Tiled Microprocessors

by

Michael Bedford Taylor

A.B., Computer Science
Dartmouth College, 1996

S.M., Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 1999

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

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2007

Signature of Author: _____
Department of Electrical Engineering and Computer Science
February 2007

Certified by: _____
Anant Agarwal
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by: _____
Arthur C. Smith
Chairman, Departmental Graduate Committee

Tiled Microprocessors

by

Michael Bedford Taylor

Submitted to the Department of Electrical Engineering and Computer Science
on February 2007 in partial fulfillment of the
requirements for the Degree of

Doctor of Philosophy

in Electrical Engineering and Computer Science

ABSTRACT

Current-day microprocessors have reached the point of diminishing returns due to inherent scalability limitations. This thesis examines the tiled microprocessor, a class of microprocessor which is physically scalable but inherits many of the desirable properties of conventional microprocessors. Tiled microprocessors are composed of an array of replicated tiles connected by a special class of network, the Scalar Operand Network (SON), which is optimized for low-latency, low-occupancy communication between remote ALUs on different tiles. Tiled microprocessors can be constructed to scale to 100's or 1000's of functional units.

This thesis identifies seven key criteria for achieving physical scalability in tiled microprocessors. It employs an archetypal tiled microprocessor to examine the challenges in achieving these criteria and to explore the properties of Scalar Operand Networks. The thesis develops the field of SONs in three major ways: it introduces the 5-tuple performance metric, it describes a complete, high-frequency <0,0,1,2,0> SON implementation, and it proposes a taxonomy, called AsTrO, for categorizing them.

To develop these ideas, the thesis details the design, implementation and analysis of a tiled microprocessor prototype, the Raw Microprocessor, which was implemented at MIT in 180 nm technology. Overall, compared to Raw, recent commercial processors with half the transistors required 30x as many lines of code, occupied 100x as many designers, contained 50x as many pre-tapeout bugs, and resulted in 33x as many post-tapeout bugs. At the same time, the Raw microprocessor proves to be more versatile in exploiting ILP, stream, and server-farm workloads with modest to large amounts of parallelism.

Thesis Advisor:

Anant Agarwal
Professor
Electrical Engineering and Computer Science

Contents

Acknowledgments

My ten years at MIT have been a fruitful decade. For me, time is marked not in days but by the people that I meet and the systems that I work on. I've met many great people and worked on many great systems at MIT. However, as these acknowledgements will reflect, the story and the thanks start before I arrived at MIT.

Advisors and Thesis Committee

First, I'd like to thank my advisor, Anant Agarwal, for providing me with the responsibility and autonomy that I was looking for, and for providing the Raw group with the resources and mandate to build great systems. I also thank Saman Amarasinghe, co-leader of the Raw project, who was a co-advisor to me and offered both technical advice and many words of support over the years. Finally, I thank Tom Knight, the third member of my committee, who inspires me to do creative research and has patiently presided over several of my MIT exams over the years. In high school, I read Steven Levy's *Hackers*, which chronicles, among other things, Tom's arrival at MIT in the mid-sixties as a teenager. Little did I know that he would end up on my thesis committee! Tom's feedback greatly strengthened the "Related Work" section of this document.

Pre-MIT

The earliest credit must go to my family. My brother was truly my earliest computer collaborator; I still remember us pulling long hours hacking *Escape from Old Khazan* on our dad's timesharing system. My mom and dad were avid supporters of my education and interests and helped me when times were rough.

When I was in Lakeville, Rodney Page, my computer science teacher, forgave me for hacking his network and fueled my interests with his remarkable enthusiasm. There, I also met my long-time friend, Douglas S. J. DeCouto, Hacker Mac, the man with five names, who ultimately brought me to MIT. Later, Doug worked in the same office one floor below me in 545 Tech Square. Quite rudely, he started his PhD after I started and finished it years before I did. His advice and friendship has truly kept me on course through the years. In Lakefile, I also met two of my good friends, Alex and Jason Ferries, erstwhile Macintosh hackers and great friends who I later roomed with while I was in Hanover and Boston.

In Denver, I learned a lot from Jeffrey Swartz, an engineer at StorTek, who combed through

my x86 code, line by line, and hacked with me on a variety of projects. From him, I learned many valuable lessons about software engineering and other things. Amazingly, almost all of this was done over the telephone or through email.

At Dartmouth, I met Scott Silver, with whom I spent many days hacking happily. We worked on many great projects together. He and I learned much about hacking together; but from him I also learned many things about life and about *chutzpah*. Scott and I journeyed to Silicon Valley together twice and had many adventures there. Through Scott, I also met Joy Silver, who is a constant source of enthusiasm and advice. At Dartmouth, I was indebted to my educators David Kotz, Dennis Healy, Samuel Rebelsky, Thomas Cormen, and Daniela Rus, who all invested a lot of time outside of class teaching me about research and computer science.

While in Silicon Valley, I had the pleasure of working with Tim Craycroft, Bruce Jones, Eric Traut, Jorg Brown, and Chad Walters – passionate software engineers and PowerPC coders who I learned many things from.

MIT

Not surprisingly given my tenure here, the list of people who impacted me at MIT is the longest.

Jonathan Babb, whose Virtual Wires research inspired the Raw project, was a frequent partner-in-crime. Among other things, we created Russian chess robots, argued over the Raw project, and compared our wildly intersecting libraries – microprocessors and VLSI, yes, but helicopter physics, space mission design, propaganda about propaganda, and ...? Jonathan is distinguished by an ability to inspire people to seek greatness and to realize that what they thought was big could be even bigger. Beyond that, Jonathan and I had many entertaining adventures - car accidents while drag racing, teaching streetwalkers to play chess at 3 am at Dunkin' Donuts, twilight Bova Bakery runs, and of course, screening the essential and formative works in the important emerging japanese medical horror film genre.

Walter Lee, whose parallelizing compiler was used for many of the results in this thesis, was my officemate for many years. We spent countless hours hacking on elegant systems that will never see the light of publication. We both enjoyed taking engineering problems, no matter how mundane sounding, and crafting the most amazing solutions for them. Somehow, after all of the projects and papers we've worked on, I consider Walter's characterization of me as a perfectionist (in his dissertation, no less!) as a badge of honor rather than an insult.

David Wentzlaff was involved in practically every aspect of the Raw hardware prototype. Walt and I had ear-marked him as our officemate from the beginning, and it was a great choice. In addition to his brilliant ideas, enthusiasm, and technical alacrity, he provided us with no end of entertainment with his inexorable annexation of office territory through the gradual sprawl of his acquisitions from `reuse@mit`. I always look forward to running into Dave somewhere in Cambridge to have a five hour discussion about something interesting.

Matthew Frank, who was instrumental in the founding discussions of the Raw project (and indeed, authored the original 1997 Raw paper), was a compassionate counselor for many members of the Raw group. He was always there to appreciate my latest software hack (and vice versa) and provide thoughtful and motivating words.

Finally, Jonathan Eastep, my last office-mate at MIT, impressed me with his quick assimilation of practically everything about the Raw project. His enthusiasm for technology – out-gadgeting perhaps even Saman - is peerless and inspiring.

The contributions of the many members of the Raw group are obviously great and would be difficult to enumerate. No part of Raw was done in isolation. Without the effort of so many talented people, it would never have happened. To be comprehensive is impossible, but rather than just list names, I will list a few things that come to mind for each person – in all cases, a subset of what each person did. Jason Kim and I had frequent trips to Toscanini's to discuss Raw micro-architecture. We also supported each other through several week-long sessions, locked in various windowless rooms at IBM, learning that Einstimer truly was our friend. I still remember an episode with a certain Burger King hallucinogenic milkshake. IBM's ASIC team was a constant help as we iterated on the Raw ASIC design. USC ISI contributed essential software tools, benchmarks, and board design expertise. Jason Miller was my steady counterpart in pushing the Raw processor through the IBM tools; the Raw system could not have happened without his solid and unwavering engineering energy. Jason also designed the software i-caching system and was instrumental in developing the Raw motherboards. Andras Moritz examined the impact of different grain sizes in the Raw architecture. Rajeev Barua contributed MAPS, part of the Raw parallelizing compiler. Mark Seneski, master of logic emulation, ushered the Raw design through the SpecInt benchmark suite. Sam Larsen hacked on many parts of the Raw system, exuding quiet competence in every part of the toolchain he touched. Hank Hoffman and Volker Strumpen developed the Stream Algorithms for Raw, among other things. Albert Ma lent his extensive chip-building experience to the project and taught me about implementing datapaths. Jae-Wook Lee and I worked together to design the Raw placement methodology. Jae also found several pre-tapeout bugs in the design. Paul Johnson found more bugs in the Raw RTL than any other person and supported our software distribution efforts. Chris Kappler lent his extensive industry experience during the early stages of microarchitecture development. Devabhaktuni Srikrishna contributed his patient thoughtfulness to many interesting discussions. Mark Stephenson and Michael Zhang hacked on early versions of the Raw system; as roommates, our collaboration on a massive Allston party brought cops from all over. Russ Tessier offered the wisdom that comes from spending too long at MIT.

Arvind Saraf perfected the physical design for Raw's integer multiplier. Ben Greenwald contributed his fountain of knowledge and well-principled taste for elegant software engineering to many parts of the Raw infrastructure. Fae Ghodrat offered her chip-building expertise and was often there

with an upbeat and persevering attitude during the long process of building the chip and struggling with the tools. Nathan Shnidman, whose research focused on software radio for Raw, gave great, thoughtful advice and was a fantastic traveling companion through Europe. With Ian Bratt, I had many frequent discussions about high-performance compilation for Raw. James Donald, a summer visitor, and now graduate student at Princeton, hacked on a number of projects with Walter and I.

Saman Amarasinghe's Commit group members were eager collaborators and provided the StreamIt infrastructure and results, which are also featured in this thesis. In particular, the efforts of Bill Thies, Mike Gordon, Jasper Lin, and Rodric Rabbah were instrumental in the Raw results.

Outside of the Raw group, I enjoyed interacting with the members of Martin Rinard's programming systems group. Martin Rinard, Karen Zee, Viktor Kuncak, Patrick Lam, Brian Demsky, and Alex Salcianu all contributed to an intellectual office camaraderie that MIT is all about. Karen Zee also designed the logo that graces the Raw microprocessor's package. I found Maria-Cristina's friendship, unique perspective and insights invaluable to me over the years. I also enjoyed brainstorming with Enoch Peserico, whose principled perspective on both computer architecture and life was refreshing.

I also thank the MIT Scale group, including Krste Asanovic, Chris Batten, Ronny Krashinsky, Mark Hampton, Jessica Tseng, Ken Barr, Seongmoo Heo, and Albert Ma, for many interesting discussions about microprocessor design.

John Redford and Doug Carmean, top-notch microprocessor designers at Chipwrights and Intel, provided me with much helpful advice and insight over time. I ran into Carmean in the hotel bar the night before my first-ever talk (at Hotchips) and his encouraging feedback more or less got me through it.

Mario Serna and Andy Kim, my roommates at Edgerton, offered many interesting conversations about military theory and materials science, as well as fresh baked bread and a hilarious escapade involving a turkey, a bike rack, and a bout of litigation. Another one of my roommates, Kirsten Ferries, was frequently a source of outside perspective and advice. Tsoline Mikaelian, a fellow Sidney and Pacific resident, was a welcome source of brainstorming and lively debate during my thesis grind at MIT.

The staff at MIT are often underappreciated but deserve the utmost credit; in my time at MIT, I've enjoyed the support, humor and advice of Marilyn Pierce, Cornelia Colyer, Anne McCarthy, Jennifer Tucker, Simone Nakhoul and Mary McDavitt.

I would like to thank those who supported my expensive tour of duty at MIT: the American Taxpayer (through the auspices of DARPA and NSF), ITRI, Intel's PhD Fellowship program, and the Michael B. Taylor Distinguished Presidential Doctoral Fellowship Program, which financed my last year of MIT tuition.

Post-MIT

At UCSD, I've also benefited from the advice and encouragement of my colleagues; Sorin Lerner and Ranjit Jhala for their discussions on gravitas; Matthias Zwicker, who sympathized on the imperfections of taxonomy, and Steve Swanson, a fellow hawker of tiled architectures. Mohammed Al-Fares, a graduate student in my CSE 240B class, implemented a class project that led the way to the collection of the 5-tuple number for the Power 4 machine. Finally, I thank my graduate students, who provide a constant source of inspiration.

Major Stakeholders

I thank all of the members of my immediate family – my mom, my dad, John, Cliff, Ingrid, Lars, and Lauren – who have all helped me through MIT in many more ways than one.

Finally, I would like to thank Karen, one of the few remaining speakers of the ancient dialect of High Taylorian, who has gone beyond the call of duty again and again and has sustained me over the years. She is one in $18,446,744,073,709,551,616$.

MBT

February 2, 2006

Chapter 1

Introduction

Over the last thirty-five years, microprocessor designs have constantly been re-adapted to accommodate enormous advances in the properties of integrated circuit (IC) implementation technology. The most famous of these advances, referred to as Moore's Law, is the exponential growth of the number of transistors that can be placed on an integrated circuit. In 1971, the first microprocessor, the Intel 4004, employed 2,300 transistors. In 2005, the Intel Pentium 670 used 169,000,000.

Microprocessor scalability refers to the ability of a microprocessor design to exploit changing technology properties to gain additional performance or energy efficiency. Over time, it has become difficult to exploit these properties and to scale microprocessor designs effectively with existing centralized microprocessor organizations. This dissertation proposes that these scalability problems can be overcome by structuring microprocessors in a *tiled* fashion. Accordingly, microprocessors structured in this way are termed *tiled microprocessors*.

This chapter motivates the approach as follows. First, it examines the many scalability problems that centralized microprocessor designs encounter as a result of growth in IC resources and the increasing importance of wire delay (Section 1.1). Second, the chapter argues that these scalability challenges can be overcome by implementing *physically scalable* microprocessors (Section 1.2). Third, the chapter introduces seven key criteria for attaining this physical scalability. Finally, the chapter details how tiled microprocessor designs fulfill the seven criteria and enable future microprocessor scalability (Section 1.3). The chapter concludes with an overview of the thesis and its contributions.

1.1 Emerging Issues in Microprocessor Design

1.1.1 VLSI Resources

Since the days of the first microprocessor, the Intel 4004, the central drive of microprocessor research has been the adaptation of microprocessor designs to improvements in semiconductor fabrication

technology. The most significant of these improvements have been the exponential increases in the quantities of three types of VLSI resources: *transistors*, *wires*, and *pins*. However, with these improvements have come the negative effects of *wire delay*. This thesis will refer generally to these changes as the *VLSI scaling* properties.

Transistors The number of transistors that can be integrated onto a VLSI chip has been increasing exponentially for 40 years. This trend is referred to as Moore's Law [83]. This trend is evident in Table 1.1 which contrasts the 2,300 transistor 1971-era Intel 4004 microprocessor [55, 54, 32] against the 169,000,000 transistor 2005-era Intel Pentium 670.

Intel Microprocessor	Feature Size (nm)	Transistors	Frequency (MHz)	Pins	Wire Bandwidth (Est., Tracks)	Wire Delay (Est.) (clk cycle/cm)
4004 (1971)	10000	2,300	0.750	16	190	1/1000
Pentium 670 (2005)	90	169,000,000	3,800	775	125,000	4
Improvement	$\sim$ 100x	$\sim$ 75,000x	$\sim$ 5,000x	$\sim$ 50x	$\sim 660x$	1/4000x

Table 1.1: Microprocessor Properties in 1971 and in 2005

The size reduction of transistors carries with it side effects, a phenomenon that MOSFET (metal-oxide-semiconductor field effect transistor) scaling theory [25, 24] attempts to elucidate for integrated circuits. The transistor size reduction brought on by each successive process generation[1] brings proportionally lower delay and smaller power consumption per transistor; however, it also presents a greater burden of difficulty in designing the circuits that exploit these properties. Table 1.1 shows the great improvement in frequency of the Pentium 670 versus the Intel 4004. A significant portion of this (100x out of 5000x) comes from the lower delay due to transistor scaling.

Wires and Pins Of course, VLSI (very large scale integration) chips do not include just transistors; they include *wires*, for connecting the transistors within the chip, and *pins*, for connecting the integrated circuit to the outside world. The quantity of these resources available to the microprocessor designer have also increased substantially over the years. The pin-count increase for microprocessors is also evident in Table 1.1.

Wire Delay One of the properties which has not improved as favorably with successive process generations has been wire delay [13, 79]. Wire delay has stayed mostly constant (in $\frac{cm}{s}$, for instance) across process generations. Historically, wires were so fast relative to transistors that they appeared to provide an almost instantaneous connection between their endpoints. Unfortunately, as transistor frequencies have improved with each process generation, the relative speed of the wires (measured in terms of transistor delay) has worsened exponentially. While they once barely impacted processor

[1]The semiconductor industry tries to scale the minimum feature size of transistors by approximately .7 every two to three years. The term *process generation* is used to refer to the group of VLSI parameters that is employed to create integrated circuits of a given minimum feature size.

operating frequencies, wires now consume a significant fraction of the cycle time in the critical paths of current microprocessors. Wire delay has become a central concern for microprocessor designers [2], and significant effort has been expended by the semiconductor community to mitigates its effect through better materials (such as copper wires and low-K dielectrics) and better computer aided design (CAD) tools. There is evidence that wire delay will worsen in coming process generations because of complications in manufacturing wires [45], and because the wires are becoming small relative to the mean-free path of electrons in conductors.

1.1.2 Putting VLSI Resources to Use

The exponentially increasing availability of transistors, pins and wires according to VLSI scaling drives us to create microprocessors that put these resources to effective use.

In the early days of microprocessor design, many of these additional resources were expended to make microprocessors more usable; for instance, extending datapaths to accommodate reasonable address and data widths, adding floating points units to facilitate scientific calculation, and incorporating operating system protection mechanisms. Today, relatively few transistors are expended on improving microprocessor usability. Architects, recognizing that many features are better left implemented in C than in silicon, have focused on expending these additional resources to improve performance.

This focus led to the incorporation of microarchitectural innovations which exist only for performance reasons, such as caching, dynamic branch predictors, pipelining, wider memory interfaces, and superscalar and out-of-order execution. These microarchitectural approaches have the positive aspect that they improve performance without exposing to the programmer or compiler the fact that the microprocessor is utilizing exponentially more resources.

1.1.3 Problems with Addressing Scalability through Microarchitecture

Unfortunately, addressing scalability solely through microarchitectural mechanisms has led to a number of problems:

Problem 1. Diminishing Performance Returns on VLSI Resources Microarchitectural approaches have reached the point of diminishing returns. Today's wide-issue microprocessor designers have found it increasingly difficult to convert burgeoning silicon resources into usable, general-purpose functional units. The percentage of transistors that are performing actual programmer-specified computation has been steadily declining in successive versions of these modern designs. An extreme example of this can be found by comparing the 2004-era Intel 32-bit Xeon 4M to the 1989-era 32-bit Intel 80486. The Xeon has 238 times more transistors than the 80486 [51] but can sustain only three general purpose operations per cycle versus the 80486's one operation per cycle.

At the root of these difficulties are the centralized structures responsible for orchestrating operands between the functional units. These structures grow in size much faster asymptotically than the number of functional units. The prospect of microprocessor designs for which functional units occupy a disappearing fraction of total area is unappealing but tolerable. Even more serious is the poor frequency-scalability of these designs; that is, the unmanageable increase in logic and wire delay as these microprocessor structures grow [2, 92, 91, 121]. A case in point is the Itanium 2 processor, which sports a zero-cycle fully-bypassed 6-way issue integer execution core. Despite occupying less than two percent of the processor die, this unit spends half of its critical path in the bypass paths between the ALUs [85].

Scalability issues in microprocessors extend beyond the components responsible for functional unit bypassing. Contemporary processor designs typically contain unscalable, global, centralized structures such as fetch units, multi-ported register files, load-store queues, and reorder buffers. Furthermore, future scalability problems lurk in many of the components of the processor responsible for naming, scheduling, orchestrating and routing operands between functional units [92].

Problem 2. Large and Growing Design and Validation Effort Microarchitectural approaches to scalability have led to exponentially increasing design complexity. Intel's Pentium 4 required on the order of 500 people to design, and consisted of over 1 million lines of RTL code [12]. Of those people, 70 worked on verification. 7,855 bugs were found in the design before silicon and 101 errata were detected in the released product [11]. The number of pre-silicon bugs has approximately tripled for each of the last three significant microarchitectural revisions of the Intel IA32 architecture (from Pentium to Pentium Pro to Pentium 4) [97]. Correspondingly, the number of post-silicon errata continues to rise with each major release [49, 52, 53], despite advances in verification methodology and the use of larger verification groups.

Problem 3. Low Power Efficiency Microprocessor designers have recently found that the rising thermal dissipation of high performance microprocessors has affected these systems' applicability even in non-mobile environments. High thermal dissipation delayed the introduction of early Itanium II and Pentium 4 processsors in dual-processor 1U rack-mount configurations. Heat production also affects the cost of these systems: the price of an appropriate cooling solution increases drastically with rising microprocessor temperatures. These power concerns led the Pentium 4 architects to include a thermal monitor and a mechanism to stop the processor's clock in order to prevent overheating [39].

Current wide-issue superscalar processor designs use large, centralized microarchitectural structures to extract parallelism from serial instruction streams. These microarchitecture features are inherently energy-inefficient because they add overhead to the execution of every instruction. Furthermore, these designs have undesireable power density properties – because they tend to cluster like components together (such as the execution units), which causes high power consumption com-

ponents to be situated near to each other. Complex microarchitectures also prove a difficult challenge for energy optimization through clock-gating because they consist of many interdependent components, each of which must be examined carefully to determine the cases in which the component is not required for correct operation. Additionally, in cases where large structures, such as multiported register files or instruction scheduling windows are not being fully used, it adds significant complexity to add the ability to "partially" shut them off. Finally, because superscalar processors do not exploit parallelism efficiently and scalably, they are not well-equiped to save power by trading off frequency (and thus voltage and energy through voltage-scaling) with parallelism.

