Design Decisions in the Implementation of a Raw Architecture Workstation

by

Michael Bedford Taylor

A.B., Dartmouth College 1996

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

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1999

Signature of Author ....
Department of Electrical Engineering and Computer Science
September 9, 1999

Certified by ....
Anant Agarwal
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

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

Design Decisions in the Implementation of a Raw Architecture Workstation

by
Michael Bedford Taylor

Submitted to the
Department of Electrical Engineering and Computer Science
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 9, 1999

in partial fulfillment of the requirements for the degree of
Master of Science in Electrical Engineering and Computer Science.

Abstract

In this thesis, I trace the design decisions that we have made along the journey to creating the first Raw architecture prototype. I describe the emergence of extroverted computing, and the consequences of the billion transistor era. I detail how the architecture was born from our experience with FPGA computing. I familiarize the reader with Raw by summarizing the programmer's viewpoint of the current design. I motivate our decision to build a prototype. I explain the design decisions we made in the implementation of the static and dynamic networks, the tile processor, the switch processor, and the prototype systems. I finalize by showing some results that were generated by our compiler and run on our simulator.

Thesis Supervisor: Anant Agarwal
Title: Professor, Laboratory for Computer Science

Dedication

This thesis is dedicated to my mom.

-- Michael Bedford Taylor, 9-9-1999

TABLE OF CONTENTS

1 INTRODUCTION

1.0 MANIFEST

In the introduction of this thesis, I start by motivating the Raw architecture discipline, from a computer architect's viewpoint.

I then discuss the goals of the Raw prototype processor, a research implementation of the Raw philosophy. I elaborate on the research questions that the Raw group is trying to answer.

In the body of the thesis, I will discuss some of the important design decisions in the development of the Raw prototype, and their effects on the overall development.

Finally, I will conclude with some experimental numbers which show the performance of the Raw prototype on a variety of compiled and hand-coded programs. Since the prototype is not available at the time of this thesis, the numbers will come from a simulation which matches the synthesizeable RTL verilog model on a cycle by cycle basis.

1.1 MOTIVATION FOR A NEW TYPE OF PROCESSOR

1.1.1 The sign of the times

The first microprocessor builders designed in a period of famine. Silicon area on die was so small in the early seventies that the great challenge was just in achieving important features like reasonable data and address widths, virtual memory, and support for external I/O.

A decade later, advances in material science provided designers with enough resources that silicon was neither precious nor disposable. It was a period of moderation. Architects looked to advanced, more space consuming techniques like pipelining, out-of-order issue, and caching to provide performance competitive with minicomputers. Most of these techniques were borrowed from supercomputers, and were carefully added from generation to generation as more resources became available.

The next decade brings with it a regime of excess. We will have billions of transistors at our disposal. The new challenge of modern microprocessor architects is very simple: we need to provide the user with an effective interface to the underlying raw computational resources.

1.1.2 An old problem: SpecInt

In this new era, we could continue on as if we still lived in the moderation phase of microprocessor development. We would incrementally add micro-architectural mechanisms to our superscalar and VLIW processors, one by one, carefully measuring the benefits.

For today's programs, epitomized by the SpecInt95 benchmark suite, this is almost certain to provide us with the best performance. Unfortunately, this approach suffers from exponentially growing complexity (measured by development and testing costs and man-years) that is not being sufficiently mitigated by our sophisticated design tools, or by the incredible expertise that we have developed in building these sorts of processors. Unfortunately, this area of research is at a point where increasing effort and increasing area is yielding diminishing returns [Hennessey99].

Instead, we can attack a more fuzzy, less defined goal. We can use the extra resources to expand the scope of problems that microprocessors are skilled at solving. In effect, we redirect our attention from making processors better at solving problems they are already, frankly, quite good at, towards making them better at application domains which they currently are not so good at.

In the meantime, we can continue to rely on the as-yet juggernaut march of the fabrication industry to give us a steady clock speed improvement that will allow our existing SpecInt applications to run faster than ever.

1.1.3 A new problem: Extroverted computing

Computers started out as very oblivious, introverted devices. They sat in air-conditioned rooms, isolated from their users and the environment. Although they communicated with EACH OTHER at high speeds, the bandwidth of their interactions with the real world was amazingly low. The primary input devices, keyboards, provided at most tens of characters per second. The output bandwidth was similarly pathetic.

With the advent of video display and sound synthesis, the output bandwidth to the real world has blossomed to 10s of megabytes per second. Soon, with the advent of audio and video processing, the input bandwidth will match similar levels.

As a result of this, computers are going to become more and more aware of their environments. Given sufficient processing and I/O resources, they will not only become passive recorders and childlike observers of the environment, they will be active participants. In short, computers will turn from recluse introverts to extroverts.

The dawn of the extroverted computing age is upon us. Microprocessors are just getting to the point where they can handle real-time data streams coming in from and out to the real world. Software radios and cell phones can be programmed in a 1000 lines of C++ [Tennenhouse95]. Video games generate real-time video, currently with the help of hardware graphics back ends. Real-time video and speech understanding, searching, generation, encryption, and compression are on the horizon. What once was done with computers for text and integers will soon be done for analog signals. We will want to compose sound and video, search it, interpret it, and translate it.

Imagine, while in Moscow, you could talk to your wrist watch and tell it to listen to all radio stations for the latest news. It would simultaneously tune into the entire radio spectrum (whatever it happens to be in Russia), translate the speech into English, and index and compress any news on the U.S. At the same time, your contact lens display would overlay English translations of any Russian word visible in your sight, compressing and saving it so that you can later edit a video sequence for your kids to see (maybe you'll encrypt the cab ride through the red light district with DES-2048). All of these operations will require massive bandwidth and processing.

1.1.4 New problem, old processors?

We could run our new class of extroverted applications on our conventional processors. Unfortunately, these processors are, well, introverted.

First off, conventional processors often treat I/O processing as a second class citizen to memory processing. The I/O requests travel through a hierarchy of slower and slower memory paths, and end up being bottlenecked at the least common denominator. Most of the pins are dedicated to caches, which ironically, are intended to minimize communication with the outside world. These caches, which perform so well on conventional computations, perform poorly on streaming, extroverted, applications which have infinite data streams that are briefly processed and discarded.

Secondly, these new extroverted applications often have very plentiful fine grained parallelism. The conventional ILP architectures have complicated, non-scalable structures (multi-ported or rotating register files, speculation buffers, deferred exception mechanisms, pools of ALUs) that are designed to wrest small degrees of parallelism out of the most twisty code. The parallelism in these new applications does not require such sophistication. It can be exploited on architectures that are easy to design and are scalable to thousands of active functional units.

Finally, the energy efficiency of architectures needs to be considered to evaluate their suitability for these new application domains. The less power microprocessors need, the more and more environments they can exist in. Power requirements create a qualitative difference along the spectrum of processors. Think of the enormous difference among 1) machines that require large air conditioners, 2) ones that need to be plugged in, 3) ones that run on batteries, and ultimately, 4) ones that runs off their tiny green chlorophyllic plastic case.

1.1.5 New problems, new processors.

It is not unlikely that existing processors can be modified to have improved performance on these new applications. In fact, the industry has already made some small baby steps with the advent of the Altivec and MAX-2 technologies [Lee96].

The Raw project is creating an extroverted architecture from scratch. We take as our target these data-intensive extroverted applications. Our architecture is extremely simple. Its goal is to expose as much of the copious silicon and pin resources to these applications. The Raw architecture provides a raw, scalable, parallel interface which allows the application to make direct use of every square millimeter of silicon and every I/O pin. The I/O mechanism allows data to be streamed directly in and out of the chip at extraordinary rates.

The Raw architecture discipline also has advantages for energy efficiency. However, they will not be discussed in this thesis.

1.2 MY THESIS AND HOW IT RELATES TO RAW

My thesis details the decisions and ideas that have shaped the development of a prototype of the new type of processor that our group has developed. This process has been the result of the efforts of many talented people. When I started at MIT three years ago, the Raw project was just beginning. As a result, I have the luxury of having a perspective on the progression of ideas through the group. Initially, I participated in much of the data gathering that refined our initial ideas. As time passed on, I became more and more involved in the development of the architecture. I managed the two simulators, hand-coded a number of applications, worked on some compiler parallelization algorithms, and eventually joined the hardware project. I cannot claim to have originated all of the ideas in this thesis; however I can reasonably say that my interpretation of the sequence of events and decisions which lead us to this design point probably is uniquely mine. Also uniquely mine probably is my particular view of what Raw should look like.

Anant Agarwal and Saman Amarasinghe are my fearless leaders. Not enough credit goes out to Jonathan Babb, and Matthew Frank, whose brainstorming planted the first seeds of the Raw project, and who have continued to be a valuable resource. Jason Kim is my partner in crime in heading up the Raw hardware effort. Jason Miller researched I/O interfacing issues, and is designing the Raw handheld board. Mark Stephenson, Andras Moritz, and Ben Greenwald are developing the hardware/software memory system. Ben, our operating systems and tools guru also ported the GNU binutils to Raw. Albert Ma, Mark Stephenson, and Michael Zhang crafted the floating point unit. Sam Larsen wrote the static switch verilog. Rajeev Barua and Walter Lee created our sophisticated compiler technology. Elliot Waingold wrote the original simulator. John Redford and Chris Kappler lent their extensive industry experience to the hardware effort.

1.2.1 Thesis statement

The Raw Prototype Design is an effective design for a research implementation of a Raw architecture workstation.

1.2.2 The goals of the prototype

In the implementation of a research prototype, it is important early on to be excruciatingly clear about one’s goals. Over the course of the design, many implementation decisions will be made which will call into question these goals. Unfortunately, the “right” solution from a purely technical standpoint may not be the correct one for the research project. For example, the Raw prototype has a 32-bit architecture. In the commercial world, such a paltry address space is a guaranteed trainwreck in the era of gigabit DRAMs. However, in a research prototype, having a smaller word size gives us nearly twice as much area to further our research goals. The tough part is making sure that the implementation decisions do not invalidate the research's relevance to the real world.

Ultimately, the prototype must serve to facilitate the exploration and validation of the underlying research hypotheses.

The Raw project, underneath it all, is trying to answer two key research questions:

1.2.3 The Billion Transistor Question

What should the billion transistor processor of the year 2007 look like?

The Raw design philosophy argues for an array of replicated tiles, connected by a low latency, high throughput, pipelined network.

This design has three key implementation benefits, relative to existing superscalar and VLIW processors:

First, the wires are short. Wire length has become a growing concern in the VLSI community, now that it takes several cycles for a signal to cross the chip. This is not only because the transistors are shrinking, and die sizes are getting bigger, but because the wires are not scaling with the successive die shrinks, due to capacitive and resistive effects. The luxurious abstraction that the delay through a combinational circuit is merely the sum of its functional components no longer holds. As a result, the chip designer must now worry about both congestion AND timing when placing and routing a circuit. Raw's short wires make for an easy design.

Second, Raw is physically scalable. This means that all of the underlying hardware structures are scalable. All components in the chip are of constant size, and do not grow as the architecture is adapted to utilize larger and larger transistor budgets. Future generations of a Raw architecture merely use more tiles with out negatively impacting the cycle time. Although Raw offers scalable computing resources, this does not mean that we will necessarily have scalable performance. That is dependent on the particular application.

Finally, Raw has low design and verification complexity. Processor teams have become exponentially larger over time. Raw offers constant complexity, which does not grow with transistor budget. Unlike today's superscalars and VLIWs, Raw does not require a redesign in order to accommodate configurations with more or fewer processing resources. A Raw designer need only design the smaller region of a single tile, and replicate it across the entire die. The benefit is that the designer can concentrate all of one's resources on tweaking and testing a single tile, resulting in clock speeds higher than that of monolithic processors.

1.2.4 The "all-software hardware" question

What are the trade-offs of replacing conventional hardware structures with compilation and software technology?

Motivated by advances in circuit compilation technology, the Raw group has been actively exploring the idea of replacing hardware sophistication with compiler smarts. However, it is not enough merely to reproduce the functionality of the hardware. If that were the case, we would just prove that our computing fabric was Turing-general, and move on to the next research project. Instead our goal is more complex. For each alternative solution that we examine, we need to compare its area-efficiency, performance, and complexity to that of the equivalent hardware structure. Worse yet, these numbers need to be tempered by the application set which we are targeting.

In some cases, like in leveraging parallelism, removing the hardware structures allows us to better manage the underlying resources, and results in a performance win. In other cases, as with a floating point unit, the underlying hardware accelerates a basic function which would take many cycles in software. If the target application domain makes heavy use of floating point, it may not be possible to attain similar performance per unit area regardless of the degree of compiler smarts. On the other hand, if the application domain does not use floating point frequently, then the software approach allows the application to apply that silicon area to some other purpose.

1.3 SUMMARY

In this section, I have motivated the design of a new family of architectures, the Raw architectures. These architectures will provide an effective interface for the amazing transistor and pin budgets that will come in the next decade. The Raw architectures anticipate the arrival of a new era of extroverted computers. These extroverted computers will spend most of their time interacting with the local environment, and thus are optimized for processing and generating infinite, real-time data streams.

I continued by stating my thesis statement, that the Raw prototype design is an effective design for a research implementation of a Raw architecture workstation. I finished by explaining the central research questions of the Raw project.

2 EARLY DESIGN DECISIONS

2.0 THE BIRTH OF THE FIRST RAW ARCHITECTURE

2.0.1 RawLogic, the first Raw prototype

Raw evolved from FPGA architectures. When I arrived at MIT almost three years ago, Raw was very much in its infancy. Our original idea of the architecture was as a large box of reconfigurable gates, modeled after our million-gate reconfigurable emulation system. Our first major paper, the Raw benchmark suite, showed very positive results on the promise of configurable logic and hardware synthesis compilation. We achieved speedups on a number of benchmarks; numbers that were crazy and exciting [Babb97].

However, the results of the paper actually considerably matured our viewpoint. The term “reconfigurable logic” is really very misleading. It gives one the impression that silicon atoms are actually moving around inside the chip to create your logic structures. But the reality is, an FPGA is an interpreter in much the same way that a processor is. It has underlying programmable hardware, and it runs a software program that is interpreted by the hardware. However, it executes a very small number of very wide instructions. It might even be viewed as an architecture with a instruction set optimized for a particular application; the emulation of digital circuits. Realizing this, it is not surprising that our experiences with programming FPGA devices show that they are neither superior nor inferior to a processor. It is merely a question of which programs run better on which interpreter.

In retrospect, this conclusion is not all that surprising; we already know that FPGAs are better at logic emulation than processors; otherwise they would not exist. Conversely, it is not likely that the extra bit-level flexibility of the FPGA comes for free. And, in fact, it does not. 32-bit datapath operations like additions and multiplies perform much more quickly when optimized by an Intel circuit hacker on a full-custom VLSI process than when they are implemented on a FPGA substrate. And again, it is not much wonder, for the processor's multiplier has been realized directly in silicon, while the multiplier implementation on the FPGA is running under one level of interpretation.

2.0.2 Our Conclusions, based on Raw logic

In the end, we identified three major strengths of FPGA logic, relative to a microprocessor:

FPGAs make a simple, physically scalable parallel fabric.

For applications which have a lot of parallelism, we can easily exploit it by adding more and more fabric.

FPGAs allow for extremely fast communication and synchronization between parallel entities.

In the realm of shared memory multiprocessors, it takes tens to hundreds of cycles for parallel entities to communication and synchronize [Agarwal95]. When a silicon compiler compiles parallel verilog source to an FPGA substrate, the different modules can communicate on a cycle-by-cycle basis. The catch is that this communication often must be statically scheduled.

FPGAs are very effective at bit and byte-wide data manipulation.

Since FPGA logic functions operate on small bit quantities, and are designed for circuit emulation, they are very powerful bit-level processors.

We also identified three major strengths of processors relative to FPGAs:

Processors are highly optimized for datapath oriented computations.

Processors have been heavily pipelined and have custom circuits for datapath operations. This customization means that they process word-sized data much faster than an FPGAs.

Compilation times are measure in seconds, not hours [Babb97].

The current hardware compilation tools are very computationally intensive. In part, this is because the hardware compilation field has very different requirements from the software compilation field. A smaller, faster circuit is usually much more important than fast compilation. Additionally, the problem sizes of the FPGA compilers are much bigger -- a net list of NAND gates is much larger than a dataflow graph of a typical program. This is exacerbated by the fact that the synthesis tools decompose identical macro-operations like 32-bits adds into separately optimized netlists of bit-wise operations.

Processors are very effective for just getting through the millions of lines of code that AREN'T the inner loop.

The so-called 90-10 rule says that 90 percent of the time is spent in 10 percent of the program code. Processor caches are very effective at shuffling infrequently used data and code in and out of the processor when it is not needed. As a result, the non-critical program portions can be stored out to a cheaper portion of the memory hierarchy, and can be pulled in at a very rapid rate when needed. FPGAs, on the other hand, have a very small number (one to four) of extremely large, descriptive instructions stored in their instruction memories. These instructions describe operations on the bit level, so a 32-bit add on an FPGA takes many more instruction bits than the equivalent 32-bit processor instruction. It often takes an FPGA thousands or millions of cycles to load a new instruction in. A processor, on the other hand, can store a large number of narrow instruction in its instruction memory, and can load in new instructions in a small number of cycles. Ironically, the fastest way for an FPGA to execute reams of non-loop-intensive code is to build a processor in the FPGA substrate. However, with the extra layer of interpretation, the FPGA's performance will not be comparable to a processor built in the same VLSI process.

2.0.3 Our New Concept of a Raw Processor

Based on our conclusions, we arrived at a new model of the architecture, which is described in the September 1997 IEEE Computer "Billion Transistor" issue [Waingold97].

We started with the FPGA design, and added coarse grained functional units, to support datapath operations. We added word-wide data memories to keep frequently used data nearby. We left in some FPGA-like logic to support fine grained applications. We added pipelined sequencers around the functional units to support the reams of non-performance critical code, and to simplify compilation. We linked the sequenced functional units with a statically scheduled pipelined interconnect, to mimic the fast, custom interconnect of ASICs and FPGAs. Finally, we added a dynamic network to support dynamic events.

The end result: a mesh of replicated tiles, each containing a static switch, a dynamic switch, and a small pipelined processor. The tiles are all connected together through two types of high performance, pipelined networks: one static and one dynamic.

Now, two years later, we are on the cusp of building the first prototype of this new architecture.

3 WHAT WE'RE BUILDING

3.0 THE FIRST RAW ARCHITECTURE

In this section, I present a description of the architecture of the Raw prototype, as it currently stands, from an assembly language viewpoint. This will give the reader a more definite feel for exactly how all of the pieces fit together. In the subsequent chapters, I will discuss the progress of design decisions which made the architecture the way it is.

3.0.1 A mesh of identical tiles

A Raw processor is a chip containing a 2-D mesh of identical tiles. The tiles are connected to its nearest neighbors by the dynamic and static networks. To program the Raw processor, one programs each of the individual tiles. See the figure entitled "A Mesh of Identical Tiles."

3.0.2 The tile

Each tile has a tile processor, a static switch processor, and a dynamic router. In the rest of this document, the tile processor is usually referred to as "the main processor," "the processor," or "the tile processor." "The Raw processor" refers to the entire chip -- the networks and the tiles.

The tile processor uses a 32-bit MIPS instruction set, with some slight modifications. The instruction set is described in more detail in the "Raw User's Manual," which has been appended to the end of this thesis.

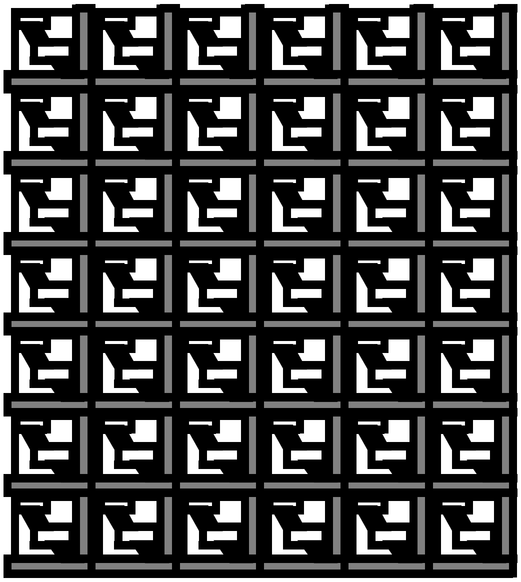
The switch processor (often referred to as "the switch") uses a MIPS-like instruction set that has been stripped down to contain just moves, branches, jumps and branches. Each instruction also has a ROUTE component, which specifies the transfer of values on the static network between that switch and its neighboring switches.

The dynamic router runs independently, and is under user control only indirectly.

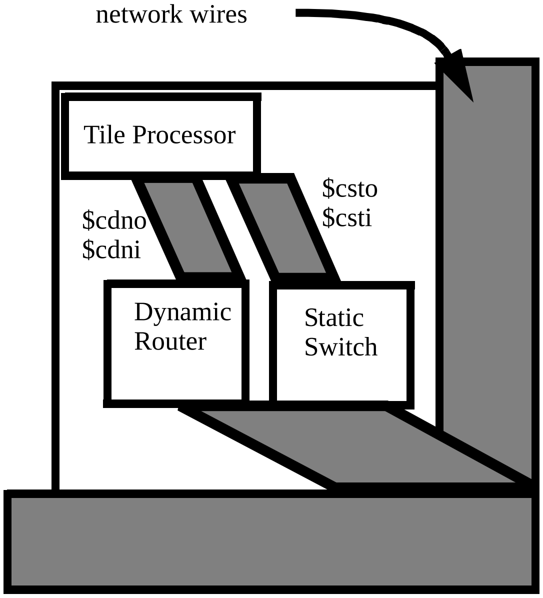
3.0.3 The tile processor

The tile processor has a 32 Kilobyte data memory, and a 32 Kilobyte instruction memory. Neither of them are cached. It is the compiler's responsibility to virtualize the memories in software, if this is necessary.

The tile processor communicates with the switch through two ports which have special register names, $csto and $csti. When a data value is written to $csto, it is actually sent to a small FIFO located in the switch.



A Mesh of Identical Tiles



Logical View of A Raw Tile

When a data value is read from $csti, it is actually read from a FIFO inside the switch. The value is removed from the FIFO when the read occurs.

If a read on $csti is specified, and there is no data available from that port, the processor will block. If a write to $csto occurs, and the buffer space has been filled, the processor will also block.

Here is some sample assembly language:

```
# XOR register 2 with 15,
# and put result in register 31

xori $31,$2,15

# get value from switch, add to
# register 3, and put result
# in register 9

addu $9,$3,$csti

# an ! indicates that the result
# of the operation should also
# be written to $csto

and! $0,$3,$2

# load from address at $csti+25
# put value in register 9 AND
# send it through $csto port
# to static switch

ld! $9,25($csti)

# jump through value specified
# by $csti

j $csti
nop           # delay slot
```

The dynamic network ports operate very similarly. The input port is $cdni, and the output port is $cdno. However, instead of showing up at the static switch, the messages are routed through the chip to their destination tile. This tile is specified by the first word that is written into $cdno. Each successive word will be queued up until a `dlaunch` instruction is executed. At that point, the message starts streaming through the dynamic network to the other tile. The next word that is written into $cdno will be interpreted as the destination for a new dynamic message.

```
# specify a send to tile #15

addiu $cdno,$0,15

# put in a couple of datawords,
# one from register 9 and the other
# from the csti network port

or $cdno,$0,$9
ld $cdno,$0,$csti

# launch the message into the
# network

dlaunch

# if we were tile 15, we could
# receive our message with:

# read first word
or $2,$cdni,$0

# read second word,
or $3,$cdni,$0

# the header word is discarded
# by the routing hardware, so
# the recipient does not see it
# there are only two words in
# this message
```

3.0.4 The switch processor

The switch processor has a local 8096-instruction instruction memory, but no data memory. This memory is also not cached, and must be virtualized in software by the switch's nearby tile processor.

The switch processor executes a very basic instruction set, which consists of only moves, branches, jumps, and nops. It has a small, four element register file. The destinations of all of the instructions must be registers. However, the sources can be network ports. The network port names for the switch processor are $csto, $csti, $cNi, $cEi, $cSi, $cWi, $cNo, $cEo, $cSo and $cWo. These correspond to the main processor's output queue, the main processor's input queue, the input queues coming from the switch's four neighbors, and the output queues going out to the switch's four neighbors.

Each switch processor instruction also has a ROUTE component, which is executed in parallel with the instruction component. If any of the ports specified in the instruction are full (for outputs) or empty (for inputs), the switch processor will stall.

```
   # branch instruction
   beqz $9, target
   nop

   # branch if processor
   # sends us a zero

   beqz $csto, target
   nop

   # branch if the value coming
   # from the west neighbor is a zero

   beqz $cWi, target
   nop

   # store away value from
   # east neighbor switch

   move $3, $cEi

   # same as above, but also route
   # the value coming from the north
   # port to the south port

   move $3, $cEi   route $cNi->$cSo

   # all at the same time:
   # send value from north neighbor
   # to both the south and processor
   # input ports.
   # send value from processor to west
   # neighbor.
   # send value from west neighbor to
   # east neighbor

   nop  route $cNi->$cSo, $cNi->$csti,
              $csto->$cWo,$cWi->$cEo

   #      jump to location specified
   # by west neighbor and route that
   # location to our east neighbor

   jr $cWi    route $cWi->$cEo
   nop
```

3.0.5 Putting it all together

For each switch-processor, processor-switch, or switch-switch link, the value arrives at the end of the cycle. The code below shows the switch and tile code required for a tile-to-tile send.

```
TILE 0:

   or $csto,$0,$5

SWITCH 0:

   nop  route $csto->$cEo

SWITCH 1:

   nop  route $cWi->$csti

TILE 1:

   and $5, $5, $csti
```

This code sequence takes five cycles to execute. In the first cycle, tile 0 executes the OR instruction, and the value arrives at switch 0. On the second cycle, switch 0 transmits the value to switch 1. On the third cycle, switch 1 transfers the value to the processor. On the fourth cycle, the value enters the decode stage of the processor. On the fifth cycle, the AND instruction is executed.

