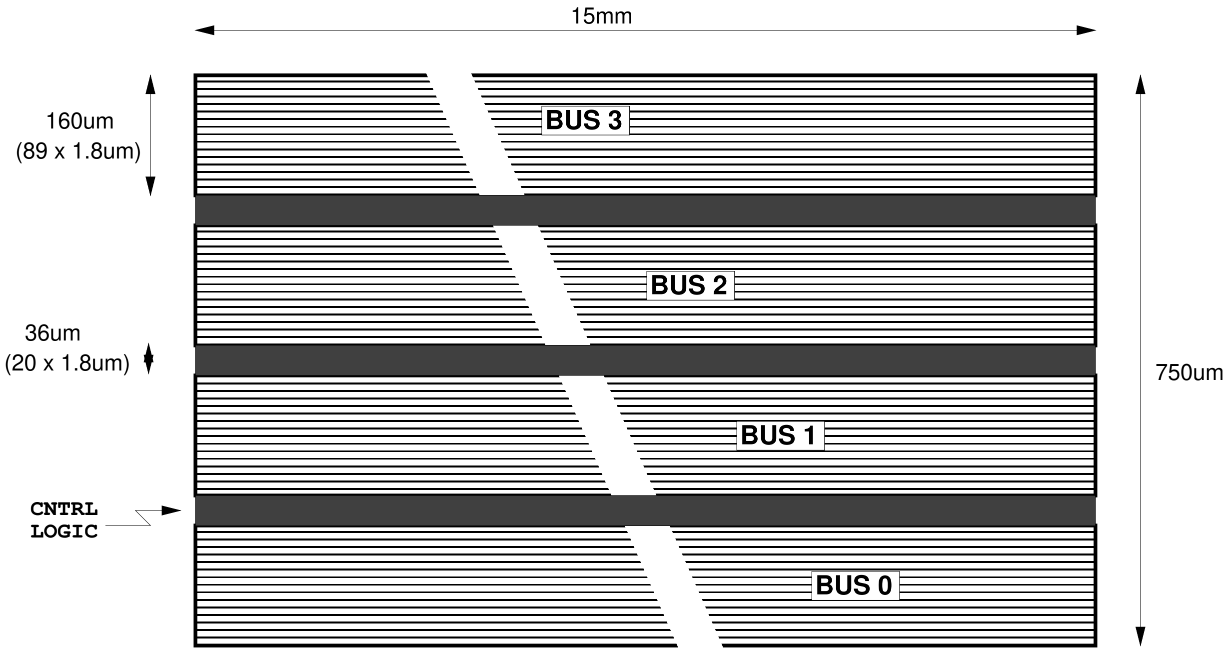


Figure 10.19: *layout of C-Switch*

Chapter 11

The M-switch

11.1 Functional Overview

The memory communication switch (M-switch) provides communication pathways from the clusters of the multi-ALU processor (MAP) to the configuration space controller and the memory system (cache banks). Fig. 11.1 shows the M-Switch in relation to the other major components of the MAP chip.

The M-switch has 4 regular input ports, 1 bypass input port, and 4 output ports. Each regular input ports is written to by the memory units (MEMU) in one clusters. The bypass input port is written to by the memory units (MEMU) in cluster 1 [1]. The 4 output ports, each with its own bus, are read by the 4 cache banks. In addition, output port #3 is shared by Cache Bank 3 and the Configuration Space controller (CFGCTRL). Up to four transfers can take place simultaneously in each clock cycle. The M-Switch design takes after the C-Switch, as shown in fig. 11.2.

11.1.1 Operation

An M-switch transfer takes place in the following cases:

- Any memory load/store operation.
- Any cache management operation (`FLNE, FCA, GETCSTAT, PUTCSTAT`)
- Any Configuration Space operation.

A Configuration Space operation is invoked by any load/store operation to a configuration-space-mapped address. A number of pseudo operations are also mutated into configuration space load/store operations by the MEMU: `HFORK`, `VGSU`, `VAC`, `VAC`, `HEXIT` and `VEXIT`.

[1] The regular and bypass ports in cluster 1 can function independently, such that the M-switch may accept a request from each of the two ports simultaneously.

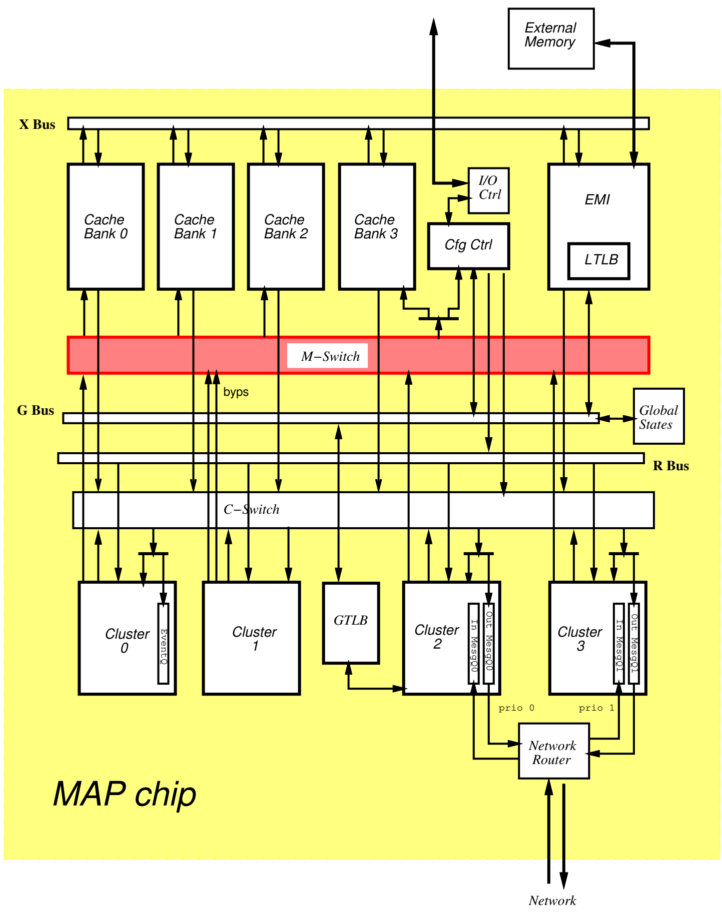


Figure 11.1: *The M-switch within the MAP chip*

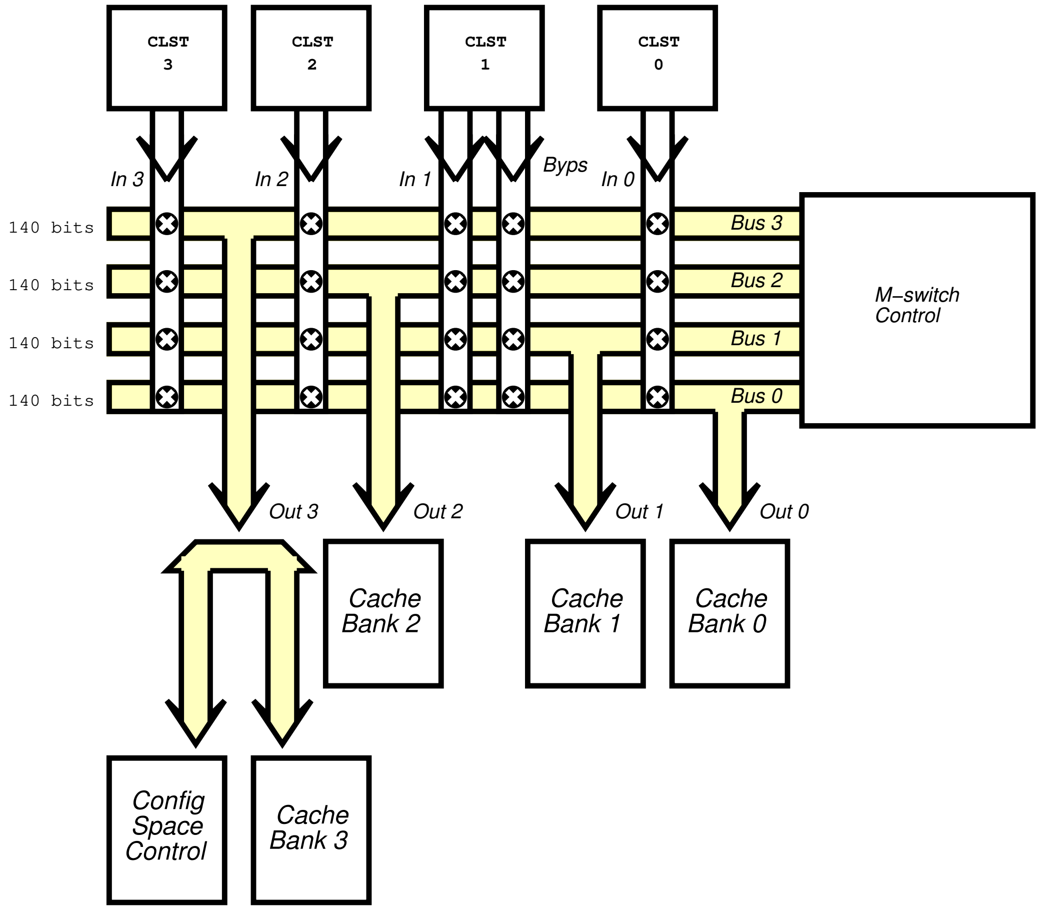


Figure 11.2: *High level view of the M-switch*

Each transaction over the M-Switch takes one cycle. The writers request/arbitrate for the use of the M-Switch busses in the first half of the cycle, and the transfer takes place in the second half of the cycle.

11.1.2 Memory Request Ordering

The M-Switch guarantees that all requests from the regular input ports to the *same* memory address are completed in the order the requests reach the M-Switch. This guarantee applies to regular memory references as well as configuration space references. However, *no* ordering is enforced between bypass port requests and regular port requests.

From the programmer's view, this ordering means that all *non-bypassed* configuration-space/memory operations to the *same memory address* will be completed in the order they are issued. Bypassed memory operations may be issued in *cluster 1* only, and are therefore inherently serialized amongst themselves.

11.1.3 Queueing Arbitration

The M-switch enforces queueing arbitration amongst all its regular port writers with an aging arbiter. Since some of the readers (cache banks/configuration space controller) may be busy and cannot accept a request at some time, the oldest queued request may not be fulfillable. The oldest *fulfillable* request then wins arbitration. *All* unsuccessful requests however preserve their relative ages on every cycle, regardless of the busy-ness of the resource requested.

The *bypass* input port is identical to other input ports, except that:

- The request is always given highest priority. It is not aged, and the `newreq` signal is ignored. (see next section).
- it has priority over all other ports, if the requested resource is available.

All regular requests however retain their relative ages when they yield to a bypass request.

11.1.4 Sharing of output ports

Output port # 3 of the M-Switch is shared by Cache Bank # 3 and the Configuration Space controller (CFGCTRL). Each of these resources have its own handshake signals with the M-Switch controller. This allows Cache Bank #3 and the CFGCTRL to be arbitrated for independently. A busy Cache Bank #3 will not prevent the use of the CFGCTRL if the latter is ready, and vice versa.

11.2 Interfacing to the M-switch

All input ports of the M-switch are identical, as are output ports # 0 through # 2. Output port # 3 is shared as described above.

11.2.1 Structure of the input ports

Each input port of the M-switch expects the following signals from the MEMU in the cluster:

Request					*Transfer*			*Address*		*Return*					*Data*	
req	newreq	reqcfg	bank	grnt	op	pre	post	phy	addr	rtnclst	tslot	rf	reg	rtncc	sync	data
1	1	1	2	1	7	2	1	1	51	2	3	1	4	3	1	65 bits
MSB																LSB

All signals are inputs from the MEMU to the M-switch except the `grnt` signal which is a handshake signal going from the M-switch back to the cluster. The signal groupt `Return` is identical to the signals known as `Destination` and `Condition` in the *c-switch* document/chapter. The signals have the following meaning (the naming convention "`in_signalname`" is used to identify the signal as belonging to the *group* "`in`". The signals for the 4 regular ports will be prefixed by `in`i_" for $i = 0to3$, while the bypass port signals will be prefixed by "`byps_`"):

- *Request*
  - `in_req` : request for a M-switch access. This must be asserted for all M-Switch requests. If only `req` is asserted, the request arbitrates for a normal cache bank access.
  - `in_newreq` : This line must be asserted during the first cycle of a new request. This signal is used to ensure that a requester's age is properly reset. This signal does *not* exist in the *bypass* input port.
  - `in_reqcfg`: request for a M-switch access to CFGCTRL.
  - `in_bank`: specifies the destination cache bank. This is ignored if `reqcfg` is asserted.
  - `grnt`: this is the grant signal to indicate that the port has won arbitration for the bus and data at the input port will be routed in the second half of the cycle.
- *Transfer*
  - `in_op`: This field specifies the operation to take place in the destination. It is devided into the following subfields

MSB LSB

type	r/$\overline{\text{w}}$	sz	ft	cdnl	$\overline{\text{cach}}$	byps
if *type* = "1"	read/$\overline{\text{write}}$ 1 bit	synchronize 1 bit	faulting 1 bit	conditional 1 bit	$\overline{\text{caching}}$ 1 bit	bypass 1 bit

if *type* = "0"	000000	*Abort Transaction*
	001000	FLNE
	001001	FCA
	00101x	PUTCSTAT
	10101x	GETCSTAT
	0100xx	SMS
	1100xx	LMS

The *Abort Transaction* code is used for a late cancellation of a transfer as explained in sec. 11.2.1.

  - `in_pre`: pre-condition of the synchronization bit.
  - `in_post`: post-condition of the synchronization bit.

- *Address*
  - `in_phy`: Physical address. This bit is set if `in_addr` is a physical address.
  - `in_addr`: The address of the memory *word* location to access.

- *Return*

  These fields are used to specified the return destination of the transaction. They correspond to the *Destination* fields expected by the C-Switch input port. Refer to the *c-switch* chapter/document for more details.

  - `in_rtnslot`: The cluster to return the data resulting from this transaction.
  - `in_tslot` : V-thread slot identifier within a cluster.
  - `in_rf`: register file identifier.
  - `in_reg`: register identifier field indicates the destination for the resulting data.
  - `in_rtncc`: The condition register for the returning result.

- *Data*
  - `in_sync`: memory synchronization bit.
  - `in_data`: 65 bits of data including the pointer bit.

The M-Switch arbitration is concerned with the *Request* fields only. All other fields are treated as simple data, and fed directly to the cache bank/CFGCTRL.

Sequence of handshake signals

The timing sequence of the handshake signals are specified in this chapter. There are a few requirements:

- The *Request* signals must become stable by the first quarter of the cycle (positive edge). These signals must be held stable until the following rising edge of the clock.
- The rest of the input signals should be stable by the falling edge of the clock, and be held stable until the end of the cycle.
- The MEMU may latch the `grnt` line at the rising edge after the *Request* lines have stablized. If `grnt` were asserted, it indicates that the transaction had completed, and the `Request` must be removed within the next first clock quarter.

Fig. 11.3 shows a timing diagram for one requester (regular input port) with two successive requests. The first request is successful, and the second request is stalled for one cycle waiting for the requested resource to become ready. If the requester had instead wanted to cancel the second request after it found out it was stalled, it must stabilize the *Request* lines by the first quarter of cycle 3.

Fig. 11.4 shows a timing diagram for two writers (regular input ports) requesting for the same output port of the M-switch. Note that after requester 1 lost the arbitration during the first cycle, it must hold the *Request* lines asserted during the next cycle in order to preserve its position in the aging arbiter. If it had dropped and re-asserted the *Request* lines at the rising clock edge after it lost arbitration, it would have been counted as a new request by the aging arbiter.

Fig. 11.5 shows bypass port requests overriding the requests from two regular ports. Notice that the regular port requests retain their relative ages while being displaced by the bypass requests. Also note that the bypass port does not use the `newreq` signal.

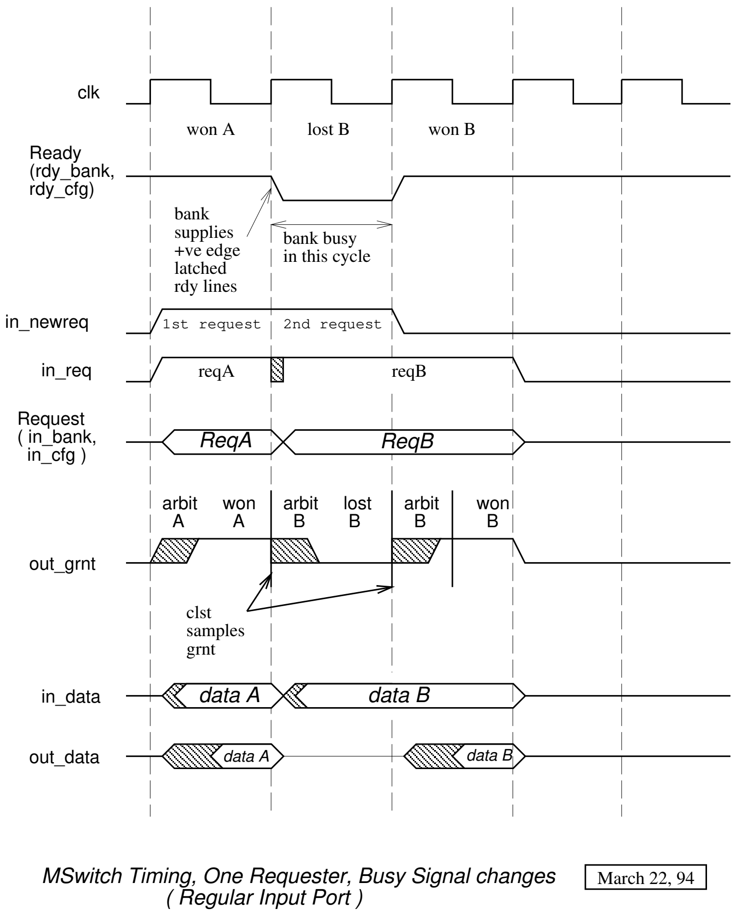


Figure 11.3: *Timing diagram for a single requester, successive requests*

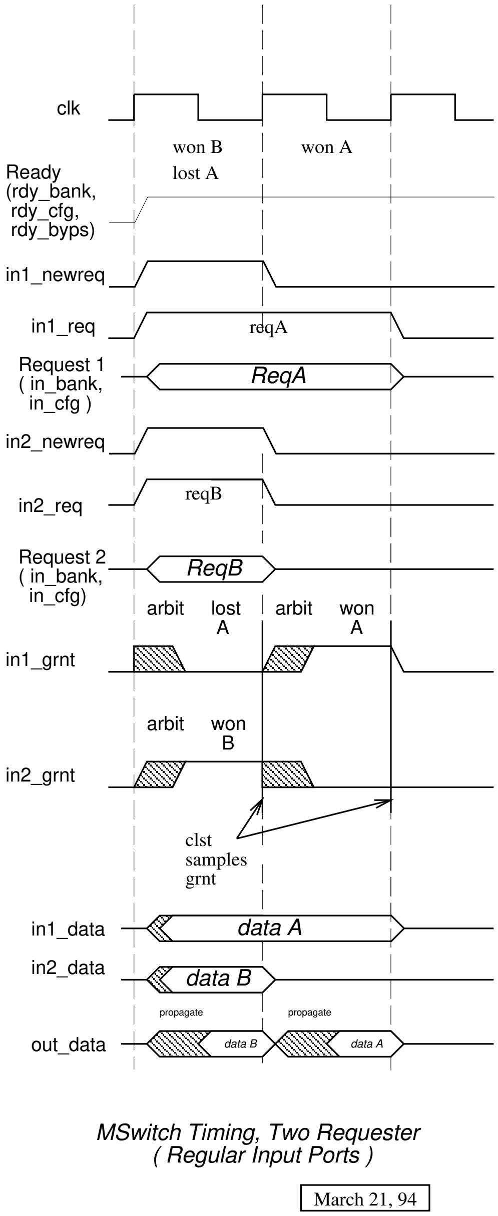


Figure 11.4: *Timing diagram for two requesters arbitrating for an output port*

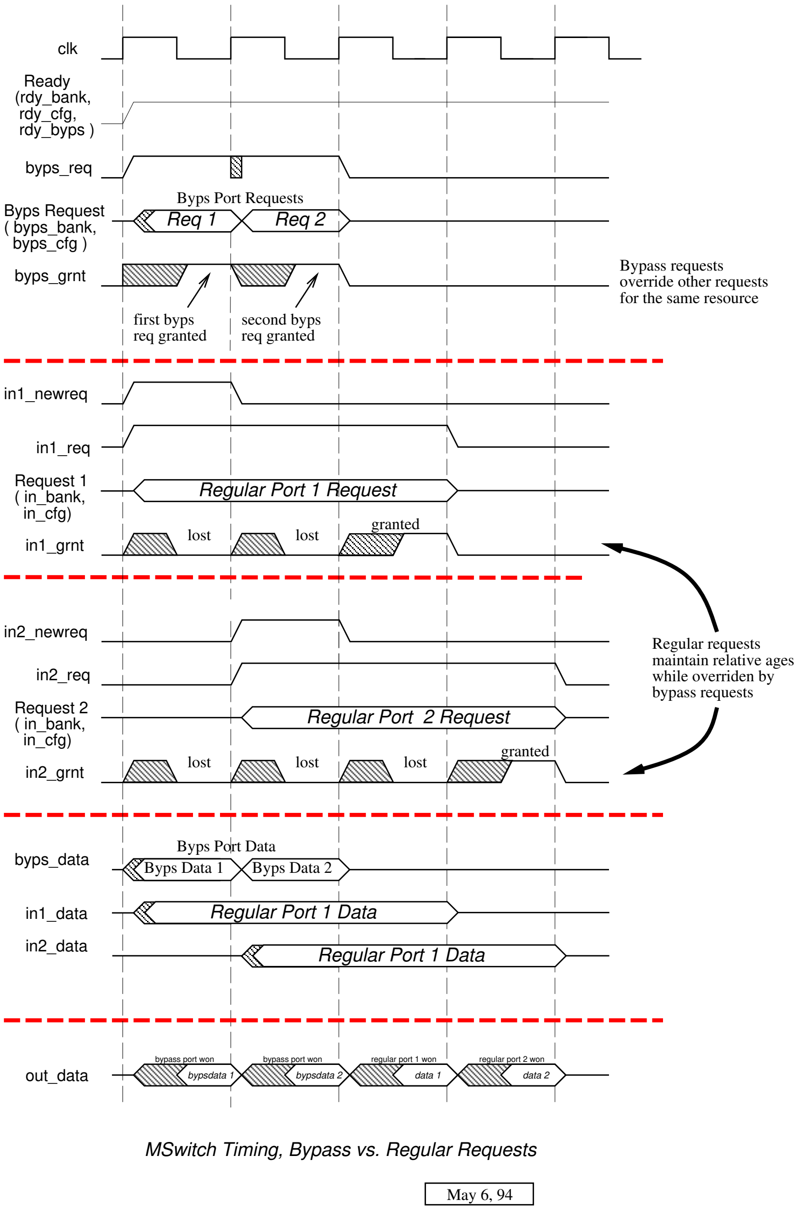


Figure 11.5: *Bypass requests displacing requests from regular input ports*

Aborting M-switch requests

Because the M-Switch must complete a transaction within one cycle, a request must be stably deasserted by the first clock quarter if it is to be aborted. Otherwise, the only way to abort a granted request is by setting the `op` field to signify *Abort Transaction*. The bus cannot be be reassigned to a different requester until the following cycle. It is the responsibility of the cache bank/CFGCTRL to interprete `op` appropriately.

11.2.2 Structure of the output ports

The output ports of the M-Switch is made up of the following signals:

Data Available			*Ready*			*Transfer*			*Address*		*Return*					*Data*	
dav	davcfg	davbyps	bank	cfg	byps	op	pre	post	phy	addr	rtnclst	tslot	rf	reg	rtncc	sync	data
1	1	1	1	1	1	7	2	1	1	51	2	3	1	4	3	1	65 bits
MSB																	LSB

All signals except the *Ready* fields are outputs from the M-switch module. The *Ready* signals are M-switch inputs expected from the cache banks and CFGCTRL to indicate the busy/ready state of the corresponding resource. The *Data Available* lines are used to signal the validity of the data on the bus. Except for the *Data Available* and *Ready* fields, all the outputs are directly passed through from the M-switch input port.

The *Data Available* signals can be sampled by the reader at the rising edge of the clock, and are defined as the following:

- `out_dav`: Asserted when the data on the corresponding bus is valid.
- `out_davcfg`: Asserted when the data on the corresponding bus is meant to go to CFGCTRL. Otherwise the data is meant for the cache bank. This line is used only in output port # 3.
- `out_davbyps`: Asserted when the data on the corresponding bus is meant to take the bypass path.

The *Ready* signals are defined as the following:

- `rdy_bank`: Asserted when the cache bank is ready to accept data.
- `rdy_cfg`: Asserted when CFGCTRL is ready to accept data. This signal is used only in output port # 3.
- `rdy_byps`: Asserted when the bypass path of the module connected to this output port is ready to accept data.

