Design of the Configuration and Diagnostic Units of the MAP Chip

by

Keith Klayman

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1997

Author . .....
Department of Electrical Engineering and Computer Science
May 27, 1997

Certified by .....
William J. Dally
Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Design of the Configuration and Diagnostic Units of the MAP Chip

by

Keith Klayman

Submitted to the Department of Electrical Engineering and Computer Science
on May 27, 1997, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

This thesis presents the design of the Configuration and Diagnostic units of the MAP chip. The MAP chip is a new microprocessor being developed for the M-Machine project. The Configuration units allow for the storage and access to various counters, thread state, and cluster data needed for thread swap. The diagnostics unit is responsible for booting the processor and is used as the interface between the external pins and the DIP and SCAN chains.

Thesis Supervisor: William J. Dally
Title: Professor

Acknowledgments

There are many people without whom this thesis would not be possible. First of all, I would like to thank Professor Bill Dally for giving me the opportunity to work in his group. The past year has been the most grueling but most rewarding time for me at MIT. I learned so much by working on this real world type project. Many thanks go to Steve Keckler and Andrew Chang for being so patient with me and all of my questions. Also, thank you to Nick Carter for the documentation on the Memory System Config, Albert Ma for his work on the I/O subsystem and Whay Lee for all his help with the switches.

Finally, I would like to thank my family. Without their support I could have never made it to MIT. I could never give you all the credit you deserve, but here is a little.

Contents

List of Figures

Chapter 1

Introduction

The M-Machine is an experimental multicomputer developed by the Concurrent VLSI Architecture Group in the Artificial Intelligence Lab at the Massachusetts Institute of Technology. As a member of the group, I participated in many facets of the project. My main contributions were made in the design and implementation of the Configuration and Diagnostic units. Work was also done to test these components at the behavioral level. This thesis presents the design of these units and discusses how the Configuration and Diagnostic units will be used when the MAP chip returns from fabrication.

1.1 Architectural Overview

The M-Machine consists of a collection of computing nodes interconnected by a bidirectional 2-D mesh network. Each node will consists of a multi-ALU (MAP) chip and 8 MBytes of synchronous DRAM [2]. The MAP chip includes a network interface and router which provide low latency communication in the X and Y directions. A dedicated I/O bus is also available on each node.

As shown in figure 1-1, a MAP contains: three execution clusters, a memory subsystem comprised of two cache banks and an external memory interface, and a communication subsystem consisting of the network interfaces and the router. Two crossbar switches interconnect these components.

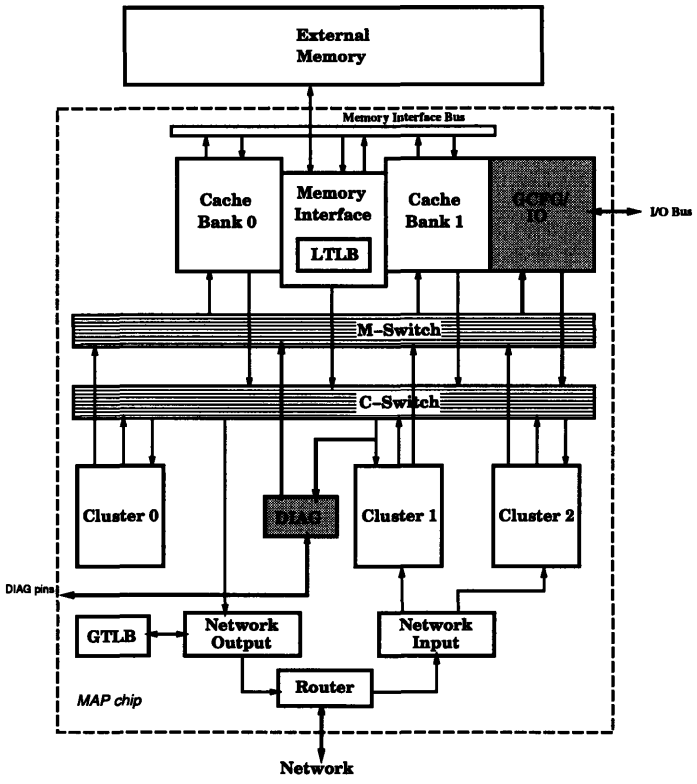

Figure 1-1: *The MAP chip*

The M-Machine provides hardware that allows many threads to run concurrently on a node. There are enough resources for five V-Threads, each of which consists of three H-Threads, one H-Thread per cluster. Two of the five V-Threads are user threads, while the other three are for handling exceptions, events and the system runtime software. H-Threads can communicate through registers, with each having the capability to write values to the register files of the other H-Threads within the same V-Thread. The H-Threads in a V-Thread can execute as independent threads with different control flows for exploiting loop or thread-level parallelism. On the other hand, the H-Threads can be scheduled statically as a unit to exploit instruction level parallelism, as in a VLIW machine. The V-Threads are interleaved at run-time over the clusters on each cycle. See Figure 1-2 for an illustration of the relationship between V-Threads and H-Threads.

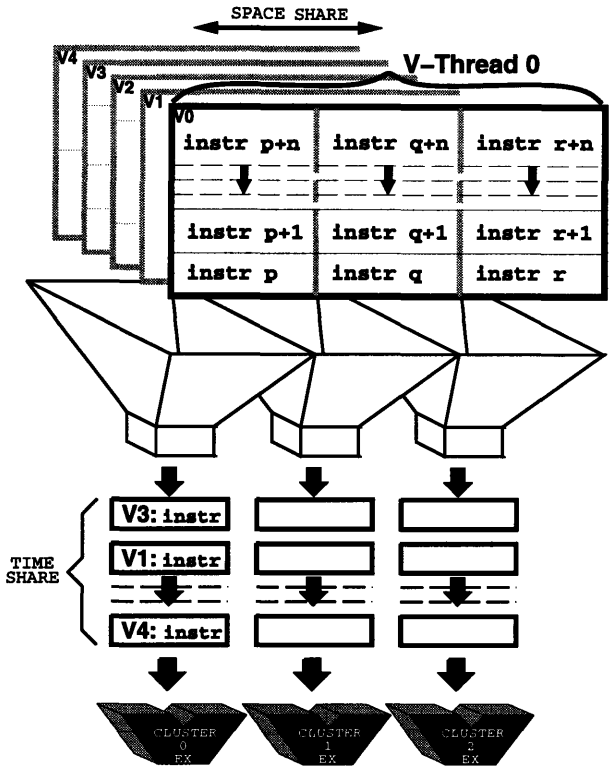

Figure 1-2: Relationship of V-Threads to H-Threads

1.2 Switches

The two crossbar switches provide single cycle latency communication between all units of the MAP chip. The M-Switch (MSW) provides a means for the clusters to request data and instructions from the memory system [5]. The C-Switch (CSW) is used by the memory system and queues to write to any cluster. In addition, the CSW provides a path between the clusters [6].

The switches also accommodate the configuration and diagnostic units. Both units require the ability to read and modify distributed pieces of state throughout the MAP chip. The switches provide efficient access to this state that reduces the wiring required in alternative solutions. The Global configuration unit looks much like a cache bank to the switches. It consumes data from the MSW and produces over the CSW. The Diagnostic unit requires the ability to send over the MSW and

listens on the CSW output port of Cluster 1.

1.3 Global Configuration Unit

A Global Configuration Unit is necessary to facilitate the management of hardware located throughout the MAP chip. A central unit is required due to the thread architecture of the MAP. It is expected that a single thread will act as the run time system, it is essential that this thread have the ability to access all state throughout the MAP.

Expected primary uses of the Global Config include thread swap, performance metering, filling registers on software handled loads, I/O, LTLB and Network access. By controlling these functions from a central location and sharing the switch interfaces we do not waste routing resources. In addition, we conveniently use the load and store operations to address the Configuration address space. This enables the run time system to efficiently access the hardware.

1.4 Diagnostic Unit

The MAP diagnostic unit is required for two reasons:

1. System initialization and reset (warm and cold resets). This includes:

   - Whole chip reset - ie. clearing, queues etc..
   - Setting "DIP" switches
   - Initializing off chip memory
   - Loading boot strap program into memory

2. Ability to test control and data functionality.

   This implies that all control NLTCH and PDFF cells can be isolated and tested for functionality. The main purpose of this level of testing is to ensure logic works as designed.

The verification function of the diagnostic system is based on the following assumptions:

- The diagnostic system will be employed to isolate logic errors so that they may be more fully explored in the RTL simulator.
- The diagnostic system will not be used to detect and isolate manufacturing bugs (ie. shorts, opens).
- Application program debug will be performed via breakpoints (implies ability to access data without effecting threads.)

1.5 Thesis Outline

This thesis will overview the design of the Configuration and Diagnostic units. This will include trade-offs made throughout the design and the information needed to design an interface. The next chapter will detail the Global Configuration unit. Following that chapter will be a discussion of the requirements for thread swap on the MAP chip. Then a chapter on the design of the Diagnostic unit. This chapter will include a detailed description of the timing and steps required to boot the MAP chip. The final chapter serves as the conclusion to this thesis. Appendix A provides a reference for Event addressing while Appendix B does the same for composing GCFG addresses. Appendix C provides sample code used by the runtime system for thread swap.

Chapter 2

Global Configuration Unit

2.1 Overview

The configuration space controller is divided into several pieces and is distributed throughout the MAP. A global configuration space controller (GCFG) resides in a central location. It receives all configuration space requests via the M-Switch and either takes the appropriate action or forwards the request to a local configuration space controller. A local configuration space controller (LCFG) resides within each cluster. It is responsible for all state modifications within that cluster. Requests are sent from the GCFG to the LCFG via the C-Switch (CSW).

The GCFG shares an M-Switch data port with Cache bank one and has it's own ready and data available signals. When the GCFG receives data from the MSW it decodes the request based on the address provided, and does one of the following:

- Write a piece of state located in the GCFG
- Read a piece of state located in the GCFG, returning value across the CSW
- Forwards the request to a cluster configuration space controller via the CSW
- Generates a generic CSW command
- Accesses the Network state via a generic CSW command

- Access the I/O controller

The various components of the GCFG are accessed by an extremely large address space that is sparsely populated by physical resources. The GCFG decode module uses the address from and MSW packet to target the destination of the request.

The GCFG contains a top wrapper that includes a state machine and decode module along with five major modules that manipulate data: MSW interface, CSW interface, Counters (cycle and event), I/O and GCFG state registers (HACT, HRUN, VRUN, TC, User, Event). Figure 2-1 shows the block diagram of the GCFG. Data flows from the MSW and either writes to the various modules or reads a value and is held in the CSW module until arbitration is won.

The MSW module contains a set of recirculating PDFF cells that load when empty and the MSW asserts the data available signal. Data is held in the PDFF's until it is consumed by the CSW or terminates in the other modules. The CSW interface performs the multiplexing of the data and arbitrates with the CSW, netout and LCFG. The data packet is held in a recirculating NLTCH while arbitrating for the CSW. The cycle and event counters, 64 and 32 bits respectively, are implemented in eight bit chunks with a fast carry. The I/O module consumes data from the MSW interface and produces to the CSW an arbitrary time later.