Since two of those cycles were spent performing useful computation, the send-to-use latency is three cycles.

More information on programming the Raw architecture can be found in the User’s Manual at the end of this thesis. More information on how our compiler parallelizes sequential applications for the Raw architecture can be found in [Lee98] and [Barua99].

3.1 RAW MATERIALS

Before we decided what we were going to build for the prototype, we needed to find out what resources we had available to us. Our first implementation decision, at the highest level, was to build the prototype as a standard-cell CMOS ASIC (application specific integrated circuit) rather than as full-custom VLSI chip.

In part, I believe that this decision reflects the fact that the group's strengths and interests center more on systems architecture than on circuit and micro-architectural design. If our research shows that our software systems can achieve speedups on our micro-architecturally unsophisticated ASIC prototype, it is a sure thing that the micro-architects and circuit designers will be able to carry the design and speedups even further.

3.1.1 The ASIC choice

When I originally began the project, I was not entirely clear on the difference between an ASIC and full-custom VLSI process. And indeed, there is a good reason for that; the term ASIC (application specific integrated circuit) is vacuous.

As perhaps is typical for someone with a liberal arts background, I think the best method of explaining the difference is by describing the experience of developing each type of chip.

In a full-custom design, the responsibility of every aspect of the chip lies on designer's shoulders. The designer starts with a blank slate of silicon, and specifies as an end result, the composition of every unit volume of the chip. The designer may make use of a pre-made collection of cells, but they also are likely to design their own. They must test these cells extensively to make sure that they obey all of the design rules of the process they are using.

These rules involve how close the oxide, poly, and metal layers can be to each other. When the design is finally completed, the designer holds their breath and hopes that the chip that comes back works.

In a standard-cell ASIC process, the designer (usually called the customer) has a library of components that have been designed by the ASIC factory. This library often includes RAMs, ROMs, NAND type primitives, PLLs, IO buffers, and sometimes datapath operators. The designer is not typically allowed to use any other components without a special dispensation. The designer is restricted from straying too far from edge triggered design, and there are upper bounds on the quantity of components that are used (like PLLs). The end product is a netlist of those components, and a floorplan of the larger modules. These are run through a variety of scripts supplied by the manufacturer which insert test structures, provide timing numbers and test for a large number of rule violations. At this point, the design is given to the ASIC manufacturer, who converts this netlist (mostly automatically) into the same form that the full-custom designer had to create.

If everything checks out, the ASIC people and the customer shake hands, and the chip returns a couple of months later. Because the designer has followed all of the rules, and the design has been checked for the violation of those rules, the ASIC manufacturer GAURANTEES that the chip will perform exactly as specified by the netlist.

In order to give this guarantee however, their libraries tend to be designed very conservatively, and cannot achieve the same performance as the full custom versions.

The key difference between an ASIC and full custom VLSI project is that the designer gives up degrees of flexibility and performance in order to attain the guarantee that their design will come back "first time right". Additionally, since much of the design is created automatically, it takes less time to create the chip.

3.1.2 IBM: Our ASIC foundry

Given the fact that we had decided to do an ASIC, we looked for an industry foundry. This is actually a relatively difficult feat. The majority of ASIC developers are not MIT researchers building processor prototypes. Many are integrating an embedded system onto one chip in order to minimize cost. Closer to our group in terms of performance requirements are the graphics chips designers and the network switch chip designers. They at least are quite concerned with pushing performance envelope. However, their volumes are measured in the hundreds of thousands, while the Raw group probably will be able to get by on just the initial 30 prototype chips that the ASIC manufacturer gives us. Since the ASIC foundry makes its money off of the volume of the chips produced, we do not make for great profits. Instead, we have to rely on the generosity of the vendor and on other, less tangible incentives to entice a partnership.

We were fortunate enough to be able to use IBM's extraordinary SA-27E ASIC process. It is IBM's latest ASIC process. It is considered to be a "value" process, which means that some of the parameters have been tweaked for density rather than speed. The "premium," higher speed version of SA-27E is called SA-27.

Please note that all of the information that I present about the process is available off of IBM's website (www.chips.ibm.com) and from their databooks. No proprietary information is revealed in this thesis.

Table 1: SA-27E Process

Param	Value
L_{eff}	.11 micron
L_{drawn}	.15 micron
Core Voltage	1.8 Volts
Metallization	6 layers, copper
Gates	Up to 24 Million 2-input NANDs, based on die size
Embedded Dram	SRAM MACRO 1 MBit = $8mm^2$ DRAM MACRO first 1 MBit = 3.4 mm^2 addt'l MBits = 1.16 mm^2 50 MHz random access
I/O	C4 Flip Chip Area I/O up to 1657 pins on CCGA (1124 signal I/Os) Signal technologies: SSTL, HSTL, GTL, LVTTL AGP, PCI...

The 24 million gates number assumes perfect wireability, which although we do have many layers of metal in the process, is unlikely. Classically, I have heard of wireability being quoted at around %35 - %60 for older non-IBM processes.

This means that between %65 and %40 of those gates are not realizable when it comes to wiring up the design. Fortunately, the wireability of RAM macros is at %100, and the Raw processor is mostly SRAM!

We were very pleasantly surprised by the IBM process, especially with the available gates, and the abundance of I/O. Also, later, we found that we were very impressed with the thoroughness of IBM's LSSD test methodology.

3.1.3 Back of the envelope: A 16 tile Raw chip

To start out conservatively, we started out with a die size which was roughly 16 million gates, and assume 16 Raw tiles. The smaller die size gives us some slack at the high end should we make any late discoveries or have any unpleasant realizations. This gave us roughly 1 million gates to allocate to each tile. Of that, we allocated half the area to memory. This amounts to roughly 32 kWords of SRAM, with 1/2 million gates left to dedicate to logic. Interestingly, the IBM process also allows us to integrate DRAM on the actual die. Using the embedded DRAM instead of the SRAM would have allowed us to pack about four times as much memory in the same space. However, we perceived two principal issues with using DRAM:

First, the 50 MHz random access rate would require that we add a significant amount of micro-architectural complexity to attain good performance. Second, embedded DRAM is a new feature in the IBM ASIC flow, and we did not want to push too many frontiers at once.

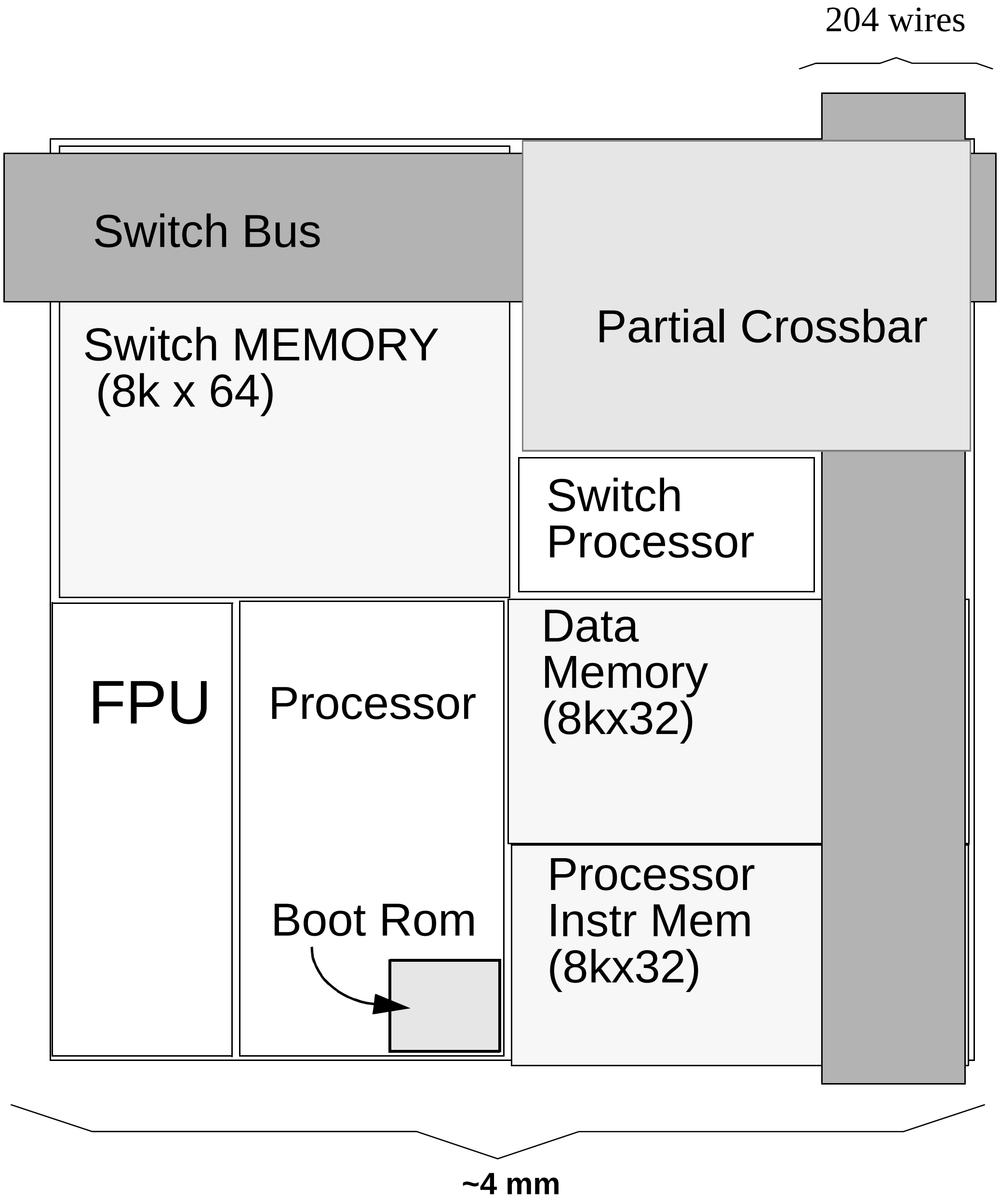
We assume a pessimistic utilization of %45 for safeness, which brings us to 225,000 "real" gates. My preferred area metric of choice, the 32-bit Wallace tree multiplier, is 8000 gates. My estimate of a processor (with multiplier) is that it takes about 10 32 bit multipliers worth of area. A pipelined FPU would add about 4 multipliers worth of area.

The rest remains for the switch processor and crossbars. I do not have a good idea of how much area they will take (the actual logic is small, but the congestion due to the wiring is of concern) We pessimistically assign the remaining 14 multipliers worth of area to these components.

Based on this back-of-the-envelope calculation, a 16 tile Raw system looks eminently reasonable.

This number is calculated using a very conservative wireability ratio for a process with so many layers of metal. Additionally, should we require it, we have the possibility of moving up to a larger die. Note however, that these numbers do not include the area required for I/O buffers and pads, or the clock tree. The addition area due to LSSD (level sensitive scan design) is included.

The figure "A Preliminary Tile Floorplan" is a possible floorplan for the Raw tile. It is optimistic because it assumes some flexibility with memory footprints, and the sizes of logic are approximate. It may well be neces-



A Preliminary Tile Floorplan

sary that we reduce the size of the memories to make things fit. Effort has been made to route the large buses over the memories, which is possible in the SA-27E process. This should improve the routability of the processor greatly, because there are few global wires. Because I am not sure of the area required by the crossbar, I have allocated a large area based on the assumption that crossbar area will be proportional to the square of the width of input wires.

In theory, we could push and make it up to 32 tiles. However, I believe that we would be stretching ourselves very thinly -- the RAMs need to be halved (a big problem considering much of our software technology has code expansion effects), and we would have to assume a much better wireability factor, and possibly dump the FPU.

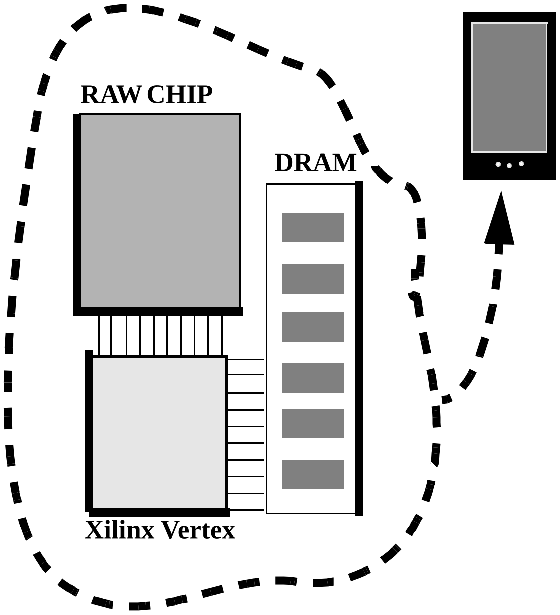
Table 2: Ballpark clock calculation

Structure	Propagation Delay
8192x32 SRAM read	2.50 ns
2-1 Mux	0.20 ns
Register	0.25 ns
Required slack	0.50 ns (estimated)
Total	3.45 ns

For an estimate on clock speed, we need to be a bit more creative because memory timing numbers are not yet available in the SA-27E databooks. We approximate by using the SA-27 "premium" process databook numbers, which should give us a reasonable upper bound. At the very least, we need to have a path in our processor which goes from i-memory to a 2-1 mux to a register. From the databook, we can see the total in the "Ballpark clock calculation" table.

The slack is extra margin required by the ASIC manufacturer to account for routing anomalies, PLL jitter, and process variation. The number given is only an estimate, and has no correlation with the number actually required by IBM.

This calculation shows that, short of undergoing micro-architectural heroics, 290 Mhz is a reasonable strawman UPPER BOUND for our clock rate.



A Raw Handheld Device

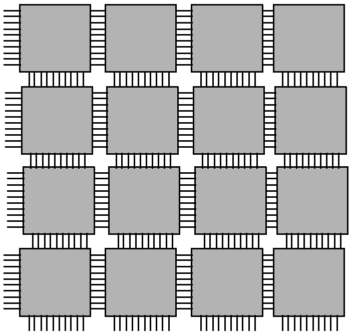
3.2 THE TWO RAW SYSTEMS

Given an estimate of what a Raw chip would look like; we decided to target two systems, a Raw Handheld device, and a Raw Fabric.

3.2.1 A Raw Handheld Device

The Raw handheld device would consist of one Raw chip, a Xilinx Vertex, and 128 MB of SDRAM. The FPGA would be used to interface to a variety of peripherals. The Xilinx part acts both as glue logic and as a signal transceiver. Since we are not focusing on the issue of low-power at this time, this handheld device would not actually run off of battery power (well, perhaps a car battery.).

This Raw system serves a number of purposes. First, it is a simple system, which means that it will make a good test device for a Raw chip. Second, it gets people thinking of the application mode that Raw chips will be used in -- small, portable, extroverted devices rather than large workstations. One of the nice aspects of this device is that we can easily build several of them, and distribute them among our group members. There is something fundamentally more exciting about having a device that we can toss around, rather than a single large prototype sitting inaccessible in the lab. Additionally, it means that people can work on the software required to



A Raw Fabric

get the machine running without jockeying for time on a single machine.

3.2.2 A Multi-chip Raw Fabric, or Supercomputer

This device would incorporate 16 Raw Chips onto a single board, resulting in 256 MIPS processor equivalents on one board. The static and dynamic networks of these chips will be connected together via high-speed I/O running at the core ASIC speed. In effect, the programmer will see one 256-tile Raw chip.

This would give the logical semblance of the Raw chip that we envisioned for the year 2007, where hundreds of tiles fit on a single die. This system will give us the best simulation of what it means to have such an enormous amount of computing resources available. It will help us answer a number of questions. What sort of applications can we create to utilize these processing resources? How does our mentality and programming paradigm change when a tile is a small percentage of the total processing power available to us? What sort of issues exist in the scalability of such a system? We believe that the per-tile cost of a Raw chip will be so low in the future that every handheld device will actually have hundreds of tiles at their disposal.

3.3 SUMMARY

In this chapter, I described the architecture of the Raw prototype. I elaborated on the ASIC process that we are building our prototype in. Finally, I described the two systems that we are planning to build: a hand-held device, and the multi-chip supercomputer.

4 STATIC NETWORK DESIGN

4.0 STATIC NETWORK

The best place to start in explaining the design decisions of the Raw architecture is with the static network.

The static network is the seed around which the rest of the Raw tile design crystallizes. In order to make efficient fine-grained parallel computation feasible, the entire system had to be designed to facilitate high-bandwidth, low latency communication between the tiles. The static network is optimized to route single-word quantities of data, and has no header words. Each tile knows in advance, for each data word it receives, where it must be sent. This is because the compiler (whether human or machine) generated the appropriate route instructions at compile time.

The static network is a point-to-point 2-D mesh network. Each Raw tile is connected to its nearest neighbors through a series of separate, pipelined channels -- one or more channels in each direction for each neighbor. Every cycle, the tile sequences a small, per-tile crossbar which transfers data between the channels. These channels are pipelined so that no wire requires more than one cycle to traverse. This means that the Raw network can be physically scaled to larger numbers of tiles without reducing the clock rate, because the wire lengths and capacitances do not change with the number of tiles. The alternative, large common buses, will encounter scalability problems as the number of tiles connected to those buses increases. In practice, a hybrid approach (with buses connecting neighbor tiles) could be more effective; however, doing so would add complexity and does not seem crucial to the research results.

The topology of the pipelined network which connects the Raw tiles is a 2-D mesh. This makes for an efficient compilation target because the two dimensional logical topology matches that of the physical topology of the tiles. The delay between tiles is then strictly a linear function of the Manhattan distances of the tiles. This topology also allows us to build a Raw chip by merely replicating a series of identical tiles.

4.0.1 Flow Control

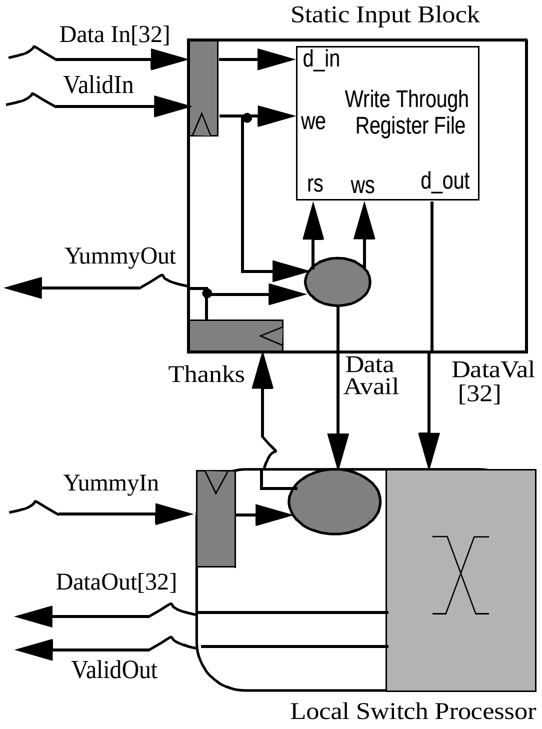
Originally, we envisioned that the network would be precisely cycle-counted -- on each cycle, we would know exactly what signal was on which wire. If the compiler were to incorrectly count, then garbage would be read instead, or the value would disappear off of the wire. This mirrors the behaviour of the FPGA prototype that we designed. For computations that have little or no variability in them, this is not a problem. However, cycle-counting general purpose programs that have more variance in their timing behaviour is more difficult. Two classic examples are cache misses and unbalanced if-then-else statements. The compiler could schedule the computation pessimistically, and assume the worst case, padding the best case with special multi-cycle noop instructions. However, this would have abysmal performance. Alternatively, the compiler could insert explicit flow control instructions to handshake between tiles into the program around these dynamic points. This gets especially hairy if we want to support an interrupt model in the Raw processor.

We eventually moved to a flow-control policy that was somewhere between cycle-counting and a fully dynamic network. We call this policy *static ordering* [Waingold97, 2]. Static ordering is a handshake between crossbars which provides flow control in the static network. When the sequencer attempts to route a dataword which has not arrived yet, it will stall until it does arrive. Additionally, the sequencer will stall if a destination port has no space. Delivery of data words in the face of random delays can then be guaranteed. Each tile still knows a priori the destination and order of each data word coming in; however, it does not know exactly which cycle that will be. This constrasts with a dynamic network, where neither timing nor order are known a priori. Interestingly, in order to obtain good performance, the compiler must cycle count when it schedules the instructions across the Raw fabric. However, with static ordering, it can do so without worrying that imperfect knowledge of program behaviour will violate program correctness.

The main benefits of adding flow control to the architecture are the abstraction layer that it provides and the added support for programs with unpredictable timing. Interestingly, the Warp project at CMU started without flow control in their initial prototypes, and then added it in subsequent revisions [Gross98]. In the next section, we will examine the static input block, which is the hardware used to implement the static ordering protocol.

4.0.2 The Static Input Block

The static input block (SIB) is a FIFO which has both backwards and forwards flow control. There is a



Static Input Block Design

local SIB at every input port on the switch's crossbar. The switch's crossbar also connects to a remote input buffer that belongs to another tile. The figure "Static Input Block Design" shows the static input block and switch crossbar design. Note that an arrow that begins with a squiggle indicates a signal which will arrive at its destination at the end of the cycle. The basic operation of the SIB is as follows:

1. Just before the clock edge, the `DataIn` and `ValidIn` signals arrive at the input flops, coming from the remote switch that the SIB is connected to. The `Thanks` signal arrives from the local switch, indicating if the SIB should remove the item at the head of the fifo. The `Thanks` signal is used to calculate the `YummyOut` signal, which gives the remote switch an idea of how much space is left in the fifo.

2. If `ValidIn` is set, then this is a data word which must be stored in the register file. The protocol ensures that data will not be sent if there is no space in the circular fifo.

3. `DataAvail` is generated based on whether the fifo is empty. The head data word of the queue is propagated out of `DataVal`. These signals travel to the switch.

4. The switch uses `DataAvail` and `DataVal` to perform its route instructions. It also uses the `YummyIn` information to determine if there is space on the remote side of the queue. The `DataOut` and `ValidOut` signals will arrive at a remote input buffer at the end of the cycle.

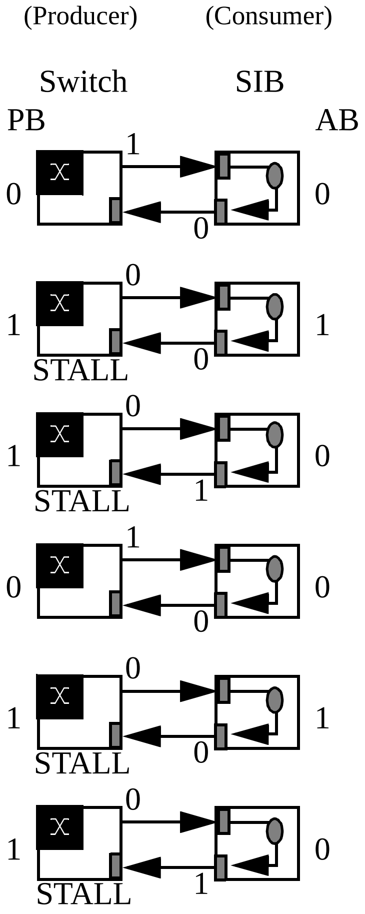
5. If the switch used the data word from the SIB, it asserts `Thanks`.

The subtlety of the SIB comes from that fact that it is a distributed protocol. The receiving SIB is at least one cycle away from the switch that is sending the value. This means that the sender does not have perfect information about how much space is available on the receiver side. As a result, the sender must be conservative about when to send data, so as not to overflow the fifo. This can result in suboptimal performance for streams of data that are starting out, or are recovering from a blockage in the network. The solution is to add a sufficient number of storage elements to the FIFO.

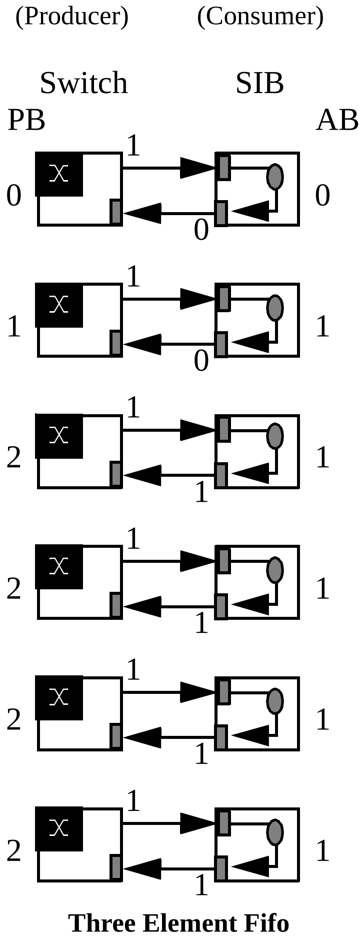
The worksheets "One Element Fifo" and "Three Element Fifo" help illustrate this principle. They show the state of the system after each cycle. The left boxes are a simplified version of the switch circuit. The right boxes are a simplified version of a SIB connected to a remote switch. The top arrow is the `ValidIn` bit, and the bottom arrow is the "`Yummy`" line. The column of numbers underneath "PB" (perceived buffers) are the switch's conservative estimate of the number of elements in the remote SIB at the beginning of the cycle. The column of numbers underneath "AB" (actual buffers) are the actual number of elements in the fifo at the beginning of the cycle.

