

Prototyping a Scalable Massively-Parallel Supercomputer Architecture with FPGAs

by

Michael P. Phillips

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2001

© Michael P. Phillips, MMI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author

Department of Electrical Engineering and Computer Science

May 23, 2001

Certified by

Thomas F. Knight, Jr.

Senior Research Scientist, MIT Artificial Intelligence Laboratory

Thesis Supervisor

Accepted by

Arthur C. Smith

Chairman, Department Committee on Graduate Theses

Prototyping a Scalable Massively-Parallel Supercomputer Architecture with FPGAs

by

Michael P. Phillips

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2001, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

There are many computationally-intensive applications that hunger for high-performance computer systems to manage their enormous memory and processing requirements. There is a need for a scalable, massively-parallel processing supercomputer that can satisfy this hunger. Developing such an architecture requires evaluation of hard data when making design decisions. Software-based simulations can be helpful, but a physical implementation of the hardware runs faster and can reveal fatal flaws that were overlooked. It also grounds the design in reality. Since fabrication of a chip implementing a brand new architecture is not immediately feasible, the use of configurable logic devices is favored. This thesis presents an implementation of the ARIES Hamal Parallel Computer architecture, targeted at FPGA configurable logic devices. This experience revealed that although FPGAs are extremely flexible devices, synthesis of a complete processor is a time-consuming task and that the logic capacity of typical FPGAs are insufficient for processors with very large (128-bit) data paths.

Thesis Supervisor: Thomas F. Knight, Jr.

Title: Senior Research Scientist, MIT Artificial Intelligence Laboratory

Acknowledgments

The author would like to thank the entire ARIES group for their support and ideas. In particular: Tom Knight for his extraordinary architectural wisdom; J.P. Grossman for the countless discussions of Hamal details; Andrew “bunnie” Huang for help with the FPGA technology and tools; and Jeremy Brown for keeping the author tuned-in to important programming-language factors. Finally, the author would like to thank his lovely wife, Carmen, for her moral support and editing efforts.

For further information

The author may be contacted at mpp@alum.mit.edu. Additional information about Project ARIES may be found at <http://www.ai.mit.edu/projects/aries/>.

Contents

1	Introduction	15
1.1	Motivation and Goals	15
1.2	Scope of this Work	17
1.3	Organization	17
2	Project ARIES	19
2.1	Motivations	19
2.2	Design Overview	21
3	The Hamal Architecture	25
3.1	Memory	26
3.1.1	Atomic Operations	26
3.1.2	Memory Model and Capabilities	26
3.1.3	Paging	29
3.2	Network Interface	30
3.3	Instruction Memory	30
3.4	Node Controller	30
3.5	Processor	31
3.5.1	Instruction Group Format	31
3.5.2	Multiple Hardware Contexts	31
3.6	Instruction Issue	35
3.7	Execution Pipelines	36
3.7.1	Arithmetic Pipeline	36

3.7.2	Memory Pipeline	36
3.8	Instruction Set	38
3.9	Arithmetic Instructions	40
3.10	Memory Instructions	41
3.10.1	Address Computation Instructions	41
3.10.2	U and V Bits	43
3.10.3	Load/Store Instructions	43
3.10.4	Non-trapping Instructions	44
3.10.5	Atomic Memory Instructions	44
3.10.6	The Fork and Join Instructions	45
3.11	Control Instructions	45
3.12	Privileged Instructions	46
3.12.1	Context Swapping	46
3.12.2	Managing the Fork Buffer	47
4	Processor Implementation	49
4.1	FPGA Details and Synthesis Tools	49
4.1.1	FPGA Features	50
4.1.2	Tools Features	51
4.2	Implementation Details	53
4.2.1	Node Controller Interface	53
4.2.2	Instruction Memory Interface	55
4.2.3	Split-Pipeline	58
4.2.4	Trace Controller	59
4.2.5	Issue Logic	64
4.2.6	General-Purpose Register File	65
4.2.7	Execution Unit	67
4.2.8	Memory Pipeline	68
4.2.9	Arithmetic Pipeline	69
4.2.10	Memory Output Path	70

4.2.11	Thread Management and the Context Control Unit	74
4.2.12	Global Events	78
4.2.13	Moving signals on and off the chip	79
4.3	Re-targeting Guidelines	80
4.3.1	Multiplexors and Tri-State Bus Drivers	80
4.3.2	Clock Generation and Distribution	81
4.3.3	Input/Output Drivers	81
4.3.4	General-Purpose Register File	81
5	Beyond the Processor	83
5.1	Current Status	83
5.2	The Next Step	84
6	Verification	87
6.1	Testing environment	87
6.2	Module testing	88
6.2.1	Testing the Register File	89
6.3	Higher-level Testing	90
6.3.1	An Assembler	90
6.3.2	Simulating Instruction Memory	90
6.3.3	Simulating the Node Controller	91
6.4	Limits	91
7	Conclusion	93
7.1	Results	93
7.2	Lessons	95
7.3	Future Work	96

List of Figures

2-1	ARIES chip-level design	21
2-2	Block diagram of a single processor-memory node	22
2-3	Block diagram for a processing node	22
2-4	Block diagram for a network node	23
2-5	Basic Box and Rack Architecture	24
2-6	Processor-Intensive Configuration	24
2-7	Communications-Intensive Configuration	24
3-1	The Hamal Processing Node	25
3-2	Capability Format	27
3-3	GPR Granularity	32
3-4	Parallel Arithmetic Operation	41
4-1	Detailed View of a Virtex-E Slice, from [37]	50
4-2	Tool Flow	52
4-3	Processor-Node Controller Interface	53
4-4	Memory Reply Data Path	54
4-5	Processor-Instruction Memory Interface	56
4-6	Hamal Pipeline Stages	59
4-7	Trace Controller Instruction Flow	59
4-8	Trace Controller Data Paths	65
4-9	General Purpose Register File Read Ports	67
4-10	Memory Output Data Path	71
4-11	Memory Request Format	72

4-12 Memory Request Table Entry	72
4-13 Fork Control Finite State Machine	75
4-14 Context Store FSM and Context Load Look-up ROM	77

List of Tables

3.1	Event Registers	33
3.2	Events	34
3.3	Predicate updates	39
3.4	Arithmetic Predicate Instructions	40
3.5	Arithmetic Instructions	42
3.6	Address Computation Instructions	43
3.7	Atomic Addition and Subtraction Operations	45
3.8	Control Instructions	46
3.9	Privileged Instructions	47
4.1	Decoded Instruction Format	57
4.2	Memory Instruction Flags	58
4.3	Next Address Determination	60
4.4	Thread Events	69
4.5	Context State	76
4.6	Resource use for a 182-bit wide FIFO	79
4.7	Time-multiplexing the FPGA I/O pins	80
7.1	FPGA Resource Use	94
7.2	Slice Usage Detail	94

Chapter 1

Introduction

Although advances in microchip fabrication technology are constantly making faster and denser microprocessors available, there are still many applications that require far more computational power than is achievable with current computer architectures. Problems such as protein folding, physical simulations, graphics rendering, computer-assisted design, and cryptographic application demonstrate this need. To solve this problem, it is best to start with a “clean-slate” approach, looking at what we need and what we have available from a fresh perspective, without any ties to legacy architectures or implementations.

1.1 Motivation and Goals

A major motivator is the fact that there are problems which could be better solved if there existed supercomputers that could compute results faster than is currently possible. In particular, these are problems for which algorithms exist for operating on subsets of them in parallel, such that having multiple processing elements would be helpful, if they could communicate efficiently. The goal is not to create a single processor which has the best performance characteristics possible on sequential programs—the computer industry has been doing a good job of this recently. Multi-threaded performance is the goal.

Performance itself is not the only concern. When taking the time to design the

next best supercomputer architecture, there are several other key points which must not be overlooked. The first is scalability. It should be the case that expanding the size of the system also expands its processing power. If there is to be a point beyond which adding additional processing hardware is no longer feasible or helpful, it should be well beyond just a handful (or a few hundred) processing elements. If doubling the performance of the machine takes fifty times the hardware, then this goal is not being met. A large part of the issue of scalability is dependent on the design of the communication network that allows processors to communicate, as well as the semantics in the architecture for utilizing this resource.

Programmability is also an important goal of such a system. The memory model, architectural features, instruction set, programming language support, and operating system should be designed in such a fashion that the machine is not incredibly difficult to program. Specifically, it is not unreasonable, as a programmer, to expect to be able to craft software for the machine in a relatively straight-forward manner and have it perform well— this precludes, for example, a situation where only expert assembly-language hackers who are intimately familiar with the machine’s architectural details are able to produce code that causes the machine to perform at its maximum efficiency. This goal influences the design of the instruction set architecture, along with compilers, libraries, the operating system, and possibly new programming language design.

The final goal is engineering a high-confidence system. One such feature would be fault-tolerance. Given a supercomputer of any large size, faults will be extremely common [30]. There is a need to plan for this when designing the architecture, so that the detection and recovery from faults is possible. Another aspect of a high-confidence system is perhaps less obvious: hardware protections from misbehaving programs or users. Architectural features should be implemented to help programmers avoid common mistakes that can lead to incorrect results or even a breach in system security. For example, many contemporary architectures are renowned for the common problem of a faulty program writing beyond the end of an array, either accidentally or intentionally [28].

1.2 Scope of this Work

Designing a supercomputer architecture from scratch is an onerous task. There are several different pieces of the system which need to be designed: the processor, the memory system, the individual nodes, and the network. This work is concerned only with issues relating to the implementation of the processor component. Specifically, this includes an approach to prototyping a potential processor design in order to determine feasibility and potential performance. A hardware implementation has many advantages over a software-only solution. Grounding the design in reality by actually building it often finds overlooked design flaws. More importantly, the hardware implementation can run faster than the software solution and is more scalable. While a software-only simulation of potential architectures is feasible [31, 23], prior work, such as Stanford’s DASH project [26] found that software simulations of complex hardware are orders of magnitude slower than the real system would be. To this end, this thesis presents a hardware description, in Verilog, of the processor that is suitable for synthesis into an FPGA. This thesis also discusses lessons learned in using an FPGA as a processor prototyping platform, along with verification tools and strategies.

1.3 Organization

Chapter 2 expands upon this introduction’s description of a desirable supercomputer architecture by introducing the ARIES high-performance computing (HPC) research project, of which this work is a part. The third chapter focuses on a particular processor architecture, named Hamal. Chapter 4 describes all of the details of this work’s implementation of that architecture. Following that, Chapter 5 details of interfacing the processor with other elements of an HPC system. The matter of verifying the implementation is tackled in Chapter 6. Finally, Chapter 7 concludes with a summary of results, lessons learned, and recommendations for future work.

Chapter 2

Project ARIES

This research is part of the Advanced RAM Integration for Efficiency and Scalability (ARIES) project. The project's goals are to meet the previously mentioned HPC design challenges. Specifically, the project's objective is to feature massive scalability, silicon efficiency, and RAM integration in the design of an HPC system. In order to help provide context for this thesis, this chapter will give some background on the ARIES project.

2.1 Motivations

At the system level, the ARIES project wishes to produce a scalable and efficient HPC system which is well-suited to applications that have enormous memory and/or computational requirements. Currently, parallel systems suffer because they are difficult to program, have poor network performance, or cannot be scaled to a very large number of processors. A goal of the ARIES project is to design a system which has support for a million or more simple processing elements and that still is programmable.

There have been enormous strides in technology during the last twenty years. We have seen transistor counts increase by a factor of a thousand in this time period. Because of the steps taken during the evolution of the technology, one particular path has been followed in the design of microprocessors, progressively adding on new

layers of functionality as technology improves. This results in modern microprocessors using a small amount of the available die area for computation and the rest for control and speed-optimizations to make that small area better. Thus, another goal of Project ARIES is to achieve better silicon-efficiency. This can be accomplished by eliminating some of the complex area-consuming logic that results in only small performance gains, popular in processors targeted as maximal single-threaded performance, and using the area saved to perform more computation through the use of multiple hardware execution contexts and multiple processing elements on a single die.

Additionally, advances in the area of tight processor and memory integration [15, 29, 25] combined with improvements in memory technology [14] suggest and allow for new design paradigms for efficient processor architectures. Specifically, the ability to closely couple memory and processing elements results in being able to perform computation with lower memory latency and much greater memory bandwidth. RAM integration is an important goal of Project ARIES.

Scalability is one of the biggest motivations. The project seeks to develop an architecture where having more than one million nodes is both feasible and worthwhile. That is, with each processor added there is performance gained, such that 1 million processors would have far more processing power than half as many. Any point of diminishing returns must be pushed well above the million-processor mark. The network interconnect between all of the processors is the largest constraint. The plan is to use a variation of the MIT Transit Group's METRO network [10], keeping in mind existing work on networks for HPC systems [7, 9]. An important property of the network is that it is fault-tolerant; there are multiple ways to route from one processor to another such that some number of failures in the network infrastructure can be tolerated.

2.2 Design Overview

At the chip level, the vision is of a single silicon die containing an array of processor-memory nodes connected by an on-chip network, as shown in Figure 2-1. Notice that there are no caches in the system. Globally-consistent caches inherently work against scalability [2] and have high costs in die area and complexity. Figure 2-2 is a look at the design of a single processor-memory node with this array.

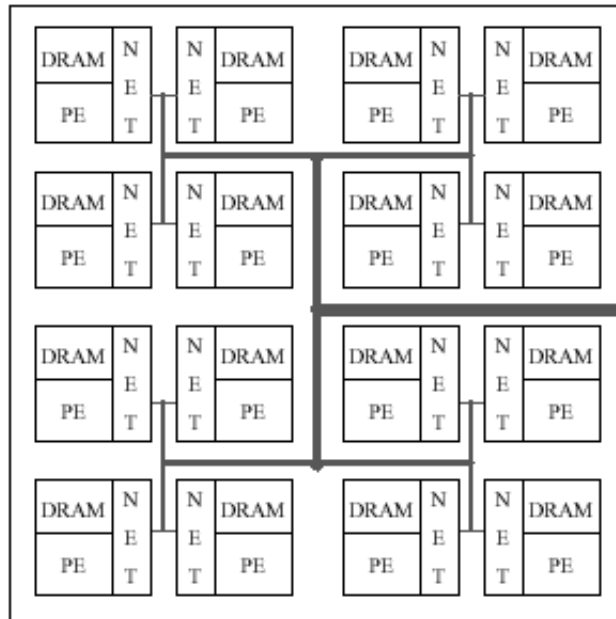


Figure 2-1: ARIES chip-level design

The next level of the plan is to assemble several of the ARIES nodes along with appropriate routing into a “processing node.” The node will also need additional hardware— some local disk to use for permanent storage, power supply hardware, cooling hardware, and a processor for monitoring the status of the node. Figure 2-3 illustrates this with a block diagram of a processing node.

Some applications may require a large amount of network interconnect and redundancy in order to enable a large amount of communication between processor nodes and to benefit from a higher level of fault tolerance. As a result, the concept of a “network node” was conceived; it is a collection of network interface (NI) chips which essentially make a much larger number of wires available for the processor nodes to

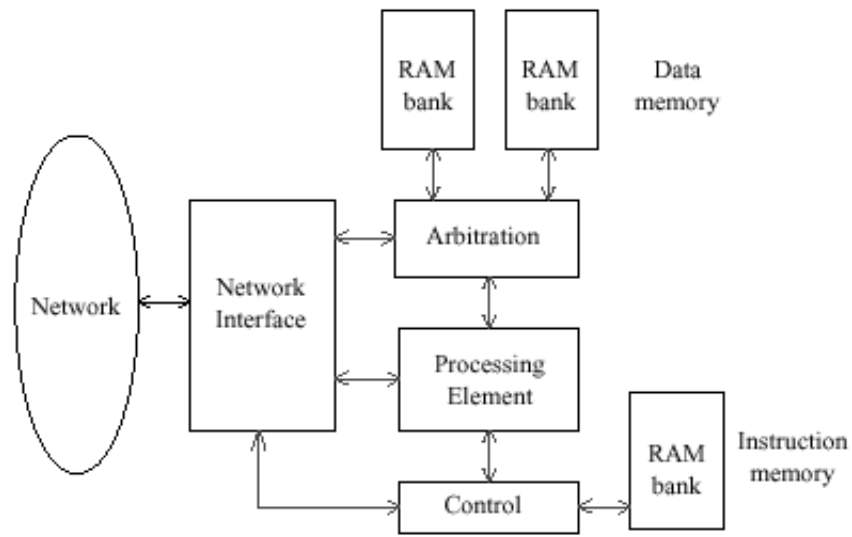


Figure 2-2: Block diagram of a single processor-memory node

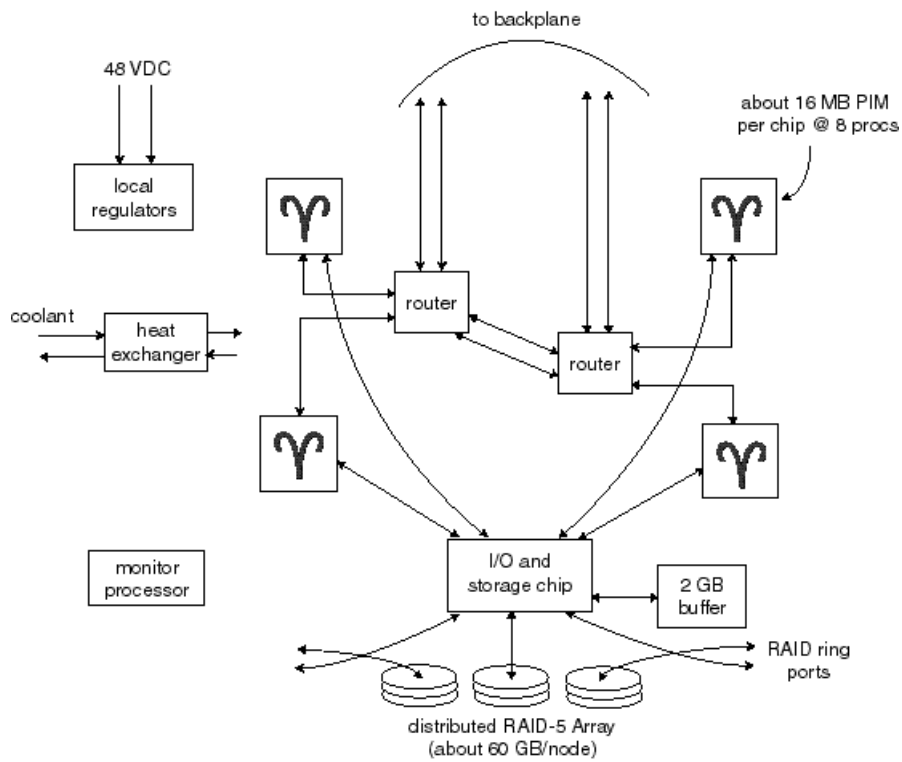


Figure 2-3: Block diagram for a processing node

use in communications. This is shown in Figure 2-4.