Problem 4. Lack of Performance Transparency The increase in microarchitectural complexity has resulted in the loss of *performance transparency* which confounds the ability of the programmer or compiler to generate code that runs efficiently on a microprocessor. Unlike in the past, the execution time (and order) of a program on modern-day microprocessors cannot be determined simply from instructions and their register dependences. One of the biggest culprits has been the introduction of various forms of hidden state in the processor, such as reorder buffers, dynamic branch predictors, way predictors, memory dependence predictors, branch target buffers, prefetch engines, and hierarchical TLBs. This state can significantly impact performance, and in general, may be left as a trade secret by the designers. To the end goal of overcoming these transparency problems, Intel has written an optimization guide for the Pentium 4 [50]. Even at 331 pages, the guide is incomplete; for instance, it fails to mention the presence of the P4's L1 data cache way-predictor [15] and its related performance effects.

In fact, the performance of applications is so much an emergent property that the Pentium 4 underwent a late revision that introduced an infrequently-invoked *cautious mode*, which overrides one of the predictors in order to address a performance problem on an important Windows application. This problem only manifested itself on 2 other traces out of 2200 [15]. Of course, these transparency problems are not limited to the Pentium 4; however due to its high profile, more data has emerged. A recent MIT thesis [69] noted a number of similar performance transparency issues in the Ultrasparc.

Problem 5. Limited Design Adaptivity One desirable characteristic of a microprocessor design is that it be easy to modify to accommodate different levels of VLSI resources and different operating constraints. This allows the design to target a greater market, which helps amortize the large initial design cost. Current microprocessor designs are relatively fixed for a given amount of VLSI resources. For instance, it is typically extremely difficult to add more general purpose functional units to a given microprocessor design. Because current microprocessor designs are not easily changed to more and fewer resources, they can not be easily adapted for use in other operating environments with different cost, power and size budgets. For example, most desktop superscalar designs like the Pentium 4 could not be made suitable for use in the MP3, handheld or mobile

telphone market. Even when these designs are deployed in the server and laptop markets, adaptations are limited, for instance, to cache configurations, bus interfaces and sleep functionality.

Furthermore, the complexity of these designs makes them brittle to technology changes. For instance, Intel had to write off its investment in the Pentium 4 architecture far earlier than planned because of the emerging energy limitations of the latest VLSI process generations.

1.2 Addressing Scalability through Physically Scalable Microprocessor Designs

This thesis examines the idea of using a *physically scalable microprocessor* design to overcome these five problems. A physically scalable microprocessor is one that can be modified in a straightforward mannner to accommodate more or fewer underlying VLSI resources (such as transistors, wires, and pins), thereby providing proportionally more or fewer gainfully employable computational resources to the user. Such a design would address the problems described in the previous section. It solves Problems 2 (design and verification complexity) and 5 (design adaptivity) because it by definition can be easily modified to accommodate different levels of resources. It solves Problems 1 (diminishing returns) and 3 (power efficiency) because it provides usable processing resources proportional to the underlying resources. Finally, Problem 4 (performance transparency) can be addressed if VLSI resources are exposed through programmer-visible architectural structures rather than through hidden microarchitectural mechanisms.

1.2.1 Seven Criteria for a Physically Scalable Microprocessor Design

What properties would a physically scalable microprocessor design have? Surprisingly, a physically scalable microprocessor combines elements of existing microprocessor designs with elements traditionally found in scalable multiprocessor systems. In a sense, scalable microprocessors generalize existing microprocessors by incorporating multiprocessor design principles. Inherited from microprocessors is the concept that scalable microprocessors should be optimized for the communication of scalar values between the computational logic elements, or ALUs, and that they need to support traditional mechanisms such as interrupts, branches, and caching. At the same time, these scalable microprocessors must, like multiprocessors, be *frequency scalable* and *bandwidth scalable*, and manage exceptional conditions such as *deadlock and starvation* that arise from being implemented in a distributed fashion. We would also like them to offer *usage-proportional resource latencies* and to support the *exploitation of locality*. These properties ("The Seven Criteria For Physical Scalability") are discussed next.

CRITERION 1 ("C1"): Frequency Scalability Frequency scalability describes the ability of a design to maintain high clock frequencies as the design is adapted to leverage more VLSI

resources. In other words, we do not want the frequencies of these designs to drop as they get bigger. When an unpipelined, two-dimensional VLSI structure increases in area, speed of light limitations dictate that the propagation delay of this structure must increase asymptotically at least as fast as the square root of the area. More practically speaking, the increase in delay is due to both increased interconnect[2] delay and increased logic levels. If we want to build larger structures and still maintain high frequencies, a physically scalable design must pipeline the circuits and turn the propagation delay into pipeline latency. Scalability in this context thus refers to both individual components ("intra-component") and to collections of components ("inter-component") within a microprocessor.

Intra-component frequency scalability As the issue width of a microprocessor increases, monolithic structures such as multi-ported register files, bypassing logic, selection logic, and wakeup logic grow linearly to cubically in size. Although extremely efficient VLSI implementations of these components exist, their burgeoning size guarantees that intra-component interconnect delay will inevitably slow them down. Put another way, these components have an asymptotically unfavorable growth function that is partially obscured by a favorable constant factor. As a result, physically scalable microprocessors cannot contain monolothic, unregistered components that grow in size as the system scales. The Itanium 2 is an example of a microprocessor that is pushing the limits of frequency scalability: it has twice the number of general-purpose functional units as the Pentium 4 but the cycle-time is twice as great.

Inter-component frequency scalability Frequency scalability is a problem not just within components, but between components. Components that are separated by even a relatively small distance are affected by the substantial wire delays of modern VLSI processes. This inherent delay in interconnect is a central issue in multiprocessor designs and is now becoming a central issue in microprocessor designs. Two recent examples of commercial architectures addressing inter-component delay are the Pentium IV, which introduced two pipeline stages that are dedicated to the crossing of long wires between remote components; and the Alpha 21264, which has a one cycle latency cost for results from one integer cluster to be used by the other cluster. Once interconnect delay becomes significant, high-frequency systems must be designed out of components that operate with only partial knowledge of what the rest of the system is doing. In other words, physically scalable microprocessors need to be implemented as a collection of distributed processes. *If a component depends on information that is not generated by a neighboring component, the architecture needs to assign a time cost for the transfer of this information.* Non-local information includes the outputs of physically remote ALUs, stall signals, branch mispredicts, exceptions, and the existence of memory dependencies.

CRITERION 2 ("C2"): Bandwidth Scalability While many frequency scalability problems

[2]We use this term loosely to refer to the set of wires, buffers, multiplexers and other logic responsible for the routing of signals within a circuit.

can be addressed by distributing centralized structures and pipelining paths between distant components, there remains a subclass of scalability problems which are fundamentally linked to a microprocessor's underlying architectural algorithms. These *bandwidth scalability* problems occur when the amount of information that needs to be transmitted and processed grows disproportionately with the size of the system.

Physically scalable microprocessors inherit the bandwidth scalability challenge from multiprocessor designs. One key indicator of a non-bandwidth scalable architecture is the use of broadcasts that are not directly mandated by the computation. For example, superscalars often rely on global broadcasts to communicate the results of instructions to consuming reservation stations. Because every functional unit's result must be processed by every reservation station, the demands on an individual reservation station grow as the system scales. Although, like the Alpha 21264, superscalars can be pipelined and partitioned to improve frequency scalability, this is not sufficient to overcome the substantial area, latency and energy penalties due to poor bandwidth scalability.

To overcome this problem, physically scalable microprocessors must employ a method of decimating the volume of messages sent in the system. Directory-based cache-coherent multiprocessors provide insight into a potential solution to this problem: employing directories to eliminate the broadcast inherent in snooping cache systems. Directories are distributed, known-ahead-of-time locations that contain dependence information. The directories allow the caches to reduce the broadcast to a unicast or multicast to only the parties that need the information. The resulting reduction in necessary bandwidth allows the broadcast network to be replaced with a point-to-point network of lesser bisection bandwidth[3].

A directory scheme is one candidate for replacing broadcasts in a scalable microprocessor's SON and achieving bandwidth scalability. The source instructions can look up destination instructions in a directory and then multicast output values to the nodes on which the destination instructions reside. In order to be bandwidth scalable, such a directory must be implemented in a distributed, decentralized fashion. This thesis will show one such implementation, and examines some alternatives.

CRITERION 3 ("C3"): Usage-Proportional Resource Latencies The ability to provide usage-proportional resource latencies is a subtle but fundamental criteria for physically scalable microprocessors. Architectures with *usage-proportional resource latencies* offer inter-resource latencies that are proportional to the amount of resources used in a computation rather than the total quantity of resources in the system. Without this property, as the design is scaled up, existing programs with small amounts of parallelism would slow down due to increased latencies. More generally, this

[3]We define bisection bandwidths with a sequence of definitions. First, a *cut* of the network is a separation of the nodes of the network into two equal sized groups. The *min cut* is the cut which minimizes the sum of the edge weights of edges that connect the two groups. Finally, the *bisection bandwidth* is the edge weight sum corresponding to the min cut of network graph in which the edges weights correspond to the bandwidths of individual links.

criteria allows the compiler to make tradeoffs between using fewer resources at a lower average latency (e.g., a subset of the machine), or greater numbers of resources at a higher average latency. In general, dancehall-style resource topologies cannot provide usage-proportional resource latencies, while those with recursively substructured (such as trees or meshes) resource topologies are more easily able to.

CRITERION 4 ("C4"): Exploitation of Locality In addition to providing usage-proportional resource latencies, physically scalable architectures should connect resources via interconnect which offers communication latencies that reflect the underlying physical communication costs (i.e., wire delay) in the implementation substrate. To complement these networks, physically scalable microprocessors must provide architectural mechanisms that allow the program to exploit positional[4] locality between the program's virtual objects (e.g., data, instructions, operations, operands) in order to minimize latency and transport occupancy. Hardware or software mechanisms must be provided to enable related objects (e.g., communicating instructions) to be placed close together in order to avoid paying worst-case costs for communication.

CRITERION 5 ("C5"): Efficient Inter-node Operation-Operand Matching The most fundamental criteria for any microprocessor is that it perform *operation-operand matching*. Operation-operand matching is the process of gathering operands and operations to meet at some point in space to perform the desired computation. In the context of a physically scalable microprocessor, *operation-operand matching* must be performed across distributed ALUs, which can be more challenging than for standard centralized designs.

Efficient inter-node operation-operand matching is what distinguishes a physically scalable microprocessor from other types of distributed systems, such as multiprocessors, which have relatively high overheads for inter-node operation-operand matching.

This thesis introduces the term *scalar operand network* ("SON") to refer to the set of mechanisms in a microprocessor that implement operation-operand matching. These mechanisms include the physical interconnection network used to transport the operands (referred to as the *transport network*) as well as the operation-operand matching system (hardware or software) which coordinates these values into a coherent computation. This choice of definition parallels the use of the word "Internet" to include both the physical links and the protocols that control the flow of data on these links.

SONs have evolved as microprocessors have evolved – from early, monolithic register file interconnects to more recent ones that incorporate point-to-point mesh interconnects. Section 2.4.1 examines this evolution in detail.

This thesis examines SONs in depth, as they are the central component of any physically scalable

[4] I would use the term spatial locality, but the semantic nuances of the term (as currently employed in the context of caches) are different.

microprocessor.

CRITERION 6 ("C6"): Deadlock and Starvation Management Superscalar SONs use relatively centralized structures to flow-control instructions and operands so that internal buffering cannot be overcommitted. With less centralized SONs, such global knowledge is more difficult to attain. If the processing elements independently produce more values than the SON has storage space, then either data loss or deadlock must occur [29, 102]. This problem is not unusual; in fact some of the earliest large-scale SON research – the dataflow machines – encountered serious problems with the overcommitment of storage space and resultant deadlock [6]. Alternatively, priorities in the operand network may lead to a lack of fairness in the execution of instructions, which may severely impact performance. Transport-related deadlock can be roughly divided into two categories; endpoint deadlock, resulting from a lack of storage at the endpoints of messages, and in-network deadlock, which is deadlock inside the transport network itself. Because effective solutions for in-network deadlock have already been proposed in the literature, endpoint deadlock is of greater concern for SONs.

CRITERION 7 ("C7"): Support for Exceptional Events Exceptional events, despite not being the common case, play a prominent role in microprocessor design. The introduction of new architectural mechanisms, such as those likely to be found in physically scalable microprocessor designs, will carry with them the need to devise strategies for handling these exceptional events. Microprocessors of all types must be able to support events like cache misses, interrupts, memory dependencies, branch mispredictions, and context switches. The design of these items is a challenging task for any microprocessor, but is especially challenging for physically scalable microprocessors due to the necessity that they be implemented in a distributed manner.

1.3 Tiled Microprocessors: A Class of Physically Scalable Microprocessor

How do we design physically scalable microprocessors? In this thesis, we examine *tiled microprocessors*, which are microprocessors that have the property of being physically scalable *by construction*. The central idea is to compose the microprocessor out of three core architectural entities – *tiles*, *networks*, and *I/O ports* – which are in correspondence with the three classes of VLSI resources described in Section 1.1.1: transistors, wires and pins. This correspondence is shown in Table 1.2. The computation and storage abilities of on-chip transistors are exposed through the tile abstraction. The communication capabilities of on-chip wires are exposed through the network abstraction. Finally, pin connections to the off-chip world are exposed through the I/O port abstraction.

Each tile occupies a rectangular region of the silicon area and is sized so that its width and

VLSI Resource	Architectural Entity
Gates	Tiles
Wires	Networks
Pins	I/O Ports

Table 1.2: Correspondence of VLSI resources and architectural entities in Tiled Microprocessors

height are approximately the distance that a signal can travel in a single cycle. In addition to computational resources, a tile contains an incremental portion of the network – the network links that connect to the nearby neighbors and the router that controls those links. These point-to-point, pipelined on-chip network links contain at least one register between neighbor tiles. The I/O ports are connected to the periphery of the network.

Systems are scaled up or down by changing the number of tiles and I/O ports in accordance with the amount of die array and the number pins available, rather than by changing the contents of the tiles themselves. A latency, proportional to the distance between tiles, is assigned for communication between tiles.

1.3.1 Tiled Microprocessors Meet the Physical Scalability Criteria

We develop the definition of a tiled microprocessor by examining how they fulfill the seven criteria of physical scalability:

1. **Frequency Scalability** Tiled microprocessors meet the frequency scalability criteria because they do not have wires or structures that grow in size as the system is scaled up. In particular, because network wires are registered at tile boundaries, we know that wire lengths in the system do not grow as the system is scaled up by adding more tiles.

2. **Bandwidth Scalability** Tiled microprocessors implement all inter-tile networks using point-to-point rather than broadcast networks. Furthermore, all protocols implemented on these networks will be implemented using point-to-point communication patterns rather than broadcast. This includes protocols for performing both operation-operand matching as well as those that manage memory accesses and control operations. This way network congestion and latency will be solely a function of the computation being performed and not the number of tiles in the system.

3. **Proportional (non-uniform) Resource Latency** Tiled microprocessors are implemented with point-to-point networks that have latencies proportional to the distances between tiles. Furthermore, the programmer has the choice of determining how many tiles will be used to implement the computation. Thus, as microprocessors with more resources become available, the option still remains to use fewer tiles in order to minimize latency.

4. **Exploitation of Locality** Tiled microprocessors provide architectural interfaces for software and hardware mechanisms to control the placement of instructions, data objects, and I/O connections so that communication in the system can be localized.

5. **Efficient Inter-node Operation-Operand Matching** Of course, in order to be a microprocessor at all, the functional units have to be connected by a scalar operand network. Tiled microprocessors contain a scalar operand network that performs operation-operand matching both within and between all tiles[5].

6. **Deadlock and starvation management** Tiled microprocessors, like the greater class of physically scalable microprocessors, must handle deadlock and starvation inside tiles and in all inter-tile networks, such as the SON. Later sections of this dissertation will describe some ways that tiled microprocessors can attain this.

7. **Support for Exceptional Events** Tiled microprocessors implement the same class of functionality as conventional microprocessors, such as interrupts, memory requests, branches and context switches. This requirement extends to functionality implemented over the inter-tile networks, e.g. the tiled microprocessor must be able to context switch even in the presence of inter-tile network traffic.

1.3.2 Tiled Microprocessors Overcome Microarchitectural Scalability Limitations

Tiled microprocessors overcome the limitations inherent in modern-day microprocessors that attempt to employ the microarchitectural approach to scalability. We examine in turn each of the issues discussed in Section 1.1.3 ("Problems with addressing scalability through microarchitecture"):

a. **Proportional returns on VLSI resources** Tiled microprocessors expose transistor resources through a proportional number of tiles. They expose the VLSI wiring resources through the on-chip networks. Finally, they expose the VLSI pin resources via the I/O port abstraction. These tiles, networks and I/O ports are in turn exposed to the programmer, providing proportional programmable resources.

b. **Constant Design and Validation Effort** Because systems are built by replicating the same basic tile and I/O port design, the system require only marginally more design and validation effort as more VLSI resources are employed.

c. **Power Efficiency** Tiled systems are easy to optimize for power and are inherently power-efficient. First, tiling provides an easy way to power down portions of the chip when they

[5]In this thesis, we examine a system that is composed of only one type of tile, however we do not preclude the possibility of having a heterogeneous tiled system composed out of multiple types of tiles.

are unneeded [62]. For instance, individual tiles can be powered down by gating the clock coming into the tile. Second, tiled systems can easily implement frequency and voltage scaling to tradeoff parallelism for energy because they can exploit parallelism with little overhead. Third, because tiles do not grow in complexity as the system scales, the designer only has to optimize the power consumption of a single tile and the power gains will be realized across all of the tiles and thus throughout the microprocessor. Furthermore, because the processing resources available to the programmer are proportional to the underlying amount of VLSI resources, i.e., because the system overhead does not grow as more ALUs are added, the system does not experience the same declining efficiency found in successive generations of modern-day microprocessors. Finally, tiled systems can be implemented with short wires and simple, fixed-sized VLSI structures which makes them inherently power-efficient.

d. **Performance Transparency** Since the central way in which the processor transforms VLSI resources into improved performance is exposed to the programmer via the number of tiles available, the user has a clear picture of the performance model. Furthermore tiles tend to be simple, because the mechanism for improving successive generations of processors is not through adding microarchitectural mechanisms, but by adding tiles.

e. **Design Adaptivity** Tiled microprocessors are eminently design adaptable, because the designer can easily vary the number of tiles to accommodate different operating environments. Furthermore, because the system is composed out of a replicated element, it is easy to adapt tiled systems to new technology parameters – simply design a new tile, and the rest of the system is a few copy commands away.

1.3.3 The Raw Tiled Microprocessor Prototype

In order to investigate tiled microprocessors further, we designed and built a full tiled microprocessor prototype in 180 nm technology. We call this microprocessor the Raw microprocessor, in reference to the direct correspondence between VLSI resources and architectural entities in the architecture. A photomicrograph of the Raw microprocessor is shown in Figure 1-1. The 330 mm^2 chip attains 16-issue, greater than any Intel microprocessor in the same process generation.

1.3.4 Contributions

The contributions of this thesis (and the underlying research) focus around the formulation, formalization, and evaluation of tiled microprocessors as a way of attaining physically scalable microprocessors. To this end, the thesis proposes and examines seven criteria for physical scalability in microprocessor designs. It uses the ATM, an archetypal tiled microprocessor, to illustrate the design considerations that arise in realizing the seven criteria. One of these criteria leads to the introduction

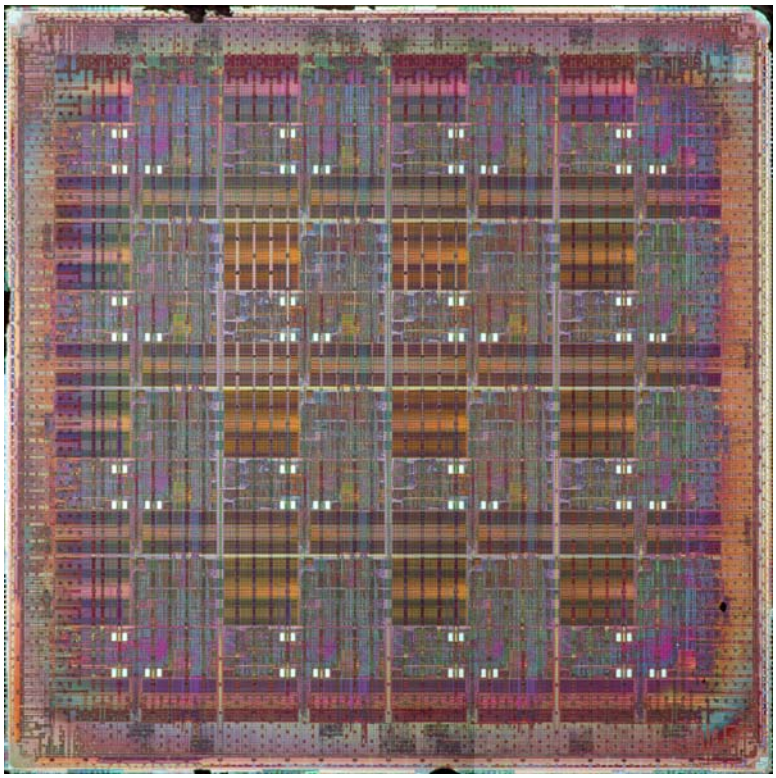


Figure 1-1: Photomicrograph of the scalable 16-tile Raw microprocessor.

and formalization of a new class of network, Scalar Operand Networks, which are the core communication mechanism inside scalable microprocessors. Finally, the thesis describes and evaluates a complete 180 nm prototype tiled microprocessor implementation, the Raw microprocessor.