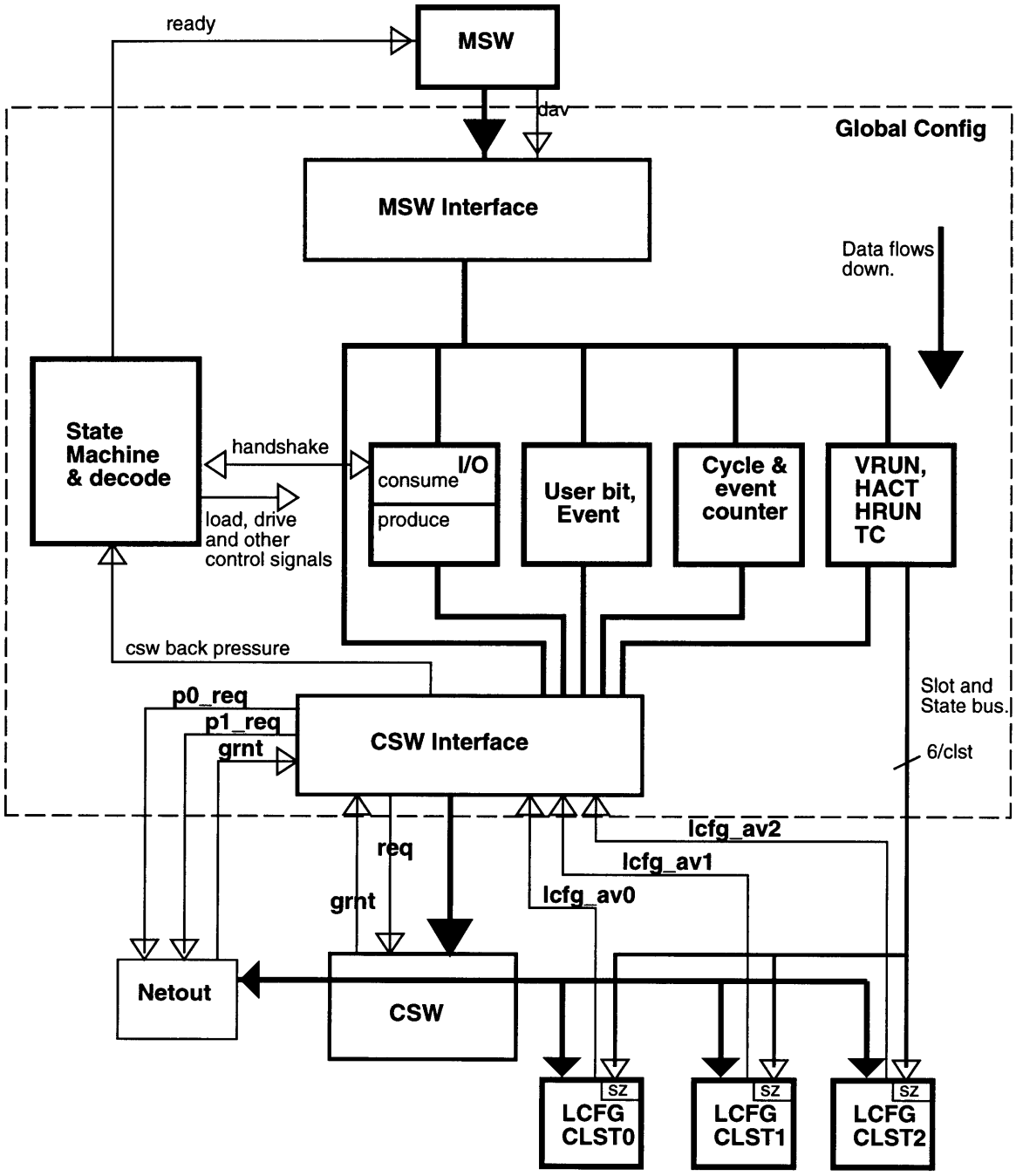

Figure 2-1: GCFG Block Diagram

2.2 Addressing

The Configuration Space is accessed by software loads and stores to pointers tagged with the configspace type as shown below.

Pntr Type	Seg Length	Address
1100	X	Config Address
4	6	54

The large address space of the Configuration unit allows the hardware to quickly decode which partition to send a request. These partitions are encoded in a one-hot fashion at the top of a Config address as shown in Figure 2-2.

~mem	**csw**	**gbl**	**clst**	**I/O**	**other**	**address**	**X**
1	**1**	**1**	**1**	**1**	**22**	**24**	**3**

Figure 2-2: Configuration space address bits.

The five bits (`~mem, csw, global, clst, i/o`) identify where the access must be serviced within the global configuration space controller. If the *mem* bit is set to **zero** then the access is intended for the Memory System. These transactions are sent over the M-Switch and through the bank without disturbing the GCFG. Only one of the other four bits should be active in any GCFG address. The address field identifies a piece of hardware for the selected destination of the access.

2.2.1 LCFG

LCFG state is addressable by setting the `clst` bit at the top of the address field. LCFG addresses will consist only of the low 11 bits of the address field. The top two bits of this 11 bit field will specify the relative cluster. When accessing Local state that is duplicated for each Vthread the next three bits will specify the thread and

Thread Slot Data	LT offset	Access Mode
Integer Registers	0x0 - 0x78	Read/Write
FP Registers	0x80 - 0xF8	Read/Write
HThread IPs	Mapped into GPRs	Read/Write
CC Registers	0x100 - 0x138	Read/Write
CC Registers at once	0x140	Read Only
MEMBAR Counter	0x148	Read/Write
HThread Init	0x150	Write Only
HThread Priority	0x158	Read/Write
Preempt/Stall Counter	0x160	Read/Clear
Preempt Threshold Register	0x168	Read/Write
Stall Cycle Bit	0x178	Read/Clear
Integer Scoreboard	0x188	Read/Write
FP Scoreboard	0x190	Read/Write
CC Scoreboard	0x198	Read/Write
Thread Restart IP Write	0x1A0	Write Only
HFork IP	0x1A8	Write Only
Thread Restart IP Read	0x1B0	Read Only
Branch Pending bit	0x1C0	Read Only

Table 2.1: LT offsets

Thread Slot Data	LC offset	Access Mode
LCFG NOP	0x1D8	Write Only
Exception bit	0x1F0	Read/Clear
Catastrophic Exception Bit	0x1F8	Read

Table 2.2: LC offsets

the low six bits will specify the offset as shown in Table 2.1. The low six bits, LT offset, target a specific piece of state within a thread. The per Cluster LCFG state is addressable by the 9 bit address given in Table 2.2. These addresses, LC offset, target the exception information and thread independent NOP.

Below is the format for accesses to the per thread state.

1	0	0	1	0	X	cid	tid	LT offset	X
1	1	1	1	1	35	2	3	6	3

Below is the format for accesses to the cluster state.

1	0	0	1	0	X	cid	111	LC offset	X
1	1	1	1	1	35	2	3	6	3

Reading Cluster State

Reads to state located in the cluster are forwarded to the LCFG via the CSW. This request includes the address of the state to read, as well as the name of the destination register (the LSB of the register ID is place in the MEM Sync bit). Refer to figure 2-3 for the organization of the CSW packet sent to the LCFG.

Writing Special Registers

Like reading state, writing register requests are forwarded to the LCFG. The GCFG sends a command of the same format to the LCFG with the 'Load Destination' field unused.

Writing General Registers

General registers are written directly from the GCFG across the CSW without intervention from the LCFG. Both data and CC registers are written directly.

Handshake		*Load Destination*					*Address*			*Data*	
dav	hld	rf	Reg [3:1]	clstid	slotid	Tslot	Store	slot	LCFG Addr	Mem sync	Data word
1	1	1	3	2	3	3	1	3	6	1	65 bits
MSB										LSB	

Figure 2-3: *Partition of CSW Packet to LCFG*

2.2.2 Global State

Global state is addressable by setting the `gbl` bit at the top of the address field. The state is then accessible by setting the offset to that given in Table 2.3. TC Mode, VThread run, HThread run and HThread active each have six addressable locations, one per thread. These six locations are addressable by using the low six bits of the address as a one-hot field specifying the desired thread. Each VThread run and TC Mode location are one bit, the HThread Run and HThread Active are 3 bits wide

Thread Slot Data	GBL Offset	Access Mode
VThread run	0x208 - 0x300	Read/Write
HThread Run	0x408 - 0x500	Read/Write
HThread Active	0x808 - 0x900	Read/Write
User Enable	0x1000	Read/Write
Cycle counter	0x2000	Read/Clear
Event counter	0x4000	Read/Clear
Event register	0x8000	Read/Write
Echo CLST and THREAD	0x10000	Read
TC Mode	0x20008 - 0x20100	Read/Write

Table 2.3: Global state offsets (Local access).

(one bit per cluster.) User Enable is one bit and Event Register is seven bits. A write to the cycle (64 bits) and event (32 bits) counters will clear them. When echoing the CLST and THRD ID they will be returned in the low five bits of the result respectively.

Below is the format for accesses to the global state.

1	0	1	0	0	X	GBL offset	X
1	1	1	1	1	32	14	3

2.2.3 CSW Transactions and Network State

CSW transactions are generated by setting the `csw` bit at the top of the address field. The low 22 bits are placed directly into the 22 bits of CSW control and the next two bits are the cluster destination. The Network state is accessible by generating the correct CSW transaction. The GCFG recognizes the destination is the NETOUT unit and requests access to the NETOUT in addition to the CSW bus [4].

Below is the format for generating generic CSW transactions. The structure of a generic CSW packet is shown in figure 2-4.

1	1	0	0	0	X	dst cid	CSW control	X
1	1	1	1	1	22	2	22	3

Control		*Source*		*Destination*			*Condition*	
mf	Xfr type	sclstid	sslotid	Dst Tslot	rf	Data reg	CC flag	CC reg
1	3	2	3	3	1	4	1	4
MSB								LSB

Figure 2-4: *Partition of Generic CSW Control Packet*

2.2.4 I/O Addressing

Figure 2-5 shows the structure of I/O transactions. Two bits encode the address length in packets, and 2 bits encode the data length in packets. An address length of 2'b11 indicates a burst load or store [7].

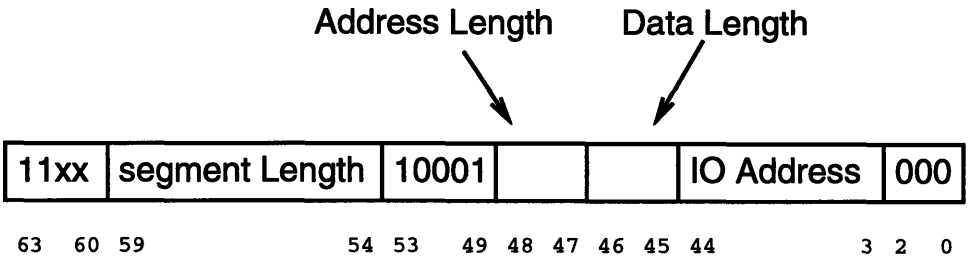

Figure 2-5: I/O Address fields

2.2.5 Memory System Configuration Space Addressing

The memory system's configuration space state is accessed through config space pointers which have their high bit set to zero, which causes the M-Switch to route requests based on these pointers to the memory system to be handled. The configuration space state in the memory system consists of the contents of the LTLB, which may be read and written, and four write-only registers which control the locking of the ltlb during an LTLB miss, the error correction code (ECC) circuitry, and whether or not blocks in the cache are marked invalid on a block status miss. Attempts to read the write-only registers have unspecified results. Table 2.4 shows the addresses allocated to each of these regions. Note that these address values include the high 10

bits of the pointer, which indicates that the pointer refers to configuration space. As shown, the memory system ignores bits 28-59 of configuration space addresses when determining which piece of state is being accessed.

Memory State	Address	Access
LTLB	0xC000000000000000–0xC000000000FFFFF8	MSW (r/w)
Clear Lock	0xC000000001000000	MSW (write)
Discard Req	0xC000000002000000	MSW (write)
ECC Control	0xC000000004000000	MSW (write)
Inv. on Miss	0xC000000008000000	MSW (write)

Table 2.4: Memory system configuration space address allocation

2.2.6 Performance Monitoring

Performance monitoring uses the event register and event counter. The event register (7 bits) selects a particular event type. These seven wires are run to all event generating sites. A single bit runs through the event generating sites through a mux and PDFF at each site. This bit enters the GCFG as `event_inc` and the event counter is incremented for every cycle it is high. The event register is read and writable while the event counter can be read and is cleared on a write. For a list of the events and their addresses refer to appendix A.

This distributed system is quite simple, but is not without it's problems. Specifically, it is not possible to get a precise count of an event. There will always be on the order of 10 cycles at the beginning and end of your count which you will not have control of. This is acceptable since the expected use of the event system will be over extremely long stretches of code.