The two figures model the "Balanced Producer-Consumer" problem, where the producer is capable of producing data every cycle, and the consumer is capable of consuming it every cycle. This would correspond to a stream of data running across the Raw tiles. Both figures show the cycle-by-cycle progress of the communication between a switch and its SIB.

We will explain the "One Element Fifo" figure so the reader can get an idea of how the worksheets work. In the first cycle, we can see that the switch is asserting its `ValidOut` line, sending a data value to the SIB. On the second cycle, the switch stalls because it knows that the Consumer has an element in its buffer, and may not have space if it sends a value. The `ValidOut` line is thus held low. Although it is not indicated in the diagram, the Consumer consumes the data value from the previous cycle. On the third cycle, the SIB asserts the YummyOut line, indicating that the value had been con-



One Element Fifo



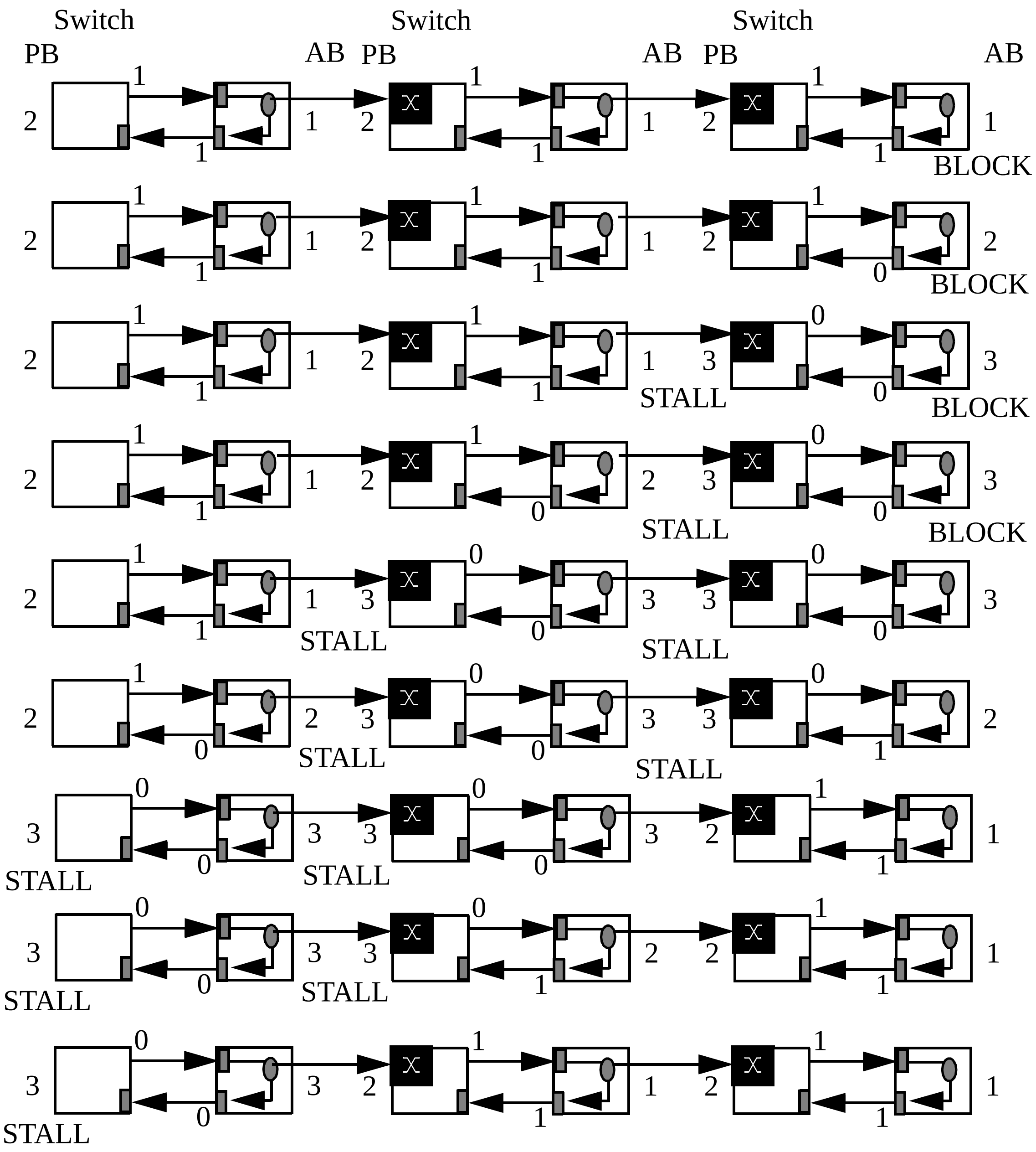
Three Element Fifo

sumed. However, the Switch does not receive this value until the next cycle. Because of this, the switch stalls for another cycle. On the fourth cycle, the switch finally knows that there is buffer space and sends the next value along. The fifth and sixth cycles are exactly like the second and third.

Thus, in the one element case, the static switch is stalling because it cannot guarantee that the receiver will have space. It unfortunately has to wait until it receives notification that the last word was consumed.

In the three element case, the static network and SIBs are able to achieve optimal throughput. The extra storage allows the sender to send up to three times before it hears back from the input buffer that the first value was consumed. It is not a coincidence that this is also the round trip latency from switch to SIB. In fact, if Raw were moved to a technology where it took multiple cycles to cross the pipelined interconnect between tiles (like for instance, for the Raw multi-chip system), the number of buffers would have to be increased to match the new round trip latency. By looking at the diagram, you may think that perhaps two buffers is enough, since that is the maximum perceived element size. In actuality, the switch would have to stall on the third cycle because it perceives 2 elements, and is trying to send a third out before it received the first positive “`Yummy-Out`” signal back.

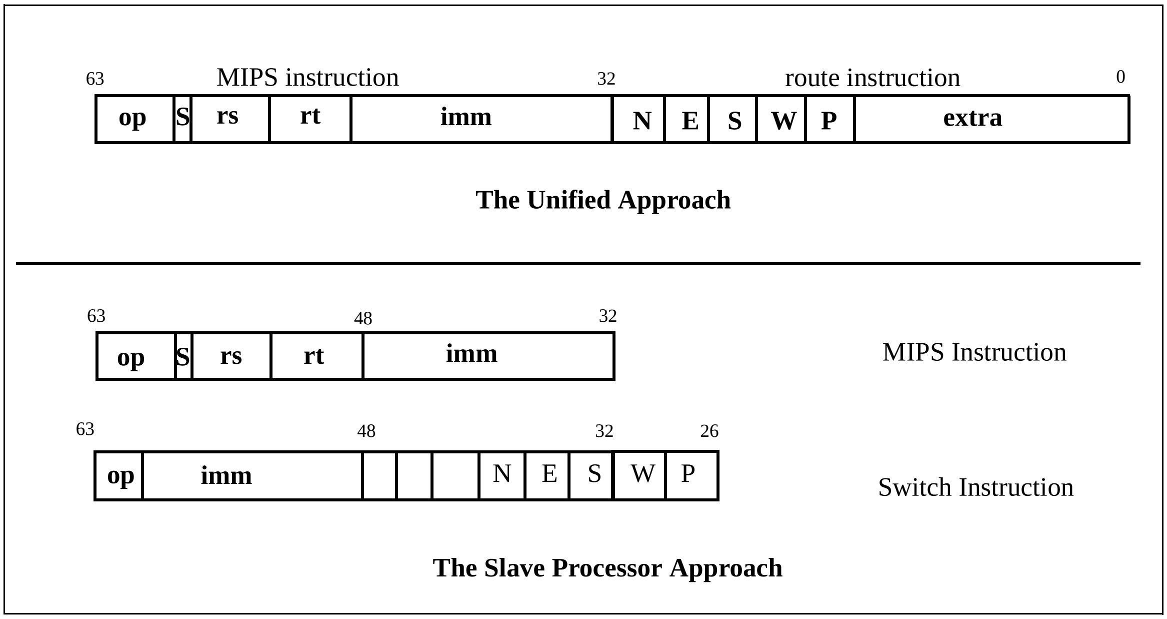
The other case where it is important that the SIBs perform adequately is in the case where there is head-of-line blocking. In this instance, data is being streamed through a line of tiles, attaining the steady state, and then one of the tiles at the head becomes blocked. We want the SIB protocol to insure that the head tile, when unblocked, is capable of reading data at the maximum rate. In other words, the protocol should insure that no bubbles are formed later down the pipeline of producers and consumers. The “Three Element Fifo, continued” figure forms the basis of an inductive proof of this property.



Three Element Fifo, continued
Starts at Steady State, then Head blocks (stalls) for four cycles

I will elaborate on “Three Element Fifo, continued,“some more. In the first cycle, the “BLOCK” indicates that no value is read from the input buffer at the head of the line on that cycle. After one more BLOCKs, in cycle three, the switch behind the head of the line STALLs because it correctly believes that its consumer has run out of space. This stall continues for three more cycles, when the switch receives notice that a value has been dequeued from the head of the queue. These stalls ripple down the chain of producers and consumers, all offsetted by two cycles.

It is likely that even more buffering will provide greater resistance to the performance effects of blockages in the network. However, every element we add to the FIFO is an element that will have to be exposed for draining on a context switch. More simulation results could tell us if increased buffering is worthwhile.



The Unified Approach

The Slave Processor Approach

4.0.3 Static Network Summary

The high order bit is that adding flow control to the network has resulted in a fair amount of additional complexity and architectural state. Additionally, it adds logic to the path from tile to tile, which could have performance implications. With that said, the buffering allows our compiler writers some room to breath, and gives us support for events with unpredictable timing.

4.1 THE SWITCH (SLAVE) PROCESSOR

The switch processor is responsible for controlling the tile's static crossbar. It has very little functionality -- in some senses one might call it a "slave parallel move processor," since all it can do is move values between a small register file, its PC, and the static crossbar.

One of the main decisions that we made early on was whether or not the switch processor would exist at all. Currently, the switch processor is a separately sequenced entity which connects the main processor to the static network. The processor cannot access the static network without the slave processor's cooperation.

A serious alternative to the slave-processor approach would have been to have only the main processor, with a VLIW style processor word which also specified the routes for the crossbar. The diagram "The Unified Approach" shows an example instruction encoding. Evaluating the trade-offs of the unified and slave designs is difficult.

A clear disadvantage of the slave design is that it is more complicated. It is another processor design that we have to do, with its own instruction encoding for branches, jumps, procedure calls and moves for the register file. It also requires more bits to encode a given route.

The main annoyance is that the slave processor requires constant baby-sitting by the main processor. The main processor is responsible for loading and unloading the instruction memory of the switch on cache misses, and for storing away the PCs of the switch on a procedure call (since the switch has no local storage). Whenever the processor takes a conditional branches, it needs to forward the branch condition on to the slave processor. The compiler must make sure there is a branch instruction on the slave processor which will interpret that condition.

Since the communication between the main and slave processors is statically scheduled, it is very difficult and slow to handle dynamic events. Context switches require the processor to freeze the switch, set the PC to an address which drains the register files into the processor, as well as any data outstanding on the switch ports.

The slave switch processor also makes it very difficult to use the static network to talk to the off-chip network at dynamically chosen intervals, for instance, to read a value from a DRAM that is connected to the static network. This is because the main processor will have to freeze the switch, change the switch's PC, and then unfreeze it.

The advantages of the switch processor come in tolerating latency. It decouples the processing of networking instructions and processor instructions. Thus, if a processor takes longer to process an instruction than normal (for instance on a cache miss), the switch instructions can continue to execute, and visa versa. However, they will block when an instruction is executed that requires communication between the two. This model is reminiscent of Decoupled-Execute Access Architectures [Smith82].

The Unified approach does not give us any slack. The instruction and the route must occur at precisely the same time. If the processor code takes less time than expected, it will end up blocked waiting for the switch route to complete. If the processor code takes more time than expected, a "through-route" would be blocked up on unrelated computation. The Unified approach also has the disadvantage that through route instructions must be scheduled on both sides of an if-statement. If the two sides of the if-statement were wildly unbalanced this would create code bloat. The Slave approach would only need to have one copy of the corresponding route instructions.

In the face of a desire for this decoupling property, we have further entertained the idea of another approach, called the Decoupled-Unified approach. This would be like the Unified approach, except it would involve having a queue through which we would feed the static crossbar its route instructions. This is attractive because it would decouple the two processes. The processor would sequence, and queue up switch instructions, which would execute when ready.

With this architecture, the compiler would push the switch instructions up to pair with the processor instructions at the top of a basic block. This way through-routes could execute as soon as possible.

Switch instructions that originally ran concurrently with non-global IF-ELSE statements need some extra care. Ideally, the instructions would be propagated above the IF-ELSE statement. Otherwise, the switch instructions will have to be copied to both sides the IF-ELSE clause. This may result in code explosion, if the number of switch instructions propagated into the IF-ELSE statement is greater than the length of one of the sides of the statement.

When interrupts are taken into account, the Decoupled-Unified approach is a nightmare, because now we have situations where half of the instruction (the processor part) has executed. We can not just wait for the switch instructions to execute, because this may take an indefinite amount of time.

To really investigate the relative advantages and disadvantages of the three methods would require an extensive study, involving modifications of our compilers and simulators. To make a fair comparison, we would need to spend as much time optimizing the comparison simulators as we did the originals. In an ideal world, we might have pursued this issue more. However, given the extensive amount of infrastructure that had already been built using the Slave model, we could not justify the time investment for something which was unlikely to buy us performance, and would require such an extensive reengineering effort.

4.1.1 Partial Routes

One idea that our group members had was that we do not need to make sure that all routes specified in an instruction happen simultaneously. They could just fire off when they are possible, with that part of the instruction field resetting itself to the "null route." When all fields are set to null, that means we can continue onto the next instruction. This algorithm continues to preserves the static ordering property.

From a performance and circuit perspective, this is a win. It will decouple unrelated routes that are going through the processor. Additionally, the stall logic in the switch processor does not need to OR together the success of all of the routes in order to generate the "`ValidOut`" signal that goes to the neighboring tile.

The problem is, with partial routes, we again have an instruction atomicity problem. If we need to interrupt the switch processor, we have no clear sense of which instruction we are currently at, since parts of the instruction have already executed. We cannot wait for the instruction to fully complete, because this may take indefinite amount of time. In order to make this feature, we would have had to add special mechanisms to overcome this problem. As a result, we decided to take the simple path and stall until such a point as we can route all of the values atomically.

4.1.2 Virtual Switch Instruction Memory

In order to be able to run large programs, we need a mechanism to page code in and out of the various memories. The switch memory is a bit of an issue because it is not coupled directly with the processor, and yet it does not have the means to write to its own memory. Thus, we need the processor to help out in filling in the switch memory.

There are two approaches.

In the first approach, the switch executes until it reaches a “trap” instruction. This trap instruction indicates that it needs to page in a new section of memory. The trap causes an interrupt in the processor. The processor fetches the relevant instructions and writes it into the switch processor instruction memory. It then signals the switch processor, telling it to resume.

In the second approach, we maintain a mapping between switch and processor instruction codes. When the processor reaches a junction where it needs to pull in some code, it pulls in the corresponding code for the switch. The key issue is to make sure that the switch does not execute off into the weeds while this occurs. The switch can very simply do a read from the processor’s output port into its register set (or perhaps a branch target.) This way, the processor can signal the switch when it has finished writing the instructions. When the switch’s read completes, it knows that the code has been put in place. Since there essentially has to be a mapping between the switch code and the processor code if they communicate, this mapping is not hard to derive. The only disadvantage is that due to the relative sizes of basic blocks in the two memories, it may be the case that one needs to page in and the other doesn’t. For the most part I do not think that this will be much of a problem. If we want to save the cost of this output port read after the corresponding code has been pulled in, we can re-write that instruction.

In the end, we decided on the second option, because it was simpler. The only problem we foresee is if the tile itself is doing a completely unrelated computation (and communicating via dynamic network.) Then, the switch, presumably doing through routes, has no mechanism of telling the local tile that it needs new instructions. However, presumably the switch is synchronized with at least one tile on the chip. That tile could send a dynamic message to the switch’s master, telling it to load in the appropriate instructions. We don’t expect that anyone will really do this, though.

4.2 STATIC NETWORK BANDWIDTH

One of the questions that needs to be answered is how much bandwidth is needed in the static switch. Since a ALU operation typically has two inputs, having only one $csti port means that one of the inputs to the instruction must reside inside the tile to not be bottlenecked. The amount of bandwidth into the tile determines very strongly the manner in which code is compiled to it. As it turns out, the RAWCC compiler optimizes the code to minimize communication, so it is not usually severely affected by this bottleneck. However, when code is compiled in a pipeline fashion across the Raw tiles, more bandwidth would be required to obtain full performance.

A proposed modification to the current Raw architecture is to add the network ports csti2, cNi2, cSi2, cEi2, and cWi2. It remains to be evaluated what the speedup numbers and area (both static instruction memory, crossbar and wire area) and clock cycle costs are for this optimization. As it turns out, the encoding for this fits neatly in a 64-bit switch instruction word.

4.3 SUMMARY

The static network design makes a number of important trade-offs. The network flow control protocol contains flow-controlled buffers that allow our compiler writers some room to breath, and gives us support for events with unpredictable timing. This protocol is a distributed protocol in which the producers have imperfect information. As a result, the SIBs require a small amount of buffering to prevent delay. In this chapter, I presented a simple method for calculating how big these buffer sizes need to be in order to allow continuous streams to pass through the network bubble-free.

The static switch design also has some built-in slack for dynamic timing behaviour between the tile processor and the switch processor. This slack comes with the cost of added complexity.

Finally, we raised the issue of the static switch bandwidth, and gave a simple solution for increasing it.

All in all, the switch design is a success; it provides an effective low-latency network for inter-tile communication. In the next section, we will see how the static network is interfaced to the tile’s processor.

5 DYNAMIC NETWORK

5.0 DYNAMIC NETWORK

Shortly after we developed the static network, we realized the need for the dynamic network. In order for the static network to be a high performance solution, the following must hold:

1. The destinations must be known at compile time.

2. The message sizes must be known at compile time.

3. For any two communication routes that cross, the compiler must be able generate a switch schedule which merges those two communication patterns on a cycle by cycle basis.

The static network can actually support messages which violate these conditions. However, doing this requires an expensive layer of interpretation to simulate a dynamic network.

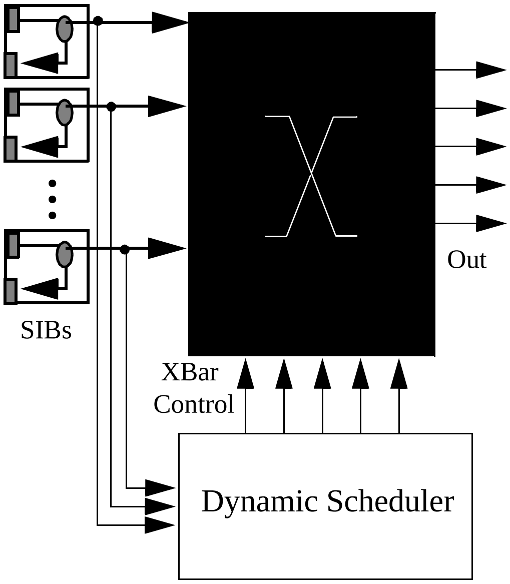
The dynamic network was added to the architecture to provide support for messages which do not fulfill these criteria.

The primary intention of the dynamic network is to support memory accesses that cannot be statically analyzed. The dynamic network was also intended to support other dynamic activities, like interrupts, dynamic I/O accesses, speculation, synchronization, and context switches. Finally, the dynamic network was the catch-all safety net for any dynamic events that we may have missed out on.

In my opinion, the dynamic network is probably the single most complicated part of the Raw architecture. Interestingly enough, the design of the actual hardware is quite straight-forward. Its interactions with other parts of the system, and in particular, the deadlock issues, can be a nightmare if not handled correctly. For more discussions on the deadlock issues, please refer to the section entitled “Deadlock.”

5.1 DYNAMIC ROUTER

The dynamic network is a dimension-ordered, wormhole routed flow-controlled network [Dally86]. Each dynamic network message has a header, followed by a number of datawords. The header is constructed by the hardware. The router routes in the X direction, then in the Y direction. We implemented the protocol on top of the SIB protocol that was used for the static network. The figure entitled “The Dynamic Network Router” illustrates this. The dynamic network device is identical to the static network, except it has a dynamic scheduler instead of the actual switch processor. The processor’s interface is also slightly different.



The Dynamic Network Router

The scheduler examines the header of incoming messages. The header contains a route encoded by a relative X position and a relative Y position. If the X position is non zero, then it initializes a state machine which will transfer one word per cycle of the message out of the west or east port, based on the sign of the distance. If the X position is zero, then the message is sent out of the south or north ports. If both X and Y are zero, then the message is routed into the processor. Because multiple input messages can contend for the same output port, there needs to be a priority scheme. We use a simple round robin scheduler to select between contenders. This means that an aggressive producer of data will not be able to block other users of the network from getting their messages through.

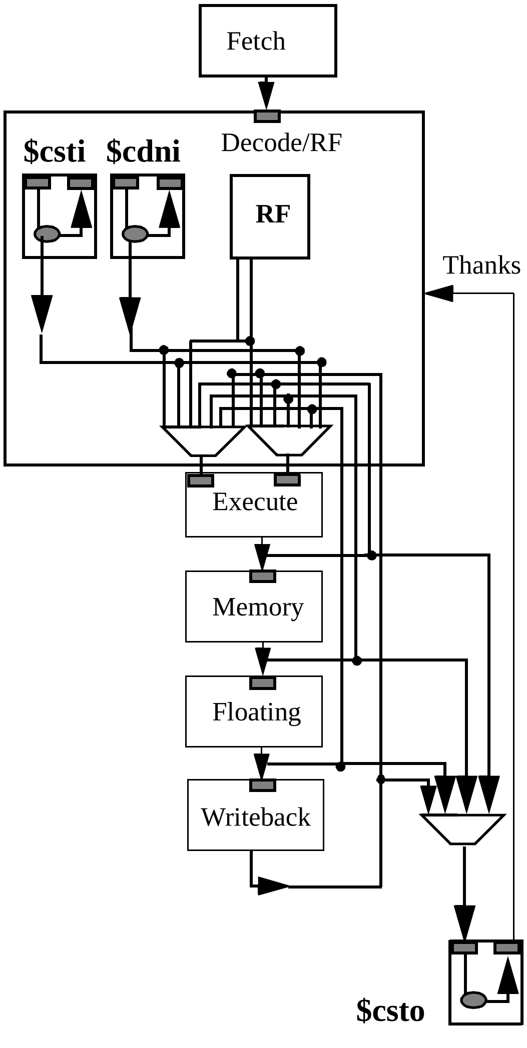
Because the scheduler must parse the message header, and then modify it to forward it along, it currently takes two cycles for the header to pass through the network. Each word after that only takes one cycle. It may be possible to redesign the dynamic router to pack the message parsing and send into one cycle. However, we did not want to risk the dynamic network becoming a critical path in our chip, since it is in many senses a backup network. It may also be possible that we could have a speculative method for setting up dynamic channels. With more and more Raw tiles (and the more hops to cross the chip), the issue of dynamic message latency becomes increasingly important. However, for the initial prototype, we decided to keep things simple.

5.2 SUMMARY

The dynamic network design leveraged many of the same underlying hardware components as the static switch design. Its performance is not as good as the static network's because the route directions are not known a priori. A great deal more will be said on the dynamic network in the Deadlock section of this thesis.

6 TILE PROCESSOR DESIGN

When we first set out to define the architecture, we chose the 5-stage MIPS R2000 as our baseline processor for the Raw tile. We did this because it has a relatively simple pipeline, and because many of us had spent hundreds of hours staring at that particular pipeline. The R2000 is the canonical pipeline studied in 6.823, the graduate computer architecture class at MIT. The discussion that follows assumes familiarity with the R2000 pipeline. For an introduction, see [Hennessey96]. (Later, because of the floating point unit, we expanded the pipeline to six stages.)



$csto, Bypass, and Writeback Networks

6.0 NETWORK INTERFACE

The most important part of the main processor decision is the way in which it interfaces with the networks. Minimizing the latency from tile to tile (especially on the static network) was our primary goal. The smaller the latency, the greater the number of applications that can be effectively parallelized on the Raw chip.

Because of our desire to minimize the latency from tile to tile, we decided that the static network interface should be directly attached to the processor pipeline. An alternative would have been to have explicit MOVE instructions which accessed the network ports. Instead, we wanted a single instruction to be able to read a value from the network, operate on it, and write it out in the same cycle.

We modified the instruction encodings in two ways to accomplish this magic.

For writes to the network output port SIB, $csto, we modified the encoding the MIPS instruction set to include what we call the “S” bit. The S bit is set to true if the result of the instruction should be sent out to the output port, in addition to the destination register. This allows us to send a value out of the network and keep it locally. Logically, this is useful when an operation in the program dataflow graph has a fanout greater than one. We used one of the bits from the opcode field of the original MIPS ISA to encode this.

For the input ports, we mapped the network port names into the register file name space:

Reg	Alias	Usage
$24	$csti	Static network input port.
$25	$cdn[i/o]	Dynamic network input port.

This means, for instance, that when register $24 is referenced, it actually takes the result from the static network input SIB.

With the current 5-bit addressing of registers, additional register names would only be possible by adding one more bit to the register address space. Aliasing it with an existing register name allows us to leave most of the ISA encodings unaffected. The choice of the register numbers was suggested by Ben Greenwald. He believes that we can maximize our compatibility with existing

MIPS tools by reserving that particular register because it has been designated as “temporary.”