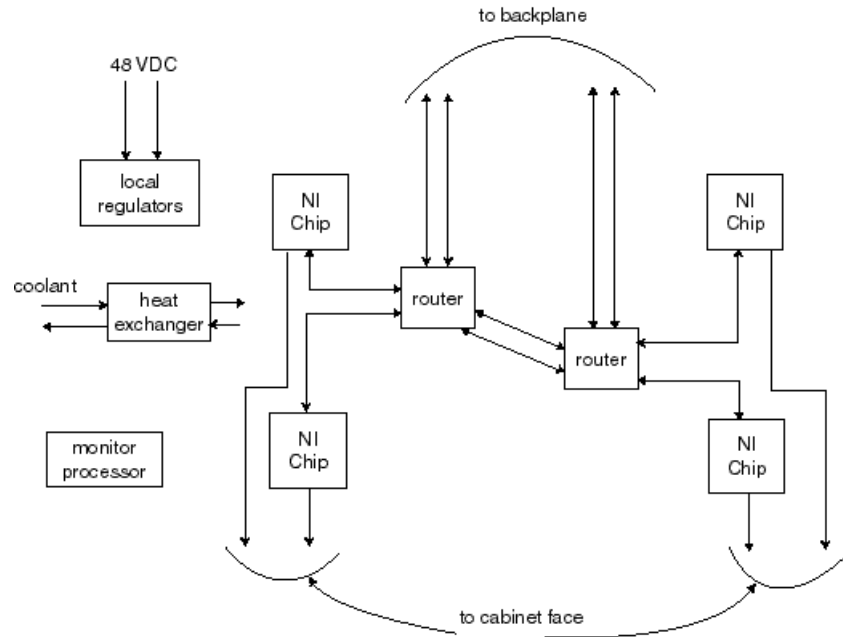


Figure 2-4: Block diagram for a network node

Since the processing and communication requirements of different applications vary greatly, there needs to be some level of configurability. One user may want to maximize computational power for an application, while another might have a communications-intensive application which performs better when lots more wires are available. A solution is to construct “boxes” containing either a processing node or a network node. These boxes can be placed into a rack which has a backplane for inter-box communication (see Figure 2-5). It is possible to add additional racks if more nodes are needed. In this manner, the end user can decide the best way to configure his machine. This flexibility is demonstrated in Figures 2-6 and 2-7 which show different configurations for four racks.

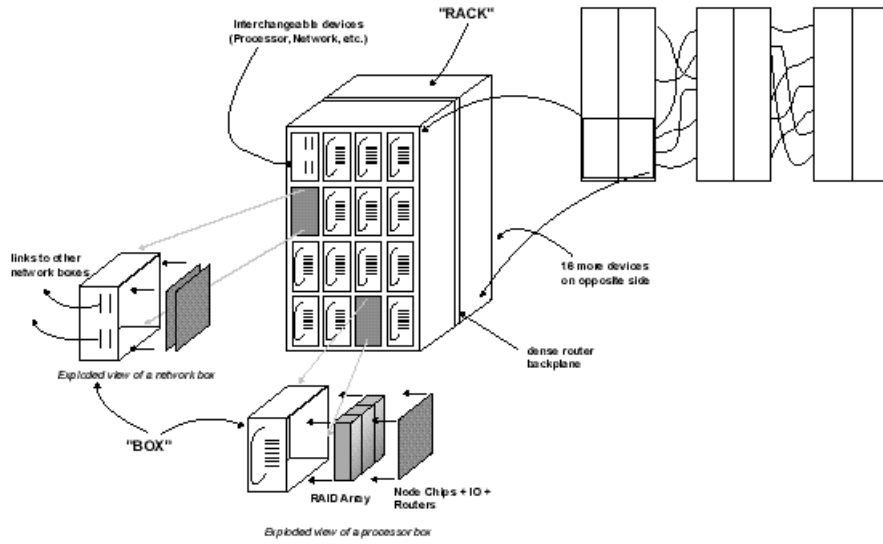


Figure 2-5: Basic Box and Rack Architecture

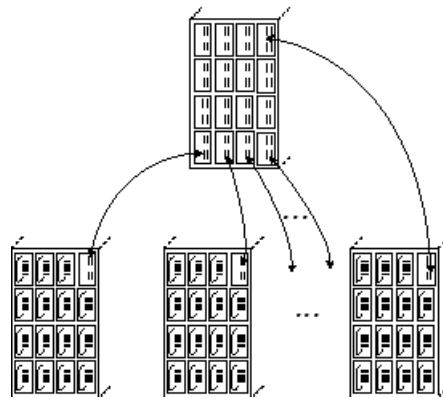


Figure 2-6: Processor-Intensive Configuration

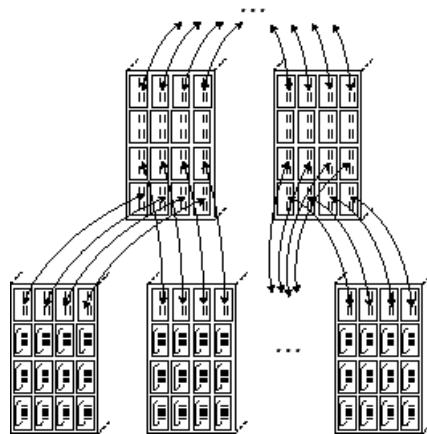


Figure 2-7: Communications-Intensive Configuration

Chapter 3

The Hamal Architecture

This thesis implements a subset of J.P. Grossman’s on-going work with his Hamal Parallel Computer architecture [19, 17, 18, 16]. Part of Project ARIES, Hamal is guided by research into design principles for the creation efficient, scalable, massively parallel high-performance computer systems. This chapter summarizes this architecture in order to provide context before the author’s actual implementation is presented. Emphasis is placed on the processor component of Hamal, as that is the component implemented in this work.

The Hamal architecture specifies a computer constructed out of many processor-memory nodes, connected by a fat tree network. Each node consists of data memory, instruction memory, a network interface, the processor, and controller logic to arbitrate data flow among these parts. This is shown in Figure 3-1, from [16].

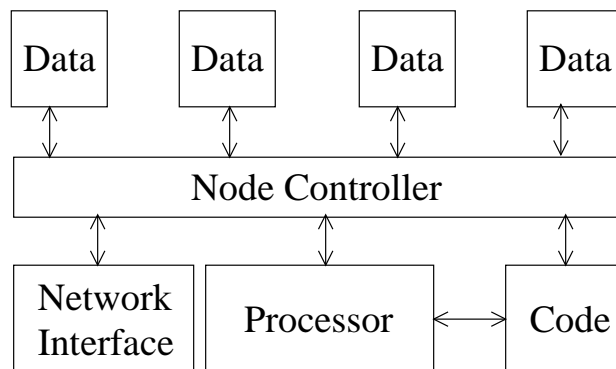


Figure 3-1: The Hamal Processing Node

3.1 Memory

The word size of the machine is 128-bits. Each word in memory is tagged with four additional bits: **P**, **T**, **U**, and **V**. The **P** (pointer) bit is used to distinguish pointers from non-pointer data. The **T** (trap) bit, when set, will cause the processor to generate an event any time this word of memory is accessed. The operating system's trap handler would then be notified, allowing implementation of features such as debugger breakpoints and forwarding pointers. The **U** and **V** (user) bits can cause traps to a handler specified by a user program. Each memory instruction specifies how the **U** and **V** bits should be interpreted and updated on each access.

3.1.1 Atomic Operations

The memory subsystem itself is capable of performing simple, single-cycle, atomic operations on the data it contains and a value from the processor. These operations are addition, subtraction, minimum, maximum, and any two-input boolean operation.

3.1.2 Memory Model and Capabilities

The memory model is that of a single shared-memory image over the entire machine. There is no global caching of data, as that would severely restrict scalability, and is difficult to implement correctly without considerable effort. There is one virtual address space. This address space can be shared safely between all processes, without requiring them to trust each other, through the use of unforgeable capabilities ([8] [11]). Capabilities are a construct that give specific access rights to a region of memory, known as a segment. Past capability architectures have used special tables [24], or registers [1], or software support [3] to verify that memory was being accessed legitimately. However, these approaches incur an expensive performance penalty while the system performs the lookup. Instead, Hamal extends the capability format proposed in [6] to pack all of the relevant permission and segment information along with a 64-bit address into a 128-bit data word[5]. When the word's **P** bit is set, the 128-bit value is interpreted as a pointer with capability information encoded into it.

Figure 3-2 illustrates Hamal’s capability format.

P	Type	Perm	User	SQUID	K	L	B	Node	Offset
1	2	10	24	12	5	5	6	20	44

Figure 3-2: Capability Format

Capability types

There are four kinds of capabilities: **code** capabilities which give access to program memory, **data** capabilities which give access to data memory, **input/output** capabilities which allow the use of I/O devices, and **join** capabilities which allow the use of the special **join** inter-thread synchronization instruction. **Join** will be covered later.

Permission bits

For data and I/O capabilities, the permission bits control the use of the pointer for reading (**R**) from and writing (**W**) to that address. Additionally, the Take (**T**), Grant (**G**), Distributed Take (**DT**), and Distributed Grant (**DG**) bits restrict how this pointer may be used to read and write other pointer data, as in [32]. Code capabilities do not use the W, T, G, DT, or DG permissions. They add two more capabilities, Execute (**X**) and Execute Privileged (**P**). All capabilities also have Decrement-only (**D**) and Increment-only (**I**) bits. These restrict the addition or subtraction of offsets to or from the pointer, respectively.

Segment size

Three values in the capability place restrictions on the size of the segment for which the capability applies: B , L , and K . The segment is composed of blocks, each being 2^B bytes in size. There are $L + 1$ blocks in the segment, for total size of $(L + 1) * 2^B$ bytes. In the worst case, with the given sizes of L and B , a loss of 6% to internal fragmentation is possible as a slightly larger object than requested would need to be allocated. For any given pointer address, K indicates the block offset into the segment. It is possible to recover the base address of any object by first zeroing the

last B bits of the address (to achieve block alignment) and then by subtracting $K * 2^B$ bytes. The equation for this operation is: $base = ((offset \gg B) - K) \ll B$, where $base$ is the base address and $offset$ is the address stored in the pointer.

As a matter of vocabulary, this encoding of capability information and the pointer together means that this work uses the words “capability” and “pointer” interchangeably. Also, since the segment size can be relatively small with little slack, it is possible to allocate individual data objects as segments unto themselves; thus, “segment” and “object” are also synonyms.

Migrated Objects and SQUIDs

The use of forwarding pointers, which indicate that the object to which a pointer points has been relocated, has been shown to be useful in garbage collection and data compaction [4] [27]. However, this introduces the possibility that two pointers may have a different base address but actually refer to the same object, making pointer comparisons an expensive operation. Hamal’s solution to this is the use of Short Quasi-Unique IDs (SQUIDs) [22, 21]. A SQUID is assigned to an object when it is allocated. This is a randomly generated identifier, which is not guaranteed to be unique. A uniqueness guarantee would require global communication, which works against scalability. Any two pointers to the same object will always have the same squid; if the SQUIDs do not match, then there are definitely two different objects involved. Only when the squids do match is a more complicated analysis, pointer dereference, required to determine if the objects are the same. To assist with this process, the Migrated (**M**) capability bit will be set in any pointers that point to the new home of a moved object. This bit being set means that objects with the same squid and differing base addresses might be the same object; without the M bit being set in either capability, they cannot be.

Sparse Objects

The Hamal architecture has support for lazily-allocated distributed objects. Known as sparse objects, these objects have “facets” on multiple nodes of the machine. That

is, the object appears to be the same segment base and length on all nodes. However, when allocated, space is only reserved on the node that requested the allocation. When a pointer to a facet on some other node is used, that node will see that it is now time to allocate the facet, (thus, lazily). This node is free to put the object wherever it would like in the memory it has, and is not restricted to using the same base that the original node did. Indeed, such a restriction would require all nodes to co-operate and find free space in their memories that were in common. Instead, the network interface keeps a translation cache that maps between the object's common, (or 'global'), base address and the actual base address used locally. The processor recognizes these special sparse objects, which will have the Sparse (**S**) bit set in any capabilities that refer to them. The network interface will notify the processor if the translation cache misses, enabling the operating system to take appropriate action.

User Bits

Twenty-four bits are left over in the capability. These are reserved for use by the operating-system. One example is to use them to encode additional permission bits. Another is to produce a sub-segmentation scheme to further restrict a capability over a subset of its segment.

3.1.3 Paging

The data memory is divided into multiple banks, each equipped with an SRAM hardware page table. Each entry corresponds to a physical page in the memory. An entry contains the virtual base address of the page, status bits, and the P, T, U, and V bits for each word in the page. A page-out transfers these bits and then the words in the page to the node controller, which will pass the data on to the network interface, and from there to a secondary storage device. A page-in accepts this same data from the node-controller along with the physical page number to occupy. To assist the operating system in deciding which page to evict when space is needed, the data memory tracks exact Least Recently Used (LRU) information for the pages it

controls.

3.2 Network Interface

The network interface is responsible for sending and receiving data to and from remote nodes. It is equipped with request buffers to track pending requests, caches for speeding up local-to-global and global-to-local sparse object pointer translations, and a fork buffer for holding data for a fork operation. This buffer collects the thread state from a thread issuing a fork operation and delivers it to the node referenced in the request.

3.3 Instruction Memory

As one would expect, instruction memory holds code to be executed. Similar to the data memory, there is a hardware page table with one entry per physical page. However, the words in the instruction memory are not tagged with the pointer (P) and user (U, V) bits. Instruction memory is read-only. Page-outs are not supported. The only way to get code into the instruction memory is by performing a page-in in response to an instruction memory page fault. The instruction memory is otherwise read-only.

3.4 Node Controller

The node controller has direct control over all other components in a node. Its job is to manage the flow of data between all of the components. It can receive data and requests from data memory, instruction memory, the network interface, the fork buffer, the context-load controller, and the processor. The potential sinks for this data are data memory, instruction memory, the network interface, the fork buffer, the processor's data input path, and the processor's event input path. The controller's management of resources helps the processor devote more cycles to computation, and

fewer to tasks such as micro-managing page-ins and page-outs.

3.5 Processor

The processor is a Very-Large Instruction Word (VLIW) processor with 129-bit data paths. Additionally, the processor has support for four hardware thread contexts. The memory execution unit includes hardware for enforcing the permissions in capabilities, and the arithmetic unit includes support for fine-grained Single Instruction Multiple Data (SIMD) style parallelism.

3.5.1 Instruction Group Format

Each VLIW instruction group consists of three instructions and an immediate value. One control instruction, one arithmetic instruction, and one memory instruction may appear in an instruction group. The control instruction may have zero or one operands, the arithmetic instruction up to two, and the memory instruction up to three.

3.5.2 Multiple Hardware Contexts

The Hamal architecture dictates that the processor must store enough state to be able to issue from any hardware thread each cycle without suffering context-switching penalties. The architecture further suggests that four hardware threads be supported. That number may change as additional data is collected regarding the number of concurrent threads needed to make most efficient use of the remaining silicon [16]. The first context, Context 0, is reserved for event-handling, while the others are available for user programs. Each context's state includes the general-purpose register file, predicate registers, special-purpose registers, resource-use data for in-flight instructions, and a small instruction queue. Context 0 includes an event queue which stores global events which are awaiting processing. A piece of hardware known as a "trace controller" is responsible for operating a single hardware thread.

General-Purpose Register File

The Hamal register file consists of 32 129-bit registers. The 129th bit is the **P** bit, which indicates the presence of a pointer, just as the **P** bit in the memory subsystem does. This width accommodates the pointer format well. To support arithmetic operations, each register may be divided into 64-bit and 32-bit pieces. Each piece may be referenced independently. For example, the name “r0” would refer to all 129 bits of the first GPR. “r0x” and “r0y” refer to the low 64 and high 64 (not including the P bit) bits of r0. Finally, “r0a,” “r0b,” “r0c,” and “r0d” refer to non-overlapping 32-bit pieces of r0. This is illustrated in Figure 3-3.

P		r3			P	r2y		r2x	
P	r1y	r1b	r1a	P	r0d	r0c	r0b	r0a	

Figure 3-3: GPR Granularity

Predicate Registers

Hamal supports predicated execution; each instruction in an instruction group may depend on the true/false value of one of 31 predicate registers to determine if it will be executed or ignored. These one-bit registers can be set or cleared individually using the comparison instructions in the instruction set, or they can be handled en-masse like a 32-bit register, named **pr** with the highest bit always set. This 32-bit register can be further decomposed into a pair of 16-bit registers, **px** and **py**. Some of the predicates have special meaning. The highest modifiable predicate, **p30**, is used by the non-trapping memory instructions to indicate an error, as explained later. Predicates **p29** and **p28** are used to indicate signed and unsigned overflows when performing arithmetic instructions.

