DRAFT: Do not distribute

MIT Concurrent VLSI Architecture Memo 58

Massachusetts Institute of Technology
Artificial Intelligence Laboratory

M-Machine Architecture v1.0[1]

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

August 24, 1994

Abstract

This document describes the goals, architecture, and software model of the MIT M-Machine. Also described is the programmer's view of the machine including visible processor state, data types, memory system, and communication mechanisms. The instruction set is detailed in *The MAP Instruction Set Reference Manual.*

[1]The research described in this paper was supported by the Advanced Research Projects Agency and monitored by the Air Force Electronic Systems Division under contract F19628-92-C-0045.

Contents

1 Scope

This document describes the goals, architecture, and software model of the MIT M-Machine. Also described is the programmer's view of the machine including visible processor state, data types, memory system, and communication mechanisms. The instruction set is detailed in *The MAP Instruction Set Reference Manual.*

2 Overview

The M-Machine is an experimental parallel computer. It is being built to (1) experiment with mechanisms for parallelism, (2) investigate architectures and design techniques that make efficient use of 1995 technology, and (3) provide a vehicle to help develop parallel software systems.

Network: The M-Machine consists of a collection of nodes connected by a 3-D mesh network. Each network channel will operate at 800 megabytes per second[2]. The channels will use simultaneous bidirectional transmission. The network channels on the faces of the mesh (face channels) are available to connect to external devices. An I/O bus is available on each node to facilitate the connection of peripheral devices. These devices may be connected to either internal nodes, or to special I/O nodes attached to the face channels. The M-Machine is intended to be scalable to 64K nodes, but the design will be optimized for machines of 1K to 4K nodes.

Node: Each M-Machine node consists of a multi-ALU processor (MAP) chip and at least 1MW (8MB) of external memory (a total of five chips per node). As shown in Figure 1, a MAP chip contains four operation clusters, a network router, a network interface, a 4-bank, 16KW (128KB) internal cache memory unit, and internal communication switches. Each cluster includes two integer ALUs and a floating-point arithmetic unit with their associated register files, and a 1KW (8KB) instruction cache (for a total of 4KW (32KB)). The four clusters can communicate with one another through the Cluster Switch (C-Switch). One of the integer ALUs in a cluster serves as the interface to the memory system, and is termed the memory unit. Memory is accessed through the Memory Switch (M-Switch) and is byte addressed. However, only aligned words may be read from or written to memory. A message is composed in a subset of the general registers of a cluster. It is then launched atomically by a `SEND` instruction that is executed in that cluster. Messages arrive at one of two hardware priorities and are queued in a limited size buffer that is mapped to a register. The message contents may be extracted from the buffer by repeatedly reading that register. The priority 0 message queue may be accessed in cluster 2, while the priority 1 message queue is available in cluster 3. The nodes are expected to operate at 100MHz, and each function unit (except the divider) will be able to start an operation each cycle.

[2]One acceptable implementation has 16-bit channels, with each wire operating at 400 megabits per second.

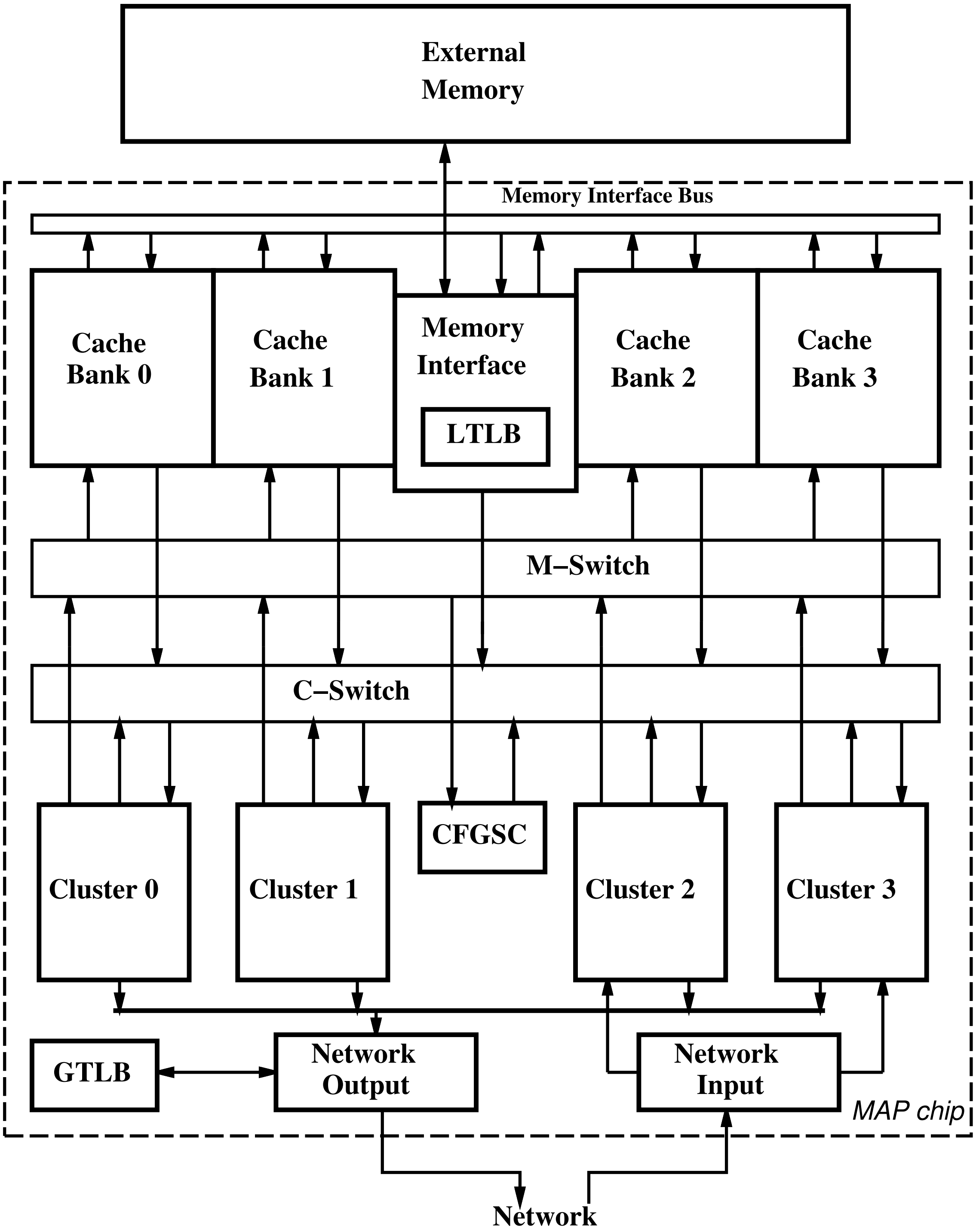

Figure 1: M-Machine node with Multi-ALU Processor (MAP) chip, external memory and network connection.

Intra-Node Mechanisms: The M-Machine provides hardware resources for *threads* that may run concurrently on the node. Each node contain resources for 6 vertical threads (`V-Threads`), each of which contains 4 horizontal threads (`H-Threads`). Four of the V-Threads are for application code. The fifth is reserved for critical system software, and the sixth handles exceptions that occur within each cluster. Each V-Thread consists of up to 4 H-Threads, each of which runs independently on a different cluster. V-Threads on the same node may communicate through memory, while V-Threads on different nodes communicate through messages. The system V-Thread is reserved for software message reception and for asynchronous handling of memory faults. The exception V-Thread consists of four exception H-Threads, each of which is responsible for handling processor faults that occur on its cluster.

All H-Threads within a V-Thread are assumed to share the same protection domain and to be loaded and unloaded simultaneously. Thus the compiler may schedule communication and synchronization between them. H-Threads may communicate through registers, writing results to one another's register files. Each H-Thread runs independently with its own instruction pointer on a different cluster. The compiler schedules code for a cluster as a VLIW. All operations in an instruction from a given H-Thread are issued simultaneously. At runtime, H-Threads from different V-Threads are interleaved over the cluster's execution resources. However, if only one H-Thread is present on a cluster, it may issue instructions on every cycle.

Data dependence between function units is enforced by using a presence bit associated with each register. When an operation is issued, its destination register is marked *not present*. When the register is written, it is marked *present*. Operations waiting on that register may then issue. Communication and synchronization between H-Threads within the same V-Thread are performed using two operations. Operations with destinations in remote clusters do not automatically mark their destination register *not present*. Instead, the H-Thread on the destination cluster must first execute an `EMPTY` operation to explicitly clear the register synchronization bit. Once this has been done, the source cluster can write the destination register without a race condition existing, as any attempt to read the destination register will block until the synchronization bit is set by the register write. The same synchronization scheme is used to pass data between the integer and floating point register files within a cluster.

Inter-Node Mechanisms: The M-Machine provides inter-node mechanisms for communication, synchronization, and naming. Communication is implemented using a `send` instruction that transmits a message to a remote node. The destination of a send may be translated from a virtual address using a TLB that is managed by software.

Synchronization is implemented using presence tags. A synchronization bit is associated with each word of memory. Certain load and store operations may specify a precondition and a postcondition on the synchronization bit. If the bit does not match the specified precondition, an exception handler is invoked. If the bit does match the precondition, or if no precondition is specified, the memory operation completes and the bit is set to the postcondition.

Memory resides in a global virtual address space, and virtual addresses are translated into local physical addresses with the aid of a local translation buffer (LTLB). Virtual addresses may refer to physical locations on other nodes. This naming is implemented using a global translation buffer (GTLB) to store associations between virtual addresses and node identifiers. Unlike a conventional TLB, entries in the GTLB may refer to data objects as small as a physical page (512 words) or as large as the entire address space. The GTLB provides support for mapping data objects across multiple nodes, allowing large data objects to be distributed throughout the machine.

The address spaces of individual V-Threads are separated and protected using segments and privileged access pointers. Each segment is a power of two in size and is aligned. Each pointer identifies its segment size with an internal six bit field. In user mode, a pointer can only be used to perform accesses within a given segment and to create other pointers within that segment. Pointers can be created and manipulated freely in supervisor mode.

Outline: The remainder of this document discusses the various programmer visible components of the MAP. Section 3 describes the software model of the M-Machine. Sections 4 through 11 discuss in turn the following mechanisms of the MAP: arithmetic and control flow, thread management, exception handling, memory, and communication. Section 12 outlines performance monitoring capabilities, and Section 13 presents a map of configuration space, which is the mechanism for supervisor threads to access internal registers of the MAP. Finally, Sections 14 and 15 describe the external interfaces to the MAP and the startup boot sequence.

3 Software Model

A systems programmer sees the M-Machine as a collection of *nodes*. Each node comprises a set of *thread slots* that can be used to run *threads*, and the corresponding memory. The M-Machine memory is logically viewed as a collection of *blocks* or lines. Blocks are grouped into *pages* and pages into *segments*. The software view of each of these elements is briefly described below:

memory block (line): the smallest contiguous unit of physical memory (nominally 8 words) on a single node for which status information is maintained.

node: a set of thread slots and memory blocks. A thread executing in one of a node's thread slots can access its memory blocks directly with read and write operations (if it is enabled to do so). The memory blocks on other nodes must be accessed via messages.

virtual page: a contiguous portion of global virtual address space described by a single entry in the local page table (or TLB). A virtual page may be mapped into a collection of memory blocks either within a single node or on separate nodes. The status bits in the local page table entry determine which blocks in the page are valid on the local node. One or more of the blocks within a page may be replicated – mapped to more than one physical memory block on different nodes. This is accomplished by nodes having distinct local local page table entries for the same page.

segment: the unit of protection – a contiguous portion of global virtual memory space that is accessible in user mode via a particular pointer. Segments are restricted to be a power of two words in size and aligned to their base address.

page group: a block of global virtual address space described by a single global page table entry. A page group may be mapped to physical memory on different nodes in an interleaved fashion.

thread slot: a set of hardware registers within a node that can hold the state of a V-Thread and supports instruction execution.