The M-switch expects to get positive-edge latched *Ready* signals from the reader modules. The appropriate *Ready* signal must be de-asserted at the *beginning* of the cycle during which a transaction would be refused, and asserted at the *beginning* of a ready-to-accept-data cycle. Figure 11.6 shows the *Ready* signals in relation to the rest of the signals.

11.3 Internal architecture

The internal structure of the M-switch, shown in fig. 11.7, is similar to the C-switch, and leverages off some modules already designed for the latter.

At the input ports, the *Request* signals go to the M-switch controller module, while all other signals are fed through the data busses to the output port. The controller is responsible for arbitration and management of the input/output port handshake signals, while the crossbar provides the data path.

11.3.1 The M-switch data path

The M-switch bus module is structurally identical to the C-switch data bus. As shown in Fig. 11.8, there are 4 M-switch busses, each 140-bits wide.

Each M-switch bus is fixed to one of the output ports, and connected to all the input ports via switching points (shown as a black circle with a white cross) . Each switching point is controlled by a one-hot bit within a set of 4-bit control busses. The control busses are ***not*** latched, but are guaranteed to be one hot at any time.

11.3.2 The M-switch controller

The M-switch controller is made up of a request decoder, 4 independent queuing arbiters, and a grant combiner, as shown in Fig. 11.9. Each arbiter controls one M-switch bus.

The M-switch controller functions similarly to the C-switch controller:

- Request decoder: routes the *Request* lines to the appropriate queuing arbiter according to the values of the `in_reqcfg/byps_reqcfg` and `in_bank/byps_bank` fields. (`in_bank/byps_bank` is ignored when `in_reqcfg/byps_reqcfg` is asserted.)
- Queuing arbiter: Each arbiter keeps track of the *age* of all its input requests, and grants its associated bus to the *oldest fulfillable* request by asserting the appropriate `arbGrnts[]`, `out_dav/out_davcfg`, and the control bus lines. A request from the bypass port however overrides all other requests.
- Grant combiner: performs bit OR function of the four `arb{0-4}Grnts` buses to combine the corresponding signals from each arbiter into one `grnt` per input port. It implements

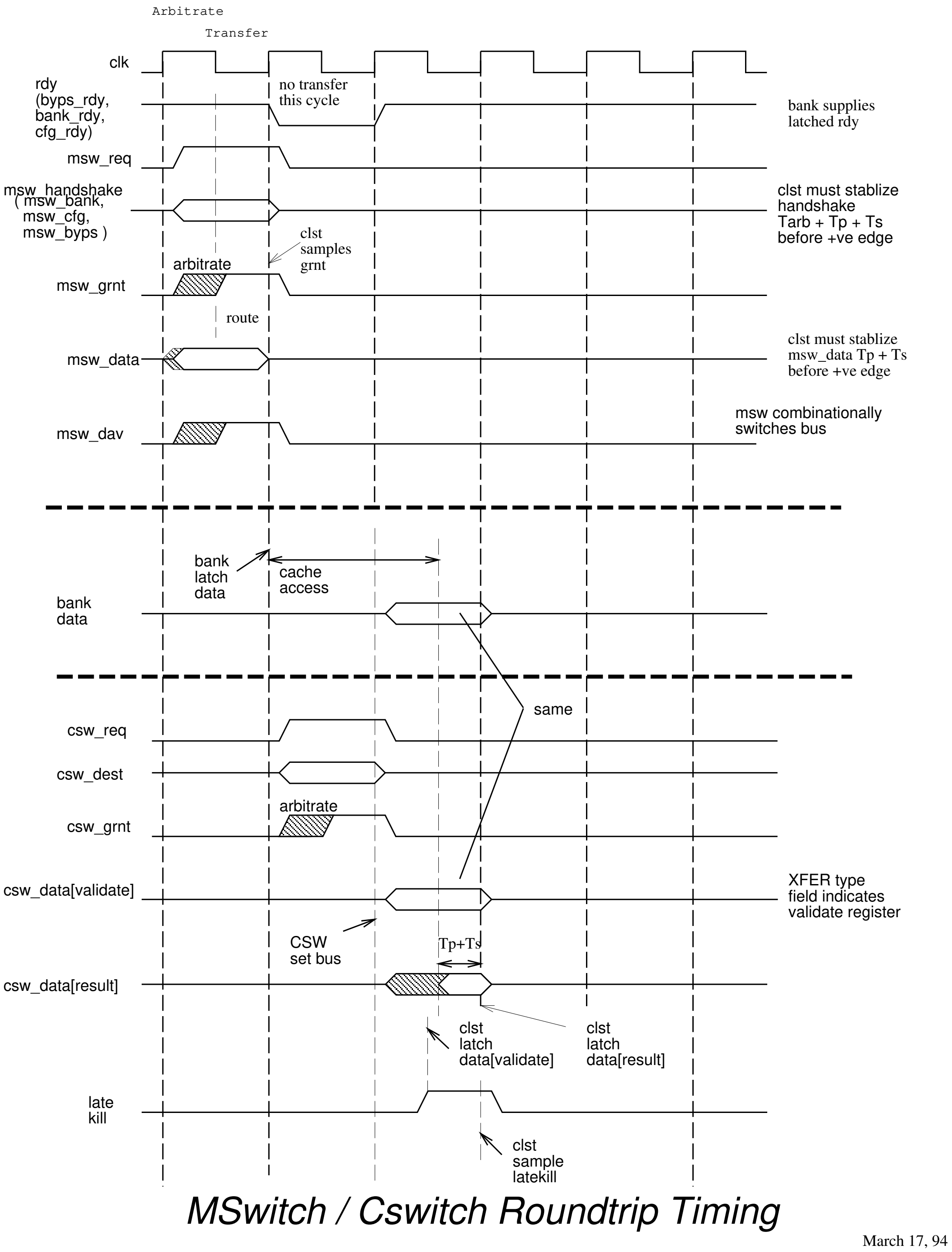


Figure 11.6: *M-switch timing in relation to the C-switch*

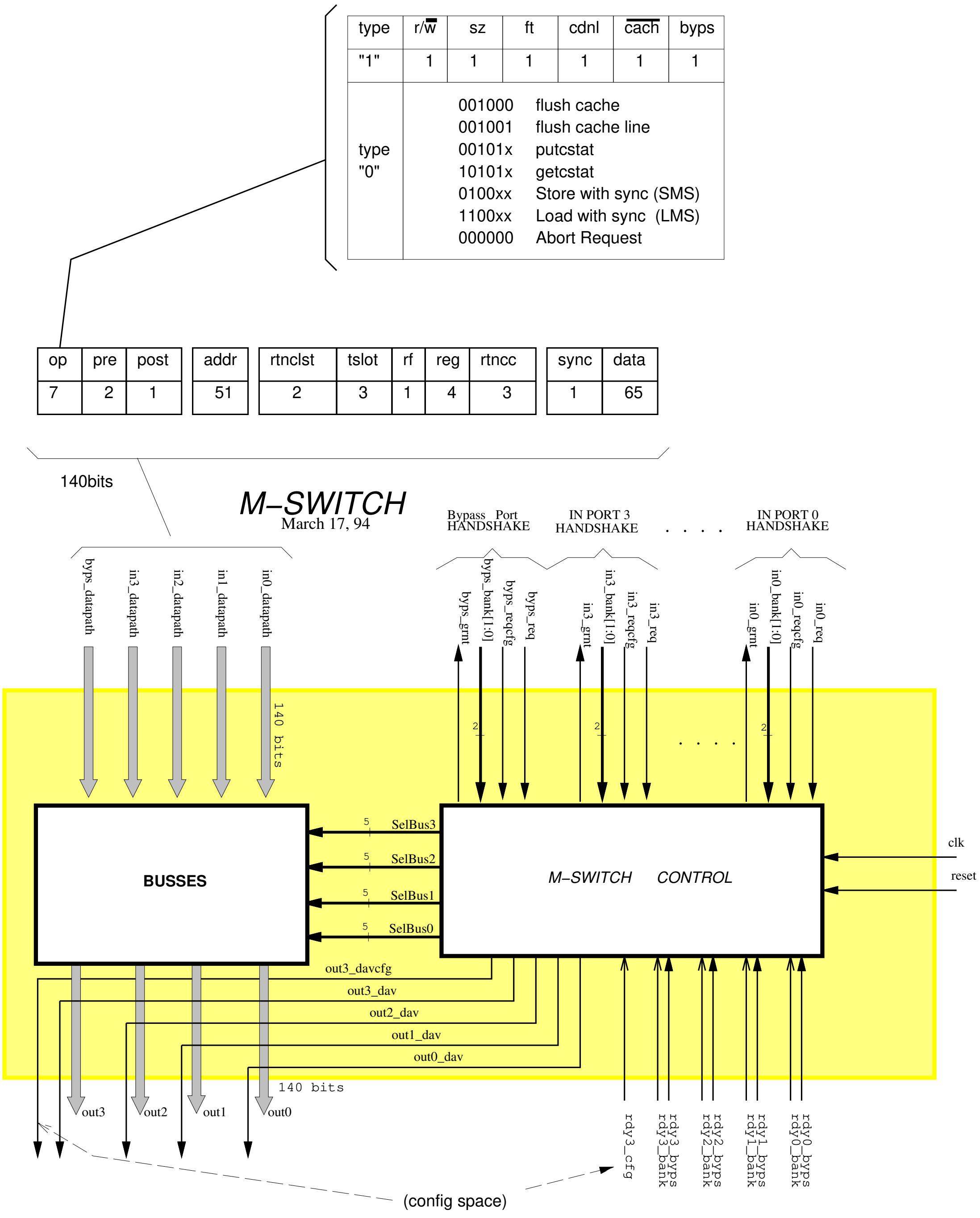


Figure 11.7: *Internal structure of the M-switch*

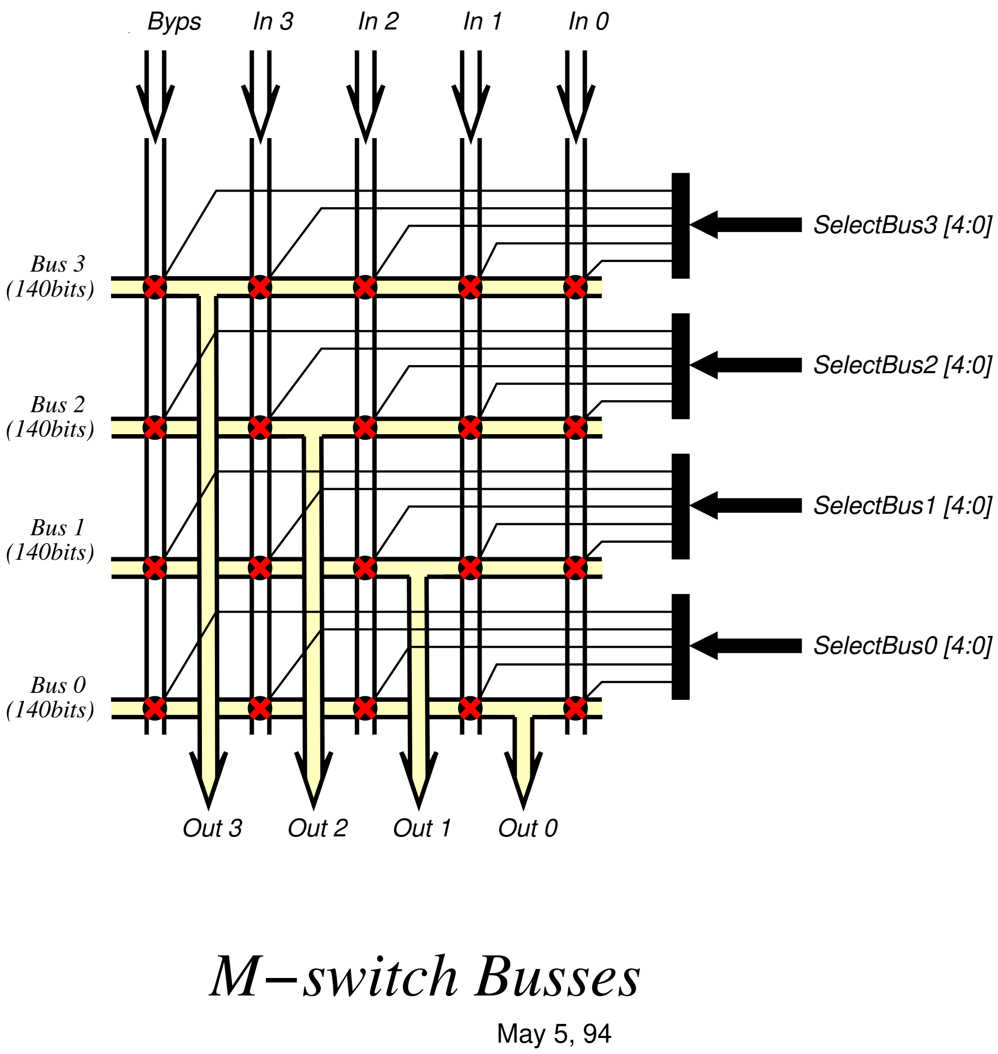


Figure 11.8: *Internal structure of the crossbar module*

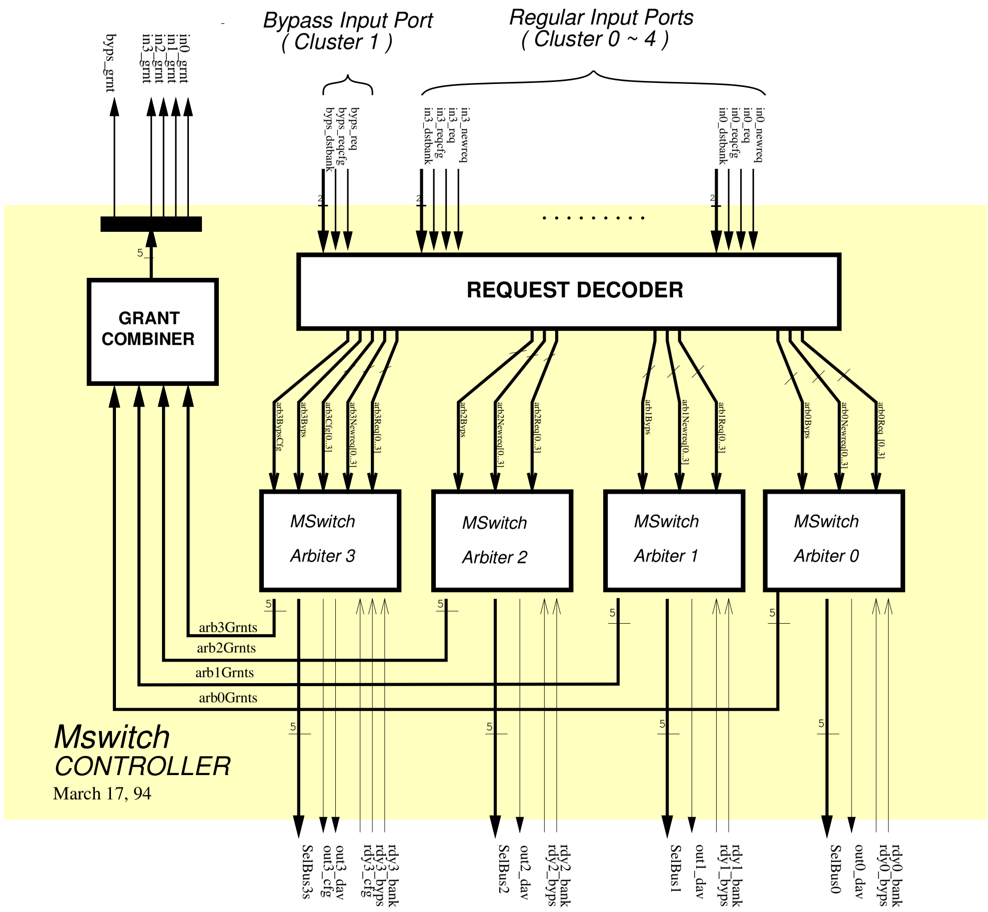


Figure 11.9: *Internal structure of the control module*

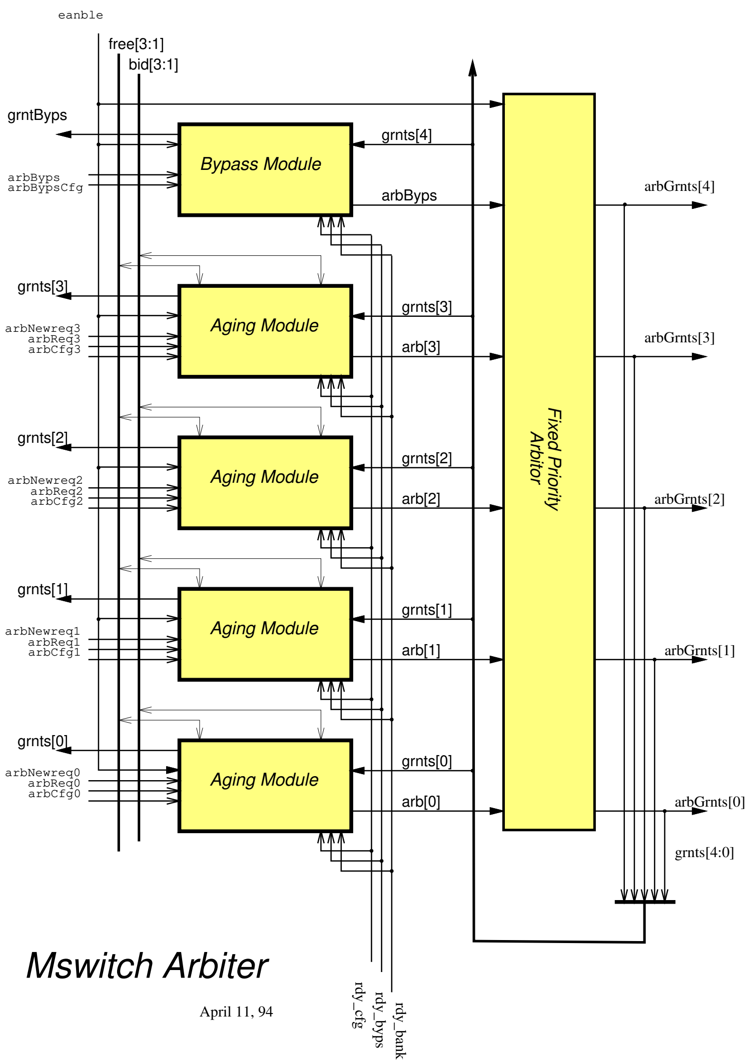


Figure 11.10: *M-switch queueing arbiter*

the following function `in`j`_grnt = arb0Grnts[`j`] | arb1Grnts[`j`] | arb2Grnts[`j`] | arb3Grnts[`j`]`, $\forall j : 0 \leq j \leq 3$, and `byps_grnt = arb0GrntsByps | arb1GrntsByps] | arb2GrntsByps | arb3GrntsByps`.

11.3.3 The Queueing arbiter

The queueing arbiter is made up of 4 aging modules, a bypass module, and a fixed priority arbiter, as shown in fig. 11.10.

The aging modules enforce a fifo order amongst the regular port requests. Conceptually, at the end of each cycle, the age of each unfulfilled request is incremented, up to age 3, subject to the restriction that no request will ever overtake another in age. Th implementation is

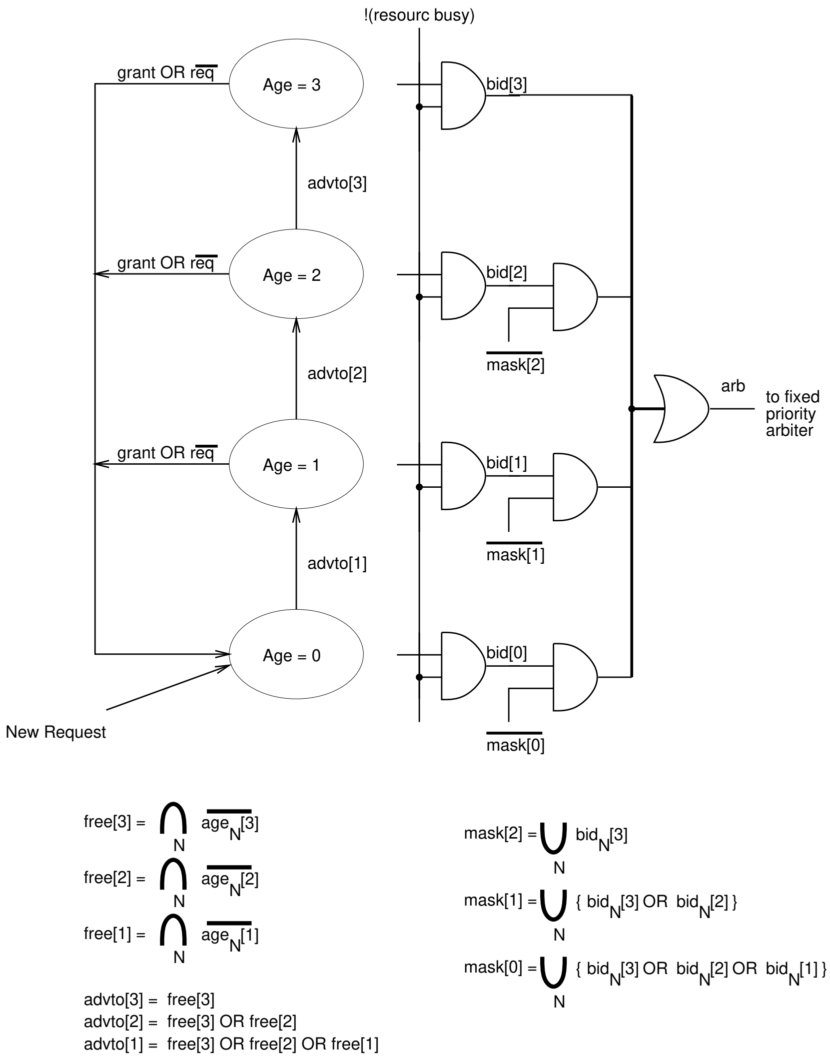


Figure 11.11: *Request Aging*

shown in figure 11.11.

The internal organization of the aging module is shown in figure 11.12. Note that a request's age is reset to zero when the request is granted or when the corresponding *in_req* line is de-asserted. Therefore, a requester that failed to be granted access to the resource must hold its *in_req* line stable if it wants to preserve its priority over other newer requests.

The box marked *Busy?* in fig. 11.12 represents combination logic that asserts `req{0-3}Valid` only when the requested resource is not busy:

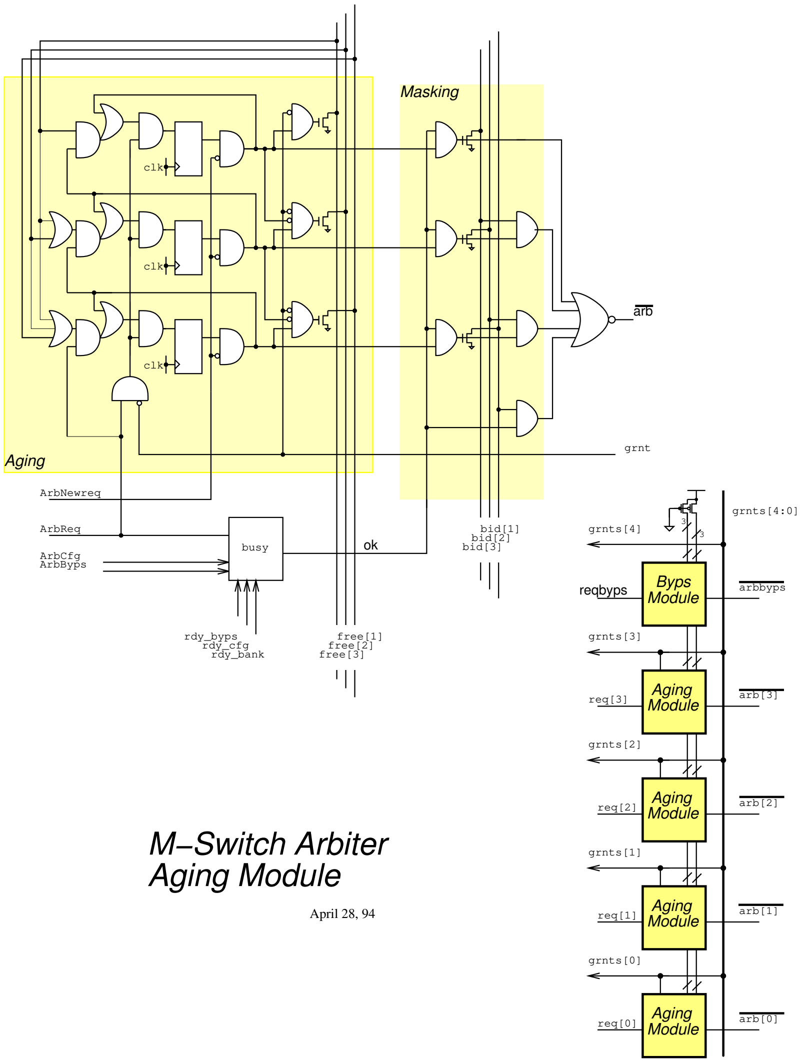


Figure 11.12: *M-switch Aging Module*

Inputs					Output
`req`	`reqcfg`	`rdy_bank`	`rdy_cfg`	`rdy_byps`	`reqValid`
No Request					
0	x	x	x	x	0
Cache Bank Access					
1	0	0	x	x	0
1	0	1	x	x	1
Configuration Space Access					
1	1	x	0	x	0
1	1	x	1	x	1

The bypass module enforces the following logic:

Inputs					Output
`byps_req`	`byps_cfg`	`rdy_bank`	`rdy_cfg`	`rdy_byps`	`bypsValid`
No Request					
0	x	x	x	x	0
Cache Bank Access					
1	0	x	x	0	0
1	0	x	x	1	1
Configuration Space Access					
1	1	x	0	x	0
1	1	x	1	x	1

The output of each aging module is a one bit `arb` signal. At this point, a fixed priority arbitration scheme is used to select between `arb[4..0]`, with preference given to requests with higher port IDs. The `arb[4]` signal is an alias for `arbByps`. The fixed priority arbiter is shown in Fig. 11.13.