Pointer Registers

In addition to the predicate registers, there are four kinds of registers which cannot rightfully be called “general-purpose.” The first are the pointer registers: the branch

pointer (**bp**), the trap return pointer (**tr**), and the trap vector (**tv**). **bp** holds a code capability which indicates an address to transfer control to if a branch instruction is executed. **tv** holds the address of the thread's trap handler. In the case of a trap, **tr** will be filled-in with the address where execution should resume after the trap is serviced.

Other Special-Purpose Registers

The seven event registers, **e0** through **e6**, are filled in with information regarding an event which just occurred. These values can be read, but not modified, by an event handler to determine how to recover from the event. Table 3.1, adapted from [18], shows how these registers are used. Not all events make use of all registers. Finally, there are two global registers shared by all threads: **g0** contains a 20-bit node identifier, and **g1** contains the number of clock cycles that have passed since processor reset, in a 64-bit value.

Register	Contents
e0	Event code
e1	Operation that caused the event
e2	Address of the instruction that caused the event
e3	First operand to the instruction
e4	Second operand to the instruction
e5	Extra information, varies with event type
e6	Instruction's destination register

Table 3.1: Event Registers

Event Queue

The event queue can only be accessed by context 0. There is a special instruction named **poll** which will cause the thread to become inactive until an event arrives. These events are global events which originate in the memory subsystem, the network subsystem, or the processor itself and require special operating-system handling. Other events which happen in the processor are known as thread, or local, event.

Thread events are handled by the trap handler of the thread that caused them; they are not placed into the event queue. Table 3.2, adapted from [18], lists the different kinds of events.

Event	Description
<i>Capability thread events - detected in memory pipeline</i>	
EV_BOUND	Segment bound violation during address calculation
EV_ALIGN	Misaligned memory address
EV_BADP	Attempt to use non-pointer as an address
EV_PERM	Permission violation
EV_SQUID	Pointer dereference needed to check pointer equality
<i>Instruction thread event - detected by trace controller</i>	
EV_BADIP	Attempt to execute code at an invalid address
EV_BADOP	Invalid op code or register specification in instruction
<i>Global memory events - generated by memory subsystem</i>	
EV_PGF_CODE	Page fault: code page not present
EV_PGF_DATA	Page fault: data page not present
EV_PG_IN	Requested page-in operation has completed
EV_MEM_T	Trap due to T bit set on memory word
EV_MEM_UV	Trap due to U or V bits on memory word
<i>Global processor events</i>	
EV_BREAK	A thread has executed a break instruction
EV_SUSPEND	A thread has executed a suspend instruction
EV_REPLY	Memory reply received for a swapped-out thread
<i>Global network interface events</i>	
EV_XLATEIN	Global to local translation cache miss
EV_XLATEOUT	Local to global translation cache miss
EV_FORK	Fork buffer contains fork request for this processor

Table 3.2: Events

Instruction Queue

The instruction queue is a small First In First Out (FIFO) structure which stores instruction groups which have had their control portions executed, but are waiting to issue into the processor's main data paths. If a context's queue fills up, the context will not request additional instructions from the instruction memory. On a trap, the instruction queue is cleared and any side-effects caused by execution of the control portion of those cleared instruction groups are un-done. The contents of the instruction queue is also discarded in the event that the context is swapped-out to memory in favor of another thread; it is not considered part of the context state for this purpose.

Tracking the Resource Use of In-flight Instructions

The trace controller must track which registers will be written to by instructions that have issued but not yet completed execution. This is necessary to avoid read-after-write (RAW) and write-after-write (WAW) hazards on a register. A RAW hazard occurs when an instruction issues and reads a register that is going to be modified by an instruction that issued before it. The second instruction should operate on the value that the first writes and not the older value. Thus, the second instruction must not be allowed to issue until the first is finished. A WAW hazard can occur because instructions are not guaranteed to complete in the order they issued. This is due to the fact that instructions do not have a fixed execution latency.

A “busy bit” is stored for each addressable register in the machine. For the GPRs, this means four bits per register, one for each 32-bit piece. Each predicate has its own busy bit. A busy bit is set when an instruction that will write to the corresponding register has issued. The bit is cleared when the write-back has completed. These busy bits are examined by the trace controller when determining if its next instruction group is allowed to issue.

Since control instructions are executed when the instruction group arrives from instruction memory, the trace controller must make sure none of the instructions in the instruction queue target a register needed by the control instruction it is executing. This is in addition to any instruction which has issued but not completed. The “pre-busy bits” are used to track the predicates and few SPRs upon which a control instruction can depend. These pre-busy bits operate like the normal busy bits, but they reflect the pending writes of instructions that are in the instruction queue.

3.6 Instruction Issue

Each trace controller indicates to the issue logic if it has an instruction group, (minus the control instruction), which is ready to issue. This readiness test means assuring that no writes are pending on a register that needs to be read, and that any other resources which will be used are available. Context 0 is always given first shot at issuing,

followed by a round-robin ordering of the others. The chosen instruction's arithmetic instruction is passed on to the Arithmetic Pipeline, and its memory instruction is passed on to its Memory Pipeline.

3.7 Execution Pipelines

The Arithmetic and Memory Pipelines have a different number of pipeline stages which serve different purposes. The first stage of both has some similarities, however. For each instruction, the predicate is checked, if one is specified, to determine if the instruction will execute or not. Also, the first execution stage is used to compute new predicate values for instructions that modify predicates.

3.7.1 Arithmetic Pipeline

Arithmetic instructions may take one, two, or three cycles to complete. The Arithmetic Pipeline is tasked with producing only one value per cycle for register write-back, while moving other candidates along the pipeline for write-back on a future cycle. Arithmetic exceptions are not generated in the Hamal architecture; however, predicate bits p28 and p29 are set in the case of signed or unsigned integer overflow, respectively.

3.7.2 Memory Pipeline

The first stage of the memory pipeline is the only source of thread events. A trap is generated if the trace controller indicated one should happen, (such as in the case of an invalid op code), or due to a bad memory operation. If a trap occurs, the corresponding trace controller is notified that it should flush its instruction queue and start issuing instructions from its context's trap handler. These are not global events, so the context 0 event queue is not involved.

In the absence of events, the first stage's purpose is to perform address calculations and manipulations. Some memory instructions do not need to manipulate

data memory and will instead perform a write-back to the register file at the end of the cycle. The second memory pipeline stage is used by instructions wanting to be exported to the data memory via the Processor's data output. In the case that the memory instruction in the second memory pipeline stage is not permitted to go out to memory, (it is rejected, conflicts with another read or write from this context, or a higher priority source was selected), it will be queued-up in a per-context memory request buffer.

Most requests that go out have book-keeping information placed into a per-context request table. The table records which register, if any, should receive the data when the memory subsystem replies. The table is also used to prevent read-after-write (RAW), write-after-write (WAW), and write-after-read (WAR) hazards involving a particular location in memory; the assumption is that the requests might get re-ordered after they leave the processor, so conflicting writes should not be issued. The table also notes the permission bits of the capability used to access the data memory, so pointers that are read into the processor can be modified in accordance with the Take (**T**) and Diminished Take (**DT**) rights. Pointers read in without either permission have their **P** bit turned off; pointers read in with just the **DT** right have their **W**, **T**, **G**, and **DG** bits stripped in accordance with [32].

It should be noted that Hamal makes very few guarantees about the relative ordering of memory operations. It guarantees that operations to a single location in memory will appear to occur in the order they were issued, thus avoiding WAW, RAW, and WAR hazards. However, no guarantee is made about the relative ordering of requests to different locations in memory. Sequential consistency is not guaranteed. The `wait` instruction may be used to force a thread to stall until all of its outstanding memory operations have been completed. This is enough to implement "release consistency," as described in [12].

There is the possibility of contention among multiple sources of data wishing to access the data memory from the processor at the same time. Additionally, the node controller may reject a request if it lacks the resources to handle it at that moment in time. Context 0 is also given priority, as it is reserved for performing

critical operating-system operations and event handling. Requests that come out of the memory pipeline but cannot be dispatched due to contention are placed in a context-specific buffer. The sources are prioritized as follows:

1. A request from context 0 that was previously rejected by the node controller.
2. A buffered request from context 0.
3. A new request from the memory pipeline, belonging to context 0.
4. A request from a context other than 0 that was previously rejected by the node controller.
5. A buffered request from a context other than 0.
6. A new request from the memory pipeline, belonging to a context other than 0.

Another complication is that the processor may be in the process of swapping a context's state out to make room for another thread. If there exists a rejected request from a context other than 0, then this context swap is prioritized between items 3 and 4, above. Otherwise, it has a priority lower than all of the items in the list. Finally, if the event queue is starting to get too full, all memory requests from contexts other than 0 are suppressed.

3.8 Instruction Set

This section presents the instructions that are specified in the Hamal Instruction Set Architecture [18]. Control instructions which accept a source operand only accept an immediate value. Control instructions do not have destination registers, although they may modify `bp` in the course of execution. For arithmetic and memory instructions, however, operands may be:

- General-purpose registers.
- Special-purpose registers (pointer, event, and global).

- Predicates, as `pr`, $\sim pr^1$, `px`, and `py`.
- A 64-bit immediate.
- The constants 0 and 1.

GPRs can be used for any operand if an instruction has multiple operands. The others can only be used for one operand. Possible destinations for arithmetic and memory instructions are:

- General-purpose registers.
- The pointer registers `bp`, `tr`, and `tv`.
- Predicate registers en-masse: `pr`, `px`, and `py`.
- Individual predicates – compare and test instructions only.

Eight bits are required to encode the name of the operand or destination. For the set of compare and test instructions, the eight destination bits are interpreted not as a name of a register, but as five bits indicating a predicate and three bits indicating how the predicate should be updated. The update methods are enumerated in Table 3.3, from [18]. In the table, p refers to the predicate being set, and v refers to the one-bit result of the comparison instruction.

Bits	Meaning
000	$p = v$
001	$p = p \ \& \ v$
010	$p = p \ \ v$
011	$p = p \ \wedge \ v$
100	$p = !v$
101	$p = !(p \ \& \ v)$
110	$p = !(p \ \ v)$
111	$p = !(p \ \wedge \ v)$

Table 3.3: Predicate updates

¹The bit-wise negation of `pr`

3.9 Arithmetic Instructions

Name	Width	Ops	Description
<i>Predicate Instructions</i>			
eq, neq	64	2	Integer equality test
gt, lt	64	2	Integer comparison
igt, ilt	64	2	Signed integer comparison
feq, fneq	N	2	N bit fp equality test. $N = 64, 32$.
fgt, flt	N	2	N bit fp comparison. $N = 64, 32$.
test	128	2	AND test – set predicate to (src1 AND src2) != 0

Table 3.4: Arithmetic Predicate Instructions

The arithmetic instructions are meant to be handled by some number of functional units (FUs); implementations may decide the number of FUs and distribution of instructions to them. Predicate instructions, which perform comparisons and set or clear a single predicate, must complete in one cycle. The latency of other instructions is left as an implementation detail, although three cycles is likely a reasonable maximum.

The arithmetic instructions which have register destinations are listed in Table 3.5. The instructions which modify a predicate are listed in Table 3.4. The tables contain the name of the instruction, the width of each operand, the number of operands accepted, and a short description. The format of an arithmetic instruction is

$$\text{dst} = \text{OP src1 } [, \text{src2}]$$

where `dst` is a valid destination, and `src1` and the optional `src2` are valid sources, as described earlier. There may be a mismatch between the width of the data expected by the instruction, and the data provided by the selected sources. If the source data is narrower than the instruction data width, the data is zero or sign-extended depending on the instruction. Conversely, if the source data is too wide, it is truncated to the correct width. Extension and truncation are also used when attempting to place the data in the desired destination register.

Many of the arithmetic instructions operate on packed data in parallel, in a SIMD fashion. For example, the `add64` instruction operates on 128-bit operands, and thus

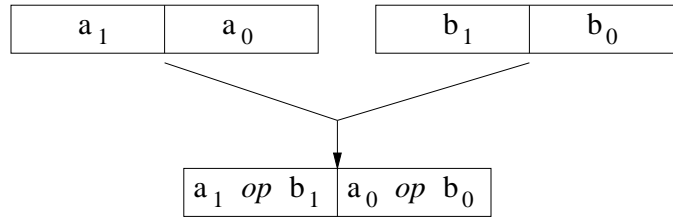


Figure 3-4: Parallel Arithmetic Operation

does two independent 64-bit adds in parallel. The low 64-bits of the destination will contain the result of adding the low 64-bits of the two sources, and likewise for the high 64-bits. Figure 3-4 illustrates this.

The `busy` instruction is a special instruction. It operates on no operands. Its purpose is to mark the destination register busy. This is useful for thread synchronization; see the description of the `join` memory instruction in section 3.10.6.

3.10 Memory Instructions

Five types of memory instructions are available: address computation instructions, load/store instructions, atomic memory operations, fork/join instructions, and the `wait` memory barrier instruction. The format of the address in memory instructions is `pointer[offset]`, where `pointer` is one of `r0-r31` and must contain a valid pointer. The `offset` is a 64-bit register or immediate and a multiplier of 1, 4, 8, or 16. Thus `r2[4*r3x]` is in the correct format. The third operand which can be provided to a memory instruction, `src3`, is a data word.

3.10.1 Address Computation Instructions

Address computation instructions, shown in Table 3.6, do not actually access memory. Instead, they perform calculations on pointers. The pointer comparison and equality testing instructions target predicates and can update the chosen predicate in the manner described earlier. Furthermore, `peq` and `pneq` will cause the `EV_SQUID` thread event if they cannot make the equality determination without more information— this happens when one or both pointers are for migrated objects and the SQUIDs and

Name	Width	Ops	Description
<i>Integer Arithmetic</i>			
add N	128	2	N bit parallel integer addition. $N = 64, 32, 16, 8$
sub N	128	2	N bit parallel integer subtraction. $N = 64, 32, 16, 8$
mul N	128	2	N to $2N$ bit parallel unsigned multiplication. $N = 64, 32$
muli N	128	2	N to $2N$ bit parallel signed multiplication. $N = 64, 32$
max N	128	2	N bit parallel integer maximum. $N = 64, 32, 16, 8$
min N	128	2	N bit parallel integer minimum. $N = 64, 32, 16, 8$
<i>Floating Point (fp) Arithmetic</i>			
fadd N	64	2	N bit parallel fp addition. $N = 64, 32$
fsub N	64	2	N bit parallel fp subtraction. $N = 64, 32$
fmul N	64	2	N bit parallel fp multiplication. $N = 64, 32$
fmax N	64	2	N bit parallel fp maximum. $N = 64, 32$
fmin N	64	2	N bit parallel fp minimum. $N = 64, 32$
fabs N	64	1	N bit parallel fp absolute value. $N = 64, 32$
frcpa N	64	1	N bit parallel fp reciprocal approximation. $N = 64, 32$
frsqta N	64	1	N bit parallel fp inverse square root approx. $N = 64, 32$
<i>Shifting</i>			
shl N	64	2	N bit parallel logical shift left. $N = 64, 32$
shr N	64	2	N bit parallel logical shift right. $N = 64, 32$
asr N	64	2	N bit parallel arithmetic shift right. $N = 64, 32$
rol N	64	2	N bit parallel rotate left. $N = 64, 32$
ror N	64	2	N bit parallel rotate right. $N = 64, 32$
<i>Bit Rearrangement</i>			
rev N	src1	1	Reverse the order of N bit blocks. $N = 16, 8, 1$
swiz	64	2	Swizzle
iswiz	64	2	Inverse swizzle
<i>Move</i>			
move	src1	1	Copy the contents of one register into another
imove	src1	2	Copy src1 into the register specified by src2
pack N	128	2	Parallel pack two N bit values $\rightarrow 2N$ bits. $N = 64, 32$.
<i>Bitwise Boolean</i>			
not	128	1	128-bit bitwise negation
and	128	2	128-bit bitwise AND
or	128	2	128-bit bitwise OR
xor	128	2	128-bit bitwise XOR
<i>Pattern Searching</i>			
scan N	src1	2	Scan N bit blocks for specified pattern. $N = 16, 8, 1$
scanr N	src1	2	Scan N bit blocks in reverse for pattern. $N = 16, 8, 1$
count N	src1	2	Count N bit pattern in N bit blocks. $N = 16, 8, 1$
<i>Hashing</i>			
hash	128	2	256 to 128 bit hash (two 128 bit inputs)
<i>Miscellaneous</i>			
busy	0	0	Marks the destination register busy

Table 3.5: Arithmetic Instructions

Instruction	Description
addr	Address computation (pointer + offset)
base	Compute the base address of an object
bound	Compute the maximum valid address of an object
peq, pneq	Pointer equality test
plt, pgt	Pointer comparison
psub	Subtract two pointers into the same object

Table 3.6: Address Computation Instructions

other properties are the same. Whether or not they really are the same would require dereferencing. Finally, `plt` and `pgt` will only return meaningful information if used on two pointers to the same object, although object migration is not an issue.

3.10.2 U and V Bits

User programs are allowed to modify, read, and trap on the `U` and `V` bits of each 128-bit word in memory. Every instruction which access memory is prefixed by eight bits which instruct the data memory how to modify and when to trap on these bits. Each bit is dealt with independently. For updates, the choices are set, clear, toggle, and no change. For trapping, the choices are ignore, trap if clear, and trap if set. To specify this in writing assembly language for Hamal, suffixes are added to memory instructions. The suffixes `@u0` and `@u1` indicate to trap if the `U` bit is 0 or 1. The update suffixes for `U` are `+u`, `-u`, and `^u`, which cause `U` to be set, cleared, and toggled, respectively. The same style is used for the `V` bit. Thus, to indicate a desire to load a 64-bit value from memory, toggling the `U` bit, ignoring `U` for purposes of trap, not modifying the `V` bit, and and trapping if `V` is set, one would write:

```
r1x = load64@v1^u r3[0]
```

3.10.3 Load/Store Instructions

These instructions allow writing to and reading from memory, at multiple granularities; 8, 16, 32, 64, or the full 128 bits (plus the `P` bit) can be operated on at a time.