V-Thread: a task that can execute up to four independent H-Threads (under compiler control) when loaded into a thread slot.

H-Thread: a subtask within a V-Thread that may communicate through registers with other H-Threads within the same V-Thread.

The system software will hide the details of nodes, blocks, pages, and slots. Hence the applications programmer will see the machine as a collection of V-Threads and segments as shown in Figure 2.

A V-Thread is represented by up to 4 *instruction pointers* (IPs) which point to the next instruction to be executed for each of the subsidiary H-Threads, and a *context pointer* (CP) which points to the V-Thread's *context*, i.e., the memory region where the V-thread status and all of the H-Threads' registers are preserved when the thread is inactive (evicted from a thread slot).

Each H-Thread may access 32 programmer-visible registers, which are divided into 16 floating-point registers and 16 fixed-point registers. Using the preallocated space in the context segment, i.e., the *register block*, register values are saved and restored when the V-Thread is swapped in and out of a thread slot. If a context segment is larger than the register block, the remaining space can be used as temporary storage.

The code segment contains instructions and constant values that can be referenced relative to the IP. At any point in time, the IP points to the currently issuing instruction.

As illustrated in Figure 2, pointers within a thread's code or context segment may identify other segments – code segments, context segments, or general purpose data segments. Threads may share code segments without restriction. They may share context segments as well, but must do so with non-overlapping register blocks.

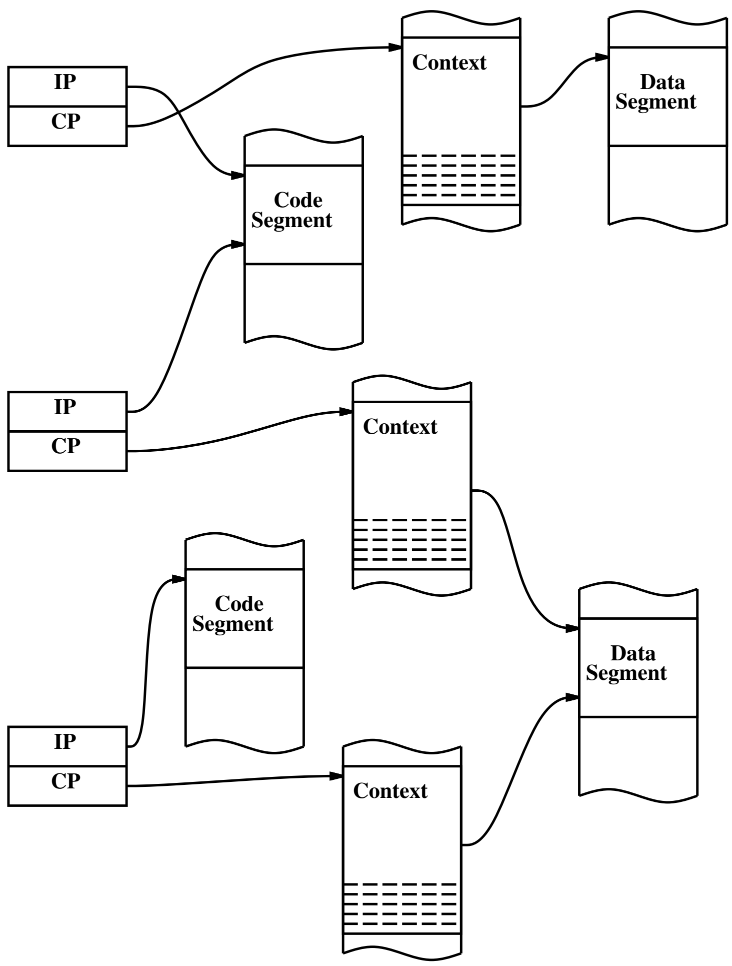

Figure 2: A programmer sees the M-Machine as a collection of V-Threads and segments. Each thread includes an instruction pointer (IP) and a context pointer (CP). Each pointer identifies both a segment and a location within that segment. The IP identifies the code segment and the current instruction within the segment. The CP identifies the context segment and the start of the register block within the segment.

4 Cluster Resources

The MAP arithmetic unit consists of 12 function units, divided into 4 clusters. The resources of a single cluster are used to execute the instructions of an H-Thread. This section describes the resources visible to a *single H-Thread* running in the cluster (note that a distinct set of registers for each V-Thread slot is implemented in hardware).

The internal structure of a cluster is shown in Figure 3. Each cluster has an integer unit, a memory unit, a floating-point unit, an integer register file (16 registers), and a floating-point register file (16 registers). The integer and floating-point units each have a write port into their respective register files, allowing concurrency of local data accesses. Operands must reside in the local register file, while results of arithmetic operations may be directed to either register file in the cluster or to a register file in a remote cluster using a remote destination specifier. Remote register writes and writes to the other register file in the cluster are transmitted via the C-Switch. Each cluster has access to 2 sets of condition code registers: a 4-entry read/writable set of local CCs, and an 8-entry global set (only two of which are writable). A condition code register may be used as a target of a comparison operation and as a source to conditional operations. The memory unit sends requests to the memory system via the M-Switch.

4.1 Registers

The register state accessible within a cluster consists of:

- General purpose registers
- Special registers
- Condition registers

4.1.1 General Purpose Registers

Of the 16 registers on each of the two register files on a cluster, an H-Thread can consider 14 integer and 15 floating point as general purpose registers (the remaining ones are special purpose). 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.

Both register files are divided into two portions: *public* and *private*. The private registers reside in the lower half of the register file, 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 write to a remote private register results in a write to the remote

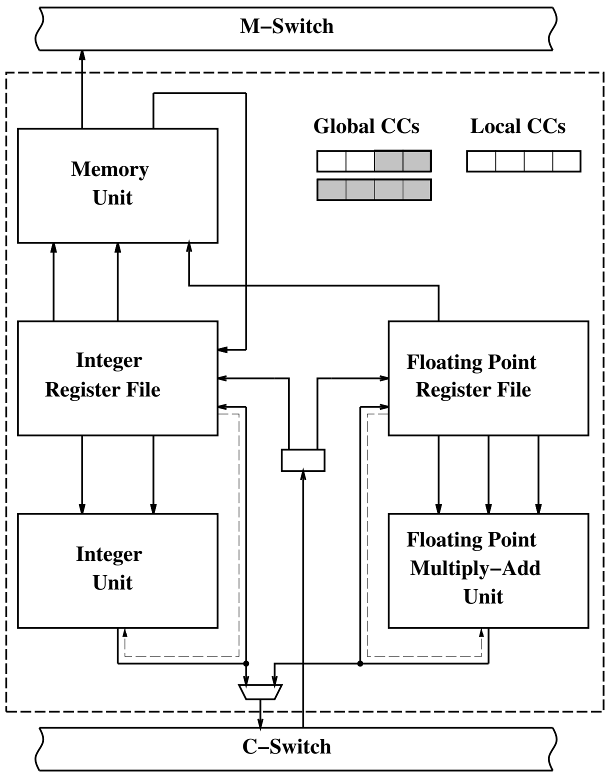

Figure 3: Cluster components including arithmetic units, a memory unit, register files, and connection to the M-Switch and the C-Switch.

public register whose identifier matches the 3 least significant address bits (e.g., a remote write to reg. *h2.i3*, results in a write to reg. *h2.i11*).

4.1.2 Special Registers

The special registers reside in the same name space as the general registers. Table 1 specifies the special register names and meanings.

Cluster	Register	Meaning
all	i0	Always zero. Writes to i0 disappear.
all	f0	Always fp zero. Writes to f0 disappear.
all	i1	Instruction Pointer
2 & 3	i14	Head (Event V-Thread only).
2 & 3	i15	Body (Event V-Thread only).
all	i4–i13 & f4–f13	User thread Message composition (MC).

Table 1: Special registers.

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 effect on the state of the register file. Register *f0* holds floating point zero, but otherwise has the same behavior as *i0*.

`IP` : Integer register 1 (*i1*) in all clusters ia a read only register that reflects the current value of the instruction pointer, which points to the instruction that the cluster is executing. This register may not be written directly, but is modified via *branch*, *jump*, or *fork* operations.

`HEAD` : The dedicated system V-Thread slot on each cluster has an associated *event* or *message* queue. The first word of the next message i.e., the *Head* is accessed by reading integer register i14. This flushes any remaining words from the current event or message.

`BODY` : This is mapped to integer register i15 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, zero is returned. The use of the event and message queues is further discussed in Section 9. User threads may access *i14* and *i15* as general purpose registers.

`MC` : Each user thread has 10 registers used for message composition. Outgoing messages are composed in these registers and then launched using a *send* instruction. (The number of MC registers is different in the system slot, more details are given in Section 11). These registers may also be used by the programmer as general purpose registers.

4.1.3 Condition Code Registers

Also available on each cluster are two sets of condition code registers one local and one global. Condition registers store single bit conditions which are set either by compare operations or special test operations. A synchronization bit is also associated to each CC register. The value of CC registers is set as a result of compare operations to either TRUE (= 1) or FALSE (= 0) based upon the result of the comparison. Additional test operations may also set condition registers.

The contents are used to conditionally execute subsequent operations. An operation may specify a condition register to test and a condition (TRUE or FALSE) that must be matched. If the condition is satisfied, the operation is executed. Otherwise, it is nullified. Four local and eight global condition registers are available in a cluster. As with general registers, no guarantee is provided as to the order of writes the same register without any intervening read.

Local Condition Code Registers Four single-bit Local Condition Code Registers are available on each cluster. These registers can be read, written and emptied only locally.

The synchronization bit of a local condition register is automatically marked invalid when a comparison operation is issued, and full when the result is available. A condition register value is readable by all function units in the cluster on the cycle after it is produced. There is no contention visible to the programmer.

Global Condition Code Registers Each V-Thread has also 8 global condition registers, which may be read by any H-Thread. However, only 2 of the condition registers may be written by any given H-Thread. These registers are physically replicated on each of the clusters. Writes to global CC registers are broadcasted, so that local copies are updated in the same cycle. However, emptying of global CC registers only affects their local copy and not the copy on any other cluster. This mechanism is particularly efficient for the implementation of an intercluster synchronization protocol in which each H-Thread, after reading the global CC value provided by another H-Thread, empties its own register.

Unlike local condition register writes, operations which write to a global condition register do not mark the register invalid when the operation is issued. However, they are marked valid when the operation completes and data is available; such validation is broadcasted to all clusters. In addition they can be emptied explicitly using the `CCEMPTY` operation. Table 2 shows a summary of the CC registers.

CC reg addr.	Type
0	Local
1	Local
2	Local
3	Local
4-7	INVALID
8	Global, writable by h0
9	Global, writable by h0
10	Global, writable by h1
11	Global, writable by h1
12	Global, writable by h2
13	Global, writable by h2
14	Global, writable by h3
15	Global, writable by h3

Table 2: CC register addressing

4.2 Function Units

4.2.1 Integer Unit

The integer unit executes arithmetic operations, address calculations, control flow operations, and communication operations. The result of an integer operation may be routed to the local integer register file or to a remote register file via the C-Switch. Privileged operations may only be executed in privileged mode and will cause a fault if executed in user mode.

4.2.2 Floating Point Unit

The floating point unit performs floating point calculations, including multiply, divide, and multiply and add. In addition, the floating point unit is responsible for executing integer multiply and divide, and fixed point multiply operations. Fixed point multiply takes two numbers with the binary point fixed to the right of the sign bit and multiplies them returning a fixed point result aligned in the same manner.

4.2.3 Memory Unit

The memory unit issues memory operations to the memory system. The memory unit is allocated two read ports and one write port on the integer register file. It also has a read port on the floating point register file to extract data for *store* operations. The address of a memory reference is a tagged pointer and must be specified in a single integer register. There are no complex addressing modes, except for *postincrement*, in which an offset is added to the pointer in the address register after the request is sent to the memory system. During `load` or integer `store` operations, while one of the integer register file ports supplies the address, the postincrement offset can be provided by the second read port, or it can be specified as an immediate.