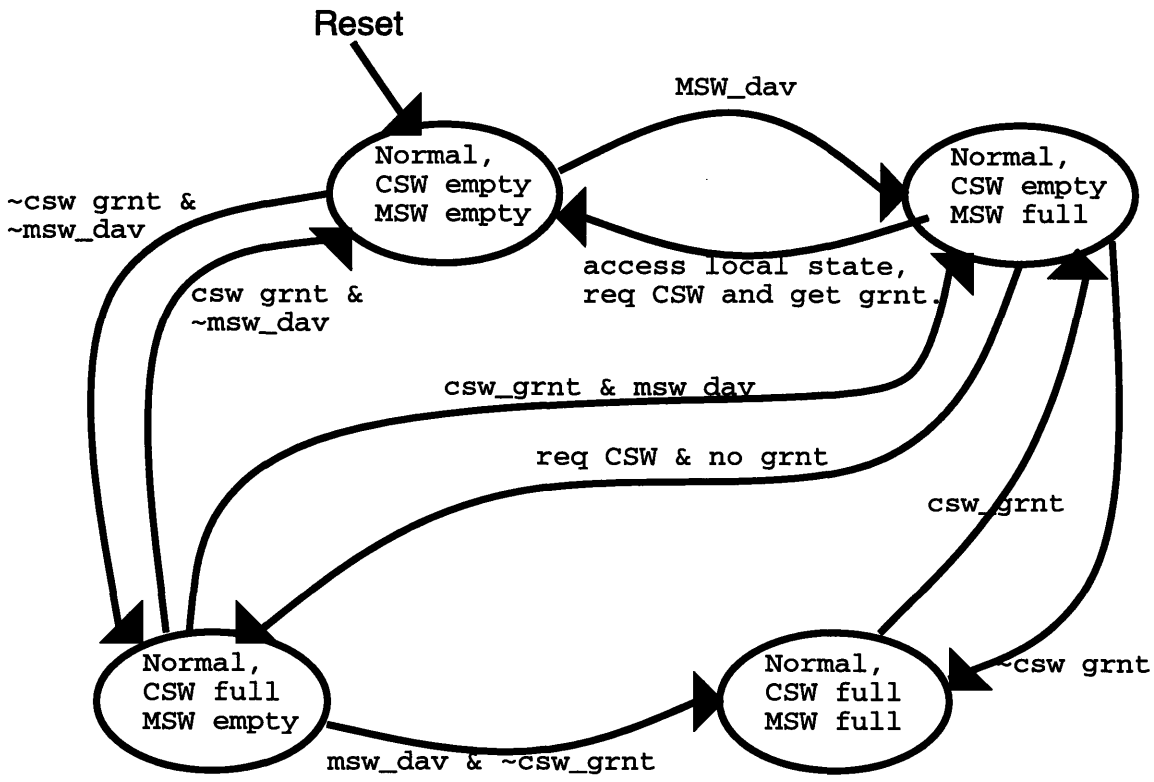

Figure 2-6: GCFG State Machine diagram

2.3 Global Configuration Space Control

2.3.1 GCFG State Machine

The GCFG is designed to minimize the back pressure placed on the MSW during high usage periods and provide minimal latency for infrequent usage. Minimal back pressure on the MSW is desirable for two reason: this pressure could cause threads in the cluster to stall and it can allow I/O access while waiting on the CSW.

With this in mind the GCFG is pipelined to hold two accesses. When a request is pulled off of the MSW it is held in a set of recirculating PDFF's. When it is necessary to send data out via the CSW the packet is formed and subsequently held in a set of recirculating NLTCH's. The four combinations of these two states being full or empty provide the first four states of the state machine shown in Figure 2-6.

A consequence of the pipelining is that it is possible for GCFG requests to complete

out of order. This can occur when the first request is waiting for a CSW grant when a second request enters from the MSW and updates state within the GCFG. In most cases this is acceptable behavior. However, it can be avoided by sending a Configspace write to a register I0. This ensures that all requests before register write will leave the GCFG before any following requests update state.

2.3.2 MSW and CSW interface

The MSW interface is simply a recirculating PDFF with a load based on current and next state. If the PDFF's are currently empty or will be empty on the next state they are loaded. The CSW interface on the other hand requires two levels of muxing and additional logic for the csw_req signal.

The first level of muxes in the CSW interface compiles the packet from the GCFG. The second level takes that packet, I/O packet and the recirculation path to feed the NLTCH's that drive to the CSW.

The request logic is responsible for conforming to the normal CSW timing and also requesting the NETOUT unit or LCFG when appropriate. In order to reduce delay, a normal CSW operation is assumed, in the case where the destination is not available but the CSW is granted, late_kill signal is used to send an abort packet. The late_kill can be used when the available signal from the LCFG (`lcfg_av`) is not high or the netout unit does not grant.

2.3.3 Decode and Global State

The decode in the GCFG enables the various write signals and muxes. It decodes the command and address in the MSW PDFF registers and asserts the correct signals to write or read local state. These signals will be held until the data leaves the MSW registers.

The accessible state residing within the cluster controller is listed in Table 2.3. Most state is held in PDFF's and is read through a series of muxes that lead to the CSW interface.

The HACT, HRUN, VRUN and TC data are held in NDFF's and thus have a redundant pltch in the path. However, this is necessary to satisfy the timing to the cluster. All writes to these bits must be conveyed to the cluster SZ stages. This is facilitated by a three bit thread ID and a two bit state bus between the GCFG and each cluster. The thread ID will be set to 3'b111 in the idle case. When it identifies a thread, the top bit of state will specify whether TC Mode should be active and the low bit of state is one if HRUN, HACT and VRUN are all one. This transaction occurs on the second half of the cycle.

2.3.4 I/O subsystem

In addition to the normal flow of data, the I/O subsystem can consume data from the PDFF's off the MSW and provide data for the NLTCH's out to the CSW. A handshaking protocol for both of these exchanges has been implemented. Whenever the NLTCH's in the CSW interface are available, the I/O has priority over GCFG data.

2.4 Timing

Figure 2-7 shows the round trip timing from M-Switch to C-Switch through the GCFG. The GCFG is constrained by the requirements of the MSW and CSW. It must read data from the MSW with an NLTCH and drive data to the CSW with an NLTCH. The PLTCH required to complete the path is added to the input NLTCH to form a PDFF. This leaves a full cycle (minus setup time for an NLTCH) from the PDFF load until data must be ready for the NLTCH.

The top half of figure 2-7 shows the GCFG interaction with the MSW arbiter. `Ready` is provided by the GCFG to the MSW, `msw_dav` is is an input to the GCFG from the MSW, and `msw_in_data` is driven on the MSW bus to the GCFG. The lower half of the figure shows that `csw_req` and `csw_dst` are derived quickly inside the GCFG and sent to the CSW arbiter.

When a request comes in from the M-Switch, the decoder quickly determines if the C-Switch is required and immediately begins arbitration for it. The data from global state reads is delivered across the C-Switch on the subsequent cycle or when arbitration is won. The input and output latches allow the next request to enter the GCFG even if C-Switch arbitration is lost.

In the event CSW arbitration is won, however the NETOUT or LCFG resource is not available, an abort packet is sent while the data remains in the CSW interface module. Arbitration for the CSW is suspended until the resource is available. Figure 2-8 shows the case when late_kill causes an abort packet to be sent and CSW arbitration to wait one cycle.

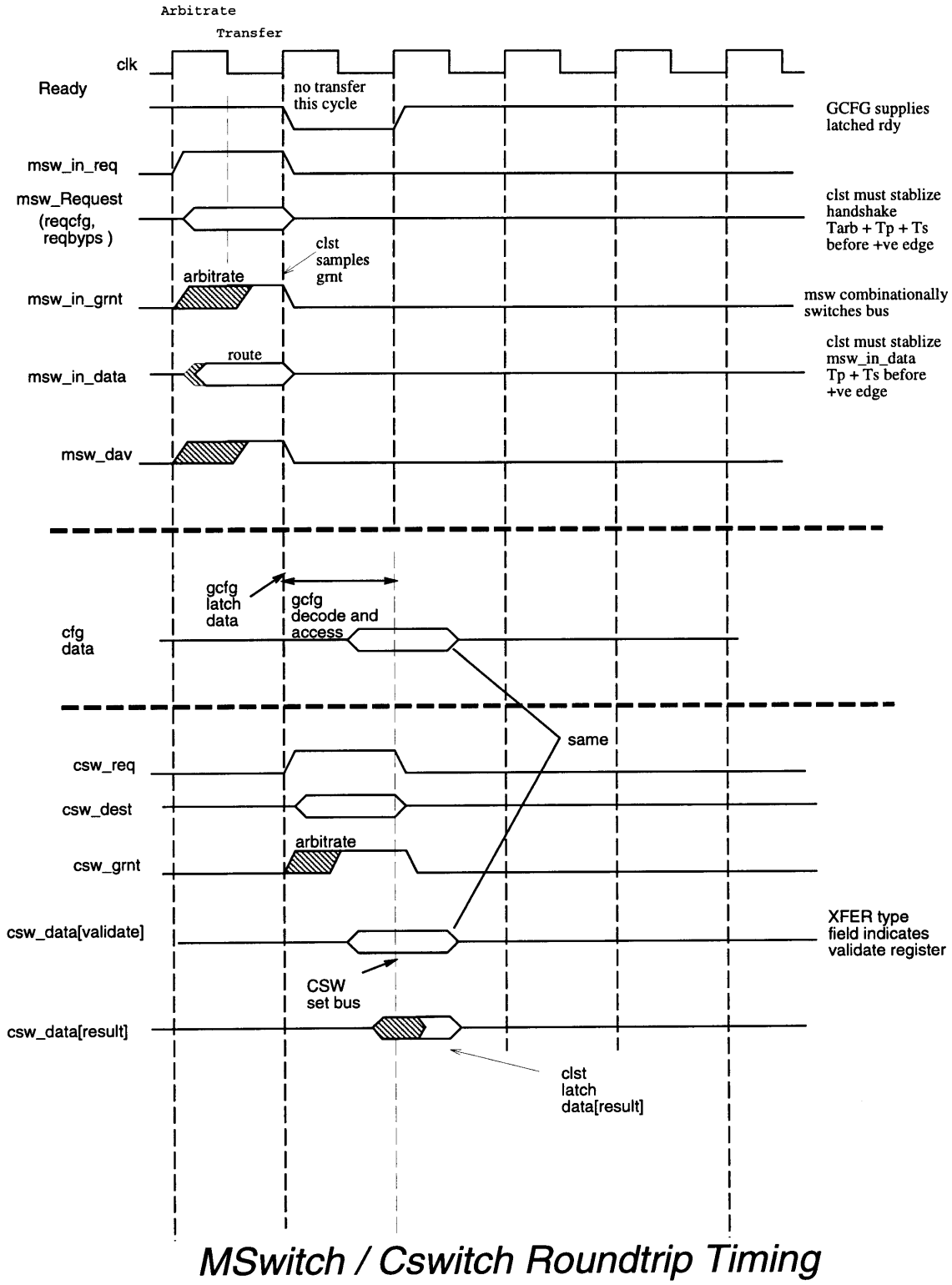

Figure 2-7: Timing from M-Switch to C-Switch in GCFG.