1.3.5 Thesis Overview

This thesis continues as follows. Chapter 2 employs an archetypal tiled microprocessor, labeled the *Archetypal Tiled Microprocessor*, to explore the key issues in designing tiled microprocessors. Chapter 3 and Appendices A and B describe the Raw architecture, a concrete tiled microprocessor implementation. Chapter 4 examines the Raw prototype and Raw VLSI implementation. Chapter 5 evaluates the Raw prototype and describes the various programming systems that were designed at MIT to program it. Chapter 6 examines related work, and Chapter 7 concludes.

Chapter 2

The Archetypal Tiled Microprocessor

In the introduction, we discussed the favorable characteristics of a tiled microprocessor. However, concretely, what are the essential architectural components of a tiled microprocessor? How does a tiled microprocessor function? In what ways is a tiled microprocessor different than conventional microprocessors? How does the tiled microprocessor address the seven criteria for physically scalable microprocessors discussed in Section 1.2.1? These are the questions which this chapter addresses.

In order to do this, we discuss a prototypical tiled microprocessor which we call the *Archetypal Tiled Microprocessor*, or ATM. As with the Raw prototype, the ATM has the essential characteristics that we would expect of a tiled microprocessor; however, at the same time, it allows us to abstract away from the nitty-gritty characteristics which may vary from design to design. It also allows us to generalize some of the findings that resulted from our experience with the Raw prototype.

2.1 Basic Elements of a Tiled Microprocessor

Fundamentally, the purpose of a tiled microprocessor is to execute programs in a parallel fashion. To do this, tiled microprocessors are comprised of an array of *tiles*. Each tile contains a number of functional units. Using their functional units, the tiles work together to collectively execute the instructions of a single program. The basic idea is to take the individual instructions of a program and assign them to different tiles, so that they can execute in parallel. Figure 2-1 shows an example program sequence and the mapping of instructions to an array of tiles.

Given a mapping of instructions to different tiles, tiled microprocessors must provide a way to transport the outputs of an instruction to the inputs of one or more dependent instructions. For this purpose, tiled microprocessors employ a class of on-chip networks called *scalar operand networks* ("SON") that is specialized for exactly this purpose. We call them scalar operand networks because

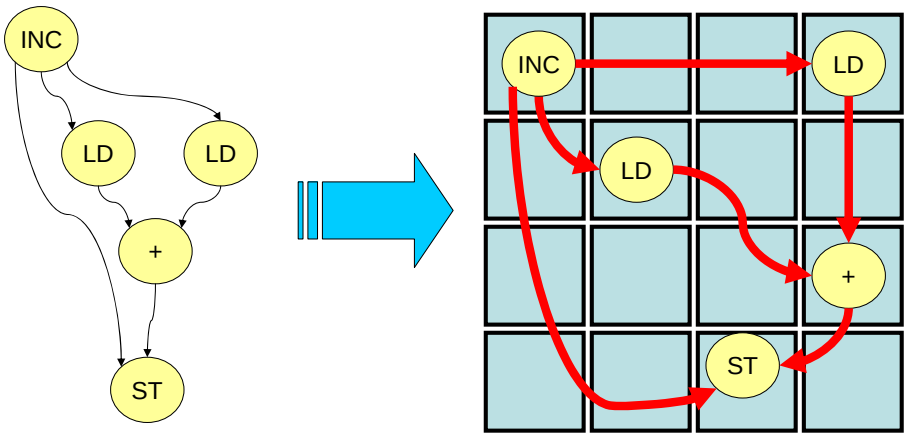


Figure 2-1: The goal of a tiled microprocessor: parallel execution of a program's instructions across an array of tiles. The figure depicts a program graph being mapped to an array of tiles.

they are a type of network optimized for transporting scalar operands between physically distributed functional units.

Furthermore, we want these tiles to operate autonomously; so a tile will contain a fetch unit that sequences a unique program counter ("PC") through an instruction stream. Tiles will also contain a data and instruction cache so that they can take advantage of instruction and data locality much like conventional microprocessors.

Finally, programs need to have inputs; and the instruction and data caches need to be backed by a larger pool of memory. For this purpose, we add two components. First, we will add a class of on-chip networks that is intended for generalized data transport, including cache misses. The tiles can use these networks to communicate between each other (for instance, to share their cache memory) and with the outside world; we'll call these the *generalized transport networks* ("GTN"). In some cases, these may be the same network as the SON. Furthermore, we want to ensure that our on-chip networks are also able to communicate with the outside world. To do this, we surround the array of tiles with a number of *I/O ports*, which interface the on-chip networks at the array edges with the pins of the VLSI package. These pins in turn are connected to off-chip devices like DRAM's, A/D converters, and PCI slots. Using these I/O ports, not only can tiles communicate with I/O devices, but the I/O devices can use the on-chip networks to communicate with each other.

2.2 Attaining Frequency Scalability (C1)

At this point, we have touched upon the basic components of a tiled microprocessor. However, as is often the case with parallel systems, one of our goals in designing the tiled microprocessor is to attain *scalability*. In this context, we seek *physical scalability*, which is the ability to create designs that exploit the VLSI processes made available with subsequent generations of Moore's law. To first

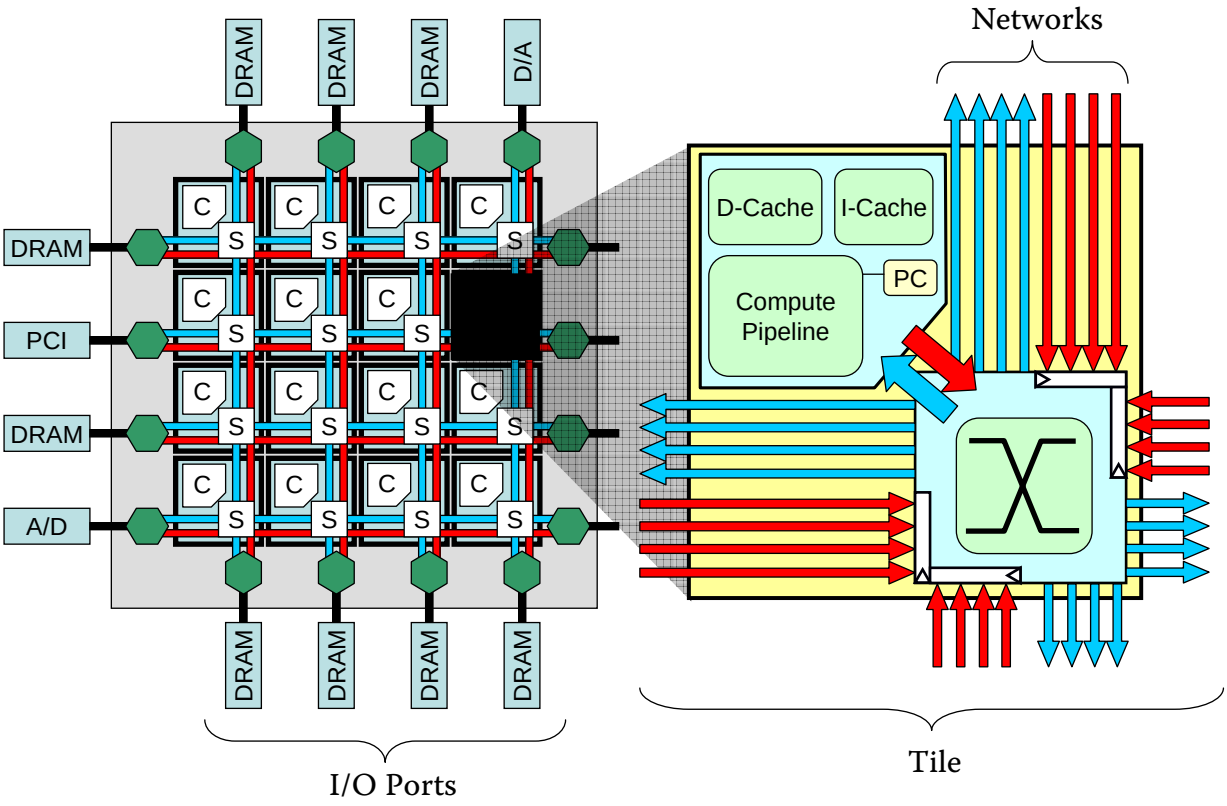


Figure 2-2: The Archetypal Tiled Microprocessor.

order, the tiled microprocessor as we've described it has a straight-forward path to doing this. As Moore's law provides more transistors, we can add more tiles. As Moore's law provides more pins, we add more I/O ports. Finally, as Moore's law provides more wiring resources, we can provide more network links.

However, physical scalability is more subtle then simply adding more tiles, networks links and I/O ports. Ultimately it is the way that these elements are tied together which determines this scalability. Much of the discussion in this thesis centers upon the architectural mechanisms that emerge from the desire to achieve scalability. In this section, we focus on attaining *frequency scalability*, the first of our seven criteria for physically scalable microprocessors.

To further the discussion on frequency scalability, we are ready to introduce the Archetypal Tiled Microprocessor. The Archetypal Tiled Microprocessor is an example of a physically scalable microprocessor based upon a set of engineering principals that help ensure scalability. The ATM is depicted in Figure 2-2. On the left hand side, we see an array of identical tiles, which are connected to their nearest neighbors via a number of point-to-point networks. At the periphery of the array of the tiles, the network links are connected to the I/O ports, which are in turn connected to the pins of the chip and to off-chip devices, such as DRAMs, I/O buses, and A/D converters.

Depicted on the right side of Figure 2-2 is the tile. A tile is comprised of computing resources

(including caches), routers (indicated by the crossbar "X" symbol), and a set of network links, which span the tile and connect the tile to its four neighbors. These network links are registered on input. Because the network links are the only wires (except for the clock) that run between tiles, this configuration gives us an important guarantee – that the longest wire in the system is no longer than the width or height of a tile. The idea is that the cycle time of the tiled microprocessor should be approximately the time it takes a signal to pass through a small amount of logic (e.g., a simple router) and across one inter-tile link. We call these inter-tile links *point-to-point interconnections* because there is a single sender on one end and a single receiver on the other. The advantage of using nearest neighbor point-to-point links is that the wire lengths of the architecture do not grow as we add more tiles. The alternative communication mechanism, shared global buses, has the disadvantage that as more tiles are added to the bus, the bus gets longer, has more contention and has greater capacitance. All of these result in degraded performance as the number of tiles increases. None of these negative effects occur with the ATM's point-to-point links.

To make the issue of frequency scalability more concrete, let's examine graphically what happens when we attempt to scale the ATM design. In Figure 2-3a, we have depicted the baseline ATM design. In one scenario, we might anticipate using the same process generation, but with a more expensive and bigger die, as well as a more expensive package with more pins. This is shown in Figure 2-3b. As is evident from the diagram, although we have increased the number of tiles in order to exploit the larger transistor budget, none of the architectural wire lengths have changed. Thus, we can reasonably expect that the new design will have a frequency similar to the original design. In Figure 2-3c, we have pictured a implementation of the ATM in a VLSI process two generations of Moore's law later. The amount of on-chip resources has roughly quadrupled. As a result, in the same space, we are able to fit four times as many tiles and network links. In this new process, transistors are twice as fast, but the wire delay, per mm, is about the same. Thus, for a sensible design, the wire lengths need to be approximately half the original length. Because the ATM has only local wires, the wire lengths have shrunk with the tiles, and are a factor of two shorter. As a result, wire delay should not impact the design's ability to exploit the faster transistors and run at twice the frequency as the original ATM.

The design discipline for ensuring short wires is an important step towards frequency scalability. Today, it can take ten to twenty cycles for a signal to travel from corner to corner of a silicon die. As a result, wire delay has the potential to decrease the frequency of a chip by a factor of twenty or more in comparison to its frequency if wire delay were not considered. By ensuring that the architecture has no wires that increase in size as the number of tiles increases, tiled microprocessor designs can attain high frequencies and scale to future process generations even if wire delay worsens.

Wire delay is not the only potential problem area for frequency scalable systems – logic delay is also an important consideration. As VLSI structures – such as FIFO buffers, memories, bypass paths

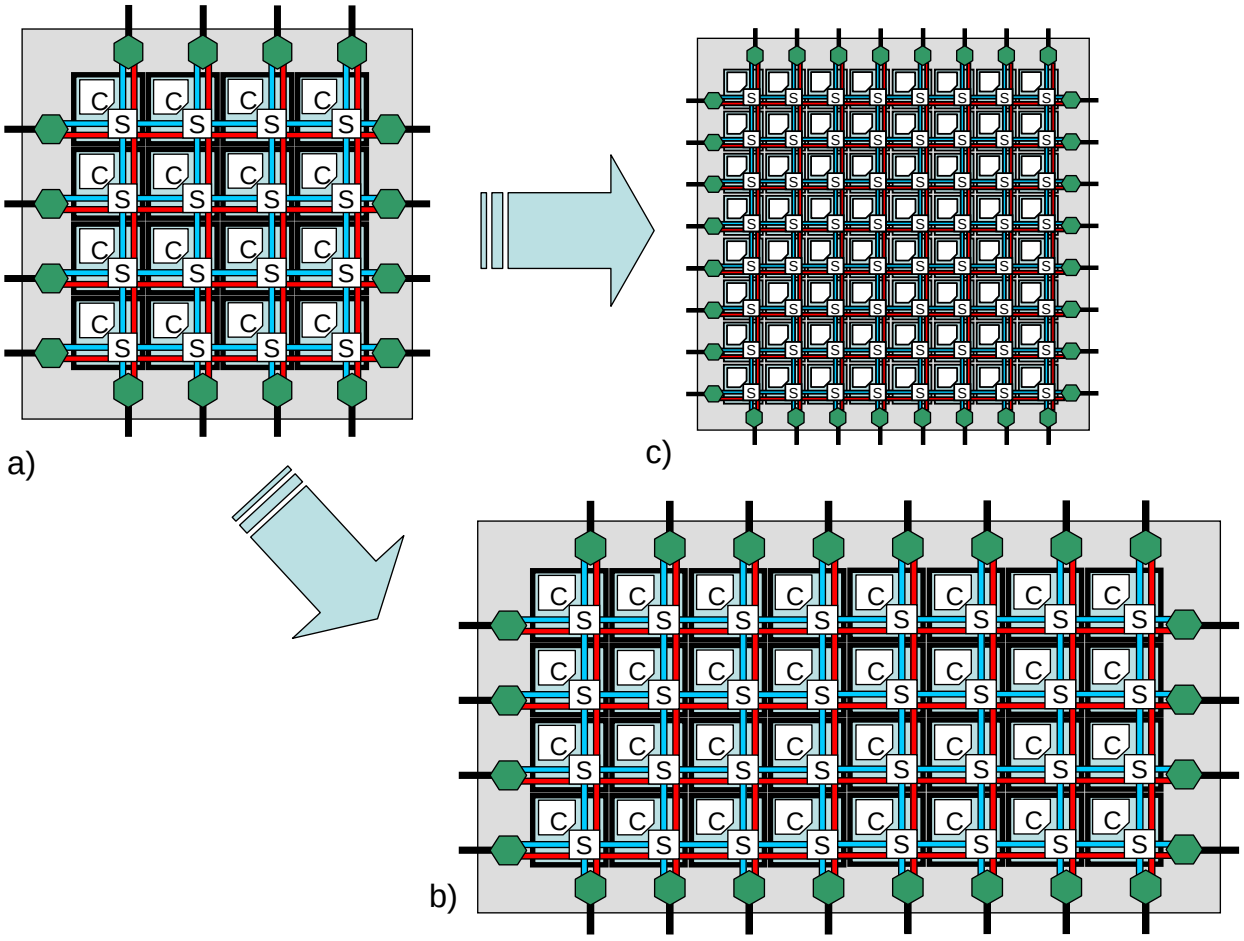


Figure 2-3: Scalability Scenarios for The Archetypal Tiled Microprocessor. Part a) is the baseline ATM. Part b) depicts an ATM implemented in the same VLSI process, but with a bigger die size and a higher pin-count package. We can expect that this ATM will have a frequency similar to the original ATM, because wire lengths have not changed. Part c) depicts an ATM implemented two VLSI process generations later. In this new process transistors are twice as fast; but the wire delay, per mm, is about the same. For a sensible design, the wire lengths need to be approximately half the original size. Because the ATM has only local wires, the wire lengths have shrunk by a factor of two; as a result, wire delay should not impact the design's ability to exploit the faster transistors; the design should run at twice the frequency as the original ATM.

and out-of-order issue logic – grow in size, they also get slower. This is a key problem in today's superscalar processor designs. As the issue-width of superscalars increases, their micro-architectural structures grow in size, and the frequency of the design drops. As a result, superscalars are often characterized as either "brainiacs" – lower-frequency designs that are wide issue – or "speed-demons" – high-frequency designs that are narrow issue [40]. One example of a brainiac is the Itanium 2, which is a 6-way issue microprocessor. Its frequency is significantly lower than 3-way issue superscalars like the Pentium 4.

By increasing the capabilities of a machine by adding more tiles, rather than by increasing the size of micro-architectural structures, tiled microprocessors can be both "speed-demons" and

"brainiacs" at the same time. By using the same tile design regardless of the scale of the system, tiled microprocessors ensure that frequency is not adversely impacted by the size of the system.

The ATM is one example of how to attain frequency scalability in a tiled microprocessor. Certainly it is not the only possibility – we could imagine more exotic interconnection topologies with different tile shapes that are also frequency scalable. However, the symmetry of mesh networks and square tiles affords us a simple proof, by construction, that the ATM is frequency scalable. In the field of engineering, more complex solutions (for instance, utilizing different wire styles to enable more aggressive topologies) often achieve a constant-factor advantage that is desirable in practice. For these more sophisticated tiled microprocessors, a more complex chain of reasoning will be necessary to ensure frequency scalability.

2.3 Bandwidth Scalability, Locality Exploitation, and Usage-Proportional Resource Latencies (C2, C3, and C4)

Although we have shown that the ATM is frequency scalable, we would also like to show that the ATM is *bandwidth scalable*, that it *exploits locality* and that it supports *usage-proportional resource latencies*. All three of these criteria are necessary to ensure that tiled microprocessors do not suffer unnecessary performance degradation as we add more resources, such as tiles. A system with frequency scalability but without these three properties is likely to suffer from low efficiency. We examine each of the three criteria in turn.

2.3.1 Bandwidth Scalability by Eliminating Broadcasts

We can look to the program mapping in Figure 2-1 for an excellent example of an bandwidth scalability problem. In that example, we have mapped instructions to different tiles. However, we have not yet specified exactly how operands are routed between instructions on different tiles. Suppose that the scalar operand network that we employed used a broadcast protocol in order to communicate the results of an instruction to other tiles. That is, it sends the output of every instruction to all tiles. On the surface, this is a reasonable thing to do – it means that we can be more relaxed about tracking dependencies between instructions on different tiles. In fact, this is exactly what today's out-of-order issue superscalar processors do.

Unfortunately, one of the implications of broadcasts for operand communication is that every tile most process every other tile's output operands. As a result, if there are N tiles in the system, each producing one output per cycle, then each tile must be able to process N incoming operands per cycle. We could solve this problem by augmenting the tile's capabilities. First, we would need

to scale the bandwidth of each tile's network links with the number of tiles in the system. Further, we would have to augment the tile's operand bypassing hardware so it could "snoop" all of the incoming operands. Clearly, this approach creates many physical scalability issues. The frequency of the system would eventually have to drop as we increased the number of tiles.

An alternative approach is to keep the current tile's capabilities and just take multiple clocks to process the operands. Although this solution is physically scalable, it effectively means that every tile would be able to execute instructions at a rate much slower than the frequency of the processor. This is because it would be bottlenecked by the time required to process all of the operands being broadcasted from the other tiles.

Clearly, neither of these solutions is particularly palatable. A better solution is to employ multicast operand communication, that is, to have instructions send operands only to those other instructions that are dependent on a particular operand. This, combined with a routed, point-to-point (rather than broadcast-based) network, would give the system the property that adding more tiles does not increase the amount of operand traffic that an individual tile has to service.

2.3.2 Providing Usage-Proportional Resource Latencies and Exploitation of Locality

Two other desirable characteristics for physical scalability relate to the cost of accessing varying numbers of resources. First, we would like the ATM to support *usage-proportional resource latencies*. The basic idea is that as we add more resources to the system, we would like the inter-resource latencies of a fixed set of resources to remain constant. Second, an ATM that *exploits locality* can achieve an asymptotic speedup on applications which have locality, relative to an architecture that does not exploit locality.