For integer *store* operations the second read port supplies the value to be written. Thus, the postincrement offset can only be provided as an immediate in the instruction field. The write port in the register file is used to store the updated pointer value. The updated pointer must be stored in the same register in which the address was specified. If the offset or offset register is 0, no update is made. The result of memory *load* operations are delivered across the C-Switch, and do not use the memory unit's write port. Load operations that target the two special registers *i0* or *f0* do not return values across the C-Switch.

In addition to executing store operations, the memory unit can perform an explicit address calculation and a limited number of integer arithmetic operations without issuing a memory reference.

Certain portions of code and data (such as the LTLB event handler) must be always accessible even if the TLB and cache are locked up. The memory unit in cluster 1 has an extra set of pipeline shadow registers (called bypass registers) to support this requirement. The TLB event handler running in the system V-Thread on that cluster is hardwired to use the bypass registers.

4.3 Instruction Cache

Each cluster has its own local instruction cache. Each cluster has 1KW (8KB) of instruction cache. The instruction cache fetches instruction for the units. On a miss, it requests a new memory block to the memory system through the M-Switch. The instruction may be fetched directly from the on-chip cache. If the access misses there as well, the instruction must be fetched from off-chip memory.

In the event slot of cluster 1, where the TLB handler runs, the instruction cache uses the alternative memory bypass path to forward requests to the memory system. In this way, the instruction cache refill engine can fetch code even when the normal memory system path is blocked.

4.4 Multiple Threads

The resources of a cluster are time shared across multiple H-Threads from different V-Threads, each with their own state. On each cycle, the scheduling hardware selects an instruction that is ready to issue. An instruction contains one operation for each unit in the cluster (integer, memory, and floating point). An instruction is ready to issue only if the data dependencies of all its operations have been satisfied. This synchronization is performed using the presence bits in the registers. Instructions whose operands are not yet valid will stall without consuming any processor resources.

5 Programming a Cluster

This section describes the data types available to the programmer of the M-Machine, the format of instructions, and the operations that may be performed.

5.1 Data Types

The M-Machine supports the following data types:

- 64-bit signed integers
- 64-bit unsigned integers
- 32-bit unsigned integers
- 8-bit bytes
- 64-bit double precision floating-point numbers
- Pointers
- Instructions

5.1.1 Numbers

MAP pointers may address bytes, but only aligned words (64-bits) may be loaded or stored. A pointer specifies a particular byte of virtual memory and incrementing that pointer by one specifies the next byte in sequence. Bytes and halfwords may be inserted into or extracted from 64 bit words using special operations. Signed integers are represented in two's complement form. Floating point numbers are in 64-bit double precision IEEE format.

Words are 65 bits long, accounting for 64 bits of data and one tag bit that is used to identify pointers. All arithmetic operations are performed on 64 bit words but some operations are provided to help manipulate unsigned words and bytes.

An additional synchronization bit is associated with each word both in memory and in the register file. In memory, this bit is software–controlled and is used to test against preconditions and to store postconditions on memory operations. If the precondition specified by a memory operation is not met, a fault occurs. When the memory operation completes, the synchronization bit is set to the postcondition. In registers, the bit is manipulated by both hardware and software, and is used to synchronize operations, preventing an operation from issuing until its operands are available. Operations are provided to save and restore a value with its associated synchronization bit. Section 10 discusses this also in more detail.

5.1.2 Pointers

Pointers provide protection between threads from different users, and thus may only be created in a controlled manner and cannot be forged. The tag bit prevents users from forging pointers and gaining illegal access to protected code and data. Different pointer types are supported to restrict access to segments and to access different memory spaces. The segmentation scheme used to accomplish this is described in Section 10.

5.2 Operation Encoding

Each MAP *instruction* consists of three 32-bit *operation* fields. Each field corresponds to a function unit within the cluster. The first field is for the integer unit, the second for the memory unit, and the third for the floating point unit. Since each operation is targeted for a specific type of function unit, the encodings of opcodes can be overloaded. The exact mapping of operations to 32 bit quantities is presented in *The MAP Instruction Set Reference Manual*.

5.3 Instruction Packing

When in memory, instructions are packed so that NOPs need not be represented. The high two bits of each operation are reserved for the encoding. If the operation is the first encountered in the instruction, the high two bits encode the states shown in table 3.

State	Meaning
00	Instruction contains only an integer operation
01	Instruction contains only a memory operation
10	Instruction contains only a floating point operation
11	Instruction contains more than one operation

Table 3: Coding of the packing bits in the first operation of an instruction.

If the instruction contains only one operation, then the rest of the instruction is padded out with NOPs by the instruction fetch logic and the fetch IP is incremented by 4 bytes. If the instruction contains more than one operation, the high 2 bits of the next operation must be examined. Those bits encode the states shown in Table 4.

State	Meaning
00	Instruction contains I-M combination
01	Instruction contains I-F combination
10	Instruction contains M-F combination
11	Instruction contains I-M-F combination

Table 4: Coding of the packing bits in the second operation of an instruction.

Condition	Encoding	Function
UA	00	unconditionally execute always
UN	01	unconditionally execute never
CF	10	conditionally execute if FALSE
CT	11	conditionally execute if TRUE

Table 5: Condition codes.

For those instructions that contain only two operations, the instruction is padded with an additional NOP before proceeding to the pipeline, and the fetch IP is incremented by 8 bytes.

5.4 Control Flow

Control flow is changed by executing a branch (`br`) or jump (`jmp`) operation in the integer unit. Control flow changes have three delay slots; the next three instructions after the branch or jump will always be executed regardless of any branch condition.

5.5 Conditional Execution

All operations can be conditionally executed based on the contents of a condition register. This allows operations from both sides of a conditional control flow change, such as a branch, to be packed into the branch delay slots, improving both code density and performance. Conditional assignments can be easily implemented without executing branch operations. If the condition is not satisfied, the operation is nullified as it enters the execution stage.

All operations include a 5-bit field that specifies the condition under which the operation is executed. Two bits specify the test for the condition register; these states are shown in Table 5. The other three bits specify which condition register to examine. These may be local or global condition registers.

Operations which are nullified due to their test condition not being met have no side effects. The destination register is not marked *not present* and a value will not be produced.

5.6 Register Synchronization

Each register has a synchronization presence bit which is used for low level synchronization within a thread. An operation will not be issued until all of its source registers are present. For operations with a destination in the local register file, the destination is automatically marked *not present* when the operation is issued. Operations that have remote destinations must be preceded by an `EMPTY` operation in the remote cluster to explicitly mark the register *not present*.

The destination register is marked *present* one cycle before the operation completes so that a subsequent consuming operation can reach the execution pipeline stage and use the bypassed result. An operation can issue only if all of its data dependencies are satisfied, and all of the hardware resources it requires (e.g., the unpipelined divider) are available. Because different function units may have different pipeline latencies, this discipline ensures in-order operation issue, but not necessarily in-order completion.

5.7 Data Movement

The register set of an H-Thread is split between integer and floating point register files. In order to transfer data from one register file to another, an operation may specify a destination in the other register file. In order to guarantee synchronization, i.e., that the consuming operation waits for the data to arrive, the destination function unit must first execute an `EMPTY` operation to clear the synchronization bit for the register.

H-Threads within a V-Thread may communicate with one another through registers. The same mechanism used to transfer data between integer and floating point registers is used. The destination H-Thread must first mark the destination register empty. Then the source H-Thread executes an operation with a destination in the remote cluster.

Remote register writes proceed to the destination register file across the C-Switch. Results from load operations also proceed through the C-Switch. The C-Switch performs arbitration to decide which units gain access to the inter-cluster buses and register ports; thus a unit might stall to wait for these resources to become available. For correctness, the programmer is not responsible for resolving conflicts. However, performance may depend upon the contention for the shared buses in the C-Switch.

6 V-Thread Resources

A Vertical Thread (V-Thread) is a logical task of execution that may occupy a *thread slot* on the MAP. A thread slot holds the state of an active V-Thread and advances it by executing instructions. A V-Thread consists of up to 4 horizontal threads (H-Threads), each of which executes independently on one of the clusters. An H-Thread consists of a sequence of instructions to be executed on a cluster. Each instruction contains at most three operations to be issued simultaneously, one each for the integer, floating point, and memory units of the cluster. H-Threads within the same V-Thread may communicate through registers, while H-Threads in different V-Threads may only communicate through messages or memory. Multiple V-Threads are interleaved within each cluster on a cycle by cycle basis as shown in Figure 4.

H-Threads within a V-Thread are intended to execute in a loosely coupled fashion: they are activated and deactivated in a gang but may progress independently of each other. Synchronization and communication between them may occur through registers. Remote

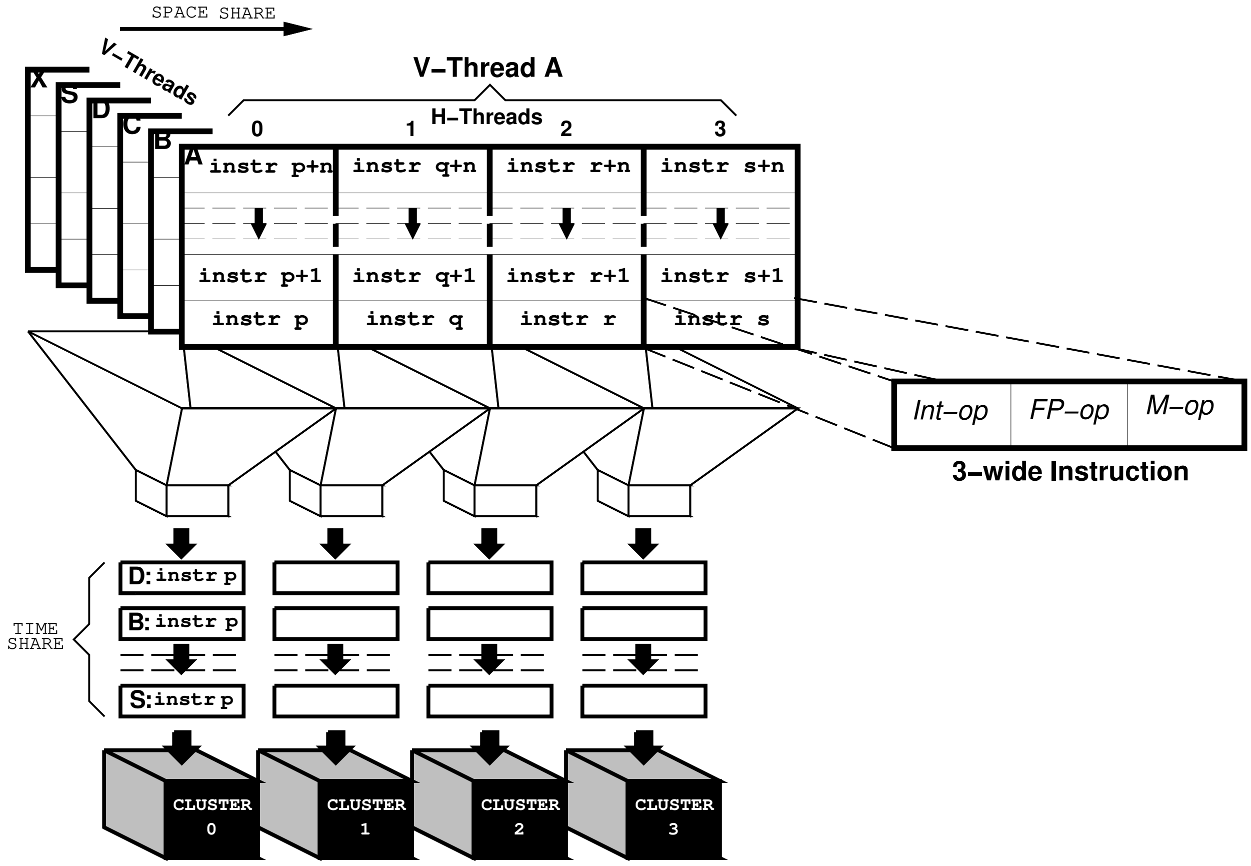

Figure 4: Multiple V-Threads are interleaved dynamically over the cluster resources. Each V-Thread has H-Threads which execute independently on different clusters.

register writes may occur without system intervention, because of the activation/deactivation constraint imposed on the V-Thread.