Fixed PriorityArbiter

April 26, 94

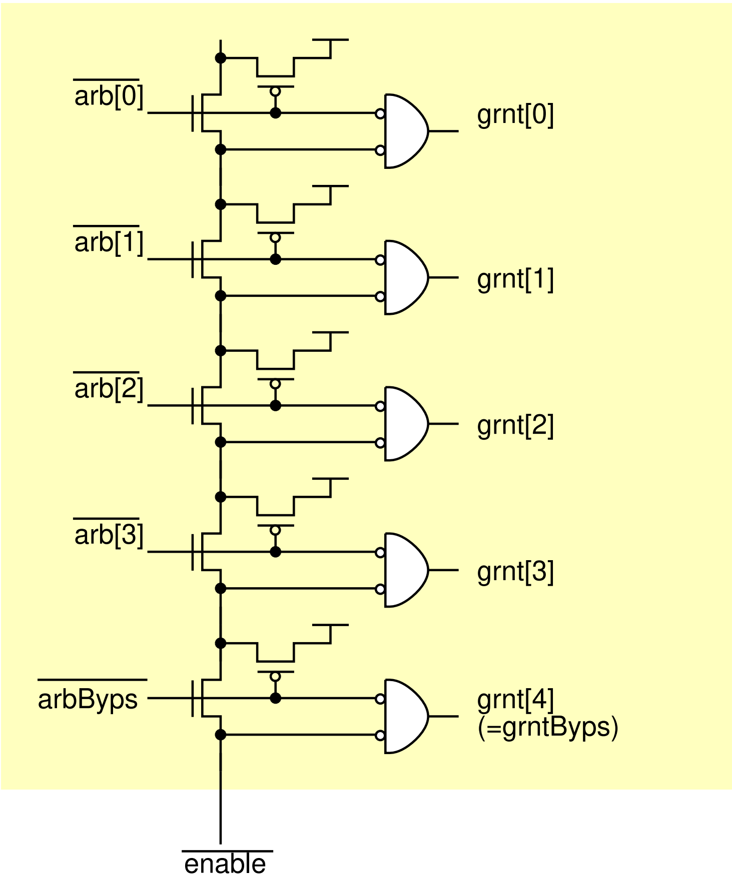


Figure 11.13: *Fixed Priority Arbiter*

Chapter 12

The Cache Banks

12.1 Overview

The cache banks of the MAP store both data and instructions in a combined cache of 32 KB per bank, times 4 banks per MAP, for a total storage of 128 KB. The cache banks receive requests from the clusters over the M-Switch, and route results from accesses that hit in the cache back to the clusters over the C-Switch. When a cache miss occurs, the bank forwards the request to the external memory interface (EMI) over the X-Bus. The cache is two-way set-associative, and is sectored, with each cache tag applying to two blocks[1] in the cache. Cache blocks are 8 words (64bytes) long, so there are 2048 blocks in the cache, and 1024 cache tags.

The cache banks are pipelined, to allow a new memory reference to be started on each cycle. Two sets of pipeline registers exist within the cache banks, one to handle normal memory references, and the other to handle bypassing memory references that allow the system to bypass the normal backpressure paths in order to handle TLB misses.

In addition to the cache itself, two block buffers are implemented. These buffers store cache blocks that have been evicted from the cache and need to be written back to the main memory, and blocks that have been read by uncached memory operations.

The four cache banks implement pseudo-LRU replacement, in order to avoid the communication required to implement true LRU in a multi-bank cache. When a cache miss occurs, the bank that sustained the miss sends the request that caused the miss to the EMI, along with a bit which tells the EMI which of the two possible cache blocks should be replaced. This bit is generated by examining the LRU status of the appropriate set within the bank that sustained the miss. This should give performance that closely approximates truly LRU replacement, while keeping the decision of which block to replace local.

Figure 12.1 shows a block diagram of a cache bank. The block labelled *CNTL* contains the

[1]Terminology: a sector (16 words) is the amount of data that a tag refers to. A block (8 words) is the amount of data fetched into the cache on a cache miss.

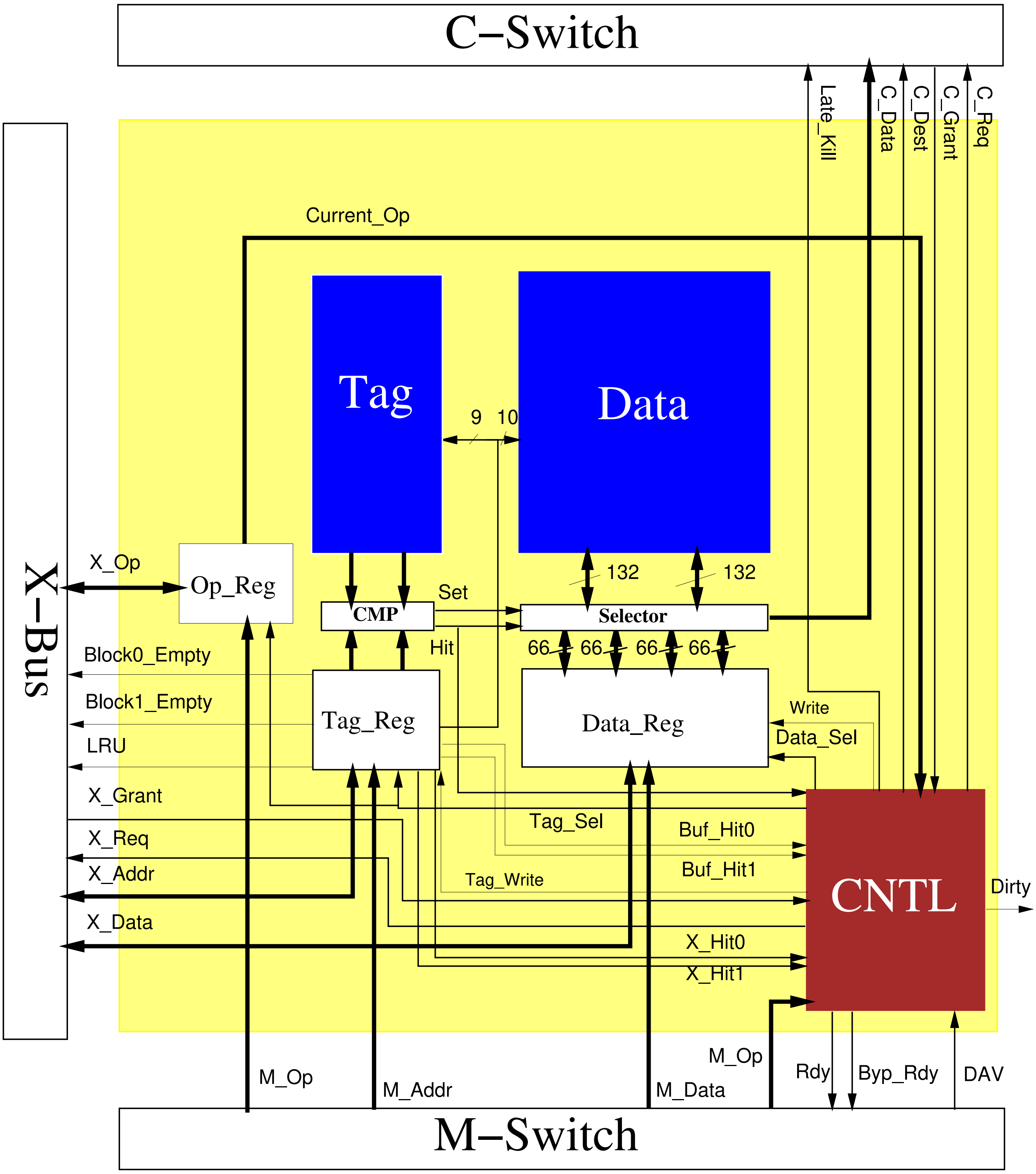


Figure 12.1: *Block Diagram of a Cache Bank*

various state machines that are used to control the cache bank, and the combinational logic that generates the control signals for the cache bank. The *Data* block contains the SRAM array that stores the cached data. The *Tag* block contains the SRAM arrays that store the cache tags and the LRU information for the data in the cache. The *Tag_Reg* and *Data_Reg* blocks contain the registers that store the requests that are submitted to the cache banks. The *Selector* block routes data between the data registers, the data array, and the C-Switch, while the *CMP* block compares the addresses produced by the tag array in response to a request to the address of that request to determine if a cache hit has occurred. Most of these modules are described in greater detail in the body of this document. The external interfaces of the cache bank are not described in this document, as the cache bank merely follows the interface specified in the design document for each of the modules that it communicates with.

12.2 Timing

Figure 12.2 shows the timing of a request that hits in the cache and sends a reply back to the originating cluster over the C-Switch. The sequence of events from the point of view of the cache bank for a load that hits in the cache is:

- Cycle 0: MEMU executes memory operation, arbitrates for M-Switch, routes request to bank
- Cycle 1: Bank receives request at rising edge of clock. Bank latches request into the appropriate buffer, passes some request bits into combinational logic to raise C_Req line in time to make C-Switch arbitration for next cycle if need to return result. Request sent to tag and data modules if no previous request waiting for module. Tag compare must complete in time for write pulse to be driven before the end of the cycle.
- Cycle 2 (first half): Bank must drive C-Switch inputs to tell sync stage to validate destination register at rising edge of clock, so that it gets driven across the C-Switch as soon as arbitration completes. Late kill and fault signals generated.
- Cycle 2 (second half): Data or late kill signal is driven over the C-Switch to the bank.
- Cycle 3: Data from the memory access is available; an instruction that uses this data as an input may issue.

Figure 12.3 shows the timing of the internal signals on a cache hit. If a cache miss occurs, the request is latched, and the cache arbitrates for the X-Bus on the cycle 3 of the memory access (the cycle during which data would have been available for use by the cluster if a cache hit had occurred).

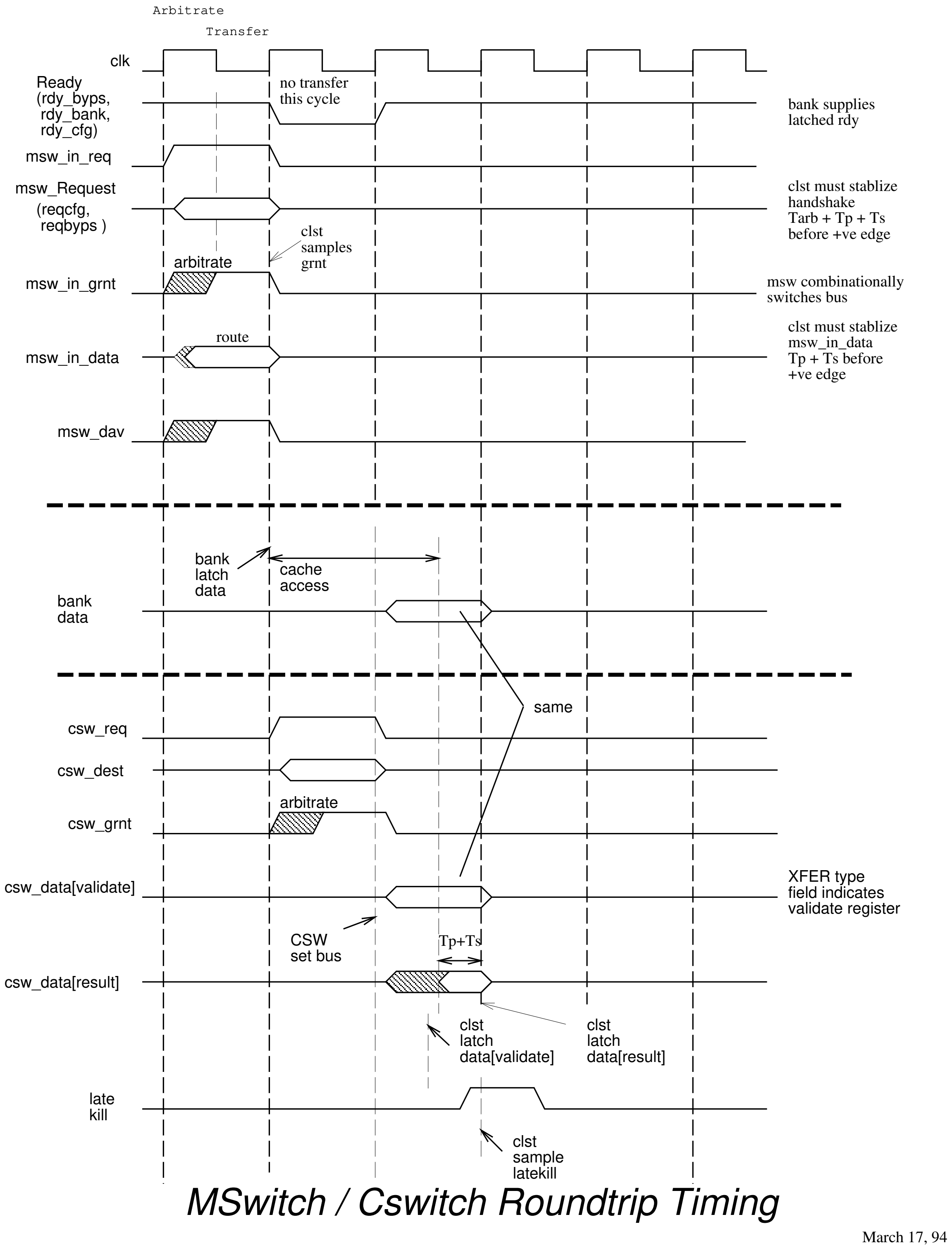


Figure 12.2: *Timing Diagram of a Memory Access*

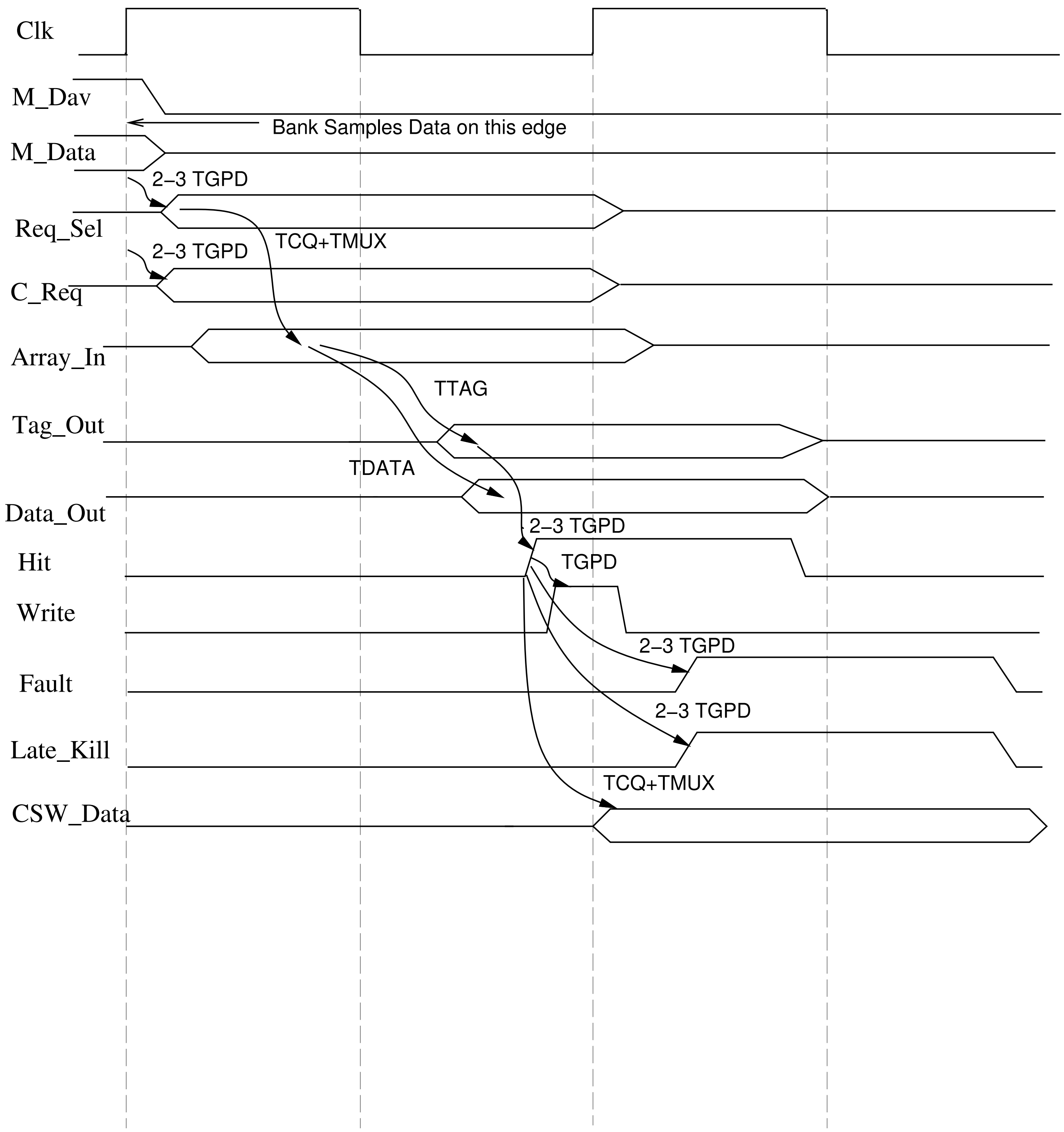


Figure 12.3: *Internal Signal Timing*

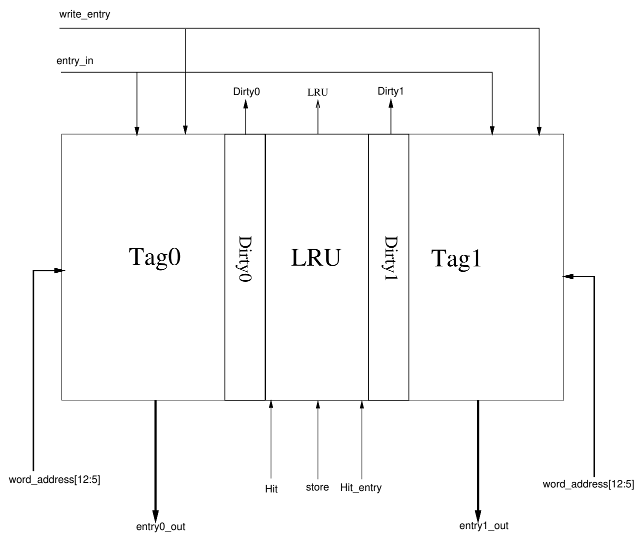


Figure 12.4: *Block Diagram of the Tag Array*

12.3 Tag Array

The tag array module consists of three SRAM arrays, two to hold the tags for the lines in the cache, and a third to hold the LRU bits for each set, as shown in Figure 12.4. When a cache access is performed, the nine set select bits of the address are used to index the tag arrays, which output the addresses of the cache lines that are contained in the set to the comparators. The comparators compare the contents of the selected tags with the rest of the bits of the address being accessed to determine if a hit has occurred. If a hit occurs, the LRU bits of the set are updated in the same cycle, as well as the dirty bit for the appropriate line, if a store operation is being performed.

When a new line is added to the cache or block buffers, the control logic asserts the Write signal, and places the address to be added to the cache on the X_Addr lines. The appropriate write strobe is then asserted, causing the address to be written into the appropriate place in the tag array. The control logic also sets the Hit and Set inputs to the LRU logic appropriately to indicate that the incoming block is the least recently used of the set.

The register file that is connected to the tag array contains the addresses of the four requests that can be waiting for the cache bank, the two requests that can be waiting for the X-bus, and the contents of the two block buffers, as shown in figure 12.5. When a request comes into the cache from the M-Switch, it is placed in either the Curr or the Curr_Byp buffer, depending on whether or not it is a bypassing request. Each cycle, the control logic selects a request from one of the Curr or Fail buffers and presents the request to the tag and data arrays. The Fail_Byp buffer has the highest priority, followed by the Curr_Byp buffer, and the Curr buffer. When a request from either the Curr or the Curr_Byp buffer is sent to the arrays, it is copied into the appropriate Fail buffer, in case a cache miss occurs, and the appropriate X_Req buffer is full. This allows a new request to be latched into the Curr

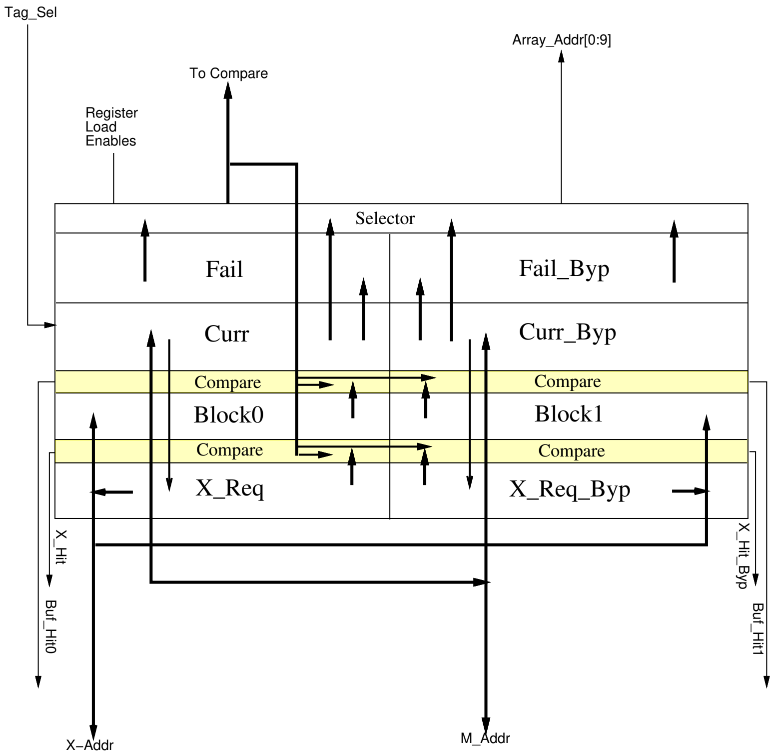


Figure 12.5: *Block Diagram of the Tag Array Register File*

buffer from the M-Switch on the cycle that the request which was previously contained in that buffer is being processed by the arrays without danger that the request will be lost if a cache miss occurs and the appropriate X_Req buffer is full.

The Block registers contain the tags for the data in the block buffers. Whenever a request is sent to the arrays, the block registers snoop on the address and assert their hit signals if the data that is needed by the request is contained in one of the block buffers. The control logic can then either route the data to the C-Switch or update the contents of the block buffers, as appropriate. Note that data is not copied from the block buffers to the cache array if a request hits in one of the block buffers.

The X_Req registers hold requests that have missed in the cache and are waiting for access to the X-Bus to be routed to the external memory interface. On a cache miss, the outputs of the tag register file selector are latched into the appropriate X_Req register, unless that register is full. If the register is full, the cache bank lowers its Ready output, to indicate that it does not have the buffer space to accept a new request, and keeps the request that could not be placed in an X_Req register in a Fail register.

When blocks of data are transferred from the EMI to the cache banks, the X_Req registers snoop on the addresses of the data being transferred, and perform their pending requests if a cache block comes in that satisfies the request in the register, allowing multiple cache misses to the same block to be satisfied by a single exetrnal memory access.

12.4 Data Array

Each cache bank contains a single data array, organized as a 560x512 grid of bit cells. Each 560-bit[2] row of the array contains the eight data words that make up a set, along with the pointer and tag bits for those words. Logically, each set in the cache is divided into two sectors, each of which has its own tag in the tag array. Each sector is divided into two blocks, where a block is the amount of data brought into the cache bank on a cache miss. This is two words per bank, for a total block size of eight words for the entire cache.