In order to demonstrate the benefits of architectures that exploit locality and usage-proportional resource latencies, in the subsequent paragraphs, we contrast the ATM with a system we call the Uniform Resource Machine ("URM"), which does not exploit locality. Figure 2-4 depicts the URM. The URM offers a number of compute pipelines ("P") equal latency, full bandwidth access to a large number of caches ("C"), using a pipelined interconnect composed of a number of switches ("S"). In this example, the caches could easily be any sort of resource, whether instruction memory, register files, or ALUs. Figure 2-4b demonstrates that as more compute pipelines are added, the interconnect must grow larger to accommodate the greater bandwidth needs. Although the pipelined interconnect is frequency scalable because its wires are registered at the switches and of constant length, the distance, measured in cycles, between a processor and cache memory grows as the system scales. Thus, because the system does not support usage-proportional resource latencies, the cost of accessing the cache increases as the system is scaled up. Furthermore, this system has an unfavorable asymptotic growth of its interconnect resources – basically, as the system scales, the percentage of

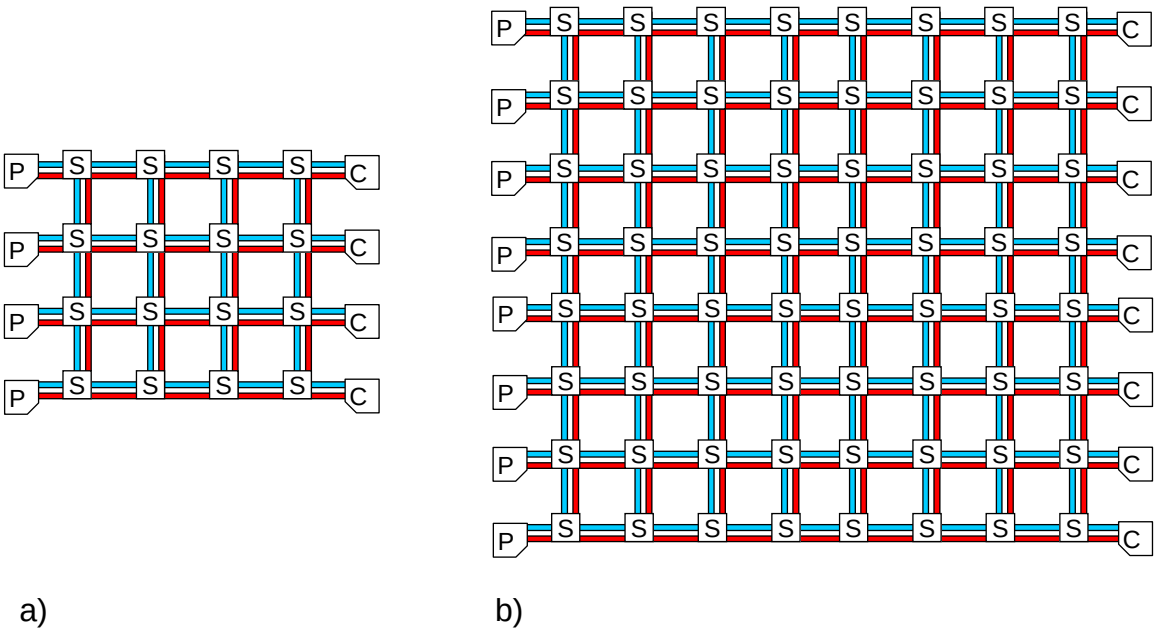


Figure 2-4: The Uniform Resource Machine, a machine that is physically scalable but that does not exploit locality. In both examples, a number of compute pipelines ("P") are connected by a pipelined interconnect to a number of caches ("C"). Figure 2-4b shows that as more compute pipelines are added, the interconnect must grow quadratically larger to accommodate the greater bandwidth needs. Although the pipelined interconnect is physically scalable because its wires are registered at the switches and are of constant length, the distance, measure in cycles, between a processor and cache memory grows as the system scales, as illustrated. Thus, because this system does not support usage-proportional resource latencies, the cost of accessing the cache increases as the system is scaled up.

the system dedicated to cache and processing (relative to interconnect) becomes a disappearing fraction.

In contrast to the processors in the URM design shown in Figure 2-4, ATM tiles are largely self-sufficient. Each tile has its own program counter, fetch unit, instruction cache, register file, data cache, and ALUs. Because these resources are local, the latency of these resources does not change as the ATM is scaled to larger number of tiles. If an ATM tile did not have its own PC, fetch unit and instruction cache, but was supplied by a single system-wide resource, the cost of mispredicted branches would greatly increase as the system was scaled up and latencies increased. Similarly, the latency of accessing data memory or using a functional unit would also increase. Finally, like the URM, the ATM would have to dedicate a disproportionate amount of die area to the interconnect to provide adequate bandwidth to keep the tiles fed with instructions and data.

In the ATM, the *average* latency between resources *does* increase as a program utilizes more resources (e.g., ALUs, caches or tiles); this is an unavoidable reality stemming from the fact that more resources take up more space, and more space takes greater latency to traverse. However, the key idea is that the latency the program observes is based on the way the program has been mapped

(and how many tiles the mapping employs) rather than the total number of resources in the system.

In Table 2.1, we compare the asymptotic costs of accessing resources in the ATM versus the costs of accessing resources in the URM. The costs are given relative to A, the silicon die area used to implement the system.

	ATM	URM
Global Bisection Bandwidth	$O(\sqrt{A})$	$\theta(\sqrt{A})$
Local Bandwidth	$\theta(A)$	$\theta(\sqrt{A})$
Global Latency	$O(\sqrt{A})$	$\theta(\sqrt{A})$
Subset Latency	$O(\sqrt{S})$	$\theta(\sqrt{A})$
Local Latency	$\theta(1)$	$\theta(\sqrt{A})$
Processors Utilized		
- Global Communication	$\theta(\sqrt{A})$	$\theta(\sqrt{A})$
- Local Communication	$\theta(A)$	$\theta(\sqrt{A})$

Table 2.1: Asymptotic analysis of ATM versus URM. A is the area of the silicon die. S is the subset of the silicon die being used by the computation, in the case where computation only uses a subset of the total die area.

As is evident from the table, the latencies and bandwidths for global communication are similar in the ATM and URM systems. By global communication, we mean communication in which there is no decipherable pattern that we can exploit to co-locate the data item and the user of the item. The global latencies for ATM and URM have the same asymptotic upper bound, $O(\sqrt{A})$, because in both cases, each resource access requires a message to be sent across a span proportional to the dimensions of the chip, $\theta(\sqrt{A})$. The bisection bandwidths have the same asymptotic upper bound, $O(\sqrt{A})$, because both designs use an area proportional to the total die area for communication, and the bisection bandwidth of a square area is the square root of that area.

The ATM is superior to the URM for local communication. By local communication, we mean communication in which there is a strong correspondence between data items and the consumers of the items. In this case, the data item can be located on the tile itself, and no inter-tile communication is necessary, which results in $\theta(1)$ communication latency. In the URM, there is no mechanism by which to exploit locality, so the latency is the same as if there were no locality to exploit, $\theta(\sqrt{A})$. We also list here what we call the "subset latency" - this is the latency of the system when only a subset of the system (i.e., of the total die area) is being used for the computation, but communication is global within this subset. The table shows that, in this scenario, the ATM's resource latency is proportional only to the die areas used for computation rather than the total die area in the system.

The final entries in the table, under the heading "Processors Utilized", give a feel for the utilization of the system in the two scenarios. By utilization, we mean the percentage of the processing

nodes that are actively computing as opposed to waiting for data. In a program with global communication, there are many communications that must span long distances across the machine. As a result, the system is bottlenecked by two factors - if the program is latency-sensitive, both systems will spend significant time ($O(\sqrt{A})$) waiting for responses to arrive from long-distance requests for resources. If the program is not latency-sensitive (i.e., it can emit many requests in parallel), then both systems will be rate-limited by the $\theta(\sqrt{A})$ network bandwidth available. In contrast, for a program with only local communication, the ATM's tiles are able to proceed independently, with little inter-tile communication, achieving $\theta(A)$ utilization; while the URM will continue to achieve $\theta(\sqrt{A})$ utilization because it must wait on requests sent out to access the remote resources.

As is evident from the table, the advantages of the ATM are very dependent on the presence of locality in the end application. However, exploitation of locality is a common precept of current microprocessor design - if most computations did not have locality, then today's microprocessors would probably not have caches. Because of this, we believe that structuring the system to exploit locality is a worthwhile goal. Nonetheless, some computations have little locality, and in these cases, the ATM will perform equivalently, but not better, than the URM design. Thus, overall, we can say that the ATM is no worse than the URM in the absence of locality, but can do much better when it is present.

2.4 Attaining Efficient Operation-Operand Matching with Scalar Operand Networks (C5)

In the beginning of this chapter, we alluded to our end goal of creating tiled microprocessors: to execute a program's instructions across an array of tiles. Now that the previous sections have established the characteristics of a tiled architecture that are necessary for physical scalability, we are now ready to discuss the core architectural component of a tiled microprocessor: the scalar operand network.

To develop our understanding of scalar operand networks, let us examine what is necessary for executing a program on a tiled microprocessor. To assist the reader, we resurrect the picture of a program graph being mapped to a tiled microprocessor in Figure 2-5. There are key stages in executing a program across a tiled microprocessor. First, we need to *assign* each operation in the program to a tile. Second, we need to *transport* operands from their source tiles to their destination tiles. Finally, the destination tiles need to join incoming operands with their corresponding operations. Collectively, we call this process *operation-operand matching*, and we call the mechanisms that perform this function, whether hardware or software, the *scalar operand network* ("SON").

The remainder of this section examines SONs as they are used in current day microprocessors (Section 2.4.1), enumerates the key components of SON operation in tiled microprocessors (Sec-

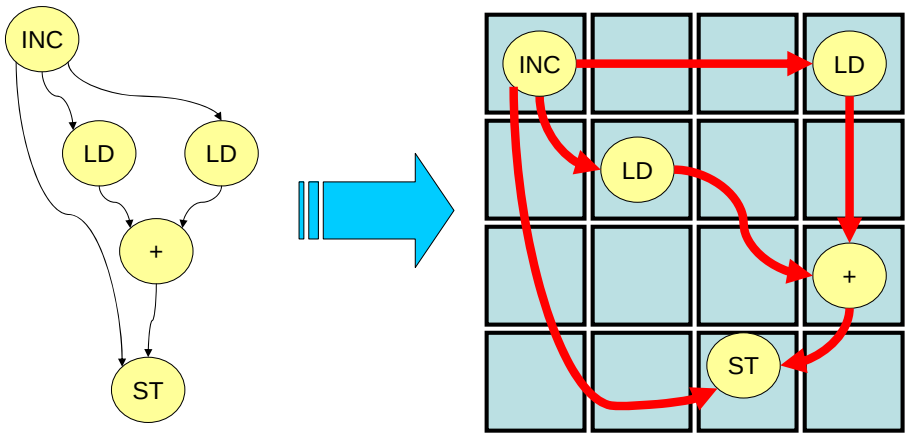


Figure 2-5: Executing a program graph: the role of a scalar operand network.

tion 2.4.2), identifies common properties and diversity of approaches in current tiled microprocessor SON research (Section 2.4.3), and proposes a metric for comparing SON implementations (Section 2.4.4). The reader may also wish to refer to Chapter 4 and Appendices A and B, which discuss the SON implemented in the Raw microprocessor.

2.4.1 Scalar Operand Networks in Current Day Microprocessors

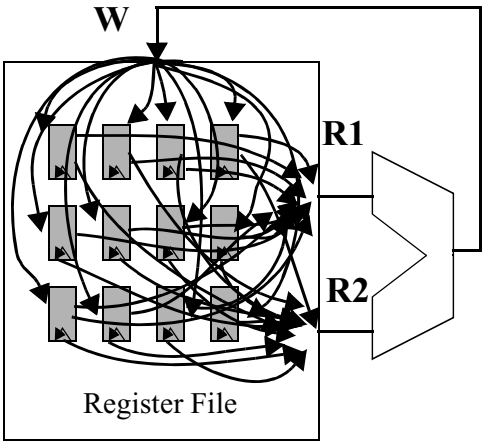


Figure 2-6: A simple SON.

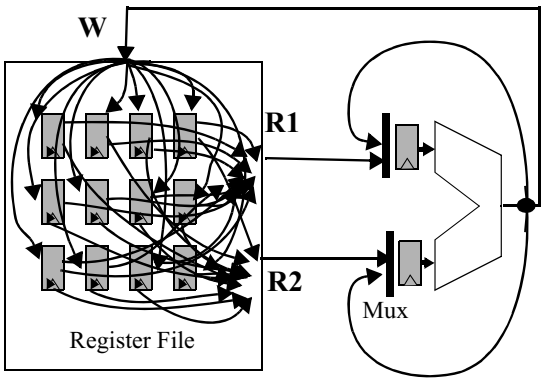


Figure 2-7: SON in a pipelined processor with bypassing links.

As it turns out, scalar operand networks are not unique to a tiled microprocessor; they exist in legacy and current day microprocessors as well. However, because tiled microprocessors are built out of distributed components, their SONs are a bit more sophisticated. To illustrate this connection with the past, we examine the scalar operand networks of a few classical microprocessor designs.

Pictured in Figure 2-6 is perhaps one of the simplest SON designs - the transport mechanism implicit in a simple register-ALU pair. Typically, register files are drawn as black boxes; instead we expose the internals of the register file in order to emphasize its role as a device capable of

performing two parallel routes from any two internal registers to the output ports of the register file, and one route from the register file input to any of the internal registers. Each arc in the diagram represents a possible operand route that may be performed each cycle. This interconnect-centric view of a register file has become increasingly appropriate due to the increased impact that wire delay has had on VLSI processes in recent years. In the context of operation-operand matching, this simple SON *assigns* all operations to the single ALU, it uses the network inside the register file to perform operand *transport*, and the joining of operands with operations is achieved through the use of register names.

Figure 2-7 shows a more advanced SON, the transport in a pipelined, bypassed register-ALU pair. The SON now adds several new paths, multiplexers and pipeline registers, and partitions operand traffic into two classes: "live" operands routed directly from functional unit to functional unit, and "quiescent-but-live" operands routed "through time" (via self routes in the register file) and then eventually to the ALU. The partitioning improves the cycle time because the routing complexity of the live values is less than the routing complexity of the resident register set. This transformation also changes the naming system – the registers in the pipeline dynamically shadow the registers in the register file. In terms of operation-operand matching, all operations are once again assigned to the single ALU, but the network topology (and routing function) is more complex, and the corresponding task of joining operands with operations requires more sophisticated logic that incorporates the knowledge of what values are currently being routed on the bypass paths.

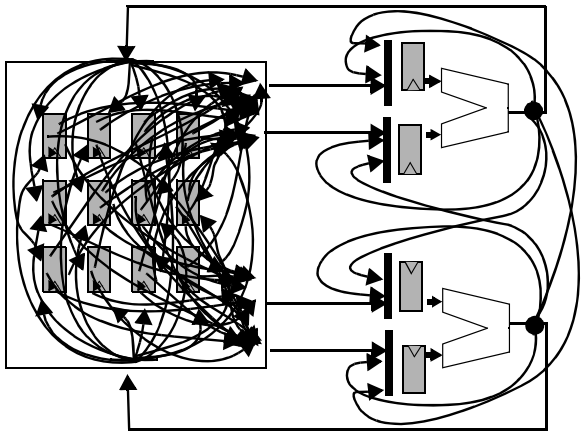


Figure 2-8: A superscalar pipelined processor with bypass links and multiple ALUs.

Finally, Figure 2-8 shows a pipelined, superscalar (i.e., with multiple ALUs), bypassed processor In this case, the SON includes many more multiplexers, pipeline registers, and bypass paths, and looks much like our traditional notion of a network. In this case, the SON must assign operations to one of the two ALUs, it must coordinate the transport of the operands over the more complex interconnect structure (multiple bypass paths, and a more complex multi-ported register file interconnect), and it has a relatively complex control for matching operations and their operands (which

may be coming from different places depending on times at which instructions issue.)

Modern microprocessors SONs typically look much like the superscalar design shown in Figure 2-8 – with the exception that they have more ALUs (they are referred to as being "wide issue"), and even more complex register files and interconnect structures. It is as these microprocessors have attempted to scale their SONs that the physical scalability problems described in the previous two section have emerged as significant concerns. For example, Intel's Itanium 2, a six-issue microprocessor, spends half of its cycle time in the bypass paths that connect its ALUs [85]. As the Itanium 2 already has a significantly lower frequency than the Pentium 4 due to physical scalability problems, the possibility of Intel's engineers increasing the number of ALUs in the Itanium design is remote.

These issues are not limited to the Itanium 2; today's microprocessors have fundamentally unscalable designs. The demands of wide issue – large numbers of physical registers, register file ports, bypass paths and ALUs distributed at increasingly large distances to accommodate interconnect – have make it progressively more difficult to build larger, high-frequency SONs that employ a single centralized register file as operand interconnect. Ultimately, overcoming these issues is the goal of tiled microprocessor research - to rework the microprocessor in a way that is physically scalable.

2.4.2 Scalar Operand Network Usage in Tiled Microprocessors

This subsection examines how the execution of programs is orchestrated on the Scalar Operand Network (SON) of a tiled microprocessor. This orchestration can be done statically by a compiler, or dynamically by a combination of hardware and/or software. In either case, the set of decisions that needs to be made is similar.

Figure 2-9 depicts these decisions through a series of four diagrams. Diagram 2-9a shows the program graph to be executed. Each numbered square denotes an instruction, which typically has a number of inputs, depicted by incoming edges. Each square typically also has outgoing edges, which indicate which other instructions consume the instruction's output(s). The first step in executing the program graph is to decide how instructions will be *assigned* to tiles. This is depicted in Diagram 2-9b, which shows four tiles and the instructions that have been assigned to each of them. Once instructions have been assigned to the tiles, the SON can decide the path that operands will take when there is an inter-tile instruction dependency. In Diagram 2-9c, these paths are indicated with hexagonal route "operations" that have been placed on the corresponding tiles where a route passes through. Finally, the SON must specify the order that routes occur in, and the order that the instructions will execute in. One such ordering is shown in Diagram 2-9d, where each instruction and route has been labeled with the cycle that it executes in. In this example, we assume the machine can execute multiple route operations in parallel with a single compute operation. We also assume that all operations (including routing) take one cycle to occur.

The reader may have noticed that the mapping given is non-optimal given this program graph

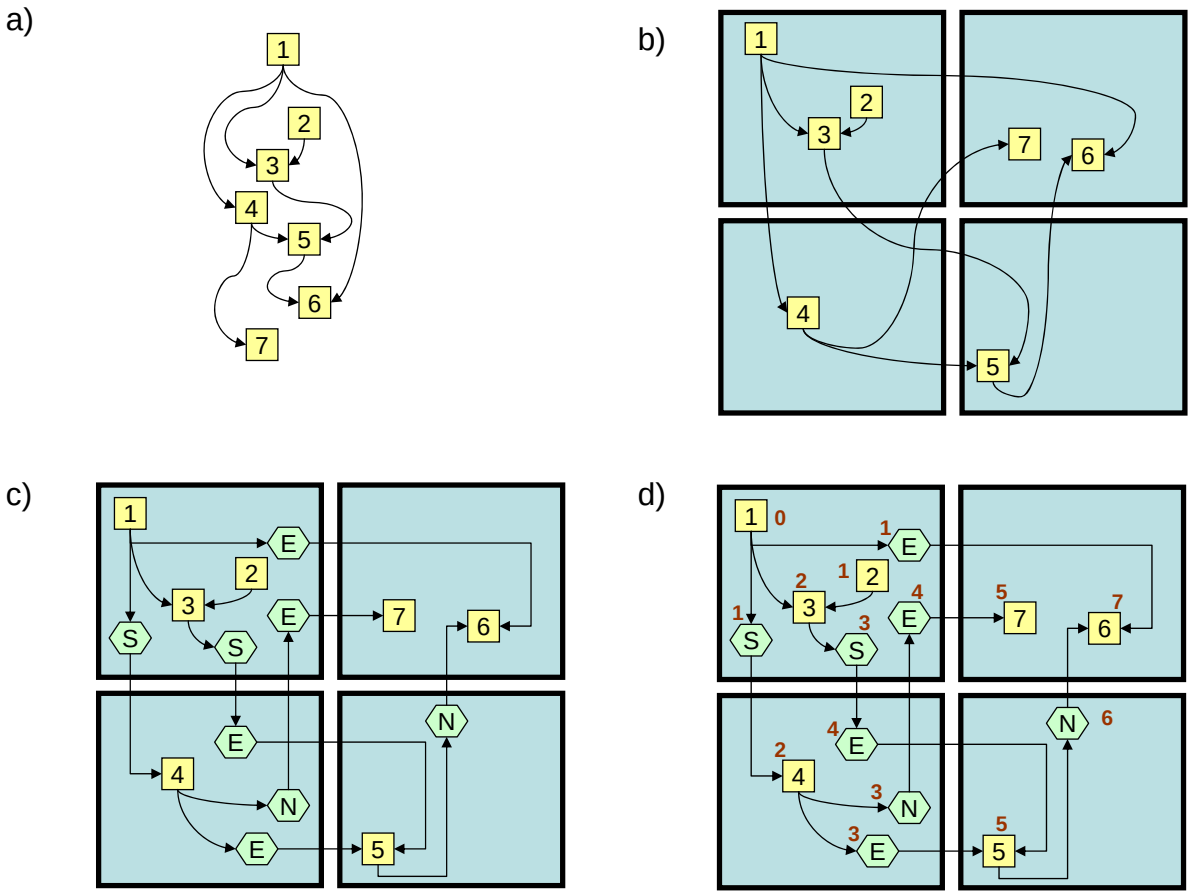


Figure 2-9: Decisions required to execute a program graph on a tiled microprocessor. Part a) is the program graph to be executed. Each numbered square is an instruction. Each incoming edge represents one of the instruction's inputs, while outgoing edges indicate which other instructions use the instruction's output operand(s). Part b) shows an assignment of instructions to tiles. Part c) shows the route operations and the tiles they occur in, determining the path that the operands take between instructions. Part d) indicates the schedule; the execution order of the instructions and route operations on a hypothetical machine in which inter-tile communication can be performed in parallel with ALU operations and in which all operations take one cycle. The cycle number that the operation occurs on is annotated next to each instruction or route operation.

and these timing constraints. Figure 2-10 shows an improved mapping, which executes in fewer cycles using fewer tiles. It's clear that effective mappings are integral to achieving good program performance on an SON. Because of this, the study of how these decisions are made (whether in software or in hardware) and the resulting machine cycle time, SON costs, and overall program performance has become an active area of research. The span of investigation extends from the scalable, compiler-focused Raw SON, to dynamic software and hardware approaches, to extensions of the unscalable hardware-focused SON of the modern-day superscalar.

In Raw's SON, the compiler makes most decisions about the way a program executes on the tiles. The compiler assigns compute instructions to tiles, and determines the paths that operands will take between tiles. Then, it schedules the route operations on each tile into a router instruction stream.