As far as address alignment is concerned, the address must be a multiple of the data size. The `loadN` and `storeN` instructions do reads and writes, where N is one of the allowed granularities. The `swapN` instruction is combination of the two, reading an address' value and then storing another in its place. The U and V bits for a memory word may be loaded by using the `loaduv` instruction.

3.10.4 Non-trapping Instructions

When a memory instruction is executed by the processor, the capability used to access memory is checked for permissions, bounds violations, and granularity violations. If there is an error, the processor will generate an exception. However, non-trapping versions of the memory instructions are available if this behavior is not desired. If a trap would normally happen, it is suppressed, and predicate `p30` is set, but the instruction is still not allowed to continue execution. Otherwise, `p30` is cleared. To specify this in assembly language, one would prepend an underscore to the instruction.

3.10.5 Atomic Memory Instructions

The data memory is expected to be able to perform simple atomic operations on the data it contains. The atomic memory instructions specify an operand and an address. The memory subsystem will perform the desired operation with the current contents of that address and the operand, storing the result back into the same address. The original value of the data at that address is optionally returned. It is also possible to negate the original contents, the operand, and the result, as indicated by the optional tildes (`~`) in the syntax:

```
[dst =] [~]operation [~]pointer[offset], [~]src3
```

The available operations are addition, bitwise AND, bitwise XOR, and integer maximum. The operands may be 8, 16, 32, or 64 bits wide. The ability to negate certain values leads to even more functionality: any two-input boolean operation, integer minimum, and the functions listed in Table 3.7 can be computed.

Addition with optional negation	Same as
$a + b$	$a + b$
$a + \sim b$	$a - b - 1$
$\sim a + b$	$-a + b - 1$
$\sim a + \sim b$	$-a - b - 2$
$\sim(a + b)$	$-a - b - 1$
$\sim(a + \sim b)$	$b - a$
$\sim(\sim a + b)$	$a - b$
$\sim(\sim a + \sim b)$	$a + b + 1$

Table 3.7: Atomic Addition and Subtraction Operations

3.10.6 The Fork and Join Instructions

Threads are a central element of the Hamal architecture. In order to start a new thread, the `fork` instruction is used. This instruction takes two operands, an address and a 32-bit mask. The address indicates where the new thread should start executing; the processor in the node where this address resides will perform the execution. The mask indicates which of the 129-bit GPRs should be copied to the new thread.

The `join` instruction is used to send a value to a register in a different thread. It takes as operands the data to send and a join-type capability which indicates which node the thread is on and what the thread's identifier (swap address) is. The intended use is to perform register-based synchronization between threads. One thread uses the `busy` instruction to mark a register busy and then attempts to read from it, causing it to stall. When the other thread writes a value into the register, the register will be marked as ready and the first thread will be allowed to continue execution.

3.11 Control Instructions

The control instructions, in Table 3.8 from [18], all take zero or one operand. The only possible operand is a 24-bit immediate. The only register which can be modified is `bp`. The other side-effect, of course, is the changing of the next address to fetch an instruction from.

Instruction	Description
branch	Branch to the address in bp
branch <i>immediate</i>	Branch to a relative offset
return	Synonym for branch
treturn	Branch to the address in tr
call	Branch to the address in bp Store the next address in bp as the return address
call <i>immediate</i>	Same as call but with a relative offset
break	Terminate the thread and generate a global event
suspend	Suspend the thread and generate a global event

Table 3.8: Control Instructions

3.12 Privileged Instructions

There are a set of instructions that require additional privilege to execute. These instructions will only be permitted to execute if they are read from instruction memory using a code capability with the Privileged (P) bit set. Conceptually, threads using these instructions would be part of the operating system. Thanks to the use of capabilities, however, any thread could be handed a very restricted capability representing an entry point for one particular part of the operating system. Table 3.9, courtesy of [18], lists these privileged instructions. Some of the instructions are meant to be executed by context 0 only, and are noted with an asterisk (*).

3.12.1 Context Swapping

The operating system must be able to stop and remove a currently executing thread from the processor. It must also be able to reload and restart previously removed threads. When a thread is created, the operating system sets up a piece of memory where its state can be placed when it is swapped out. The address of this memory is known as the thread swap address and is used as a unique identifier for that thread. The **cstore** instruction takes as an operand a thread swap address. It causes the corresponding thread to be swapped out to memory. The **cload** instruction takes as operands the swap address of a thread to load and the number of the hardware

Instruction	Functional Unit	Description
poll *	arithmetic	Block until there is an event in the queue
cload *	memory	Background load a context from memory
cstore *	memory	Background store a context to memory
cwait *	memory	Wait for all <code>cstores/cloads</code> to complete
fload *	memory	Background load a context from fork buffer
fstore *	memory	Store fork buffer to memory
setp	arithmetic	Set the pointer bit of a 128-bit data word
loadt	memory	Get the T trap bit for a word in memory
storet	memory	Set the T trap bit for a word in memory
getmap	memory	Get the virtual address for a physical page
setmap	memory	Set the virtual address for a physical page
assoc	memory	Get the physical page for a virtual address
gets	memory	Get the status bits for a physical page
sets	memory	Set the status bits for a physical page
pgin	memory	Load a page of memory from secondary storage
pgout	memory	Store a page of memory to secondary storage
xlate	memory	Place an entry in the translation cache

Table 3.9: Privileged Instructions

context into which it should be loaded.

3.12.2 Managing the Fork Buffer

When the network controller's fork buffer contains a fork destined for the local processor, the processor will receive an `EV_FORK` event. The operating system handles must determine whether or not the processor has an unused hardware context. If it does, it can use the `fload` instruction to move the context data into the free context. Otherwise, it uses the `fstore` instruction to send the data to memory. The operating system can load this data later after it has swapped out one of the contexts.

Chapter 4

Processor Implementation

This chapter describes the heart of this work, the actual implementation of a subset of the Hamal Parallel Computer Architecture. The implementation is written in the Verilog Hardware Description Language, using only language constructs which make it suitable for synthesis into a Field-Programmable Gate Array (FPGA) produced by Xilinx, Inc. The goal is to have an FPGA implementation of the processor, although it is also possible to re-target the source code at an Application-Specific Integrated Circuit (ASIC) silicon fabrication process at some point in the future.

4.1 FPGA Details and Synthesis Tools

The processor implementation is targeted at the XCV2000E model of Xilinx's Virtex-E line of FPGAs [34] in a 680-pin package. Other parts in this line may be suitable, if they have greater or equal logic capacity and I/O pins, as the implementation makes use of almost all of the available chip resources in that model. The Foundation series of tools from Xilinx provided the logic synthesis, FPGA implementation, and timing analysis tools needed to complete the implementation. The Virtex-E FPGAs and these tools are now briefly described to give background on the equipment used in this work.

4.1.1 FPGA Features

An FPGA is a reconfigurable logic device that can implement arbitrary digital logic functions. The Xilinx Virtex-E FPGAs perform this using 4-input look-up tables (LUTs), as shown in [37]. Two LUTs, combined with dedicated carry-chain logic and two flip-flops, compose a “slice.” A Combinational Logic Block (CLB) is composed of two slices, along with additional logic to tie the LUT outputs together. A slice can compute any 5-input function, a 4:1 multiplexor, or selected functions of up to nine inputs. A pair of slices in a CLB can implement any 6-input function, an 8:1 multiplexor, or selected functions of up to nineteen inputs. Additionally, a CLB has local routing resources that can move data through a CLB without consuming any logic resources. The flip-flops are suitable for creating shallow RAM structures, which are called Distributed SelectRAM in the Xilinx nomenclature. Finally, CLBs also have three tri-state drivers, called BUFTs, for driving on-chip busses. The XCV2000E has 19,200 slices and 19,520 BUFTs.

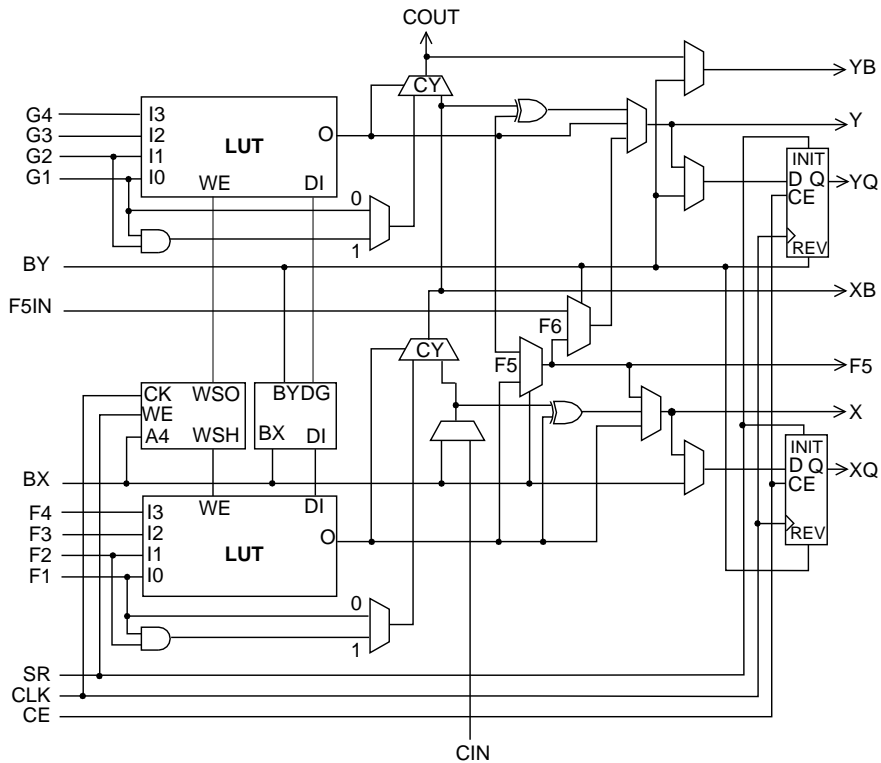


Figure 4-1: Detailed View of a Virtex-E Slice, from [37]

In addition to the logic resources in the FPGA, there are also dedicated static RAM, routing, I/O, and clock management resources. Block SelectRAM is the name given to the static RAM resource, which is organized into columns and is placed among columns of CLBs, as shown in Figure 4-1 from the Virtex-E data sheet, [37]. Each Block SelectRAM is a dual-ported synchronous 4 kilobit static RAM device. The XCV200E has 640 kilobits of this type of memory divided among 160 RAMs. Several types of programmable local and global routing resources are available, including dedicated global wires suitable for distributing clock signals in a low-skew manner. On-chip digital Delay Locked Loops (DLLs) are provided to assist in clock generation. Finally, a large number of common I/O standards are supported with the help of 512 I/O buffer (IOB) resources.

4.1.2 Tools Features

Implementation was completed with version 3.1i of the Xilinx Foundation tools, with Service Pack 7 applied. This suite of tools handles tasks such as design entry, simulation, synthesis, timing analysis, and implementation. The schematic entry tool was not used as the entire design was implemented in Verilog. Foundation's Hardware Description Language (HDL) editor was eschewed in favor of Emacs with Michael McNamara's excellent `verilog-mode` elisp package [33]. The simulator was also avoided, in favor of Model Technology, Inc.'s ModelSim product. The synthesis tools take a Verilog description and convert it to a netlist describing connections among simple logic primitives for the targeted device. The implementation tools take the output of the synthesis and map it onto the resources of the device. This is a multi-step process which includes a rather time-consuming place-and-route step. Xilinx's timing analyzer can be used after the implementation step to determine critical path length and maximum clock frequency. Figure 4-2 illustrates how the tools work together.

The Core Generator is a very useful tool which allows the creation of modules which efficiently use the resources of the FPGA to perform a certain task. For example, "cores" were available for FIFOs, tri-state multiplexors, and RAM structures. This tool outputs a netlist describing the Xilinx primitives to use, which is an input

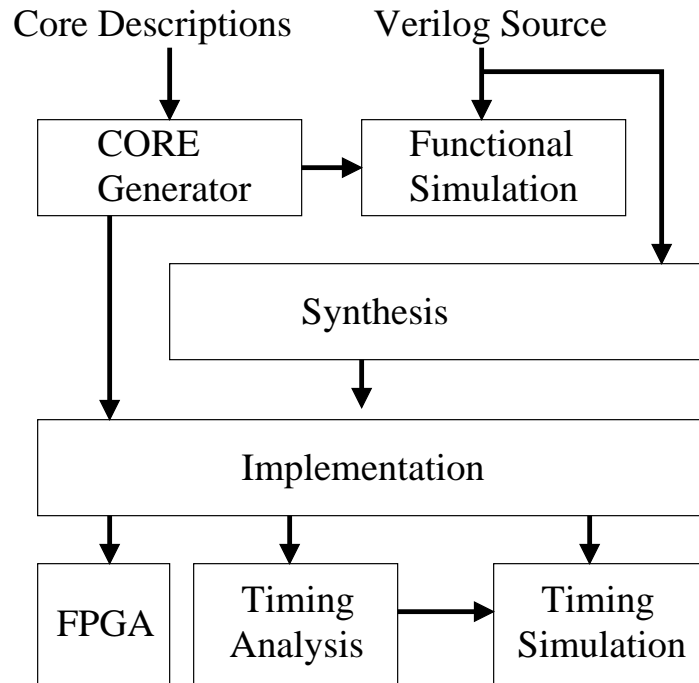


Figure 4-2: Tool Flow

to the implementation tools. Additionally, it generates a Verilog model of the module, so that functional verification of the design is possible with third-party Verilog simulators, such as ModelSim. Instantiating these modules in the Verilog code for the processor was preferable to trying to use the Xilinx primitives directly. Doing so allows the clear textual separation of the parts of the implementation which are Xilinx-dependent from those that are not. While it may have been preferable to avoid Xilinx-specific features altogether in favor of a pure Verilog description, this turned out not to be feasible. The Xilinx synthesis tools perform well in general, but to make best use of the on-chip tristate and memory resources, at least, it is necessary to be more explicit.

One really useful feature is the powerful logic optimizations that synthesis and implementation tools perform. In addition to detecting errors, such as multiple drivers on a single net, the tools also analyze the design enough to notice nets that do not have drivers, and nets that have no load. This extraneous logic is removed and noted in the tools' output. This is useful for two reasons. First, there is a chance that this is evidence of a design error. Second, the need to micro-manage busses and port

connections is considerably reduced. For instance, connecting a large bus to several modules, each of which only reads some parts of it, does not result in inefficient logic. The tools will remove unneeded ports and logic resources specified in the design. This extends to even removing registers and tri-state drivers for unused parts of the bus. Finally, the trimmed logic report may reveal to the implementor properties of the design that he was not aware of, and this information may help him make design optimizations beyond what the tools can do.

4.2 Implementation Details

Having introduced the technology being used, the implementation details of the processor are now presented. This description begins with the processor's interface to other parts of the Hamal Parallel Computer. A discussion of the various pipeline stages is next, followed by a detailed analysis of the context-specific hardware constructs, and then specifics of the memory output and input data paths. Finally the event-handling hardware is explained.

4.2.1 Node Controller Interface

There are three data paths between the node controller and the processor: Data output, data input, and event input, as shown in Figure 4-3.

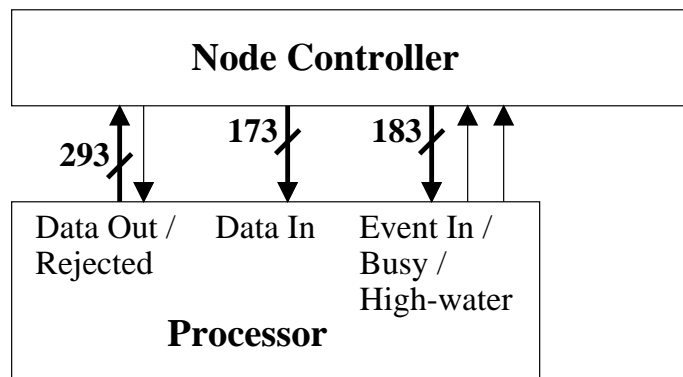


Figure 4-3: Processor-Node Controller Interface

Processor's Data Output

The data output is used to make memory requests and to swap out contexts. This data path is 273 bits wide. It is comprised of an address, the type of capability used (code, data, I/O, or join), a 129-bit tagged data word as an operand, the memory operation requested, the return address, and a bit indicating the validity of the request. Requests should be valid shortly after the rising clock edge. One additional wire is used by the node controller to reject or accept the processor's data output request. The node controller will examine the request, and may refuse it, if a necessary resource is unavailable, later in the same clock cycle.

Processor's Data Input

Replies to memory requests are received by the processor on its data input path. There are two busses and one control line associated with this interface. The busses provide the actual data and the return address. Along with one bit for indicating that the buses are valid, this data path is 193 bits wide. The processor is not at liberty to reject this input. Figure 4-4 shows the data paths involved.

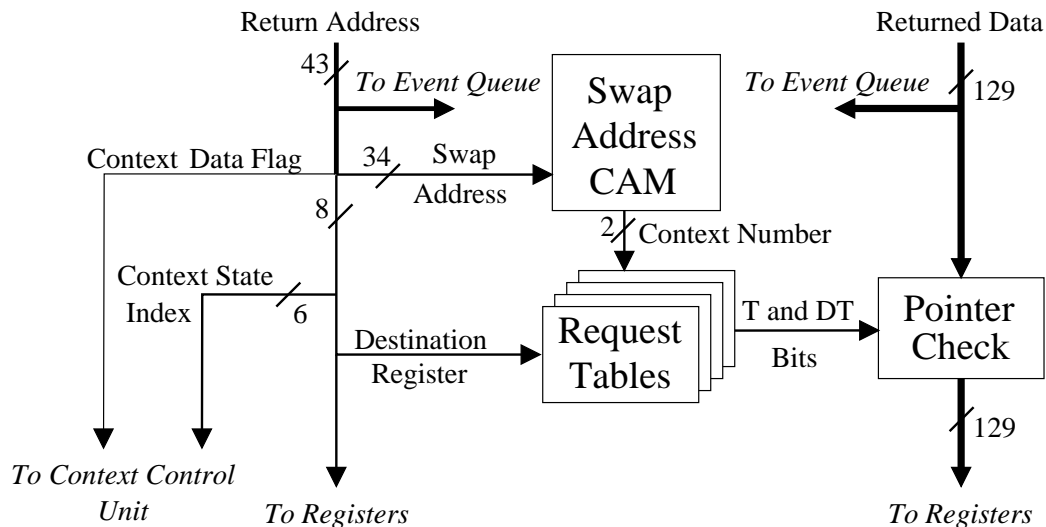


Figure 4-4: Memory Reply Data Path

The return address contains a thread swap address which identifies the thread for which the data is intended. If that thread is not loaded into a hardware context, the

processor will generate an `EV_REPLY` event and place the data into the event queue for the operating system to handle. The return address also indicates if this context data, from a `cload` or `fload` operation, or if it is a reply to a normal memory instruction. In the former case, the *context control unit* will decode the lower six bits of the return address to determine where to send the data.