When a cache access is performed, nine bits of the address are used to select the correct row of the data array, which corresponds to the set that is eligible to contain the requested data. Each of the two sectors within this set is outputted to a 2-to-1 multiplexor, and a tenth bit of the address used to select the correct block within each of the two sectors. The two 132-bit blocks, one from each set, are sent to the selector block, where an eleventh bit of the address and the result of the tag compare are used to select a single word to be written or sent to the C-Switch.

The Data_Reg block contains registers which hold the data for all requests that the cache bank is in the process of handling, as well as the block buffers that hold sectors that are waiting to be written back to the external memory and blocks that have been fetched from the external memory to satisfy uncached memory operations. Figure 12.6 contains a block diagram of the data array register file. Note that the selector block is cabable of routing an entire 132-bit block from the data array to the data registers, allowing a sector to be copied from the data array into a block buffer in two cycles, allowing dirty blocks to be evicted from the cache during the latency between the initiation of a request to the external memory and the arrival of the first data word from the external memory.

12.5 Miss Handling

If a cache miss occurs (all tag compares fail), the request that caused the miss is latched into either the *X_out* or *X_out_byp* register at the start of cycle 3 of the memory access, depending on whether the operation was a normal or bypassing operation. If the needed buffer already contains a request that is waiting for the X-Bus, the request that missed is copied into either the Fail_buffer or the Fail_buffer_byp to be retried later. If both a Curr buffer and the associated Fail_Buf buffer are full (a request was copied to the Fail_Buf and another arrived over the M-Switch in the same cycle), the bank deasserts the appropriate ready signal to the M-Switch to indicate that it is incapable of handling any more requests of that type until the buffers have emptied.

[2]Need to check the MCNC specs to figure out why they want this many bits. My count says 528.

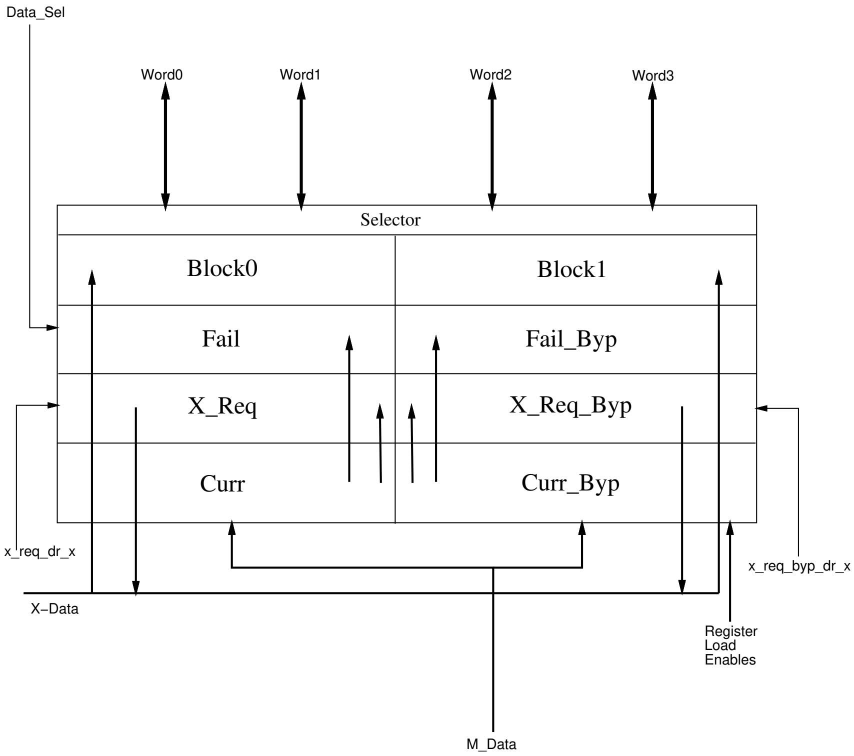


Figure 12.6: *Block Diagram of the Data Array Register File*

If one or more of the X_out buffers are full at the start of a cycle, the bank asserts the X_Req signal to indicate that it wants access to the X-Bus. The X_Byp signal is asserted if the request is bypassing, and bypassing requests are given priority over non-bypassing requests by the bank and the X-Bus. At the same time as the X_Req signal is asserted, the bank asserts the request on the X_Data lines, which will be routed to the EMI if the bank wins arbitration. At the start of each following cycle, the bank samples the X_Grant signal to determine if the request has been routed to the EMI.

When the EMI receives a memory request from a cache bank, it broadcasts a command to the cache banks, instructing them to evict the appropriate block, as spcecfied in the request, and telling them which block buffer to copy the sector containing the block into if it is dirty. If both of the block buffers contain dirty data, the EMI precedes this command with a command instructing the banks to write the contents of the LRU block buffer back to the external memory. The process of copying the previous contents of the sector into the block buffer is assumed to complete before the first data word returns from the external memory. If a block is being evicted from the cache because a flush instruction has been executed, it is not copied into a block buffer if it is dirty. Instead, the block is written directly back to the main memory, to ensure that no cached copies of the line exist.

In parallel with this, the EMI initiates an access to the external memory to retrieve the required data. When the data begins to come back to the EMI from the external memory, the EMI sends a command to the banks, informing them that it is starting to send the data back to them over the X-Bus, and indicating which bank will receive the first word[3]. FSMs in the EMI and the cache banks then co-ordinate the process of writing the block back into the cache one word at a time, ensuring that the ready signals from the cache banks to the M-Switch are set properly on cycles when the bank is receiving a datum from the EMI, and therefore cannot handle requests from the clusters. Uncached operations do not result in the eviction of a sector from the main cache, and a different command is sent from the EMI to indicate that the data being sent should be copied into one of the block buffers, not the main cache. The FSMs in the cache banks mark the incoming block valid as soon as the entire block has arrived in the cache.

12.5.1 Writebacks

Whenever a cache miss occurs, a sector that is currently in the cache is selected for replacement by the EMI. Once this has been done, the status bits for the sector being replaced are examined, to determine if the sector is dirty, and therefore needs to be written back to the external memory. Each bank examines its own status bits for the sector being evicted and asserts this signal on a single, wired-or, line that connects all of the banks. If any of the banks have written to the sector being evicted, the entire sector must be copied into one of the line buffers.

In order to reduce the amount of communication between the banks and the EMI, each bank

[3] This allows implementation of needed-word-first cache miss handling, which may be easy, given the burst mode of the sync DRAMs

provides the EMI with a dirty signal for each of its block buffers, and a LRU bit, indicating which of the block buffers was least recently used. The EMI examines these bits in order to select a block buffer for writeback, and ensures that an empty block buffer is available before evicting a sector. While this can cause block buffers to be written back to the external memory when the sector being evicted is not dirty, the data in the block buffers is not overwritten until a block is copied into the buffer, so this policy will usually just change the time at which a block buffer is written back. The only case where this policy will cause extra writebacks is when the data in a block buffer is modified between when the block buffer is written back to the external memory and when new data is written into the buffer, which should be rare. Also, if the EMI is not busy on a given cycle, it will initiate writebacks from dirty block buffers, in order to increase the chance that an empty block buffer is available when needed.

Data is copied between the cache banks and the block buffers by routing the output from the cache bank to the input of the block buffer and asserting the write strobe for the appropriate word of the block buffer. As each sector in a cache bank contains 4 words (2 words from each of the two blocks in a sector), an entire sector can be copied into a block buffer in four cycles, by doing one copy/cycle in each bank, although all four banks are incapable of handling other requests during this time.

12.6 Uncached Operations

Uncached memory operations are handled by the cache arrays in the same manner as other memory operations. The cache and block buffer tags are searched to determine if the desired data is in the cache, and the operation performed normally if a cache hit results. If a cache miss occurs, a block is not evicted from the main cache. Instead, the desired block is copied into one of the block buffers, and can be accessed from there. This allows operations like block copy to avoid having to go out to main memory for each word that they need to reference without filling the main cache with data that will only be used once.

12.7 Events

The only events that are detected in the cache banks are synchronization misses and block status misses. When an event occurs, the cache bank de-asserts its ready output, refusing any incoming requests until the event has been signalled. A FSM in the cache bank then initiates a C-Switch burst transaction to the cluster containing the event queue. This transaction causes an entry to be placed in the event queue to describe the event that has occurred. The cache bank then re-asserts its ready line to allow the M-Switch to rout new requests to it. The event system manages the depth of the event queue, halting all processes that can cause events if it fills up, so the cache bank does not have to probe the queue to determine if there is space for the event. Because the burst mode of the C-Switch is used to transmit the event queue entry, there is no need to explicitly serialize access to the event queue – the C-Switch can only route one transfer to a given cluster per cycle, and C-Switch burst transactions are uninterruptible, making the process of enqueuing an event queue entry atomic.

Chapter 13

The External Memory Interface (EMI)

13.1 Overview

The external memory interface (EMI) of the M-Machine handles the task of moving data to and from the external memory of a node. The EMI contains the local translation lookaside buffer (LTLB), which is responsible for performing virtual-physical address translation for data that is located on a given node. The EMI receives requests from the cache banks via the X-Bus, which is also used to transfer blocks of data to and from the cache banks. In addition to the connection to the X-Bus, the EMI has a write port on the C-Switch, which is used to send the results of a memory operation directly back to the cluster that initiated the request without having to go back through the cache banks, and a connection to the G-Bus, which is used by the configuration space controller to read and write the LTLB, and to signal the EMI that an LTLB miss has been resolved.

When a cache miss occurs, the cache bank that sustained the miss arbitrates for the X-Bus and sends the request that caused the miss to the EMI. Once this has been done, the EMI is responsible for resolving the request, and the cache bank deletes the request from its internal state. The EMI then ensures that a block buffer is available, writing the contents of one of the block buffers back to external memory if necessary, evicts a sector from the cache if necessary, and fetches the line that satisfies the request from the external memory to resolve the request.

If an LTLB miss occurs, software is invoked to perform the address translation and update the LTLB. While this is occurring, the EMI does not accept normal memory requests, in order to avoid having to deal with additional LTLB misses while the first miss is being resolved. If the needed data is in the external memory, the LTLB is updated by the LTLB miss handler to contain a translation for the request, and the EMI retries the request, which can now be handled by the hardware. If the needed data is not located on the local node, the EMI discards the request, and software is invoked to resolve the request.

The bandwidth from the EMI to the external memory is one word per cycle peak, although this cannot be sustained across multiple cache misses, as the same datapaths are used to transfer data and requests between the EMI and the cache banks. Latency from the receipt of a request by the EMI until the arrival of the first word from the external memory is expected to be approximately five cycles, although this will vary depending on the speed of the SDRAMs used in the external memory. The EMI can be programmed to expect between one and 32 cycles of latency between the assertion of a request to the SDRAMs and the arrival of the first word of data from the SDRAMs, but it expects data to arrive at one word/cycle once the first word has arrived. Figure 13.1 shows the overall structure of the EMI. Two address and data registers (the *Req* and *Req_Byp* registers) hold requests that have come in from the banks over the X-Bus. The LTLB module performs virtual-physical address translation. The Data_Buf and Addr_Buf blocks serve as buffers for data going to/from the SDRAMS.

13.2 Timing

The large-scale timing of the handling of a cache miss is as follows (latency of SDRAM from request to first word back = n cycles):

- Cycle 1: Request received by a cache bank. Cache miss detected.
- Cycle 2: Bank arbitrates for X-Bus, request routed to EMI if bank wins arbitration. EMI lowers Rdy line to indicate that it will not accept a normal request on the next cycle.
- Cycle 3: EMI begins processing request. EMI examines Buf_Dirty lines to determine if there is a clean block buffer available. If not, bank empties a block buffer and continues. EMI also sends the address that missed in the cache to the LTLB, and raises an event if an LTLB miss occurs.
- Cycle 4: EMI sends command to banks to evict the sector that will be used to hold the incoming line. The evicted sector is placed in a block buffer. If the request hits in the LTLB, the EMI starts the read to the SDRAMS to fetch the block that will satisfy the miss.
- Cycle 6: Banks complete the eviction of the requested sector.
- Cycle 4+n: EMI sends command to banks informing them that block transfer is about to begin. First data word from SRAM arrives at EMI.
- Cycle 5+n: EMI sends first word to cache banks. This is the word that satisfies the cache miss request, using the SDRAM wrapped-load capability. If the request was a load, the EMI sends the word back to the originating cluster over the C-Switch. If the request was a store, the EMI substitutes the stored data for the data from the SDRAM.

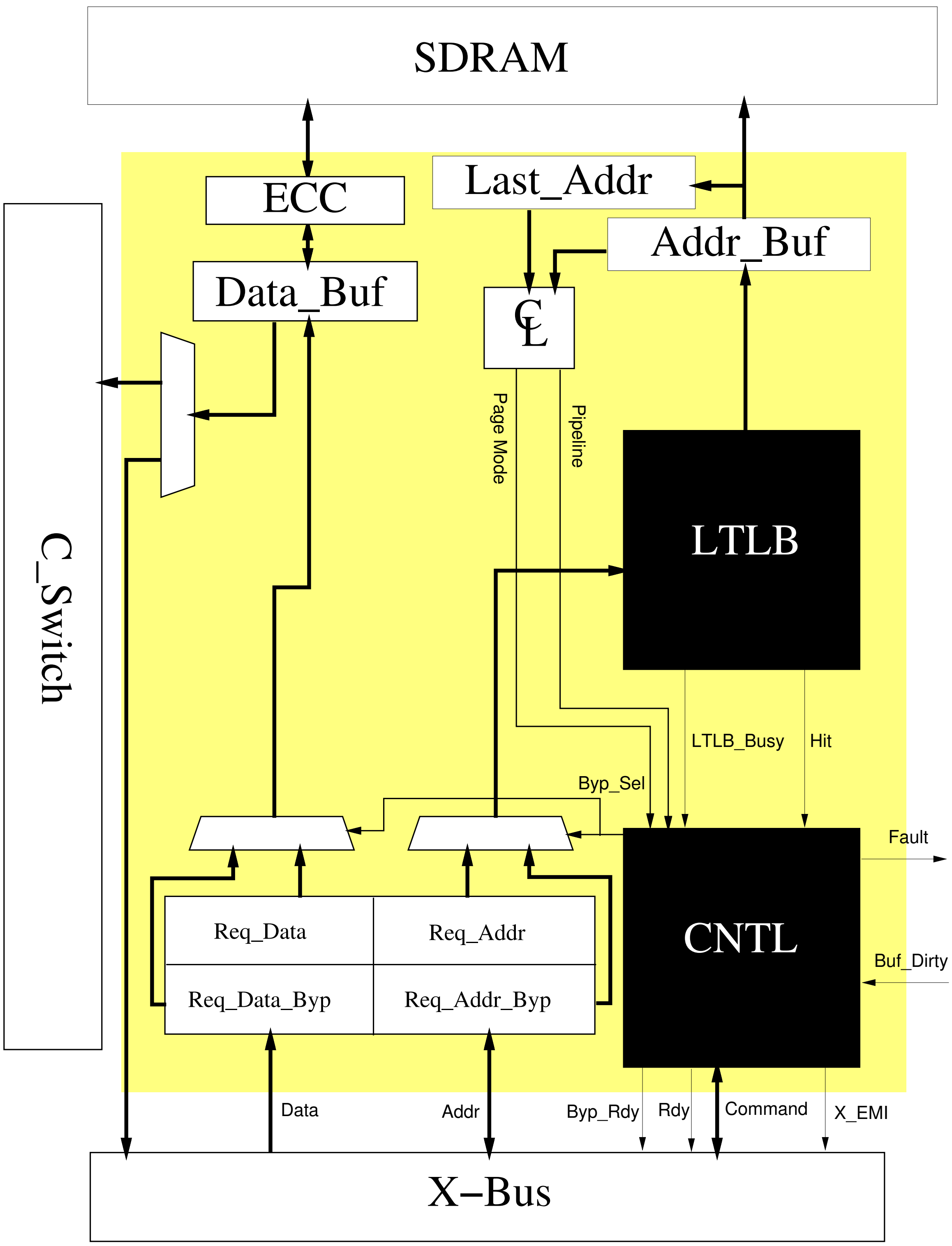


Figure 13.1: *Block Diagram of the EMI*

- Cycle 12+n: Last word of data block transferred to banks. EMI raises Rdy line to indicate that it can accept a new request on the next cycle.

- Cycle 13+n: EMI able to accept next request.

Note that one of the input bits to the EMI for each request tells the EMI whether or not the request is to an unloaded block of a sector whose other block is in the cache. If this occurs, the EMI does not evict a sector from the cache, as it can simply load the required block into the sector which is already in the cache. If the EMI needs to evict a sector, and cannot find a clean block buffer, it must write one of the block buffers back to the SDRAMs before it can load the block needed to satisfy the request. This is accomplished by sending a command to the cache banks, instructing them to write the buffer back. The cache lines then transfer the contents of the buffer to the EMI over the X-Bus, and the EMI writes the data back to the SRAMs. This process requires either 8 or 16 cycles to transfer the contents of the buffer over the X-Bus, depending on whether one or both of the blocks in the sector are dirty, 2 cycles to set up the transfer, and n cycles of latency between the start of the SDRAM access and the first write, for a total of $10 + n$ or $18 + n$ cycles to complete the writeback. This delay is inserted between the operations that are performed in cycles 3 and 4 of the above timeline, which assumed that a clean block buffer was available.

13.3 The LTLB

The LTLB, which handles virtual-physical address translation for data contained in the SDRAMs of the local node, is one of the major subcomponents of the EMI. Figure 13.2 shows the internal structure of the LTLB. The main array of the LTLB is 64 sets by two entries/set. When an address is submitted to the LTLB, the appropriate bits of the address are used to select a set from the array. The virtual fields of the two entries in the set are routed to a comparator, which compares the contents of these fields to the address being translated to determine if a hit has occurred. If the pointer bing translated by the LTLB is a *physical* pointer, no translation is performed, as the address in the pointer refers to the physical address space of the node. If a hit occurs, the Hit and Sel lines are set appropriately, and cause the physical and status fields of the appropriate set to be driven on the output lines by the multiplexors. The Hit and Sel lines are fed into the LRU bits of the array, to update the LRU status of the array. If a miss occurs, the Hit output of the LTLB is seen by the control logic of the EMI, which signals an LTLB miss event.

13.3.1 LTLB Entry Replacement

The contents of the LTLB can be read and written by way of the config space. The config space controller interprets references to certain config space locations as references to the LTLB, and issues requests over the G-Bus to the LTLB to read or write the LTLB state. The interface between the G-Bus and the units that it connects has not yet been fully specified.

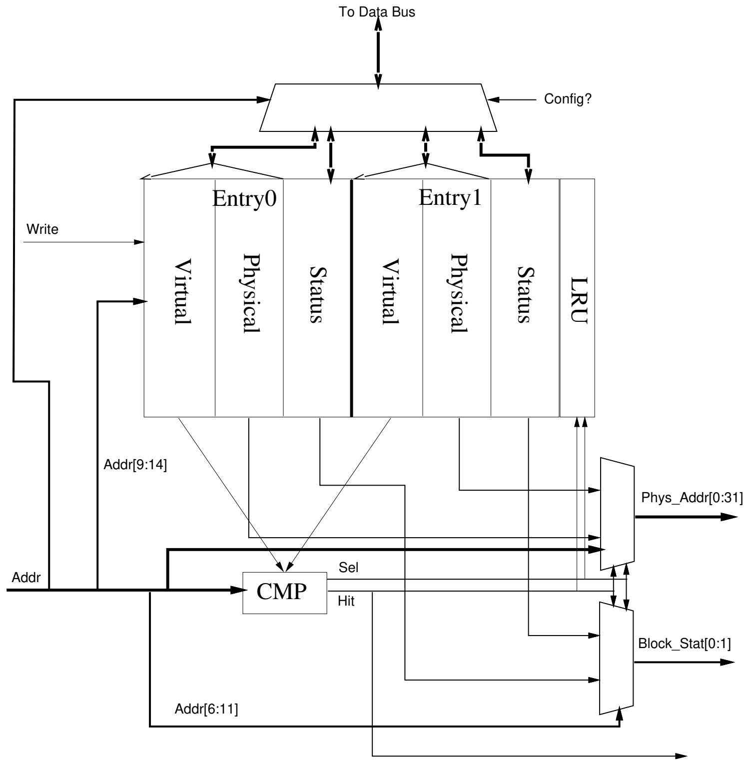


Figure 13.2: *Block Diagram of the LTLB*

The software that handles LTLB misses is responsible for determining which entry in the appropriate set should be replaced by reading the first word in the first entry in the set. A bit in this word is set to one if the first entry in the set is the LRU entry, zero if the second entry is the LRU entry. The LTLB miss handler must then write the new entry into the appropriate place in the set. The sequence of events that occurs on an LTLB miss is:

1. Cache bank sends request that missed to EMI
2. EMI receives request and sends address to LTLB for translation
3. LTLB determines that a miss has occurred
4. EMI signals that LTLB miss event has occurred. EMI holds Rdy output low until LTLB miss is resolved and the original request handled to indicate that it cannot accept new memory requests.
5. EMI routes the data needed by the LTLB miss handler to the appropriate cluster over the C-Switch.
6. LTLB miss handler notes that an LTLB miss has been queued and begins handling the request
7. LTLB miss handler accesses page table and finds a translation for the address
8. LTLB miss handler writes an entry containing the translation into the LTLB
9. LTLB miss handler signals EMI that LTLB has been updated.
10. EMI retries translation and completes the handling of the original request
11. EMI raises Rdy line to indicate that it is ready to accept new requests

If the LTLB miss handler is unable to find a local translation for the address, (if it determines, for example, that the address is contained on another node) it signals the EMI that the request will be resolved through software. The EMI then discards the request that caused the miss and raises its Rdy line to indicate that it is ready to accept new requests.

13.3.2 Status Bits

Each page of memory is divided into 64 memory blocks, each of which has two tag bits that encode the states invalid, read-only, read-write, and dirty. These tag bits are intended for use in implementing relocation and cache coherence schemes, by allowing protection states to be enforced on small blocks of data. The memory system hardware enforces the protection implied by these states, signalling an event if any attempt is made to reference an invalid line or to write a read-only line. These status bits are contained in the page table entry for each page, and are shadowed in the LTLB and the cache. When the LTLB translates an address, it also outputs the status bits for the block that it is providing a translation for, and the EMI sends this information to the cache banks along with the line being fetched. When a line is evicted from the cache, the status bits in the LTLB are changed to dirty if any word in the line has been written.

13.4 SDRAM Interface

All accesses to the SDRAMS are made using the burst mode of these devices to transfer data in 8-word blocks. For cache block reads, the burst sequence feature is used to implement needed-word-first memory access. It appears that writebacks of entire 16-word sectors can be accomplished by using the full-page burst mode and asserting the burst stop command after the 16-word sector is written back. If this is not possible, it will probably be necessary to use two eight-word bursts to write back a sector if both blocks are dirty. The somewhat sketchy documentation we have suggests that it may be possible to pipeline these two accesses to reduce latency by starting the second access during the last few cycles of the first. At 100 MHz, it appears that the latency between the start of a memory read and te return of the first word from the SDRAMs is six cycles. This suggests that the latency to fetch the datum required by a load that misses in the cache will be 11-12 cycles, with the total time to fetch a cache line being 18-19 cycles.

ECC logic will be implemented on the external memory. The exact error correction parameters have not been specified. The SDRAM interface logic will handle the task of encoding/decoding data as it moves to/from the SDRAMs, as ECC will not be implemented on the on-chip memory.

Chapter 14

The X-Bus

14.1 X-Bus Interface Specification

This appendix describes the interface between the cache banks and the X-Bus. It is included as an appendix to this document because the detailed design of the X-Bus has not yet been done, but it is necessary to have some specification of the X-Bus in order to design the cache banks. At some point, this appendix will be removed, and its contents migrated into the X-Bus design document.

The X-Bus must differ from the C-Switch and the M-Switch in a number of ways. First, it must be bidirectional, able to carry requests from the banks to the EMI, and commands and data from the EMI to the banks. A broadcast capability is also desirable, to allow commands that need to go to all of the cache banks to be sent in one cycle instead of one cycle/bank. The X-Bus must be wide enough to carry an address and a request specifier in one cycle.

Each bank communicates with the X-Bus arbiter through four signals:

- X_Req: Requests access to the X-Bus for the cache bank
- X_Byp: Indicates whether the bank wants the X-Bus for a normal or a bypassing operation
- X_Grant: Tells the bank whether or not the arbiter has granted it access to the bus this cycle
- X_EMI: Indicates that the EMI has taken control of the X-Bus this cycle. No bank is granted access to the bus, and all banks should examine the contents of the bus to determine if it applies to them.

In addition to these control signals, each bank has three outputs, which are used by the EMI to decide which block buffer should be used when a cache block is evicted or an uncached access is performed:

Transfer			*Address*	*Return*					*Data*	
op	pre	post	addr	rtnclst	tslot	rf	reg	rtncc	sync	data
7	2	1	51	2	3	1	4	3	1	65 bits

Figure 14.1: X-Bus Organization When Bank Driving

Transfer			*Address*	*Return*					*Data*	
cmd	bank	all	addr	unused	unused	unused	unused	unused	sync	data
7	2	1	51	2	3	1	4	3	1	65 bits

Figure 14.2: X-Bus Organization When EMI Driving

- Buf_Dirty0: Indicates whether or not line buffer 0 contains dirty data
- Buf_Dirty1: Indicates whether or not line buffer 1 contains dirty data
- Buf_LRU: Indicates which line buffer has been least recently used

During transmissions of requests from the banks to the EMI, the data portion of the X-Bus is identical to the Transfer, Address, Return, and Data fields of the M-Switch. Figure 14.1 shows this format.