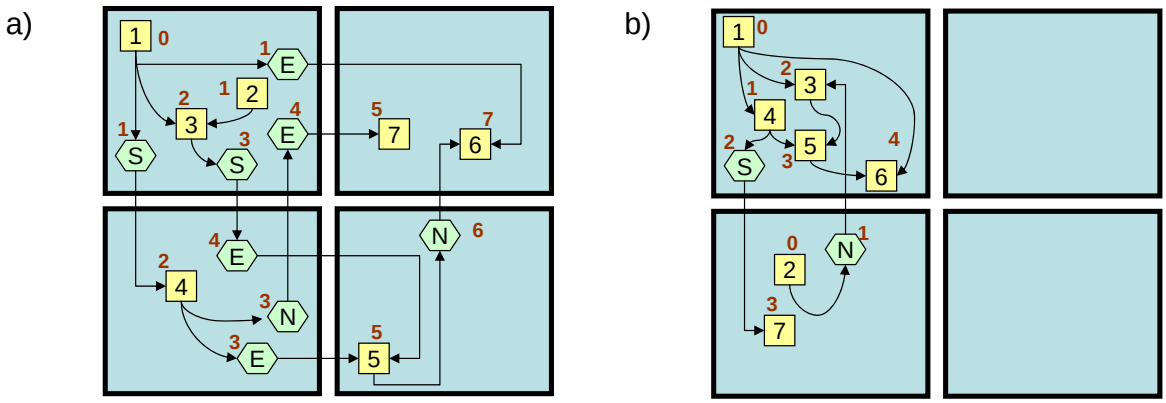


Figure 2-10: Optimized assignment and execution order. On the left is the original instruction and route assignment and ordering given in Figure 2-9. On the right is a more optimized instruction and route assignment and ordering, which happens to require only two tiles.

Finally, it schedules the compute instructions into a separate compute instruction stream. When the program is executed, the routers and compute pipelines execute their instructions in the order that they are specified in the instruction stream, much like a simple in-order processor. Chapter 3, Chapter 4, and Appendices A and B examine aspects of Raw's SON in greater detail.

In contrast, a conventional, un-tiled out-of-order superscalar performs all of these decisions in hardware. It uses arbitration logic to assign instructions to functional units as they are executed. Operands are transported between functional units over a broadcast network – or if the operand has not been recently produced, through the register file. Finally, instruction and operand route orderings are determined by a hardware scheduler as the program executes.

2.4.3 The AsTrO Taxonomy

What alternatives do tiled microprocessor designers have in designing tiled SONs? Which parts go into software and which parts go into hardware? A number of interesting proposals have recently emerged in the literature [38, 121, 108, 87, 60, 96, 105, 66, 112]. In an effort to explore the design space of SONs and to synthesize these proposals, we created the AsTrO taxonomy [112] to describe their key differences and similarities[1]. We first give a formal definition of the AsTrO taxonomy and subsequently discuss and apply it more informally.

> The AsTrO taxonomy consists of three components - **As**signment, **Tr**ansport, and **O**rdering. Each of these parameters can be decided in an SON using a *static* or *dynamic* method. Typically, the static methods imply less flexibility but potentially better cycle times, lower power, and better latencies. An architecture's categorization in the taxonomy can be expressed by a sequence of three characters – each one either an "S" for static, or a

[1] Technology constraints and engineering experience will give us more insight into which SONs are most appropriate in which environments; the ATM does not assume a specific SON.

"D" for dynamic – which are given according to the ordering in the AsTrO name: **As** signment, then **Tr** ansport, then **O** rdering.

An SON uses *dynamic assignment* if active dynamic instances of the same instruction can be assigned to different nodes. In *static assignment*, active dynamic instances of the same static instruction are assigned to a single node. A dynamic assignment architecture attempts to trade implementation complexity for the ability to dynamically load-balance operations across tiles. A static assignment architecture, on the other hand, has a much easier task of matching operands and operators.

An SON employs *dynamic transport* if the ordering of operands over transport network links is determined by an online process. The ordering of operands across *static transport* links are pre-computed (e.g., by a compiler). A dynamic transport SON benefits from the ability to reorder operand transmissions between nodes in response to varying timing conditions, for instance, cache-misses or variable latency instructions. Conversely, static transport SONs can prepare for operand routes long before the operand has arrived.

An SON has *static ordering* if the execution order of operations on a node is fixed. An SON has *dynamic ordering* of operations if the execution order can change, usually in response to varying arrival orderings of operands. A dynamic ordering SON has the potential to be able to reschedule operations on a given node in response to varying time conditions.

Thus, as will see later, we might use the AsTro taxonomy in a sentence as follows: "Raw and Scale have SSS SONs, TRIPS and Wavescalar have SDD SONs, ILDP has a DDS SON, and the conventional superscalar has a DDD SON". Generally speaking, the dynamic variants use hardware to change the way that the program executes in response to varying runtime conditions (such as cache miss patterns). Furthermore, dynamic variants can lead to more concise program representations (e.g., with less loop unrolling), since the compiled program representation is not responsible for conveying many of the details of execution. At the same time, the dynamic variants also need additional hardware logic because these decisions – such as where to place an instruction for execution (and where to find its operands), which operand to route next, and which instruction to issue next. This hardware can impact the power consumption, cycle time and inter-tile operand latency of the SON.

The line between dynamic and static can be somewhat arbitrary – imagine for instance, if we extended Raw with a profile-based runtime code generation system that can re-optimize code as it executes. Certainly this proposed system is "more dynamic" than the baseline Raw system. However, on the other hand, it remains significantly less dynamic than the baseline out-of-order superscalar.

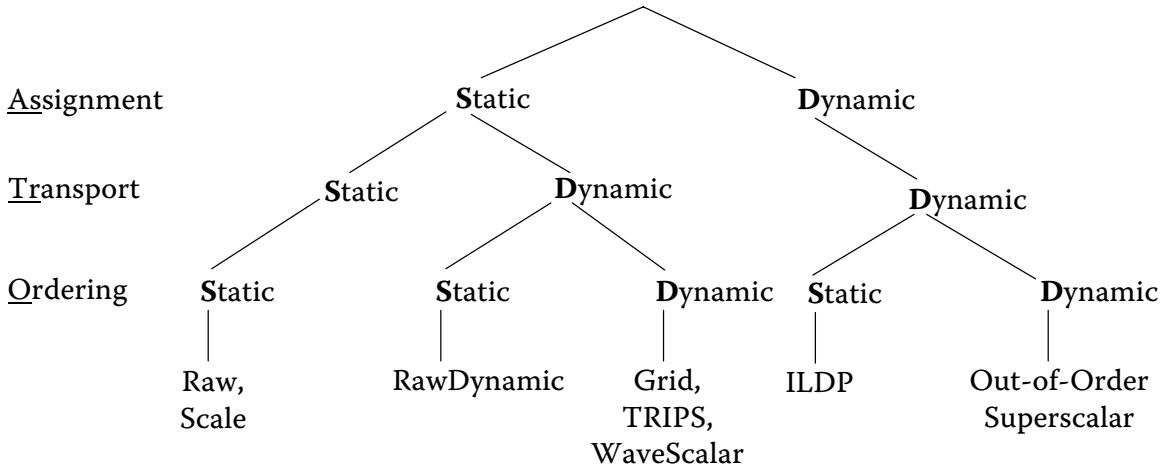


Figure 2-11: The AsTrO classification for Scalar Operand Networks: **As**signment, **Tr**ansport and **O**rdering. Of the systems we categorized, Raw [121], RawDynamic [112], and WaveScalar [105] are fully tiled, physically scalable architectures. Scale [66], Grid [87], TRIPS [96], and ILDP [60] are partially tiled architectures, and enjoy some but not all of the benefits of being physically scalable. Current out-of-order superscalars are untiled and lack physical scalability.

Nonetheless, many of the current research systems are, after careful consideration, neatly characterized by this taxonomy. Figure 2-11 categorizes a number of recent tiled (or "partially" tiled) proposals using the AsTrO taxonomy. We overview each of the categorized architectures briefly in the next few paragraphs. Note that architectures shown span the spectrum of being fully tiled and physically scalable (Raw, RawDynamic and Wavescalar), to being partially tiled and partially physically scalable (Scale, Grid, TRIPS, ILDP). We also include the out-of-order superscalar, which is not tiled and is not scalable.

ILDP [60] is a partially tiled microprocessor that uses a centralized fetch unit to issue groups of instructions dynamically to a number of distributed in-order cores that are connected via a broadcast-based transport network. Since several copies of the same instruction can be executing simultaneously on different in-order cores, ILDP qualifies as having a dynamic assignment SON. Because the SON is dynamic assignment, the order of operands traveling over the broadcast network can change in response to varying execution conditions. Therefore, it has a dynamic transport SON. Finally, each core executes instructions within an instruction group in-order, indicating an SON with static ordering. Thus, ILDP has an DDS SON.

TRIPS [96] (originally referred to as Grid [87]) is a partially tiled microprocessor that employs a unified fetch unit that maps a compiled array of instructions across an array of processing elements. These instructions execute in a data-flow fashion, and send their operands as they become available across a network to consuming instructions. Because it is the responsibility of TRIPS's compiler to assign instructions to individual nodes, TRIPS is a static assignment architecture. TRIPS has dynamic ordering because each node issues instructions in response to incoming operands, much like

an out-of-order superscalar. Finally, TRIPS forwards operands over a network as they are produced by the out-of-order nodes, thus it employs dynamic transport. Thus, TRIPS has an SDD SON.

RawDynamic [112] operates in much the same way as Raw, except that it employs a packet-switched dynamic SON that is controlled by incoming packet headers instead of a router instruction stream. It uses compile-time assignment of instructions to ALUs, so its SON is static assignment. Since the ordering of operands over links is dependent upon the arrival time of packets to the routers, RawDynamic employs a dynamic transport SON. Finally, RawDynamic, like Raw, employs an in-order compute pipeline (and logic to reorder incoming operands accordingly), so RawDynamic possesses a static ordering SON. Thus, RawDynamic has an SDS SON.

The Scale [66] partially-tiled architecture employs a 1-D array of vector lanes, each of which contains a number of clusters. These clusters and lanes receive their instructions from a centralized fetch-unit that broadcasts AIB ("atomic instruction blocks") across the lanes. These clusters and lanes are connected by an inter-lane interconnect with a ring topology, as well as an inter-cluster interconnect. In Scale, the assignment of AIBs (atomic instruction blocks) to virtual processing elements is determined at compile time, so the architecture is static assignment[2]. Because the ordering of operands on Scale's inter-lane and inter-cluster communication networks is fixed, Scale employs a static transport SON. Finally, because the execution of instructions on a virtual processing element is determined by ordering of instructions in the instruction stream, Scale SON uses static ordering. Thus, Scale has an SSS SON.

Overall, the bulk of the tiled or partially-tiled systems have focused upon static assignment architectures – the hypothesis being that achieving even partial physical scalability is easier if instructions are mapped to known places. Dynamic ordering architectures are often paired with dynamic transport, and static ordering architecture are often paired with static transport. The reasoning is relatively straight-forward – in dynamic assignment architectures, the hardware already exists to manage operands that arrive out-of-order on the network, so the cost of adding dynamic transport is low. Conversely, in static assignment architectures, the systems have been optimized around the fact that instruction orderings are predictable, so the lesser complexity and fewer levels of logic (presumably resulting in lower latency networks) entailed by static transport is attractive. Finally, the benefits that a dynamic assignment SON receives from out-of-order execution would be hampered by the in-order nature of static transport.

[2]Classifying Scale is a little subtle, because of its support for virtualization through a vector-style ISA. Because of this, operation assignment, transport ordering, and operation ordering do differ between machine implementations with different quantities of physical lanes. However, here the intent is to have the program adapt to varying machine configurations, rather than to have the program adapt to varying program timing conditions (such as cache misses). Assignment, transport and ordering are all compiler-determined on a given Scale implementation. Nonetheless, the Scale approach is interesting because it achieves one of the benefits of dynamic behavior – running existing binaries across a spectrum of implementations with different quantities of resources) without the full hardware overhead of the full dynamic mechanisms.

2.4.4 The 5-tuple Metric for SONs

How do we evaluate the basic goodness of a scalar operand network? How is an SON differentiated from conventional microprocessor networks? How do we parameterize SONs so that we can determine program sensitivity to SON properties? It is these questions that drove us to develop the *5-tuple performance metric* for scalar operand networks.

In order to address these questions, we decided that it was necessary to create a way of describing the cost of the most fundamental part of an SON: operation-operand matching. For each machine, the 5-tuples encapsulate the complete end-to-end cost of routing an operand from the output of one ALU to the input of a remote ALU. Defining the performance metric in this way allows us to compare operation-operand matching across a spectrum of network types, including both SONs and multiprocessor networks.

More formally, we define the performance 5-tuple as a 5-tuple of costs <SO, SL, NHL, RL, RO>:

Send occupancy (SO)	average number of cycles that an ALU on the source tile wastes in transmitting an operand to dependent instructions at other ALUs.
Send latency (SL)	average number of cycles incurred by the message at the send side of the network without consuming ALU cycles.
Network hop latency (NHL)	average transport network hop latency, in cycles, between physically adjacent ALUs
Receive latency (RL)	average number of cycles between when the final input to a consuming instruction arrives at the receiving tile and when that instruction is issued.
Receive occupancy (RO)	average number of cycles that an ALU on the receiving tile wastes by using a remote value.

Figure 2-12 depicts graphically the components of the operand transmission that the 5-tuple incorporates. Informally speaking, *send occupancy* incorporates the opportunity cost of instructions that the sender incurs by initiating a communication with a remote node. The *send latency* incorporates the latency of message injection – this cost is simply the delay before the operand "hits the network." The transport cost is used to express the actual cost of routing the operand through the network. It is given in the form of a per-hop *network hop latency*[3] rather than network diameter (as

[3]The use of a per-hop latency raises some issues when comparing network with greatly varying topologies. Unless otherwise specified, 5-tuples apply to 2-D meshes and other topologies for which nodes are embedded uniformly embedded in 2-D space and whose latencies, in cycles, are roughly proportional to the physical distances between nodes. This is a reasonable assumption for on-chip networks in modern VLSI processes, for which 2D node packing minimizes communication latency and for which wire delay is significant in comparison to router delay. This wire delay prevents physically-scalable hyper-dimensional topologies from reducing effective network diameter, in cycles, beyond a constant factor of the underlying physical topology. Networks with constant communication costs between nodes (e.g., crossbars or multistage networks) can be modeled in the 5-tuple by counting their fixed inter-node latency as part of the send latency and setting NHL to 0. Then their 5-tuples can be directly compared to that of all other networks regardless of packing dimensionality. One approximate approach for normalizing NHL latency of more exotic topologies to a mesh is to graph the number of nodes which are mutually reachable in the network for a given number of cycles, and find the mesh 5-tuple whose corresponding graph most closely matches this curve.

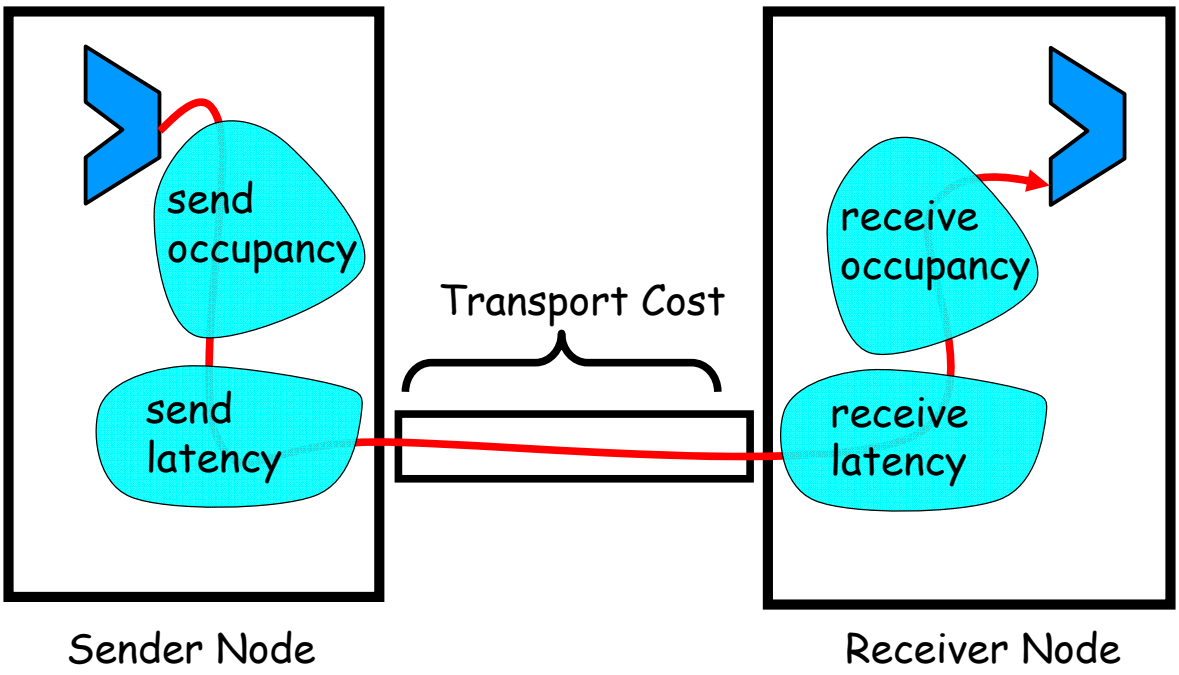


Figure 2-12: Graphical depiction of the 5-tuple, which characterizes the end-to-end costs of routing an operand from the output of one ALU to the input of a remote ALU.

in LogP [21]) in order to emphasize the importance of locality in tiled microprocessors. The *receive latency* is the injection delay associated with pulling the operand in from the transport network at the receiving node. The *receive occupancy* incorporates the opportunity cost that the receiver incurs by receiving an operand from the sender. Generally speaking, all of the costs are exclusive. So, for instance, if the sender has to execute an instruction that occupies one issue slot but takes 30 cycles (including a cycle to issue the instruction) for the message to appear on the network, this would be notated as a send occupancy of 1 and a send latency of 29.

For reference, these five components typically add up to tens to hundreds of cycles [68, 67] on a conventional multiprocessor. In contrast, all five components in conventional superscalar bypass networks add up to zero cycles! The challenge of tiled microprocessor design is to realize efficient operation-operand matching systems that also scale.

In the following subsections, we examine SONs implemented on a number of conventional systems and describe the components that contribute to the 5-tuple for that system. At one end of the spectrum, we consider superscalars, which have perfect 5-tuples, <0,0,0,0,0>, but are not physically scalable. On the other end of the spectrum, we examine message passing, shared memory and systolic systems, which have physically scalable implementations but poor 5-tuples. Tiled microprocessors, such as Raw, strive to incorporate the favorable aspects of both types of systems, achieving physical scalability and a 5-tuple that comes closer to that of the superscalar. For instance, Raw's SON attains a 5-tuple of <0,0,1,2,0>.

2.4.4.1 Example: Estimating the 5-tuple for Superscalar operation-operand matching

Out-of-order superscalars achieve operation-operand matching via the instruction window and result buses of the processor's SON. The routing information required to match up the operations is inferred from the instruction stream and routed, invisible to the programmer, with the instructions and operands. Beyond the occasional move instruction (say in a software-pipelined loop, or between the integer and floating point register files, or to/from functional-unit specific registers), superscalars do not incur send or receive occupancy. Superscalars tend not to incur send latency, unless a functional unit loses out in a result bus arbitration. Receive latency is often eliminated by waking up the instruction before the incoming value has arrived, so that the instruction can grab its inputs from the result buses as it enters the ALU. This optimization requires that wakeup information be sent earlier than the result values. Thus, in total, the low-issue superscalars have perfect 5-tuples, i.e., <0,0,0,0,0>. We note that non-zero network latencies have begun to appear in recent wider-issue superscalar designs such as the Alpha 21264 [59] and Power4 [114].

2.4.4.2 Example: Estimating the 5-tuple of Multiprocessor operation-operand matching

One of the unique issues with multiprocessor operation-operand matching is the tension between commit point and communication latency. Uniprocessor designs tend to execute early and speculatively and defer commit points until much later. When these uniprocessors are integrated into multiprocessor systems, all potential communication must be deferred until the relevant instructions have reached the commit point. In a modern-day superscalar, this deferral means that there could be tens or hundreds of cycles that pass between the time that a sender instruction executes and the time at which it can legitimately send its value on to the consuming node. We call the time it takes for an instruction to commit the *commit latency*. Barring support for speculative sends and receives (as with a superscalar!), the send latency of these networks will be adversely impacted.

The two key multiprocessors communication mechanisms are message passing and shared memory. It is instructive to examine the 5-tuples of these systems. As detailed in [111], the 5-tuple of an SON based on Raw's relatively aggressive on-chip message-passing implementation falls between <3,1+c,1,2,7>and <3,2+c,1,2,12>(referred to subsequently as *MsgFast* and *MsgSlow*) with c being the commit latency of the processor. A shared-memory chip-multiprocessor SON implementation based on Power4, but augmented with full/empty bits, is estimated to have a 5-tuple of <1,14+c,2,14,1>. Later, we were able to confirm experimentally that the actual Power4 has a 5-tuple of <2,14,0,14,4>. For completeness, we also examine a systolic array SON implementation, iWarp, with a 5-tuple of <1,6,5,0,1>.

Message-passing operation-operand matching For this discussion, we assume that a dynamic transport network [22] is used to *transport* operands between nodes. Implementing operation-

operand matching using a message-passing style network has two key challenges.

First, nodes need a processor-network interface that allows low-overhead sends and receives of operands. In an instruction-mapped interface, special send and receive instructions are used for communication; in a register-mapped interface, special register names correspond to communication ports. Using either interface, the sender must specify the destination(s) of the out-going operands. (Recall that the superscalar uses indiscriminate broadcasting to solve this problem.) There are a variety of methods for specifying this information. For instruction-mapped interfaces, the send instruction can leave encoding space (the log of the maximum number of nodes) or take a parameter to specify the destination node. For register-mapped interfaces, an additional word may have to be sent to specify the destination. Finally, dynamic transport networks typically do not support multicast, so multiple message sends may be required for operands that have non-unit fanout. These factors will impact send and receive occupancy.

Second, receiving nodes must match incoming operands with the appropriate instruction. Because timing variances due to I/O, cache misses, and interrupts can delay nodes arbitrarily, there is no set arrival order for operands sent over dynamic transport. Thus, a tag must be sent along with each operand. When the operand arrives at the destination, it needs to be demultiplexed to align with the *ordering* of instructions in the receiver instruction stream. Conventional message-passing implementations must do this in software [120], or in a combination of hardware and software [76], causing a considerable receive occupancy.