6.1 V-Thread State

The state of an active V-Thread is held in the registers of each of its H-Threads and by an entry in the Slot Status Table (SST). Each H-Thread's state resides its own particular cluster and consists of 16 integer registers, 16 floating point registers, 4 local condition code registers and two global condition code registers. The V-Thread is identified by a context pointer (CP) which refers to the region in memory where to store the state of the V-Thread when it is deactivated.

The SST entry consist of the following fields:

- An *Active* bit: indicates whether a V-Thread is currently occupying the slot.
- The *CP* of the currently active V-Thread, if any.
- A *Halt* bit: indicates whether any of the H-Threads are allowed to issue instructions.
- An *Acquired* bit: used to prevent conflicts of multiple acquisitions of a slot. This bit is tested and set atomically.
- An *Evict* counter: indicates whether or not the V-Thread may be evicted from the slot. It is incremented automatically when one of the H-Threads takes an exception. It is decremented manually by the exception handler when it terminates.
- 4 *Vacant* bits (one per possible H-Thread): indicates whether the corresponding H-Thread exists.
- 4 *Memory Barrier* counters (one per possible H-Thread): counts the number of outstanding memory load references.
- A *Stall Cycle Counter* (SCC): counts the cycles elapsed from the last instruction issued by any H-Thread. Reset to zero whenever an H-Thread issues an instruction.
- A *Stall Cycle Limit* (SCL): defines the maximum number of cycles that all the H-Threads can remain stalled. When the SCC reaches the SCL, a fault is generated.

In addition to the hardware state (SST entry and registers), some extra information about the V-Thread state may be kept in a software–managed structure (typically an entry in a list of running threads). Such list may include a priority field, a time of activation, and other status fields that may be used by the scheduler.

The state of a V-Thread that does not currently occupy a thread slot is maintained in memory. The register values are kept in the context which is pointed to by the CP. An existing V-Thread in a node has an entry in an appropriate scheduling queue (e.g. ready

V-Thread Slot	Cluster 0	Cluster 1	Cluster 2	Cluster 3
User 0	0	1	2	3
User 1	3	0	1	2
User 2	2	3	0	1
User 3	1	2	3	0
Event	0	1	2	3
Exception 1	3	0	1	2

Table 6: Logical mapping of H-Thread identifiers to clusters.

queue, blocked queue, etc.). The scheduling queue entry contains all the information required by the scheduler to determine when to activate a thread.

6.2 H-Threads

A V-Thread may create up to 4 H-Threads in order to exploit all of the node's clusters. When a V-Thread is initiated, only one H-Thread (called the *home*) is started. The home thread is responsible for forking any additional threads to other H-Thread slots. All of the H-Threads are managed "statically" by the compiler. The compiler names H-Threads explicitly and manages all communication between them. H-Threads are bound to clusters by the compiler. When an H-Thread is forked, it runs in the cluster assigned by the compiler.

6.2.1 Naming H-Threads

H-Thread identifiers are given relative to the home. This allows the home to be relocated to different clusters without having to manage dynamic allocation of H-Threads to clusters. Each V-Thread slot has a given home cluster (by software convention). The home H-Thread is assigned to the home cluster. Remote write operations specify the destination H-Thread with a wrap-around numbering relative to the current cluster (0 = local cluster, 1 = first cluster to the right, 2 = second to the right, 3 = third to the right). Table 6 shows the mapping of H-Thread identifiers to physical clusters for each of the V-Thread slots.

6.2.2 Static H-Thread Scheduling

The compiler is responsible for managing the use of the H-Threads and ensuring that all H-Threads are following the same general flow of control. To ease this task, the hardware provides limited synchronization support in the form of:

- *Inter H-Thread register communication*: any public register of a remote H-Thread register file can be specified as the target of local integer or floating point operation. For proper synchronization the remote write must be preceded by a local `EMPTY` on the destination H-Thread.

- *Global condition code registers*: the implementation of global CC register with broadcast writes and local `EMPTY` supports a generic and efficient synchronization mechanism decoupled from any algorithmic data dependencies.

The compiler must make use of these mechanism in order to ensure the required flow is followed by all H-Threads.

6.3 H-Thread Manipulation

When a V-Thread is created, only the home H-Thread is active (the other 3 H-Threads are vacant). The home H-Thread may create additional H-Threads within the V-Thread by executing a special fork operation. While running, H-Threads communicate through registers. Finally, an H-Thread suspends by executing a `HEXIT` operation, which updates the vacant bits in the V-Threads SST (slot status table) entry.

6.3.1 Forking H-Threads

H-Threads are initiated and terminated by two special thread management operations:

- `HFORK IP, id, ccdest`
- `HEXIT`

`HFORK` is used to initiate the execution of a new H-Thread. It executes the following sequence of actions, within the same V-Thread slot:

- all the register presence bits in the cluster identified by `id` are set to empty,
- the value specified by `IP` is copied into the IP register (*i1*) of cluster identified by `id`,
- the vacant bit in the SST corresponding to the new H-Thread is unset.

The `id` field is mapped to a cluster name by taking the current cluster number, adding the id, and using the low two bits as an H-Thread identifier. By convention, the parent thread may write the H-Thread identifier (and other required arguments) into one of the child thread's register after executing the `HFORK` operation, using some type of synchronization.

The `HFORK` operation writes a result into a condition register `ccdest`. If the operation succeeds, TRUE is returned to the designated condition register. Any H-Thread may fork additional H-Threads (not just the home thread). Only 4 H-Threads may be running within a V-Thread. If an H-Thread tries to fork another H-Thread into an already occupied H-Thread slot, FALSE is returned to the destination of the `HFORK` operation.

When an H-Thread suspends, it executes a `HEXIT` operation. This causes the vacant bits to be updated in the V-Thread's SST entry, and prevents the thread from issuing any additional operations. If an H-Thread executes a HEXIT and there are no other H-Threads active within the V-Thread, then the V-Thread slot is set to vacant and the slot may be reclaimed by the scheduler at a later time.

6.3.2 Register Communication

Communication between H-Threads takes place through registers. Any operation may write to a public register in another H-Thread by specifying a remote destination. The destination specifiers are relative to the current cluster. The register synchronization bits are used to synchronize H-Threads. Thus, in order to ensure proper synchronization, an EMPTY operation must be performed at the target H-Thread before the source H-Thread writes the register. This mechanism (in conjunction with the global condition registers) can be used to implement barrier operations for loosely coupled threads running on separate clusters. More tightly coupled H-Threads may synchronize through data dependencies using the same mechanism. H-Threads may only communicate through public registers. Private registers are guaranteed not to be overwritten by remote clusters so that fault handlers and subsystems can be guaranteed that the contents of certain registers are not altered by external agents.

6.3.3 H-Thread States

While a V-Thread is loaded into a thread slot, each of the clusters may be in one of the following states:

- Vacant: no H-Thread exists and no operations are being executed
- Running: an H-Thread exists and is issuing instructions
- Stalled: an H-Thread exists and is stalled on an empty register

The running and the stalled states cannot be differentiated through the the SST entry. However if all of the H-Threads of a V-Thread are stalled, then the V-Thread's stalled cycle counter (SCC) begins incrementing.

7 V-Thread Control

Conceptually, V-Threads can be in one of two generic states: either *active* or *inactive*. The simple distinction between this two states is made on whether the V-Thread occupies a hardware thread slot and or not. An active V-Thread occupies a V-Thread slot, an inactive one is queued in software queue waiting to be activated by the scheduler.

Each generic state will comprise of a set of more specific states and a set of conditions enabling the transitions among states. In this section we are only concerned with the mechanism involved in activation of V-Threads and with the hardware support required by active V-Threads.

7.1 V-Thread table entry

An V-Thread's entry in a scheduling queue contains the following information:

- The *CP* of the thread.
- A *Locked bit*: specifies whether or not the thread can be activated. This bit is normally cleared except when an event handler is writing to the context of the thread to update register values (the functionality of this bit is similar to *Evict* bit when the thread is active).
- A *Priority* field: which specifies the priority of the thread within the system
- A *Last deactivation* field: specifies the time the thread was last deactivated.
- A *Last activation* field: specifies the time the thread was last activated.

The entry is moved from queue to queue to indicate the state of a V-Thread. When the V-Thread is activated the entry is moved from the ready queue to the active queue and some of its fields are copied to the SST. As a thread executes, its entry in the active queue may become obsolete; upon deactivation the entry in the SST is copied back to the active queue before been transferred to the appropriate inactive queue.

7.2 Event and User V-Thread Slots

Each M-machine node (i.e., MAP chip) has 4 *user* slots, one *event* slot, and one *exception* slot. The user slots run general user threads, the event slot is reserved for asynchronously handling of events and messages, and the exception slot is used to handle synchronous exceptions detected in the clusters. Within the event V-Thread, two of the H-Threads are used for message handling (one each for priority 0 and priority 1 messages) and two H-Threads are for internal events (one for TLB miss events and the other for all other events).

In terms of hardware resources, the event V-Thread slot differs from the user V-Thread slots in that: (a) it is omitted from the scheduling algorithm, thus its thread is unevictable, and (b) event slot H-Threads have access to hardware resources not visible from other V-Thread slots, namely event queues, and incoming messages queues. Each of the two message handler event H-Threads has access to one of the two incoming message queues, one for priority 0 and one for priority 1 messages. Each of the two event H-Threads has access to one of two event queues (EQs) assigned to different event type. Event information is placed in the appropriate EQ automatically by the event detection hardware and handled by the event H-Thread dedicated to that EQ.

When an event (or message) queue is empty, the appropriate H-Thread in the event V-Thread stalls, holding register resources but using no computational resources. When an event occurs (or message arrives), the appropriate H-Thread resumes, the event is pulled out of the EQ and is handled appropriately. The synchronization bits in the registers mapped to the EQ are set when the EQ is not empty. When the EQ is empty, the synchronization bits are not set and the operation must wait until an additional word arrives in the queue.

The exception thread slot has a reduced register set and handles exceptions that are detected in the cluster, including arithmetic overflow and protection violations. An exception H-Thread is automatically dispatched by hardware when an exception occurs. Events and exceptions are discussed in more detail in Section 9.

User slots do not have event queues associated with them. Threads are scheduled into user V-Thread slots either by a software scheduler or by the FORK instructions described in the next section.

7.3 V-Thread Instructions

The instructions available to manipulate V-Threads are listed below and can be executed on any cluster. The only user-level instruction for V-Thread manipulation is:

- `VEXIT` : Terminates the V-Thread, by unsetting the ACTIVE bit.

In order to fork a new thread, the program must be in privileged mode. The following privileged instructions can be executed to directly manipulate threads executing in thread slots:

- `VGSLOTU dest, ccdest` : Acquires a user V-Thread slot. If a user V-Thread slot is available (both the *ACTIVE* and *ACQUIRED* bits are clear), a configuration space pointer to that slot (representing the memory address at which the resources of that V-Thread slot are mapped, see sect. 13) is returned in register `dest` and TRUE is returned in the condition register `ccdest`. If no user slot is available, FALSE is returned in `ccdest`.

- `VACQUIRE slot, dest, ccdest` : Acquires the V-Thread slot specified in `slot`. If the slot has not been acquired already, a configuration space pointer to that slot is returned in register `dest` and TRUE is returned in the condition register `ccdest`. Otherwise FALSE is returned in `ccdest`.