6.1 SWITCH BYPASSING

The diagram entitled “$csto, Bypass and Writeback Networks” shows how the network SIBs are hooked up to the processor pipeline. The three muxes are essentially bypass muxes. The $csti and $cdni SIBs are logically in the decode/register fetch (RF) stage.

In order to reduce the latency of a network send, it was important that an instruction deliver its result to the $csto SIB as soon as the value was available, rather than waiting until the writeback stage. This can change the tile-to-tile communication latency from 6 cycles to 3 cycles.

The $csto and $cdno SIBs are connected to the processor pipeline in much the same way that register bypasses are connected. Values can be sent to $csto after the ALU stage, after the MEMORY stage, after the FPU stage, and at the WB stage. This gives us the minimum possible latency for all operations whose destination is the static network. The logic is very similar to the bypassing logic; however the priority of the elements is reversed: $csto wants the OLDEST value from the pipeline, rather than the newest one.

When a instruction that writes to $csto is executed, the S bit travels with it down the pipeline. A similar thing happens with a write to $cdno, except that the “D” bit is generated by the decode logic. Each cycle, the $csto bypassing logic finds the oldest instruction which has the S bit set. If that instruction is not ready, then the valid bit connecting to the output SIB is not asserted. If the oldest instruction has reached its stage of maturation (i.e., the stage at which the result of the computation is ready), then the value is muxed into the $csto port register, ready to enter into an input buffer on the next cycle. The S bit of that instruction is cleared, because the instruction has sent its value. When the instruction reaches the Writeback stage, it will also write its result into the register file.

It is interesting to note that the logic for this procedure is exactly the same as for the standard bypass logic, except that the priorities are reversed. Bypass logic favors the youngest instruction that is writing a particular value. $csto bypassing logic looks for the oldest instruction with the S bit set because it wants to guarantee that values are sent out of the network in order that the instructions were issued.

The $cdni and $csti network ports are muxed in through the bypass muxes. In this case, when an instruction in the decode stage uses registers $24 or $25 as a source, it checks if the `DataAvail` signal of the SIB is set. If it is not, then the instruction stalls. This mirrors a hardware register interlock. If the decode stage decides it does not have to stall, it will acknowledge the receipt of the data value by asserting the appropriate `Thanks` line.

6.1.1 Instruction Restartability

The addition of the tightly coupled network interfaces does not come entirely for free. It imposes a number of restrictions on the operation of the pipeline.

The main issue is that of restartability. Many processor pipelines take advantage of the fact that their instruction sets are restartable. This means that the processor can squash the instruction at any point in the pipeline before the writeback stage. Unfortunately, instructions which access $csti and $cdni modify the state of the networks. Similarly, when an instruction issues an instruction which writes to $cdno or $csto, once the result has been sent out to the switch’s SIB, it is beyond the point of no return and cannot be restarted.

Because of this, the commit point of the tile processor is right after it passes the decode stage. We have to be very careful about instructions that write to $csto or $cdno because the commit point is so early in the pipeline. If we allow the instructions to stall (because the output queues are full) in a stage beyond the decode stage, then the pipeline could be stalled indefinitely, This is because it is programmatically correct for the output queue to be full indefinitely. At that point, the processor cannot take an interrupt, because it must finish all of the “committed” instructions that passed decode.

Thus, we must also insure that if an instruction passes decode, it must not be possible for it to stall indefinitely.

To avoid these stalls, we do not let instruction pass decode unless there is guaranteed to be enough room in the appropriate SIB. As you might guess, we need to use the same analysis as we used to calculate how much buffer space we needed in the network SIBs. Having the correct number of buffers will ensure that the processor is not too conservative. Looking at the “$csto, Bypass, and Writeback Networks”, diagram, we count the number of pipeline registers in the longest cycle from the decode stage through the Thanks line, back to the decode stage. Six buffers are required.

An alternative to this subtle approach is that we could modify the behaviour of the SIBs. We can keep the values in the input SIB FIFOs until we are sure we do not need them any more. Each SIB FIFO will have three pointers: one marks the place where data should be inserted, the next marks where data should be read from, and the final one marks the position of the next element that would be committed. If instructions ever need to be squash, the "read" pointer can be reset to equal the "commit" pointer. I do not believe that this would affect the critical paths significantly, but the $csti and $cdni SIBs would require nine buffers each instead of three.

For the output SIB, creating restartability is a harder problem. We would have to defer the actual transmittal of the value through the network until the instruction has hit WRITEBACK. However, that would mean that we could not use our latency reducing bypassing optimization. This approach mirrors what some conventional microprocessors do to make store instructions restartable -- the write is deferred until we are absolute sure we need it. An alternative is to have some sort of mechanism which overrides a message that was already sent into the network. That sounded complicated.

6.1.2 Calculating the Tile-to-Tile Communication Latency

A useful exercise is to examine the tile-to-tile latency of the network send. The figure "Processor-Switch-Switch-Processor" path helps illustrate it. It shows the pipelines of two Raw tiles, and the path over the static network between them. The relevant path is in bold. As you can see, it takes three cycles for nearest neighbor communication.

It is possible that we could reduce the cost down to two cycles. This would involve removing the register in front of the $csti SIB, and rearranging some of the logic. We can do this because we know that the path between the switch's crossbar and the SIB is on the same tile, and thus short. However, it is not at all clear that this will not lengthen the critical path in the tile design. Whether we will be able to do this or not will become more apparent as we come closer to closing the timing issues of our verilog.

6.2 MORE STATIC SWITCH INTERFACE GOOK

A number of other items are required to make the static switch and main processor work together.

The first is a mechanism to write and read from the static network instruction memory. The sload and sstore operations stall the static switch for a cycle.

Another mechanism allows us to freeze the switch. This lets the processor inspect the state at its leisure. It also simplifies the process of loading in a new PC and NPC.

During context switches and booting, it is useful to be able to see how many elements are in the switch's SIBs. There is a status register in the processor which can be read to attain this information.

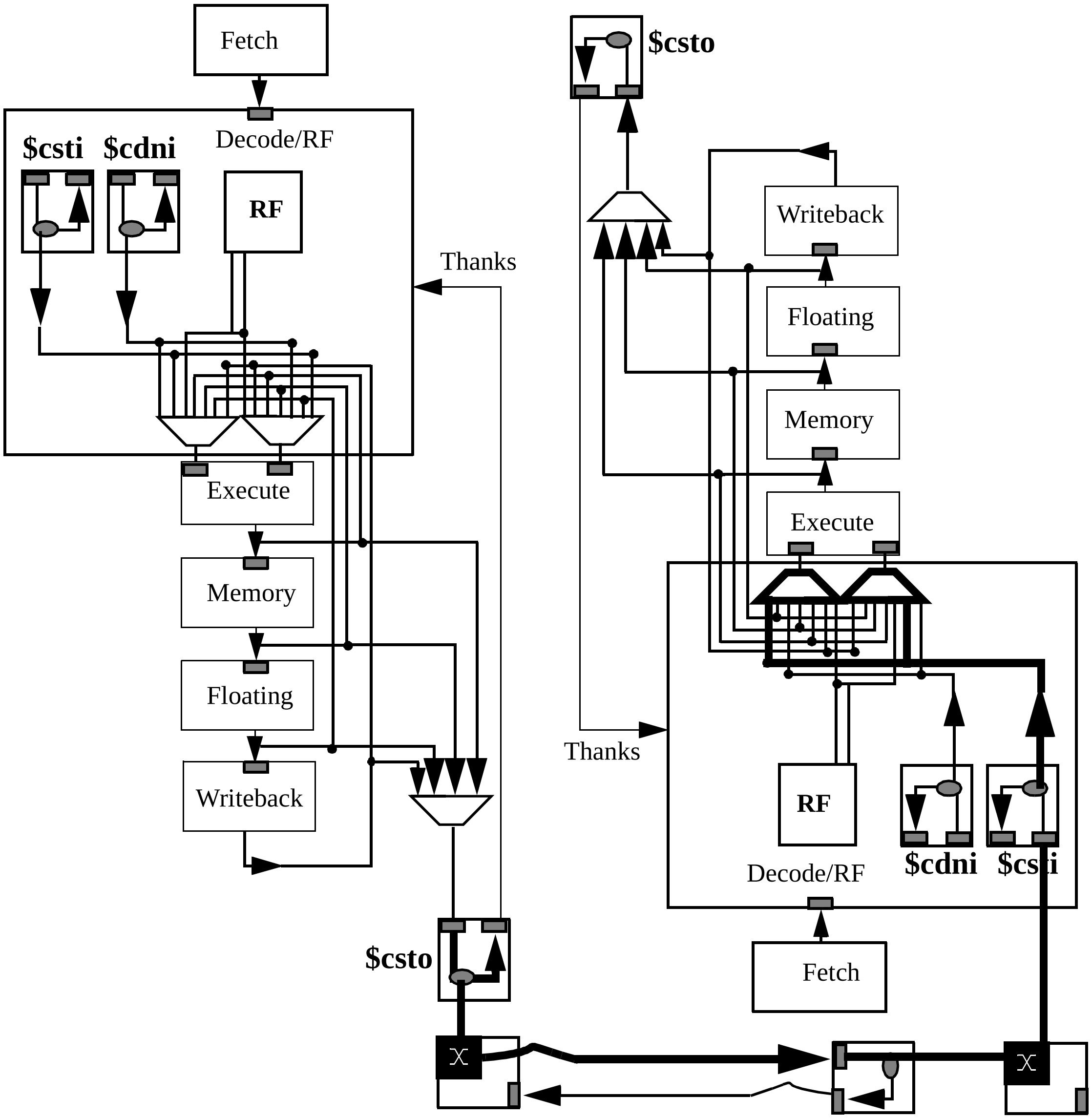
Finally, there is a mechanism to load in a new PC and NPC, for context switches, or if we want the static switch to do something dynamic on our behalf.

6.3 MECHANISM FOR READING AND WRITING INTO INSTRUCTION MEMORY

In order for us to change the stored program, we need some way of writing values into the instruction memory. Additionally, however, we want to be able to read all of the state out of the processor (which includes the instruction memory state), and we would like to support research into a sophisticated software instruction VM system. As such, we need to be able to treat the instruction memory as a true read-write memory. The basic thinking on this issue is that we will support two new instructions -- "iload" and "istore" -- which mimic the data versions but which access the instruction memory. The advantage of these instructions is that it makes it very explicit when we are doing things which are not standard, both in the hardware implementation and in debugging software. These instructions will perform their operations in the "memory" stage of the pipeline, stealing a cycle away from the "fetch" stage. This means that every read or write into instruction memory will cause a one cycle stall. Since this is not likely to be a common event, we will not concern ourselves with the performance implications.

Associated with an instruction write will be some window of time (i.e. two or three cycles unless we add in some sort of instruction prefetch, then it would be more) where an instruction write will not be reflected in the processor execution. I.E., instructions already fetched into the pipeline will not be refetched if they happen to be the ones that were changed. This is a standard caveat made by most processor architectures.

We also considered the alternative of using standard "load" and "store" instructions, and using a special address range, like for instance ("0xFFFFxxxx"). This



The Processor-Switch-Switch-Processor path

approach is entirely valid and has the added benefit that standard routines (“memcpy”) will be able to modify instruction memory without having special version. (If we wanted true transparency, we’d have to make sure that instruction memory was accessible by byte accesses.) We do not believe this to be a crucial requirement at this time. If needbe, the two methods could also easily co-exist.

6.4 RANDOM TWEAKS

Our baseline processor was the MIPS R2000. We added load interlocks into the architecture, because they aren’t that costly. Instead of a single multi-cycle multiply instruction, there are three low-latency pipelined instructions, MULH, MULHU, and MULLO which place their results in a GPR instead of HI/LO. We did this because our 32-bit multiply takes only two cycles. It didn’t make sense to treat it as a multi cycle instruction when it has no more delay than a load. We also removed the SWL and SWR instructions, because we didn’t feel they were worth the implementation complexity.

We have a 64 bit cycle counter which lists the number of cycles since reset. There is also a watchdog timer, which is discussed in the DEADLOCK section of the thesis.

Finally, we decided on a Harvard style architecture, with separate instruction and data memories; because the design of the pipeline was more simple. See the Appendage entitled “Raw User’s Manual” for a description of the instruction set of the Raw prototype. The first Appendage shows the pipeline of the main processor.

6.5 THE FLOATING POINT UNIT

In the beginning, we were not sure if we were going to have a floating point unit. The complexity seemed burdensome, and there were some ideas of doing it in software. One of our group members, Michael Zhang, implemented and parallelized a software floating point library [Zhang99] to evaluate the performance of a software solution. Our realization was that many of our applications made heavy use of floating point, and for that, there is no subsitute for hardware. We felt that the large dynamic range offered by floating point would further the ease of writing signal processing applications -- an important consideration for enticing other groups to make user of our prototype. This was an important consideration To simplify our task, we relaxed our compliance of the IEEE 754 standard. In particular, we do not implement gradual underflow. We decided to support only single-precision floating point operations so we would not need to worry about how to integrate a 64 bit datapath into the RAW processor. All of the network paths are 32bits, so we would have package up values and route them, reassemble them and so on. However, if we were building an industrial version, we would probably have a 64 bit datapath throughout the chip, and double precision would be easier to realize.

It was important that the FPU be as tightly integrated with the static network as the ALU. In terms of floating point, Raw had the capability of being a supercomputer even as an academic project. With only a little extra effort in getting the floating point right, we could make Raw look very exciting.

We wanted to be able to send data into the FPU in a pipelined fashion and have it stream out of the tile just as we would do with a LOAD instruction. This would yield excellent performance with signal processing codes, especially with the appropriate amount of switch bandwidth. The problem that this presented was with the $csto port. We need to make sure that values exit the $csto port in the correct order from the various floating point functional units, and from the ALU.

The other added complexity with the floating point unit is the fact that its pipeline is longer than the corresponding ALU pipeline. This means that we needed to do some extra work in order to make sure that items are stored back correctly in the writeback phase, and that they are transferred into the static network in the correct order.

The solution that we used was simple and elegant. After researching FPU designs [Oberman96], it became increasingly apparent that we could do both floating point pipelined add and multiply in three cycles. The longest operation in the integer pipeline is a load or multiply, which is two cycles. Since they are so close, we discovered that we could solve both the $csto and register file writeback problems by extending the length of the overall pipeline by one cycle. As a result, we have six pipeline stages: instruction fetch(IF), instruction decode(ID), execution(EXE), memory(MEM), floating point (FPU) and write-back(WB). See Appendix B for a diagram of the pipeline. The floating point operations execute during the Execute, Memory, and FPU stages, and write back at the same stage as the ALU instructions.

This solves the writeback and $csto issues -- once the pipelines are merged, the standard bypass and stall logic can be used to maintain sanity in the pipeline.

This solution becomes more and more expensive as the difference in actual pipeline latencies of the instructions grows. Each additional stage requires at least one more input to the bypass muxes.

As it turns out, this was also useful for implementing byte and half-word loads, which use an extra stage after the memory stage.

Finally, for floating point division, our non-pipelined 11-cycle divider uses the same decoupled HI/LO interface as the integer divide instruction.

A secondary goal we had in designing an FPU is that we make the source available for other research projects to use. Our design is constructed to be extremely portable, and will probably make its way onto the web in the near future.

6.6 RECONFIGURABLE LOGIC

Originally, each Raw tile was to have reconfigurable logic inside, to support bit-level and byte-level computations. Although no research can definitely say that this is a bad idea, we can say that we had a number of problems realizing this goal. The first problem is that we had trouble finding a large number of applications that benefited enormously from this functionality. Median filter and Conway’s “game of life”

[Berklekamp82] were the top two contenders. Although this may seem surprising given RawLogic's impressive results on many programs, much of RawLogic's performance came from massive parallelism, which the Raw architecture leverages very capably with tile-level parallelism. Secondly, it was not clear if a reconfigurable fabric could be efficiently implemented on an ASIC. Third, interfacing the processor pipeline to the reconfigurable logic in a way that effectively used the reconfigurable logic proved difficult. Fourth, it looked as if a large area would need to be allocated to each reconfigurable logic block to attain appreciable performance gains. Finally, and probably most fundamentally for us, the complexity of the reconfigurable logic, its interface, and the software system was an added burden to the implementation of an already quite complicated chip.

For reference, here is the description of the reconfigurable logic interface that we used in the first simulator:

The pipeline interface to the reconfigurable logic mimiced the connection to the dynamic network ports. There were two register mapped ports, RLO (output to RL) and RLI (input from RL to processor). These were aliased with register 30. There was a two element buffer on the RLI connection on the processor pipeline side, and a two element buffer on the reconfigurable logic input side.

6.7 DYNAMIC NETWORK INTERFACE

The processor initiates a dynamic network send by writing the destination tile number, writing the message into the $cdno commit buffer and then executing the `dlaunch` instruction [Kubiatowicz98]. $cdno is different than other SIBs because it buffers up an entire message before a `dlaunch` instruction causes it to trickle into the network. If we were to allow the messages to be injected directly into the network without queueing them up into atomic units, we could have a phenomenon we call dangling. This means that a half-constructed message is hanging out into the dynamic network. Dangling becomes a problem when interrupts occur. The interrupt handler may want to use the dynamic network output queue; however, there is a half-completed message that is blocking up the network port. The message cannot be squashed because some of the words have already been transmitted. A similar problem occurs with context switches -- to allow dangling, the context switch routine would need to save and restore the internal state of the hardware Dynamic scheduler -- a prospect we do not relish. The commit buffer has to be of a fixed size. This size imposes a maximum message size constraint on the dynamic network. To reduce the complexity of the commit buffer, a write to $cdno blocks until all of the elements of the previous message have drained out.

One alternative to the commit buffer would be to require the user to enclose their dynamic network activity to constrained regions surround by interrupt enables and disables. The problem with this approach is that the tile may block indefinitely because the network queue is backed up (and potentially for a legitimate reason.) That would make the tile completely unresponsive to interrupts.

$cdni, on the other hand, operates exactly like the $csti port. However, there is a mask which, when enabled, causes a user interrupt routine to be called when the header of a message arrives at the tile.

6.8 SUMMARY

The Raw tile processor design descended from the MIPS R2000 pipeline design. The most interesting design decisions involved the integration of the network interfaces. It was important that these interfaces (in particular the static network interface) provide the minimal possible latency to the network so as to support as fine-grained parallelism as possible.

7 I/O AND MEMORY SYSTEM

7.0 THE I/O SYSTEM

The I/O system of a Raw processor is a crucial but up until now mostly unmentioned aspect of Raw. The Raw I/O philosophy mirrors that of the Raw parallelism philosophy. Just as we provide a simple interface for the compiler to exploit the gobs of silicon resources, we also have a simple interface for the compiler to exploit and program the gobs of pins available. Once again, the Raw architecture proves effective not because it allocates the raw pin resources to special purpose tasks, but because it exposes them to the compiler and user to meet the needs of application. The interface that we show scales with the number of pins, and works even though pin counts are not growing as fast as logic density.

An effective parallel I/O interface is especially important for a processor with so many processing resources. To support extroverted computing, a Raw architecture's I/O system must be able to interface to, at high-speed, a rich variety of input and output devices, like PCI, DRAM, SRAM, video, RF digitizers and transmitters and so on. It is likely, that in the future, a Raw device would also have direct analog connections - RF receivers and transmitters, and A/D and D/A converters, all exposed to the compiler. However, the integration of analog devices onto a silicon die is the subject of another thesis.

For the Raw prototype, we will settle for being able to interface to some helper chips which can speak these dialects on our behalf.

Recently, there has been a proliferation of high speed signalling technologies like that chips SSTL, HSTL, GTL, LVTTL, and PCI. For our chip, we have been looking at SSTL and HSTL as potential candidates.

We expect to use the Xilinx Vertex parts to convert from our high-speed protocol of choice to other signaling technologies. These parts have the exciting ability to configurably communicate with almost all of the major signaling technologies. Although, in our prototype, these chips are external, I think that it is likely configurable I/O cells will find their way into the new extroverted processors. This is because it will be so crucial for these processors to be able to communicate with all shapes and forms of devices. It may also be the case that extroverted processors will have bit-wise configurable FPGA logic near the I/O pins, for gluing together hardware protocols. After all, isn't glue logic what FPGAs were invented for? Perhaps our original conception of having fine-grained configurable logic on the chip wasn't so wrong; we just had it in the wrong place.

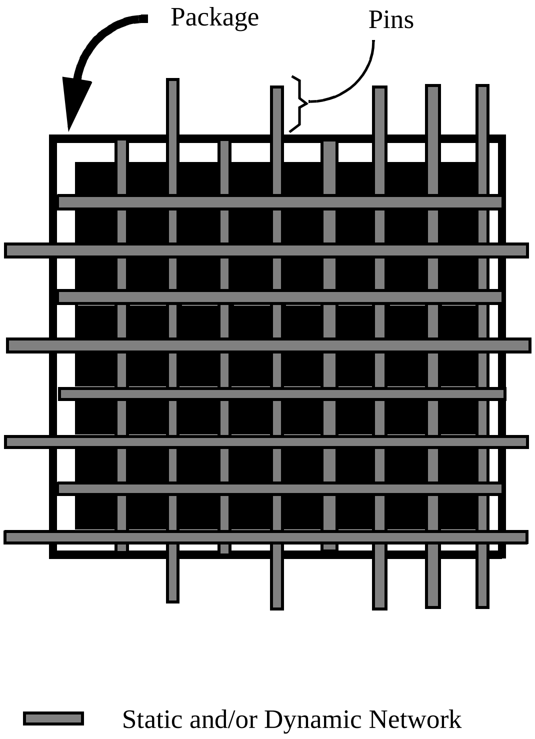
7.0.1 Raw I/O Model

I/O is a first-class software-exposed architectural entity on Raw. The pins of the Raw processor are an extension of both the mesh static and dynamic networks. For instance, when the west-most tiles on a Raw chip route a dynamic or static message to the west, the data values appear on the corresponding pins. Likewise, when an external device asserts the pins, they appear on-chip as messages on the static or dynamic network.

For the Raw prototype, the protocol spoken over the pins is the same static and dynamic handshaking network protocols spoken between tiles. If we actually had the FPGA glue logic on chip, the pins would support arbitrary handshaking protocols, including ones which require the pins to be bidirectional. Of course, for super-high speed I/O connections, there could be a fast-path straight to the pins.

The diagram "Logical View of a Raw Chip" illustrates the pin methodology. The striped lines represent the static and dynamic network pipelined buses. Some of them extend off the edge of the package, onto the pins. The number of static and dynamic network buses that are exposed off-chip is a function of the number of I/O pins that makes sense for the chip. There may only be one link, for ultra-cheap packages, or there may be total connectivity in a multi-chip module. In some cases, the number of static or dynamic buses that are exposed could be different. Or there may be a multiplex bit, which specifies whether the particular word transferred that cycle is a dynamic or static word. The compiler, given the pin image of the chip, schedules the dynamic and static communication on the chip such that it maximizes the utilization of the ports that exist on the particular Raw chip. I/O sends to non-existent ports will disappear.

The central idea is that the architecture facilitates I/O flexibility and scalability. The I/O capabilities can be scaled up or down according to the application. The I/O interface is a first-class citizen. It is not shoehorned through the memory hierarchy, and it provides an inter-



Logical View of a Raw Chip

face which gives the compiler the access to the full bandwidth of the pins.

Originally, only the static network was exposed to the pins. The reasoning was that the static network would provide the highest bandwidth interface into the Raw tiles. Later, however, we realized that, just as the internal networks require support for both static and dynamic events, so too do the external networks. Cache line fills, external interrupts, and asynchronous devices are dynamic, and cannot be efficiently scheduled over the static network. On the other hand, the static network is the most effective method for processing a high bandwidth stream coming in at a steady rate from an outside source.

7.0.2 The location of the I/O ports (Perimeter versus Area I/O)

Area I/O is becoming increasingly common in today's fabrication facilities. In fact, in order to attain the pincounts that we desire on the SA-27E process, we have to use area I/O. This creates a bit of a problem, because all of our I/O connections are focused around the outside of the chip. IBM's technology allows us to simulate a peripheral I/O chip with area I/O. However, this may not be an option in the future. In that event, it is possible to change the I/O model to match. In the Area I/O model, each switch and dynamic switch would have an extra port, which could potentially go in/out to the area I/O pads. This arrangement would create better locality between the source of the outgoing signal and the position of the actual pad on the die. Like in the peripheral case, these I/Os could be sparsely allocated.

7.0.3 Supporting Slow I/O Devices

In communicating with the outside world, we need to insure that we support low-speed devices in addition to the high-speed devices. For instance, it is unlikely that the off-the-shelf Virtex or DRAM parts will be able to clock as fast as the core logic of our chip. And we may have trouble finding a RS-232 chip which clocks at 250 Mhz! As a result, the SIB protocol needs to be reexamined to see if it still operates when connected to a client with a lesser clock speed. Ideally, the SIB protocol will support a software-settable clock speed divider feature, not unlike found on DRAM controllers for PCs. It is not enough merely to program the tiles so they do not send data words off the side of the chip too frequently; the control signals will still be switching too quickly.

7.1 THE MEMORY SYSTEM

The Raw memory system is still much in flux. A number of group members are actively researching this topic. Although our goal is to do as much as possible in software, it is likely that some amount of hardware will be required in order to attain acceptable performance on a range of codes.

What I present in this section is a sketch of some of the design possibilities for a reasonable memory system. This sketch is intended to have low implementation cost, and acceptable performance.

7.1.1 The Tag Check

The Raw compiler essentially partitions the memory objects of a program across the Raw tiles. Each tile owns a fraction of the total memory space. The Raw compiler currently does this with the underlying abstraction that each Raw tile has an infinite memory. After the Raw compiler is done, our prototype memory system compiler examines the output and inserts tag checks for every load or store which it cannot guarantee resides in the local SRAM. If these tag checks fail, then the memory location must be fetched from an off-chip DRAM.