Shared-memory operation-operand matching On a shared-memory multiprocessor, operation-operand matching can be implemented via a large software-managed operand buffer in cache RAM. Each communication edge between sender and receiver could be assigned a memory location that has a full/empty bit. In order to support multiple simultaneous dynamic instantiations of an edge when executing loops, a base register could be incremented on loop iteration. The sender processor would execute a special store instruction that stores the outgoing value and sets the full/empty bit. The readers of a memory location would execute a special load instruction that blocks until the full/empty bit is set, then returns the written value. This would eliminate branch misprediction penalties inherent in polling. Every so often, all of the processors would synchronize so that they can reuse the operand buffer. A special mechanism could flip the sense of the full/empty bit so that the bits would not have to be cleared.

The send and receive occupancy of this approach are difficult to evaluate. The sender's store instruction and receiver's load instruction only occupy a single instruction slot; however, the processors may still incur an occupancy cost due to limitations on the number of outstanding loads and stores. The send latency is the latency of a store instruction plus the commit latency. The receive latency includes the delay of the load instruction as well as the non-network time required for the cache protocols to process the receiver's request for the line from the sender's cache.

This approach has number of benefits: First, it supports multicast (although not in a way that saves bandwidth over multiple unicasts). Second, it allows a very large number of live operands due to the fact that the physical register file is being implemented in the cache. Finally, because the memory address is effectively a tag for the value, no software instructions are required for demultiplexing. In [111], we estimated the 5-tuple of this relatively aggressive shared-memory SON implementation to be <1,14+c,2,14,1>. Subsequently, experimentally, we were able to measure the 5-tuple between two cores on a 1.3 GHz Power4 node (in which the two cores are located on the same die and are connected via shared memory through a shared on-chip L2) as being <2,14,0,14,4> – under the optimistic assumption that the top bit of each word is unused and can be used as a full-empty bit.

Systolic array operation-operand matching Systolic machines like iWarp [38] were some of the first systems to achieve low-overhead operation-operand matching in large-scale systems. iWarp sported register-mapped communication, although it is optimized for transmitting streams of data rather than individual scalar values. The programming model supported a small number of pre-compiled communication patterns (no more than 20 communications streams could pass through a single node). For the purposes of operation-operand matching, each stream corresponded to a logical connection between two nodes. Because values from different sources would arrive via different logical streams and values sent from one source would be implicitly ordered, iWarp had efficient operation-operand matching. It needed only execute an instruction to change the current input stream if necessary, and then use the appropriate register designator. Similarly, for sending, iWarp would optionally have to change the output stream and then write the value using the appropriate register designator. Unfortunately, the iWarp system is limited in its ability to facilitate ILP communication by the hardware limit on the number of communication patterns, and by the relatively large cost of establishing new communication channels. Thus, the iWarp model works well for stream-type bulk data transfers between senders and receivers, but it is less suited to ILP communication. With ILP, large numbers of scalar data values must be communicated with very low latency in irregular communication patterns. iWarp's 5-tuple can modeled as <1,6,5,0,1> - one cycle of occupancy for sender stream change, six cycles to exit the node, four or six cycles per hop, approximately 0 cycles receive latency, and 1 cycle of receive occupancy. An on-chip version of iWarp would probably incur a smaller per-hop latency but a larger send latency because, like a multiprocessor, it must incur the commit latency cost before it releases data into the network.

2.4.4.3 5-tuple for the Archetypal Tiled Microprocessor

What 5-tuple should the Archetypal Tiled microprocessor possess? Ideally, the components would be as close to zero as physics allows. However, the network hop latency ("NHL") should not be zero because the absence of a distance-related cost for routing operands implies that the machine

is not physically scalable. How do we know that the NHL of our ATM is respectable? One way is to measure the percentage of the latency due to wire (and repeater) delay. If the percentage is relatively high (for instance, 40%), then we know that overhead of intermediate routing is low.

Furthermore, in the ATM, we would like the send and receive occupancy to be zero. The reason is that these parameters have a large effect on performance (see [111]), and yet it is relatively easy, through the use of parallel hardware and appropriate instruction set support, to eliminate these costs. The communication hardware can run independently of the tile's compute pipeline. Send and receive occupancy have a large negative impact on performance because they penalize nodes for communicating. For instance, in Figure 2-10, the speedup attained on the application is attained because we are able to offload two of the instructions from the upper-left hand tile ("tile 0") to the lower-left hand tile ("tile 1"). The existence of a single cycle of receive occupancy would make it unprofitable to execute instruction 2 on tile 1, since it would be just as cheap for tile 0 to execute the instruction itself as to receive the operand. Similarly, the existence of a single cycle of send occupancy would make it unprofitable to execute instruction 7 on the tile 1 because it would as cheap for tile 0 to execute the instruction itself as it would to send the operand to the tile 1.

The ideal send and receive latency parameters are more difficult to discern. Looking back on our experiences with the Raw prototype, it appears that it is possible to implement an SSS SON with zero send latency and unit receive latency. For other SONs from other parts of the AsTrO taxonomy, it remains to be seen how low the latencies can go, since the additional sophistication of these SONs tends to incur additional levels of logic, which will result in either lower cycle times or higher latencies. On the other, the additional dynamicity of these architectures may allow them to compensate for slower SON operation-operand matching performance. As the research matures and more implementations emerge, more will become known.

2.5 Parallel Execution on Tiled Microprocessors

The previous section discussed scalar operand networks and how they facilitate fast communication between remote functional units. In this section, we examine the larger picture of how the SON and other resources are programmed to achieve the goal of parallel execution – both for single-threaded and multi-thread programs. In this section, we examine the general themes in compilation for fully-tiled architectures such as the ATM – the reader can refer to the description of Raw's compilers in Section 5 and [71, 37, 7, 72, 70, 3, 58, 116, 36, 82, 86, 100, 18] for further details about compiling for specific architectures or languages.

2.5.1 Multi-Threaded Parallel Execution on Tiled Microprocessors

From Section 2.1 we know that every tile has its own program counter, fetch unit and instruction cache. The self-sufficiency of tiles is part of the physical scalability argument, because it eliminates many of the centralized components – such as fetch units, control broadcast networks and global barriers – that contribute to scalability problems. Because of this self-sufficiency, tiles have the ability to operate independently in addition to in tightly-coupled groups.

As a result, this allows the ATM to support a naturally threaded mode of execution much like that found in multiprocessor systems – groups of tiles can be organized into threads in which each thread operates autonomously with respect to the other threads. Of course, should the need arise, separate threads can periodically synchronize, through the on-chip networks, or through memory. Generally speaking, in the threaded model, there is a notion of resources that are associated with a particular thread. In the ATM, at a minimum these resources include a set of program counters, fetch units and functional units. However, it may also be necessary to partition the SON, that is, to divide the resources of the SON among the threads. This is particularly true if the SON is a static transport SON, where separate threads with no reconciliation of control-flow will not be able to easily derive a unified communication schedule. On the other hand, a dynamic transport SON may very well be able to share large parts of the SON between threads, although there may be some negative performance effects due to contention. This "partition the resources and periodically synchronize" mode of usage is much like that of a chip-multiprocessor, although the latencies are typically smaller if the SON is employed for communication.

2.5.2 Single-Threaded Parallel Execution on Tiled Microprocessors

Within a thread, ATM tiles operate in a much more tightly coupled fashion – executing the same instruction stream with a notion of a shared – but possibly slightly decoupled – control-flow structure. In this model, the instruction streams for each tile are typically created with mutual knowledge of the other tiles' instruction streams. Because of this shared knowledge of control-flow structure, the SON communication patterns can be quite rich, complex and fine-grained in comparison to conventional multi-threaded programs which are programmed without detailed knowledge of the instruction streams of the other threads. For conventional threaded programs, the control flow structure of different physical threads can be wildly diverging, but communication (over SON or otherwise) is typically quite structured (e.g., through homogeneous streaming channels, or through the use of meta-data to describe messages) – otherwise neither thread would know when or what the other thread was transmitting.

Between threads, much of the existing knowledge in multiprocessors directly applies to the ATM. It is for a multi-tile execution of a single thread (such as in Figure 2-10) that we are faced with the

task of orchestrating these otherwise self-sufficient tiles (and their SONs) within a common control-flow context. Generally speaking, control-flow instructions are replicated to all tiles that have been assigned non-control instructions that are guarded by those control-flow instructions. Then, it remains to multicast the branch condition, if necessary, to the replicated instructions. Fortunately, the scalar operand network provides a fast way of communicating this information. Control-flow values (e.g., branch conditions) can easily be transmitted over the scalar operand network just as any other type of data.

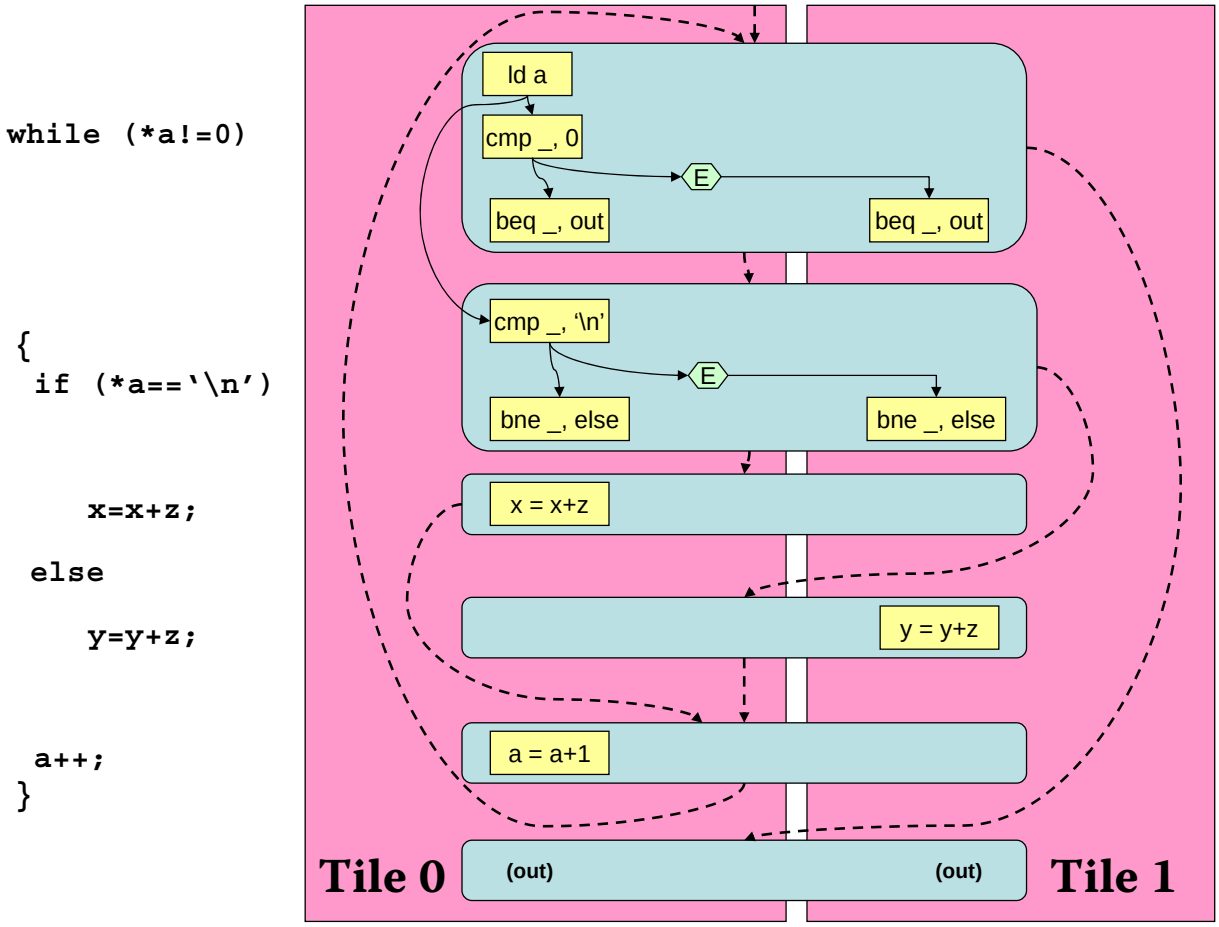


Figure 2-13: A program segment and a mapping of its control flow graph ("CFG") to two tiles. The rectangles with rounded edges are the CFG nodes. The dotted arcs are the CFG edges. The white boxes are instructions, which belong to one CFG node.

Figure 2-13 shows an example of a control-flow intensive program segment and a mapping of its *control flow graph* ("CFG") to two tiles. The CFG's nodes, represented using rectangles with rounded corners, correspond to basic blocks in the program. A basic block is a single-entry, single-exit region of the program. Each instruction in the original program segment belongs to one of the CFG's nodes. An important invariant of control flow graph execution is that if the flow of control proceeds upon one of the CFG's outbound edges, then all of the operations in the CFG have

executed.

We describing the process of mapping of a CFG to an array of ATM tiles in the following subsections on instruction assignment, scalar assignment, route assignment, and instruction scheduling.

2.5.2.1 Instruction Assignment

The process of mapping a CFG to an array of tiles begins with the assignment of instructions to tiles, as described in this section. First, each node of the control-flow graph is conceptually mapped across every tile in the array, regardless of whether that tile is used in the computation. Then, each instruction in each CFG node is assigned to a single tile, with one exception – conditional control flow instructions, of which there can be at most one per CFG node, are mapped to all of the tiles.

Once every instruction has been mapped, the challenge is to manage the flow of scalar values within the control flow graph. If all sources and destinations of the scalar value reside in the same control flow block, then the process is much as we've seen in the previous discussion of scalar operand networks. The appropriate route operations are created and assigned to the same CFG node as the sending and receiving instructions. This ensures that they are executed accordingly with the other operations in the same CFG node.

In the case where the sources and/or destinations of scalar values do not reside in the same control flow block, the process is somewhat more involved, especially because there may be multiple entry and exit arcs to a given control flow node. In essence, we need to perform a task that is similar in spirit to register allocation. We call this task *scalar assignment*.

2.5.2.2 Scalar Assignment

The purpose of scalar assignment is to determine the "hand-off locations" for scalar variables between CFG nodes. Scalar assignment takes as input a set of *live-ins* and *live-outs* for each CFG node. These are typically determined through a compiler data-flow analysis. Live-ins are variables whose values are still possibly needed in the computation, and are valid upon entrance to the control flow block. Live-outs are variables whose values are still possibly needed in a subsequent phase of the computation upon exiting a control flow block. The job of scalar assignment is to create a contract, for each control-flow block, of the expected location(s) of live-ins and live-outs – these may be in a particular register in the register-file, or in a particular network buffer, or at a particular memory location[4]. Each live-in or live-out may have more than one location, to improve locality.

Scalar assignment must further ensure that the contracts between CFG nodes are consistent. For each control-flow edge (the dotted lines in Figure 2-13), the live-outs of the source must meet (or

[4]To simplify the compiler implementation, scalar assignment may make use of virtual registers or virtual buffers to defer the assignment of storage locations until a later phase of the compiler. This is especially practical if a backing store exists (e.g., spill instructions and a cache) that is guaranteed to be accessible in spite of any resources the compiler may have reserved for other parts of the computation.

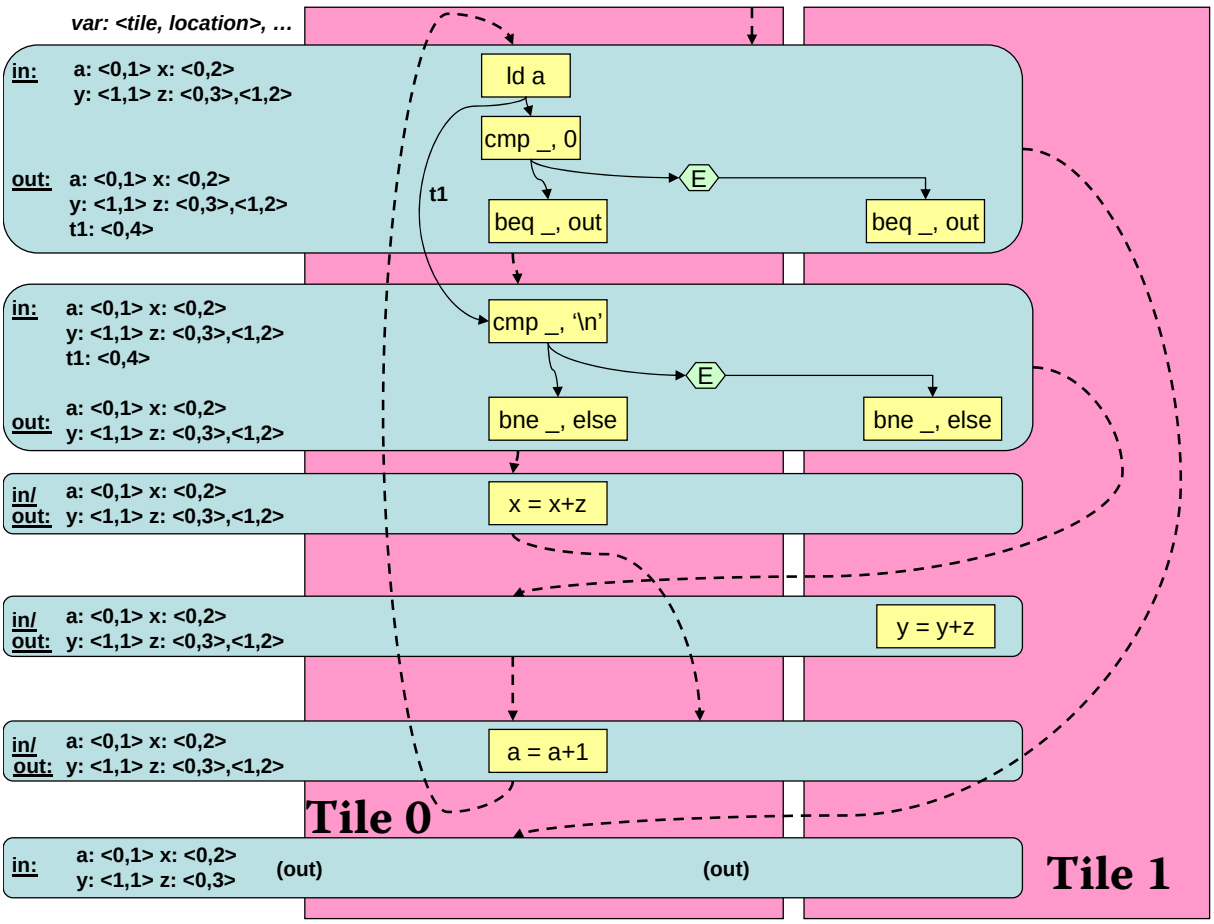


Figure 2-14: A valid scalar assignment for a control flow graph. Each CFG node's live-out and live-in operands have been assigned one or more locations. Furthermore, live-out and live-in locations of connected CFG nodes are consistent.

exceed) all of the expectations of the live-ins of the destination node. For instance, if the destination node expects variable z to be placed in register 3 on tile 0, then the source would satisfy this if, for instance, its live-out list had variable z in register 3 on tile 0, and register 2 on tile 1. In some cases, scalar assignment may chose to insert an additional new "stitch" control flow node on a given control flow edge in order to transition between CFG nodes which have different ideal scalar assignments – for instance, for CFG nodes belonging to different loops that have different ideal scalar placement. In this case, the original source node will have an output edge going to the stitch CFG node, which in turn will have an output edge going to the original destination node. The CFG node will be given live-in and live-out signatures that are compatible with the CFG nodes it connects to.

Scalar assignment can also be used to exchange data that is "in flight" between control flow nodes – that is, a scalar value's assigned location can be a FIFO on the SON rather than a location in a tile's local register file. In this case, the interconnect semantics are somewhat more challenging – namely, unlike with values stored in the register file, excess operands can have the side effect of

"blocking" other operands from passing through. In this case, the live-in and live-out consistency must be enforced more strictly – not only must the live-outs of the source be a superset of the live-ins of the destination, but vice versa as well. Put another way, the FIFO-resident live-out and live-in scalar assignments must match exactly. In this case, the insertion of "stitch" CFG nodes can also be used to dequeue unnecessary elements upon transition between control flow nodes.

Figure 2-14 shows an example of consistent live-in and live-out annotations for the original graph in Figure 2-13. Note that there are two copies of the variable `z` in the loop, which reduces communication latency.

How are good scalar assignments determined? Ideally, the ideal locations will minimize the net occupancies and latencies involved in transmitting live-in values to remote instructions that require them – an interesting compiler optimization problem.

2.5.2.3 Route Assignment

After scalar assignment has been run, the compiler can now focus exclusively on one CFG node at a time. It must perform *route assignment*. First, it must assign route operations to tiles in order to route values from live-ins to consuming instructions. Since a given variable may have multiple live-in locations, it is up to the compiler to choose the one that best optimizes program execution. Second, the compiler must generate routes that go between dependent instructions. Third, the compiler must generate routes that go from the last definition of the scalar value (which may be a live-in) to all of the live-out locations. Finally, if the SON has a concept of in-network state, route assignment must perform the task of ensuring that unwanted operands are not left "hanging" in the network unless this is specified by the live-out contract. A key idea is that, because of the use of the scalar assignment pass, code generation occurs within control-flow nodes as opposed to across them. Each route operation has a definite CFG node that it is associated with. This is because inter-CFG routes have been divided, through scalar assignment, into a series of intra-CFG routes.

2.5.2.4 Scheduling and Code Generation

At this point, route operations have been assigned and all instructions have been assigned to tiles. Next, we want to schedule the operations within each CFG node, taking into account instruction and routing latencies[5]. We may also need to perform register allocation to map local virtual registers (assigned through scalar assignment) to a tile's local register file and stack. We may also have some final optimizations we want to perform. For instance, we may want perform per-tile optimization with respect to minimizing unnecessary control flow – for instance, the elimination of unconditional jump instructions, or eliminating conditional branches that do not guard any instructions.

[5]Of course, static ordering SONs benefit the most from scheduling; however, dynamic ordering SONs also benefit from an informed ordering of the instruction stream.