- `STTPROBE cp, dest, ccdest` : probes the Slot Status Table in an associative manner to determine if a particular V-Thread is present in any of the slots. The `cp` arguments specifies a a particular V-Thread context pointer, which uniquely identifies the V-Thread. If the particular V-Thread is found active and unacquired then the condition register `ccdest` is marked TRUE and the corresponding configuration space pointer is written into the general purpose register `dest`. If no match is found then a FALSE is returned into the condition register `ccdest`.

7.4 V-Thread Manipulation

7.4.1 Creation

A user level program may create a V-Thread by calling a privileged FORK procedure. In privileged mode, the thread manipulation operations may be accessed directly.

First, a new context must be allocated for the new thread. The context may be retrieved from a context manager or may be synthesized in some other way. The `VGSLOTU` operation can then be used to acquire a user thread slot. This sets the *ACQUIRED* bit in the slot, preventing it from being acquired by another task. The procedure then downloads all of the arguments to the new thread into the registers, and then updates the *CP*, vacant bits, and counters in the SST. Finally it sets the *ACTIVE* bit to start the thread executing and unsets the *ACQUIRED* bit. Figure 5 demonstrates this process.

```
GVSLOTU Rx, CC1 ;; Rx gets pointer to configuration space.
IF (CC1) THEN
  ;; Store operands in register file via config space.
  STORE CP,[Rx, CP_offset]   STORE IP,[Rx, IP_offset]
  STORE Op1,[Rx, Op1_offset]
  STORE Op2,[Rx, Op2_offset]
     .
     .
  STORE TRUE,[Rx, Active_bit_offset]
  STORE FALSE,[Rx, Acquired_bit_offset]

ELSE
  Software_Thread_Scheduler
```

Figure 5: Forking a thread using GVSLOTU.

7.4.2 Termination

A V-Thread can suspend by calling a privileged subsystem to save the registers back to the context and execute `VEXIT`, clearing the *ACTIVE* bit. If the thread need not save state, it may execute `VEXIT` directly. The thread slot may be reclaimed later by the scheduler. Alternatively, the scheduler may be called immediately to load a new thread from the ready queue.

7.4.3 Swapping Threads

A V-Thread can be swapped out by the thread manager or a fault handler. To do this, the `VACQUIRE` instruction is used to acquire a thread slot. The acquired thread slot can then be stalled by clearing the *ACTIVE* bit and allowing the pipeline to drain by waiting for a fixed number of cycles. The state of the old thread may be saved to the old context, and the state of the new thread may loaded from the new context. These registers must be saved and restored using the memory operations that preserve the synchronization state. Finally the *ACTIVE* bit is set to start the thread, and the *ACQUIRED* bit is cleared. Figure 6 shows pseudocode for swapping two threads.

```
VACQUIRE slot1, Rx, CC1 ;; Rx gets pointer to configuration space.
IF (CC1) THEN
  LOAD [Rx, CP_offset], CP1
  ;; Store registers into the context.
  STORE R1,[CP1, Reg1_offset]
  STORE R2,[CP1, Reg2_offset]
     .
     .
  STORE Rn,[CP1, RegN_offset]

  LOAD [CP2, Reg1_offset], R1
  LOAD [CP2, Reg2_offset], R2
     .
     .
  LOAD [CP2, RegN_offset], Rn

  STORE TRUE,[Rx, Active_bit_offset]
  STORE FALSE,[Rx, Acquired_bit_offset]

ELSE
  Try Another Slot
```

Figure 6: Swapping a thread using VACQUIRE.

If an active thread remains idle for too long (the *SCC* exceeds the *SCL*), a thread timeout event occurs. This event invokes an exception handler that can schedule another V-Thread, thus avoiding a potential deadlock situation. Deactivating a V-Thread is expensive due to the register state that must be saved back to the context. Thus, evicting a V-Thread should not be a common action.

8 Exceptions

An *exception condition* within a cluster occurs when an operation that normally would be able to correctly complete cannot do so on its own. Depending on its type, an exception may be handled immediately, or it may be deferred. If the exception is to be handled immediately, the operation that caused it is terminated before writing any results and an exception handler is automatically invoked. The exception handler executes on the cluster that experienced the exception, and uses the execution resources of the cluster's *Exception*

Bit locations	Purpose
0–3	Excepting Cluster id
4–6	Excepting V-Thread id
7–13	Integer Unit exception code encoding
14–20	Memory Unit exception code encoding
21–27	Floating Point Unit exception code encoding

Table 7: S Register Format.

Register	Mapped to
$i0$	Integer 0
$i1$	Exception Handler IP
$i2$	(General Purpose Integer Register)
$i3$	(General Purpose Integer Register)
$i4$	FIP register
$i5$	S register
$f0$	Floating Point 0
$f1$	(General Purpose FP register)
$f2$	(General Purpose FP Register)
$f3$	(General Purpose FP Register)

Table 8: Exception Handler Registers

V-Thread. Exceptions that are deferred result in a special pointer, known as an *errval*, being written into the destination register of the operation that caused the exception. The *errval* can be checked manually or later cause an exception handler to be invoked automatically if the errval is used as an input to an particular set of operations.

8.1 Exception H-Thread

A cluster's *Exception H-Thread* belongs to the MAP's Exception V-Thread, and has a reduced set of registers: 4 integer registers, 3 floating-point registers, and 4 condition registers. Two fault state registers are mapped into the integer register file. The *FIP* captures the IP of the faulting instruction. The S (*status*) register contains a status word indicating the types of exceptions raised. Table 7 shows the format of the S register. Multiple operations within a single instruction may cause exceptions to occur; these are all signalled simultaneously in the S register. Operations in the excepting instruction that did not cause an exception complete normally. Table 8 summarizes the registers accessible in an exception H-Thread. Integer registers $i4$ and $i5$ can also be used as general purpose registers, provided that the exception handler saves away the FIP and S values if they are needed later.

8.2 Exception Handling

When an exception occurs, the *Exception Bit* in the cluster is set, which prevents the user H-Threads within the cluster from issueing additional operations. User H-Thread instruction issue is prohibited since only one exception at a time may be serviced within a cluster. Thus, only the *Event* and *Exception* H-Threads in the cluster can make forward progress. If either of these H-Threads cause an exceptional condition, a *catastrophic* exception occurs.

The exception handler can determine the fault(s) that occurred by examining the S register, and is responsible for handling *all* fault conditions within the faulted instruction (integer, memory and floating-point units) before terminating. The registers of the thread that cause the exception can be examined and written through memory mapped locations within *configuration space*. Configuration space is described more completely in Section 13. The conditions that cause the exception handler to be invoked include:

- *errval* input to comparison operation
- Store to *errval*
- Store using illegal pointer type
- Illegal opcode
- Execute privileged opcode while in non-privileged mode
- Branch or jump to illegal target
- Send with invalid destination specifier
- Send with invalid handler IP

Illegal control flow changes (such as out of range branch targets) do not cause an exception to occur immediately. Instead, an illegal instruction is placed in the first instruction following the last branch delay slot. The exception handler will be invoked immediately after all of the instructions in the delay slots are executed.

When the exception handler completes, it clears the exception bit (through configuration space), which allows the cluster again execute instructions from the user H-Threads. The faulted H-Thread, if still active, resumes execution at its current IP.

8.3 Catastrophic Exceptions

A catastrophic exception occurs if an exception is detected in an Exception or Event H-Thread. When a catastrophic exception occurs, the instruction pointer of the event causing the exception is saved in the FIP slot of the faulted thread's register file, and a fault indicator is saved in the S register slot. The hardware then forces the faulted H-thread to branch to a *catastrophic exception handler*, whose instruction pointer must be set up at boot time via the configuration space.

Note that a catastrophic exception taken in an event H-Thread is handled in the event H-Thread, not in the exception H-Thread. This is because the exception handlers may cause events to occur. Handling the catastrophic exception in the exception thread slot would create a cyclic dependency between the event and exception H-Threads that could lead to deadlock.

As the name reflects, a catastrophic exception indicates a gross error in the system software. In the case of an catastrophic exception within an exception slot, the state of the original fault is likely to be lost. In the case of an event handler error, the original event causing the exception will not be completed.

8.4 Deferred Exceptions

A deferred exception writes an *errval* into the destination register of the operation that caused it to occur. The *errval* is a tagged pointer–a primitive data type of the MAP–with the pointer fields set to the following:

- Permission = *error*
- Length = type of error
- Address = virtual address of instruction that created the errval

The pointer format is discussed in detail in Section 10. When a errval is seen as an input to an operation, either an exception is raised immediately, or the errval is written to the destination register of the operation. This behavior will be detailed on an instruction by instruction basis in the MAP Instruction Set Architecture Reference Manual.

9 Events

An *event* occurs when an action outside of the cluster cannot be completed without software assistance. This includes events in the memory system such as synchronization miss faults, and communication events such as message arrival. When an event occurs, information regarding the event is placed in an *Event Queue* (EQ) or Message Queue (MQ) in the cluster that is dedicated to handling that particular type of event. The *Event H-Thread* in that cluster, which belongs to the *Event V-Thread* of the MAP, reads events from the queue and handles them. Local translation lookaside buffer (LTLB) misses are handled specially in a dedicated cluster.

9.1 Event Queue

The event queue is a hardware FIFO in cluster 0 which holds event records. The types of events that create entries in the event queue include:

- Synchronization failure
- Block status faults
- Global translation lookaside buffer (GTLB) misses
- Thread timeout

When an event occurs, and event queue is written automatically by the hardware event detection mechanisms. Multiple requests to insert an event queue entry occurring simultaneously will be sequentialized by hardware. In addition, the event queue may be written through configuration space. This queue is read via 2 special registers, R_{head} and R_{body}, which are mapped to integer registers $i14$ and $i15$ of each Event H-Thread. The R_{head} register returns the first word of the next unread event entry in the event queue. Any words left over from the previous event entry in that queue are flushed at the same time. Each read to R_{body} then pops and returns one word from the event entry at the front of the corresponding queue. If the queue is empty, any attempt to read R_{head} or R_{body} stalls the operation until a word is available. If R_body is read beyond the message length, an errval is returned. Condition register 0 of the event H-Threads indicates whether the queue is empty: 0 if empty, 1 otherwise.

Each entry in the EQ begins with an event identifier number, followed by some number of arguments. The Event H-Thread within the Event V-Thread is responsible for reading the corresponding Event Queue and calling the appropriate *event handler* code, which is determined from the event number.

The event queue FIFO is 96 words long, and each event entry is 4 words. Queue overflow can be prevented by blocking user threads from issueing memory requests when the event queue has more than 7 entries. This will allow enough room in the queue should all 17 potential outstanding memory requests require entries in the event queue.

9.2 TLB Miss Event

The LTLB miss event is somewhat different from the rest of the events. Since only one LTLB miss may be outstanding at a time, no queue is necessary and the event record is written directly into registers i2, i3, and i4 of the event H-Thread in cluster 1. No queue access through i14 and i15 is required.

When a LTLB event occurs, the external memory interface (EMI) stops accepting requests from the cache banks. The cache banks continue operating normally as long as no subsequent cache misses occur. A cache bank that sustains a cache miss stops accepting requests until the EMI has accepted the request from the cache bank to handle the cache miss. This mechanism uses backpressure through the memory system to block instruction issue should all of the cache banks become blocked. A special bypass path, used only by the LTLB miss H-Thread, is used to allow LTLB misses to be serviced even if the memory system is blocked.

9.3 Message Queues

Message queues are identical to the event queue, except that they are filled by the network interface. Thus message handlers may behave similarly to the event handlers. The message queue in cluster 2 is used to hodl arriving priority 0 messages, while the queue in cluster 3 holds priority 1 messages. Message handling is discussed in further detail in Section 11.

10 Memory Hierarchy

Each M-Machine node contains 1MW of data in off-chip physical memory. An on-chip data cache is controlled by hardware to hold recently referenced data. The on-chip cache hold both instructions and data. Instructions are fetched into each cluster's local instruction cache by cluster instruction cache refill engines before the code may be executed. All other data references go to the data cache.