Because these tag checks can take from between 3 and 9 cycles to execute [Moritz99], the efficiency of this system depends on the compiler's ability to eliminate the tag checks. Depending on the results of this research, we may decide to add hardware tag checks to the architecture. This will introduce some complexity into the pipeline. However, the area impact will probably be neglible -- the tags will simply move out of the SRAM space into the dedicated tag SRAM. There would still be the facility to turn off the hardware tag checks for codes which do not require it, or for research purposes.

7.1.2 The Path to Copious Memory

We also need to consider the miss case. We need to have a way to reach the DRAMs residing outside of the Raw chip.This path is not as crucial as the Tag Check; however it still needs to be fairly efficient.

For this purpose, we plan to use a dynamic network to access the off-chip DRAMs. Whether this miss case is handled in software or hardware will be determined when we have more performance numbers.

7.2 SUMMARY

The strength of Raw's I/O architecture comes from the degree and simplicity with which the pins are exposed to the user as a first class resource. Just as the Raw tile expose the parallelism of the underlying silicon to the user, the Raw I/O architecture exposes the parallelism and bandwidth of the pins. It complements the key Raw goal -- to provide a simple interface to as much of the raw hardware resources to the user as possible.

8 DEADLOCK

In my opinion, the deadlock issues of the dynamic network is probably the single most complicated part of the Raw architecture. Finding a deadlock solution is actually not all that difficult. However, the lack of knowledge of the possible protocols we might use, and the constant pressure to use as little hardware support as possible makes this quite a challenge.

In this section, I describe some conditions which cause deadlock on Raw. I then describe some approaches that can be used to attack the deadlock problem. Finally, I present Raw's deadlock strategy.

8.0 DEADLOCK CONDITIONS

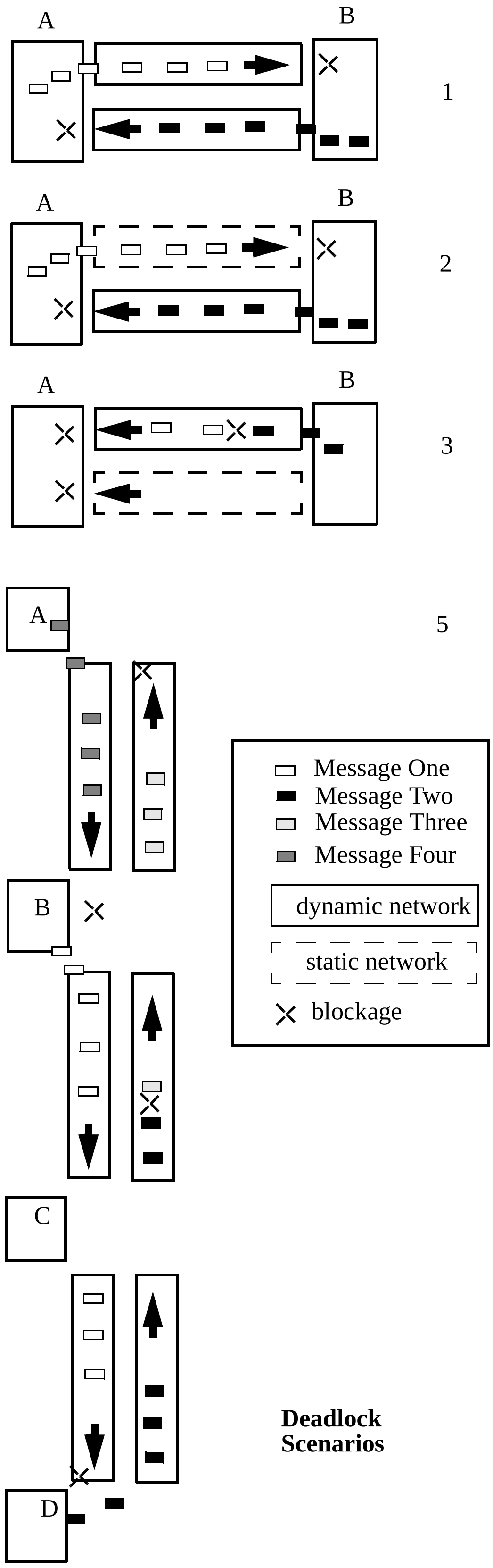
For the static network, it is the compiler's responsibility to ensure that the network is scheduled in a way that doesn't jam. It can do this because all of the interactions between messages on the network have been specified in the static switch instruction stream. These interactions are timing independent.

The dynamic network, however, is ripe with potential deadlock. Because we use dimension-ordered wormhole routing, deadlocks do not actually occur inside the network. Instead, they occur at the network interface to the tile. These deadlocks would not occur if the network had unlimited capacity. In every case, one of the tiles, call it tile A, has a dynamic message waiting at its input queue that is not being serviced. This message is flow controlling the network, and messages are getting backed up to a point where a second tile, B, is blocked trying to write into the dynamic network. The deadlock occurs when tile A is dependent on B's forward progress in order to get to the stage where it reads the incoming message and unblocks the network.

Below is an enumeration of the various deadlock conditions that can happen. Most of them can be extended to multiple party deadlocks. See the figure entitled "Deadlock Scenarios."

8.0.1 Dynamic - Dynamic

Tile A is blocked trying to send a dynamic message to Tile B. It was going to then read the message arriving from B. Tile B is blocked trying to send to Tile A. It was going to then receive from A. This forms a dependency cycle. A is waiting for B and B is waiting for A.



Deadlock Scenarios

8.0.2 Dynamic - Static

Tile A is blocked on $csto because it wants to statically communicate with processor B. It has a dynamic message waiting from B. B is blocked because it is trying to finish the message going out to A.

8.0.3 Static - Dynamic

Tile A is waiting on $csti because it is waiting for a static message from B. It has a dynamic message waiting from B.

Tile B is waiting because it is trying to send to tile C which is blocked by the message it sent to A. It was then going to write to processor A over the static network.

8.0.4 Static - Static

Processor A is waiting for a message from Processor B on $csti. It was then going to send a message.

Processor B is waiting for a message from Processor B on $csti. It was then going to send a message.

This is a compiler error on Raw.

8.0.5 Unrelated Dynamic-Dynamic

In this case, tile B is performing a request, and getting a long reply from D. C is performing a request, and getting a long message from A. What is interesting is that if only one or the other request was happening, there may not have been deadlock.

8.0.6 Deadlock Conditions - Conclusions

An accidental deadlock can exist only if at least one tile has a waiting dynamic network in-message and is blocked on either the $cdno, $csti, or $csto. Actually, technically, the tile could be polling either of those three ports. So we should rephrase that: the tile can only be deadlocked if there is a waiting dynamic message coming in and one of {$cdno is not empty, $csti does not have data available, or $csto is full}.

In all of these cases, the deadlock could be alleviated if the tile would read the dynamic message off of its input port. However, there may be some very good reasons for why the tile does not want to do this.

8.1 POSSIBLE DEADLOCK SOLUTIONS

The key two deadlock solutions are deadlock avoidance and deadlock recovery. These will be discussed in the next two sections.

8.2 DEADLOCK AVOIDANCE

Deadlock avoidance requires that the user restrict their use of the dynamic network to a certain pattern which has been proven to never deadlock.

The Deadlock avoidance disciplines that we generally arrive at are centered around two principles:

8.2.1 Ensuring that messages at the tail of all dependence chain are always sinkable.

In this discipline, we guarantee that the tile with the waiting dynamic message is always able to “sink” the waiting message. This means that the tile is always able to pull the waiting words off the network and break any cycles that have formed. The processor is not allowed to block while there are data words waiting.

These disciplines typically rely on an interrupt handler being fired to receive messages, which provides a high-priority receive mechanism that will interrupt the processor if it is blocked.

Alternatively, we could require that polling code be placed around every send.

Two examples disciplines which use that “always sinkable” principal are “Send Only” and “Remote Queues.”

Send Only

For send-only protocols; like protocols which only store values, the interrupt handler can just run and process the request. This is an extremely limited model.

Remote Queues

For request-reply protocols, Remote Queues [Chong95], relies on an interrupt handler to dequeue arriving messages as they arrive. This handler will never send messages.

If this request was for the user process, the interrupt handler will place the message in memory, and set a flag which tells the user process that data is available The user process then accesses the queue.

Alternatively, if the request is to be processed independently of the user process, the interrupt handler can drop down to a lower priority level, and issue a reply. While it does this will remain ready to pop up the higher priority level and receive any incoming messages.

Both of these methods have some serious disadvantages. First of all, the model is more complicated and adds software overhead. The user process must synchronize with the interrupt handler, but at the same time, make sure that it does not disable interrupts at an inopportune time. Additionally, we have lost that simple and fast pipeline-coupled interface that the network ports originally provided us with.

The Remote Queue method assumes infinite local memories, unless an additional discipline restricting the number of outstanding messages is imposed. Unfortunately, for all-to-all communication, each tile will have to reserve enough memory to handle the worst case -- all tiles sending to the same tile. This memory overhead can take up a significant portion of the on-tile SRAM.

8.2.2 Limit the amount and directions of data injected into the network.

The idea here is that we make sure that we never block trying to write to our output queue, making us available to read our input queue. Unless there is a huge amount of buffering in the network, this usually requires that we know a priori that there is some limit on the number of tiles that can send to us (and require replies) at any point, and that there is a limited on the amount of data in those messages. Despite this heavy restriction, this is nonetheless a useful discipline.

The Matt Frank method

One discipline which we developed uses the effects of both principles. I called it the Matt Frank method. (It might also be called the client-server method, or the two party protocol.) In this example, there are two disjoint classes of nodes, the clients and the servers, which are connected by separate “request” and “reply” networks. The clients send a message to the servers on the request network, and then the servers send a message back on the reply network. Furthermore, each client is only allowed to have one outstanding message, which will fit entirely in its commit buffer. This guarantees that it will never be blocked sending.

Since clients and servers are disjoint, we know that when a client issues a message, it will not receive any other messages except for its response, which it will be waiting to dequeue. Thus, the client nodes could never be responsible for jamming up the network.

The server nodes are receiving requests and sending replies. Because of this, they are not exempt from deadlock in quite the same way as the client nodes. However, we know that the outgoing messages are going to clients which will always consume their messages. The only possibility is that the responses get jammed up on their way back through the network by the requests. This is exactly what happened in the fifth dead-lock example given in the diagram “Deadlock Scenarios.” However, in this case, the request and reply networks are separate, so we know that they cannot interact in this way. Thus, the Matt Frank method is deadlock free.

One simple way to build separate request-reply networks on a single dimension-ordered wormhole routed dynamic network is to have all of the server nodes on a separate side of the chip; say, the south half of the chip. With X-first dimension-ordered routing, all of the requests will use the W-E links on the top half of the chip, and then the S links on the way down to the server nodes. The replies will use the W-E links on the bottom half of the chip, and the N links back up to the clients. We have effectively created a disjoint partition of the network links between the requests and the replies.

For the Matt Frank protocol, we could lift the restriction of only one outstanding message per client if we guaranteed that we would always service all replies immediately. In particular, the client cannot block while writing a request into the network. This could be achievable via an interrupt, polling, or a dedicated piece of hardware.

8.2.3 Deadlock Avoidance - Summary

Deadlock avoidance is an appealing solution to handling the dynamic network deadlock issue. However, each avoidance strategy comes with a cost. Some strategies reduce the functionality of the dynamic network, by restricting the types of protocols that can be used. Others require the reservation of large amounts of storage, or cause a low utilization of the underlying network resources. Finally, deadlock avoidance can complicate and slow down the user’s interface to the network. Care must be made to weigh these costs against the area and implementation cost of more brute-force hardware solutions.

8.3 DEADLOCK RECOVERY

An alternative approach to deadlock avoidance is deadlock recovery. In deadlock recovery, we do not restrict the way that the user employs the network ports. Instead, we have a recovery mode that rescues the program from deadlock, should one arise. This recovery mode does not have to be particularly fast, since deadlocks are not expected to be the common case. As with a program with pathological cache behaviour, a program that deadlocks frequently may need to be rewritten for performance reasons.

Before I continue, I will introduce some terminologies. These are useful in evaluating the ramifications of the various algorithms on the Raw architecture.

Spontaneous Synchronization is the ability of a group of Raw chips to suddenly (not scheduled by compiler) stop their current individual computations and work together. Normally, a Raw tile could broadcast a message on the dynamic network in order to synchronize everybody. However, we obviously cannot use the dynamic network if it is deadlocked. We cannot use the static network to perform this synchronization, because the tiles would have to spontaneously synchronize themselves (and clear out any existing data) in order to communicate over that network!

We could have a interrupting timer which is synchronized across all of the Raw tiles to interrupt all of the tiles simultaneously, and have them clear out the static network for communication. If we could guarantee that they would all interrupt simultaneously, then we could clear out the static network for more general communication. Unfortunately, this would mean that the interrupt timer would have to be a non maskable interrupt, which seems dangerous.

In the end, it may be that the least expensive way to achieve *spontaneous synchronization* is to have some sort of non-deadlocking synchronization network which does it for us. It could be a small as one bit. For instance, the MIT-Fugu machine had such a one bit rudimentary network [Mackenzie98].

Non-destructive observability requires that a tile be able to inspect the contents of the dynamic network without obstructing the computation. This mechanism could be implemented by adding some extra hardware to inspect the SIBs. Or, we could drain the dynamic network, store the data locally on the destination nodes, and have a way of virtualizing the $cdni port.

8.3.1 Deadlock Detection

In order to recover from deadlock, we first need to detect deadlock. In order to determine if a deadlock truly exists, we would need to analyze the status of each tile, and the network connecting them, looking for a cyclic dependency.

One deadlock detection algorithm follows:

The user would not be allowed to poll the network ports, otherwise, the detection algorithm would have no way of knowing of the program's intent to access the ports. The detection algorithm runs as follows: The tiles would synchronize up, and run a statically scheduled program (that uses the static network) which analyzes the traffic inside the dynamic network, and determines whether the each tile was stalled on a instruction accessing $csto, $csti, or $cdno. It can construct a dependency graph and determine if there is a cycle.

However, the above algorithm requires both *spontaneous synchronization* and *non-destructive observability.* Furthermore, it is extremely heavy-weight, and could not be run very often.

8.3.2 Deadlock Detection Approximation

In practice, a deadlock detection approximation is often sufficient. Such an approximation will never return a false negative, and ideally will not return too many false positives. The watchdog timer, used by the MIT-Alewife machine [Kubiatowicz98] for deadlock detection is one such approximation.

The operation is simple: each tile has a timer that counts up every cycle. Each cycle, if $cdni is empty, or if a successful read from $cdni is performed, then the counter is reset. If the counter hits a predefined user-specified value, then a interrupt is fired, indicating a potential deadlock.

This method requires neither *spontaneous synchronization* nor *non-destructive observability.* It also is very lightweight.

It remains to be seen what the cost of false positives is. In particular, I am concerned about the case where one tile, the aggressive producer, is sending a continuous stream of data to a tile which is consuming at a very slow rate. This is not truly a deadlock. The consumer will be falsely interrupted, and will run even slower because it will be the tile who will be running the deadlock recovery code. (Ideally, the producer would have been the one running the deadlock code.) Fugu

[MacKenzie98] dealt with these sorts of problems in more detail. At this point in time, we stop by saying that the user or compiler may have to tweak the deadlock watchdog timer value if they run into problems like this. Alternatively, if we had the spontaneous synchronization and non-destructive observability properties, we could use the expensive deadlock detection algorithm to verify if there was a true deadlock. If it was a false positive, we could bump up the counter.

8.3.3 Deadlock recovery

Once we have identified a deadlock, we need to recover from the deadlock. This usually involves draining the blockage from the network and storing it in memory. When the program is resumed, a mechanism is put in place so that when the user reads from the network port, he actually gets the values stored in memory.

To do this, we have a bit that is set which indicates that we are in this “dynamic refill” mode. A read from $cdni will return the value stored in the special purpose register, “DYNAMIC_REFILL.” It will also cause an interrupt on the next instruction, so that a handler can transparently put a new value into the SPR. When all of the values have been read out of the memory, the mode is disabled and operation returns to normal.

An important issue is where the dynamic refill values are stored in memory. When a tile’s watchdog counter goes off, it can store some of the words locally. However, it may not be expedient to allocate significant amounts of buffer space for what is a reasonably rare occurrence. Additionally, since the on-chip storage is extremely finite, in severe situations, we eventually will need to get out to a more formidable backing store. We would need spontaneous synchronization to take over the static network and attain the cooperation of other tiles, or a non-deadlocking backup network to perform this. [Mackenzie98]

8.3.4 More deadlock recovery problems

Most of the deadlock problems describe here have been encountered by the Alewife machine, which used a dynamic network for its memory system. However, those machines have the fortunate property that they can put large quantities of RAM next to each node. This RAM can be accessed without using the dynamic network. On Raw, we have a very tiny amount of RAM that can be accessed without travelling through the network. Unless we can access a large bank of memory deadlock-free, the deadlock avoidance and detection code must take up precious instruction SRAM space on the tile. Ironically, a hardware deadlock avoidance mechanism may have a lesser area cost than the equivalent software ones**.**

8.3.5 Deadlock Recovery - Summary

Deadlock recovery is also an appealing solution to handling the deadlock problem. It allows the user unrestricted use of the network. However, it requires the existence of a non-deadlockable path to memory. This can be attained by using the static network and adding the ability to spontaneously synchronize. It can also be realized by adding another non-deadlocked network.

8.4 DEADLOCK ANALYSIS

The issue of deadlock in the dynamic network is of serious concern. Our previous solutions (like the NEWS single bit interrupt network) have had serious disadvantages in terms of complexity, and the size of the resident code on every SRAM. For brevity, I have opted not to list them here.

In this section, I propose a new solution, which I believe offers extremely simple hardware, leverages our existing dynamic network code base, and solves the deadlock problem very solidly. It creates an abstraction which can be used to solve a variety of other outstanding issues with the Raw design. Since this is preliminary, the features described here are not described in the “User’s View of Raw” section of the document.

First, let us re-examine the dynamic network manifesto:

`The primary intention of the dynamic network is to support memory accesses that cannot be statically analyzed. The dynamic network was also intended to support other dynamic activities, like interrupts, dynamic I/O accesses, speculation, synchronization, and context switches. Finally, the dynamic network was the catch-all safety net for any dynamic events that we may have missed out on.`

Even now, the Raw group is very excited about utilizing deadlock avoidance for the dynamic network. We argue that we were not going to be supporting general-purpose user messaging on the Raw chip, so we could require the compiler writers and runtime system programmers to use a discipline when they use the network.

The problem is, the dynamic network is really the extension mechanism of the processor. Its strength is in its ability to support protocols that we have left out of the hardware. We are using the dynamic network for many protocols, all of which have very different properties. Modifying each protocol to be deadlock-free is hard enough. The problem comes when we attempt to run people's systems together. We then have to prove that the power set of the protocols is deadlock free!

Some of the more flexible deadlock avoidance schemes allow near-arbitrary messaging to occur. Unfortunately, these schemes often result in decreased performance, or require large buffer space.

The deadlock recovery schemes provide us with the most protocol flexibility. However, they require a deadlock-free path to outside DRAM. If this is implemented on top of the static network, then we have to leave a large program in SRAM just in case of deadlock.

8.5 THE RAW DEADLOCK SOLUTION

Thinking about this, I realized that the dynamic network usage falls into two major groups: memory accesses and essentially random unknown protocols. These two groups of protocols have vastly different properties.

My solution is to have two logically disjoint dynamic networks. These networks could be implemented as two separate networks, or they could be implemented as two logical networks sharing the same physical wires. In the latter case, one of the networks would be deemed the high priority network and would always have priority.

The high priority network would implement the Matt Frank deadlock avoidance protocol. The off-chip memory accesses will easily fit inside this framework. In this case, the processors are the "clients" and the DRAMS, hanging off the south side of the chip, are the "servers." Interrupts will be disabled during outstanding accesses. Since the network is deadlock free, and guaranteed to make forward progress, this is not a problem. This also means that we can dangle messages into the network without worry, improving memory system performance. This network will enforce a round-robin priority scheme to make sure that no tile gets starved. This network can also be used for other purposes that involve communication with remote devices and meet the requirements. For instance, this mechanism can be used to notify the tiles of external interrupts. Since the network cannot deadlock, we know that we will have a relatively fast interrupt response time. (Interrupts would be implemented as an extra bit in the message header, and would be dequeued immediately upon arrival. This guarantees that they will not violate the deadlock avoidance protocol.)

The more general user protocols will use the low-priority dynamic network, which would have a commit buffer, and will have the $cdno/$cdni that we described previously. They will use a deadlock recovery algorithm, with a watchdog deadlock detection timer. Should they deadlock, they can use the high priority network to access off-chip DRAM. In fact, they can store all of the deadlock code in the DRAM, rather than in expensive SRAM. Incidentally, the DRAMs can be used to implement *spontaneous synchronization.*

One of the nice properties that comes with having the separate deadlock-avoidance network is that user codes do not have to worry about having a cache miss in the middle of sending a message. This would otherwise require loading and unloading the message queue. Additionally, since interrupt notifications come on the high priority network, the user will not have to process them when they appear on the input queue.

8.6 THE HIGH-PRIORITY DYNAMIC NETWORK

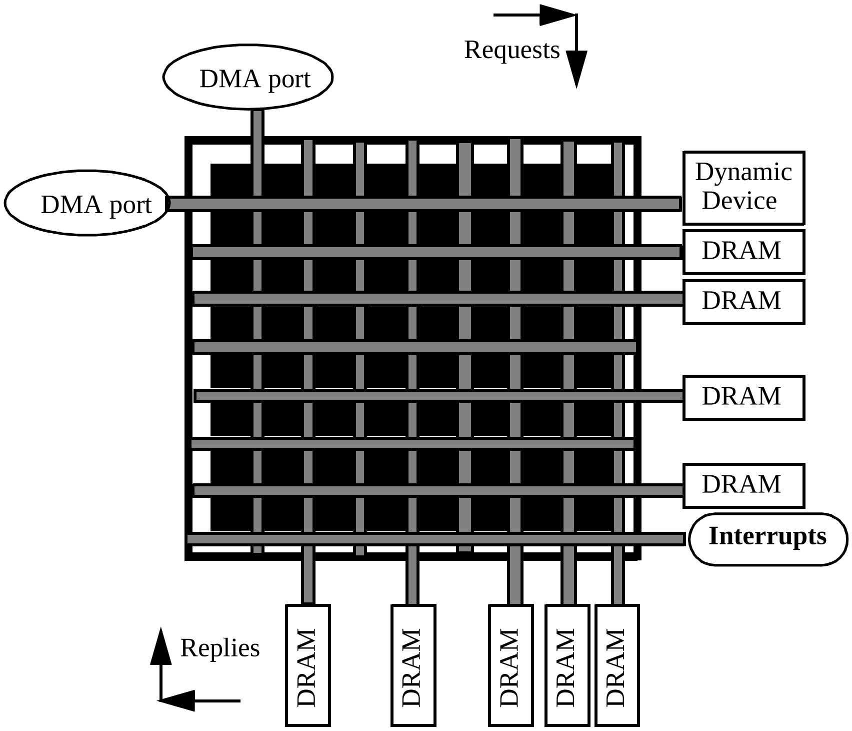
Since the low-priority dynamic network corresponds exactly to the dynamic network described in the previous dynamic network section, it does not merit further discussion.

The use of the high-priority network needs some elaboration, especially with respect to the deadlock avoidance protocol.

The diagram "High-Priority Memory Network Protocol" helps illustrate. This picture shows a Raw chip with many tiles, connected to a number of devices (DRAM, Firewire, etc.) The protocol here uses only one logical dynamic network, but partitions it into two disjoint networks. To avoid deadlock, we restrict the selection of external devices that a given tile can communicate with. For complete connectivity, we could implement another logical network. The rule for connectivity is:

Each tile is not allowed to communicate with a device which is NORTH or WEST of it. This guarantees that all requests travel on the SOUTH and EAST links, and all replies travel on the NORTH and WEST links.

Although this is restrictive, it retains four nice properties. First, it provides high bandwidth in the common case, where the tile is merely communicating with its



High Priority Memory Network Protocol

partner DRAM. The tile's partner DRAM is a DRAM that has been paired with the tile to allocate the network and DRAM bandwidth as effectively as possible. Most of the tile's data and instructions are placed on the tile's partner DRAM.

The second property, the *memory maintainer* property, is that the northwest tile can access all of the DRAMs. This will be extremely useful because the non-parallelizeable operating system code can run on that tile and operate on all of the other tile's memory spaces. Note that with strictly dimensioned-ordered routing, the memory maintainer cannot actually access all of the devices on the right side of the chip. This problem will be discussed in the "I/O Addressing" section.

The third property, the *memory dropbox* property, is that the southeast DRAM is accessible by all of the tiles. This means that non performance-critical synchronization and communication can be done through a common memory space. (We would not want to do this in performance critical regions of the program, because of the limited bandwidth to a single network port.)

These last two properties are not fundamental to the operation of a Raw processor; however they make writing setup and synchronization code a lot easier.

Finally, the fourth nice property is that the system scales down. Since all of the tiles can access the southeast-most DRAMs, we can build a single DRAM system by placing the DRAM on the southeast tile.

We also can conveniently place the interrupt notification on one of the southeast links. This black box will send a message to a tile informing it that an interrupt has occurred. The tile can then communicate with the device, possibly but not necessarily in a memory-mapped fashion. Additionally, DMA ports can be created. A device would be hooked up to these ports, and would stream data through the dynamic network into the DRAMs, and visa versa. Logically, the DMA port is just like a client tile. I do not expect that we will be implementing this feature in the prototype.