The EMI can get control of the X-Bus at any time by asserting its Grab line. Multiword transfers, such as cache line writebacks and fills, do not use this request/acknowledge protocol. Instead, the EMI sends commands to the banks to start the operation, and state machines within the banks and the EMI ensure that the X-Bus is used properly.

The X-Bus is viewed in a different way when the EMI is driving it, as shown in figure 14.2. The bank and all fields determine which bank the command is intended for. If the all bit is set, the command is a broadcast, to be read by all banks. Otherwise, the bank field contains the number of the bank for which the command is intended. The addr field contains the address argument to the command, and the data field contains any other arguments to the command. The cmd field contains a command from the EMI to the banks, selected from the following list:

- Block Incoming: The EMI is about to transfer a block of data to the cache banks. The addr field encodes the set containing the line. This command is sent as a broadcast to all cache banks. The low two bits of the data field encode the bank that will receive the first word, and the third bit of the data field specifies the line that will receive the data. The cache banks are responsible for monitoring the incoming data in order to detect incoming lines that satisfy requests that have missed in the bank and complete the requests without sending the requests to the EMI.
- Block Buffer Fill: The EMI is about to transfer a block of data to one of the block buffers. This command is sent as a broadcast to all cache banks. The low two bits of

the data field encode the bank that will receive the first word, and the third bit of the data field specifies the block buffer that will receive the data.

- Evict Block: This command is sent as a broadcast to all banks, and instructs the banks to evict a cache line. The set containing the line is specified in the addr field. The low bit of the data field specifies the line buffer into which the line should be copied if dirty, and the third bit of the data field specifies which of the two lines within the specified set should be flushed.

- Flush Block: Format is the same as Evict Block, but instructs the banks to write the block back to the external memory if it is dirty, instead of copying it to a line buffer. This command is used when the EMI wishes to ensure that no cached copies of a datum remain. On the cycle following the issue of this command, the EMI monitors the wired-or dirty line to determine if the flushed block was dirty, in order to know whether or not to write back the block.

- Flush Buffer: Instructs the banks to write the contents of a block buffer back to main memory. The block buffer to be flushed is specified in the third bit of the addr field.

Chapter 15

The Network Interface

15.1 Functional Overview

The function of the network units is to bridge the communication between the network router and the clusters in a MAP chip. There are two types of network interface units on each MAP chip:

- *Network Input Unit.* The **input** unit handles message **reception**.
- *Network Output Unit.* The **output** unit handles message **injection**.

Each of the input and output units is replicated to serve the two message priorities in the M-Machine, so that the resources used by priority 0 messages are independent from those used by priority 1.

The network input units are logically situated between the network router and clusters 2 and 3 of the MAP chip, while the network output units are located logically between the C-Switch and the router, as shown in Figure 15.1.

15.2 Network Input Unit

The description of the network interface units is divided into two sections. The Network Input Unit is described in this section, while the Network Output Unit is discussed later.

15.2.1 Operation of the Network Input Unit

The network input unit accepts messages from the router at the rate of one word per processor cycle (max), places each word into a *network input queue*, and notifies the cluster it is

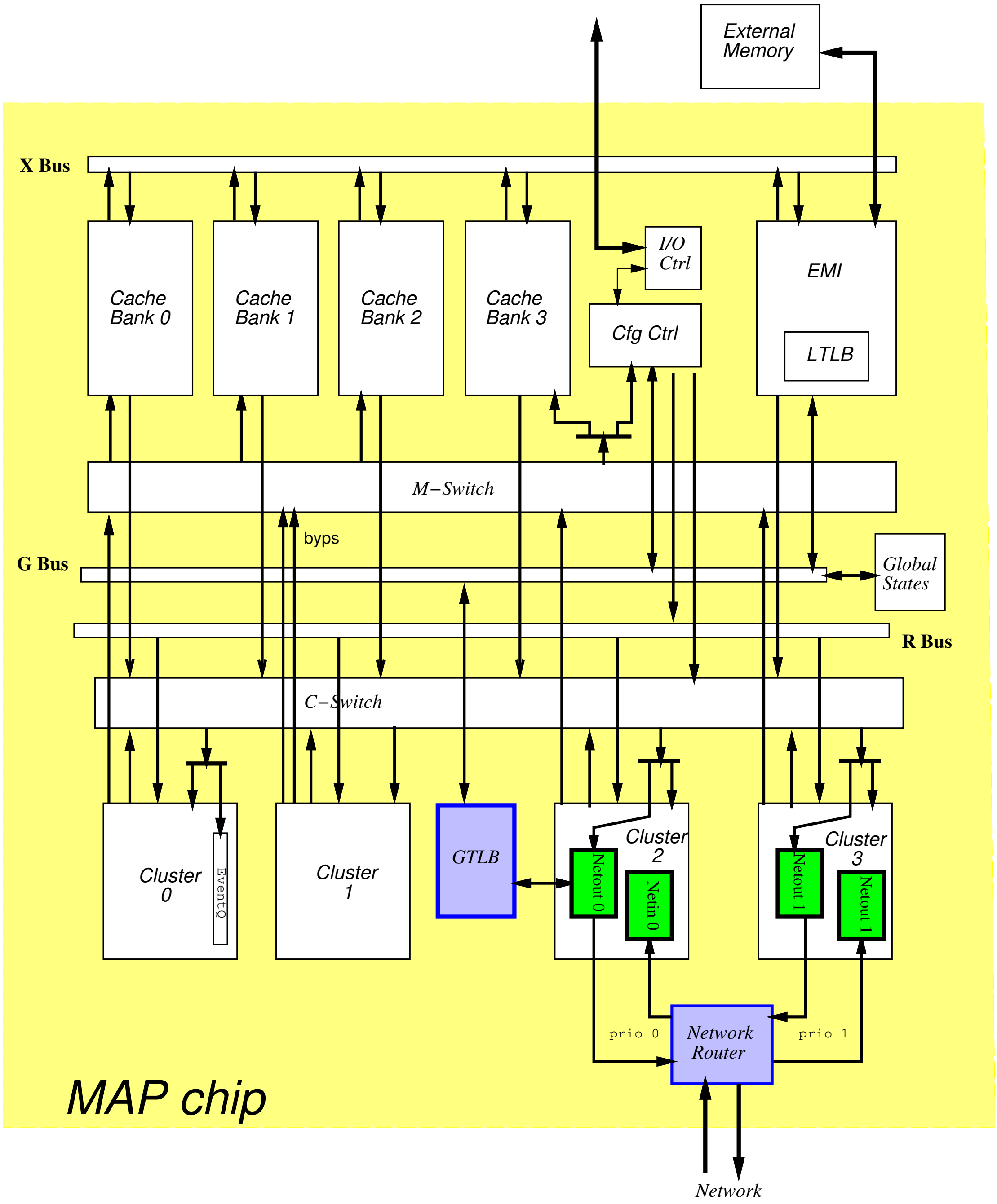


Figure 15.1: *The Network Interface Units within the MAP chip*

attached to of the availability of the word. The message words are read out of the queue in a FIFO order by the cluster, via the *rhead/rbody* mechanism as detailed in *M-Machine Architecture v1.0.*

The network input unit relies on the SZ/EX stages to indicate to it when a word should be popped from the queue, but is otherwise responsible for enforcing the following characteristic of the M-machine message subsystem:

- Reading *rbody* **beyond** the end of a message returns an *errval*. (Should this be a constant, or derived from the something else like the IP?)
- Reading *rhead* **before** the end of a message flushes the message, and returns the *dispatchIP* of the next message.

Operations that contain two or more simultaneous reads to *rhead* **or** *rbody* have the same effect on the network input queue as a single read to the corresponding register, although the return value is bypassed to both destination registers. Operations that contain simultaneous reads to *rhead* **and** *rbody* will have the same effect as simultaneous reads to only *rhead*. The same holds for multi-operation instructions that read *rhead* and/or *rbody* simultaneously.

Examples. In the following table, the sequences in column 1 and column 2 return the same result in *r_result*, and leave the network input queue in identical states:

column 1	column 2	Effect
instr ialu add rbody, rbody, r_result;	instr ialu mov rbody, rtemp; instr ialu add rtemp, rtemp, r_result;	pops/returns one word from input queue.
instr ialu add rhead, rhead, r_result;	instr ialu mov rhead, rtemp; instr ialu add rtemp, rtemp, r_result;	flushes previous message, pops/returns first word of new message.
instr ialu add rhead, rbody, r_result;	instr ialu mov rhead, rtemp; instr ialu add rtemp, rtemp, r_result;	flushes previous message, pops/returns first word of new message.
instr ialu mov rhead, rtemp1 memu mov rbody, rtemp2; instr ialu add rtemp1, rtemp2, r_result;	instr ialu mov rhead, rtemp; instr ialu add rtemp, rtemp, r_result;	flushes previous message, pops/returns first word of new message.

15.2.2 Interfacing to the Network Input Unit

The netin unit communicates with the router, the SZ stage, and the EX stage, as shown in Figure 15.2. The signals shown carry the following meanings:

- `router`:
  - `data`: Data from the router is read by the netin unit via the `data` lines, one word at a time. The `data` lines are word-wide (66 bits, including the memory sync bit).
  - `wordav`: Signals that the `data` lines actually carry a valid word.

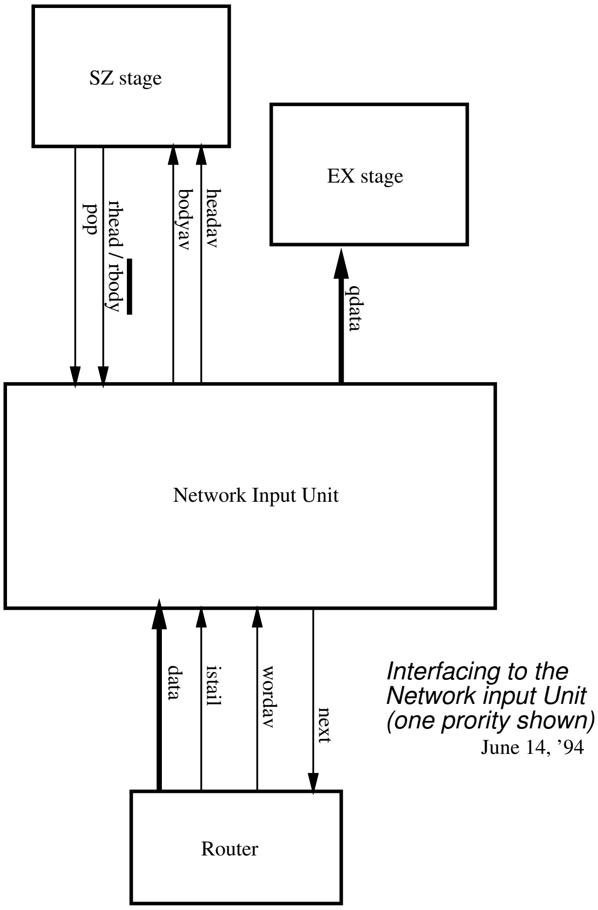


Figure 15.2: *Interfacing to the Network Interface Unit*

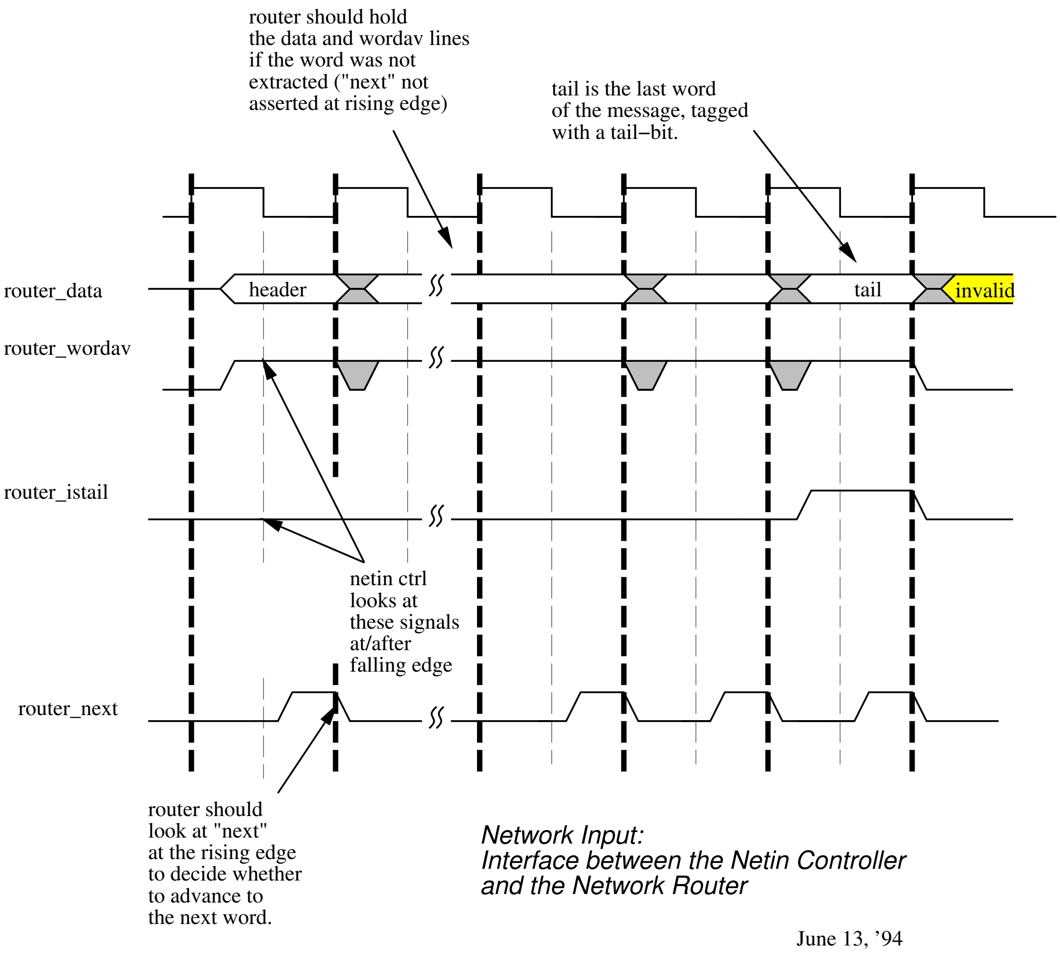


Figure 15.3: *Handshake between Netin Unit and Router*

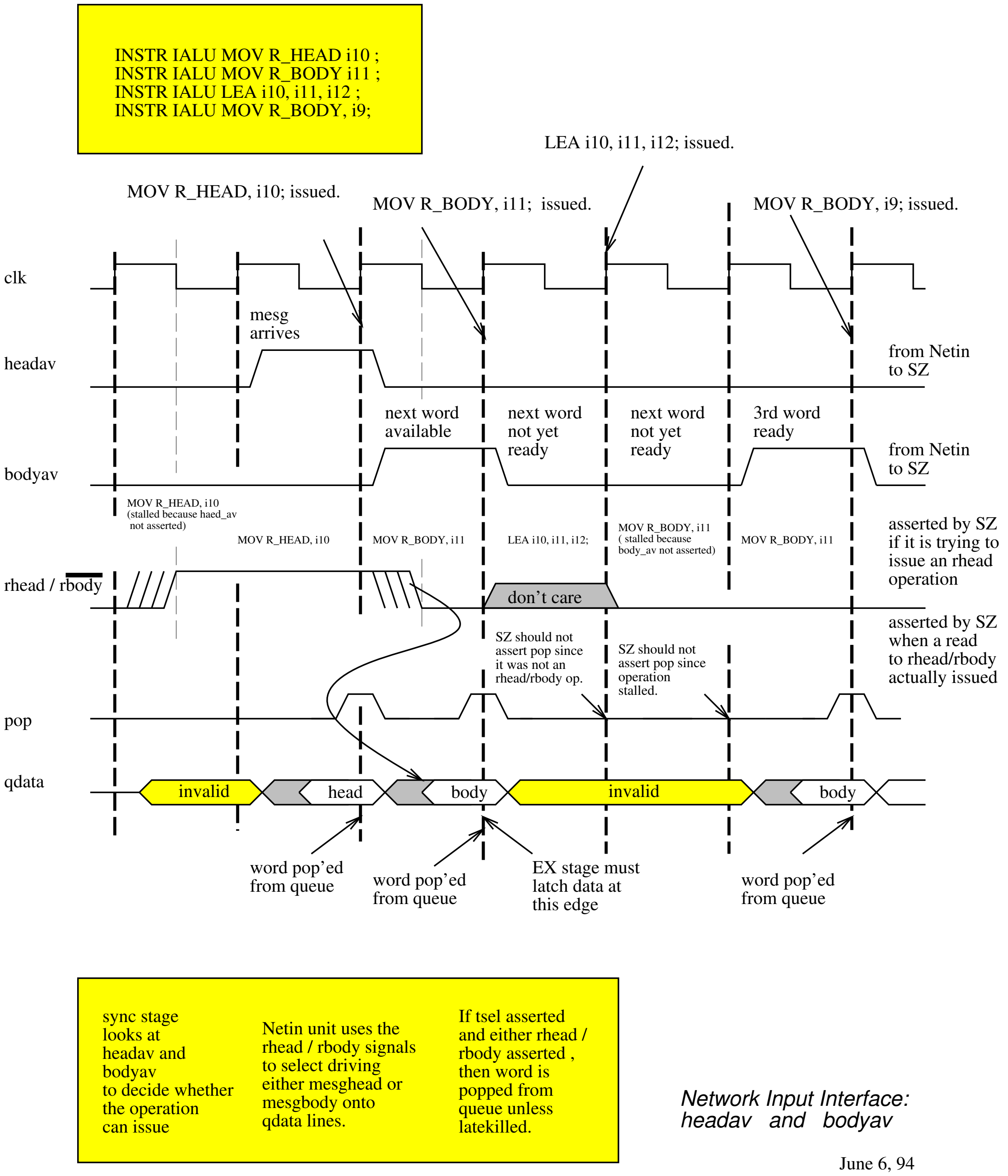


Figure 15.4: *Handshake:* `headav` *and* `bodyav`

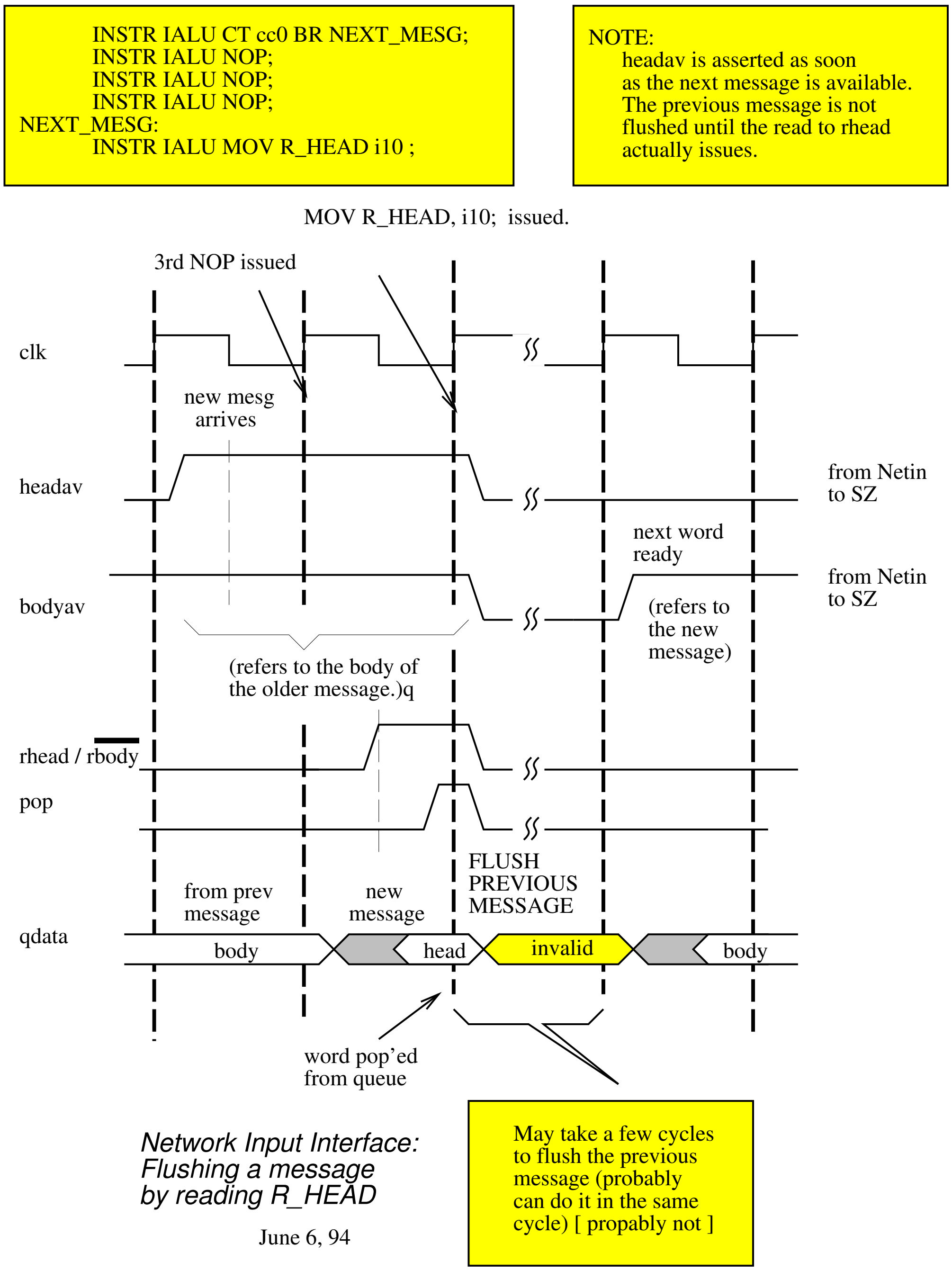
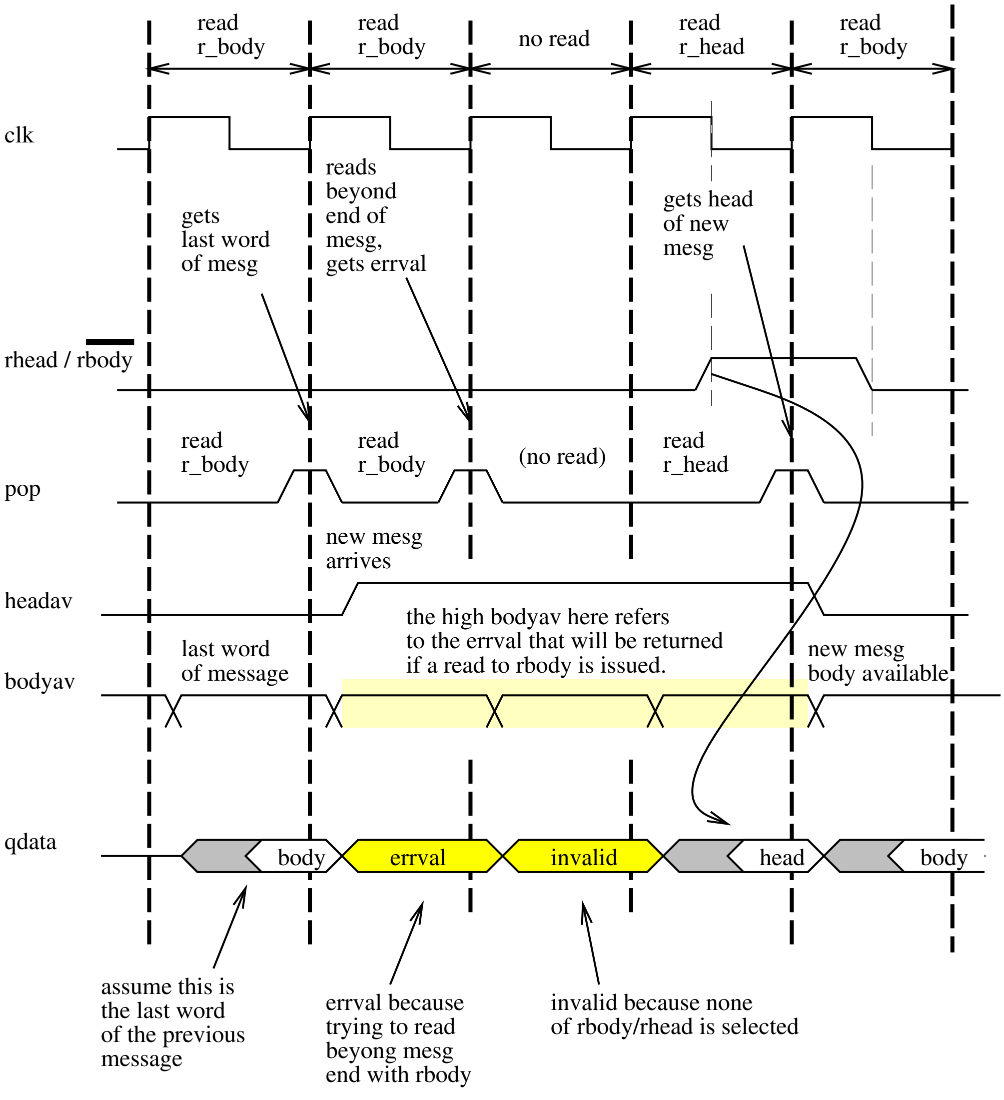


Figure 15.5: *Handshake: Flushing a Message by reading* rhead

(this diagram assumes that an op from the message handler slot is issued in each of these cycles)



Network Input Interface: Read Beyond End of Message

June 14, '94.