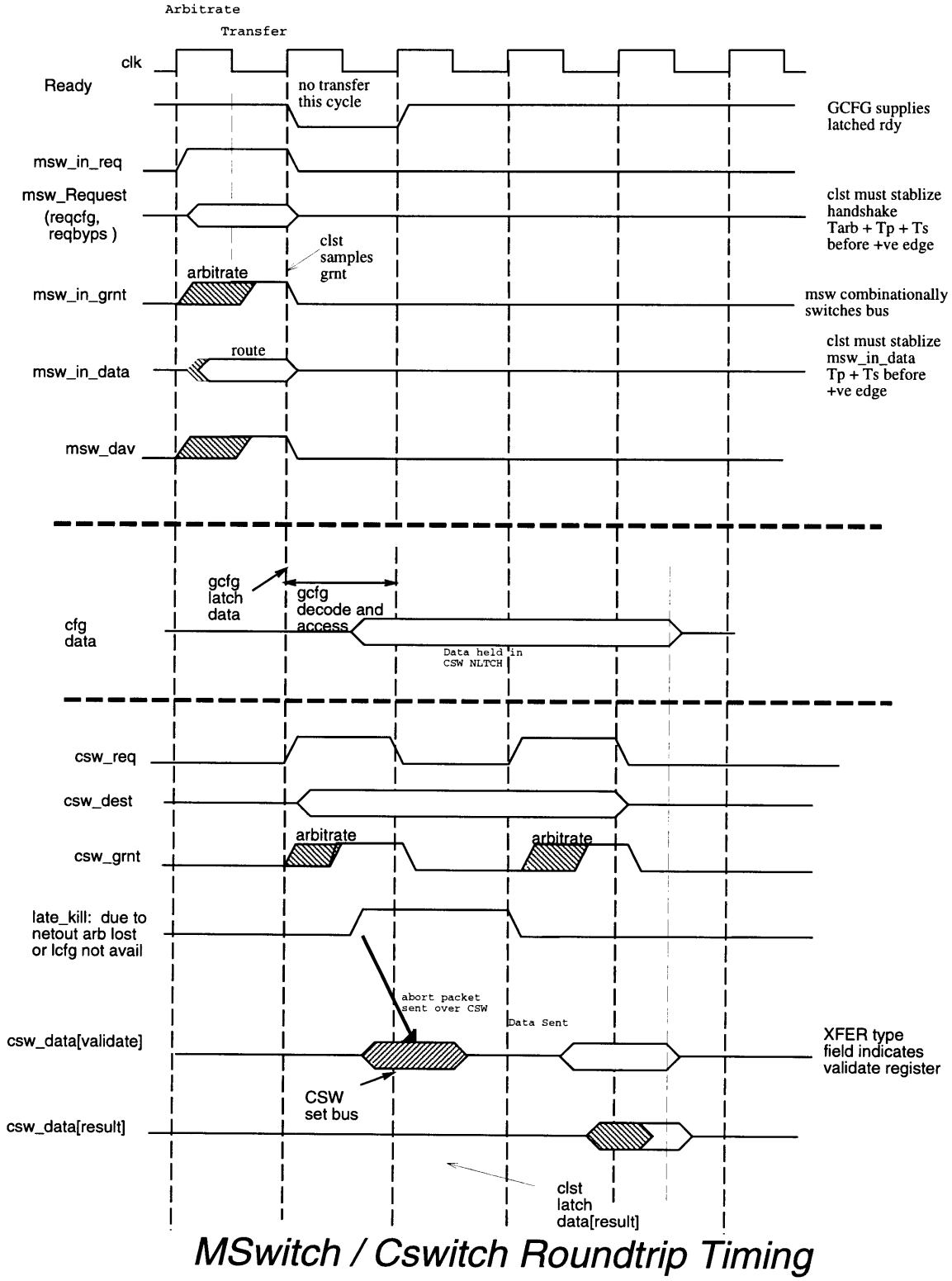

Figure 2-8: MSW to CSW timing with late_kill

Chapter 3

Putting it all Together: Playing with Threads

At this point, all the facilities needed to start and swap a thread are available. We have the ability to turn threads on and off, read and write thread state, and manipulate the IP's. There are two mechanisms for starting a new thread. The first, HFORK, is available to all users but is limited in functionality [3]. The second, referred to as VFORK, requires more steps but it allows the destination to be any of the 15 VThreads. Finally, the steps required to remove an active thread and install it at a later time in any thread are discussed in the second section.

3.1 Starting a New Thread

HFORK The HFORK instruction allows any user thread to start a new HTHREAD on a neighboring Cluster. The HFORK command is limited in that the new HTHREAD will always fall within the same VTHREAD as the parent. The HFORK command is robust in that it does not allow the overwriting of active threads. It accomplishes this by first checking the HThread Active bit for the destination thread. In the case where there is an active thread the GCFG will return a CC value of zero and is done. When the HFORK is allowed to complete the GCFG state machine sequences through a series of steps that target the destination Thread with the following actions:

- Write the new IP and set HThread Active
- Send the LCFG Hinit
- Send an LCFG NOP
- Return the CC value and set Hthread Run

The LCFG NOP is required to ensure the IP advances to the top of the pipeline before the thread is activated by turning on HThread Run. This new thread can complete by executing an HEXIT command, this will clear the HThread Active and run bits. Below is a code segment that will FORK a new process one relative HThread to the right:

```
instr ialu imm (_hfork_dest - _point1), i7;
_point1:
instr ialu lea i1, i7, i7;
instr ialu mov #1, i6;
instr memu hfork i7, i6, cc1;
instr ialu cf cc1 br _hfork_fail;
        .
        .
        .
        .
        .
_hfork_dest:
        .
        .
instr memu hexit;
__spin:
instr ialu br __spin;
instr ialu add i0, i0, i0;
instr ialu add i0, i0, i0;
instr ialu add i0, i0, i0;
```

VFORK The second mechanism leaves each of these steps to the master thread. This allows the destination to be a different slot and cluster combination by specifying

the address. In order to ensure that the IP is loaded in the IFU, two instructions should go through the GCFG to the destination LCFG before the thread is activated. The following example show the sequence of instructions to execute a VFORK onto Cluster 1, VThread 1 from anywhere assuming:

```
i3  =  0xF4240000000000fd8   /*  CFG_LCFG_NOP  */
i4  =  0xc028000000000210    /*  V1_VRUN    */
i5  =  0xc028000000000410    /*  V1_HRUN    */
i6  =  0xc028000000000810    /*  V1_HACT    */
i7  =                0x1a8   /*  CFG_LT_HFORK_IP   */
i15 =                0x150   /*  CFG_LT_HINIT    */
i8  =  0xF4240000000001200   /*  CFG_BASE_LC for CLST 1 SLOT 1*/
                            /* i4, i5, i6 and i8 have pointer bit set*/

instr ialu imm (_vfork_dest - _point1), i11;
_point1:
instr ialu lea i1, i11, i11;

instr memu ld i6, i10;
instr ialu and i10, #2, i12; /* Is there a thread active already */
instr ialu ine i12, i0, cc0; /* in cluster 1 */
instr ialu ct cc0 _vfork_fail;

instr ialu lea i8, i7, i9;   /* Create Pointer to write IP */
instr memu st i11, i9;       /* Write the IP */
instr ialu lea i8, i15, i15; /* Make the HINIT pointer */
instr memu st i0, i15;       /* Assert HINIT */
instr memu ld i5, i13;       /* Get the HRUN 3bit vector  */
instr ialu or i13, #2
      memu st i0, i3;        /* Send LCFG NOP  */
instr memu st i13, i5        /* Send new HRUN vect */
      ialu mov #1, i14;
instr memu st i14, i4;       /* Assert the VRUN bit */

instr ialu or i10, #2, i10;  /* i10 still has HACT vector */
instr memu st i10, i6;       /* St the new HACT with CLST1 Active */
```

3.2 Thread Swap

The Configuration Space controller provides the ability to read the state of a thread and store it in memory for a period of time before installing it. The runtime system for the M-Machine is to be coded in the C programming language. The following structure is to be used to store thread state:

```
struct HContext {
  int int_reg_file[16];       /* 16 integer registers            */
  int fp_reg_file[16];        /* 16 floating-point registers     */
  int cc[8];                  /* cc registers                    */
  int cc_all;                 /* all 8 cc registers at once      */
  int hardware_mbar_counter; /* hardware memory-barrier counter */
  int restartIPvector[4];     /* IP's to restart the thread      */
  int int_empty_scoreboard;  /* register empty-state scoreboard */
  int fp_empty_scoreboard;   /* register empty-state scoreboard */
  int cc_empty_scoreboard;   /* register empty-state scoreboard */
  int bp;                     /* Branch pending bit */
  int stall_bit;              /* Stall cycle bit */
  int priority;               /* Hthread priority */
  int PC;                     /*preempt counter*/
  int PT;                     /*preempt threshold*/
};
```

3.2.1 Thread Evict

When reading out the thread state it is necessary to check that the Memory Barrier Counter, MEMBAR_CTR, is zero. This ensures there are no outstanding memory requests. After this check is complete the state can be read. The only state of interest when reading is the Restart IP. It is read four times to obtain all four IP's in the pipeline, in addition, it should be noted that this read is destructive. Due to area limitations on the MAP chip, the floating point units were removed from Clusters 1 and 2. This is resolved in the thread swap code by a conditional statement surrounding all reads and writes to FP state:

```
/* Set CP to address of Floating Point Register Zero  */
    if(ht == 0)
      for (i = 1; i < 16; i++) {
        hc->fp_reg_file[i] = CP[i];
      }
/* When CP address is Floating Point Scoreboard  */
    if(ht == 0)
      hc->fp_empty_scoreboard = CP[0];
```

For the full subroutine used to swap a thread context out see Appendix C.

3.2.2 Thread Install

The code sequence used to install a thread is much the same, however, there are two peculiarities of the Configuration Unit that require special attention. The first involves writing the three scoreboards. The second is due do the way in which IP's are loaded.

Scoreboard Write The three scoreboards, integer, floating point and CC, are implemented in identical fashions. In each case, relatively large amounts of routing resources and nontrivial control logic would be needed to implement a familiar parallel load. To avoid this, a write to a scoreboard via Config space is a two stage process. While the Config request is in the RR stage of the pipeline all scoreboard bits are set to valid. Thus, by requiring the data vector written to the scoreboard to have ones in locations the scoreboard should get zeroes, we have reduced the problem to an empty instruction. This is exactly what occurs, the RR stage of the LCFG creates a modified empty instruction and sends it down the pipeline. To save routing resources, the 16 bit vector should be shifted left 48 bits. Add the inversion detailed above and you have the vector needed. Below is a code segment that writes the three scoreboards assuming `CP` is a pointer the the address of the integer scoreboard of the destination thread:

```
    CP[0] = 1 - (hc->int_empty_scoreboard << 48);
    if(ht == 0)
```

```
  CP[1] = 1 - (hc->fp_empty_scoreboard << 48);
CP[2] = 1 - (hc->cc_empty_scoreboard << 48);
```

IP Write Both the state of the BP bits and the integer unit operation referenced by the first TRIP (`ip_v2`) must be examined to determine how to properly restore the thread state when installing a thread. If the IU operation referenced by `ip_v2` was a branch or a jump, then a configuration space store to HT_HFORKIP should be used to properly setup the pipeline IP's. If the IU operation was not a branch or a jump, then four successive configuration space stores to HT_TRESTART_W should be used to install the IP's. If the Branch Pending bit for the selected thread was asserted then all four of the IP's that were read out and saved are valid and should be re-installed. If the BP bit was not asserted then the last IP (`fip_2`) previously saved is NOT valid. The fourth IP required for TRIP load should be generated by incrementing the third IP (`if_iip_2`) by 4, 8, or 12, based on the length of the instruction. Here is the code segment to accomplish this (the functions called are assembly routines provided in Appendix C):

```
if (isbranch(hc->restartIPvector[0]))
  HT_HFORKIP = hc->restartIPvector[0];
else
  {
    HT_TRESTART_W = hc->restartIPvector[0];
    HT_TRESTART_W = hc->restartIPvector[1];
    HT_TRESTART_W = hc->restartIPvector[2];
    if (hc->bp)
      HT_TRESTART_W = hc->restartIPvector[3];
    else
      HT_TRESTART_W = nextIP(hc->restartIPvector[2]);
  }
```

Chapter 4

Diagnostic Unit

4.1 Introduction

The MAP Diagnostic system is required for all aspects of system initialization and test control. This includes control of the chip wide DIP and Scan chains, MSW write ability and CSW read function. The design should minimize the number of pins, chip area and routing resources used in order to leave those for portions of the chip designed for high performance.

A block diagram of the DIAG unit and the chip wide shift chains it controls is shown in figure 4-1. It should be noted that the four shift chains are clocked by three different clocks. This design choice simplified the hardware and placed numerous timing constraints on the interface.

The DI_SI, DI_CIN and DI_SO signals are the main source of communication for control and data between the DIAG unit and the off chip module. This off chip module will be expected to send in requests to the DIAG and supply data when necessary. The off chip module has not been designed, however, a behavioral module exists to demonstrate functionality.

This chapter describes the design and interface of the Diagnostic unit of the MAP chip. First the interface and functions are discussed. Each of the shift chains are then reviewed. Finally, the steps required for booting the MAP chip are presented.