Finally, this configuration does not require that the devices have their own dynamic switches. They will merely inject their messages onto the pins, with the correct headers, and the routes will happen appropriately. This means that the edges of the network are not strictly wormhole routed. However, in terms of the wormhole routing, these I/O pins look more like another connection to the processor than an actually link to the network. Furthermore, the logical network remains partitioned because requests are on the outbound links and the replies are inbound.

8.7 PROBLEMS WITH I/O ADDRESSING

One of the issues with adding I/O devices to the periphery of the dynamic network is the issue of addressing. When the user sends a message, they first inject the destination tile number (the "absolute address"), which is converted into a relative X and Y distance. When we add I/O devices to the periphery, we suddenly need to include them in the absolute name space.

However, with the addition of the I/O nodes, the X and Y dimensions of the network are no longer powers of two. This means that it will be costly to convert from an absolute address to a relative X and Y distance when the message is sent.

Additionally, if we place devices on the left or top of the chip, the absolute addresses of the tiles will no longer start at 0. If we place devices on the left or right, the tile numbers will no longer be consecutive. For programs whose tiles use the dynamic network to communicate, this makes mapping a hash key to a tile costly.

Finally, I/O addressing has a problem because of dimension ordered routing. Because dimension ordered routing routes X, then Y, devices on the left and the right of the chip can only be accessed by tiles that are on the same row, unless there is an extra row of network that links all of the devices together.

8.8 THE "FUNNY BITS"

All of these problems could be solved by only placing devices on the bottom of the chip.

However, the "funny bits" solution which I propose allows us full flexibility in the placement of I/O devices, and gives us a unique name space.

The "funny bit" concept is simple. An absolute address still has a tile number. However, the four highest order bits of the address, previously unused, are reserved for the funny bits. These bits are preserved upon translation of the absolute address to relative address. These funny bits, labelled North, South, East, and West, specify a final route that should be done after all dimensioned ordered routing has occurred. These funny bits can only be used to route off the side of the chip. It is a programmer error to use the funny bits when to send to a tile. No more than one funny bit should be set at a time.

With this mechanism, the I/O devices no longer need to be mapped into the absolute address space. To route to an I/O device, one merely specifies the address of the tile that the I/O device is attached to, and sets the bit corresponding to the direction that the device is located at relative to the tile.

The funny bits mechanism is deadlock free because once again, it acts more like another processor attached to the dynamic network than a link on the network. A more rigorous proof will follow in subsequent theses.

An alternative to the funny bits solution is to provide the user with the ability to send messages with relative addresses, and to add extra network columns to the edge of the tile. This solution was used by the Alewife project [Kubiatowicz98]. Although the first half of this alternative seemed palatable, the idea of adding extra hardware (and violating the replicated uniform nature of the raw chip) was not.

8.9 SUMMARY

In this section, I discussed a number of ways in which the Raw chip could deadlock. I introduced two solutions, deadlock avoidance and deadlock recovery, which can be used to solve this problem.

I continued by re-examining the requirements of the dynamic network for Raw. I showed that a pair of logical dynamic networks was an elegant solution for Raw's dynamic needs.

The high-priority network uses a deadlock-avoidance scheme that I labelled the "Matt Frank protocol." Any users of this network must obey this protocol to ensure deadlock-free behaviour. This network is used for memory, interrupt, I/O, DMA and other communications that go off-chip.

The high-priority network is particularly elegant for memory accesses because, with minimal resources, it provides four properties: First, the memory system scales down. Second, the high-priority network supports ***partner memories***, which means that each tile is assigned to a particular DRAM. By doing the assignments intelligently, the compiler can divide the bandwidth of the high-priority network evenly among the tiles. Third, this system allows the existence of a ***memory dropbox***, a DRAM which all of the tiles can access directly. Lastly, it allows the existence of a ***memory***

maintainer; which means at least one tile can access all of the memories.

The low-priority network uses deadlock recovery and has maximum protocol flexibility and places few restrictions on the user. The deadlock recovery mechanism makes use of the high-priority network to gain access to copious amounts of memory (external DRAM). This memory can be used to store both the instructions and the data of the deadlock recovery mechanism, so that precious on-chip SRAM does not need to be reserved for rare deadlock events.

This deadlock solution is effective because it prevents deadlock and provides good performance with little implementation cost. Additionally, it provides an abstraction layer on the usage of the dynamic network that allows us to ignore the interactions of the various clients of the dynamic network.

Finally, I introduced the concept of “funny bits” which provides us with some advantages in tile addressing. It also allows all of the tiles to access the I/O devices without adding extra network columns.

With an effective solution to the deadlock problem, we can breath easier.

9 MULTITASKING

9.0 MULTITASKING

One of the many big headaches in processor design is enabling multitasking -- the running of several processes at the same time. This is not a major goal of the Raw project. For instance, we do not provide a method to protect errant processes from modify memory or abusing I/O devices. It is nonetheless important to make sure that our architectural constructs are not creating any intractable problems. Raw could support both spatial and temporal multitasking.

In spatial multitasking, two tiles could be running separate processes at the same time. However, a mechanism would have to be put in place to prevent spurious dynamic messages from obstructing or confusing unrelated processes. A special operating system tile could be used to facilitate communication between processes.

9.1 CONTEXT SWITCHING

Temporal multitasking creates problems because it requires that we be able to snapshot the state of a Raw processor at an unknown location in the program and restore it back later. Such a context switch would presumably be initiated by a dynamic message on the high priority network. Saving the state in the main processor would be much like saving the state of a typical microprocessor. Saving the state of the switch involves freezing the switch, and loading in a new program which drain all of the switch's state into the processor.

The dynamic and static networks present more of a challenge. In the case of the static network, we can freeze the switches, and then inspect the count of values in the input buffers. We can change the PC of the switch to a program which routes all of the values into the processor, and then out to the southeast shared DRAM over the high-priority dynamic network. Upon return from interrupt, that tile's neighbor can route the elements back into the SIBs. Unfortunately, this leaves no recourse for tiles on the edges of the chip, which do not have neighbor tiles. This issue will be dealt with later in the section.

The dynamic network is somewhat easier. In this case, we can assume command of all of the tiles so that we know that no new messages are being sent. Then we can have all of the tiles poll and drain the messages out of the network. The tiles can examine the buffer counts on the dynamic network SIBs to know when they are done. Since they can't use the dynamic network to indicate when they are done (they're trying to drain the network!) they can use the common DRAM, or the static network to do so. Upon return, it will be as if the tile was recovering from deadlock; the DYNAMIC REFILL mechanism would be used. For messages that are in the commit buffer, but have not been LAUNCHed, we provide a mechanism to drain the commit buffer.

9.1.1 Context switches and I/O Atomicity

One of the major issues with exposing the hardware I/O devices to the compiler and user is I/O atomicity. This is a problem that occurs any time resources are multiplexed between clients. For the most part, we assume that a higher-order process (like the operating system) is ensuring that two processes don't try to write the same file or program the same sound card.

However, since we are exposing the hardware to the software, there is another problem. Actions which were once performed in hardware atomically are now in software, and are suddenly not atomic. For instance, on a request to a DRAM, getting interrupted before one has read the last word of the reply could be disastrous.

The user may be in the middle of issuing a message, but suddenly get swapped out due to some sort of context switch or program exit. The next program that is running may initiate a new request with the device. The hardware device will now be thoroughly confused. Even if we are fortunate enough that it just resets and ignores the message, the programs will probably blithely continue, having lost (or gained) some bogus message words. I call this the I/O Message Atomicity problem.

There is also the issue that a device may succeed in issuing a request on one of the networks, but context switch before it gets the reply. The new program may then receive mysterious messages that were not intended for it. I call this the I/O Request Atomicity problem.

The solution to this problem is to impose a discipline upon the users of the I/O devices.

9.1.1.1 Message atomicity on the static network

To issue a message, enclose the request in an interrupt disable/enable pair. The user must guarantee that this action will cause the tile to stall with interrupts disabled for at most a small, bounded period of time.

This may entail that the tile synchronize with the switches to make sure that they are not blocked because they are waiting for an unrelated word to come through.

It also means that the message size must not overflow the buffer capacity on the way to the I/O node, or if it does, the I/O device must have the property that it sinks all messages after a small period of time.

9.1.1.2 Message atomicity on the dynamic network

If the commit buffer method is used for the high-or-low priority dynamic networks, then the message send is atomic. If the commit buffer method is not used, then again, interrupts must be disabled, as for the static network. Again, the compiler must guarantee that it will not block indefinitely with interrupts turned off. It must also guarantee that sending the message will not result in a deadlock.

9.1.1.3 Request Atomicity

Request atomicity is more difficult, because it may not feasible to disable interrupts, especially if the time between a request and a reply is long.

However, for memory accesses, it is reasonable to turn off interrupts until the reply is received, because we know this will occur in a relatively small amount of time. After all, standard microprocessors ignore interrupts when they are stalled on a memory access.

For devices with longer latencies (like disk drives!), it is not appropriate to turn off interrupts. In this case, we really are in the domain of the operating system. One or more tiles should be dedicated to the operating system. These tiles will never be context switched. The disk request can then be proxied through this OS tile. Thus, the reply will go to the OS tile, instead of the potentially swapped out user tile. The OS tile can then arrange to have the data transferred to the user's DRAM space (possibly through the DMA port), and potentially wake up the user tile so it can operate on the data.

9.2 SUMMARY

In this section, I showed a strategy which enables us to expose the raw hardware devices of the machine to the user and still support multi-tasking context switches. This method is deadlock free, and allows the user to keep the hardware in a consistent state in the face of context switches.

10 THE MULTICHIP PROTOTYPE

10.0 THE RAW FABRIC / SUPERCOMPUTER

The implementation of the larger Raw prototype creates a number of interesting challenges, mostly having due to with the I/O requirements of such a system. Ideally, we would be able to expose all of the networks of the peripheral tiles to the pins, so that they could connect to an identical neighbor chip, creating the image of a larger Raw chip. Just as we tiled Raw tiles, we will tile Raw chips! To the programmer, the machine would look exactly like a 256 tile Raw chip. However, some of the network hops may have an extra cycle of latency.

10.1 PIN COUNT PROBLEMS AND SOLUTIONS

Our package has a whopping 1124 signal pins. This in itself is a bit of a problem, because building a board with 16 such chips is non-trivial. Fortunately, our mesh topology makes building such a board easier. Additionally, the possibility of ground bounce due to simultaneously switching pins is sobering.

For the ground bounce problem, we have a potential solution which reduces the number of pins that switch simultaneously. It involves sending the negation of a signal vector in the event that more than half of the pins would change values. Unfortunately, this technique requires an extra pin for every thirty-two pins, exacerbating our pin count problem.

Unfortunately, 1124 pins is also not enough to expose all of the peripheral networks to the edges of the chip so that the chips can be composed to create the illusion of one large tile. The table entitled “Pin Count - ideal” shows the required number of pins. In order to build the Raw Fabric, we needed to find a way to reduce the pin usage.

TABLE 3. Pin Count - ideal

Purpose	**Count**
Testing, Clocks, Resets, PSROs	10
Dynamic Network Data	32x2x16
Dynamic Network Thanks Pins	2x2x16
Dynamic Network Valid Pins	1x2x16
Dynamic Network Mux Pins	1x2x16
Static Network Data	32x2x16
Static Network Thanks Pins	1x2x16
Static Network Valid Pins	1x2x16
Total	70*32+10 = 2250

We explored a number of options:

10.1.1 Expose only the static network

One option was to expose only the static network. Originally, we had opted for this alternative. However, over time, we became more and more aware of the importance of having a dynamic I/O interface to the external world. This is particularly important for supporting caching. Additionally, not supporting the dynamic network means that many of our software systems would not work on the larger system.

10.1.2 Remove a subset of the network links

For the static network, this is not a problem -- the compiler can route the elements accordingly through network to avoid the dead links.

For a dimension ordered wormhole routed network, a sparse mesh created excruciating problems. Suddenly, we have to route around the “holes”, which means that the sophistication of the dynamic network would have to increase drastically. It would be increasingly hard to remain deadlock free.

TABLE 4. Pin Count - with muxing

Purpose	**Count**
Testing, Clocks, Resets, PSROs	10
Network Data	32x2x16
Dynamic Network Thanks	2x2x16
Dynamic Network Valid	1x2x16
Mux Pins	2x2x16
Static Network Thanks	1x2x16

TABLE 4. Pin Count - with muxing

Purpose	Count
Static Network Valid Pins	1x2x16
Total	39*32+10
	= 1258

10.1.3 Do some more muxing

The alternative is to retain all of the logical links and mux the data pins. Essentially, the static, dynamic and high-priority dynamic networks all become logical channels. We must add some control pins which select between the static, dynamic and high-priority dynamic networks. See the Table entitled "Pin Count - with muxing."

10.1.4 Do some encoding

The next option is to encoding the control signals:

TABLE 5. States -- encoded

State	Value
No value	0
Static Value	1
High Priority Dynamic	2
Low Priority Dynamic	3

This encoding combines the mux and valid bits. Individual thanks lines are still required.

TABLE 6. Pin Count - with muxing and encoding

Purpose	Count
Testing, Clocks, Resets, PSROs	10
Network Data	32x2x16
Dynamic Network Thanks	2x2x16
Encoded Mux Pins	2x2x16
Static Network Thanks	1x2x16
Total	37*32+10
	= 1194

At this point, we are only 70 pins over budget. At this point, we can:

10.1.5 Pray for more pins

The fates at IBM may smile upon us and provide us with a package with even better pin counts. We're not too far off.

10.1.6 Find a practical but ugly solution

As a last resort, there are some skanky but effective techniques that we can use. We can multiplex the pins of two adjacent tiles, creating a lower bandwidth stripe across the Raw chip. Since these signals will not be coming from the same area of the chip, the latency will probably increase (and thus, the corresponding SIB buffers). Or, we can reduce the data sizes of some of the paths to 16 bits and take two cycles to send a word.

More cleverly, we can send the value over as a 16 bit signed number, along with a bit which indicates if the value fit entirely within the 16 bit range. If it did not, the other 16 bits of the number would be transmitted on the next cycle.

10.2 SUMMARY

Because of the architectural headaches involved with exposed only parts of the on chip networks, we have decided to use a variety of muxing, encoding and praying to solve our pin limitations. These problems are however, just the beginning of the problems that the multi-chip Raw system of 2007 would encounter. At that time, barring advances in optical interconnects or optical interconnects, there will have an even smaller ratio of pins to tiles. At that time, the developers will have to derive more clever dynamic networks [Glass92], or will have to make heavy use of the techniques described in the "skanky solution" category.

11 CONCLUSIONS

11.0 CURRENT PROGRESS ON THE PROTOTYPE

We are fully in the midst of the implementation effort of the Raw prototype. I have written a C++ simulator named btl, which corresponds exactly to the prototype processor that we are building. It accurately models the processor on a cycle-by-cycle basis, at a rate of about 8000 cycles per second for a 16 tile machine. My pet multi-threaded, bytecode compiled extension language, bC, allows the user to quickly prototype external hardware devices with cycle accurate behaviour. The bC environment provides a full-featured programmable debugger which has proven very useful in finding bugs in the compiler and architecture. I have also written a variety of graphic visualization tools in bC which allow the user to gain a qualitative feel of the behaviour of a computation across the Raw chip. See the Appendages entitled “Graphical Instruction Trace Example” and “Graphical Switch Animation Example.” Running `wordcount` reveals that the simulator, extension language, debugger and user interface code total 30,029 lines of `.s`,`.cc`,`.c`,`.bc`, and `.h` files. This does not include the 20,000 lines of external code that I integrated in.

(More along the lines of anti-progress, Jon Babb and I reverse-engineered the Yahoo chess protocol, and developed a chess robot which became quite a sensation on Yahoo. To date, they still believe that the Chesspet is a Russian International Master whose laconic disposition can be attributed to his lack of English. The chesspet is 1831 lines of Java, and uses Crafty as its chess engine. It often responds with a chess move before the electron gun has refreshed the screen with the opponent’s most recent move.)

Rajeev Barua and Walter Lee’s parallelizing compiler, RawCC, has been in development for about two years. It compiles a variety of benchmarks to the Raw simulators. There are several ISCA and ASPLOS papers that describe these efforts.

Matt Frank and I have ported a version of GCC for use on serial and operating system code. It uses inline macros to access the network ports.

Ben Greenwald has ported the GNU binutils to support Raw binaries.

Jason Kim, Sam Larsen, Albert Ma, and I have written synthesizeable verilog for the static and dynamic networks, and the processors. It runs our current code base, but does not yet implement all of the interrupt handling and deadlock recovery schemes.

Our testing effort is just beginning. We have Krste Asanovic’s automatic test vector generator, called Torture, which generates random test programs for MIPS processors. We intend to extend it to exert the added functionality of the Raw tile.

We also have plans to emulate the Raw verilog. We have a IKOS logic emulator for this purpose.

Jason Kim and I have attended IBM’s ASIC training class in Burlington, VT. We expect to attend the Static Timing classes later in the year.

A board for the Raw handheld device is being developed by Jason Miller.

This document will form the kernel of the design specification for the Raw prototype.

11.1 PRELIMINARY RESULTS

We have used the Raw compiler to compile a variety of applications to the Raw simulator, which is accurate to within %10 of the actual Raw hardware. However, in both the base and parallel case, the tile has unlimited local SRAM. Results are summarized below.

TABLE 7. Preliminary Results - 16 tiles

Benchmark	**Speedup versus one tile**
Cholesky	10.30
Matrix Mul	12.20
Tomcatv	9.91
Vpenta	10.59
Adpcm-encode	1.26
SHA	1.44
MPEG-kernel	4.48
Moldyn	4.48
Unstructured	5.34

More information on these results is given in [Barua99].

Mark Stephenson, Albert Ma, Sam Larsen, and I have all written a variety of hand-coded applications to gain an idea of the upper bound on performance for a

Raw architecture. Our applications have included median filter, DES, software radio, and MPEG encode. My hand-coded application, median filter, has 9 separate interlocking pipeline programs, running on 128 tiles, and attains a 57x speedup over a single issue processor, compared to the 4x speedup that a hand-coded dual-issue Pentium with MMX attains. Our hope is that the Raw supercomputer, with 256 MIPS tiles, will enable us to attain similarly outrageous speedup numbers.

11.2 EXIT

In this thesis, I have traced the design decisions that we have made along the journey to creating the first Raw prototype. I detail how the architecture was born from our experience with FPGA computing. I familiarize the reader with Raw by summarizing the programmer's viewpoint of the current design. I motivate our decision to build a prototype. I explain the design decisions we made in the implementation of the static and dynamic networks, the processor, and the prototype systems. I finalize by showing some results that were generated by our compiler and run on our simulator.

The Raw prototype is well on its way to becoming a reality. With many of the key design decisions determined, we now have a solid basis for finalizing the implementation of the chip. The fabrication of the chip and the two systems will aid us in exploring the application space for which Raw processors are well suited. It will also allow us to evaluate our design and prove that Raw is, indeed, a realizable architecture.

11.3 REFERENCES

J.L. Hennessey, **“The Future of Systems Research,”** IEEE Computer Magazine, August 1999. pp. 27-33.

D. L. Tennenhouse and V. G. Bose, **“SpectrumWare - A Software-Oriented Approach to Wireless Signal Processing,”** ACM Mobile Computing and Networking 95, Berkeley, CA, November 1995.

R. Lee, **“Subword Parallelism with MAX- 2”**, IEEE Micro, Volume 16 Number 4, August 1996, pp. 51-59.

J. Babb et al. **“The RAW Benchmark Suite: Computation Structures for General Purpose Computing,”** IEEE Symposium on Field-Programmable Custom Computing Machines, Napa Valley, CA, April 1997.

Agarwal et al. “**The MIT Alewife Machine: Architecture and Performance,”** Proceedings of ISCA ‘95, Italy, June, 1995.

Waingold et al. **“Baring it all to Software: Raw Machines,”** IEEE Computer, September 1997, pp. 86-93.

Waingold et al. **“Baring it all to Software: Raw Machines,”** MIT/LCS Technical Report TR-709, March 1997.

Walter Lee et al. **“Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine,”** Proceedings of ASPLOS-VIII, San Jose, CA, October 1998.

R. Barua et al. **“Maps: A Compiler Managed Memory System for Raw Machines,”** Proceedings of the Twenty-Sixth International Symposium on Computer Architecture (ISCA), Atlanta, GA, June, 1999.

T. Gross. **“A Retrospective on the Warp Machines,”** 25 Years of the International Symposia on Computer Architecture, Selected Papers. 25th Anniversary Issue. 1998. pp 45-47.

J. Smith. **“Decoupled Access/Execute Computer Architectures,”** 25 Years of the International Symposia on Computer Architecture, Selected Papers. 25th Anniversary Issue. 1998. pp 231-238. (Originally in ISCA 9)

W. J. Dally. **“The torus routing chip,”** Journal of Distributed Computing, vol. 1, no. 3, pp. 187-196, 1986.

J. Hennessey, and D. Patterson **“Computer Architecture: a Quantitative Approach (2nd Ed.)”**, Morgan Kauffman Publishers, San Francisco, CA, 1996.

M. Zhang. **“Software Floating-Point Computation on Parallel Machines,”** Master’s Thesis, Massachusetts Institute of Technology, 1999.

S. Oberman. **“Design Issues in High Performance Floating Point Arithmetic Units,”** Ph.D. Dissertation, Stanford University, December 1996.

E. Berlekamp, J. Conway, R. Guy, **“Winning Ways for Your Mathematical Plays,”** vol. 2, chapter 25, Academic Press, New York, 1982.

John D. Kubiatowicz. “**Integrated Shared-Memory and Message-Passing Communication in the Alewife Multiprocessor,”** Ph.D. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, February 1998.

C. Moritz et al. “**Hot Pages: Software Caching for Raw Microprocessors,”** MIT CAG Technical Report, Aug 1999.

Fred Chong et al. **“Remote Queues: Exposing Message Queues for Optimization and Atomicity,”** Symposium on Parallel Algorithms and Architecture (SPAA) Santa Barbara, July 1995.

K. Mackenzie et al. “**Exploiting Two-Case Delivery for Fast Protected Messaging.”** Proceedings of 4th International Symposium on High Performance Computer Architecture Feb. 1998.

C. J. Glass et al. **“The Turn Model for Adaptive Routing,”** 25 Years of the International Symposia on Computer Architecture, Selected Papers. 25th Anniversary Issue. 1998. pp 441-450. (Originally in ISCA 19)

12 APPENDAGES

Packaging list:

Raw pipeline diagrams

Graphical Instruction Trace Example

Graphical Switch Animation Example

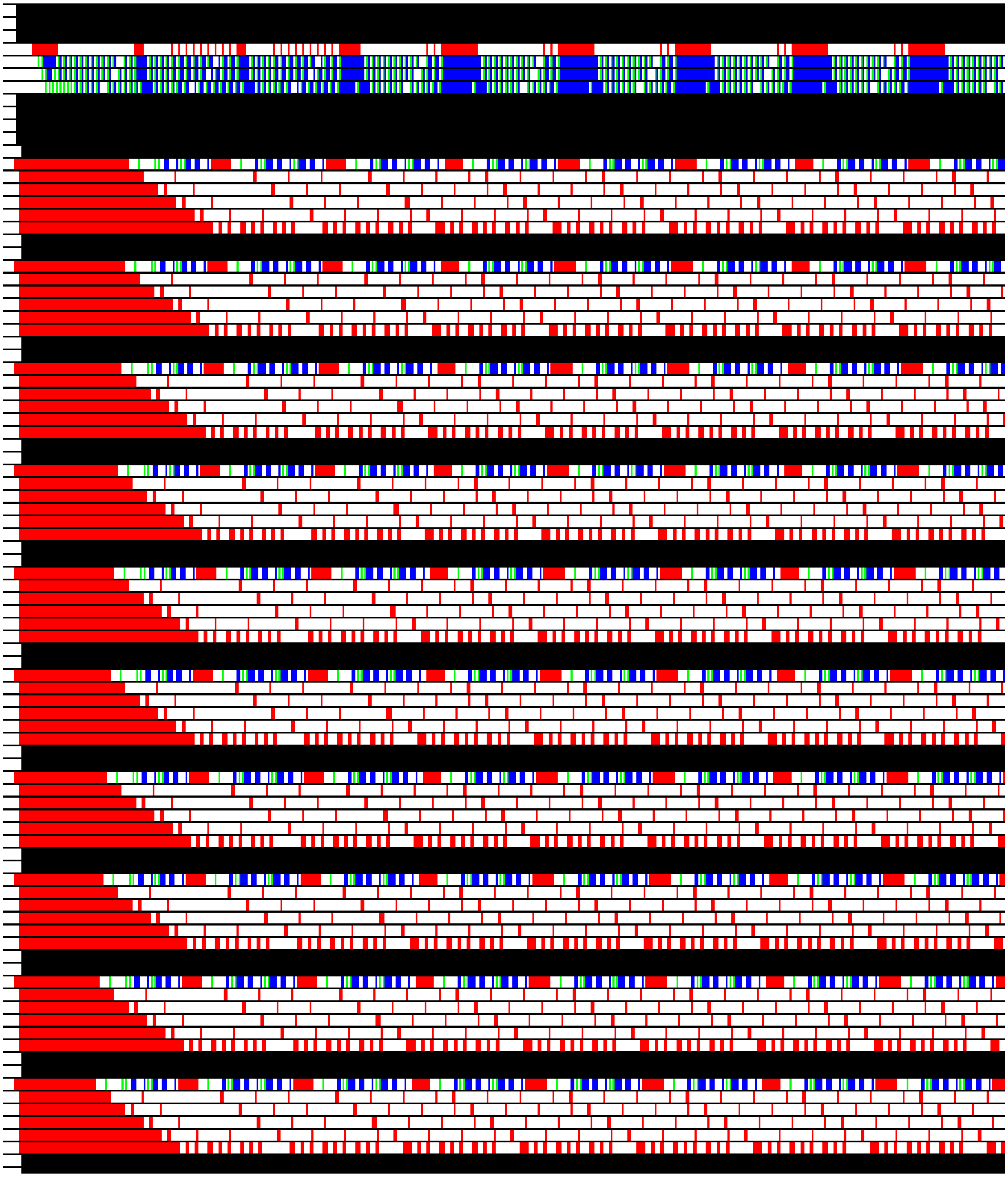
Raw user’s manual

This page intended to be replaced by printed color schematic.