Figure 15.6: *Reading beyond the end of a message*

- `istail`: Signals that the word carried on the `data` lines is the last word of a message.
- `next`: Used by the netin unit to request for the next word to be put onto the `data` lines.

- SZ
  - $\texttt{rhead}/\overline{\texttt{rbody}}$: Set to 1 by the SZ stage when it is attempting to issue an operation that reads the *rhead* register, and 0 otherwise. *rbody* register is requeste.
  - `pop`: Asserted by the SZ stage to signal that the read to rhead/rbody was indeed issued. Not asserted if the instruction issued does not contain a read to *rhead/rbody*.
  - `headav`: Asserted when the SZ stage may issue a read to the *rhead* register.
  - `bodyav`: Asserted when the SZ stage may issue a read to the *rbody* register.

- EX
  - `qdata`: Carries a data word from the network input queue, selected by `rhead` and `rbody`.

The timing diagrams in Figure 15.3 through Figure 15.6 define the handshake signals between the modules described.

As defined in *M-Machine Architecture v1.0*, the condition register *CC0* in the system thread slot (for H-Threads 2 and 3) is automatically set to TRUE when the corresponding message queue contains the next message. This information can be derived by the RR/SZ stage by reading the `headav` line.

15.2.3 Architecture of the Network Input Unit

The network input unit contains two main blocks, *i.e.* the Network Input Queue, and the Netin Controller, as shown in Figure 15.7. (There is one network input unit for each message priority.)

The network input queue is implemented as a register file with 1 read port and 1 write port. Two counter registers, *qhead* and *nxtmsg*, are used to keep track of the first word in the queue, and the location of the next dispatchIP in the queue, respectively. A separate small FIFO (*NXTMSG FIFO*) is used to keep track of the *delta's* between the register addresses of the beginning of messages (*i.e.* the dispatchIP) in the input queue. As each new message arrives, the length of the message is enqueued to the small FIFO. As each message is flushed from the input queue, a *delta* value is dequeued from the FIFO, and added to *nxtmsg* to form the read address for the next *dispatchIP*. A 5-bit *msgcount* counter is used to track the number of *dispatchIP*s present in the input queue.

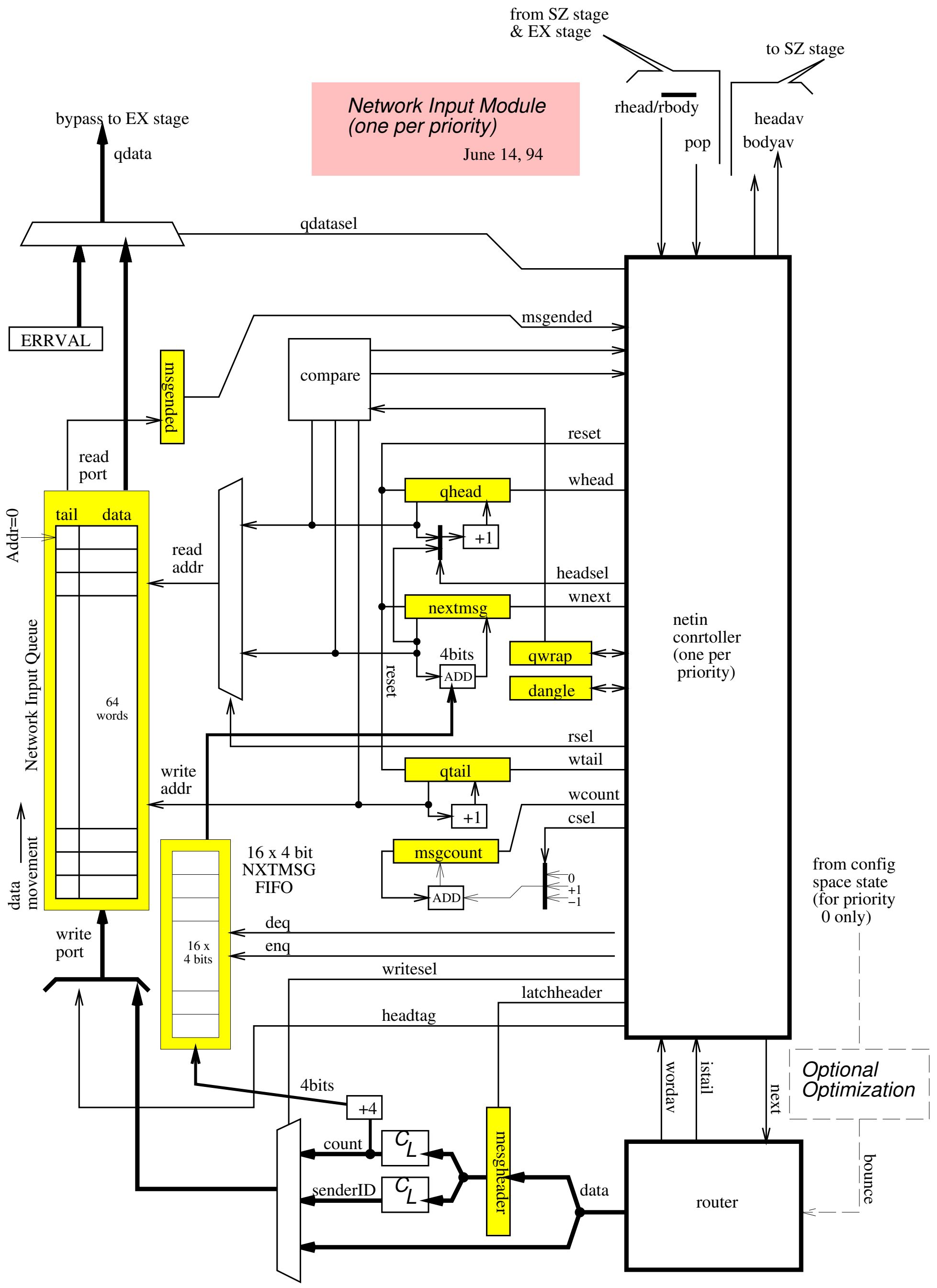


Figure 15.7: *A Network Interface Unit*

A *tail* tag is associated with each word in the network input queue (set to one for the last word in each message), and is used to trigger the netin controller into returning ERRVALs for read attempts to *rbody*, until the next read to rhead is issued.

The `qdatasel` line which select between an ERRVAL and the value from the input queue read port to be driven onto `qdata` is derived from the `rhead/`$\overline{\texttt{rbody}}$ signal (along with the internal state of the controller). The write port is addressed by a *qtail* counter. A *dangle* register is used to flag when a message is in the process of being extracted from the network (*i.e.* part of the message is still arriving). The *qwrap* state bit flags when the *qhead*/*qtail* pointers have wrapped around.

Each of the network input queues is 64 words long.

15.2.4 Operation of the Netin Controller

The netin controller is made up of a FSM and a number of small combinational logic blocks, instead of being implemented as a monolithic large FSM. The functions of the controller are described in this section.

At initialization, the counters are reset to the following states:

- $qhead = 0$
- $nxtmsg = 0$
- $qtail = 0$
- $dangle = 0$
- $qwrap = 0$
- $msgended = 1$
- $msgcount = 0$

When a message arrives, the following actions take place at the router/netin-controller interface (the controller waits if the input queue is full):

- Latch the message header into the *mesgheader* register for later use. Set the *dangle* bit on.
- Get the next word from the router (the *dispatchIP*) and put it into the input queue. Enqueue ($argcount + 2$) onto the NXTMSG fifo. Increment *msgcount* by 1.
- Decode/expand *mesgheader* to deposit two words, the senderID and the ArgCount, into the input queue (2 cycles).

- Extract the rest of the message into the input queue. Clear the *dangle* flag when the tail word is extracted. Tag the last word as *tail* in the input queue.

At the other end of the queue, `headav` is asserted when *msgcount* > 0, while `bodyav` is lowered only when the queue is empty and *dangle* is set. The `qdatasel` line selects ERRVAL to be driven onto `qdata` when an attempt is made to read *rbody* beyond the end of a message. Otherwise, the value from the read port of the input queue is driven onto `qdata`. Whenever a read to *rhead* is successfully issued, *msgcount* is decremented by one, and the *nxtmag* counter is incremented by the *delta* value dequeued from the front of the NXTMSG fifo. This leaves *nxtmsg* pointing to the queue location where the next *dispatchIP* is, or will be stored when it arrives. (If the next *dispatchIP* has not actually arrived, another read to *rhead* would not issued until *msgcount* becomes > 0 when the next *dispatchIP* is actually written into that location.)

The *msgended* flag register latches the *tail* tag from the input queue read port at the rising clock edge.

15.2.5 Internal Architecture of the Netin Controller

The message extraction interface between the router and the network input queue is handled by a simple FSM, shown in Figure 15.8.

For the other end of the queue, all of the state necessary for managing the interface is captured in the *dangle*, *msgended* and *qwrap* flags, the *msgcount*, *qhead*, *qtail* and *nextmsg* counters. This interface can therefore be realized via a number of small combinational logic blocks, shown in Figures 15.9 and 15.10. This implementation assumes the queue length to be an integer power of 2. (We assume a length of 64 words.)

15.2.6 Formats

Message flits are transfered in the following format:

tag			
head	tail	mem sync	data word
1 bit	1 bit	1 bit	65 bit

Messages received from the router are expected to be in the following format (each *flit* = 2 *phits* = 1 word + control signals):

- Flit 0 = Message Header
- Flit 1 = Dispatch IP (permission = ENTERSYS)

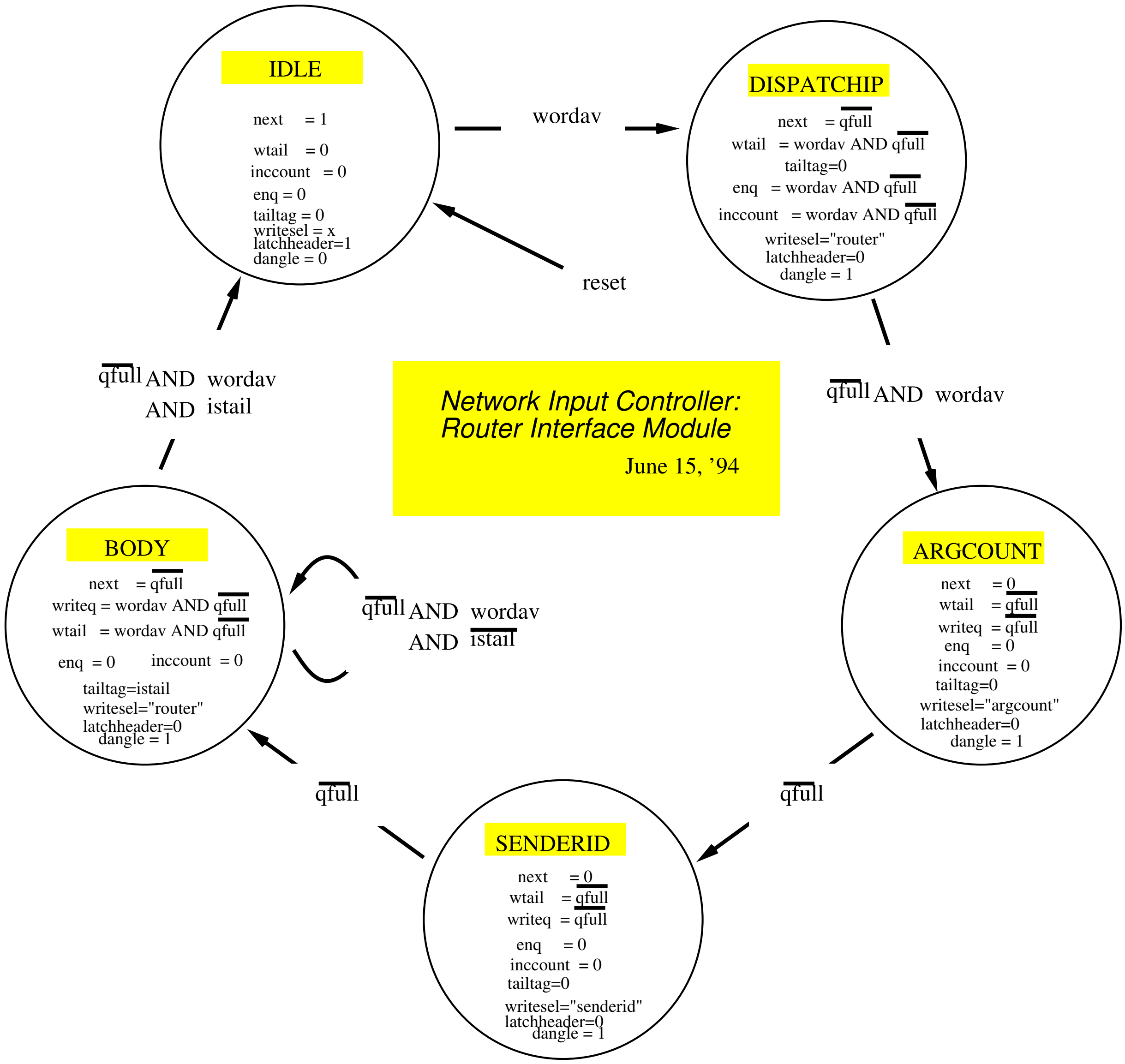


Figure 15.8: *Network Input Controller*

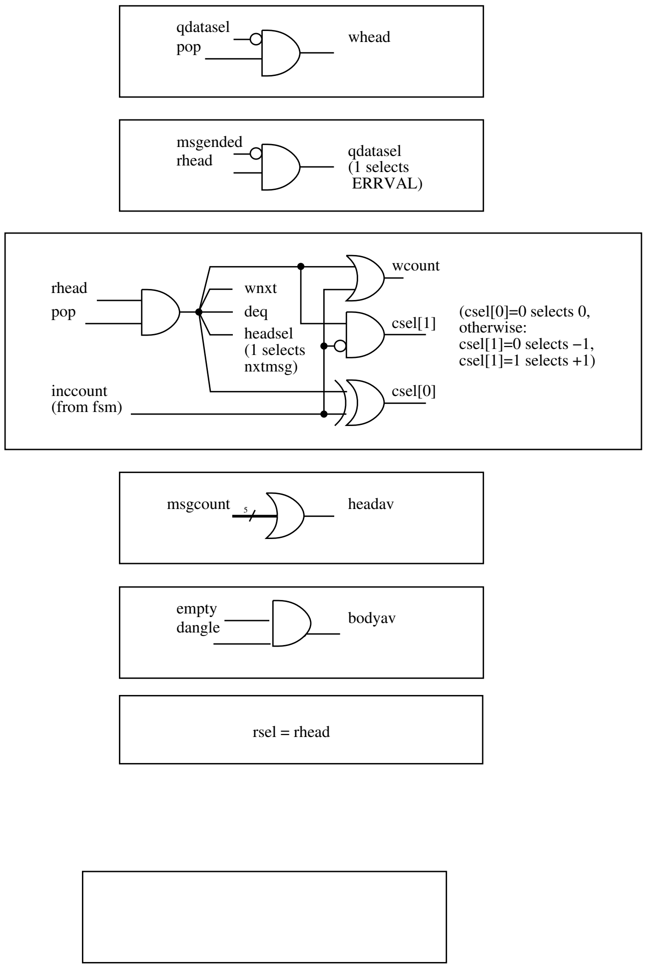


Figure 15.9: *Network Input Queue Control Logic*

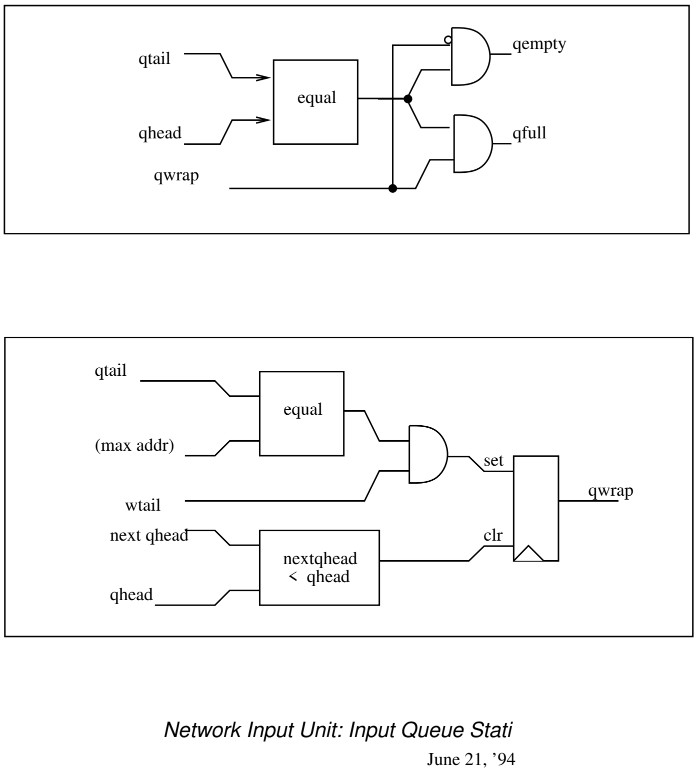


Figure 15.10: *Network Input Queue Status Logic*

- Flit 2 = Destination (as specified in the SEND operation)
- Flit 3 onwards = arguments as specified in the *MC* registers.

The message header flit is expected to be in the following format (the router is expected to generate the proper *head* tag for the header at the injection point):

68 bits											
Header Tag	Tail Tag	–	Sender ID			ArgCount	priority	Destination ID			
1 bit	1 bit	29	6 bits	5 bits	5 bits	4 bits	1 bit	6 bits	5 bits	5 bits
1	0	unused	Z	Y	X	Argument Count	0/1	Z	Y	X

15.3 Network Output Unit

The network output unit is responsible for accepting SEND requests from the C-Switch, assembling messages, and managing message injection. A separate SEND arbitration module is responsible for arbitrating amongst simultaneous SEND requests.

15.3.1 Sending a Message

When a cluster needs to send a massage, it first ensures that the network resources for the appropriate priority are available by examining a `netqav` line from the network output unit. (The SZ stage should also look at the `ombc` line if the SEND operation is of the throttling varient, and not arbitrate for network resources if the `ombc` line is 0 in that case.) If the required resources are free, the SZ stage raises a `sendreq` line leading to the SEND Arbiter to arbitrate for the resources. If the cluster loses arbitration, it will re-assert `sendreq` at a later time.

If the requset is granted, the SZ stage must issue the SEND operation in the immediately following cycle.

If the SEND operation is issued, the cluster arbitrates for a burst mode transfer across the C-Switch targetted at the Network Output Queue of the appropriate priority. The following data sequence is expected by the network output unit in this transfer:

1. *destination.* This is either a virtual address, or a *Node Identifier* (nodeID) which specifies the XYZ routing information. (The nodeID is returned by the GTLB when translating a destination virtual address.)
2. *dispatchIP.*
3. rest of the message from the *MC* registers (if any), beginning from *MC 0.*

		control			*destination*			*condition*		*data*	
# bits	1	1	1	3	3	1	4	1	3	1	65
Regular meaning	dav	lk	mb	xfer type	tslot	rf	data reg	*CC* flag	*CC* reg	mem sync	data word
Overloaded during SEND	(same)	not used	decombc	sender tslot	tslot= "queue"	0=ISEND 1=FSEND	*MC* count	xlate	return *CC* reg	mem sync	data word

Table 15.1: Overloading of C-Switch Signals During SEND

		control			*destination*			*condition*		*data*	
# bits	1	1	1	3	3	1	4	1	3	1	65
Regular meaning	dav	lk	mb	xfer type	tslot	rf	data reg	*CC* flag	*CC* reg	mem sync	data word
Overloaded at CC writeback	same	same	same	xfertype= "cc xfer only"	same	same	not used	*CC* flag	return *CC* reg	sdone	not used

Table 15.2: Overloading of C-Switch Signals During *CC* Write Back

Some of the C-Switch data path wires are overloaded with different meanings during a SEND transfer as shown in the Table 15.1. These overloaded signals must be valid during the *destination* cycle. (These signals are defined in the next section.)

When a SEND request is granted, the corresponding network resources are marked unavailable. Upon receiving a SEND packet from the C-Switch, the network output unit initiates a GTLB translation if necessary and assembles the appropriate message into a Network Output Queue. If a GTLB miss occurs, the network output unit discards the message, and creates/inserts an entry into the event queue.

The network router may initiate the message injection as soon as the message header is ready (*i.e.* without waiting for the entire message to finish being assembled).

After the burst mode transfer across the C-Switch completes, The network output unit writes back a TRUE to the *CC* register of the sender (as specified in the SEND operation) via the C-Switch. During this write back transfer, the C-Switch wires are overloaded as shown in Table 15.2. The `sdone` line, when asserted, signals the cluster to clear the *SEND bit*. The *SEND bit* is a state kept in the cluster, which is set when a SEND operation issues, and used to indicate that the thread should not be evicted until the SEND bit is deasserted.

15.3.2 Interfacing to the Network Output Unit

The netout unit connects to the SZ stage, the C-Switch, and the network router, as shown in Figure 15.11. The signals shown have the following definitions (recall that there are two set of the following signals, for the two message priorities):

- `csw`
  - `sender tslot` and `return CC reg`: identify the *CC* register to be written to when the SEND operation completes.