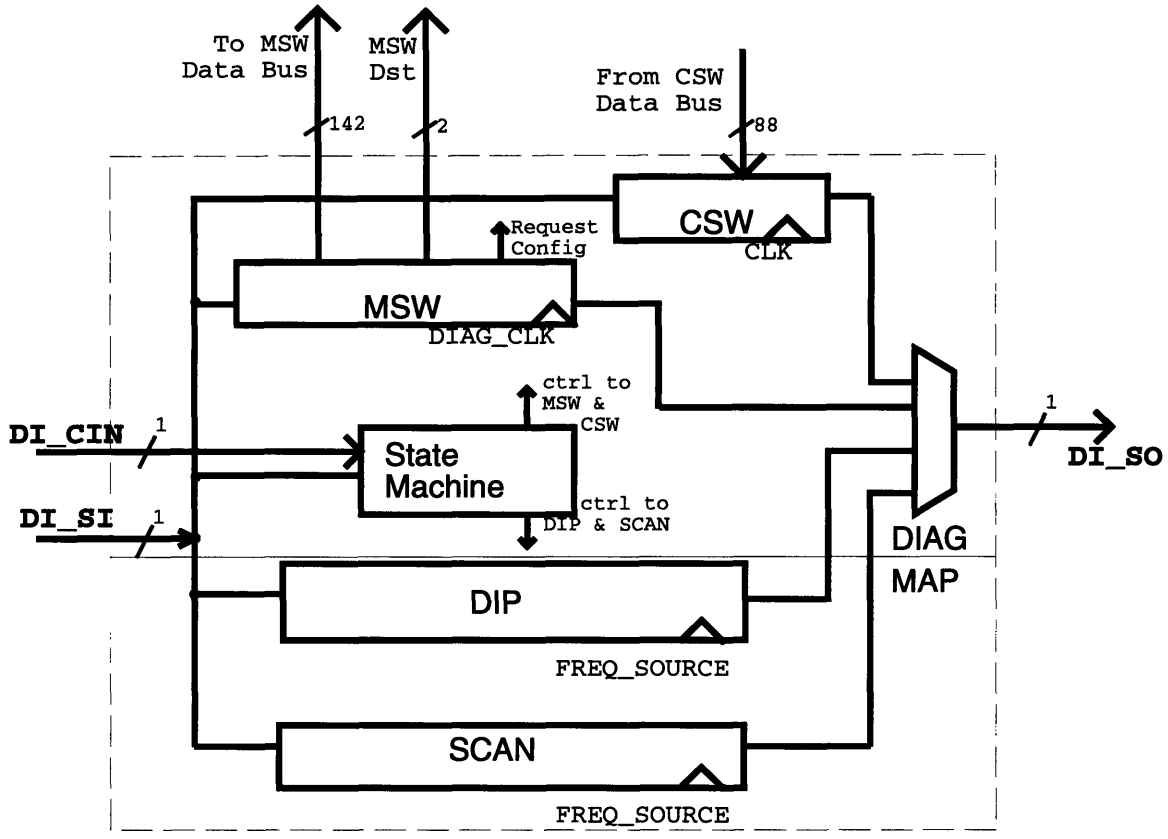

Figure 4-1: *Block diagram of the DIAG*

4.2 High Level Design and Interface

The block diagram shown in figure 4-1 shows the flow of data through the DIAG. There are additional control signals that communicate between the state machine, the four shift chains and the off chip module. The DIAG is responsible for controlling four shift chains, requesting MSW transactions, accepting CSW data, executing a soft reset, and controlling the IC bit bucket. The four shift chains will be discussed in detail in later sections.

4.2.1 Interface

The DIAG unit has an eight pin interface off the chip. They include the following inputs: external reset (`e_rst`), data input (`din`), control input (`cin`), an off chip

FREQ_SOURCE (diag_clk_in) which serves as the basis for the chip wide DIPCLK and SCLK, SRUN input bit (srun_in), and outputs: data out (dout), MSW grant acknowledge (msw_out), and CSW packet arrival (csw_full). With this interface, the off chip module has complete control over the functions of the DIAG.

In addition, the DIAG unit interfaces with a CSW read port, MSW write port, outputs the chip wide RUN, DIPRUN, produces the first bit of the SCAN and DIP chains and receives the last bit of each chain, produces IFU_TEST and ICD_SLD and generates the chip wide rst.

4.2.2 Functions

The DIAG unit has a simple state machine which functions as a state recognizer. The inputs are the e_rst, cin, and diag_cmd. The single bit, cin, is used to decipher when the off chip interface is shifting in data or command info. Thus, the state machine only has to recognize when it should do something (external reset = cin = 0) and match the command to the state encoding.

The DIAG works as the slave of an off chip module. Commands are sent in from off chip as the only way to transition the DIAG out of the idle state. A command causes the DIAG to set it's muxes and clock gating to perform the specified task.

Commands are stored in the serial shift register, diag_cmd. The schematic of the diag_cmd register is shown in figure 4-3. After a command is entered into the diag_cmd register, the cin bit is lowered and the state machine will transition. On

DIAG command name	Bit pattern	comment
IDLE	4'b0000	
RESET	4'b0011	Usually multiple cycles
IFURST	4'b1000	
IFUSET	4'b1001	
REQUEST MSW	4'b0001	Only ONE cycle
SHIFT MSW	4'b0100	Should be 154 cycles
SHIFT CSW	4'b0101	Should be 88 cycles
SHIFT SCAN	4'b0110	One cycle for each SCAN cell
SHIFT DIP	4'b0111	One cycle for each DIP cell

Table 4.1: DIAG commands

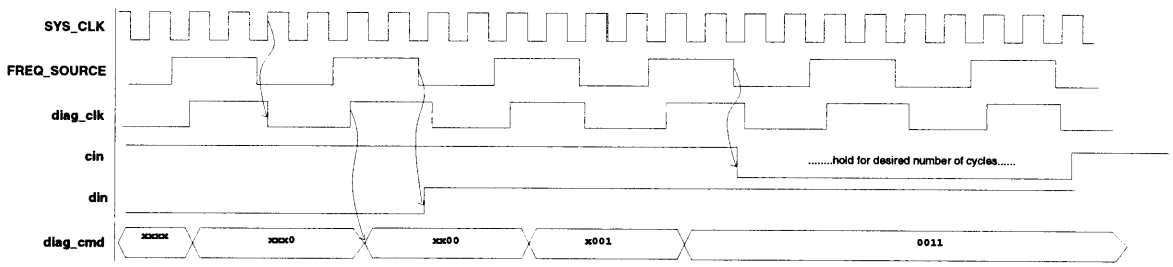

Figure 4-2: *Timing diagram for setting reset*

the following cycles the DIAG will execute the command. This command will be executed until `cin` is raised. The list of commands recognized by the DIAG are in table 4.1.

The two clock domains make the shift register timing a non-trivial task exaggerated by the fact that the two domains have no phase relationship. Rather then use an analog circuit to bring the two clocks into phase, no assumptions are made about the off chip FREQ_SOURCE except that we can give it a limit. We bring the FREQ_SOURCE into the on chip clocks domain by sending through a PDFF, after which it is referred to as diag_clk.

The `diag_cmd` register is clocked with diag_clk. When the control input pin transitions from 1 to 0 a command should begin and the four data inputs in the `diag_cmd` specify the command. Refer to figure 4-2 for the timing diagram of entering a command (Reset in this case.)

4.3 MSW Module

The DIAG unit requires an MSW write port in order to initialize memory, write bootstrap code, install IP via Config space and to read data values via Config space. The MSW interface module resides between the Global Configuration unit and the MSW busses along with the DIAG state machine and additional logic. A single shift chain is used to hold the MSW packet, destination, and config bit. The first bit shifted in is used to decide if this is a config request, the next two bits are used as

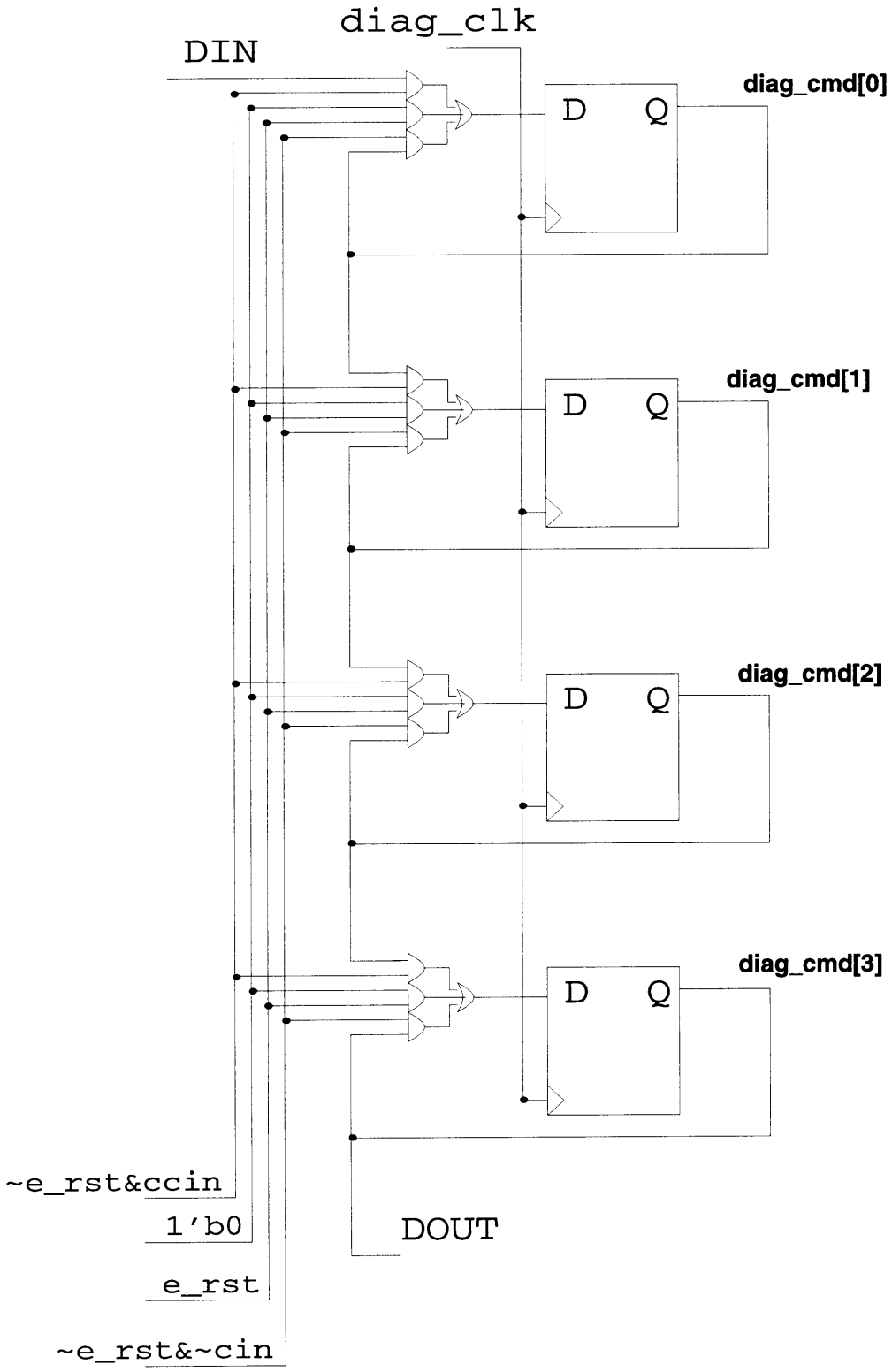

Figure 4-3: *Schematic of diag_cmd registers*

the msw_dst bits followed by the pass through bundle.

The shift chain does not have a parallel load and therefore requires only a two input mux in front of each register giving the ability to shift and hold. The registers are clocked by diag_clk (see definition above.) The off chip module is responsible for shifting the correct number of bits into the register and then initiating the MSW request.

After the MSW data is shifted in the off chip module should initiate the request. This is done by shifting in the MSWREQ command for a single cycle of the off chip FREQ_SOURCE. The msw_req bit sent to the MSW is generated by recognizing this state and holding the bit high until msw_grnt is asserted. At this point the mswout signal can be polled, this is the signal that the MSW data has been sent.

Figure 4-4 shows the timing diagram for entering a MSW packet. The first bit of data should be valid prior to the first positive edge of FREQ_SOURCE following `cin` going low. Successive data bits should be supplied prior to each positive edge. It is suggested that din be changed on the falling edge of FREQ_SOURCE rather then the positive edge. This is due to the potential erratic routing of the FREQ_SOURCE relative to din. Thus, the first bit of data should be driven on `din` at the same edge `cin` is lowered. Successive changes should occur on falling edges of FREQ_SOURCE and cin should be raised one full cycle after the last bit was placed on `din`.