In the latter case, the lower eight bits indicate the register to receive the data. This value will refer to a read-only register value if the memory instruction does not put data in a register when it completes, as is the case for `store`. This value, and the number of the hardware context which is receiving the reply, are sent to the memory request tables. The table corresponding to that context will clear the corresponding entry to indicate the memory operation has completed. Additionally, a check is made to see if the data read in is a pointer. If it is, the **T** and **DT** bits of the pointer originally used to make the memory reference are examined, and if necessary the incoming pointer is modified, as described in the previous chapter.

Processor's Event Input

The memory and network subsystems may have events to report to the processor, such as a page fault or a fork request arriving from a remote node. The processor's 185-bit wide event interface supports this. This interface is composed of the type of event, the memory operation involved in the event, the return address, and an additional event-dependent data word. The processor has a control line for indicating that an event will not be accepted in the following cycle. It also has the "event high-water" control line, which is asserted when the processor's event queue is starting to fill up. The node controller will stop accepting network events, joins, or remote memory requests to ease the pressure.

4.2.2 Instruction Memory Interface

Each clock cycle, the processor looks at the trace controllers which are ready to receive another instruction group from the instruction memory. It selects one of these

contexts, giving context 0 priority and using a round-robin schedule for the others. The processor outputs the address being requested and the number of the context which requested it and asserts the instruction memory request line to notify the instruction memory of the desire for service. The selected trace controller is notified that the request succeeded; it will not make further instruction memory requests until it is supplied with the requested data. If the instruction memory decides it is busy and unable to accept a request this cycle, it will assert `imem_busy`, and the processor will not make a request. Figure 4-5 shows how the processor and instruction memory communicate.

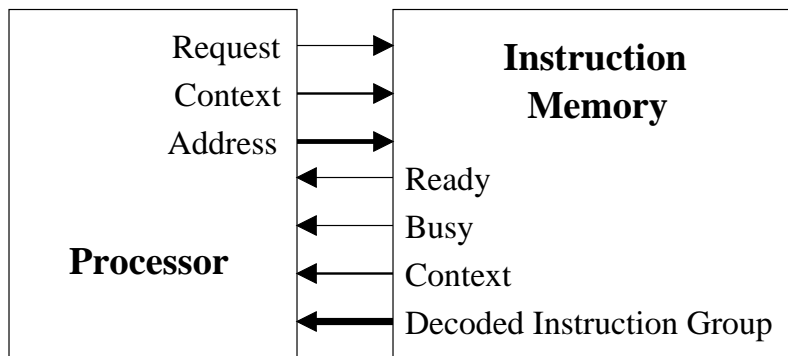


Figure 4-5: Processor-Instruction Memory Interface

The instruction memory responds with the desired instruction group and the number of the context which requested it. It may take multiple cycles to fulfill the request. If the latency in cycles is always greater than the number of contexts that typically are active accessing instruction memory at the same time, then the processor will be starving for instructions to execute. Another issue is that if a context with a pending request is stopped by the operating system to be swapped out, the new context which is swapped in must not start until the instruction memory has responded to the request. This prevents the new context from being handed an instruction group meant for the old context. The instruction memory's maximum latency must be less than the minimum number of cycles required to install and start a new thread. Since a context swap takes at least 110 clock cycles, this should not be an issue. Contexts which terminate or suspend themselves by executing a `break` or `suspend` will not have pending requests.

Decoded Instruction Group Format

Field	Size	Field	Size
<i>Predication</i>		<i>Memory Instructions</i>	
Predicated	1	Predicate Destination	1
Invert Predicate	1	Privileged only	1
Predicate Number	5	Context 0 only	1
<i>Arithmetic Instructions</i>		Opcode	6
Predicate destination	1	Flags	20
Privileged only	1	First operand	8
Context 0 only	1	Second operand	8
Opcode	6	Third operand	8
N	2	Destination	8
Sets overflow flags	1	Immediate	64
First operand	8	<i>Control Instructions</i>	
Second operand	8	Opcode	3
Destination	8	Immediate	21
Immediate	64		

Table 4.1: Decoded Instruction Format

Table 4.1 lists the elements of a decoded instruction group. Included are the opcodes for the three instructions in the group, the sources and destinations for the arithmetic and memory instructions, predication information for each instruction, and immediate values. The 64-bit immediate is shared between by arithmetic and memory instructions. In encoded form, which is only 128 bits long, an instruction group will not be able to provide data for all of these fields. Different instruction templates would require different subsets; the implementation is able to support the union of these sets. Without much effort, a control instruction can be packed into 3 bits, an arithmetic instruction into 40, and a memory operation into 56. Allowing 6 bits for predicate selection and two for template selection, four possible ways to pack instructions into a single instruction group are:

1. Arithmetic + Memory + Control + 21 bit immediate
2. Arithmetic + Memory + 24 bit immediate

3. Arithmetic + Control + 64 bit immediate + 13 bit immediate

4. Memory + 64 bit immediate

Other instruction templates are certainly possible. This implementation does not make assumptions regarding instruction group encoding. It expects instruction groups to be delivered in decoded form.

Flag Name	Size
Scale	2
Granularity	3
uv options	8
Physical Page Operand	1
Store Operation	1
Load Operation	1
Uses ptr[offset] addressing	1
Address computation only	1
Do not add to request table	1
Atomic memory operation	1
No trap on exception; set flag	1

Table 4.2: Memory Instruction Flags

4.2.3 Split-Pipeline

The Hamal architecture involves a pipeline that splits in two after the issue stage. As shown in Figure 4-6, there are two stages before the split, instruction fetch and issue. The control instruction is executed in the instruction fetch stage, if its dependents are not going to be overwritten by a queued or in-flight instruction; otherwise the trace controller waits for the next cycle to try again. Only one context can receive an instruction group from instruction memory per cycle, although all contexts can execute stalled control instructions at any time.

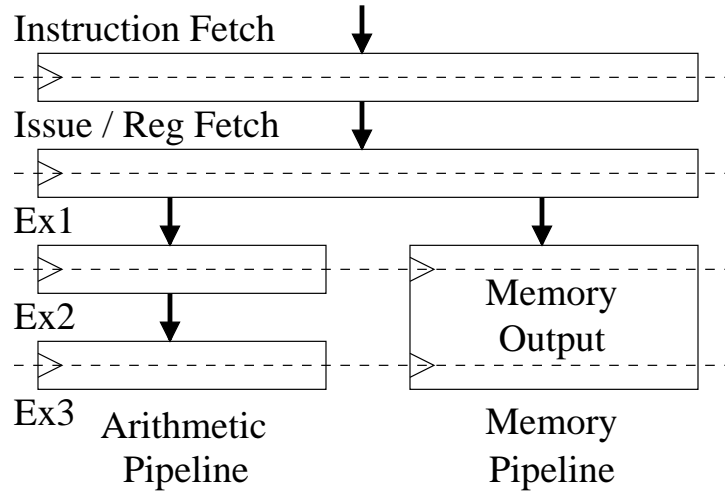


Figure 4-6: Hamal Pipeline Stages

4.2.4 Trace Controller

The trace controller is responsible for managing all of the context-specific resources. It determines the address of the next instruction to request from the instruction memory, executes control instructions, tracks pending register use, determines if its context is ready to issue, handles thread events, and executes write-backs to its special-purpose registers. A register named `tp` serves as the thread's current instruction pointer. This is not an architecturally-visible register, but is needed for the trace controller to know where to get the next instruction from.

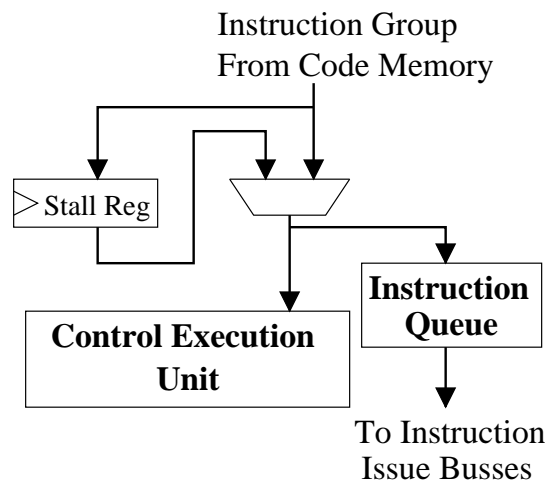


Figure 4-7: Trace Controller Instruction Flow

Control Execution Unit

Each trace controller includes a control execution unit. This unit is responsible for executing the control portion of an instruction group. For a given control instruction, this unit first does dependency checking to determine if the register(s) which need to be read from are going to be modified by an instruction that is in the instruction queue or has already issued. If the resources are available, the predication of the instruction is checked. Then, the address of the next address is determined. As shown in Table 4.3, this will be one of the following: `tp + 1`, `tp + an immediate`, the contents of `bp`, or the contents of `tr`. Finally, the opcode is examined to determine if `bp` should be loaded with the next address to execute; this happens in the case of `call` instructions only. The trace controller is notified if: the instruction executed successfully, resources were not available, or the executed instruction was `suspend` or `break`. If the resources are not available, the trace controller will place the instruction group in its stall registers and try again next cycle. It will also refrain from requesting another instruction group from the instruction memory. Figure 4-7 shows how instruction groups flow through the trace controller.

Instruction	Next Address
<code>branch</code>	<code>bp</code>
<code>branchi</code>	<code>tp + immediate</code>
<code>treturn</code>	<code>tr</code>
<code>call</code>	<code>bp</code>
<code>calli</code>	<code>tp + immediate</code>
The rest	<code>tp + 16</code>

Table 4.3: Next Address Determination

Generating Events

There are four events, or traps, that could be generated by the trace controller. They are `EV_SUSPEND`, `EV_BREAK`, `EV_BADOP`, and `EV_BADIP`. The first two are processor (global) events, and the last two are thread (local) events. In the case of the global events, execution of the thread stops, and the the trace controller waits for any

arithmetic and single-cycle memory instructions it has issued to finish. Then, the trace controller indicates to the issue logic that it has an event it would like to submit to the processor. It will continue to indicate such until an acknowledgement is received, at which point the context will be idle, waiting for operating system intervention.

For the local events, a special memory operation is created to replace the actual memory operation for the instruction group in question. This special operation indicates that a trap should be generated when the instruction reaches the first stage of the memory pipeline. The arithmetic instruction is replaced with the no-operation instruction, `NOP`. The bad opcode event, `EV_BADOP`, will occur if the instruction decode logic deems the instruction group to be invalid; in this case, the instruction received from the decode logic will already have the memory instruction altered as described. The bad instruction pointer event `BAD_IP` occurs when the the trace controller attempts to fetch an instruction but `tp` contains a capability that is not valid.

Instruction Queue

After instruction groups are successfully executed by the control execution unit, they are added to the instruction queue. Attempts to issue are done using the instruction group that is at the head of the queue, by peeking at this entry but not dequeuing it. The instruction queue must also report on the writing of the predicates, `bp`, and `tr` by instructions sitting in the queue. These are known as the pre-busy bits, and are needed by the control execution unit to determine if it is safe to read or modify these registers.

Thus the implementation requires a FIFO-structure, but with the ability to read certain bits from all entries and the ability to peek at the item at the front of the queue. A modest depth, two entries, is the minimum desired. Whether or not a greater depth is an efficient use of resources is not yet known; simulations should reveal if these queues typically fill causing the instruction memory to go idle or not. This was implemented in straight Verilog, hard-coding for two elements, since the Xilinx FIFO core does not provide the desired features— the top entry cannot be read

without dequeuing it, and parallel access to all of the pre-busy bits of all the entries is not possible.

Resource Usage and Issue Readiness Determination

Each trace controller must guarantee that that sequential consistency is achieved for reads and writes to a given register. As described in the previous chapter, busy bits are used for to track pending register writes. The trace controller checks the sources and destinations of the instruction group that is at the front of the instruction queue. If any of those registers have a write pending, the instruction is not allowed to issue.

The GPRs, due to their multi-granularity, require a small amount of additional effort. If even part of the chosen GPR is busy, the instruction must wait. This could happen if, for example, `r0x` is busy, and the candidate instruction group has `r0` or `r0b` as a source or destination. On the other hand, using `r0y` would be fine. The predicates have a similar concern, as they can be read and written en-masse, as `pr`, `px`, or `py`, and they can be read and written individually, by comparison instructions or predicated instructions.

Other than the sources, destinations, and optional predication, there are other more subtle register resource uses that must be checked for. Non-trapping memory instructions will set or clear `p30`, and some arithmetic operations will set or clear the inexact flags, `p28` and `p29`. Finally, any memory instruction which could trap implicitly reads the trap vector register, `tv`, and writes the trap return register, `tr`.

As stated before, the control execution unit has to do these resource checks as well, although on a much smaller set of registers due to what control instructions can actually manipulate. The need for pre-busy bits is now clear—checking the busy bits for instructions which have already issued is not enough, and it is important to consider the resource use of the instructions in the instruction queue.

The busy bits for an instruction which issued are not committed at the end of the issue stage. Instead, they are committed at the end of the next pipeline stage, `ex1`. The reason for this is that a trap could occur, or one or both instructions could get squashed due to failed predication. The possibility of a trap is not known until the

first stage of the memory pipeline, and predicates are also not checked until `ex1`. The execution unit reports to the trace controller whose instruction group is in `ex1` about the result of predication and the presence of absence of a trap.

There are resources other than registers whose usage which must be tracked. First, there are a finite number of memory request buffers and request table slots for each context. These structures are described in the description of the memory output path below. If these structures do not have enough space to hold any more memory requests, then instruction groups with memory instructions that access memory are not allowed to issue. Additionally, the `wait` instruction causes issue to be delayed until there are no pending (or in-flight) memory requests.

Finally, there are global resources which must be shared by all contexts on the processor. Only one fork or context swap-out can be happening at a time. Also, only one page-out operation is allowed at a time. The trace controller must check these resources as well. Each of the two resources has a busy bit associated with, like the registers, but these busy bits reside outside the individual trace controllers.

Handling Thread Events

All thread events, or traps, are generated in the first stage of the memory pipeline. All of the trace controllers share a bus which reports these events. This bus, shown in Figure 4-8, carries the type of trap, the address of the instruction group that caused the trap, and details of the memory instruction in that group. Arithmetic and control instructions do not directly generate thread events. The details of the memory instruction include the operation being performed, the operands, and the destination register. If this trap is the result of bad instruction encoding, as described earlier, then only the trap type and instruction group address are valid. This information is stored in the seven event registers `e0` through `e6`. Additionally, the busy bits for the memory instruction are not committed, as if the instruction had been squashed by failed predication. The address of the erring instruction is placed in the trap return register, `tr`, and the thread instruction pointer, `tp`, is loaded with the contents of the the thread trap vector register, `tv`.

Traps are precise in the Hamal architecture. To undo the execution of the control portions of instruction groups in the instruction queue, the instruction queue is flushed and the branch pointer, `bp`, is restored to the value saved when the faulty instruction group was executed by the control execution unit. The only other side-effect that the execution of control instructions has is modifications to `tp`, but this register receives a new value as a result of the trap anyway. These modifications are all made at the end of the `ex1` stage for the instruction that caused the problem. Additionally, the trace controller will not indicate it is able to issue this cycle due to the trap. However, it will be able to begin execution of the trap handler on the cycle following, if chosen to issue. Thus, if this were the only thread ready to issue, there would be a one-cycle bubble where no context issued.

Special-Purpose Register Write-back

The general-purpose register file handles all GPR writes, but it is up to each trace controller to execute writes to all other registers, the SPRs. Predicate write-back is one part of this. The execution unit will provide information about how predicates change as a result of executing the arithmetic and memory instructions; this always happens during the `ex1` pipeline stage.

Instructions are also able to write to a subset of the SPRs, including the predicates en-masse, as described in the previous chapter. At the end of each cycle, the trace controllers must check to see if there are any SPR write-backs for their context happening and execute them if there are. The trace controllers may assume that there will never be multiple writes to a single SPR in the same cycle, as it would be a breach of the architectural rules to issue an instruction which intended to write to a register which already had a write pending.