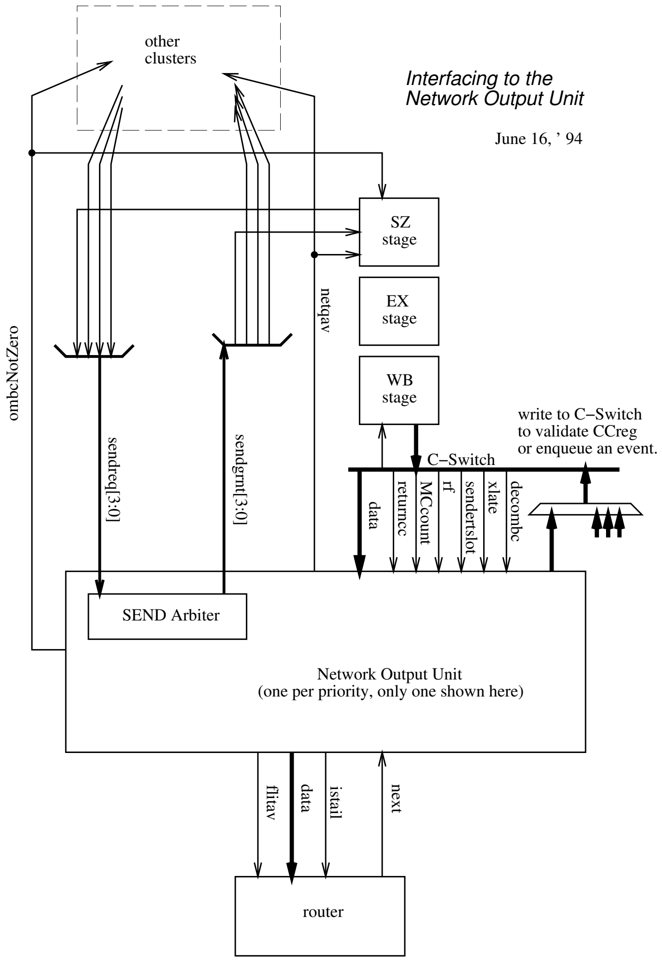


Figure 15.11: *Interfacing to the Network Output Unit*

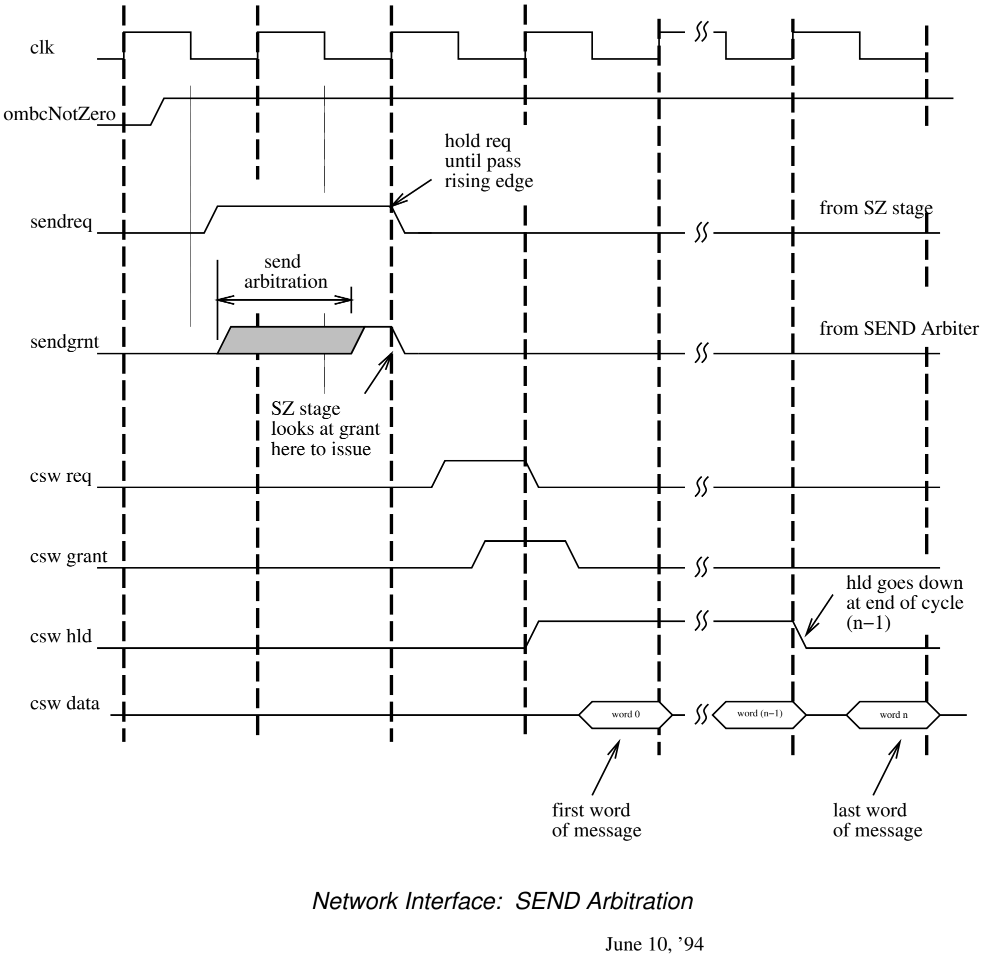


Figure 15.12: *Issuing a SEND Operation*

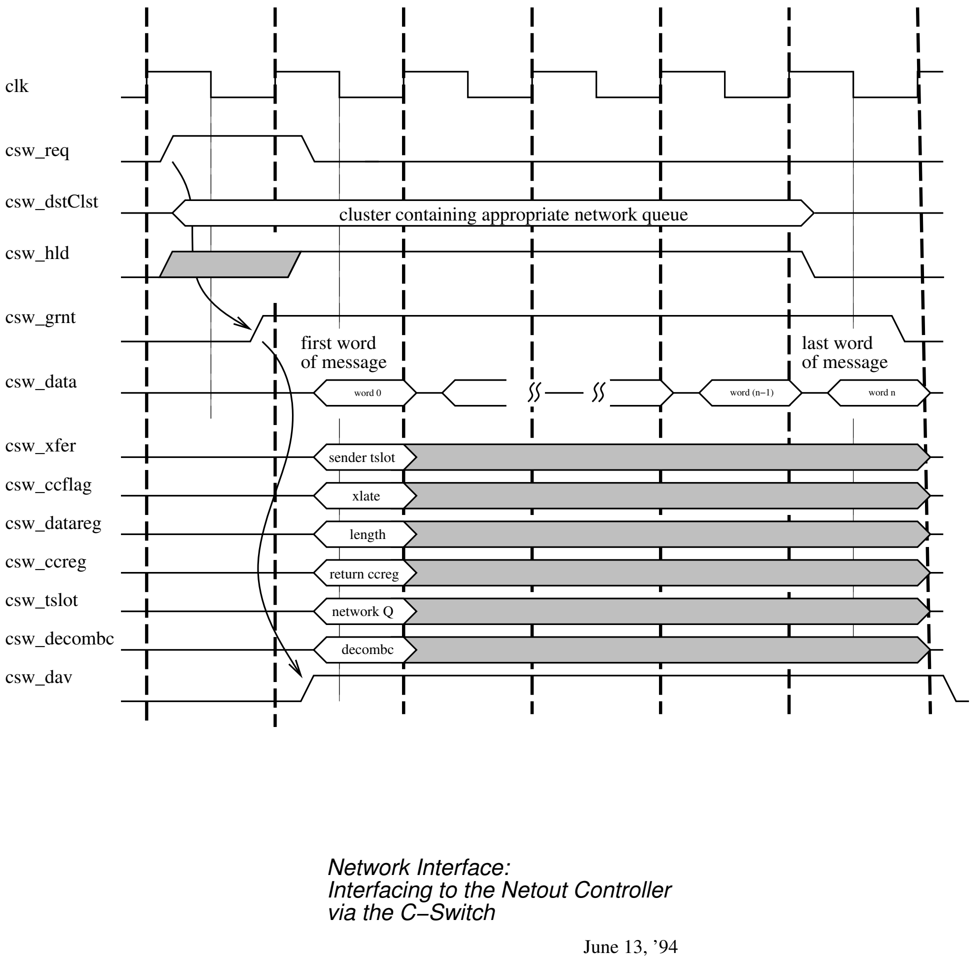


Figure 15.13: *C-Switch Transaction for SEND Operation*

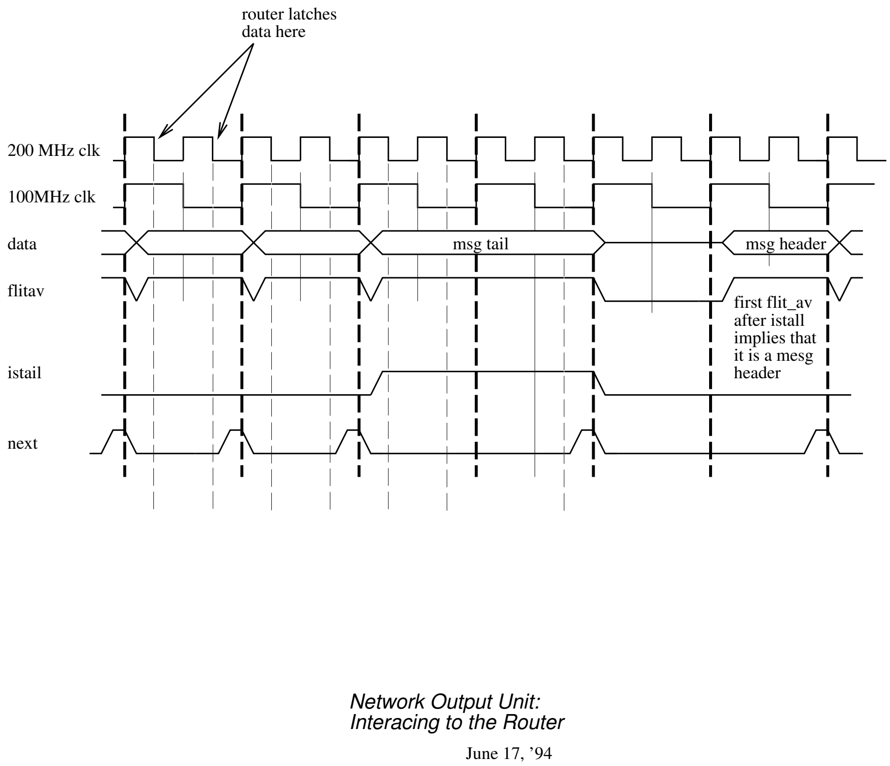


Figure 15.14: *Netout Unit – Router Interface*

- `rf`: indicates whether the message was caused by an ISEND or an FSEND. This information is needed for creating an event queue entry in the case of a GTLB miss.
- `MC count`: specifies the number of *MC* registers (in addition to *destination* and *dispatchIP*) to be transfered.
- `xlate`: set to 1 if *destination* is a virtual address (requires a GTLB translatoin), and 0 otherwise.
- `decombc`: asserted if the *Outgoing Message Buffer Count* is to be decreased by 1 for this SEND operation.
- `data`: data path.
- (more to come: netout writing back to cswitch for ccreg and event).

• `router`

- `data`: the netout unit drives data on these lines to be latched in by the router.
- `flitav`: asserted when the value on the `data` lines is valid.
- `ishead`: asserted when the `data` lines carry a message header flit.
- `istail`: asserted when the `data` lines carry a message tail flit.
- `next`: asserted when the router is ready for the next flit.

• `sendarbiter`

- `sendreq`: asserted when the SZ stage wants to issue a SEND operation.
- `throttle`: Asserted if the SEND operation should be subjected to throttling (*i.e.* not issued if OMBC = 0).
- `sendgrnt`: asserted if the SEND request is granted.

• `netqav`: asserted when the network resources are free to accept another message.

The above signals are subject to the constraints illustrated in Figures 15.12 through 15.14.

15.3.3 Architecture of the Network Output Unit

The network output unit is made up of 4 main blocks: the network output queue, the netout controller, the message header assembly module (including the GTLB module), and the event generator. The network output unit for one priority is shown in Figure 15.15.

The network output "queue" is implemented as a register file with 1 read port and 1 write port, and is 32 flits in size. In order for the network output unit to continue accepting the burst mode transfer from the C-Switch while assembling and enqueueing the message header, a 3-word FIFO is used to buffer the words from the C-Switch before they are put into the

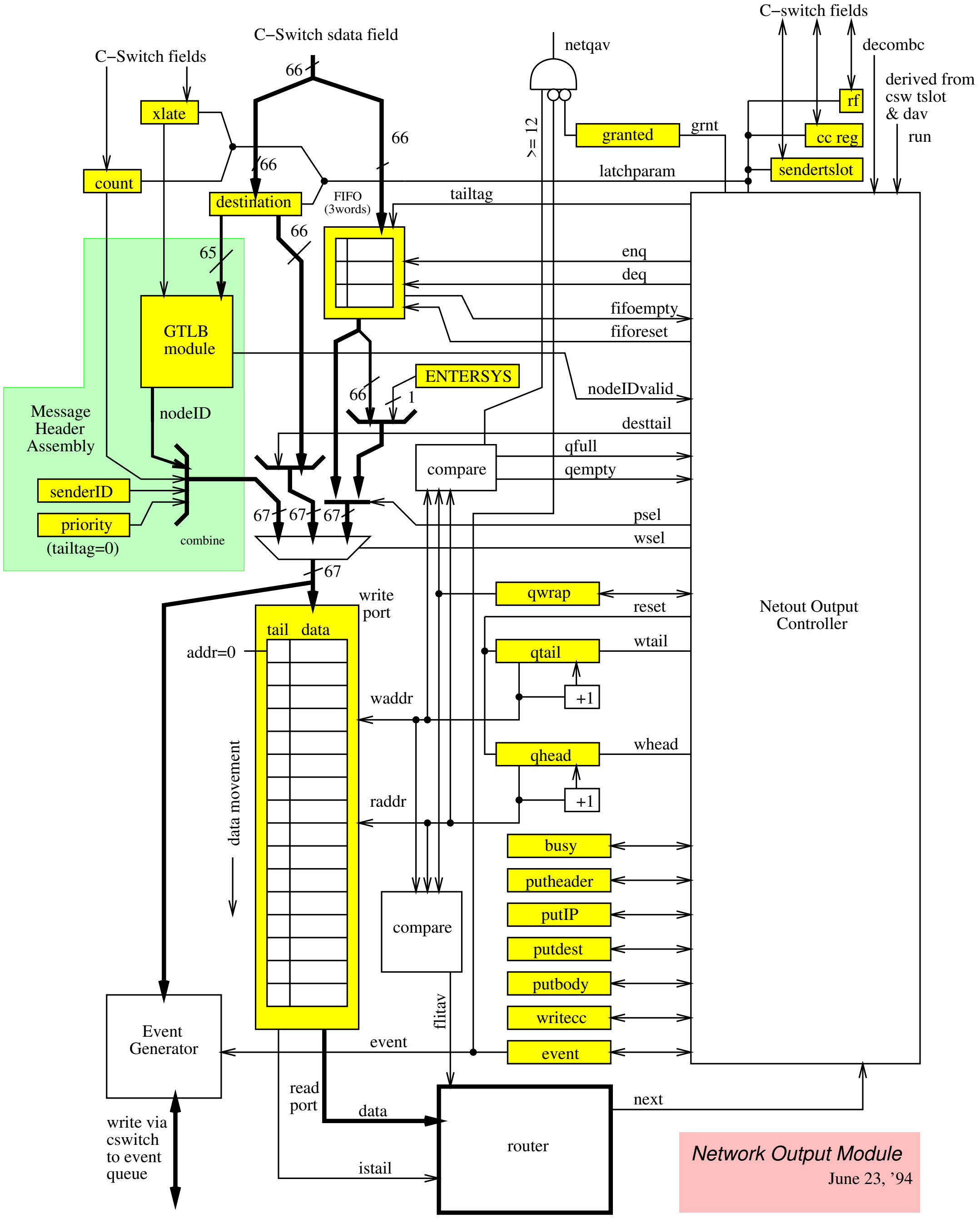


Figure 15.15: *Network Output Unit*

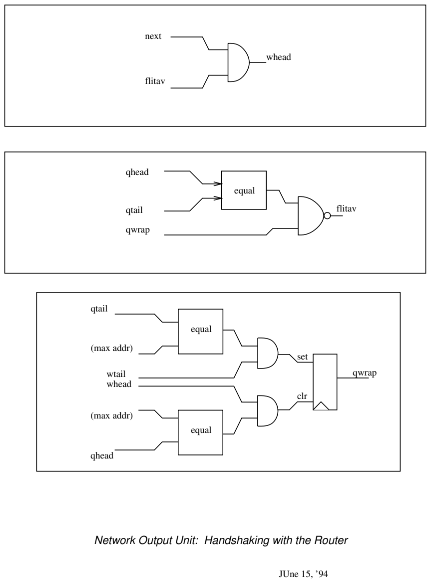


Figure 15.16: *Network Output Controller: Handshaking Logic With the Router*

output queue, allowing time for the GTLB translation to take place. The size of the 3-word FIFO is independent of the output queue length.

Note that the permission field of the *dispatchIP* is mutated into *ENTERSYS* before it is put into the output queue. This ensures that a user may not `JMP` to a *dispatchIP* locally, which is of permission type *MSGEXE* originally.

15.3.4 Operation of the Netout Controller

Similar to the netin unit, the netout controller is implemented as simple FSMs and combinational logic.

At the interface between the router and the netout unit, the logic blocks shown in Figure 15.16 implement the handshake functions.

At the enqueue end of the network output queue, the following actions take place when a message packet arrives from the C-Switch (`qhead` = `qtail` after initialization. The `tailtag` line is used to set the tail tag for the word being written into the queue):

- Cycle 0:
  - Latch in the parameters *destination*, *cc reg*, *xlate*, *count*, *rf* and *sendertslot*. The `latchparam` line also resets the 3-word FIFO.
  - Begin GTLB translation if `xlate` is asserted.
  - Decrement OMBC if `decombc` asserted.

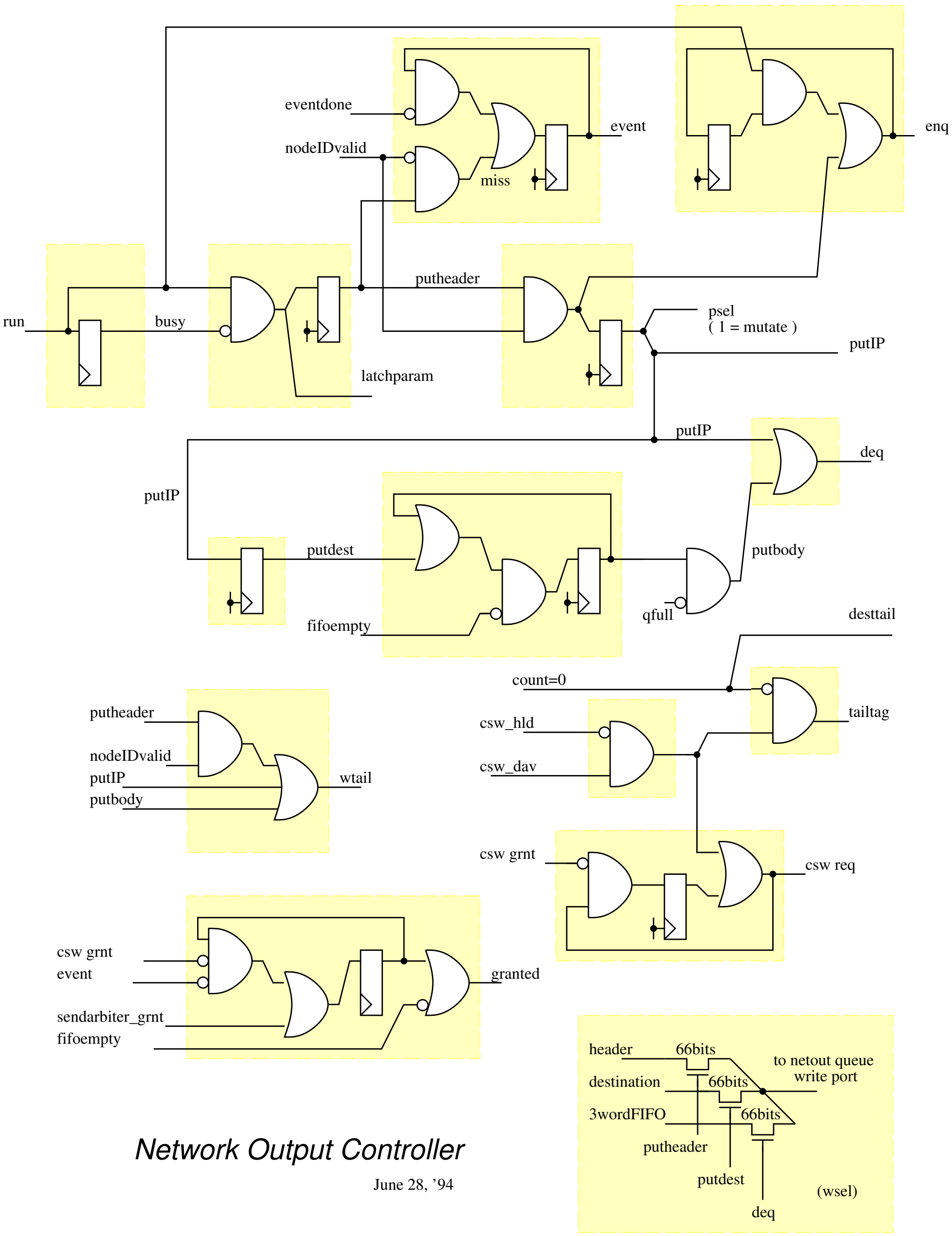
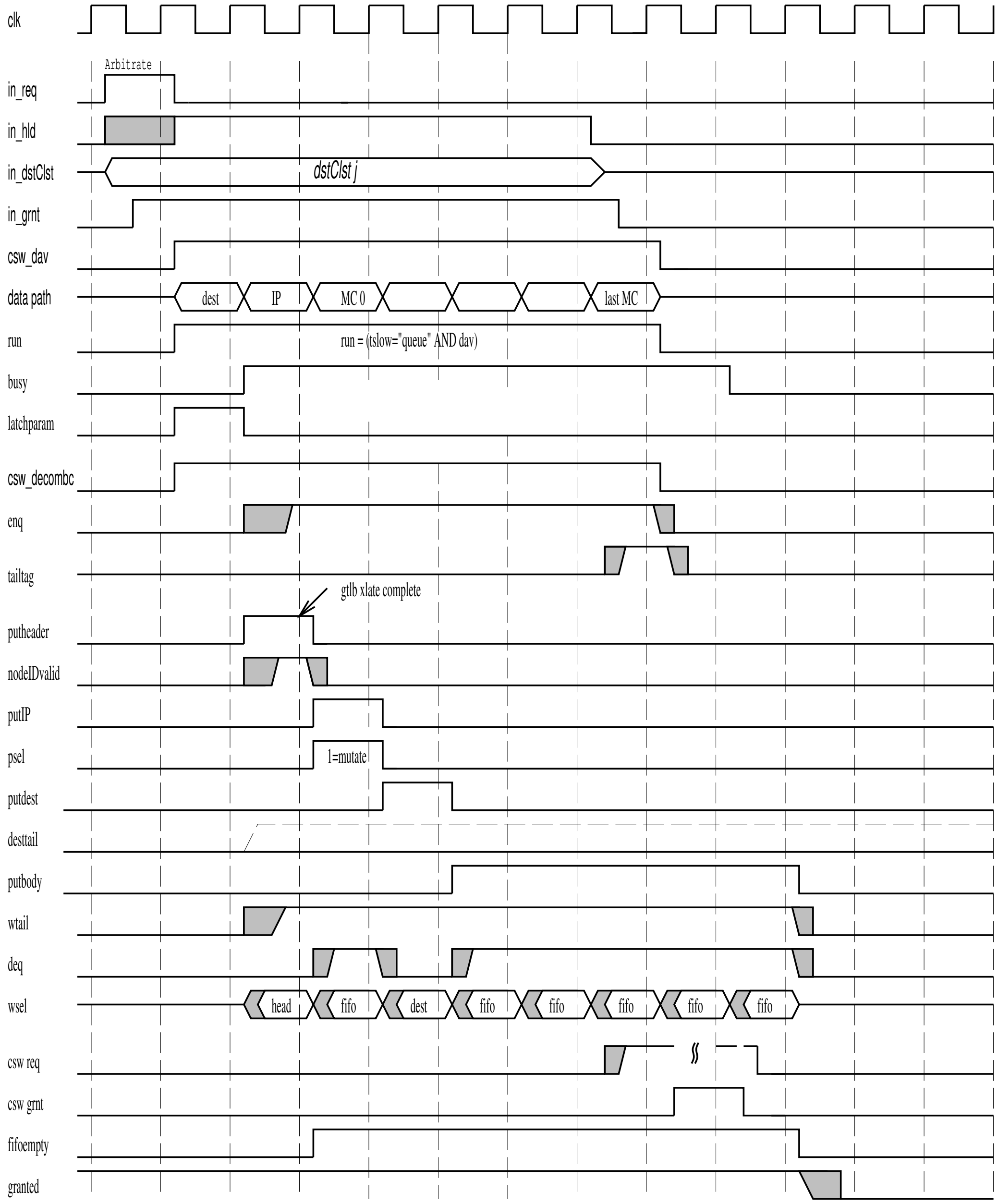


Figure 15.17: *Network Output Controller Logic*

- Cycle 1:
  - If `nodeIDvalid` (GTLB hit, or *destination* is a nodeID):
    - Buffer *dispatchIP* from C-Switch into the 3-word FIFO.
    - Write the assmbled message header into the network output queue.
  - If $\overline{\texttt{nodeIDvalid}}$ (GTLB missed):
    - Raise the event flag to initiate the event generator.
    - Ignore the rest of the message packet (and the rest of the followin steps)
- Cycle 2:
  - Move *dispatchIP* from the 3-word FIFO into the network output queue.
  - Buffer another word (if more words arriving) from the C-Switch into the 3-word FIFO.
- Cycle 3:
  - Enqueue the *destination* into the output queue.
  - Buffer another word (if more words arriving) from the C-Switch into the 3-word FIFO.
  - If *dispatchIP* was the last word in the packet, tag *destination* as a tail flit, assert `csw_req`, and skip to the final step (wait for `csw_grnt`).
- Until (including) the last word of the packet arrives:
  - Move one word from the 3-word FIFO into the output queue.
  - Buffer another word from the C-Switch into the 3-word FIFO.
- In last cycle of the C-Switch burst, assert *csw_req* to arbitrate for writing back to the sender's *CC* register.
- First cycle after the last word of the packet is in the 3-word FIFO:
  - Move one word from the 3-word FIFO into the output queue.
- Second cycle after the last word of the packet is in the 3-word FIFO:
  - Move the last word from the 3-word FIFO into the output queue, tag it as a tail flit.
- When `csw_grnt` is asserted:
  - Write to the sender's *CC* register, with the `sdone` line asserted in order to clear the *SEND bit* at the sender[1]
- When the *CC* write back completes, wait for 3-word FIFO to be empty. Then release the network resources and get ready for the next message.



Network Output Control Signals

June 27, '94

Figure 15.18: *An example of a C-Switch tranfer to the Netout Unit*

The above sequence is implemented in the logic shown in Figure 15.17, and an example of a complete transfer across the C-Switch to the network output controller is given in Figure 15.18. In this design, the 3-word FIFO is considered logically a part of the output queue in the accounting of queue vacancy (for determining `netqav`).

15.3.5 Operation of the Event Generator

The event generator is responsible for creating an event entry in the case of a GTLB miss. It uses the C-Switch to insert the entry into the event queue, and prepares the netout unit for acceptin new messages when it is done enqueuing the event. (TO BE DESIGNED)

15.3.6 Operation of the GTLB Module

The GTLB module contains the GTLB itself, plus some combinational logic which bypasses the *destination* register to the GTLB module output in the case where *destination* is a nodeID that does not require translation. The internal structure of the GTLB module and its timing requirements are shown in Figure 15.19.