However, to a large degree, overall program performance has already been determined – once the program has been sliced and diced across the array of tiles, the tiles are often so inter-dependent that improving one tile's performance often will not improve the schedule of the whole. As a result, it is desirable to design a tiled microprocessor's instruction set so that most peephole optimizations can be performed before assignment and routing are performed.

At this point, it remains to "render the binary" – to generate instruction streams for each tile that ensure that each instruction and route is contained within the surrounding control flow instructions. Because we have assigned route operations to their corresponding control-flow blocks, we can insure that route operations will be executed in their correct contexts – even if they are the only operation that executes on a given tile in a given control flow. This guarantee is particularly important in an architecture with static transport, in which the routers need to be programmed with route operations. Furthermore, because there are consistent live-in and live-out contracts between CFG nodes due to scalar assignment, we know that scalar values are correctly orchestrated in the program.

2.6 The ATM Memory System

In the previous section, we described many aspects of compilation and execution for tiled microprocessor, including how control flow is managed. In this section, we examine the memory system of the ATM . Although we have already specified that each tile in an ATM has its own data-cache, there are a number of interesting issues that arise in the ATM. In this section, we address the following questions. How do the caches interact with the external memory system and each other? How do tiled systems support a memory hierarchy? How are memory dependences handled between tiles?

First, we examine how cache misses are treated in the system. During the common case, memory operations (such as load or store instructions) find their corresponding values in the local cache of a tile. Should the value not be in the cache, then the cache miss logic is responsible for retrieving the corresponding cache line. In the most basic tiled microprocessor, cache misses cause requests (typically a packet with one or more words) to be sent out over a point-to-point on-chip network (a generalized transport network) to I/O ports which are in turn connected to off-chip DRAM. Since the tiled microprocessor may have several I/O ports, the cache miss logic must use a *memory hash function* to determine where to send the request. The hash function takes an address and converts it to a destination which is used in the header of the cache miss message. Thus, the hash function effectively partitions the address space across the DRAM ports. Should the tile support multiple outstanding cache-miss messages, it may also include a *sequence number* which is used as a way to disambiguate message responses, which may return out-of-order depending on network and DRAM congestion and latency. Since the memory configuration (such as the size and number of

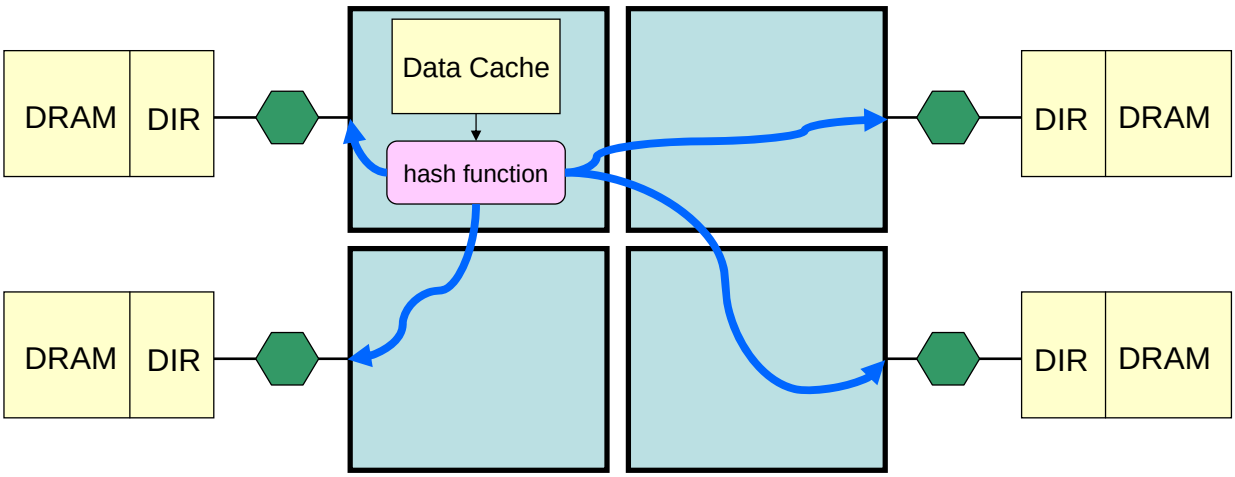


Figure 2-15: Memory system for a basic tiled microprocessor. Each tile has a local data cache and a configurable hash function, which maps addresses to I/O port addresses. The cache miss requests are packaged into a message, which is transmitted over a point-to-point on-chip network to the I/O ports. At the I/O ports are a number of DRAMs and memory controllers that optionally implement a directory-based coherence protocol.

DRAMs) may vary between machines, it is common to allow this hash function to be configured by the operating system, or even the application. Since this hash function operates much like a TLB, it may also be convenient, but not mandatory, to combine it with the functionality of a TLB.

Figure 2-15 illustrates this idea. It depicts an array of four tiles. Inside each tile is a local data cache and a hash function that maps addresses to I/O ports. Also pictured next to each DRAM is an optional cache-coherence controller (marked "DIR") which implements directory-based cache-coherence for facilitating a shared-memory programming model between multiple tiles.

There are several undesirable scalability characteristics with the basic mechanism of Figure 2-15. First, as the number of tiles is increased, the distance that cache miss messages must travel increases. These increased distances in turn increase the average latency and average bisection bandwidth requirements of each cache miss. Second, since I/O bandwidth does not tend to scale with transistor count, it is likely that there will be relatively high contention on I/O ports as the system is scaled up. Third, with subsequent generations of Moore's law, the speed of logic transistors is likely to increase relative to the speed of DRAM accesses. As a result, we can expect that DRAM latencies, measured in terms of clock cycles, will increase. All of these factors (which are effectively architectural scalability problems caused by a failure to exploit locality) argue for having larger caches in the tiles, to reduce the average latency and bandwidth requirements of memory accesses. However, if we increase tile cache sizes to address this problem, then we introduce a physical scalability problem – the cycle time of the processor, normalized for process (e.g., by the FO4 metric), would increase. An alternative is to introduce hierarchy into the tile's caches. Although this will eliminate the physical scalability issues at the cache level, it will cause tiles to grow larger, which in turn will force the

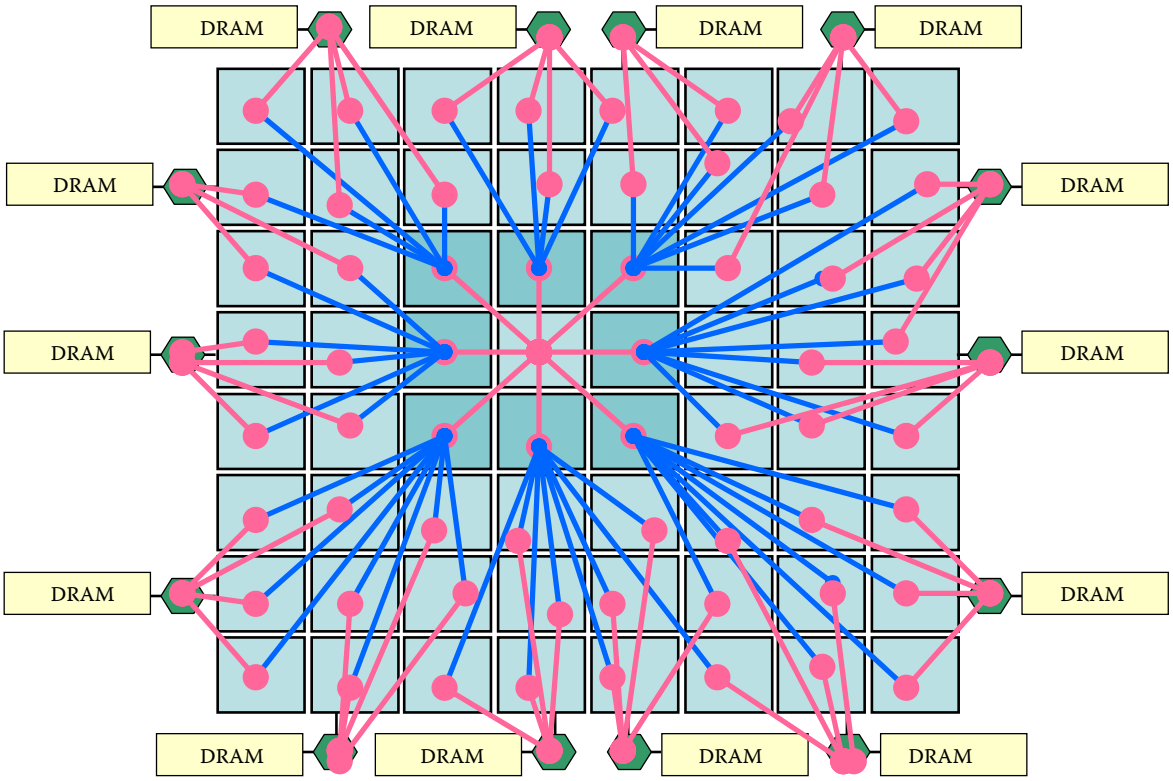


Figure 2-16: Virtual caches: using tiles to implement a memory hierarchy in the ATM to attain physical scalability. In this case, there is one compute tile (with a single L1 cache), 8 tiles at the L2 level, and 55 tiles at the L3 level. Were there an additional compute tile, coherence would be maintained at the L2 level; the tiles would share all levels of the cache hierarchy at and below L2, with invalidations and flush requests being sent up to the individual tiles' L1 caches.

inter-tile latencies to increase. Furthermore, it will reduce the proportion of the chip dedicated to computation relative to cache, which penalizes those programs which have good locality and do not need a higher relative percentage of cache.

The ATM addresses these problems by using the tiles themselves to implement hierarchical caches, as determined by the needs of the workload. By generalizing the hash function so that it can target both external I/O ports and internal tiles, the ATM gains the ability to employ groups of tiles as additional cache memory. This way, depending on the memory characteristics of the program, on how much parallelism is present, and on how many programs are being run on the ATM, more or fewer resources can be dedicated to on-chip cache for the program.

Figure 2-16 shows an example multi-level cache configuration for a program which has extremely high working sets and low parallelism. This configuration is determined by software. In this case, the center tile is the only one performing computation. Its memory hash function is configured to hash addresses to eight different tiles. These eight tiles use their local caches to implement a distributed L2 implementation with roughly eight times the capacity of the center tile. When these tiles do not contain the cache line in question, they in turn consult their configurable hash function

(or a software routine) to refer the requests to the next level of cache, which is also implemented using tiles. In total, eight tiles are used to implement the L2, and 55 tiles are used to implement the L3. The L3 tiles in turn have their hash function configured to send misses to external DRAM ports.

Clearly, there is a large design space for implementing caches out of tiles; and we will leave the details largely unspecified. However, there are a few items that are worth mentioning. First, the optimal topology of these virtual caches is dependent on the relative latencies of the cache miss hardware and the transport network's hop cost. The lower the hop cost relative to the cost of processing the misses in the cache miss hardware, the fewer the levels of hierarchy (and the greater the number of tiles per level) in the virtual cache.

Second, when a virtual cache is employed by multiple client compute tiles that also make use of cache-coherent shared memory support, there are a number of alternatives for implementing coherence. One possibility is to pick a level at which coherence is done (e.g., L2), and then to make all levels of cache (e.g., L1) above this level private to individual tiles (but possibly containing the same data in a "shared" state.) All levels below (e.g., L2, L3, ...) are used in common by the client compute tiles. Then, the L2 tiles' caches cache both program data and the directory information needed to describe the sharing relationships of the client tiles' private (e.g., L1) caches. Finally, since tiles are used to implement successively deeper levels of the memory hierarchy, there are a host of trade-offs between what is implemented in hardware (simple but fast and lower power) versus software (more sophisticated, slower, and reconfigurable).

2.6.1 Supporting Memory Parallelism

Tiles in the ATM system employ multiple caches and can optionally use standard distributed directory-based cache-coherency protocols to maintain coherence. In the case where tiles are being used to execute independent threads, memory parallelism operates much like in shared memory multiprocessor systems – it is the programmer's responsibility to insert the appropriate synchronization and/or cache-control primitives to ensure mutual exclusion as well as dependency, coherence and consistency constraints for memory accesses on different tiles.

However, in the case where multiple ATM tiles are being used to execute a single sequential program in parallel, the burden of enforcing memory semantics lies upon the compiler and architecture.

In a sequential program, memory operations are ordered by their appearance order in a sequential execution of the instruction stream – the value read from a given address is the value that was stored in the most recent store instruction. However, when a single sequential program is being executed in parallel by multiple tiles, maintaining these semantics can be more challenging, since tiles do not execute the program's instructions in strictly lock-step order.

One solution to this problem is to map all memory operations to a single tile[6], and to use traditional uniprocessor techniques to enforce memory dependences. However, this has several disadvantages. First, it means that the program is only utilizing the local cache memory of a single tile. Second, it restricts the ability of tiles to exploit parallelism. For example, if on average one-in-five instructions is a memory operation, then there would be little benefit to using more than five tiles to execute a program. Clearly such an approach is not scalable.

A more desirable solution is to be able to map memory operations to different tiles, allowing us to exploit multiple tiles' data caches and memory ports. To achieve this goal and ensure program correctness, we need to find a way to make sure that the program adheres to the memory dependences contained in the original sequential program. One possibility is to map memory operations to different tiles, and then use a token, communicated over a point-to-point network such as the SON, to signal to the next memory operation that the previous operation has completed. Although this has the benefit that it distributes memory objects across caches, it has two disadvantages. First, it increases the effective latency of memory operations because the program now has to pay the cost of transmitting the token between memory operations. Second, it does not achieve our goal of being able to execute memory accesses in parallel. Third, if this mapping is done without regard to temporal locality, cache lines may be prone to "ping-ponging" between different tiles that are accessing the same address.

The key to attaining memory parallelism is to take advantage of the fact that not all memory operations need to be ordered with respect to each other – only those memory operations which have intersecting addresses need to be ordered[7]. Thus, to parallelize memory operations across tiles, we want to divide memory operations into groups of operations such that no two groups access the same address [8, 9]. Each group can be assigned to a single tile, and can proceed without synchronizing with memory operations in other groups[8]. Periodically, we may migrate a given group from one tile to another, in which case a message can be sent over the SON from the original tile to the new tile after the original tile has confirmed that all previous operations have completed. Cache coherence or explicit cache management can handle the migration of actual data objects between tiles at migration points. As the program transitions between phases, we may reformulate the groups, so that a new set of groups is formed that repartitions the address space. A similar (but more comprehensive) synchronization is necessary for this reconfiguration. In many cases a barrier (which can be done quickly with most SONs) may be the most efficient way to achieve this. Once again, cache coherence or explicit cache management can handle the actual transfer of data.

[6] Or to a centralized hardware unit that orders memory operations.

[7] Memory consistency (as opposed to memory coherence) for inter-thread communication can be handled through the insertion of explicit constructs that inform the compiler of the need to enforce ordering of memory operations that target different addresses. The compiler can then insert synchronization communications over the SON to ensure that all memory operations proceed in order. This use of explicit constructs to enforce compiler ordering is common in many current multiprocessor systems that have relaxed memory consistency models, and is not unique to the ATM.

[8] Of course, the tiles must continue to ensure intra-tile memory ordering within a group.

This process allows us to distribute memory objects across the caches of different tiles and to sustain parallel execution of the memory operations. At the same time, dependent memory operations are located on the same tile, which makes enforcement of memory ordering fast and efficient. Finally, the assignment of groups of addresses to individual tiles helps reduce the frequency of cache ping-ponging.

2.6.2 Compiler enhancement of Memory Parallelism in Single Threaded Programs

Both compiler analyses and transformations play an important role in enabling memory parallelism in single-threaded programs on tiled microprocessors. The compiler needs to examine the input program and deduce which memory operations in the program may refer to the same addresses (whether they *may alias*). This deduction can be performed through a variety of analyses. For instance, the compiler may deduce that two addresses have different types, and thus cannot alias. Or it may determine that two addresses are referenced using different offsets from the same pointer or array. Although alias analysis is still an active area of research, tiled architectures provide a way of turning these advanced techniques into performance.

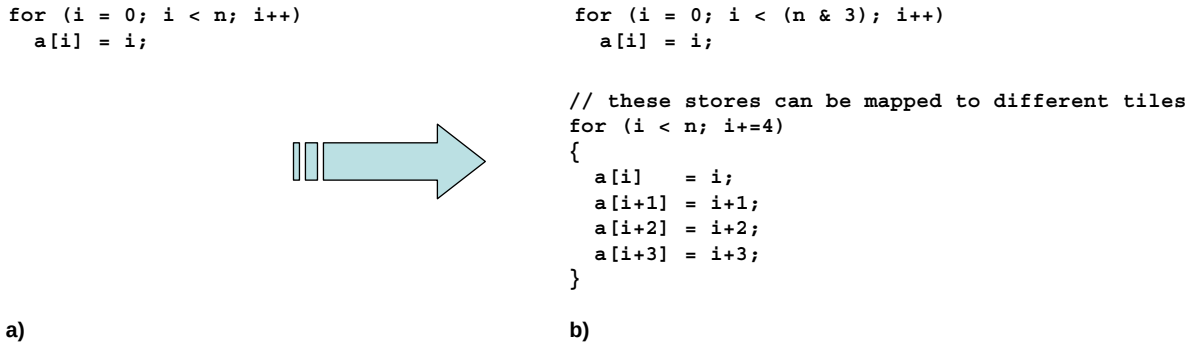


Figure 2-17: An example of a code transformation that safely increases memory parallelism. The code is transformed in a way that duplicates a given memory instruction such that each duplicate accesses a provably disjoint set of addresses.

The compiler can also help by going beyond program analysis to program transformation. In this case, the compiler transforms the input program in order to increase the number of groups. One common transformation addresses the issue that a given memory operation may access many classes of addresses. By selectively duplicating code, the memory operation may be replicated in such a way that the addresses accessed by one instruction are provably disjoint from the addresses accessed by the other. At this point, the operations can be assigned to different tiles[9]. Figure 2-17 shows an example of one such transformation. Section 5.4.1 details the program analyses and transformations

[9]In a way, we could think of this as *instruction meiosis* – the instruction is being split into two copies which each access a disjoint subset of the addresses.

that the Raw compiler performs in order to enhance memory parallelism. More details on these techniques, including *modulo unrolling* and *equivalence class unification*, can be found in [8, 9].

2.7 I/O Operation

We now turn our attention to the final architectural component of the ATM, the I/O system. The I/O ports connect the on-chip networks to the pins of the chip, which in turn are connected to I/O devices. These I/O devices – whether hard drives, video inputs, D/A or A/D converters, or memories – are essentially network devices that are logically situated on the ATM's on-chip networks. Because I/O devices are network clients, they are free to initiate and pursue communication with other tiles and other devices, as depicted in Figure 2-18. This communication is typically done through a series of network packets, if a generalized transport network is employed, or in the form of a stream of words if the scalar operand network is used. Frequently, both types of communication can be employed by the same device types. For instance, Figure 2-19 depicts a tile doing a number of cache-misses (via request-reply messages) in parallel over the generalized transport network, while Figure 2-20 shows a tile which, having programmed three DRAMs with the appropriate meta-data (e.g., addresses, strides and lengths), is summing two streams coming in from two DRAMs and routing the output to a third DRAM.

2.8 Deadlock in the ATM I/O and Generalized Transport Network (C6)

In any distributed networked system, such as the ATM, a central concern is *deadlock*. Deadlock occurs when multiple messages become lodged in the network indefinitely due to a cyclic interdependence for buffer resources. In the context of the ATM, deadlock can emerge in many parts of the system including communication between tiles, among I/O devices, and between tiles and I/O devices. Deadlock can result from communication over the SON, communication over the generalized transport network, or a combination of both. In this section, we discuss the two key types of deadlock – in-network and endpoint deadlock – and describe some effective approaches for handling them in tiled microprocessors, including the use of trusted and untrusted cores which implement deadlock avoidance and virtual buffering, respectively.

Figure 2-21 shows one such example of deadlock. Two tiles are issuing a series of requests to two DRAMs. Each DRAM repeatedly dequeues a request from the ingoing network, and then enqueues a response into the outgoing network. Each DRAM will not dequeue the next request until it has replied to the last request. Deadlock occurs if the network buffers in the DRAM request paths become full and the DRAMs are both trying to reply to messages. In this case, both DRAMs are

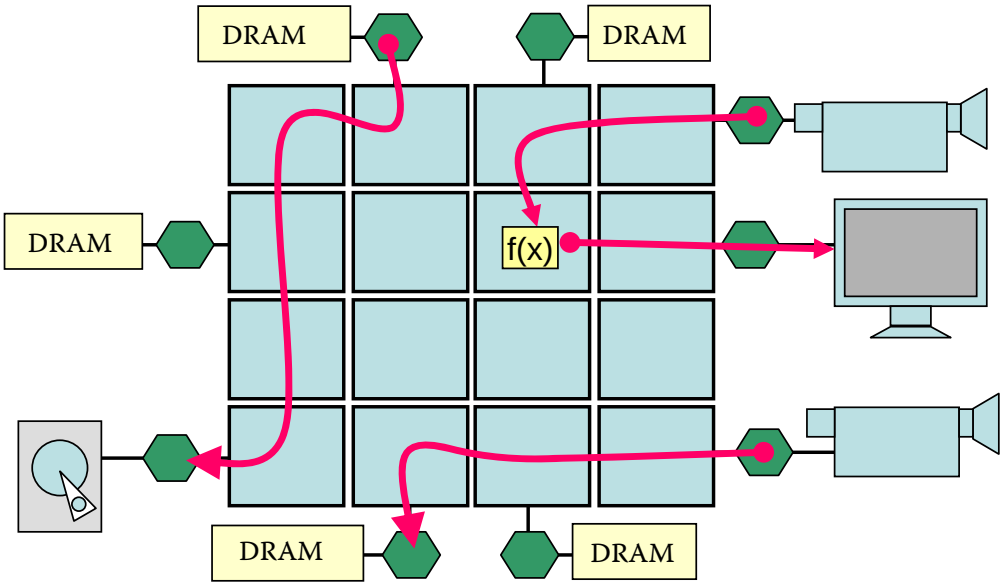


Figure 2-18: I/O communication in the ATM. I/O devices can use the ATM's network to communicate among each other and with the tiles. In the figure, this is demonstrated in three instances. First, data is routed directly from DRAM to a hard disk. Second, data is being routed directly from a video camera to a DRAM. (Although both of these cases might come under the term *direct memory access* ("DMA"), communication is also possible between non-DRAM devices, such as the video camera and the hard drive). Finally, the example shows an example in which a video stream is being processed by a tile and then routed directly to a frame buffer for display.