Accesses that hit in the cache return data in 3 cycles. Cache misses require a variable amount of time depending on whether the page containing the data is mapped into the local memory of the node. Translation Lookaside Buffers (TLBs) are used to expedite resolution of references that miss in the cache.

The M-Machine supports a single global virtual address space. Segmentation is used to implement protection, while paging is used to manage the relocation of data in physical memory within the virtual address space. Virtual to physical translation is used to resolve references that reside in the memory of the local node as well as to determine the resident node if the data is remote. The segmentation and paging mechanisms are independent so that protection may be preserved on flexibly sized segments of memory.

10.1 Segmentation

Memory access is performed using an MAP guarded pointer, which provides capability style protection without segment descriptors. A guarded pointer is a primitive MAP data type and is shown in in Figure 7. A guarded pointer contains protection information, the segment length, and a virtual address within the segment. The one bit pointer tag prevents a user from illegally altering the pointer and from arbitrarily creating pointers out of integers. Because the segment lenght is encoded logarithmically, segments must adhere to the following constraints:

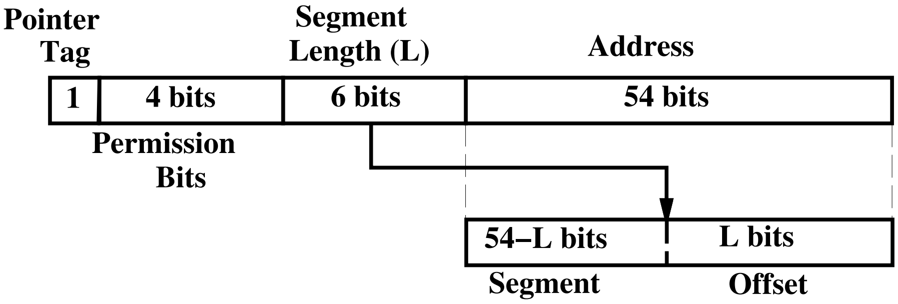

Figure 7: Pointer Format

- Power of 2 words in length
- Segment base aligned to segment size (The base address a 2^m word segment must have its $m+3$ lowest-order bits equal to zero[3])

A memory reference that attempts to store using an illegal pointer or a non-pointer address will cause an exception. A load using an illegal pointer will return an errval (see Section 8). A user may alter the address field of a pointer by using the `LEA` operation, or by using the post-incrementing feature of *load* and *store* operations. When the operation is executed, the new address is checked against the lower and upper bounds of the segment. If the new address falls outside of the segment, an errval is created. All of the same exception conditions exest in priviliged mode. However a privileged `SETPTR` operation may be used to tag any word as a pointer.

10.1.1 Permission Checking

The permission fields in the pointer indicate how a thread may access data. Four permission bits are used to encode the following states:

- *Read Only*: Data may only be read.
- *Read/Write*: Data may only be read or written.
- *Execute User*: Data may be read or executed as code, but not written. The thread is in user mode.
- *Execute Privileged*: Data may be read or executed as code, but not written. The thread is in privileged mode.
- *Execute Physical*: Data may be read or executed as code, but not written. On cache misses, the TLB is not accessed. The thread is in privileged mode.

[3]The low order three bits of both pointers and offsets specify a byte within a word and are ignored by the memory system, which performs only word-aligned loads and stores.

- *Enter User*: The pointer may not be altered, but a thread may execute starting at the address.
- *Enter Privileged*: The pointer may not be altered, but a thread may execute in system mode starting at the address.
- *Enter Message*: Code at this address may be executed in a message handler. A *send* operation faults if the designated IP is not in this state.
- *Configuration Space*: Indicates that the address refers to an internal register in the MAP.
- *Key Only*: The pointer may not be modified or referenced in any way. The contents are a secure key that may be passed around by users, but may not be forged.
- *Physical*: The pointer references data in physical memory on the local node. This bypasses the virtual memory system, ignoring the LTLB on cache misses. If the address exceeds the size of local physical memory, the top bits are ignored.
- *Errval*: The pointer has been generated by a deferred exception. Any attempt to use an Errval pointer as an operand will cause an exception.

Pointers in the *physical* state may access the cache. If the cache misses, then the LTLB is bypassed and external memory is addressed directly. However, the system programmer must take care to avoid aliasing in the cache. This may be avoided by not mapping the virtual address space that overlaps the physical address space of the MAP.

10.1.2 Segment Length

Each pointer contains a six bit segment length field that holds the log base 2 of the size of the segment in which the address resides. Thus, segments may be of any power of 2 size between 1 and 2^{54} bytes. This encoding allows the base address and the extent of a pointer's segment to be determined without accessing any additional information. User-level instructions that manipulate pointers (`lea`, `load`, `store`) have the lower and upper bounds of their segment checked automatically to ensure that the operation does not access memory outside of the allowed segment.

This segmentation and access control system provides flexibility to the user, while still permitting strictly enforced security. Segments can be overlapped and shared as long as each segment is aligned to an address that is a multiple of its size. Since all of the necessary segmentation information is contained within each pointer, a separate table of segment descriptors is unnecessary. More importantly, instructions need not access such a table to check segmentation restrictions on memory accesses. Also, access to system functions and other routines can be given to non-trusted programs, as the *enter-privileged* and *enter-user* permission states ensure that a user may only execute code starting at the specified entry

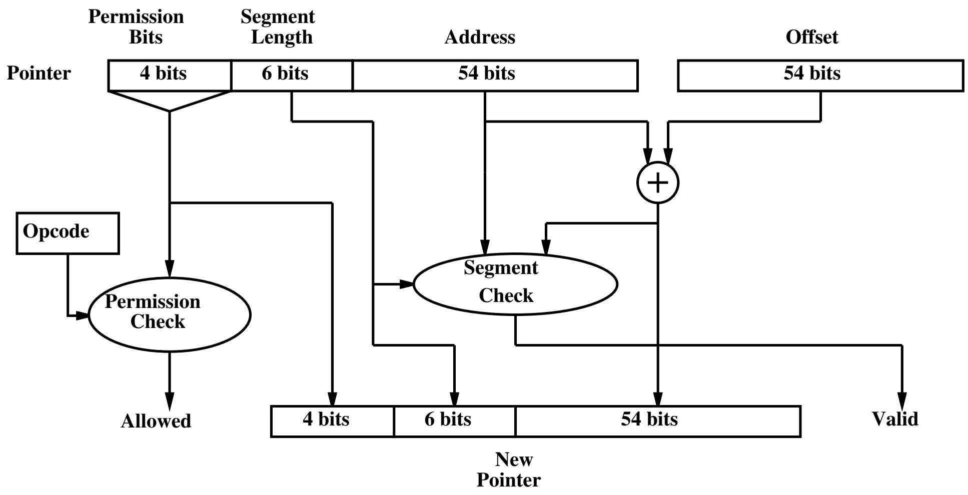

Figure 8: M-Machine Virtual Address Computation: A virtual address may be created using a LEA operation or by post-incrementing a pointer.

---

point. A MEMBAR (memory barrier) instruction is used to block further instructions from executing until all outstanding memory references have completed.

Figure 8 illustrates the computation of a new pointer, given an old pointer. Both permission and segment bounds must be checked.

10.2 Paging

There MAP employs two translation levels. The first level (*local translation*) attempts to determine a physical address in a local node's memory from the virtual address. The second level (*global translation*) determines the node location of the data from the virtual address.

10.2.1 Local Translation

Local pages have a fixed 512 word size. A local translation lookaside buffer (LTLB) is read in hardware on a cache miss. Like traditional TLBs, each LTLB entry associates a fixed-sized page of virtual address space with the page frame of a local physical page. As shown in Figure 9, each entry in the LTLB consists of three 64-bit words and also includes status bits for each memory block. A block is 8 wordsand corresponds to the cache line size. These status bits may be used to implement relocation and replication of data on a block by block basis. For example, they may be used for software cache coherence. The allowed states are defined as:

- INVALID: Any attempt to reference the block causes a block status event.

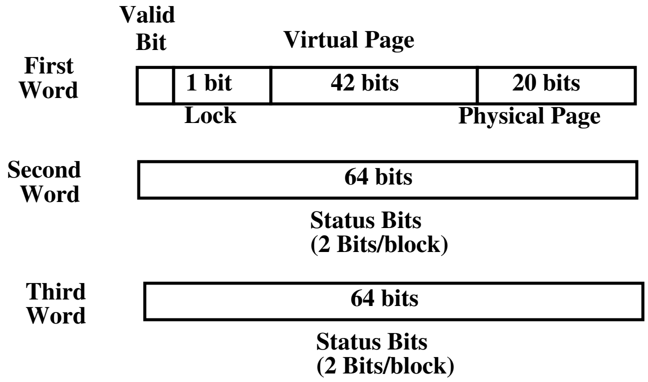

Figure 9: LTLB Entry Format

- READ ONLY: The block may be read, but a block status event occurs if a write is attempted.
- READ/WRITE: Reads and writes to the block are permitted.
- DIRTY: The node has an READ/WRITE copy of the block that has been altered.

These states allow a variety of relocation and replication schemes to be implemented efficiently, by handling the common case (the user attempting an access which is allowed) in hardware while giving the user the ability to determine how illegal accesses are handled.

Figure 10 shows the permissions check being performed on a pointer before the address field is sent to the cache banks and the virtual memory system.

10.2.2 Global Translation

The global translation mechanisms allow large portions of virtual address space to be mapped over subcubes of MAP nodes in the M-Machine. With global translation tables (accelerated by a global translation lookaside buffer GTLB), different nodes may use the same names to identify the same data, regardless of where it resides.

Global pages (page groups) are of variable size and may be interleaved over a subset of M-Machine nodes. The global translation mechanism maps virtual addresses to node identifiers based on the layout of the global page. A Global Translation Lookaside Buffer (GTLB) is attached to the outgoing network interface (NETOUT) and is read by the hardware when a message is sent. An event occurs if the appropriate entry is not found in the GTLB.

Each GTLB entry specifies a mapping between a variable-sized page group of virtual address space and a set of M-Machine nodes. Each region must be a power of 2 local pages in length and aligned to its size. Page groups must be at least 512 words and are specified by their base address and the log of their length in local pages.

As shown in Figure 11, a GTLB entry contains the following fields:

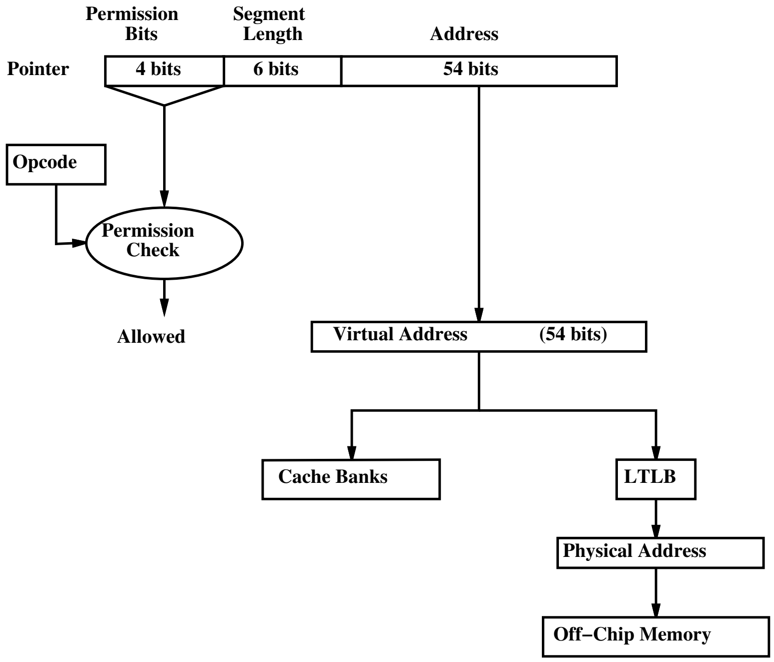

Figure 10: M-Machine memory access: A permission check is performed before the address is sent to the cache.