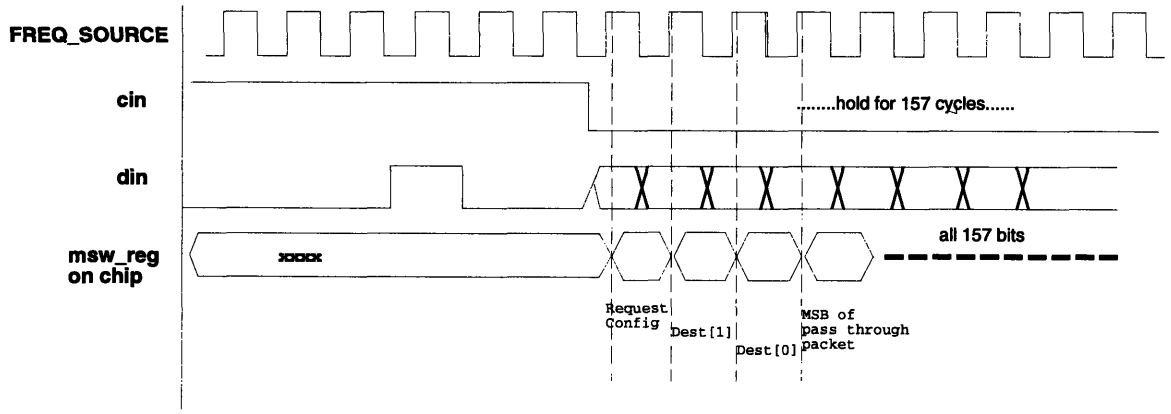

Figure 4-4: *Timing diagram for off chip MSW read*

4.4 CSW Module

The CSW read port is necessary to extract data values from the MAP chip in debugging situations. We are able to generate an MSW packet that will go to the GCFG and generate an LCFG request to read a register and target the DIAG as the destination.

The CSW interface module is designed to be physically separated from the rest of the DIAG logic. It will be placed near the CSW bus between clusters zero and one. Due to the separation from the main DIAG logic it is desirable to minimize the interface between the two units. The interface is reduced to four single bits: serial data in, serial data out, pulsed version of FREQ_SOURCE as an input and another output that signals when a match occurred.

The DIAG unit shares a CSW read port with Cluster one. CSW writes to registers $8 -> 15$ of thread 2 will cause the DIAG to register a match and set the signal fed to the off chip module. At this point the data will be parallel loaded into registers and held until the off chip module initiates a serial read. At any time while waiting or shifting, the data can be overwritten if another CSW write to a matching address occurs. This corrupting behavior is acceptable due to the expected use of the CSW port.

The off chip module is responsible for polling for a CSW match and initiating the data shift. In addition, the module will be responsible for detecting when the full word has been shifted.

The shift signal is gated by two events. First, the state machine must be in the CSW shift state. At this point the shift signal is pulsed for one on chip cycle each falling edge of the DIAG_CLK. A timing diagram is shown in figure 4-5. The '<' notation is used to express a logical left shift where the most significant bit of the CSW register has been shifted off the chip.

We have shown above how the CSW register is clocked, we now explore this process from the off chip modules point of view. Before meaningful data can be read, the `DI_CSWDAV` pin of the MAP chip must go high. This is the signal that the off chip

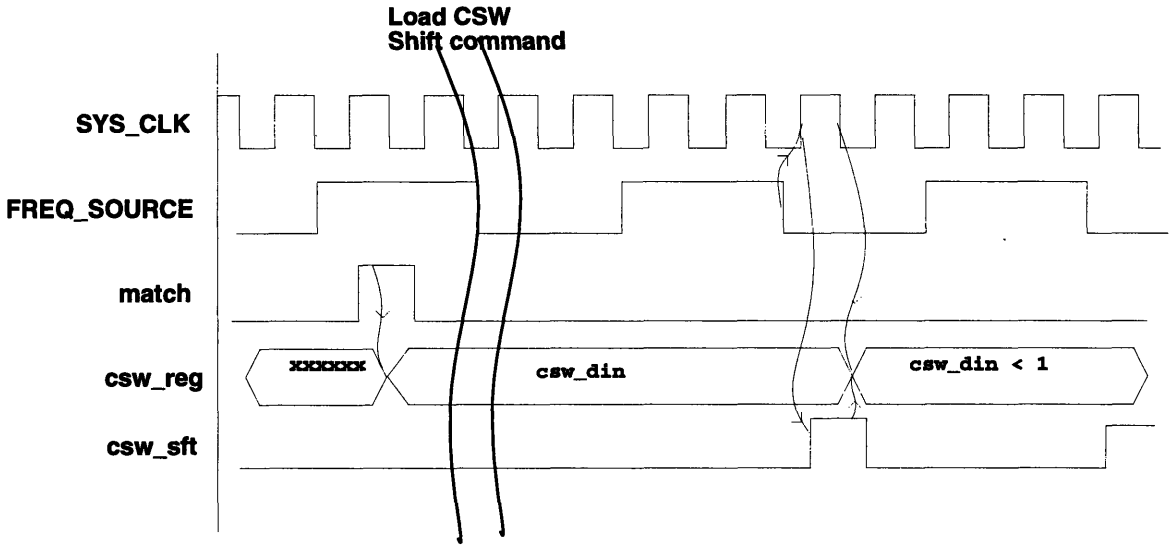

Figure 4-5: *Timing of the CSW registers*

module can initiate shifting the CSW register. As shown above in table 4.1, we shift in the command 4'b0101.

Figure 4-6 shows a timing diagram of the pins the off chip module sees. The register `csw_vect` is an 88 bit register located on the off chip module to store the CSW packet. It should be clocked on falling edges of FREQ_SOURCE starting with the edge at which `cin` falls. Then, at each successive negative edge of FREQ_SOURCE another bit of the CSW should be read (the MSB is the first bit out.) As the 88th bit is read the `cin` should also be raised to one.

4.5 SCAN chain

The SCAN chain on the MAP chip is a serial shift register through the PDFF and NLTCH cells used in the control logic [1]. The SCAN chain has some special requirements related to switching the cells from normal operation into a SCAN chain. This sequence of transitions to the RUN, CLK and SCLK are all handled by the DIAG and it's off chip interface.

The reason for this sequence can be extracted from the design of the PDFF_SCAN

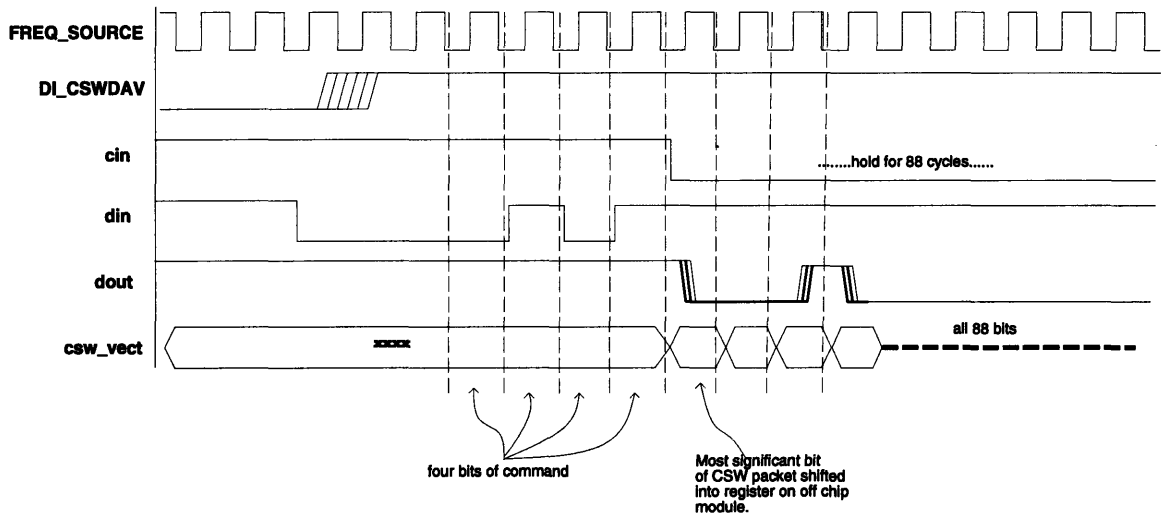

Figure 4-6: *Timing diagram for off chip CSW read*

cell shown in figure 4-7. We can see that under normal operation the RUN bit must be set to one and SCLK should be set to one so there is no feedback path from the input D through the input SIN.

The first step in switching to the SCAN chain is entering the SCAN shift command. Then the system clock must be disabled by lowering `CK_EN`. This is done in a separate module and the clock is left at the high state. At this point the chip is frozen The RUN bit should then be set to zero so that the state of the output Y is held. At this point it is safe to begin pulsing SCLK and shifting the chain.

The DIAG module uses one additional pin to facilitate the timing of the above events. This pin, `srun_in`, is set to one in normal operation and is also logically or'ed with the FREQ_SOURCE to produce chip wide SCLK. This is possible by adding the simple requirement that srun_in transition from one to zero at the positive edge of FREQ_SOURCE.

The exact sequence of events as seen by the off chip module is shown in figure 4-8. The length of the SCAN chain will not be known until the control logic has been synthesized for the final time.

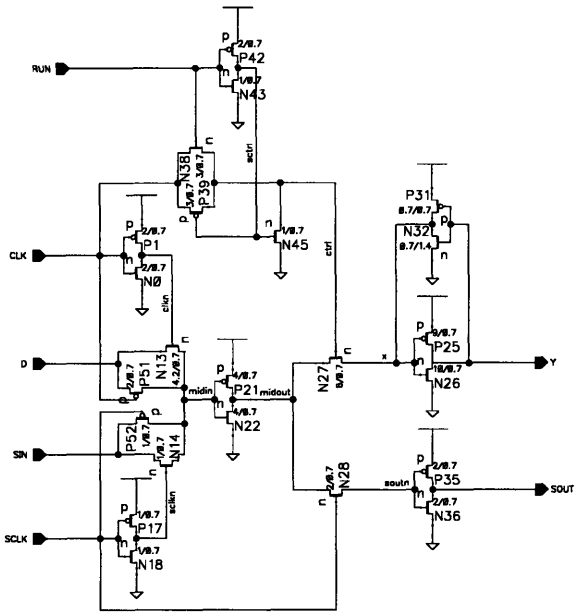

Figure 4-7: *Schematic of PDFF_SCAN register*

4.6 DIP chain

The DIP chain is used after a hard reset or power up to initialize the DIP switches throughout the chip. A DIP cell is effectively a PDFF with a load enable. Figure 4-9 shows the timing diagram for shifting the DIP chain. The diagram shows that there is a one cycle latency before `cin` is lowered and the first bit of data needs to be supplied on `din`.

The DIP chain will will provide freedom in two directions. First, it is used to provide variables in timing path that are process dependent. Second, it allows the chip to be configured to different settings for testing or interface reasons. In the first case, a working value for the DIP cell may not be known. It will therefore be necessary to test many DIP chain patterns before a correct one is found for each chip.