This page is intended to be replaced by a printed color copy of a schematic.

This page blank, unless filled in by a printed color copy of a schematic.

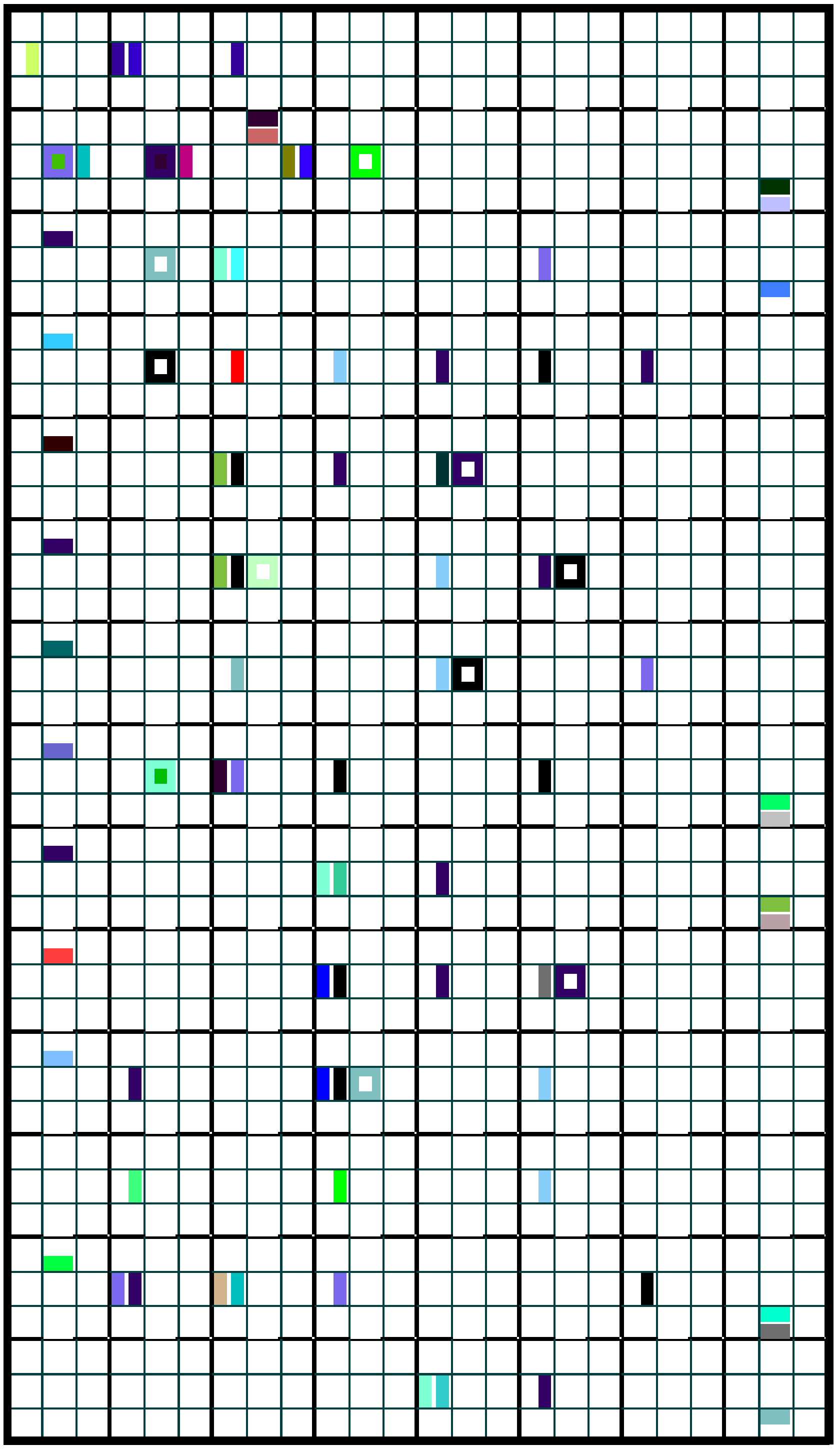
Graphical Instruction Trace Example



A section of a graphical instruction trace of median filter running on a 128 tile raw processor.
Each horizontal stripe is the status of a tile processor over ~500 cycles.
The graphic has been clipped to show only 80-odd tiles.

RED: proc blocked on $csti
BLUE: tile blocked on $csto
WHITE: tile doing useful work
BLACK: tile halted

Graphical Switch Animation Example



Shows the Data Values Travelling through the Static Switches on a 14x8 Raw processor on each cycle. Each group of nine squares corresponds to a switch. The west square corresponds to the contents of the $cWi SIB, etc. The center square is the contents of the $csto SIB.

Massachusetts Institute of Technology
Laboratory of Computer Science

RAW Prototype Chip User's Manual

Version 1.2
October 6, 1999 7:23 pm

Foreword

This document is the ISA manual for the Raw prototype processor. Unlike other Raw documents, it does not contain any information on design decisions, rather it is intended to provide all of the information that a software person would need in order to program a Raw processor. This document assumes a familiarity with the MIPS architecture. If something is unspecified, one should assume that it is exactly the same as a MIPS R2000.
(See http://www.mips.com/publications/index.html, “R4000 Microprocessor User’s Manual”.)

Processor

Each Raw Processor looks very much like a MIPS R2000.

The follow items are different:

0. Registers 24, 25, and 26 are used to address network ports and are not available as GPRs.
1. Floating point operations use the same register file as integer operations.
2. Floating point compares have a destination register instead of setting a flag.
3. The floating point branches, BC1T and BC1F are removed, since the integer versions have equivalent functionality.
4. Instead of a single multiply instruction, there are three low-latency instructions, MULH, MULHU, and MULLO which place their results in a GPR instead of HI/LO.
5. The pipeline is six stage pipeline, with FETCH, RF, EXE, MEM, FPU and WB stages.
6. Floating point divide uses the HI/LO registers instead of a destination register.
7. The instruction set, the timings and the encodings are slightly different. The following section lists all of the instructions available in the processor. There are some omissions and some additions. For actual descriptions of the standard computation instructions, please refer to the MIPS manual. The non-standard raw instructions (marked with **823**) will be described later in this document.
8. A tile has no cache and can address 8K - 16k words of local data memory.
9. cvt.w does round-to-nearest even rounding (instead of a “current rounding mode”). the trunc operation (which is the only one used by GCC) can be used to round-to-zero.
10. All floating point operations are single precision.
11. The Raw prototype is a LITTLE ENDIAN processor. In other words, if there is a word stored at address P, then the low order byte is stored at address P, and the most significant byte is stored at address P+3. (Sparc, for reference, is big endian.)
12. Each instruction has one bit reserved in the encoding, called the S-bit. The S-bit determines if the result of the instruction is written to static switch output port, in addition to the register file. If the instruction has no output, the behaviour of the S-bit is undefined. The S-bit is set by using an exclamation point with the instruction, as follows:

```
and!        $3,$2,$0                        # writes to static switch and r3
```

13. All multi-cycle non-branch operations (loads, multiplies, divides) on the raw processor are fully interlocked.

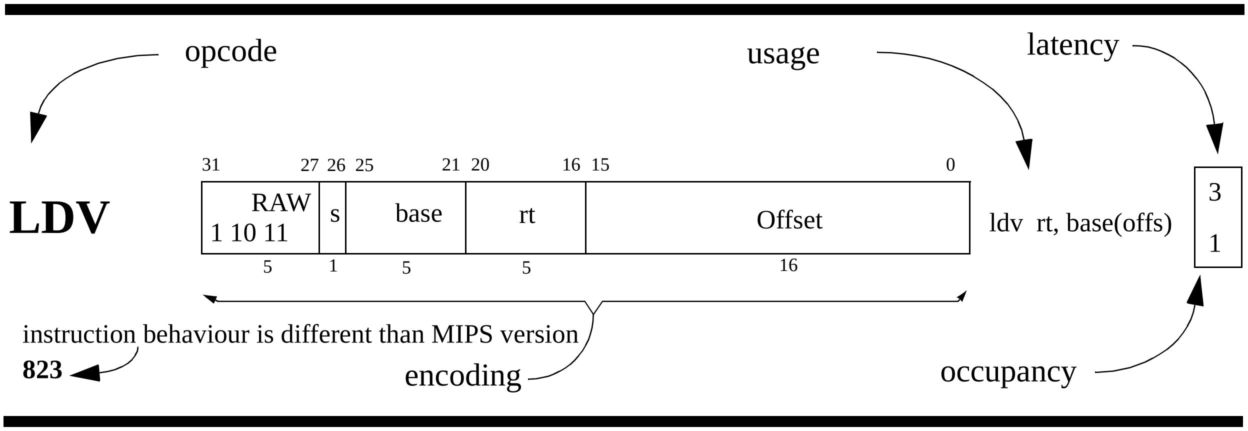
Register Conventions

The following register convention map has been modified for Raw from page D-2 of the MIPS manual). Various software systems by the raw group may have more restrictions on the registers.

Table 1: Register Conventions

reg	alias	Use
$0		Always has value zero.
$1	$at	Reserved for assembler
$2..$3		Used for expression evaluation and to hold procedure return values.
$4..$7		Used to pass first 4 words of actual arguments. Not preserved across procedure calls.
$8..$15		Temporaries. Not preserved across procedure calls
$16..$23		Callee saved registers.
$24	$csti	Static network input port.
$25	$cdn[i/o]	
$26	$cst[i/o]2	
$27		Temporary. Not preserved across procedure calls.
$28	$gp	Global pointer.
$29	$sp	Stack pointer.
$30		A callee saved register.
$31		The link register.

Sample Instruction Listing:



Integer Computation Instructions

ADDIU

31 27	26	25 21	20 16	15 0		
ADDIU 0 1 0 0 1	s	rs	rt	immediate	ADDIU rt, rs, imm	1
5	1	5	5	16		

ADDU

31 27	26	25 21	20 16	15 11	10 6	5 0		
SPECIAL 0 0 0 0 0	s	rs	rt	rd	0 0 0 0 0	ADDU 1 0 0 0 0 1	ADDU rd, rs, rt	1
5	1	5	5	5	5	6		

AND

31 27	26	25 21	20 16	15 11	10 6	5 0		
SPECIAL 0 0 0 0 0	s	rs	rt	rd	0 0 0 0 0	AND 1 0 0 10 0	AND rd, rs, rt	1
5	1	5	5	5	5	6		

ANDI

31 27	26	25 21	20 16	15 0		
ANDI 0 1 1 0 0	s	rs	rt	immediate	ANDI rs, rt, imm	1
5	1	5	5	16		

BEQ

31 27	26	25 21	20 16	15 0		
BEQ 0 0 1 0 0	s	rs	rt	offset	BEQ rs, rt, offs	2d
5	1	5	5	16		

BGEZ

31 27	26	25 21	20 16	15 0		
REGIMM 0 0 0 0 1	s	rs	BGEZ 0 0 0 1 0	offset	BGEZ rs, offs	2d
5	1	5	5	16		

BGEZAL

31 27	26	25 21	20 16	15 0		
REGIMM 0 0 0 0 1	s	rs	BGEZAL 1 0 0 1 0	offset	BGEZAL rs, offs	2d
5	1	5	5	16		

BGTZ

31 27	26	25 21	20 16	15 0		
REGIMM 0 0 0 0 1	s	rs	BGTZ 0 0 0 1 1	offset	BGTZ rs, offs	2d
5	1	5	5	16		

BLEZ

31 27	26	25 21	20 16	15 0		
REGIMM 0 0 0 0 1	s	rs	BLEZ 0 0 0 0 1	offset	BLEZ rs, offs	2d
5	1	5	5	16		

BLTZ

31 27	26	25 21	20 16	15 0		
REGIMM 0 0 0 0 1	s	rs	BLTZ 0 0 0 0 0	offset	BLTZ rs, offs	2d
5	1	5	5	16		

BLTZAL

31 27	26	25 21	20 16	15 0		
REGIMM 0 0 0 0 1	s	rs	BLTZAL 1 0 0 0 0	offset	BLTZAL rs, offs	2d
5	1	5	5	16		

BNE

31 27	26	25 21	20 16	15 0
BNE 0 0 1 0 1	s	rs	rt	offset
5	1	5	5	16

BNE rs, rt, offs

2d

DIV

31 27	26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	rs	rt	0 0 0 0 0	0 0 0 0 0	DIV 0 1 1 0 1 0
5	1	5	5	5	5	6

DIV rs, rt

36?

DIVU

31 27	26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	rs	rt	0 0 0 0 0	0 0 0 0 0	DIVU 0 1 1 0 1 1
5	1	5	5	5	5	6

DIVU rs, rt

36?

J

31 27	26	25 21	20 16	15 0
REGIMM 0 0 0 0 1	s	0 0 0 0 0	J 1 1 0 0 0	offset
5	1	5	5	16

J offs

2d

JAL

31 27	26	25 21	20 16	15 0
REGIMM 0 0 0 0 1	s	0 0 0 0 0	JAL 1 1 0 0 1	offset
5	1	5	5	16

JAL offs

2d

JALR

31 27	26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	rs	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	JALR 0 0 1 0 0 1
5	1	5	5	5	5	6

JALR rs

2d

JR

31 27	26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	rs	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	JR 0 0 1 0 0 0
5	1	5	5	5	5	6

JR rs

2d

LB

31 27	26	25 21	20 16	15 0
LB 1 0 0 0 0	s	base	rt	Offset
5	1	5	5	16

LB rt, base(offs)

3
1

LBU

31 27	26	25 21	20 16	15 0
LBU 1 0 1 0 0	s	base	rt	Offset
5	1	5	5	16

LBU rt, base(offs)

3
1

LH

31 27	26	25 21	20 16	15 0
LH 1 0 0 0 1	s	base	rt	Offset
5	1	5	5	16

LH rt, base(offs)

3
1

LHU

31 27	26	25 21	20 16	15 0
LHU 1 0 1 0 1	s	base	rt	Offset
5	1	5	5	16

LHU rt, base(offs)

3
1

LW

31 27	26	25 21	20 16	15 0
LW 1 0 0 1 1	s	base	rt	Offset
5	1	5	5	16

LW rt, base(offs)

2
1

LUI

31 27	26	25 21	20 16	15 0
LUI 0 1 1 1 1	s	0	rt	immediate
5	1	5	5	16

LUI rt, imm

1

MFHI

31 27	26	25 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	0 0 0 0 0 0 0 0 0 0	rd	0 0 0 0 0	MFHI 0 1 0 0 0 0
5	1	10	5	5	6

MFHI rd

1

MFLO

31 27	26	25 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	0 0 0 0 0 0 0 0 0 0	rd	0 0 0 0 0	MFLO 0 1 0 0 1 0
5	1	10	5	5	6

MFLO rd

1

MTHI

31 27	26	25 21	20 6	5 0
SPECIAL 0 0 0 0 0	s	rs	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	MTHI 0 1 0 0 0 1
5	1	5	15	6

MTHI rs

1

MTLO

31 27	26	25 21	20 6	5 0
SPECIAL 0 0 0 0 0	s	rs	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	MTLO 0 1 0 0 1 1
5	1	5	15	6

MTLO rs

1

MULH

823

31 27	26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	rs	rt	rd	0 0 0 0 0	MULH 1 0 1 0 0 0
5	1	5	5	5	5	6

MULH rd, rs, rt

2

The contents of register *rs* and *rt* are multiplied as signed values to obtain a 64-bit result. The high 32 bits of this result is stored into register *rd*.

Operation: $[rd] \leftarrow ([rs]*_{s}[rt])_{63..32}$

MULHU

823

31 27	26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	rs	rt	rd	0 0 0 0 0	MULHU 1 0 1 0 0 1
5	1	5	5	5	5	6

MULHU rd, rs, rt

2

The contents of register *rs* and *rt* are multiplied as unsigned values to obtain a 64-bit result. The high 32 bits of this result is stored into register *rd*.

Operation: $[rd] \leftarrow ([rs]*_{u}[rt])_{63..32}$

MULLO

823

31 27	26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	rs	rt	rd	0 0 0 0 0	MULLO 0 1 1 0 0 0
5	1	5	5	5	5	6

MULLO rd, rs, rt

2

The contents of register *rs* and *rt* are multiplied as signed values to obtain a 64-bit result. The low 32 bits of this result is stored into register *rd*.

Operation:$[rd] \leftarrow ([rs]*[rt])_{31..0}$

MULLU

823

31 27	26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	rs	rt	rd	0 0 0 0 0	MULLU 0 1 1 0 0 1
5	1	5	5	5	5	6

MULLO rd, rs, rt

2

The contents of register *rs* and *rt* are multiplied as unsigned values to obtain a 64-bit result. The low 32 bits of this result is stored into register *rd*.

Operation: $[rd] \leftarrow ([rs]*_u[rt])_{31..0}$

NOR

31 27	26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	rs	rt	rd	0 0 0 0 0	NOR 1 0 0 1 1 1
5	1	5	5	5	5	6

NOR rd, rs, rt

1

OR

31 27	26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	rs	rt	rd	0 0 0 0 0	OR 1 0 0 1 0 1
5	1	5	5	5	5	6

OR rd, rs, rt

1

ORI

31 27	26	25 21	20 16	15 0
ORI 0 1 1 0 1	s	rs	rt	immediate
5	1	5	5	16

ORI rt, rs, imm

1

SLL

31 27	26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	0 0 0 0 0	rt	rd	sa	SLL 0 0 0 0 0 0
5	1	5	5	5	5	6

SLL rd, rt, sa

1

SLLV

31 27	26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	rs	rt	rd	0 0 0 0 0	SLLV 0 0 0 1 0 0
5	1	5	5	5	5	6

SLLV rd, rt, rs

1

SLT

31 27	26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	rs	rt	rd	0 0 0 0 0	SLT 1 0 1 0 1 0
5	1	5	5	5	5	6

SLT rd, rs, rt

1

SLTI

31 27	26	25 21	20 16	15 0
SLTI 0 1 0 1 0	s	rs	rt	immediate
5	1	5	5	16

SLTI rt, rs, imm

1

SLTIU

31 27	26	25 21	20 16	15 0
SLTIU 0 1 0 1 1	s	rs	rt	immediate
5	1	5	5	16

SLTIU rt, rs, imm

1

SLTU

31 27	26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	rs	rt	rd	0 0 0 0 0	SLTU 1 0 1 0 1 1
5	1	5	5	5	5	6

SLTU rd, rs, rt

1

SRA

31 27	26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	0 0 0 0 0	rt	rd	sa	SRA 0 0 0 0 1 1
5	1	5	5	5	5	6

SRA rd, rt,sa

1

SRAV

31 27	26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	rs	rt	rd	0 0 0 0 0	SRAV 0 0 0 1 1 1
5	1	5	5	5	5	6

SRAV rd, rt, rs

1

SRL

31 27	26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	0 0 0 0 0	rt	rd	sa	SRL 0 0 0 0 1 0
5	1	5	5	5	5	6

SRL rd, rt, sa

1

SRLV

31 27	26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	rs	rt	rd	0 0 0 0 0	SRLV 0 0 0 1 1 0
5	1	5	5	5	5	6

SRLV rd, rt,rs

1

SUBU

31 27	26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	rs	rt	rd	0 0 0 0 0	SUBU 1 0 0 0 1 1
5	1	5	5	5	5	6

SUBU rd, rs, rt

1

SB

31 27	26	25 21	20 16	15 0
SB 1 1 0 0 0	s	base	rt	Offset
5	1	5	5	16

SB rt, offset(base)

1

SH

31 27	26	25 21	20 16	15 0
SH 1 1 0 0 1	s	base	rt	Offset
5	1	5	5	16

SH rt, offset (base)

1

SW

31 27	26	25 21	20 16	15 0
SW 1 1 0 1 1	s	base	rt	Offset
5	1	5	5	16

SW rt, offset(base)

1

XOR

31 27	26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0	s	rs	rt	rd	0 0 0 0 0	XOR 1 0 0 1 1 0
5	1	5	5	5	5	6

XOR rd, rs, rt

1

XORI

31 27	26	25 21	20 16	15 0
XORI 0 1 1 1 0	s	rs	rt	immediate
5	1	5	5	16

XORI rt, rs, imm

1

Floating Point Computation Instructions

ABS.s

31 27	26	25 21	20 16	15 11	10 6	5 0
FPU 1 0 1 1 1	s	rs	0 0 0 0 0	rt	fmt	ABS 0 0 0 1 0 1
5	1	5	5	5	5	6

ABS.s rd, rs, rt

3
1

ADD.s

31 27	26	25 21	20 16	15 11	10 6	5 0
FPU 1 0 1 1 1	s	rs	rt	rd	fmt	ADD 0 0 0 0 0 0
5	1	5	5	5	5	6

ADD.s rd, rs, rt

3
1

C.xx.s

31 27	26	25 21	20 16	15 11	10 6	5 0
FPU 1 0 1 1 1	s	rs	rt	rd	fmt	cond 1 1 x x x x
5	1	5	5	5	5	6

823

C.xx.s rd, rs, rt

3
1

Description: Precisely like MIPS but the result is stored in *rt*, instead of a flags register.

CVT.s.w

31 27	26	25 21	20 16	15 11	10 6	5 0
FPU 1 0 1 1 1	s	rs	0 0 0 0 0	rd	fmt	CVT.S 1 0 0 0 0 0
5	1	5	5	5	5	6

CVT.s.w rd, rt

3
1

CVT.w.s

31 27	26	25 21	20 16	15 11	10 6	5 0
FPU 1 0 1 1 1	s	rs	0 0 0 0 0	rd	fmt	CVT.W.s 1 0 0 1 0 0
5	1	5	5	5	5	6

823

CVT.w.s rd, rt

3
1

Description: Precisely like MIPS but always uses round to nearest even rounding mode.

DIV.s

31 27	26	25 21	20 16	15 11	10 6	5 0
FPU 1 0 1 1 1	s	rs	rt	0 0 0 0 0	fmt	DIV.s 0 0 0 0 1 1
5	1	5	5	5	5	6

823

DIV.s rs, rt

10?

Description: Precisely like MIPS but the result is stored in the HI register, instead of a FPR.

MUL.s

31 27	26	25 21	20 16	15 11	10 6	5 0
FPU 1 0 1 1 1	s	rs	rt	rd	fmt	MULT.s 0 0 0 0 1 0
5	1	5	5	5	5	6

MUL.s rd, rs, rt

3
1

NEG.s

31 27	26	25 21	20 16	15 11	10 6	5 0
FPU 1 0 1 1 1	s	rs	0 0 0 0 0	rd	fmt	NEG.s 0 0 0 1 1 1
5	1	5	5	5	5	6

NEG.s rd, rs

3
1

SUB.s

31 27	26	25 21	20 16	15 11	10 6	5 0
FPU 1 0 1 1 1	s	rs	rt	rd	fmt	SUB.s 0 0 0 0 0 1
5	1	5	5	5	5	6

SUB.s rd, rs, rt

3
1

TRUNC.w.s

31 27	26	25 21	20 16	15 11	10 6	5 0
FPU 1 0 1 1 1	s	rs	0 0 0 0 0	rd	fmt	TRUNC.w.s 0 0 1 1 0 1
5	1	5	5	5	5	6

TRUNC.w.s rd, rt

3
1

Floating Point Compare Options

Table 2: Floating Point Comparison Condition (for c.xxx.s)

Predicate			Relations(Results)				Invalid operation exception if unordered
Cond	Mnemonic	Definition	Greater Than	Less Than	Equal	Unordered	
0	F	False	F	F	F	F	No
1	UN	Unordered	F	F	F	T	No
2	EQ	Equal	F	F	T	F	No
3	UEQ	Unordered or Equal	F	F	T	T	No
4	OLT	Ordered Less Than	F	T	F	F	No
5	ULT	Onordered or Less Than	F	T	F	T	No
6	OLE	Ordered Less Than or Equal	F	T	T	F	No
7	ULE	Unordered or Less Than or Equal	F	T	T	T	No
8	SF	Signaling False	F	F	F	F	Yes
9	NGLE	Not Greater Than or Less Than or Equal	F	F	F	T	Yes
10	SEQ	Signaling Equal	F	F	T	F	Yes
11	NGL	Not Greater Than or Less Than	F	F	T	T	Yes
12	LT	Less Than	F	T	F	F	Yes
13	NGE	Not Greater Than or Equal	F	T	F	T	Yes
14	LE	Less Than or Equal	F	T	T	F	Yes
15	NGT	Not Greater Than	F	T	T	T	Yes

Administrative Instructions

DRET
823

31 27	26	25 16	15 11	10 6	5 0
COMM 1 1 1 1 1	s	0 0 0 0 0 0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	DRET 0 0 0 0 0 0
5	1	10	5	5	6

DRET

1

Returns from an interrupt, JUMPs through EPC.

DRET2
823

31 27	26	25 16	15 11	10 6	5 0
COMM 1 1 1 1 1	s	0 0 0 0 0 0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	DRET2 0 0 1 0 0 0
5	1	10	5	5	6

DRET2

1

Returns from an interrupt, JUMPs through ENPC, enables interrupts in EXECUTE stage.
Placed in delay slot of DRET instruction.