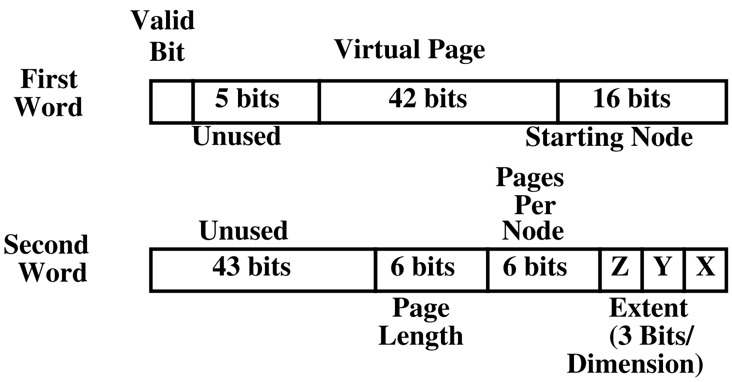

Figure 11: GTLB Entry Format

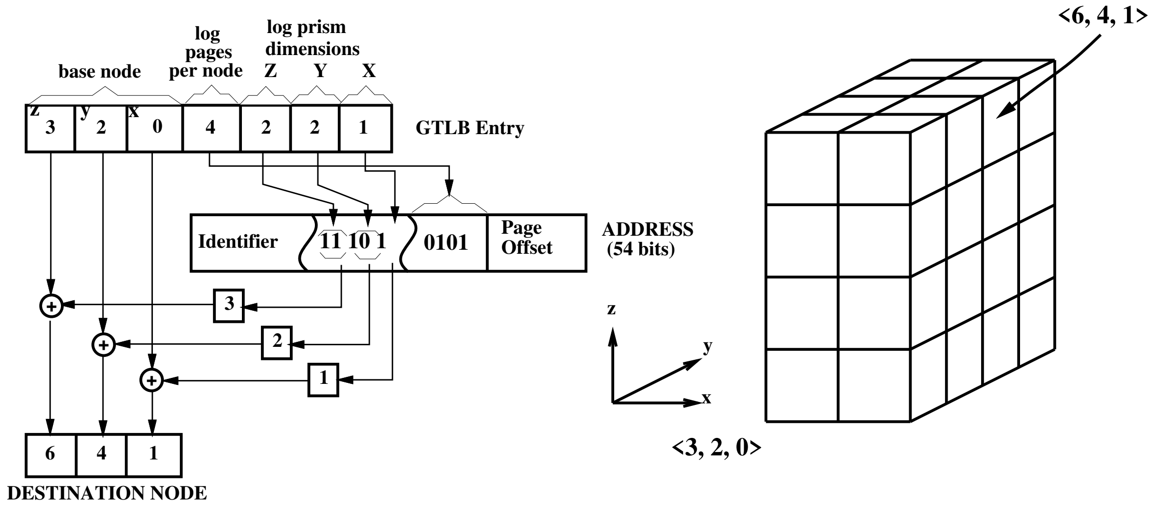

Figure 12: Example of GTLB translation.

- 42-bit virtual page number
- 16-bit starting node number
- 6-bit page length field
- 6-bit pages per node field
- 3 3-bit extent fields

The extent fields allow a page group to be mapped across a cube of nodes by specifying the log of the length (in nodes) of each side of the cube. The pages per node field defines the number of local pages of data mapped to each node in the prism. The length field specifies the log of the length of the global page in number of local pages.

Figure 12 shows how a virtual address is translated into a node number by the GTLB hardware.

Local and global TLB entries are memory-mapped into the configuration space of the M-Machine, and are written by software. The software is responsible for ensuring that the TLB entries are correctly formatted; the hardware performs no checking and will behave unpredictably with incorrect TLB entries. To prevent TLB entries that are being written by a system thread from being read during a memory reference, a single synchronization bit is used from a part of the TLB entry. The system thread must explicitly set and reset the bit to acquire and release exclusive access to the entry.

10.3 Synchronization

In order to implement synchronization, each M-Machine word in memory has one synchronization bit associated with it. Special load and store operations may specify preconditions and postconditions for the synchronization bit (`LDS, STS`). If the synchronization bit does not match the precondition bits in the memory operation, a synchronization event occurs. Otherwise, the operation completes and the synchronization bit is set to the postcondition of the operation. Additional synchronizing operations exist that set a condition code register to TRUE and complete the operation if the synchronization bit matches the precondition, and set the condition code register to FALSE instead of faulting if the synchronization bit does not match the precondition.

The precondition field of a synchronizing memory reference consists of two bits. This provides the user capability to specify any synchronization state or combination of states. Setting these bits to 10 causes the sync check to pass if the sync bit is zero. Setting them to 11 passes if the sync bit is one, while a precondition of 00 always passes, and a precondition of 01 always fails. The postcondition field is only one bit and specifies the state in which to set the synchronization bit after the reference completes.

These state bits can be used to implement a variety of synchronization policies and schemes, including spin waiting for data to be produced, barrier synchronization, and producer-consumer relationships.

10.4 Special Instructions

The M-Machine provides a number of memory unit instructions that allow the programmer to influence how memory operations interact with hardware, by specifying which levels of the memory hierarchy are to be used to satisfy a memory request, or by examining or altering elements of the memory system that are inaccessible to user level programmers.

Uncached memory operations Uncached memory operations are used to stream data in from external memory without polluting the cache. Both loads and stores are provided. These are useful for transferring large blocks of data across the network.

- `LUC src1 src2 dest` : Loads the contents of the address specified in *src1* into the register specified in *dest*, adds the value of *src2* to the pointer in *src1*, and stores the result in *src1*. If the required data is in the cache, the operation fetches the cached value, and completes exactly as if a normal load operation had been executed. If the required data is not in the cache, it is fetched from the off-chip memory and returned into the specified destination, but the block containing the data is not brought into the cache.
- `STUC src1 src2 dest` : Stores the data specified by *src1* into the address specified by *dest*. Adds the immediate value in the *src2* field to the pointer specified in *dest*,

and stores the result in *dest*. If the address to be written is in the cache, the contents of *src1* are written into the cache, and the operation completes as if a normal store operation had been executed. If the location to be modified is not contained in the cache, the contents of *src1* are written directly to the off-chip memory without bringing a line into the cache.

Cache management instructions The following instructions are provided to give the programmer control over the state of the cache:

- `FLNE src1 src2` : Removes the cache line containing the address in *src1* from the cache. If the line is dirty, it is written back to main memory. Post-increments src1 by the contents of src2.
- `FCA` : Invalidates the entire cache. This is a privileged operation.
- `GETCSTAT src1 dest` : Reads the cache status bits for the line containing the address in *src1* and places the result in the low two bits of *dest*.
- `PUTCSTAT src1 src2` : Writes the value specified in *src2* into the cache status bits of the block containing the address specified in *src1*. This operation is privileged.

TLB access The following instruction is provided to allow programs to determine if a translation for a virtual address exists in the GTLB:

- `GPRB src1 dest1 dest2`: Sets the condition register *dest1* and the destination register *dest2* based on the state of the GTLB entry containing the address in *src1*. The states are as follows:
  - *dest1* = TRUE: A translation for *src1* is found. The node ID of the node containing *src1* is returned in *dest2*.
  - *dest1* = FALSE: A translation is not found.

  This operation is intended to be used in LTLB miss handlers, to allow the handler to quickly determine where the appropriate location resides.

10.5 Memory Operation Execution

When a load or store instruction is executed, the permission bits of the address pointer are checked to determine if the access is permitted. If the pointer does not allow the access, an exception occurs. Otherwise, the request is transferred across the memory switch (M-Switch) into the appropriate cache bank.

When a memory request arrives in a cache bank, the cache directories are examined for the presence of the requested line. If the line is not present, the request is forwarded in hardware to the external memory interface (EMI) to be handled. If data is present in the cache, both the synchronization bits on the word and status bits on the line are checked. At this time an event may occur if the synchronization precondition is not matched, or the status bits require a fault. If no event occurs, the synchronization bits for that word are set to the postcondition, and (for a *load*) the data is routed to the appropriate cluster via the cluster switch (C-Switch).

When a request arrives at the EMI, the LTLB is searched for a virtual to physical translation. If a translation is found, the EMI loads the line containing the referenced address from external memory into the cache, and then handles the request as if a cache hit had occurred. If no translation is found, a fault to software occurs. The fault handler for an LTLB miss is then responsible locating the data (whether is be local or remote), retrieving it, and returning it to the requesting thread.

11 Communication

M-Machine nodes communicate with one another via a network. Threads send messages using explicit *send* operations, which transfer words in registers into the network. When a message arrives, it waits in a queue until it is extracted by a system level thread. The maximum message length is 12 words (destination, a dispatch IP, and maximum of 10 arguments). The message length and a *senderID* are packed into the message header by the network interface, and expended into a full word each at front of the message in the queue at the destination. The dispatch IP determines what code is invoked by the message extraction H-Thread at the destination node.

The network interface consists of a Network Output Unit (NETOUT) which transfers data out of registers into the network, and a Network Input Unit (NETIN) which queues incoming messages until they are extracted. Message may be sent at one of two priorities (0 or 1). Priority 1 messages are given preference for shared hardware resources.

11.1 Network Output

M-Machine messages are composed in a subset of the registers, called message composition registers (*MC registers*), and launched into the network with a `SEND` operation. There are two *banks* of MC registers per H-Thread, one each in the Integer register file and the Floating Point register file (named *mcA* and *mcB* respectively). Each MC bank has 10 registers mapped into the register files according to Table 9. The system threads have access to two additional MC registers for the purpose of returning entire messages to the sender. Threads running in the Exception slot do not have direct access to the message facility.

To send a message, a thread writes the arguments to be sent into the message composition registers in one of the banks and then executes one of the SEND operations in the following format:

User Slots		System Slot	
MCreg Bank A	MCreg Bank B	MCreg Bank A	MCreg Bank B
i4 = mcA.0	f4 = mcB.0	i4 = mcA.0	f4 = mcB.0
i5 = mcA.1	f5 = mcB.1	i5 = mcA.1	f5 = mcB.1
i6 = mcA.2	f6 = mcB.2	i6 = mcA.2	f6 = mcB.2
i7 = mcA.3	f7 = mcB.3	i7 = mcA.3	f7 = mcB.3
i8 = mcA.4	f8 = mcB.4	i8 = mcA.4	f8 = mcB.4
i9 = mcA.5	f9 = mcB.5	i9 = mcA.5	f9 = mcB.5
i10 = mcA.6	f10 = mcB.6	i10 = mcA.6	f10 = mcB.6
i11 = mcA.7	f11 = mcB.7	i11 = mcA.7	f11 = mcB.7
i12 = mcA.8	f12 = mcB.8	–	f12 = mcB.8
i13 = mcA.9	f13 = mcB.9	–	f13 = mcB.9
–	–	–	f14 = mcB.10
–	–	–	f15 = mcB.11

Table 9: Mapping of message composition registers.

SEND operation	*MC* Bank		Priority		Addressing		Throttling		Privileged	
	A	B	0	1	Virtual	Physical	Yes	No	Yes	No
SEND0	$\surd$		$\surd$		$\surd$		$\surd$			$\surd$
FSEND0		$\surd$	$\surd$		$\surd$		$\surd$			$\surd$
SEND0P	$\surd$		$\surd$			$\surd$		$\surd$	$\surd$	
FSEND0P		$\surd$	$\surd$			$\surd$		$\surd$	$\surd$	
SEND1P	$\surd$			$\surd$		$\surd$		$\surd$	$\surd$	
FSEND1P		$\surd$		$\surd$		$\surd$		$\surd$	$\surd$	

Table 10: Family of SEND operations

- `SEND0 length dest dispatchIP ccreg`:

  Sends a priority 0 message of length *length* from the integer register file starting at message composition register *mcA.0*. The *dest* register holds a virtual address that is to be translated into a node identifier by the GTLB. The *dispatchIP* register holds a pointer to code that the remote handler will execute. This pointer must be in the *Enter Message* state, or an exception will be raised. The *ccreg* is set to **empty** at the time when SEND0 is issued, and set to **true** and **full** when the entire message has been extracted out of the composition registers.