4.2.5 Issue Logic

The issue stage is examined next. At the start of the clock cycle, all of the trace controllers indicate whether or not they have an instruction group that is ready

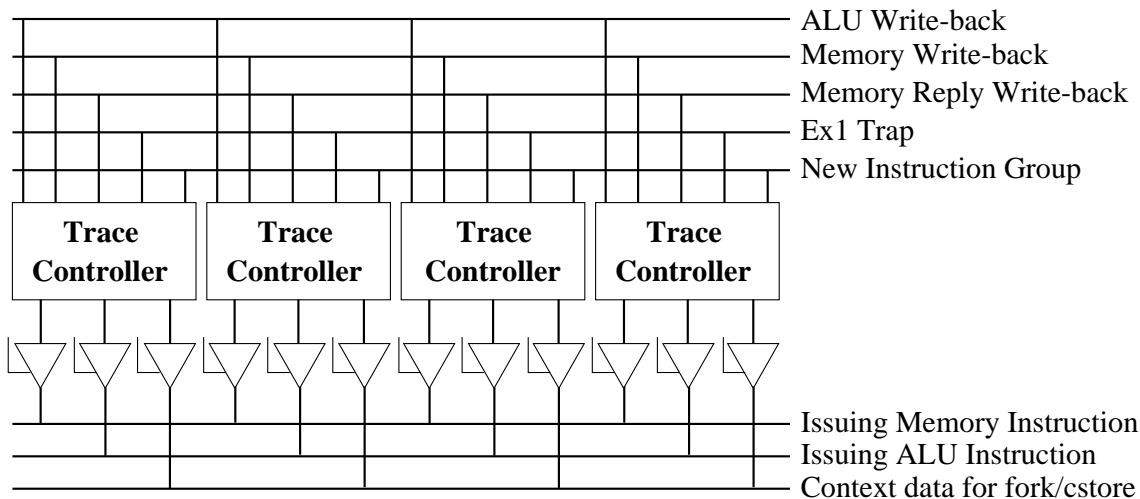


Figure 4-8: Trace Controller Data Paths

to issue. Each trace controller accesses its special-purpose registers to produce the operands requested by the instruction group. At the same time, the issue logic looks at which contexts can issue, which context issued last time, and whether or not context 0 wants to issue in order to determine which context will issue at the end of the cycle. Context 0 is given priority, with the other contexts taking turns in a round-robin fashion. For four contexts, the logic is not too complicated, although if eight contexts were used, a table-lookup approach might be more efficient. If a context is not ready to issue, it is skipped in the round-robin handling. The selected instruction group is chosen out of the candidates from each context using a multiplexor. Having decided which context will issue, the GPR operands of the instruction group are fetched. The rising clock edge at the end of the cycle is the commit point. The issuing trace-controller removes the issued instruction from its instruction queue, and the chosen instruction group and accompanying operands are clocked into the pipeline register of the first execution stage, `ex1`. The processor informs the execution unit that the instruction in `ex1` is valid and should be executed.

4.2.6 General-Purpose Register File

The general-purpose register file holds the GPRs for each hardware context. It has six read ports and 3 write ports. Two read ports are for the sources of an arithmetic

instruction, three are for the sources of a memory instruction, and the third is for swapping out a context. The three write ports are for write-backs from the arithmetic and memory pipelines as well replies from memory. It was necessary to understand the FPGA resources available when designing this part. Simply generating a Verilog description of what should happen was inadequate; the synthesis tool were not able to process this effectively. The tool spent about 12 hours executing on a dual Intel Pentium-III 550Mhz machine before conceding that it could not fit the register file into the FPGA at all.

The solution was to be more explicit about how the FPGA resources were to be used, as the synthesis tool was not able to infer this on its own. For storing the data, the FPGA's static RAM resources were used. The RAM primitive is dual-ported, so simultaneously supporting six reads and three writes took some doing. The implementation clocks the register file twice as fast as the clock for the rest of the system. This effectively splits each cycle of the system into two phases, a write phase and a read/write phase. During the write phase, both ports are used to write, while during the read/write phase, one port is used to write and one is used to read. That covers the requirement of three writes per cycle. To get the six reads, six copies of the RAMs are used, all set to write the same data.

Another complication is the fact that due to the granularity of the register file, there are four 32-bit portions and one 1-bit portion that can be written to separately. There are logically five RAMs, with six copies each. Physically, the 32-bit wide RAMs require two Xilinx Block RAMs to implement. Thus, 54 Block RAMs are used in total, requiring 216 kilobits of RAM resources to implement the 16.1 kilobit register file. This may seem wasteful, but the FPGA has enough Block RAM resources that this the optimal, and perhaps only, way to implement a register file with these specification in this FPGA.

Finally, five of the read ports, shown in Figure 4-9, must support the reading of a 32, 64, or 129-bit value. Reads from the **b**, **c**, **d**, and **y** register slices require bit shifting so that the output bits start at the least significant bit out of the output bus. The data output multiplexors handle this task.

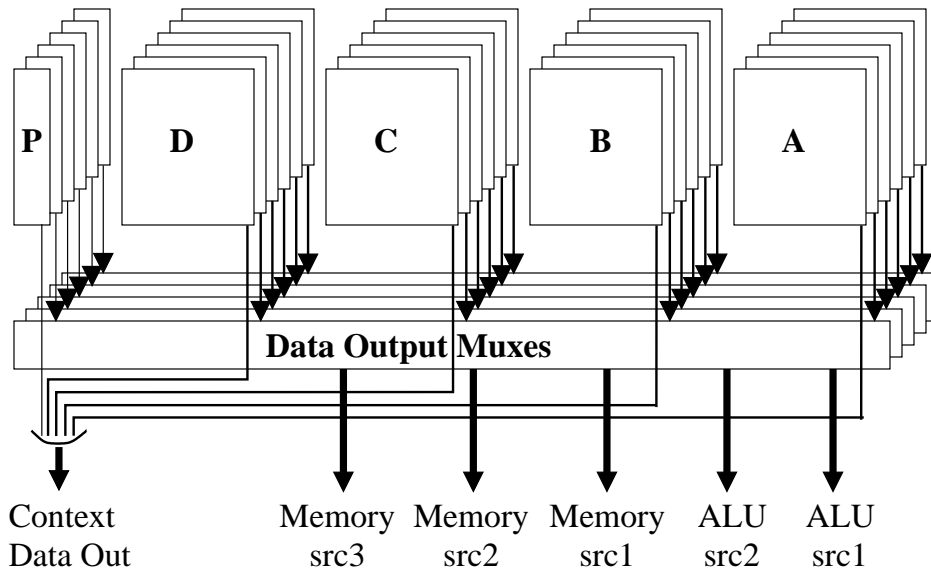


Figure 4-9: General Purpose Register File Read Ports

4.2.7 Execution Unit

There are two parallel `ex1` pipeline stages, one for arithmetic instructions and one for memory instructions. They are referred to as `apipe_ex1` and `mpipe_ex1` for short. The Execution Unit checks the predication for each of the two instructions and the current value of the predicates from the issuing context to determine if either instruction should be replaced with a NOP. Another task that the execution unit performs is combining the predicate write-back requests from the the hardware executing the arithmetic and memory instructions. Either instruction might have a predicate destination and/or want to set or clear one or more of the predicates used to flag error conditions. Additionally, instructions that target individual predicates specify an update method, and that is handled here as well.

This pipeline stage is the commit point for the instruction group. If the predication allows it, and the Memory Execution unit indicates there is no trap this cycle, then at the end of the cycle the trace controller that issued this instruction group will commit the busy bits for that group. The trace controller assumes that the instruction group will not be squashed when making the determination if it is safe to issue on the cycle following a successful issue. This is conservative, and results in an issue bubble in the case of a trap. However, traps are supposed to be exceptional situations and there

are up to three other threads waiting to issue.

4.2.8 Memory Pipeline

The first stage of the memory pipeline will execute instructions that do not require communicating with the memory subsystem. These are the address-manipulation instructions described in the previous chapter. For memory instructions that require it, the given offset and base address are combined to produce the actual address being referenced. For all memory instructions, checks are performed to determine that valid capabilities are being used for addressing, and if the requested operation violates the permissions specified in the capability. For example, data supplied as an address must have the pointer bit set, indicating it is a capability.

Bounds checking will be performed on the computed address, and security violations such as using a pointer without the Read capability to perform a read operation will be caught. Additionally, an address alignment check is made using the granularity of the memory reference. For example, if a memory instruction is writing a 64-bit value, the address' lowest three bits must be zeroes. It calculates if a trap should happen, or if the result of an address-manipulation instruction should be written back to the register file, or if the instruction needs to continue on to the second memory pipeline stage to communicate with the memory subsystem. Additionally, if instead of a memory instruction an event from the issuing trace controller is present (`EV_BADOP` or `EV_BADIP`), the need for a trap will be signalled.

All of the traps that can get signalled at this stage, listed in Table 4.4, are thread events. The notification of a trap will be forwarded to the trace controller that issued this instruction group.

The second stage of the pipeline is a staging ground for the processor's data memory output. The details of this memory output stage are discussed later.

Event	Cause
EV_BADIP	Thread's instruction pointer is invalid Attempt to execute outside allowed segment
EV_BADOP	Illegal opcode or register selection
EV_SQUID	SQUIDs of one or more migrated objects match in pointer compare; pointer dereferencing required to judge equivalence
EV_BADTHREAD	A <code>cstore</code> operation referred to a non-existent thread
EV_PERM	Capability permissions forbid the requested operation
EV_BOUND	Address is outside the segment permitted by capability
EV_ALIGN	Address is improperly aligned for the chosen granularity

Table 4.4: Thread Events

4.2.9 Arithmetic Pipeline

The arithmetic pipeline consists of three stages. The arithmetic logic unit (ALU) is divided into groups of functional units based on the number of cycles the functional units need to compute their results. There are one, two, and three stage units. Note that the depth of the pipeline is not fixed in the architecture, although three seems to be quite reasonable. One complication in the ALU's logic is the fact that up to three instructions may complete in a single cycle. This occurs when the ALU receives, on successive cycles, a three-cycle operation, a two-cycle operation, and a single-cycle operation. Any time multiple instructions complete and are ready to write-back to the register file at the same time, the instruction finishing closest to the end of the pipeline is allowed to write-back. Any other instructions which are also done have their results moved forward to the next set of pipeline registers.

Parallel Arithmetic Operations

The multi-granular parallel arithmetic operations that are part of the Hamal specification add complexity and area to the implementation of the ALU. Sometimes, it is an efficient use of logic resources to share pieces of the hardware meant to operate on data of different widths. The addition/subtraction hardware illustrates this well. The instruction set architecture calls for 8, 16, 32, and 64 bit addition and subtraction. This work's implementation of this feature uses 16 eight-bit adders. The

carry/borrow output of each adder is connected to the carry/borrow input of the next through a small multiplexor. That multiplexor controls whether or not to allow carry/borrow propagation to the next piece of the adder by choosing either the carry out from the previous stage or a 0 for addition or 1 for subtraction. The desired width of the data to be operated in parallel is used to determine the control signals for these multiplexors. For 8-bit data, none of the carries would be allowed to propagate. For a 64-bit add, all of the carries propagate except for the one between the upper and lower 64-bit chunks.

Other times, separate hardware is needed for separate bit widths. The two 32-bit and one 64-bit multipliers are an example of this. The amount of extra logic required to augment the results of the two 32-bit products to get the 64-bit product is more than the logic required to just implement them separately.

4.2.10 Memory Output Path

Requests to talk to the memory subsystem originate from two sources: the second stage of the memory pipeline, and a context swap-out. The output path ends at the interface to the node controller. There are some complications, however. First, there's the fact that there are two competing sources of data. Second, that node controller has the right to reject requests. The need to be able to buffer requests that cannot yet go out is apparent. Finally, each context needs to track pending memory operations to prevent write-after-write (WAW), read-after-write (RAW), and write-after-read (WAR) hazards since requests may get reordered after being handed off to the memory system. There is also a priority system, described in the architecture description of the previous chapter, which must be followed when multiple requests are ready to be sent out at the same time.

Rejected Requests

The node controller is permitted to reject outgoing data requests. In such a situation, two possible scenarios can occur. First, if nothing with higher priority needs to go

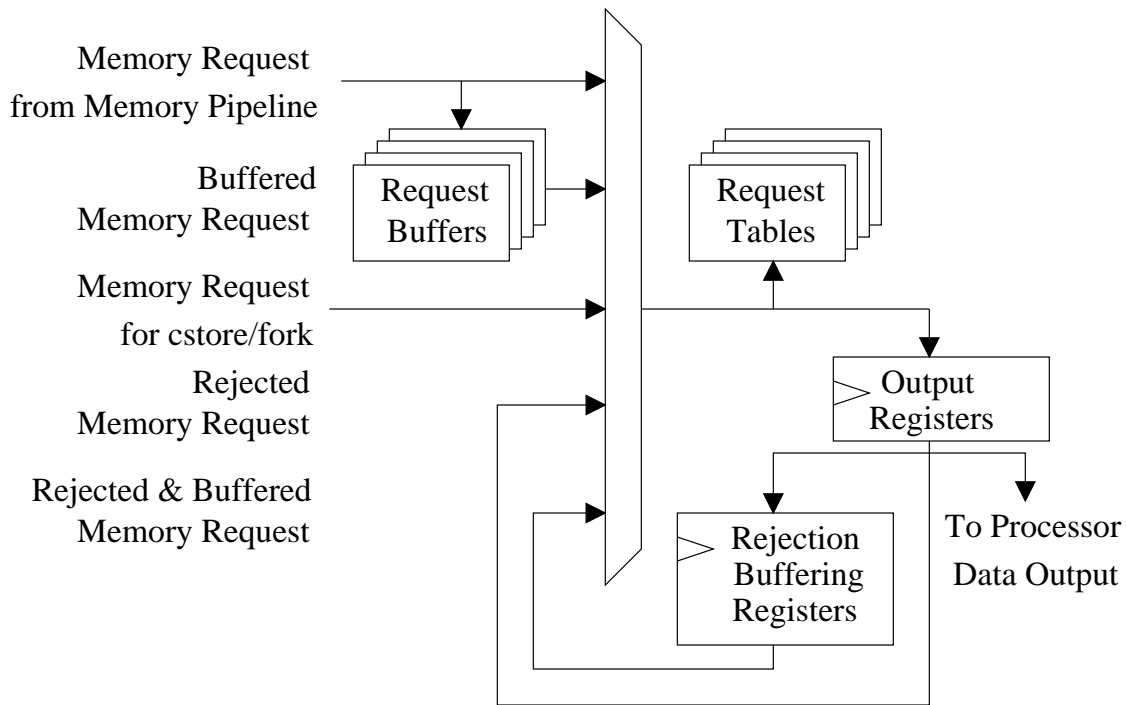


Figure 4-10: Memory Output Data Path

out, (a context 0 request), then the request is tried again the next cycle. If, however, a higher priority request exists, then the rejected request is buffered in what is known as the third stage of the memory pipeline, `mpipe_ex3`. The request stays there until there are no more higher-priority requests waiting. All of the requests that would be favored in this manner have equal priority, and would thus keep retrying if rejected. There would be no attempt to overwrite the buffered rejected request.

Memory Request Buffer

In order to handle requests from `mpipe_ex2` that cannot go out, either due to a conflict with a pending request, or because a higher-priority request has been selected to output, each context has a memory request buffer. The buffer is basically a shallow queue, and much like the instruction queue, it is possible to read the item at the front of the queue without dequeuing it. Requests sit in the buffer until that buffer is selected as the source for the next outgoing memory request. The contents of this buffer are part of the context's state for purposes of context swapping.

Request Format

Figure 4-11 shows the elements that form a complete memory request, which is 269 bits wide. This is what is stored in the memory request buffers as well as in the `mpipe_ex3` pipeline registers. When a request actually is transferred to the node controller, additional bits are added. The context ID is removed, while the node ID and the swap address of the context issuing the request are combined and added to the return address for the request. This is information which the node controller must have to reply to the request when it completes, but data which need not be stored in various buffers before the request is issued.

Address 64	Type 2	T 1	DT 1	SQUID 12	B 6	L 5	K 5	Data 129	Op 34	Context 2	Return Reg 8
---------------	-----------	--------	---------	-------------	--------	--------	--------	-------------	----------	--------------	-----------------

Figure 4-11: Memory Request Format

Once a request goes out, far less information must be retained for storage in the memory request table. This information is the SQUID, B, L, word offset, and the Take and Diminished Take permissions from the capability used to address memory. The word offset is the offset without its lowest five bits; this uniquely identifies a 128-bit word rather than a byte. With the exception of the two permission bits, these items are combined to form the “tag” in the memory request tables.

Memory Request Table

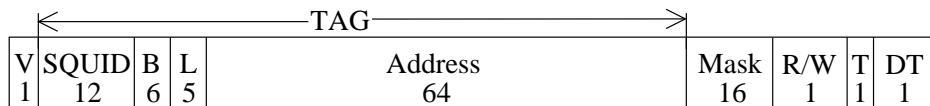


Figure 4-12: Memory Request Table Entry

Each entry in the table consists of a valid bit, an 87-bit tag, just described, a 16-bit byte mask, the two permission bits extracted from the original request, and an 8-bit destination. This is illustrated in Figure 4-12. The table must satisfy several properties. First, it must be possible to compare an incoming request with all others

in parallel to check for a conflict. Second, it must be possible to add a request to a free slot. Third, requests with no destination must store the index number of the slot position occupied rather than the null destination. This provides a unique identifier for all pending requests without destination, such that it is possible to remove the correct entry when the reply is received. Fourth, it must be possible to lookup an entry by supplying the eight destination bits and retrieve the permission bits, and optionally remove the entry from the table. Fifth, it must be possible to read and write each entry individually for the purposes of swapping context data in and out. Thus, the table must in some ways act as a typical RAM structure while in others it must be more like Content-Addressable Memory (CAM).

The ultimate purpose of this table is to prevent RAW, WAR, and WAW hazards. Thus, the table must detect memory accesses to the same bytes in an object. The first step is to create a byte mask based on which bytes out of the sixteen in a 128-bit word the request is accessing. The lower four bits of the offset combined with the granularity of the request are sufficient to determine the mask. A conflict may occur if the SQUID, word address, B, and L values for any existing entry and the new request match. It is possible for two SQUIDs to match but for different objects to be involved; this scheme is conservative to avoid having to do an expensive dereference operation to determine if two pointers are absolutely pointing to the same object when migration is involved. A conflict will be judged as having occurred if the new request entry is a write operation, or if the conflicting entry is a write operation. This catches WAR, WAR, and RAW without tripping on read-after-read, which is not hazardous.

Each entry in the table shares a bi-directional bus to enable reading and writing of the entire entry during swap operations. Also, each entry is able to accept new data, clear its valid bit, and report if the shared bus contains a conflicting request. The logic for the table itself is mostly concerned with checking to see if any entries conflict with a candidate request, and setting the write-enable for just the first available entry if there is no conflict and the request is to be committed to the table.