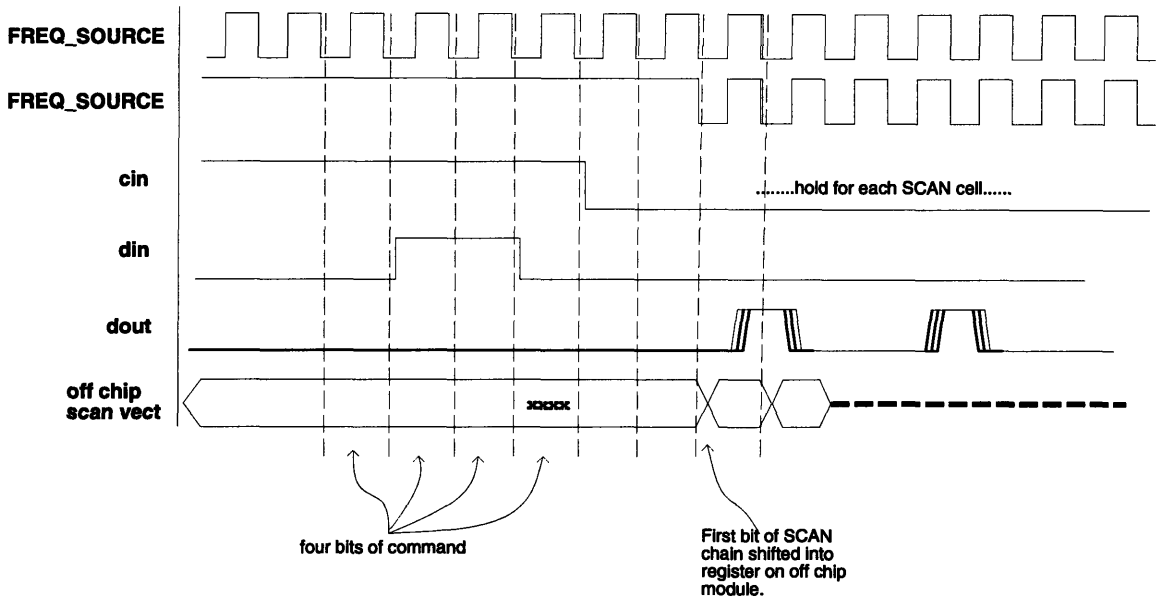

Figure 4-8: *SCAN chain timing*

4.7 Boot Sequence for the MAP Chip

The fundamental purpose of the DIAG unit is to boot the MAP chip. We now explore the sequence of steps that accomplish this.

On a cold restart, the state of the MAP, including the DIAG unit, is completely unknown. It is therefore necessary to assert the external reset pin. At this point all DIAG registers have been initialized and it will be awaiting a command. Before beginning to write the boot strap code we must set the DIP switches.

After scanning in the DIP switches it is necessary to assert reset (either the external pin or via the DIAG command) for 100μs to initialize the SDRAMs. At this point we begin to write the boot strap program into memory. This is done with successive MSW commands. Once the program is loaded into memory we perform the steps necessary to start up a thread. This includes writing the IP through the HFORK IP address of the GCFG. Then the thread must be activated by writing the VRUN, HRUN and HACTIVE bits in the GCFG. This is also accomplished with MSW writes to the corresponding addresses.

At this point we have a program loaded into memory and the thread should

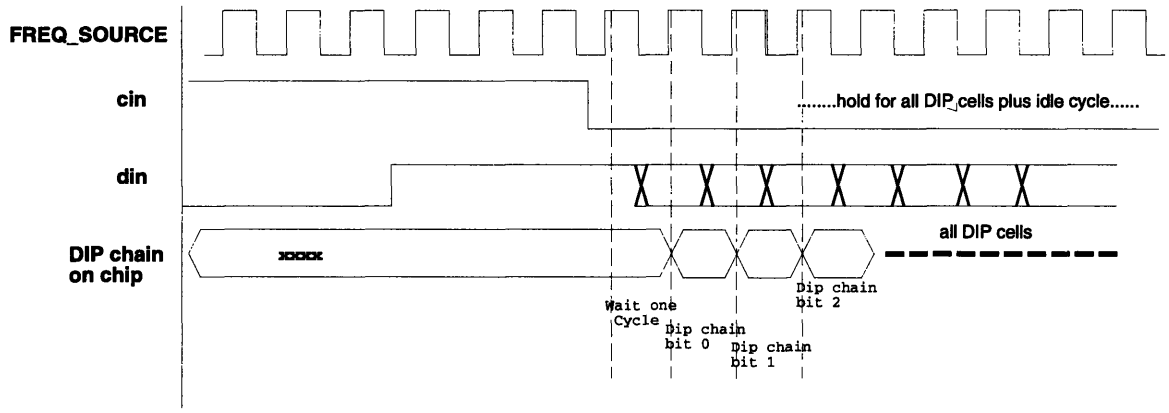

Figure 4-9: *DIP chain timing*

be issuing. While the MAP chip is operational there are functions that the DIAG can perform. First, any data register and memory location can be read. This is accomplished by creating the correct MSW transaction. The destination should be set to one of the registers that map to the DIAG. The off chip unit can then spin waiting for the signal that the DIAG has found a write to it's CSW port and shift out the data. In addition, a soft reset can be performed.

The SCAN chain and ICDRAM bit bucket testing are used to explore potential problems in the control unit and ICACHE respectively. It is expected that this testing will take place after running the chip for many cycles and the chip will not be restarted without a hard reset afterwards.

Chapter 5

Conclusions

5.1 Summary

This thesis presented the design and implementation of the Configuration and Diagnostic units of the MAP chip. The MAP chip can be found at each node of the M-Machine. The necessity of the Configuration and Diagnostic units within the MAP was presented in the introduction.

The Global Configuration unit was described first. We started with a broad overview and moved into the addressing scheme. This left a vague picture of the end functionality of the GCFG. By describing the individual modules, the flow of data through the GCFG provided the details involved in the logic design. Finally, we showed the timing path from the MSW through the GCFG to the CSW.

With this knowledge of the GCFG, we attacked the behavior of starting, evicting, and re-starting a thread on the MAP chip.

The next chapter described the Diagnostic unit. Again, we start by glossing over the functionality and requirements before attacking individual pieces in detail. Careful attention is paid to the timing requirements due to the multiple clock domains.

5.2 Suggestions For Improvement

Both the GCFG and DIAG accomplished their tasks. Through the design of both units, design time and silicon area played a large part in the end result. Throughout the design of the MAP chip functionality was removed because it would not fit or there was no time to design it.

Fortunately for both modules, these two constraints did not fight each other. The requirement on area led to simple designs. For example, the GCFG could have been designed with multiple buffers. This would have bloated area and made the control logic harder to design.

There is one area of the DIAG unit that could have been improved given more time. Given a second pass, the DIAG unit could have been designed to have more symmetric timing requirements with the off chip module. The current design was too short-sided in that it pushed much of the control off the chip. This makes the unit smaller and easier to design, however, it is not clear this will be a win in the end.

Appendix A

Event System

The division of the event space is presented below.

Address	Signal	description
0,clst_id,0000	tseld_1[0]	
0,clst_id,0001	tseld_1[1]	
0,clst_id,0010	tseld_1[2]	
0,clst_id,0011	tseld_1[3]	
0,clst_id,0100	tseld_1[4]	
0,clst_id,0101	tseld_1[5]	
0,clst_id,0110	rdy_1[0] & ~tseld_1[0]	
0,clst_id,0111	rdy_1[1] & ~tseld_1[1]	
0,clst_id,1000	rdy_1[2] & ~tseld_1[2]	
0,clst_id,1001	rdy_1[3] & ~tseld_1[3]	
0,clst_id,1010	rdy_1[4] & ~tseld_1[4]	
0,clst_id,1011	rdy_1[5] & ~tseld_1[5]	
0,clst_id,1100	match_v2	a hit in the IFU
0,clst_id,1101	miss	a miss in the IFU
0,clst_id,1110	ikill_v2	a kill in the IFU

Table A.1: Cluster addressing

Unit	Address	Event description
Cache		
	011,bank,00	cache hit
	011,bank,01	cache miss
	011,bank,10	sync op passed
	011,bank,11	sync op failed
EMI		
	1000000	ltlb hit
	1000001	request handled
	1000010	writeback to SDRAMS
	1000011	sync op passed
	1000100	sync op failed

Table A.2: Memory addressing

Unit	Address	Signal	description
CSW			
	1001000	CSW_EV_XFR	data being transfered
	1001001	CSW_EV_CFLCT	req denied.
	1001010	CSW_EV_BURST	burst mode transfer
MSW			
	1010000	MSW_EV_XFR	data being transfered
	1010001	MSW_EV_RSRC	req denied due to resource busy.
	1010010	MSW_EV_CFLCT	req denied.

Table A.3: Switch addressing

Address	Signal	description
1011000	EQ_EV_ARRV	word arrival
1011001	EQ_EV_PENDING	word pending for extraction
1011010	EQ_EV_STALL	word pending for extractin but not extracted (wait)
1011011	EQ_EV_CAPACITY	watermark flag asserted.

Table A.4: EQ addressing

Address	Signal	description
1100000	NO_P0_EV_ARRV	mesg word arrival from csw
1100001	NO_P0_EV_IDLE	idle (no reqs & no ops in progress)
1100010	NO_P0_EV_PENDING	word pending for injection
1100011	NO_P0_EV_STALL	word pending for injection but not injected
1100100	NO_P0_EV_CAPACITY	netout queue at full capacity
1100101	NO_P0_EV_FLOWCTRL	ombc at 0
1100110	NO_P0_EV_GTLBMISS	gtlb miss flag asserted
1100111	NO_P0_EV_CONFLICT	req denied due to contention

Table A.5: NETOUT P0 addressing

Address	Signal	description
1101000	NO_P1_EV_ARRV	mesg word arrival from csw
1101001	NO_P1_EV_IDLE	idle (no reqs & no ops in progress)
1101010	NO_P1_EV_PENDING	word pending for injection
1101011	NO_P1_EV_STALL	word pending for injection but not injected
1101100	NO_P1_EV_CAPACITY	netout queue at full capacity
1101101	NO_P1_EV_FLOWCTRL	ombc at 0
1101111	NO_P1_EV_CONFLICT	req denied due to contention

Table A.6: NETOUT P1 addressing

Address	Signal	description
1110000	NI_P0_EV_ARRV	p0 mesg word arrival from router
1110001	NI_P0_EV_PENDING	p0 mesg word pending for extraction
1110010	NI_P0_EV_STALL	p0 mesg word pending for extraction but not extracted.
1110011	NI_P0_EV_CAPACITY	p0 mesg from router ready but not extracted by netin
1110100	NI_P1_EV_ARRV	p1 mesg word arrival from router
1110101	NI_P1_EV_PENDING	p1 mesg word pending for extraction
1110110	NI_P1_EV_STALL	p1 mesg word pending for extraction but not extracted
1110111	NI_P1_EV_CAPACITY	p1 mesg from router ready but not extracted by netin

Table A.7: NETIN addressing

Address	Signal	description
1111000	RTR_EV_ARRV	word arrival
1111001	RTR_EV_DEPT	word departure
1111010	RTR_EV_QUEUE	mesg waiting for channel allocation
1111011	RTR_EV_STALL	word waiting for forwarding.

Table A.8: ROUTER addressing

Appendix B

Programmers guide to Config Addressing

Local configuration addresses are created by a bitwise `or` of the Local Config Base Address with the offset.

V-Thread	Base Address
Cluster 0 V-thread 0	0xC024000000000000
Cluster 0 V-thread 1	0xC024000000000200
Cluster 0 V-thread 2	0xC024000000000400
Cluster 0 V-thread 3	0xC024000000000600
Cluster 0 V-thread 4	0xC024000000000800
Cluster 0 V-thread 5	0xC024000000000a00
Cluster 1 V-thread 0	0xC024000000001000
Cluster 1 V-thread 1	0xC024000000001200
Cluster 1 V-thread 2	0xC024000000001400
Cluster 1 V-thread 3	0xC024000000001600
Cluster 1 V-thread 4	0xC024000000001800
Cluster 1 V-thread 5	0xC024000000001a00
Cluster 2 V-thread 0	0xC024000000002000
Cluster 2 V-thread 1	0xC024000000002200
Cluster 2 V-thread 2	0xC024000000002400
Cluster 2 V-thread 3	0xC024000000002600
Cluster 2 V-thread 4	0xC024000000002800
Cluster 2 V-thread 5	0xC024000000002a00

Table B.1: Local Config Base Addresses

Reg ID	Offset	Width
I0	0x0000000000000000	66
I1	0x0000000000000008	66
I2	0x0000000000000010	66
I3	0x0000000000000018	66
I4	0x0000000000000020	66
I5	0x0000000000000028	66
I6	0x0000000000000030	66
I7	0x0000000000000038	66
I8	0x0000000000000040	66
I9	0x0000000000000048	66
I10	0x0000000000000050	66
I11	0x0000000000000058	66
I12	0x0000000000000060	66
I13	0x0000000000000068	66
I14	0x0000000000000070	66
I15	0x0000000000000078	66
F0	0x0000000000000080	66
F1	0x0000000000000088	66
F2	0x0000000000000090	66
F3	0x0000000000000098	66
F4	0x00000000000000a0	66
F5	0x00000000000000a8	66
F6	0x00000000000000b0	66
F7	0x00000000000000b8	66
F8	0x00000000000000c0	66
F9	0x00000000000000c8	66
F10	0x00000000000000d0	66
F11	0x00000000000000d8	66
F12	0x00000000000000e0	66
F13	0x00000000000000e8	66
F14	0x00000000000000f0	66
F15	0x00000000000000f8	66