- `SEND0P length dest dispatchIP ccreg`: Similar to SEND0 except that no GTLB translation is performed.

- `SEND1P length dest dispatchIP ccreg`: Similar to SEND0P except that message priority 1 is used. In addition, these messages SEND1P is not subject to the throttling mechanisms (discussed below).

The set of operations `FSEND0`, `FSEND0P`, and `FSEND1P` perform the same tasks but use the floating point register file.

The family of SEND operations differ in the following ways:

- Network Resources. The message may be delivered using network resources from one of two decoupled priorities, 0 or 1.
- Addressing Mode. The `dest` address may be a virtual address which is translated automatically by the GTLB mechanism, or a physical node ID which can be explicitly derived from a virtual address using the **GPRB** (GTLB probe) operation, which is discussed in more detail in the Global translation section.
- Throttling. The SEND0/FSEND0 operations are subject to a message throttling mechanism, while some of the privileged SEND operations are not. The throttling mechanism involves an *Outgoing Message Buffer Counter* (OMBC) register in the network output unit which is *decremented* automatically with *each* message injected by this node into the network using a throttling SEND operation. The OMBC will *not* be decremented beyond zero. If the OMBC reaches zero, all throttling SEND requests are stalled until OMBC becomes non-zero again. The OMBC register is readable/writable via the configuration space, allowing the system programmer to implement network throttling schemes. When an acknowledgement message is received, the OMBC is incremented in software by the acknowledgement message handler. The non-throttling SEND operations do not affect, nor are affected by, the OMBC register.

The entire family of SEND operations is listed in Table 10.

A SEND operation is stalled until all of the MC registers specified by the instruction are marked *present*. Once the SEND operation is issued, the message words are extracted from the MC registers without regard to the presence bits. The process of injecting a message does not alter the presence bits. This allows a thread to compose two messages simultaneously in the two banks of message composition registers, or to send the same message to different destinations without reformatting the message. When the message is injected into the network, a header containing the message length (number of composition registers involved), the sender node's identification, and routing information is prepended to the message by the network output unit. The destination address, the dispatchIP, and the arguments from the message composition registers form the rest of the message.

Attempting to send a message where *dispatchIP* does not have the *Execute Message* protection bit set will cause an *message permission* exception in the cluster.

For SEND operations that expect a virtual address in `dest`, if the virtual address is not found in the GTLB, a *GTLB Miss* event is raised. The SEND operation is aborted, and all network interface resources previously allocated to this message are released. The *ccreg* however, remains marked *EMPTY*. The GTLB Miss event handler will find the correct translation, retrieve the message body via configuration space, set *ccreg* to TRUE, and retransmit the message.

Execution of a SEND operation requires multiple cycles to complete, but is an atomic instruction. A message send operation will stall the thread if a previously sent message is still occupying some required network output buffering.

If the user overwrites the message composition registers before the message has been completely extracted by the network interface, then the contents of the message might be different from the original message register contents. To synchronize with the network interface, the user can block on the *ccreg* until the network interfacer has notified the user that the message has been extracted.

11.2 Network Input Unit

The NETIN unit transfers words from the network into one of two *message queues*, *MQ0* and *MQ1*, according to the message priority. Each message queue is associated with 2 special registers, R_{head} and R_{body}, which are mapped to integer registers $i14$ and $i15$ of the Message H-Threads, which belong to the event V-Thread and reside in clusters 2 and 3, respectively.

The R_{head} register returns the *dispatchIP* specified in the next unread message in the corresponding message queue. Any words left over from the *previous* incoming message are flushed at the same time. In addition, condition code register 0, is mapped to the state of the queue. If the queue is empty, then 0 is found in *cc0*. Otherwise, 1 is read.

Each read to the R_{body} pops and returns one word from the message at the front of the corresponding message queue. The following ordering should be expected for each message accessed via R_{body}. The *ArgCount* and *SenderID* are extracted from the message header by the network input unit. *SenderID* is an integer node number identifying the sender node.

- Word 0: ArgCount (Number of arguments received in this message, n).
- Word 1: SenderID
- Word 2: Destination Virtual Address
- Rest: n words of Arguments (from the MC registers).

Note that an attempt to read both R_{head} and R_{body} simultaneously has undefined behavior. Also, since the network input is the single gateway from the network, the system programmer must guarantee that all message handling must occur quickly.

12 Performance Monitoring

To help measure the performance of the M-Machine and its software, the architecture provides two 64-bit counter/registers: a cycle counter and an event counter. Both registers are readable by user programs and can be both read and written in system mode. A third register, which is accessible only in system mode, selects the type of events to be counted by the event counter. All three registers are memory mapped into configuration register space.

Cycle Counter: The cycle counter register is incremented each processor clock. A read from this register samples its value sometime between when the instruction issues and when it completes. Arbitration for internal buses may cause the delay from issue to sampling to vary by a few cycles from instruction to instruction. Similarly, a write to this register sets its value a short time after the operation issues.

Event Counter: The event counter increments each time an event of the type selected by the event type register occurs.

Event Type Register: This register selects the type of events counted by the event counter. A list of event types is as follows:

- Operations. Two mask fields can be set to select (1) which thread slot(s) and (2) which operation units' operations are counted.
- Instructions. A mask field selects which thread slot(s) instructions are counted.
- Cache events. Mask fields select (1) which bank's events are counted and (2) the type of event counted: access, read, write, hit, miss.
- Network events. Mask fields select (1) the type of event (message send, flit send, message receive, flit receive, buffer allocates, event queue entries, overflows), (2) the network priority level(s) being monitored, and (3) the thread slot(s) being monitored.
- TLB events. Mask field selects LTLB or GTLB hits.

Other items, such as TLB misses and types of messages received may be counted in software.

13 Configuration Space

The configuration space of the M-Machine is the mechanism by which the internal state of the MAP chip can be accessed. Instead of having special instructions to access internal state, tagged pointers are used to allow internal state to be read and modified through memory operations. Pointers which have the permission field set to the *Configuration Space* state reference the hardware state of the MAP, and not the normal memory of the machine. Because the configuration space can only be accessed through tagged pointers, memory-mapping the internal state of the MAP does not interfere with the normal address space of the MAP. Programs which do not need to access the configuration space may ignore it completely.

Configuration space is divided into 5 sections: thread slot state, global state, memory system state, network state, and I/O. Configuration space includes the following:

- Thread slot state
  - General Registers
  - CC registers
  - Priority bits
  - Stall cycle counters
  - Stall counter limit registers
  - Preempt counter limit registers
  - Memory barrier counters
  - Thread instruction pointers
  - Context Pointer
  - V-Thread run bits
  - H-Thread run bits
  - Slot acquire bits
- Global state
  - Reset
  - Cycle counter
  - Event counter
  - Event type register
  - User slot enable
  - Catastrophic exception handler IP
- Memory system state
  - LTLB
  - GTLB
  - Next GTLB location to replace
  - LTLB lock
  - LTLB miss handled
  - EMI configuration
  - Little/Big endian
- Network state
  - Outgoing message buffer counter
  - Increment outgoing counter
  - Decrement outgoing counter

- XDIR, YDIR, ZDIR set
- Event queue
- Node ID

- I/O
  - I/O line buffer
  - I/O port configuration
  - Sync counter
  - Burst counter

14 External Interfaces

In addition to the communication network, the MAP chip has 3 other external interfaces: the diagnostic interface, the I/O interface, and the external memory interface.

14.1 Diagnostics

The MAP provides a serial diagnostic interface, which is used to shift initial programs and data in, as well as to run and halt the chip. The diagnostic interface consists of 2 control lines, a data in (DIAG-IN) line, and a data out (DIAG-OUT) line, and a 128 bit diagnostic register. The diag register has the following organization:

127 121	120 67	66 0
Command	Address	Data

Figure 13: The MAP diag register

Bits shift in to the most significant bit, at the left, and shift out from the least significant bit, at the right. The 67 bit data field allows space for 2 bits of synchronization state, a 1 bit pointer tag, and 64 bits of data. The 54 bit address field allows access into local node memory. The command field encodes the following commands:

- `NOP`: No operation.
- `READ-MEM`: Read word at specified physical address.

- `WRITE-MEM`: Write word at specified physical address.
- `READ-CONFIG`: Read configuration register at specified configuration space address.
- `WRITE-CONFIG`: Write configuration register at specified configuration space address.
- `SHIFT`: Shift a bit in from DIAG-IN into the internal register scan chain, and shift a bit out onto DIAG-OUT.
- `INTERRUPT0`: Trigger a priority 0 interrupt event.
- `INTERRUPT1`: Trigger a priority 1 interrupt event.
- `HALT`: Halt MAP.
- `GO`: Run MAP.
- `STEP`: Step MAP one clock cycle.

The `READ-MEM` and `WRITE-MEM` commands read and write words in the physical memory of the MAP. The `READ-CONFIG` and `WRITE-CONFIG` commands allow easy access to the configuration registers of the MAP. The `SHIFT` command is used to access the internal state of MAP via a scan. Non-visible registers of the MAP are in a linear scan chain that may be read and written using the shift facility. This feature is intended to simplify testing. Note that the MAP must be halted before executing the `SHIFT` command.

The two control lines encode the following 4 states:

- `NOP`: Do nothing.
- `EXEC`: Execute the current command in the diag register.
- `SHIFT`: Shift one bit into and out of the diag register.
- `RESET`: Reset all of the MAP's internal state.

14.2 I/O Interface

A special I/O interface is included to ease connection to (relatively) low speed peripherals. High speed peripherals, such as a graphics subsystem will need to connect to the MAP external memory interface.

The I/O interface consists of an 8 bit bidirectional bus (IOBUS), 4 bits of control out of the MAP (CIO-OUT), and two bits of control into the MAP (CIO-IN). Both address and data are multiplexed onto IOBUS. The CIO-OUT lines encode the state of the bus cycle for the peripheral state machines. The CIO-IN lines are used by a peripheral device to signal the MAP. These include *hold* and *interrupt* states.

14.3 External Memory

The external memory interface consists of a 54 bit physical address bus (ADDR), a 67 bit data bus (DATA), and a wait signal (WAIT). Most MAP chips use this interface to connect directly to their off-chip DRAMs. However, high speed I/O devices, such a graphics systems, will connect to the external memory interface. The WAIT pin is used by the peripheral device to steal a bus cycle from the processor in order to transfer data into the memory.

15 Startup Boot Sequence

An M-Machine node must be reset upon power up (or after some other catastrophic event) in order to prepare it for execution. In addition, internal registers must be set (by software) during the boot sequence. An M-Machine node may be booted via the diagnostic interface using the following three steps:

1. Reset the MAP chip via the diagnostic interface (DIAG).
2. Download the boot kernel code through DIAG.
3. Set the chip to RUN through DIAG.

When the MAP comes out of reset, all of the slots are halted. By convention, event slot 0 will serve as the initial boot thread and prepare the rest of the chip for execution. The initial CP and IP are downloaded via the diagnostic port. The CP must point to an area with enough context space for all of a thread's memory mapped registers. The CP and IP pointers must be in the *physical* state so that the virtual memory system is bypassed. The hardware instruction-cache refill engine fetches the appropriate code directly from physical memory.

The boot kernel software that is downloaded must prepare the MAP for service in the M-Machine and must perform the following tasks:

- Reset network registers
- Start event thread 1
- Start message thread 0 and 1
- Initialize LTLB and GTLB
- Flush the cache

When the boot kernel is finished running, the thread in event slot 0 should branch to the event thread 0 handler code, which resides in virtual memory space. At this time, the node is ready to receive messages from the host to start other required user level services and to begin execution of user level code.