DLNCH
823

31 27	26	25 16	15 11	10 6	5 0
COMM 1 1 1 1 1	s	0 0 0 0 0 0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	DLNCH 0 0 0 0 0 1
5	1	10	5	5	6

DLNCH

1

Launches a constructed dynamic message into the network. See Dynamic network section for more detail.

ILW
823

31 27	26	25 21	20 16	15 0
ILW 1 0 0 1 0	s	base	rt	Offset
5	1	5	5	16

ILW rt, base(offs)

2
1

The 16-bit *offset* is sign-extended and added to the contents of *base* to form the effective address. The word at that effective address in the instruction memory is loaded into register *rt*. Last two bits of the effective address must be zero.

Operation: Addr ← ((offset$_{15}$)^{16} || offset$_{15..0}$) + [base]
[rt] ← IMEM[Addr]

ISW
823

31 27	26	25 21	20 16	15 0
ISW 1 1 0 1 0	s	base	rt	Offset
5	1	5	5	16

ISW rt, base(offs)

2
1

The 16-bit offset is sign-extended and added to the contents of base to form the effective address. The contents of *rt* are stored at the effective address in the instruction memory.

Operation: Addr ← ((offset$_{15}$)^{16} || offset$_{15..0}$) + [base]
IMEM[Addr] ← [rt]

MFSR
823

31 27	26	25 21	20 16	15 11	10 6	5 0
COMM 1 1 1 1 1	s	rs	0 0 0 0 0	rd	0 0 0 0 0	MFSR 0 1 0 0 0 0
5	1	5	5	5	5	6

MFSR rd,rs

1

Loads a word from a status register. See “status and control register” table.

Operation: [rd] = SR[rs]

MTSR
823

31 27	26	25 21	20 16	15 11	10 6	5 0
COMM 1 1 1 1 1	s	rs	rt	0 0 0 0 0	0 0 0 0 0	MTSR 0 1 0 0 0 1
5	1	5	5	5	5	6

MTSR rt ,rs

1

Loads a word into a control register, changing the behaviour of the Raw tile. See “status and control register” page.

Operation: SR[rt] = [rs]

MTSRi

823

31 27	26	25 21	20 16	15 11 10 6 5 0
MTSRi 0 1 0 0 0	s	0 0 0 0 0	rt	immediate
5	1	5	5	16

MTSRi rt ,imm

1

Loads a word into a control register, changing the behaviour of the Raw tile. See “status and control register” page.

Operation: SR[rt] = 0^{16} || imm

SWLW

823

31 27	26	25 21	20 16	15 0
SWLW 1 0 1 1 0	s	base	rt	Offset
5	1	5	5	16

SWLW rt, base(ofs)

3

1

The 16-bit *offset* is sign-extended and added to the contents of *base* to form the effective address. The word at that effective address in the switch memory is loaded into register *rt*. Last two bits of the effective address must be zero.

Operation: Addr ← ($(\mathrm{offset}_{15})^{16}$ || $\mathrm{offset}_{15..0}$) + [base]
[rt] ← SWMEM[Addr]

SWSW

823

31 27	26	25 21	20 16	15 0
SWSW 11 1 1 0	s	base	rt	Offset
5	1	5	5	16

SWSW rt, base(ofs)

3

1

The 16-bit offset is sign-extended and added to the contents of base to form the effective address. The contents of *rt* are stored at the effective address in the switch memory.

Operation: Addr ← ($(\mathrm{offset}_{15})^{16}$ || $\mathrm{offset}_{15..0}$) + [base]
SWMEM[Addr] ← [rt]

Opcode Map
This map is for the first five bits of the instruction (the “opcode” field.)

	instruction[29..27]							
	000	001	010	011	100	101	110	111
00	**SPECIAL**	**REGIMM**			BEQ	BNE		
01	MTSRI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
10	LB	LH	ILW	LW	LBU	LHU	SWLW	**FPU**
11	SB	SH	ISW	SW			SWSW	**COM**

Special Map
This map is for the last six bits of the instruction when opcode == “SPECIAL”.

	instruction[2..0]							
	000	001	010	011	100	101	110	111
000	SLL		SRL	SRA	SLLV		SRLV	SRAV
001	JR	JALR						
010	MFHI	MTHI	MFLO	MTLO				
011	MULL	MULLU	DIV	DIVU				
100		ADDU		SUBU	AND	OR	XOR	NOR
101	MULH	MULHU	SLT	SLTU				
110								
111								

REGIMM Map
This map is for the rt field of the instruction when opcode == “REGIMM.”

	instruction[18..16]							
	000	001	010	011	100	101	110	111
00	BLTZ	BLEZ	BGEZ	BGTZ				
01								
10	BLTZAL		BGEZAL					
11	J	JAL						

FPU Function map

This opcode map is for the last six bits of the instruction when the opcode field is FPU.

		instruction[2..0]							
		000	001	010	011	100	101	110	111
	000	ADD.s	SUB.s	MUL.s	DIV.s	SQRT.s ?	ABS.s		NEG.s
	001						TRUNC.s		
	010								
	011								
	100	CVT.S				CVT.W			
	101								
	110	C.F	C.UN	C.EQ	C.UEQ	C.OLT	C.ULT	C.OLE	C.ULE
	111	C.SF	C.NGLE	C.SEQ	C.NGL	C.LT	C.NGE	C.LE	C.NGT

COM Function map

This opcode map is for the last six bits of the instruction when the opcode field is COM.

		instruction[2..0]							
		000	001	010	011	100	101	110	111
	000	DRET	DLNCH						
	001	DRET2							
	010	MFSR	MTSR						
	011								
	100								
	101								
	110								
	111								

Status and Control Registers (very preliminary)

	Status Register Name		Purpose
0	FREEZE	RW	Switch is frozen (1, 0)
1	SWBUF1	R	Number of elements in switch buffers (NNN EEE SSS WWW III OOO)
2	SWBUF2	R	Number of elements in switch buffers pair 2 (nnn eee sss www iii 000)
3			
4	SW_PC	RW	Switch's PC (write first)
5	SW_NPC	RW	Switch's NPC (write second)
6			
7	WATCH_VAL	RW	32 bit Timer count up 1 per cycle
8	WATCH_MAX	RW	value to reset/interrupt at
9	WATCH_SET	RW	mode for watchdog counter (S D I)
10	CYCLE_HI	R	number of cycles from bootup (hi 32 bits) (read first)
11	CYCLE_LO	R	number of cycles from bootup (low 32 bits) (read second, subtract 1)
12			
13	DR_VAL	RW	Dynamic refill value
14	DYNREFILL	RW	Whether dynamic refill interrupt is turned on (1,0)
15			
16	D_AVAIL	R	Data Available on Dynamic network?
17			
18	DYNBUF	R	Number of sitting elements in dynamic network queue not triggered
19			
20	EPC	RW	PC where exception occurred
21	ENPC	RW	NPC where exception occurred
22	FPSR	RW	Floating Point Status Register (V Z O U I) (Invalid, Div by Zero, Overflow, underflow, Inexact Operation) These bits are sticky, ie a floating point operation can only set the bits, never clear. However, the user can both set and clear all of the bits.
23			Exception Acknowledges
24			Exception Masks
25			Exception Blockers

		Status Register Name		Purpose
	26			
	27			
	28			

These status and control registers are accessed by the MTSR and MFSR instructions.

Exception Vectors (very preliminary)

	Vector Name	Imem Addr >> 3	Purpose
0	EX_FATAL	0	Fatal Exception
1	EX_PGM	1	Fatal Program Exception Vector
2	EX_DYN	2	Dynamic Network Exception Vector
3			
4			
5			
6			
7	EX_DYN_REF	7	Dynamic Refill Exception
8	EX_TIMER	8	Timer Went Off
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			

The exceptions vectors are stored in IMEM. One of the main concerns with storing vectors in unprotected memory is that they can be easily overwritten by data accesses, resulting in an unstable machine. Since we are a Princeton architecture, however, the separation of the instruction memory from the data memory affords us a small amount of protection. Another alternative is use a register file for this purposes. Given the number of vectors we support, this is not so exciting. The ram requirements of this vectors is 2 words per vector.

Switch Processor

The switch processor is responsible for routing values between the Raw tiles. One might view it as a VLIW processor which can execute a tremendous number of moves in parallel. The assembly language of the switch is designed to minimize the knowledge of the switch microarchitecture needed to program it while maintaining the full functionality.

The switch processor has three structural components:

1. A 1 read port, 1 write port, 4-element register file.
2. A crossbar, which is responsible for routing values to neighboring switches.
3. A sequencer which executes a very basic instruction set.

A switch instruction consists of a processor instruction and a list of routes for the crossbar.
All combinations of processor instructions and routes are allowed subject to the following restrictions:

1. The source of a processor instruction can be a register or a switch port but the destination must be a register.
2. The source of a route can be register or a switch port but the destination must always be a switch port.
3. Two values can not be routed to the same location.
4. If there are multiple reads to the register file, they must use the same register number. This is because there is only one read port.

For instance,

```
MOVE     $3,$2       ROUTE  $2->$csti, $2->$cNo, $2->$cSo, $cSi->$cEo
MOVE     $3,$csto    ROUTE  $2->$csti, $2->$cNo, $2->$cSo, $cSi->$cEo
```

are legal because they read exactly one register (r2) and write one register (r3).

```
JAL      $3, myAddr  ROUTE $csto->$2
```

is illegal because the ROUTE instruction is trying to use r2 as a destination.

```
JALR     $2,$3       ROUTE $2->$csti
```

is illegal because two different reads are being initiated to the register file (r2,r3).

```
JALR     $2,$3       ROUTE $2->$csti, $cNi->$csti
```

is illegal because two different writes are occurring to the same port.

Switch Processor Instruction Set

BEQZ <rp>, ofs16

beqz $cEi, myRelTarget

823 op = 0, imm = (ofs16 >> 3), rego = <rp>

BLTZ <rp>, ofs16

bltz $cNi, myRelTarget

823 op = 1, imm = (ofs16 >> 3), rego = <rp>

BNEZ <rp>, ofs16

bneqz $2, myRelTarget

823 op = 2, imm = (ofs16 >> 3), rego = <rp>

BGEZ <rp>, ofs16

bgez $cSti, myRelTarget

823 op = 3, imm = (ofs16 >> 3), rego = <rp>

JAL <rd>, ofs16

jal $2, myAbsTarget

823 op = 4, imm = (ofs16 >> 3), rego = “none”, rdst = <rd>

JALR <rd>, <rp>

jalr $2, $cWi

823 op = 7 (“other”), ext_op = 3, rego = <rp>, rdst = <rd>

J ofs16

j myAbsTarget

823 op =5, imm = (ofs16 >> 3), rego = “none”

JR <rp>

jr $cWi

823 op =7 (“other”), ext_op = 2, rego = <rp>

MOVE <rd>, <rp>

move $1, $cNi

823 op =7 (“other”), ext_op = 0, rego = <rp>, rdst = <rd>

MUX <rd>, <rpA>,<rpB>,<rpC>

mux $1, $cNi, $cSi, $cEi

823 op =6, muxB = <rpB>, muxA = <rpA>, rego = <rpC>, rdst = <rd>

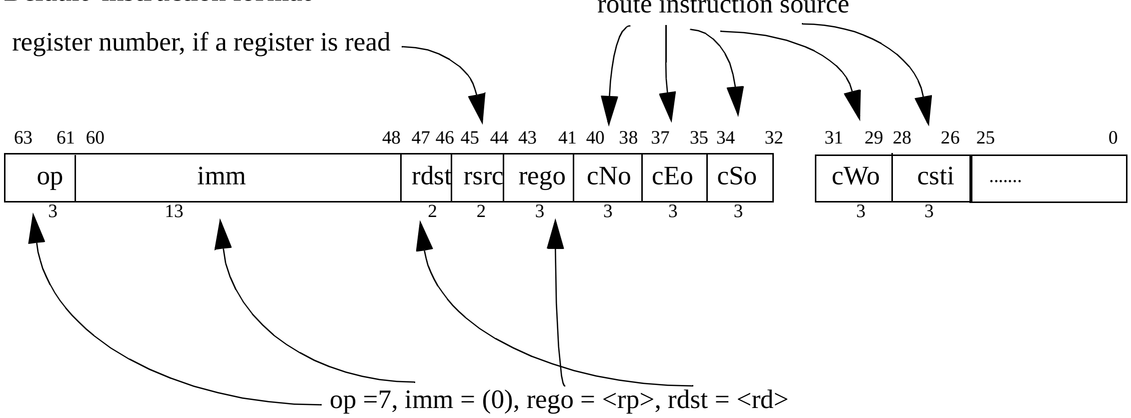
NOP

nop

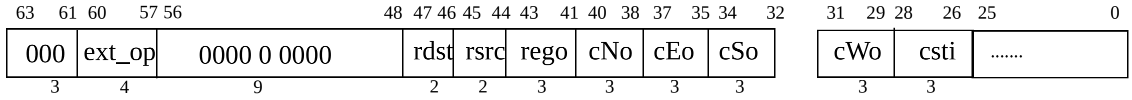
823 op =7(“other”), ext_op = 1, rego = <none>

Beastly switch instruction formats

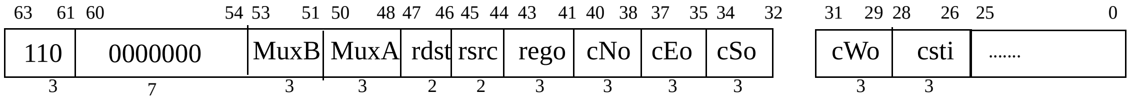
Default instruction format



“Other” instruction format



Mux instruction format



Opcode Map (bits 63..61)

	instruction[61..61]	
	0	1
00	OTHER	BLTZ
01	BNEZ	BGEZ
10	JAL	J
11	MUX	BEQZ

“Other” Map (bits 60..57)

	instruction[58..57]			
	00	01	10	11
00	NOP	MOVE	JR	JALR
01				
10				
11				

Port Name Map

	Port							
	000	001	010	011	100	101	110	111
	none	csto csti	cWi cWo	cSi cSo	cEi cEo	cNi cNo	----	regi rego

Quick Lookup by first two digits

0	**4** BNEZ	**8** JAL	**C** MUX	**00** MOVE
1 BEQZ	**5** BNEZ	**9** JAL	**D** MUX	**02** NOP
2 BLTZ	**6** BGEZ	**A** J	**E** BEQZ	**04** JR
3 BLTZ	**7** BGEZ	**B** J	**F** BEQZ	**06** JALR

Administrative Procedures

Interrupt masking

To be discussed at a later date.

Processor thread switch (does not include switch processor)

EPC and ENPC must be saved off and new values put in place. A DRET will cause an atomic return and interrupt enable.

```
mfsr     $29, EPC
sw       $29, EPC_VAL($0)
mfsr     $29, ENPC
sw       $29, ENPC_VAL($0)
lw       $29, NEW_EPC_VAL($0)
mtsr     EPC, $29
lw       $29, NEW_ENPC_VAL($0)
mtsr     ENPC, $29
dret                                # return and enable interrupt bits
lw       $29, OLD_R29($0)
```

Freezing the Switch

The switch may be frozen and unfrozen at will by the processor. This is useful for a variety of purposes. When the switch is frozen, it ceases to sequence the PC, and no routes are performed. It will indicate to its neighbors that it is not receiving any data values.

Reading or Write the Switch’s PC and NPC

In order to write the PC and NPC of the switch, two conditions must hold:

1. the switch processor must be “frozen”,
2. the SW_PC is written, followed by SW_NPC, in that order

```
# set switch to execute at address in $2

addi $3,$2,8                        # calculate NPC value
mtsri FREEZE, 1# freeze the switch
mtsr SW_PC, $2# set new switch PC to $2
mtsr SW_NPC, $3# set new switch PC to $2+8
mtsri FREEZE, 0# unfreeze the switch
```

The PC and NPC of the switch may be read at any time, in any order. However, we imagine that this operation will be most useful when the switch is frozen.

```
mtsri FREEZE, 1# freeze the switch
mfsr $2, SW_PC# get PC
mfsr $2, SW_NPC# get NPC
mtsri FREEZE, 0                     # unfreeze the switch
```

Reading or Writing the Processors's IMEM

This will stall the processor for one cycle per access. The read or write will cause the processor to stall for one cycle. Addresses are multiples of 4. Any low bits will be ignored.

```
ilw $3, 0x160($2)# load a value from the proc imem
isw $5, 0x168($2)# store a value into the proc imem
```

Reading or Writing the Switch's IMEM

The switch can be frozen or unfrozen. The read or write will cause the switch processor to stall for one cycle. Addresses are multiples of 4. Any low bits will be ignored. Note that instructions must be aligned to 8 byte boundaries.

```
swlw     $3, 0x160($2)              # load a value from the switch imem
swsw     $5, 0x168($2)              # store a value into the switch imem
```

Determining how many elements are in a given switch buffer

At any point in time, it is useful to determine how many elements are waiting in the buffer of a given switch. There are two SRs used for this purpose, SWBUF1, which is for the first set of input an output ports, and SWBUF2, which is for double-bandwidth switch implementations. The format of these status words is as follows:

SWBUF1
status reg

31 17	16 15	14 12	11 9	8 6	5 3	2 0
0	cNi	cEi	cSi	cWi	csti	csto
14	3	3	3	3	3	3

SWBUF2
status reg

31 17	16 15	14 12	11 9	8 6	5 3	2 0
0	cNi2	cEi2	cSi2	cWi2	csti2	0
14	3	3	3	3	3	3

```
# to discover how many elements are waiting in csto queue

mfsr     $2, SWBUF1                 # load buffer element counts
andi $2, $2, 0x7# get $csto count
```

Using the watchdog timer

The watchdog timer can be used to monitor the dynamic network and determine if a deadlock condition may have occurred. WATCH_VAL is the current value of the timer, incremented every cycle, regardless of what is going on in the processor.
WATCH_MAX is the value of the timer which will cause a watch event to occur:

There are several bits in WATCH_SET which determine when WATCH_VAL is reset and if an interrupt fires (by default, these values are all zero):

WATCH_SET
status reg

31	7	6	5	4	3	2	1	0
0	0	0	0	0	0	S	D	I
	1	1	1	1	1	1	1	1

	Bit	Name	effect
	0	INTERRUPT	interrupt when WATCH_VAL reaches WATCH_MAX?
	1	DYN_MOVE	reset WATCH_VAL when a data element is removed from dynamic network (or refill buffer), or if no data is available on dynamic network ?
	2	NOT_STALLED	reset WATCH_VAL if the processor was not stalled ?
	3		
	4		
	5		

```
# code to enable watch dog timer for dynamic network deadlock

mtsr WATCH_MAX, 0xFFFF              # 65000 cycles
mtsr WATCH_VAL, 0x0                 # start at zero
mtsr WATCH_SET, 0x3                 # interrupt on stall and no
                                    # dynamic network activity
jr 31
nop

# watchdog timer interrupt handler
# pulls as much data off of the dynamic network as
# possible, sets the DYNREFILL bit and then
# continues


sw       $2, SAVE1($0)              # save a reg
                                    # (not needed
                                    #  if reserved regs for handlers)
sw       $3, SAVE2($1)              # save a reg
lw       $2, HEAD($0)               # get the head index
lw       $3, TAIL($0)               # get the tail index
```

```
add      $3, $2,1
and      $3, $3, 0x1F              # thirty-one element queue
beq      $2, $3, dead              # if queue full, we need some serious work
nop
blop:
lw       $2, TAIL($0)
sw       $3, TAIL($0)              # save off new tail value
sw       $cdni, $2(BUFFER)         # pull something out of the network
mfsr     $2, D_AVAIL               # stuff on the dynamic network still?
beqz     $2, out                   # nothing on, let’s progress
lw       $2, SAVE1($0)             # restore register (delay slot)

# otherwise, let’s try to save more
move     $2, $3
add      $3, $2, 1
and      $3, $3, 0x1F              # thirty-one el queue
bne      $2, $3, blop              # if queue not full, we process another
lw       $2, SAVE1($0)             # restore register (delay slot)

out:
mtsr     DYNREFILL, 1              # enable dynamic refill
dret
lw       $3, SAVE2($1)             # restore register
```

Setting or Reading an Exception Vector

Exception vectors are instructions located at predefined locations in memory to which the processor should branch when an exceptional case occurs. They are typically branches followed by delay slots. See the Exceptions sections for more information on this.

```
ILW      $2, ExceptionVectorAddress($0)    # save old interrupt instruction
ISW      $3, ExceptionVectorAddress(40)    # set new interrupt instruction
```

Using Dynamic Refill (DYNREFILL/EX_DYN_REF/DR_VAL)

Dynamic refill mode allows us to virtualize the dynamic network input port. This functionality is useful if we find ourselves attempt to perform deadlock recovery on the dynamic network. When DYNREFILL is enabled, a dynamic read will take its value from the “DR_VAL” register and cause a EX_DYN_REF immediately after. The deadlock countdown timer (if enabled) will be reset as with an dynamic read. This will give the runtime system the opportunity to either insert another value into the refill register, or to turn off the DYNREFILL mode.

```
# enable dynamic refill

mtsri    DYNREFILL, 1               # enable dynamic refill
mtsr     DR_VAL, $2                 # set refill value
dret                                # return to user

# drefill exception vector
# removes an element off of a circular fifo and places it in DR_VAL
# if the circular fifo is empty, disable DYNREFILL
# if (HEAD==TAIL), fifo is empty
# if ((TAIL + 1) % size == HEAD), fifo is full

sw       $2, SAVE1($0)              # save a reg (not needed if
                                    #             reserved regs for handlers)
sw       $3, SAVE2($1)              # save a reg
lw       $2, HEAD($0)               # get the head index
lw       $3, $2(BUFFER)             # get next word
mtsr     DR_VAL, $3                 # set DR_VAL
add      $2, $2, 1                  # increment head index
and      $2, $2, 0xF                # buffer is 32 (31 effective) entries big
lw       $3, TAIL($0)               # load tail
sw       $2, HEAD($0)               # save new head
bne      $2,$3, out                 # if head == tail buffer is empty
lw       $2, SAVE1($0)              # restore register (delay slot)
mstri    DYNREFILL, 0               # buffer is empty, turn off DYNREFILL

out:
dret
lw       $3, SAVE2($1)              # restore register
```

Raw Boot Rom

```
# The RAW BOOT Rom
# Michael Taylor
# Fri May 28 11:53:42 EDT 1999
#
# This is the boot rom that resides on
# each raw tile. The code is identical on
# every tile. The rom code loops, waiting
# for some data show up on one of the static network ports.
# Any of North, West, East or South is fine.
# (Presumably this data is initially streamed onto the side of the
# chip by a serial rom. Once the first tile is booted, it can
# stream data and code into its neighbors until all of the tiles
# have booted.)
#
# When it does, it writes some instructions
# into the switch instruction memory which will
# repeatedly route from that port into the processor.

# At this point, it unhalts the switch, and processes
# the stream of data coming in. The data is a stream of
# 8-word packets in the following format:
#
# <imem address> <imem data> <data address> <data word>
# <switch address> <switch word> <switch word> <1=repeat,0=stop>
#
# The processor repeatedly writes the data values into
# appropriate addresses of the switch, data, and instruction
# memories.
#
# At the end of the stream, it expects one more value which
# tells it where to jump to.

.text
.set noreorder

   # wait until data shows up at the switch

sleep:
   mfsr  $2, SWBUF1    # num elements in switch buffers
   beqz  $2, sleep

   # there is actually data available on
   # the static switch now

   # we now write two instructions into switch -
   # instruction memory. These instructions
   # form an infinite loop which routes data
   # from the port with data into the processor.

   # $0 = NOP
   # $6 = JMP 0
   # $5 = ROUTE to part of instruction

   lui   $6,0xA000                  # 0xA000,0000 is JUMP 0

   # compute route instruction
   # $2 = values in switch buffers.
   lui   $7,0x0400                  # 000 001 [ten zeros]b
   lui   $5,0x1800                  # 000 110 [ten zeros]b
   sll   $3,$2,14                   # position north bits at top of word

   #
   # in this tricky little loop, we repeatedly shift the status
   # word until it drops to zero. at that point, we know that we just
   # passed the field which corresponds to the port with data available
   # as we go along, we readjust the value that we are going to write
   # into the switch memory accordingly.
   #

top:
   sll   $3,$3,3                    # shift off three bits
   bnez  $3,top                     # if it’s zero, then we fall through
   subu  $5,$5,$7                   # readjust route instruction word

setup_switch:
                                    # remember, the processor imem
                                    # is little endian
   swsw  $0,12($0)                  # 0008: NOP    route c(NW)i->csti
   swsw  $5,8($0)
   swsw  $6,4($0)                   # 0000: JMP 0  route c(NW)i->csti
   swsw  $5,0($0)

   # reload the switch’s PC.
   # (this refetch the instructions that we have written)

   MTSRi SW_PC, 0x0                 # setup switch pc and npc
   MTSRi SW_NPC, 0x8                #

   MTSRi FREEZE, 0x0                # unfreeze the switch

   # it took 19 instructions to setup the switch

   # DMA access type format
   # <imem address> <imem data> <data address> <data word>
   # <switch address> <switch word> <1=repeat,0=stop>

   or $3,$0,$csti

copy_instructions:

   isw   $csti,0($3)
   or    $4,$0,$csti
   sw    $csti,0($4)
   or    $5,$0,$csti
   swsw  $csti,0($5)
   swsw  $csti,4($5)
   bnez  $csti, copy_instructions
   or    $3,$0,$csti

stop:
   jr    $3
   nop

# it took 11 instructions to copy in the program
# and jump to it.
```