15.3.7 Send Arbiter

Not shown in Figure 15.15 is the SEND arbiter, responsible for selecting one of up to 4 simultaneous SEND requests to be issued. A simple rotating priority arbiter is used, as shown inFigure 15.20. One SEND arbiter is used for each of the two message priorities.

15.4 Issues

- GTLB is accessible from config space via GPRB. Need to have some form of lock.
- Can SZ guarantee that SEND is issued if it is granted?
- How to send lines across nodes (with sync bit intact) ?
- optimizing the path from the register file to the netout unit.

[1] The *SEND bit* is set when a SEND operation issues. A thread must not be swapped out of its hardware slot if its *SEND bit* is set.

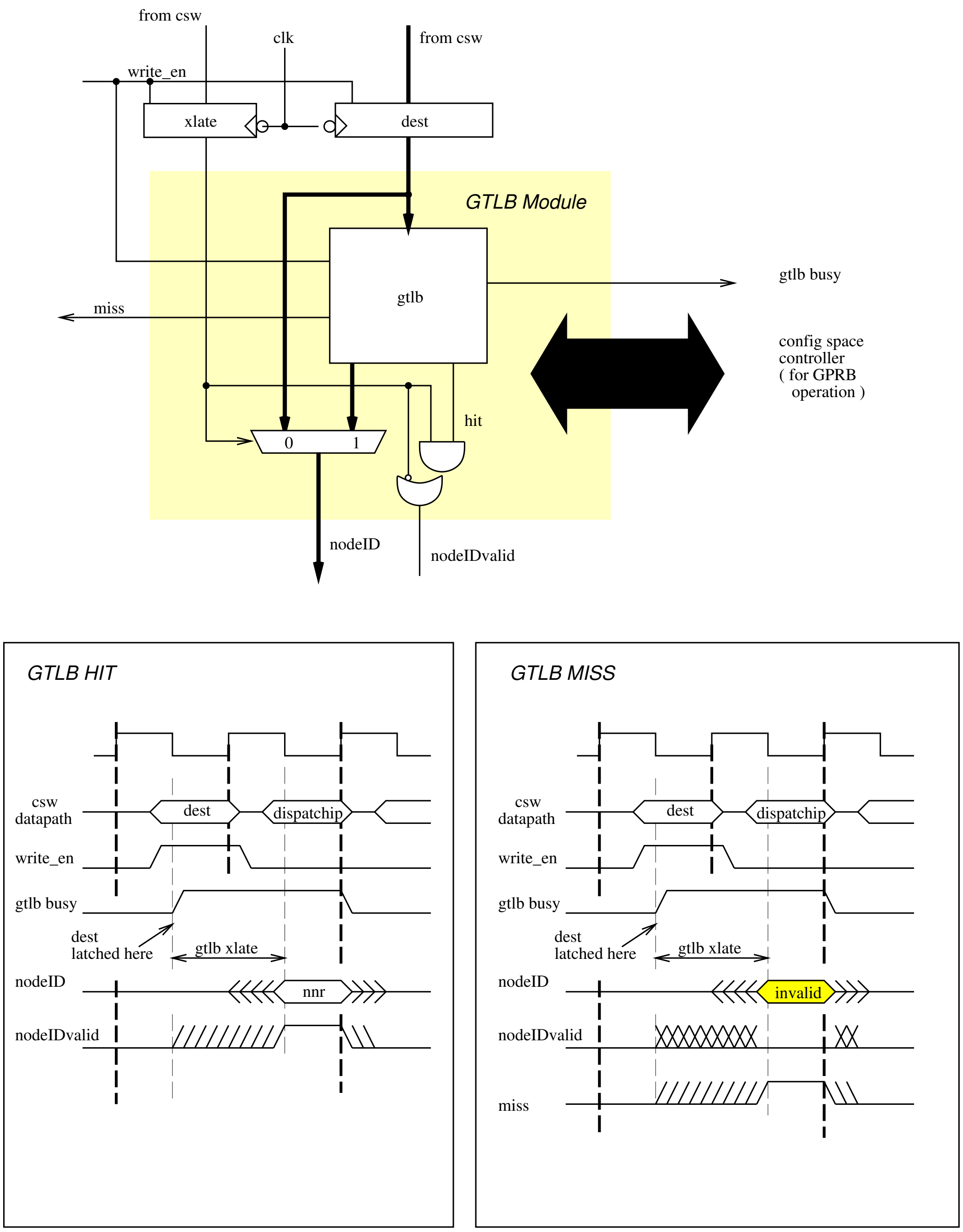


Figure 15.19: *GTLB Module*

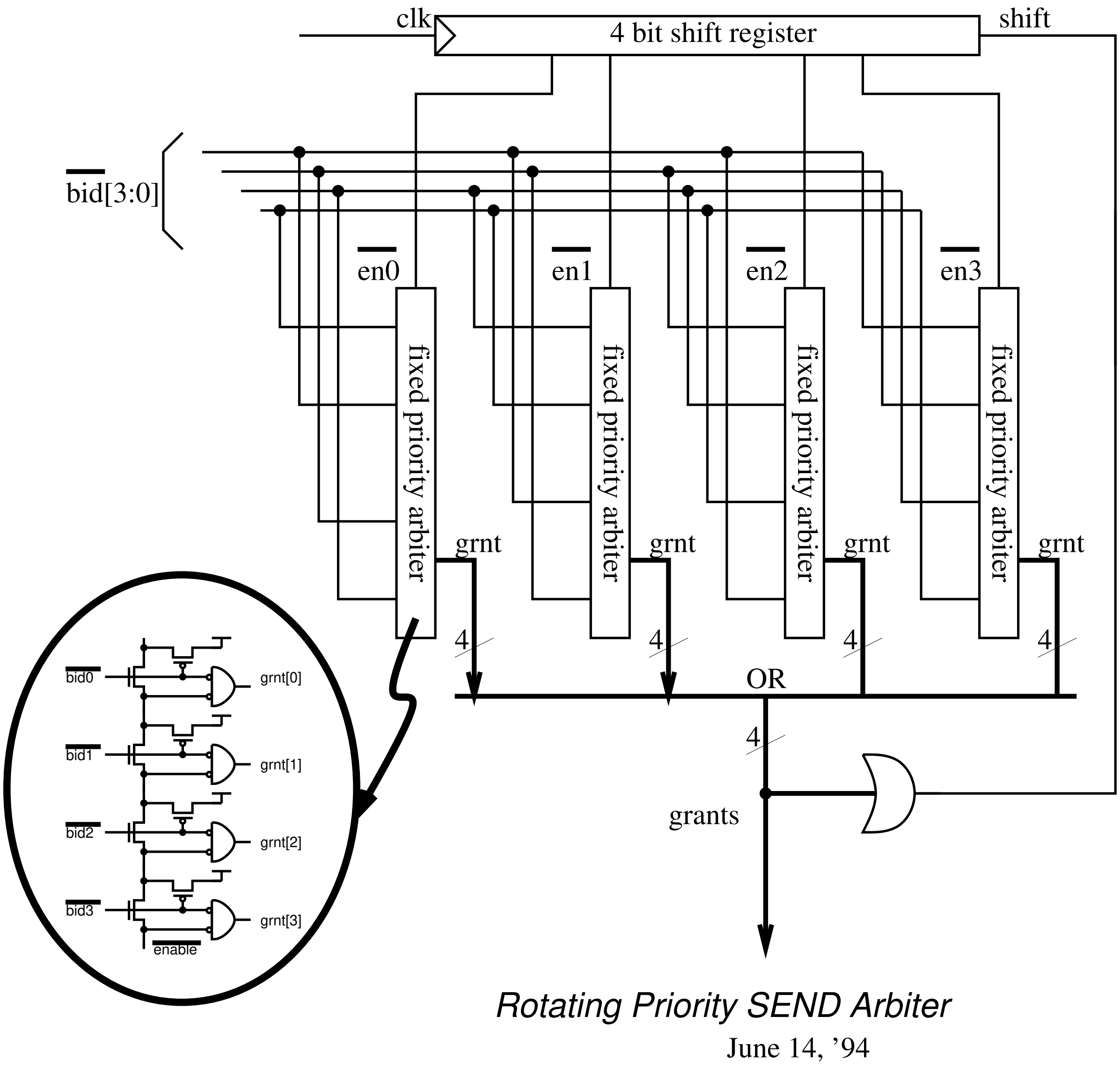


Figure 15.20: *SEND arbiter*

Chapter 16

The Event Queue Unit

16.1 Functional Overview

The event queue serves to buffer all events (except LTLB misses) in the MAP chip, to be resolved serially by the event handler in the System thread slot on cluster 0. It is logically and physically located in cluster 0 of the MAP chip, as shown in Figure 16.1.

16.2 Operation of the Event Queue

The event queue unit accepts event packets from the C-Switch via burst-mode transfer, and buffers them in a FIFO to be read out by the event handler via the *rbody* register. As each word is read from *rbody*, it is popped from the FIFO. Attempting to read *rbody* when the FIFO is empty stalls the operation, while operations (and multi-operation instructions) that contain two or more simultaneous reads to *rbody* have the same effect on the FIFO as a single read to *rbody*, although the return value is delivered to all the corresponding destination registers.

When the FIFO vacancy reaches a low *watermark*, the SZ stage is notified. The event unit then relies on the SZ stage to inhibit operations which may potentially cause more events until more vacancy becomes available in the FIFO. This guarantees that the event queue is able to absorb all events potentially outstanding in the system at any time.

Event packets may be sent by the following units:

- EMI
- configuration space controller
- cache banks
- clusters (including network output unit)

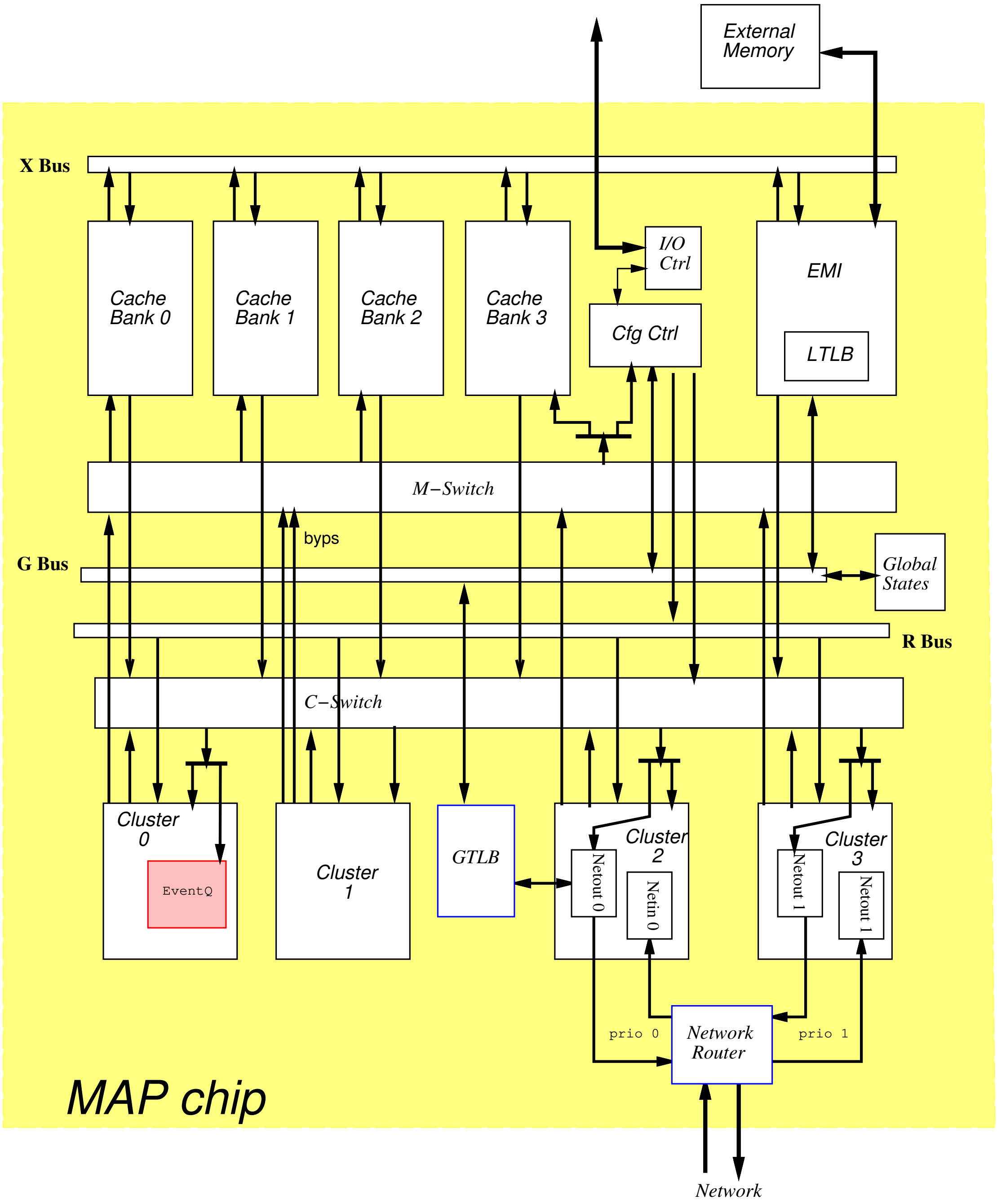


Figure 16.1: *The Event Queue within the MAP chip*

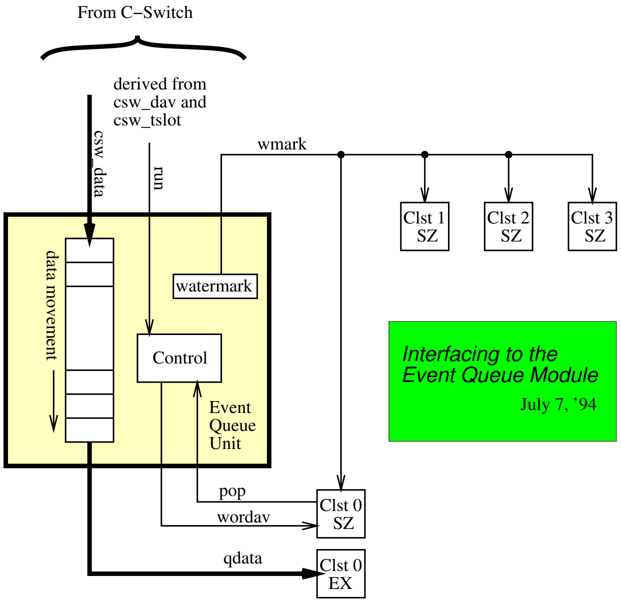


Figure 16.2: *Interfacing to the Event Queue Unit*

Note that there is no explicit arbitration for enqueuing entries into the event queue, except for the C-Switch arbitration. Since only one writer may access a given target on the C-Switch at any time, all requests to write the event queue are serialized at the C-Switch. The burst-mode transfer ensures atomicity for each event queue entry. It is expected that

16.3 Interfacing to the Event Queue Unit

The event queue unit communicates with the C-Switch, the SZ stage of all clusters, and the SZ/EX stage of cluster 0, as shown in Figure 16.2. The signals shown carry the following meanings:

- `csw`
  - `run`: Derived from `csw_dav` and `csw_tslot`, this signal is used to trigger the event queue unit into accepting a burst-mode event packet from the C-Switch.
  - `data`: The C-Switch data path.
- SZ
  - `wordav`: Asserted by the event queue unit when the SZ stage may issue a read to the *rbody* register without stalling. It also implies that the `qdata` lines (leading to the EX stage) actually carry a valid word.
  - `pop`: Asserted by the SZ stage to signal that a read to *rbody* was issued this cycle.

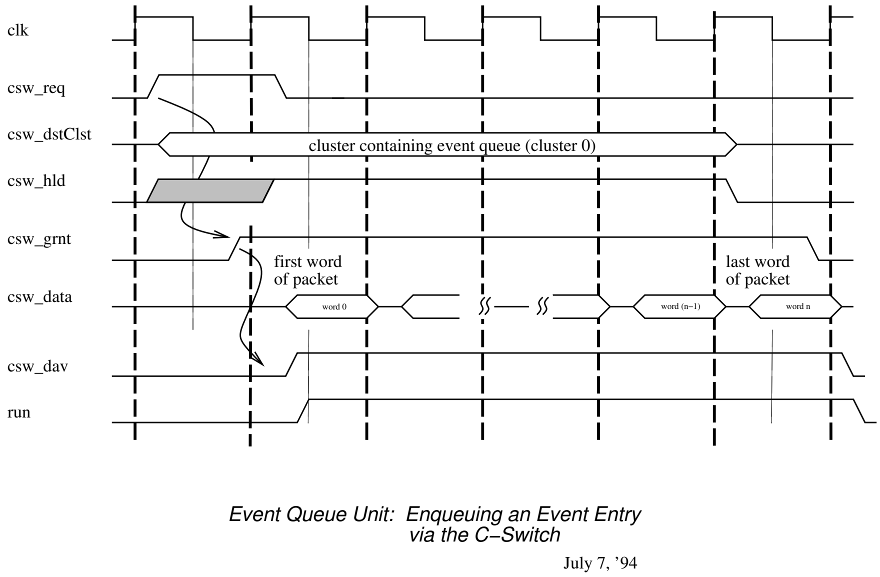


Figure 16.3: *Enqueueing an Event Entry*

- `wmark`: Asserted by the event queue unit when the vacancy in the event queue reaches a low *watermark*. When this line is asserted, the SZ stage must **not** issue any operations that *may* cause a new event.

• `EX`

- `qdata`: Carries the data word from the front of the FIFO event queue.

The expected handshake sequences are illustrated in Figures 16.3 and Figures 16.4.

16.4 Architecture of the Event Queue Unit

The event queue unit looks like a stripped-down network input/output interface unit. It is made up of two main blocks, *i.e.* a FIFO queue and a controller, as shown in Figure 16.5.

The queue is implemented as a 1-read/1-write port register file managed with the help of two *qhead* and *qtail* pointer registers, similar to the network interface units. A *qwrap* flag keeps track of the wrap-around status of the *qhead*/*qtail* pointers. A configuration-space-accessible *watermark* register determines when the vacancy in the queue is low enough to assert the `wmark` signal.

The event queue is 96 words in size, and *watermark* should be set such that `wmark` is asserted when the queue vacancy is 88 words or less.[1]

[1] Each event entry is at most 4 words long. There are at most 22 outstanding events in the system at any time: 1 in EMI,16 in the cache banks, 4 in the pipeline, 1 in netout (GTLB).

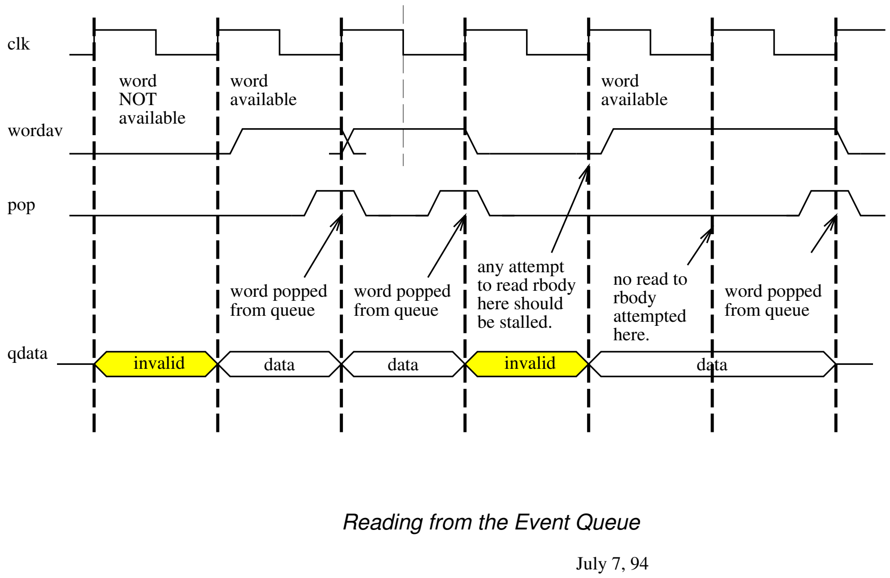


Figure 16.4: *Reading from the Event Queue*

The event queue controller is realized in combinational logic as shown in Figure 16.6.

16.5 Issues

- do we really need to reserve 88 words in the event queue when we assert `wmark`? any other alternative?
- should netout unit have higher priority in accessing event queue?

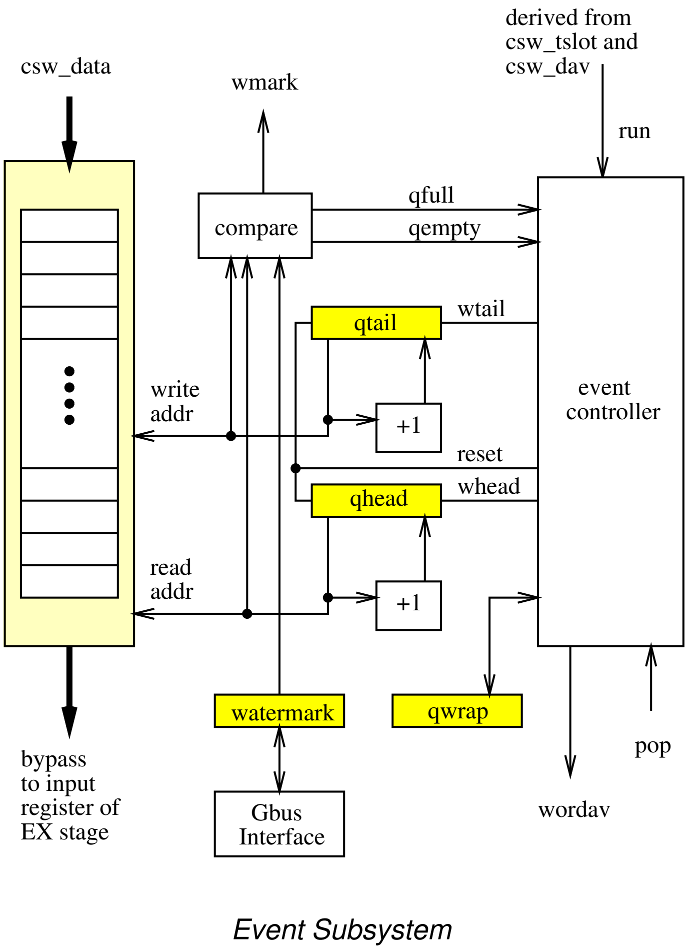


Figure 16.5: *The Event Queue Unit*

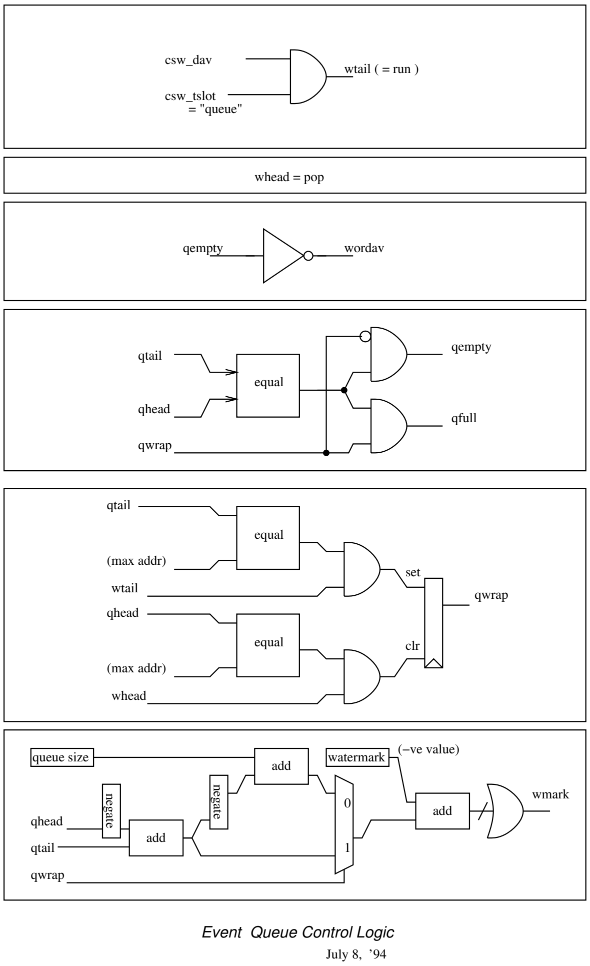


Figure 16.6: *The Event Queue Controller*

Chapter 17

The Configuration Space Controller

Chapter 18

The Diagnostic Interface

Chapter 19

Clocking Strategy

Chapter 20

System Design

Bibliography

[1] S. W. Keckler and W. J. Dally, "Processor coupling: Integrating compile time and run-time scheduling for parallelism," in *Proceedings of the 19th International Symposium on Computer Architecture*, (Queensland, Australia), pp. 202–213, ACM, May 1992.