Table B.2: Cluster Register File Offsets

State	Offset	Width
CC0 - h0.cc0	0x0000000000000100	1
CC1 - h0.cc1	0x0000000000000108	1
CC2 - h0.cc2	0x0000000000000110	1
CC3 - h0.cc3	0x0000000000000118	1
CC12 - cc0	0x0000000000000120	1
CC13 - cc1	0x0000000000000128	1
CC14 - cc2	0x0000000000000130	1
CC15 - cc3	0x0000000000000138	1
CCALL	0x0000000000000140	8
MBAR	0x0000000000000148	8
HT_INIT	0x0000000000000150	1
HT_PRIORITY	0x0000000000000158	1
HT_PCTR	0x0000000000000160	8
HT_PTR	0x0000000000000168	8
HT_STALL_CYC	0x0000000000000178	8
HT_ISB	0x0000000000000188	16
HT_FSB	0x0000000000000190	16
HT_CCSB	0x0000000000000198	16
HT_TRESTART_W	0x00000000000001a0	66
HT_HFORKIP	0x00000000000001a8	66
HT_TRESTART_R	0x00000000000001b0	66
HT_BPEND	0x00000000000001c0	1

Table B.3: Cluster State Offsets

State	Address	Width
VTRUN_0	0xC028000000000208	1
VTRUN_1	0xC028000000000210	1
VTRUN_2	0xC028000000000220	1
VTRUN_3	0xC028000000000240	1
VTRUN_4	0xC028000000000280	1
VTRUN_5	0xC028000000000300	1
HTRUN_0	0xC028000000000408	3
HTRUN_1	0xC028000000000410	3
HTRUN_2	0xC028000000000420	3
HTRUN_3	0xC028000000000440	3
HTRUN_4	0xC028000000000480	3
HTRUN_5	0xC028000000000500	3
HTACT_0	0xC028000000000808	3
HTACT_1	0xC028000000000810	3
HTACT_2	0xC028000000000820	3
HTACT_3	0xC028000000000840	3
HTACT_4	0xC028000000000880	3
HTACT_5	0xC028000000000900	3
TC_0	0xC028000000020008	1
TC_1	0xC028000000020010	1
TC_2	0xC028000000020020	1
TC_3	0xC028000000020040	1
TC_4	0xC028000000020080	1
TC_5	0xC028000000020100	1
USER	0xC028000000001000	1
CYC_CTR	0xC028000000002000	64
EV_CTR	0xC028000000004000	32
EV_REG	0xC028000000008000	7
ECHO	0xC028000000010000	5

Table B.4: GCFG State Addressing

State	Address
LTLB_E0_S0_W0	0xC000000000000000
LTLB_E0_S0_W1	0xC000000000000020
LTLB_E0_S0_W2	0xC000000000000040
LTLB_E0_S1_W0	0xC000000000000080
LTLB_E0_S1_W1	0xC0000000000000a0
LTLB_E0_S1_W2	0xC0000000000000c0
LTLB_E1_S0_W0	0xC000000000001000
LTLB_E1_S0_W1	0xC000000000001020
LTLB_E1_S0_W2	0xC000000000001040
LTLB_E1_S1_W0	0xC000000000001080
LTLB_E1_S1_W1	0xC0000000000010a0
LTLB_E1_S1_W2	0xC0000000000010c0
.	.
.	.
.	.
LTLB_E63_S0_W0	0xC00000000003f000
LTLB_E63_S0_W1	0xC00000000003f020
LTLB_E63_S0_W2	0xC00000000003f040
LTLB_E63_S1_W0	0xC00000000003f080
LTLB_E63_S1_W1	0xC00000000003f0a0
LTLB_E63_S1_W2	0xC00000000003f0c0
CLEAR	0xC000000001000000
DISCARD	0xC000000002000000
ECC	0xC000000004000000
MISS_INV	0xC000000008000000

Table B.5: Memory Addressing

State	Address
NID0_READ	0xC03000000260C0b8
NID0_WRITE	0xC03000000260C038
NID1_READ	0xC03000000460C0b8
NID1_WRITE	0xC03000000460C038
OMBC_READ	0xC03000000260C0a0
OMBC_WRITE	0xC03000000260C020
OMBC_INC	0xC03000000260C028
OMBC_DEC	0xC03000000260C030

Table B.6: Network Addressing

Appendix C

Routines used in Thread Swap

```
int tEvictHThread(int ht, struct HContext *hc, int slot) {
    int i;
    int *CP;

   CP = sysSetPtr(CFG_BASE_LT |
                  (CFG_LTLC_PER_CLUSTER * ht) |
                  (CFG_LT_PER_THREAD * slot) |
                  CFG_LT_MEMBAR);

    hc->hardware_mbar_counter = CP[0];
    if (hc->hardware_mbar_counter  != 0)
      return hc->hardware_mbar_counter;

    CP = sysSetPtr(CFG_BASE_LT |
                  (CFG_LTLC_PER_CLUSTER * ht) |
                  (CFG_LT_PER_THREAD * slot));

    for (i = 2; i < 16; i++) {
        hc->int_reg_file[i] = CP[i];
    }

    CP += 16;
    if(ht == 0)
      for (i = 1; i < 16; i++) {
        hc->fp_reg_file[i] = CP[i];
      }

    CP += 16;

    for (i = 0; i < 8; i++) {
        hc->cc[i] = CP[i];
    }
```

```
    CP += 8;

    hc->cc_all = CP[0];
    hc->priority = CP[3];

    hc->PC = CP[4];
    hc->PT = CP[5];
    hc->stall_bit = CP[7];
    hc->int_empty_scoreboard = CP[9];
    if(ht == 0)
      hc->fp_empty_scoreboard = CP[10];

    hc->cc_empty_scoreboard = CP[11];
    hc->bp = CP[16];

    hc->restartIPvector[0] = CP[14];
    hc->restartIPvector[1] = CP[14];
    hc->restartIPvector[2] = CP[14];
    hc->restartIPvector[3] = CP[14];

    return hc->hardware_mbar_counter;

}

int tInstallHThread(int ht, struct HContext *hc, int slot) {
    int i;
    int *CP;

    CP = sysSetPtr(CFG_BASE_LT |
                   (CFG_LTLC_PER_CLUSTER * ht) |
                   (CFG_LT_PER_THREAD * slot));

    for (i = 2; i < 16; i++) {
      CP[i] = hc->int_reg_file[i];
    }

    CP += 16;
    if(ht == 0)
      for (i = 1; i < 16; i++) {
        CP[i] = hc->fp_reg_file[i];
      }

    CP += 16;

    for (i = 0; i < 8; i++) {
```

```
      CP[i] = hc->cc[i];
    }
    CP += 8;

    /*mbar check here?*/
     CP[1] = hc->hardware_mbar_counter;
    /* maybe want to compare to software_membar_coutner */

    CP[3] = hc->priority;
    CP[4] = 0x0;    /* hc->PC */
    CP[5]  = hc->PT;
    CP[7] = 0x0;  /* hc->stall_bit  */
    CP[9] = 1 - (hc->int_empty_scoreboard << 48);
    if(ht == 0)
      CP[10] = 1 - (hc->fp_empty_scoreboard << 48);
    CP[11] = 1 - (hc->cc_empty_scoreboard << 48);
    /*    CP[16] = hc->bp;  */


    /* Need some checking here for
       1. if first instruction is a branch, then hfork
       2. if BP set then write all four
       3. if BP not set then write first three and compute fourth  */
    if (isbranch(hc->restartIPvector[0]))
      CP[13] = hc->restartIPvector[0];
    else
      {
            CP[12] = hc->restartIPvector[0];
      CP[12] = hc->restartIPvector[1];
      CP[12] = hc->restartIPvector[2];
      if (hc->bp)
        CP[12] = hc->restartIPvector[3];
      else
        CP[12] = nextIP(hc->restartIPvector[2]);
      }
    return hc->hardware_mbar_counter;
}




_isbranch::

instr memu ld intarg0, intarg1;
```

```
instr ialu exth intarg1, intarg0, intarg2;  /*extract right half word*/
instr ialu lsh intarg2, #-19, intarg2;
instr ialu lsh intarg2, #-11, intarg3;
instr ialu ieq intarg3, #0, cc0;  /*is an integer op?*/
instr ialu ct cc0 br _isbranchintop;
instr ialu ieq intarg3, #3, cc1; /*is a multi op?*/
instr;
instr;
instr ialu cf cc1 br _isbranchfail;  /*not integer or multi*/

/*okay, it wasn't an intop, it was a multi, so we
must check out to see what sort of multi*/
instr memu lea intarg0, #4, intarg1;
instr memu ld intarg1, intarg2;
instr ialu exth intarg2, intarg1, intarg2;
instr ialu lsh intarg2, #-30, intarg2;
instr ialu ieq intarg2, #3, cc0;  /*true if not okay*/
instr ialu ct cc0 br _isbranchfail;
/*restore what you had, because it's really an int op*/
instr memu ld intarg0, intarg1;
instr ialu exth intarg1, intarg0, intarg2;  /*extract right half word*/
instr ialu lsh intarg2, #-19, intarg2;

_isbranchintop:
instr ialu imm #0x1f, intarg3;  /*mask*/
instr ialu and intarg3, intarg2, intarg2;
instr ialu ieq intarg2, #0x0f, cc0;
instr ialu ct cc0 br _isbranchyes;
instr ialu ieq intarg2, #0x0e, cc1;
instr;
instr;
instr ialu ct cc1 br _isbranchyes;
instr;
instr;
instr;
instr ialu br _isbranchfail;
instr;
instr;
instr;

_isbranchfail:
instr ialu jmp RETIP;
instr ialu imm #0, intarg0;
instr ;
instr ;
```

```
_isbranchyes:
instr ialu jmp RETIP;
instr ialu imm #1, intarg0;
instr ;
instr ;

/*this generates the next IP given an IP.
call as foo = nextip(IP) */

_nextIP::
instr memu ld intarg0, intarg1;
instr ialu exth intarg1, intarg0, intarg2;  /*extract right half word*/
instr ialu lsh intarg2, #-30, intarg2;
instr ialu ieq intarg2, #3, cc0;
instr ialu cf cc0 br _nextipdone;
instr ialu ct cc0 lea intarg0, #4, intarg0;
instr memu ct cc0 ld intarg0, intarg1;
instr ialu ct cc0 exth intarg1, intarg0, intarg2;  /*extract right half word*/
instr ialu lsh intarg2, #-30, intarg2;
instr ialu ieq intarg2, #3, cc0;
instr ialu ct cc0 lea intarg0, #4, intarg0;
_nextipdone:
instr ialu jmp RETIP;
instr ialu lea intarg0, #4, intarg0;
instr ;
instr ;
```

Bibliography

[1] Andrew Chen. Latch and register design for the map chip. Concurrent VLSI Architecture Memo 83, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, July 1996.

[2] William J. Dally, Stephen W. Keckler, Nick Carter, Andrew Chang, Marco Fillo, and Whay S. Lee. M-Machine architecture v1.0. Concurrent VLSI Architecture Memo 58, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, January 1994.

[3] William J. Dally, Stephen W. Keckler, Nick Carter, Andrew Chang, Marco Fillo, and Whay S. Lee. The MAP instruction set reference manual v1.4. Concurrent VLSI Architecture Memo 59, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, January 1997.

[4] Whay S. Lee. The rtl of the map network interface units. Concurrent VLSI Architecture Memo 00, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, February 1995.

[5] Whay S. Lee. The architecture of the map memory switch. Concurrent VLSI Architecture Memo 00, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, August 1996.

[6] Whay S. Lee and Marco Fillo. The architecture of the map intercluster switch. Concurrent VLSI Architecture Memo 86, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, April 1996.

[7] Albert Ma. An i/o port controller for the map chip. Master's project, Massachusetts Institute of Technology, EE/CS Department, May 1997.