4.2.11 Thread Management and the Context Control Unit

The Hamal Parallel Computer features hardware support for thread management. Specifically, a single instruction is sufficient for starting a background context swap-in or swap-out operation. Also, the `fork` instruction, which creates a new thread of execution, allows the user to specify which GPRs should be copied from the executing thread to the new thread. The hardware takes care of this request without further instruction from the user.

The Context Control Unit (CCU) is the piece of logic responsible for making this a reality. It generates control signals that arbitrate access to the context state bus. This bus can be driven by the trace controllers, for writing SPRs, by the register file, for writing GPRs, and by the memory interface for writing buffered memory requests and memory request table entries. Swapping out a context, which is initiated by the `cstore` instruction, involves sending all of this data out. New thread creation, with `fork`, only involves sending out some number of GPRs. No matter what the data, the request goes onto the context state bus which is connected to the memory output path.

Fork

The `fork` instruction includes a 32-bit operand which indicates which of the 32 general-purpose registers should be copied to the new thread. The CCU must generate memory output requests for each of the registers that has the corresponding bit set in the 32-bit mask. Additionally, the CCU must be able to generate one new request per cycle, in case each request is accepted as soon as it is ready. In other words, if there are M bits set in the mask, only M cycles should be required to send the requested data out, if there is no contention for the processor's data output. A Finite State Machine (FSM) is used to accomplish this. When a `fork` request is made, the FSM state is initialized to the mask value. The FSM outputs the next state, which is the old state with the lowest set bit now cleared, as well as the GPR number to read from, and a single bit indicating that the mask is all zeroes.

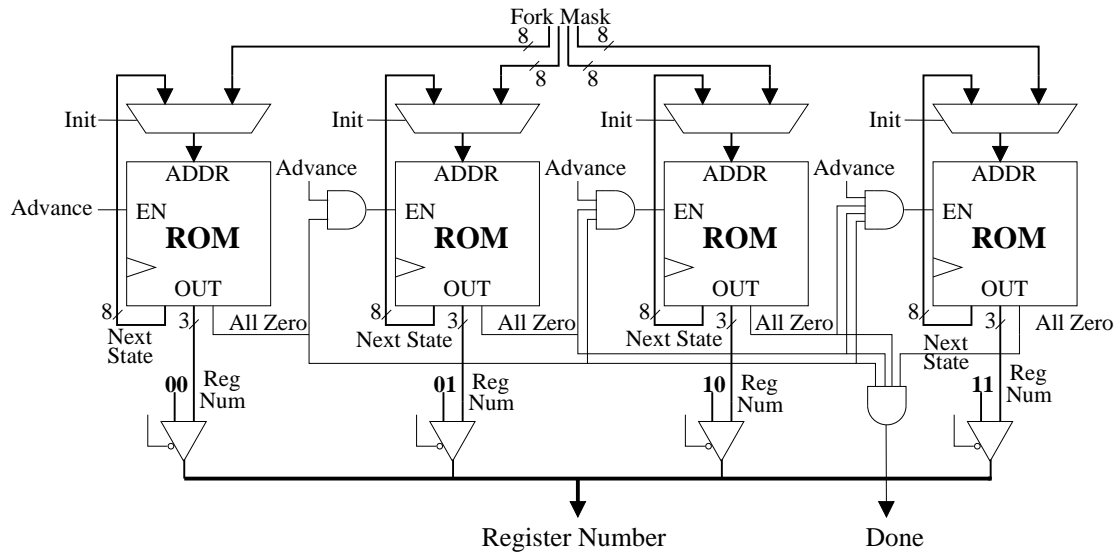


Figure 4-13: Fork Control Finite State Machine

The FSM is implemented with a ROM encoding the state information. Rather than having a single ROM with $2^3 \cdot 2$ entries, four ROMs are used as shown in Figure 4-13. The ROMs are Xilinx block RAMs, initialized to a particular pattern of bits. A simple C program creates these patterns. One useful feature of these synchronous memory devices is the enable input. If the enable is low when the rising clock edge arrives, the ROM will not perform a read operation; instead its output pins will hold the previous value. This provides a way of holding state without needed to use any of the FPGA's logic resources. The enables are wired up so that each ROM will only transition to the next state if the **Advance** signal is asserted and all of the lower-order ROMs are in the state with no bits set. This allows the clearing of bits to proceed from least significant bit to most significant bit, as desired. The **Advance** signal is asserted when the memory output logic accepts this fork data memory request; the fork FSM will then produce control signals to drive the next chosen GPR onto the context state bus.

Cstore

When swapping-out a context, the same amount of data is always transferred. The GPRs, SPRs, predicates, busy bits, memory request buffers, and memory request

table are transferred out to memory. Table 4.5 shows the data that makes up each of the 55 memory requests in a `cstore` operation. Once again, an FSM is used to generate the control signals. The state is the index into the context state table. Although the next state is always one more than the previous, it is a better use of FPGA resources to put that data into a ROM rather than employing a counter. The FSM state fits neatly into a single Xilinx block RAM device; as in the fork FSM, the ROM is also able to store the current state through judicious use of the enable pin. Very little external logic is required. The ROM outputs the next state, four control lines, a resource number, a “done” signal, as shown in Figure 4-14.

Index	Description
0	Request buffer 0, low word
1	Request buffer 0, middle word
2	Request buffer 0, high word
3	Request buffer 1, low word
4	Request buffer 1, middle word
5	Request buffer 1, high word
6	Request table entry 0
7	Request table entry 1
...	
13	Request table entry 7
14	General-Purpose Register 0
15	General-Purpose Register 1
...	
45	General-Purpose Register 31
46	Predicates
47	<code>tv</code> and <code>tp</code>
48	<code>tr</code> and <code>bp</code>
49	<code>e2</code> , <code>e1</code> , and <code>e0</code>
50	<code>e3</code>
51	<code>e4</code>
52	<code>e6</code> and <code>e5</code>
53	GPR busy bits
54	Busy bits for: <code>tr</code> , <code>tv</code> , <code>bp</code> , and <code>pr</code>

Table 4.5: Context State

The control lines are a one-hot encoding of which of the GPRs, SPRs, request

buffers, or request table is selected. The resource number is used to indicate which GPR, SPR, request buffer, or memory request table entry is being targeted. For the memory request buffers, the high bits indicate which third of the buffer is selected, as the entire buffer is too wide to fit into a single data word. One of the unused bits in the third data word is used to indicate if the entry is valid or not; this indicates whether or not the request should be enqueued on the request buffer FIFO when the data is loaded again later.

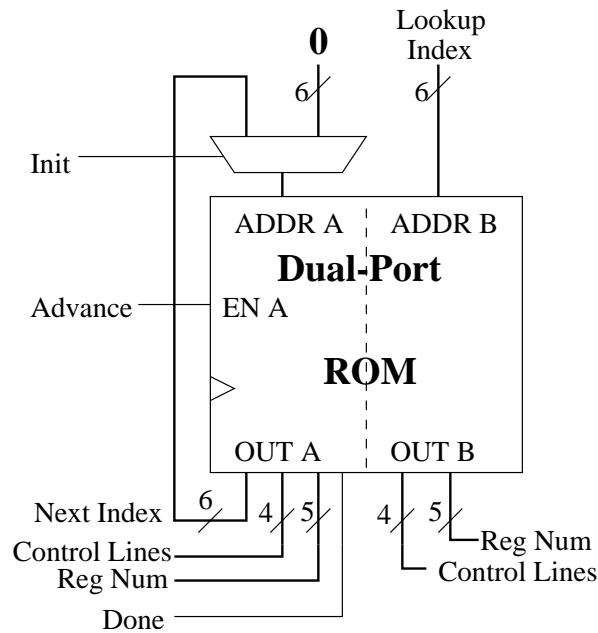


Figure 4-14: Context Store FSM and Context Load Look-up ROM

Clload

When a `cload` operation is executed, the Node controller will handle requesting the individual data words from memory. They arrive at the processor's data input with the Context bit set in the return address. This means that the destination register in the return address is an index into the Context State Table. The processor hands this index to the Context Control Unit for translation. The CCU looks up the index in a ROM and produces the corresponding resource number and control lines. This is identical to the data stored in the `cstore` FSM ROM. Since Xilinx block RAMs are dual-ported, it is possible for the CCU to lookup indexes in the same physical

memory device that is driving the finite state machine for controlling `cstore`. This is shown in Figure 4-14. The resource number and control lines produced by the lookup are used to send data into the request buffers and table via a dedicated data path, or to send data to the GPRs or SPRs on the normal Memory Reply Data write-back bus. The busy bits are treated as SPRs for this purpose, even though it is not possible to use them as SPRs in normal operation.

4.2.12 Global Events

Global events compose the class of events that require special attention from the operating system to be handled correctly. These events come from three different sources. First, the node controller may submit events to the processor due to exceptional memory or network conditions. Second, each trace controller generates a global event when its thread executes a `suspend` or `break` instruction. Finally, if the processor receives a memory reply for a context that has been swapped-out, the `EV_REPLY` event is generated.

Events go into an event queue when they occur. However, only one event may be admitted to the queue per clock cycle, so some arbitration is necessary. Any time a trace controller wants to submit an event but one of the other sources also has an event, the trace controller simply waits. If the node controller submits an event at the same time that a memory-reply event is generated, then the data for the memory-reply event is clocked into a set of registers and stored for a cycle. The event from the node controller goes to the event queue. The processor also asserts its “Event busy” output, so that the node controller will not submit an event next cycle. The following cycle, the buffered memory-reply event is sent the event queue. If a second memory-reply event happens at the same time, then it is stored into the registers that the first memory-reply event just vacated.

There is some in the Hamal architecture about overrunning the event queue. The high-water mark is used to throttle event inputs when the event queue is in danger of filling up. However, page-in events, which are generated which the node controller has finished paging-in memory on behalf of the processor, cannot be suppressed.

The operating system needs to receive these acknowledgements so that it can then take event-handling actions that were waiting on the page-in, thus making forward progress. Out of fear of swamping the event queue with page-in requests, two separate queues are actually used. One queue is for page-in events, and the other is for everything else.

The implementation uses the Xilinx FIFO core. This core uses block RAMs to store the queue’s data, which has an interesting side-effect. A queue of depth 16 uses the same amount of block RAM resources as a queue of depth 256. This is due to the way the static RAMs are constructed. There is a very marginal increase in logic resources associated with this depth increase, as shown by the statistics for the normal event queue in Table 4.6. This detail is pointed out to stress the “use-it-or-lose-it” mentality that seems to be quite applicable to FPGA designs, at least with the Xilinx Virtex-E series.

Depth	LUTs	Flip-flops	Block RAMs	Other
16	36	14	12	29
32	40	17	12	35
64	46	20	12	43
128	50	23	12	49
256	56	26	12	54
512	60	29	23	63

Table 4.6: Resource use for a 182-bit wide FIFO

4.2.13 Moving signals on and off the chip

A large amount of pin resources is needed to move data out of and into the processor. At a minimum, 931 pins are required. This does not include bringing the processor’s node ID number in, (thus requiring it be hard-coded), nor does it allow the export of extra debugging information that might be useful in a test environment. The FPGA package being targeted has only 512 I/O pins available. Time-multiplexing of the wires is used to overcome this limitation. For some of the data buses connected to the processor, the system clock cycle is divided in half. During the first phase, the

wires are used for the transport of some signals, and during the second phase, other signals get a turn. The particular break-down used is shown in Table 4.7. 422 pins are used in this manner. The other 59 signals have dedicated pins. Only 481 pins are used as a result of this process, with 31 pins left over. A more aggressive split is certainly possible to free-up more pins if this were desired. All of the pins are unidirectional despite having multiple uses. This reduces complexity and eliminates turn-around delays on the shared busses.

Phase	Data	Direction	Width
1	First half of new instruction group	Input	97
1	Data input, except valid bit	Input	172
1	Data output, except data word and valid bit	Output	143
2	Second half of new instruction group	Input	97
2	Event input, except valid bit	Input	182
2	Data word for data output	Output	129

Table 4.7: Time-multiplexing the FPGA I/O pins

4.3 Re-targeting Guidelines

Most of the source code for the processor implementation is a subset of Verilog which should be synthesizable by any Verilog synthesis tool. However, there are some pieces which are specific to the Virtex-E XCV2000E FPGA and would require some modification if a different FPGA was used, or if an ASIC was process was being targeted. This section enumerates the problems that might be encountered and suggests solutions.

4.3.1 Multiplexors and Tri-State Bus Drivers

A single source code file contains module definitions for several multiplexors and tri-state bus drivers. Some of these modules are functional models produced by the Core Generator for XCV2000E-specific modules; these models are not synthesizable. These would need to be replaced with alternate implementations for other synthesis targets.

The other modules that appear in that file are wrappers around the Xilinx modules. For example, `mux8x3` is an three-input eight-bit wide multiplexor. It is implemented by instantiating some of the XCV2000E-specific modules.

4.3.2 Clock Generation and Distribution

Xilinx-specific constructs are used once again in generating and distributing two low-skew clocks based off of an input reference clock. One clock is twice the frequency of the other, with negligible phase shift. Using a synthesis target other than the Xilinx Virtex-E series of FPGAs may require different technology-dependent implementations of these features. Specifically, the features are a delay-locked loop with frequency doubling and global buffered wires suitable for low-skew clock distribution. The Xilinx Application Note describing how the Virtex-E Delay-Locked Loop works is in [36].

4.3.3 Input/Output Drivers

Virtex-E devices support a myriad of I/O standards. Using any of them requires device-specific features, of course. Therefore, a target other than the Virtex-E FPGAs will require some effort on the I/O drivers and pads. The signalling used, simple 1.8V low-voltage CMOS, is not particularly complex. It is also likely that constraints placed on the system as a whole, such a board design and clock rate, will dictate much of the work that needs to be done in this area.

4.3.4 General-Purpose Register File

The multi-context general-purpose register file will require modification if targeted at a different line of FPGAs or at an ASIC process, if only for better efficiency. As described earlier, the Virtex-E primitive for memory is a dual-ported static RAM device. The register file, as specified by the Hamal architecture, needs to support at least four contexts and be equipped with six read ports and three write ports. If a more powerful RAM primitive is available on the new device, redesigning the way

the register file is constructed may be advisable. For an ASIC process, one would definitely want to design a reasonable register file cell and replicate it. There would be no need for six copies to accommodate porting. The Hamal architecture dictates that five of the read ports will always be using the same context, as they are used for operand fetch during the issue pipeline stage. A good approach would be to construct a register file cell that contained all four storage elements and had the small amount of decode logic to select one right in the cell. As shown in [17], the register file layout will be completely dominated by wires, and there will be ample area to place the necessary logic beneath these wires. Also, there may not be a need to split the processor clock cycle into a read and a write phase, so the generation and distribution of the second clock could be omitted.

It should also be possible to construct the cell such that a “snapshot” of one context’s state could be taken [20]. This would allow forks and cstores to truly operate in the background and not tie up a context. Once the snapshot is taken, the context can continue execution, in the case of a `fork`, or start receiving a swapping-in context, in the case of a `cstore`. Currently, executing a `fork` requires the issuing thread to wait until the fork is complete before going to the next instruction. A `cload` cannot be started on a context until it is finished swapping out. Although the swap-in and swap-out operation consist of the same number of memory requests, starting the swap-out first is not sufficient. The swap-in may overtake the swap-out, in terms of progress through the context’s state, if the processor’s data output is particularly swamped with requests.

Chapter 5

Beyond the Processor

This chapter discusses what components, in addition to the processor implementation presented in this thesis, are necessary to fabricate a prototype of a complete node of the Hamal parallel computer. While the processor is the most complex component, the node is not complete without the node controller, data and instruction memory, and the network interface. Finally, multiple nodes combined with a network and I/O devices would allow an implementation of an actual Hamal machine. This chapter makes recommendations on how best to accomplish this in a timely manner.

5.1 Current Status

In parallel with this work, Robert Woods-Corwin has been implementing a high-speed fault-tolerant interconnect fabric, described in [35], which will make an excellent network for the Hamal Parallel Computer. Additionally, Brian Ginsburg and Andrew Huang are developing boards which will allow the implementation of an ARIES network node [13], as described in the second chapter. The board uses FPGAs to implement the switching logic.

The ARIES group plans to create a processor board as well, to support both Hamal and other architectures which are being explored. This board is planned to incorporate two Xilinx Virtex-E FPGAs, an Intel StrongARM microprocessor, and a healthy amount of memory. The FPGAs will be an XCV2000E, 680-pin version, and

an XCV1000E, 1156-pin version. This second FPGA has fewer logic resources than the XCV2000E, but it makes up for it in I/O pin resources.

5.2 The Next Step

For Hamal, the smaller FPGA would make an excellent Node Controller, and Network Interface with its abundance of I/O pins. The larger FPGA will house the processor implementation presented here. This board is in turn paired with a network board to produce one Hamal node.

Memory, in the form of DRAM for the bulk and SRAM for the hardware page tables, is also necessary. It would probably be most convenient to implement the smart atomic memory operations and LRU-tracking in the same FPGA as the node controller, and use traditional commercially-available 'dumb' memory. As the entire system itself is only running at 25MHz, the fact that the FPGA can access the memory at a much higher rate (on the order of 200MHz) should make it simple to do a read, compute, and write in a single cycle to implement the atomic operations.

The matter of I/O devices has been largely ignored. It would be reasonable to imagine a Hamal node augmented with an I/O subsystem, accessed through the network interface. It might be wise to add another board to perform the interface logic, implementing perhaps the PCI bus standard for generic devices, like video cards and disk controllers. Or, perhaps, the StrongARM could be used as a gateway to such devices. To provide a uniform output path from the processor and through the node controller, the network interface would need to identify requests as heading for a local device. An added benefit is that remote nodes could access these local devices without the participation of the processor.

Another interesting consideration is that of boot-strapping a multi-node system. At boot, each node will attempt to start executing at address 0, using a code capability with all permissions and access to the entire address space. Since the instruction memory is empty, a page fault is caused. Since the O/S has not been loaded, there would be no way to process this request and ask the node controller to get some data.

A solution is to have the instruction memory, on boot, be set to map the beginning of the virtual address space to a programmable read-only-memory device (ROM). This ROM would contain code for the operating system loader— a simple program to go out to disk at a certain location and start reading bits into memory, page them into the instruction memory, and transfer execution to what was just read in. Alternatively, the ROM could contain enough of an operating system and test program to perform whatever testing and data collection is desired.

Chapter 6

Verification

The processor implementation is not useful unless it can be deduced that it is likely to be correct. Thus, a large effort was made to test the implementation, both as it was being developed and after it was completed. This chapter documents some of the verification methods which were employed to that end. These methods revolve around having scripts which the Verilog simulator can execute to stimulate the inputs of the device under test and then allow for the verification of correct behavior on the outputs. The scripts were invaluable for testing the initial implementations of each module, as well as for performing regression testing when small changes were made late in the implementation process.

6.1 Testing environment

The product used to perform verification of the implementation is ModelSim, developed by Model Technology, Inc. This tool includes a Verilog simulator which was used to perform functional verification. Additionally, the Xilinx implementation tools are capable of outputting timing information that ModelSim can import. Known as timing back-annotation, this allows ModelSim to factor time into its simulation. Without such information, that is, when doing a strictly functional simulation, the simulator assumes zero propagation delay in combinational logic.

Additional timing analysis was provided by Xilinx's Timing Analyzer. This tool

can determine the maximum clock rate that can be used and still avoid register setup and hold violations, as it computes the delays of all paths in the design. It can also be used to help locate critical paths and modify the design to achieve a higher clock rate, although it was not used for this purpose, as correctness was a larger concern, and the clock rate was high enough for what was required. Future implementations will likely be more speed-conscious; this implementation's purpose was to create a platform for evaluating the architecture at a speed greater than a software-only solution.

The tools provide excellent feedback. Since the entire design is collapsed into a single net list, static analysis of all nets, their drivers, and their loads is possible. For verification purposes, it is important to carefully inspect the tool's report on undriven and unloaded nets. In some circumstances, this may point to a bug in the design. If it is not clear on examination, then a test case should be designed to test if the behavior that was to be exhibited the supposedly excess logic actually is.

6.2 Module testing

As each module was completed, it was subjected to testing. ModelSim's scripting environment uses the Tool Command Language (TCL) with procedures added to support interacting with the simulator in the same way that the graphical user interface allows. Thus, a short TCL script was crafted for each module. The script drives inputs with test values designed to test the functionality of the module and then runs the clock. This process is repeated for several test sets, typically, to verify various modes of operation. Some tests were chosen on the basis of the specified behavior, (black-box testing), while others were selected to exercise a potentially incorrect corner case in the logic of the module, (glass-box testing). Some scripts required that the author inspect the results on the module's outputs to check for correctness, while scripts for more complex modules performed the comparison automatically.

6.2.1 Testing the Register File

One particularly interesting example of module-level verification is the work that went into verifying the general-purpose register file. First, it is important to understand some of the complexity in the implementation. There are six read ports, including the one used to swap out a context, which translates to 6 copies of each context's data. Additionally, each 129-bit register is broken into five parts. It is also possible to read and write at different granularities: 32, 64, or 129 bits at a time. Finally, writes to a 32-bit or 64-bit portion of a register must clear the pointer bit for that register.

To test this, the author wrote a program in the C programming language. The program simulates how the register file should work. It generates a series of random reads and writes to some number of ports each simulated clock cycle. The writes do not overlap each other and no portion of a register is read if it is being written to that cycle. The contexts used are generated randomly. The random seed is taken as an argument on the command line, so that the same scenario can be played back multiple times. This was very important; if a particular pattern of reads and writes illustrated a problem, it was important to be able to reproduce the pattern so that once the bug was believed fixed, the same test case could be attempted again.

The program produced as output a TCL script suitable for forcing values onto the input nets, running the clock for a cycle, and then verifying that the correct data appeared on the output ports. This script was called by a short TCL script which started the simulator and loaded the waveform viewer with the relevant signals. This second script was invoked from the ModelSim command line interface. Results from the test in progress were displayed as the script executed, and the waveform viewer would show the status of the selected signals.

This approach was useful. Several bugs were caught by this method, although the author estimates that more than half of the mismatches between the C simulation and the Verilog simulation were the fault of bugs in the C program. More specifically, it was usually a case of not observing the input assumptions made by the register

file, such as avoiding overlapping writes in a single cycle. While this approach did not guarantee that the module is correct, it provided high-quality assurances that it probably was.

6.3 Higher-level Testing

In addition to module-level tests, it was very useful to apply the same principles to groups of related modules with a common parent in the module hierarchy. For example, testing the entire ALU or Execution Unit as a whole, even after being assured that the smaller modules they instantiate work correctly. Taking that to its logical conclusion, the need for complete end-to-end testing of the entire processor should be evident.

6.3.1 An Assembler

It is useful to be able write short code sequences in the assembly language of the Hamal architecture, rather than trying to manually produce the correct stimulus bit patterns. While the latter approach was used for early testing of very specific behavior of the processor, it is cumbersome for more general testing. The tool needed to fix this is an assembler. The assembler translates a file of Hamal assembly language instructions into the bit patterns that would be presented to the processor by the instruction memory after decode.

6.3.2 Simulating Instruction Memory

The assembler is a good start, but to properly simulate the instruction memory, which is the goal for correctly feeding the processor, it is necessary to track multiple threads and to present the correct instruction group when requests are made. This could be hard-coded, if information about when the processor will request what can be predicted. That would involve a fair amount of error-prone work. What is really needed is a TCL script which checks the address being requested and then drives the

inputs with the corresponding data.

The implemented script simulates an instruction memory with single-cycle access; additional latency could be inserted easily if that was desired. This program is good for checking the arithmetic unit and the first stage of the memory pipeline. It can also test to see if requests are passed on to the node controller as they should be. To initialize the TCL representation of instruction memory for this script, a short C program was written. Given files output from the assembler, the program generates a TCL script which stores all of the instructions in a TCL list.

6.3.3 Simulating the Node Controller

The absence of the node controller is more difficult to deal with than the lack of instruction memory. One approach is to implement a subset of the node controller, in Verilog, and then be left with simulating the data memory and network interface. Another, actually implemented, is to take advantage of the fact that memory access may take an indeterminant number of cycles, so the precise timing of when responses come back to the processor is not important. The program that generates instruction memory scripts was modified to cause the resulting scripts to print out the memory requests that the node controller was accepting from the processor each cycle. Also, the scripts would pause execution from time to time. This allowed the use of the ModelSim command-line interface to force a particular reply into the processor on the data input, and then continue the script.

6.4 Limits

Ideally, there would be a second implementation of the processor to check the results of the first implementation on a cycle-by-cycle basis, as was done for the register file. While it is not feasible for the author to develop two complete Hamal implementations for this thesis, Grossman's work on a software simulator for Hamal, named Sim [16], provides exactly this. Sim, a work in progress at the time of this writing, is a cycle-accurate simulator for a 32-node Hamal machine. Sim's source code has

been instructive when trying to implement a particularly tricky or particularly undocumented feature of the architecture. This comparison has helped find bugs and shortcomings in both implementations. The FPGA implementation runs faster, as expected, but Sim provides better access to the internal state of the machine, with its full-featured graphical user interface.

Time and logistical constraints did not allow for completely integrating Sim and the Verilog simulator. While this would involve a large investment of time and energy, this would certainly be an interesting approach, as Sim could simulate the components external to the processor, including remote nodes, and it could also verify that the Verilog hardware description and the C++ software description in Sim agree with each other. One potential hurdle is that the comparison may be sensitive to implementation differences that do not affect the correctness of the processor in terms of the architecture specification.

Chapter 7

Conclusion

The processor successfully implements a subset of the Hamal architecture. This work concludes with a look at implementation results and potential directions for future work.

7.1 Results

Silicon efficiency is a major goal for project ARIES. While the resource consumption in an FPGA does not give a one-to-one correspondence to silicon area that would be consumed in an ASIC process, it is nonetheless instructive. Table 7.1 shows the resource usage for the entire processor. Table 7.2 shows details of some of the top level Verilog modules.

As the table shows, the trace controllers consume a large amount of chip resources. Constraints on logic resources allowed only three contexts to be present on the chip, rather than four as originally planned. Additionally, the logic use numbers do not include a complete arithmetic logic unit. There was only enough room for the simpler functional units. No floating point unit and integer multipliers were included. If the most area-efficient Core Generator cores were used, a 64-bit and pair of 32-bit multipliers would consume an extra 1836 slices, (9.6%), according to data from the Xilinx tools.

In terms of raw clock speed, the processor can be clocked at a maximum of 27

Resource	Used	Total Available	Percentage
Slices	19,198	19,200	100%
Slice Registers	13,290	38,400	34%
4-input LUTs	32,644	38,400	85%
Bonded IOBs	473	512	92%
BUFTs	20,116	19,520	103%
Block RAMs	60	160	37%
Global clock routes	2	4	50%
Clock IOBs	1	4	25%
Delay-locked loops	1	8	12%

Table 7.1: FPGA Resource Use

Module	Slices	Slice Percentage
Issue and Trace Control	16,631	86%
Context cstore/fork Control	30	1%
Mpipe ex1	1,771	9%
Memory Interface	5,693	29%
Event handling	257	1%

Table 7.2: Slice Usage Detail

MHz, according to the Xilinx implementation tools. Additionally, the register file can be clocked at a maximum of 51 MHz. Thus, a clock rate of 20-25MHz is extremely reasonable. In order to time-multiplex the input and output pins of the FPGA as described in Chapter 4, some of the I/O paths need to be clocked at twice the processor speed, or 40-50MHz. This should not pose a problem for board design.

7.2 Lessons

In the process of creating this work, the author learned a few lessons which may guide future efforts to implement processors with FPGAs. The first is the complexity of the logic synthesis task. On a dual Pentium III 550MHz processor with 512MB of RAM, a typical synthesis and implementation run took five hours. This was using the “-fast” option on the synthesis tool. A more thorough synthesis, where greater optimization is attempted, took closer to twenty hours. The process was CPU-bound most of the time. Sadly, the tools are single-threaded, so the second processor was not used. There was just enough physical memory in the machine to avoid swapping. A recommendation to future efforts is to use the fastest machine possible with a gigabyte of memory. A dual-processor system with enough memory could even synthesize two designs in parallel.

The other lesson learned is the complexity of logic versus the flexibility of available FPGA resources. The processor, as specified, did not fit into the FPGA. There were not nearly enough logic resources. As realized when trying to implement the register file, some tasks require dedicated hardware to do in an area-efficient manner. The availability of static RAMs on the Virtex-E devices is a step in the right direction, but more dedicated resources are required. The reconfigurable logic in FPGAs is wonderful because it allows you to test a design at high speed without fabricating a chip, and because the design can be easily changed after implementation to fix bugs or make other adjustments. However, implementation of silicon to perform known-good logic and architectural features would create far higher silicon efficiency. Xilinx’s Virtex-2 line of FPGAs have additional “hard cores” in them: dedicated multiplier hardware

and even a small PowerPC processor. This idea could probably be taken further, as well. For Hamal, the 128-bit word size meant very large busses moving around the chip. Almost all the logic resources that were consumed were implementing multiplexers or tri-state drivers for these busses. Future implementors should understand this limitation, and consider using narrower datapaths, time-multiplexing the datapaths within the chip, or evaluating the Virtex-2 FPGAs.

7.3 Future Work

There are several ways in which this work can be expanded. One first step is to implement more of the Hamal instruction set architecture. This may require additional optimization of existing components of the implementation, or it may simply require an FPGA with additional resources. Targeting a larger FPGA on the same product line, Xilinx's Virtex-E line, should be trivial.

The next logical step is to construct the printed circuit board setup described in Chapter 5. The only way that the simulation can finally run at full speed is to be actually running in the physical FPGA. For purposes of data collection, the implementation should be augmented to export interesting performance statistics to the data collection output on the processor board.

Finally, with the system constructed, large-scale simulations and testing can take place. This should bring about a cycle of data collection and modifications to try possible architectural variations. This is the key to collecting enough data on Hamal to consider a next generation hardware implementation, either another FPGA prototyping platform crafted with lessons learned from this one, a combination of FPGAs and ASICs (for the smart memory, for example), or even an all-ASIC or full-custom VLSI implementation.

Bibliography

- [1] D. A. Abarmson and J. Rosenberg. The micro-architecture of a capability-based computer. In *Proceedings of the 19th Annual Workshop on Microprogramming*, pages 138–145, 1986.
- [2] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6, 1990.
- [3] M. Anderson, R. D. Pose, and C. S. Wallace. A password-capability system. *The Computer Journal*, 29(1):1–8, 1986.
- [4] Jeremy Brown. *Memory Management on a Massively Parallel Capability Architecture*. PhD thesis, Massachusetts Institute of Technology, December 1999. Thesis Proposal.
- [5] Jeremy Brown, J. P. Grossman, Andrew Huang, and Tom Knight. A capability representation with embedded address and nearly-exact object bounds. Aries Technical Memo 5, MIT Artificial Intelligence Laboratory, April 2000. <http://www.ai.mit.edu/projects/aries/>.
- [6] Nicholas P. Carterm, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 319–327, 1994.

- [7] Andre DeHon. Fat-tree routing for Transit. AI Technical Report 1224, MIT Artificial Intelligence Laboratory, April 1990.
- [8] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.
- [9] Susan R. Dickey and Richard Kenner. Combining switches for the NYU Ultra-computer. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, October 1992.
- [10] Eran Egozy, Andre DeHon, Jr. Thomas Knight, Henry Minsky, and Samuel Peretz. Multipath Enhanced Transit Router Architecture. Transit Note 73, MIT Artificial Intelligence Laboratory, July 1992.
- [11] R. S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–412, July 1974.
- [12] Kouros Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory microprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.
- [13] Brian P. Ginsburg. Board specification for the initial Q architecture design and implementation. Aries Technical Memo 14, MIT Artificial Intelligence Laboratory, March 2001. <http://www.ai.mit.edu/projects/aries/>.
- [14] Peter N. Glaskowsky. MoSys explains 1T-SRAM technology. *Microprocessor Report*, 13(12), September 1999.
- [15] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in memory: The Terasys massively parallel PIM array. *Computer*, pages 24–31, April 1995.
- [16] J. P. Grossman. *Design and Evaluation of the Hamal Parallel Computer*. PhD thesis, Massachusetts Institute of Technology, April 2001. Thesis Proposal.

- [17] J. P. Grossman. Hamal design rationale. Aries Technical Memo 12, MIT Artificial Intelligence Laboratory, February 2001. <http://www.ai.mit.edu/projects/aries/>.
- [18] J. P. Grossman. Hamal instruction set architecture. Aries Technical Memo 11, MIT Artificial Intelligence Laboratory, January 2001. <http://www.ai.mit.edu/projects/aries/>.
- [19] J. P. Grossman. The Hamal processor-memory node. Aries Technical Memo 10, MIT Artificial Intelligence Laboratory, February 2001. <http://www.ai.mit.edu/projects/aries/>.
- [20] J. P. Grossman. Optimizing register file access during context switches. Personal communication, 2001.
- [21] J. P. Grossman, Jeremy Brown, Andrew Huang, and Tom Knight. A hardware mechanism to reduce the cost of forwarding pointer aliasing. Aries Technical Memo 8, MIT Artificial Intelligence Laboratory, September 2000. <http://www.ai.mit.edu/projects/aries/>.
- [22] J. P. Grossman, Jeremy Brown, Andrew Huang, and Tom Knight. Using SQUIDs to address forwarding pointer aliasing. Aries Technical Memo 4, MIT Artificial Intelligence Laboratory, April 2000. <http://www.ai.mit.edu/projects/aries/>.
- [23] Stephen A. Herrod. *Using Complete Machine Simulation to Understand Computer System Behavior*. PhD thesis, Stanford University, February 1998.
- [24] M. E. Houdek, F. G. Soltis, and R. L. Hoffman. IBM System/38 support for capability-based addressing. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 341–348, 1981.
- [25] Christoforos E. Kozyrakis et al. Scalable processors in the billion-transistor era: IRAM. *Computer*, pages 75–78, September 1997.
- [26] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH prototype: Logic overhead and

- performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, January 1993.
- [27] Chi-Keung Luk and Todd C. Mowry. Memory forwarding: Enabling aggressive layout optimizations by guaranteeing the safety of data relocation. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 88–99, 1999.
- [28] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996. <http://www.phrack.org/show.php?p=49&a=14>.
- [29] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active pages: A computation model for intelligent memory. In *1998 International Symposium on Computer Architecture*, 1998.
- [30] Dhiraj K. Pradhan. *Fault-Tolerant Computer System Design*, pages 126–144. Prentice Hall, 1996.
- [31] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete system simulation: The SimOS approach. *IEEE Parallel and Distributed Technology*, Fall 1995.
- [32] Jonathan Strauss Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, 1999.
- [33] Mac’s verilog mode for emacs. <http://www.verilog.com/verilog-mode.html>.
- [34] Virtex series FPGAs. http://www.xilinx.com/xlnx/xil_prodcatalog_landingpage.jsp?title=Virtex_Series.
- [35] Robert Woods-Corwin. A high-speed fault-tolerant interconnect fabric for large-scale multiprocessing. Master’s thesis, Massachusetts Institute of Technology, May 2001.
- [36] Xilinx, Inc. *Using the Virtex Delay-Locked Loop*, v2.3 edition, September 2000. Application Note 132: Virtex Series.

- [37] Xilinx, Inc. *Virtex-E 1.8V Field Programmable Gate Arrays Architectural Description*, v2.1 edition, April 2001. <http://www.xilinx.com/partinfo/ds022.htm>.