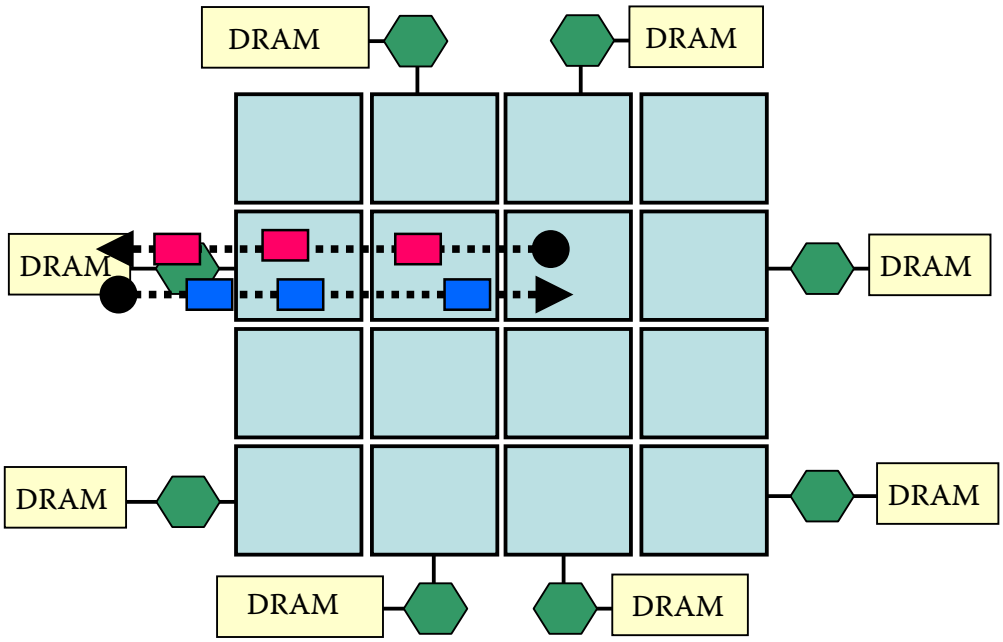


Figure 2-19: Packet-oriented communication between a tile and a DRAM. The tile is servicing multiple cache requests in parallel via the packet-oriented generalized transport network. Each cache miss causes a request packet to be sent to DRAM. The DRAM then sends a reply packet back to the requesting tile.

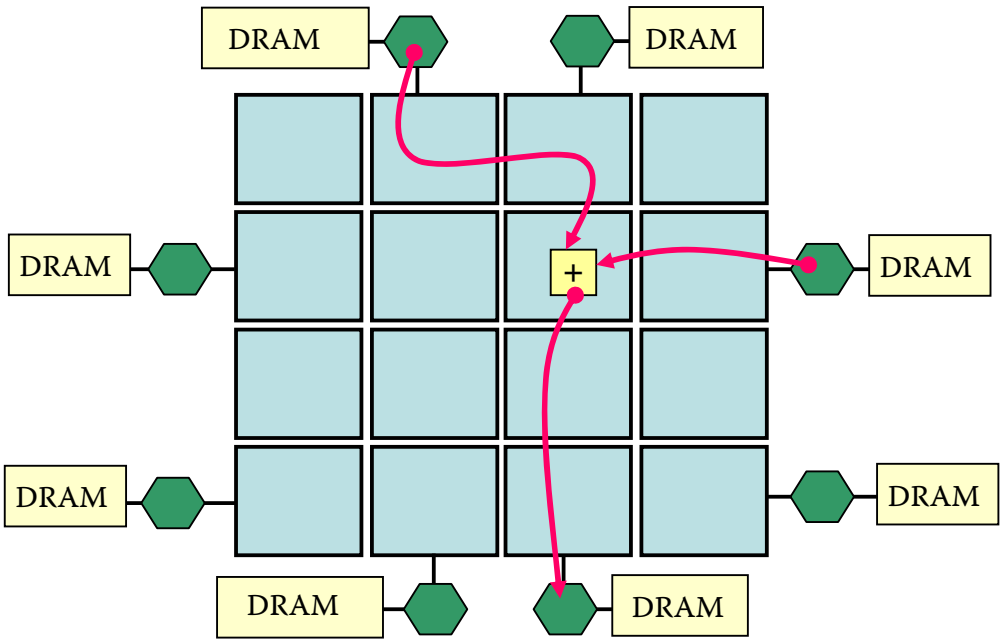


Figure 2-20: Stream-oriented communication between a tile and three DRAMs. Two of the DRAMs are streaming data into the on-chip SON, which is routing the streams to the tile. The tile is processing the data and then routing the output over the SON to a destination DRAM. Typically, this procedure requires that the DRAMs be informed of the addresses, strides and data lengths that apply to the incoming or outgoing data streams. This may occur over the SON or the generalized transport network.

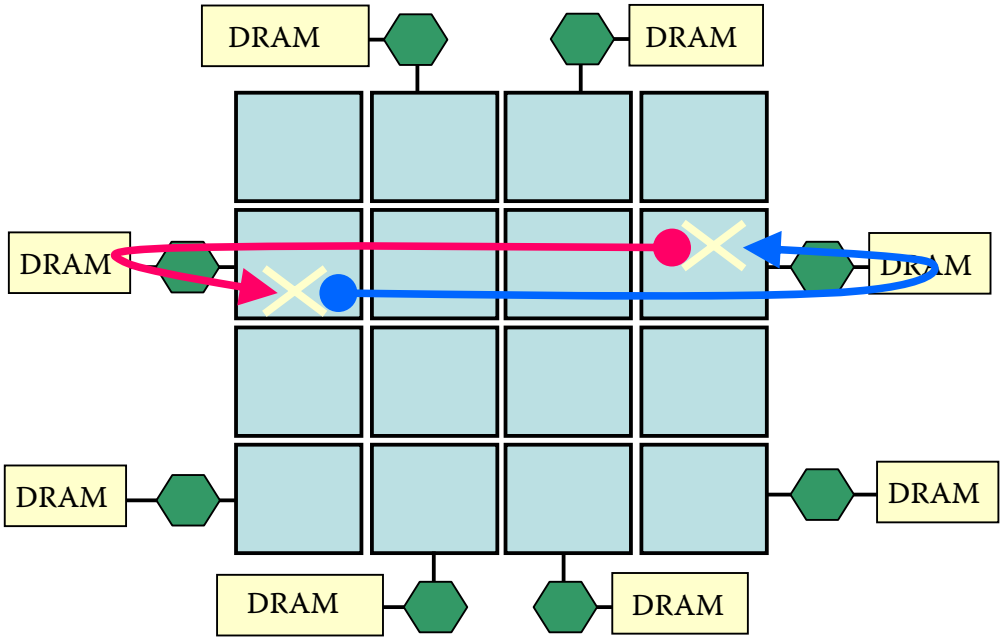


Figure 2-21: Example of deadlock in ATM.

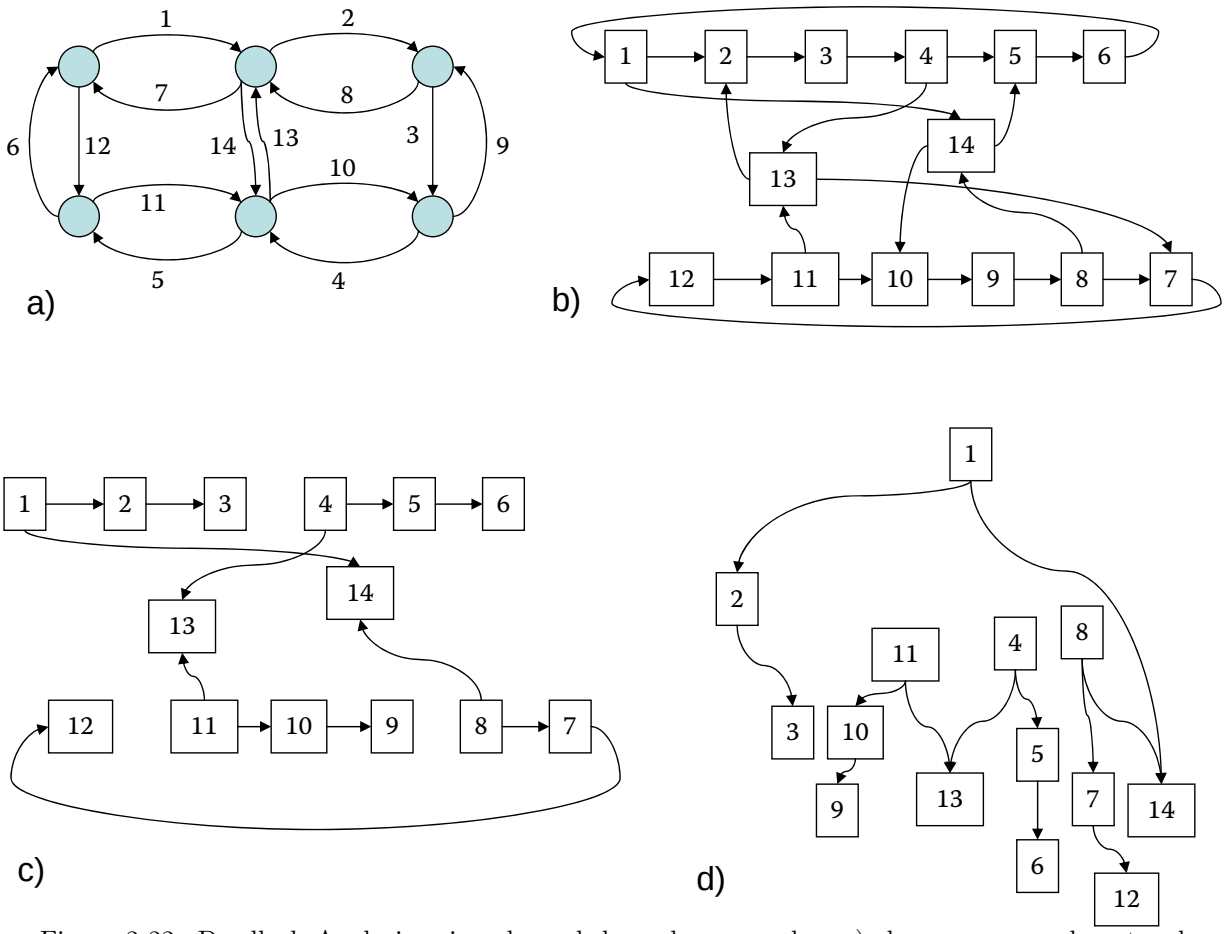


Figure 2-22: Deadlock Analysis using channel dependence graphs. a) shows an example network containing 6 nodes. b) shows the channel dependence graph, assuming that messages are never sent back on their incoming edges. There are a number of cycles. c) shows the same network, with dimension-ordered routing constraints. d) shows the same graph, but drawn in a way that shows it is clearly acyclic.

unable to proceed because they have insufficient space to enqueue their replies; they are mutually dependent on each other's forward progress to free space in the network. Although this example involves communication between two DRAMs and two tiles, it could just as easily be communication between two pairs of tiles, or two pairs of DRAMs.

More generally, the problem of deadlock in interconnection networks can be formalized through the use of a *channel dependence graph* [23]. In a channel dependence graph, a node represents a network buffer, and an edge from node a to node b indicates the fact that the "agent" (for instance, a network router) holding a may wait indefinitely until b is free. If there is no cycle in the channel dependence graph, then it is not possible for deadlock to occur.

For example, we examine a 6-node mesh network in which messages can be routed in any direction except back on the link on which they came. For this network, shown in Figure 2-22a, we can create a channel dependence graph (Figure 2-22b) which represents the dependences created by the routers

in the network. This graph has a number of cycles, indicating that deadlock may be possible. A common technique for eliminating possible deadlock is to restrict network type and the possible routes that can occur on the network. One such restriction, in the context of mesh networks without end-around connections is called *dimension-ordered routing* [22], in which all routes in one dimension (say, the X direction), are performed before all routes in the next dimension (say, the Y direction.) The 6-node system, with these additional restrictions, has no cycles in its channel dependence graph (Figure 2-22c), which has significantly fewer edges. (Figure 2-22d shows the same graph, laid out in a way that clearly demonstrates that it is acyclic.)

Although dimension-ordered routing is a handy technique for attacking the deadlock problem in meshes, it is often in itself not sufficient. The problem is that network routers are not the only parties responsible for creating edges in the channel dependence graph – the software and hardware that injects and receives messages can also be responsible for these edges. Thus, the channel dependence graph in Figure 2-22c is not quite complete – because it does not include the additional dependence arcs that the DRAM controllers introduce.

To examine this issue, we add two DRAMs to the 6-tile system, pictured in Figure 2-23a. The DRAMs (and I/O ports more generally) themselves present a minor challenge because if they are considered to be in either the X or Y plane, they become unreachable by some nodes if dimension-ordered routing is employed. Fortunately, an easy solution exists: out-arcs from an I/O port are considered to be in the "W" dimension, and in-arcs from an I/O port are considered to be in the "Z" direction. The dependences caused by this "WXYZ" dimension-ordered routing are pictured in Figure 2-23b. The new nodes, `d1`, `d2`, `d3`, and `d4`, are either source ("W" direction) or sink ("Z" direction) nodes, so we know that no cycles are created by these additional arcs. It is only when we incorporate the knowledge that the DRAM controllers wait to complete their current transaction (which may require enqueuing a message on the output channel) before moving on to dequeue the next request from the input channel that we observe the existence of a deadlock. Figure 2-23c shows the additional arcs (shown dotted) added by this dependence. Careful examination reveals that the graph now contains a cycle which, interestingly, occurs only once the second DRAM is added to the system.

The example shown in Figure 2-23 demonstrates an instance of *endpoint deadlock* as opposed to *in-network deadlock*. Endpoint deadlock occurs because one or more nodes has a message waiting that it is not dequeuing (often because the node is blocked trying to inject a message into the network). In-network deadlock, where deadlock occurs in intermediate nodes of the routing network as opposed to at the endpoints, is easily prevented in wormhole or virtual cut-through mesh networks (without end-around connections) through the use of dimensioned-ordered routing [22]. On the other hand, endpoint deadlock is often a challenging issue to address.

The endpoint deadlock problem is one with surprising and deep consequences on on-chip network

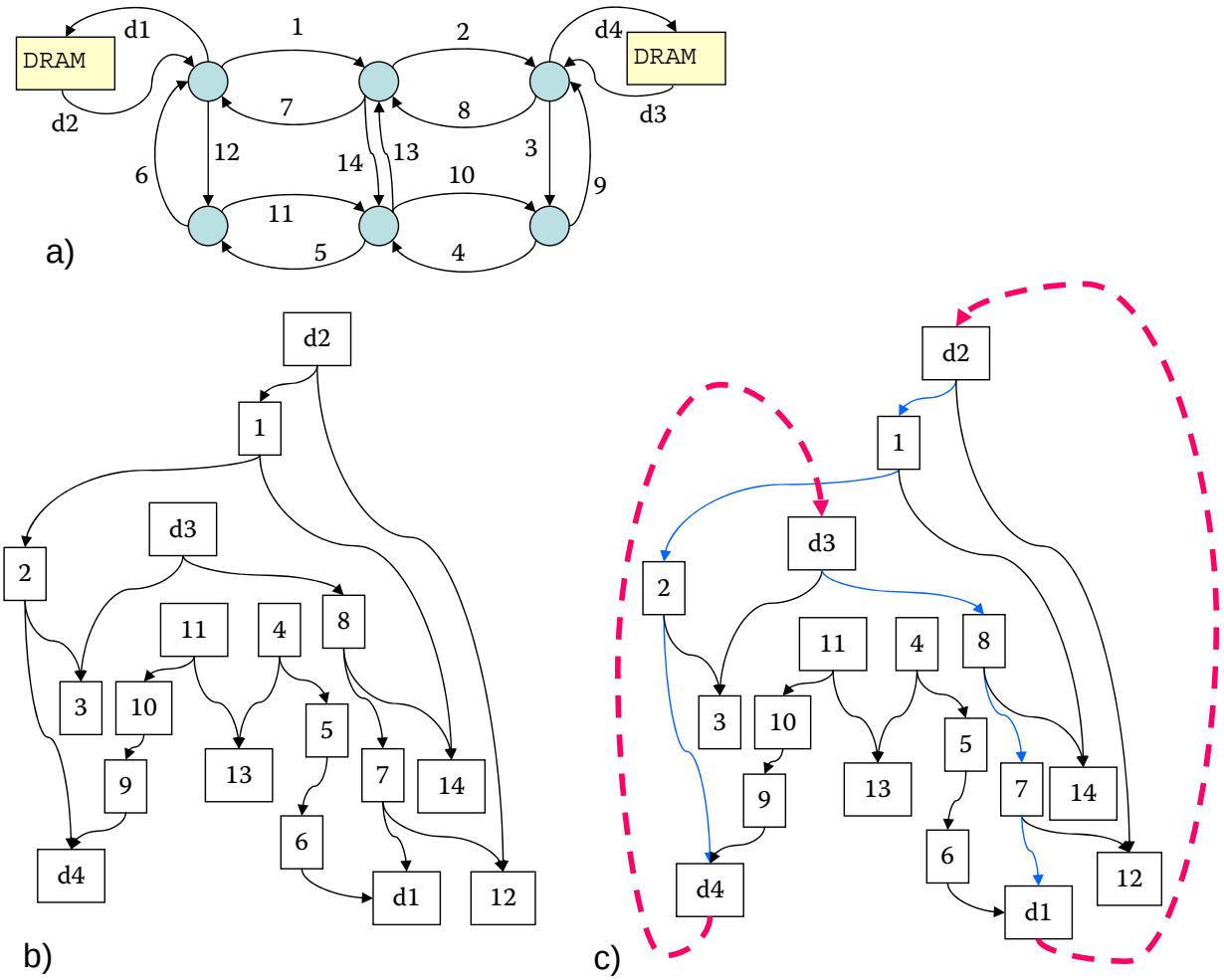


Figure 2-23: Applying deadlock analysis to a 6-tile ATM with two DRAMs. Part a) is the ATM system, ignoring that the compute pipeline portion of the tiles. Part b) shows the corresponding channel dependence graph. Part c) shows the graph given the knowledge that DRAM controllers may block on their outputs, preventing them from absorbing incoming messages.

design and use. Ultimately, the best deadlock-free network design can be thwarted by the software or hardware state machines that control each node, connecting channels, directly or indirectly, and creating cycles in the channel dependency graph. For closed, controlled networks, where the user can only inject messages indirectly through restricted system or hardware-level interfaces, deadlock can be prevented by analyzing the fixed system and providing the resources (e.g., buffers) to make it deadlock-proof. For networks that are user programmable and for which users have unfettered access, the system designer's goals must be more modest. Ultimately, nothing can save the users from themselves. Much like with a user's program that has a killer memory leak, we can try to forestall deadlock (just as virtual memory does for out-of-memory errors) but ultimately the best *guarantee* to be hoped for is the capacity to kill the program and return the machine to a usable state.

The approach to deadlock in the ATM mirrors a recent trend in many areas of computer science – first, we establish a *trusted core* that employs a closed, controlled network in conjunction with a set of protocols that has been proven deadlock-free given that network. This closed network provides reliable, deadlock-free access to external DRAMs and I/O devices. Second, the ATM provides a *untrusted* user-level, open network which supports arbitrary communication patterns. Although ultimately it is still the end programmer's responsibility to avoid deadlock in order to ensure that their application runs correctly, we assist them in two ways. First, the open network is *virtualized* using the DRAMs that are accessible by the trusted network. Thus, if the user overflows the buffer-space inside the network, the DRAM is used to transparently virtualize this space in order to facilitate forward progress in the majority of cases, with a modest performance penalty. Furthermore, should the program exhaust the virtualization resources, the facility exists to discard the user's packets and terminate the program.

2.8.1 The Trusted Core

The existence of a trusted core begets the question "Who is it that we trust to touch the trusted core?" Certainly there are many possibilities – we may, for instance, want to allow device driver writers or embedded system designers to be able to extend the trusted core. Ultimately, for the purposes of this thesis, we simply trust whoever it is that is willing and able to prove that their modifications to the core are deadlock-free. This proof is burdensome, and requires a relatively complete understanding of the system, because it often must incorporate the interactions of multiple network clients that make up the trusted core. In practice, it is the complexity of these proofs (and the necessity or benefit of putting them in the trusted core) that drives a programmer to decide whether to implement functionality inside or outside the trusted core. Fortunately, architects have developed a number of techniques which simplify the task of reasoning about potential deadlocks.

Both the hardware designer and protocol designer have significant impact on the trusted core. The hardware designer must determine the resources levels that will be provided to the protocol designers. At the same time, the protocol designers must determine what protocols are implementable given the hardware constraints. To a certain extent, this arrangement requires that the hardware designer have a reasonable idea ahead-of-time of the nature of the protocols that will be implemented. However, the design remains moderately fluid, to the extent that the protocols in the trusted core can be extended within the constraints of the existing resources and still guarantee the absence of deadlock.

In the remainder of this section, we discuss a number of disciplines that are available for the trusted core designers – both hardware and software – and their implications on ATM scalability. These approaches include a) guaranteeing message sinkability at message destinations and b) employing logical channels.

2.8.1.1 Discipline I: Guarantee Message Sinkability

Perhaps one of the most intuitive approaches to avoiding endpoint deadlock is to ensure that all message destinations are capable of pulling any incoming messages off the network without relying on the state of the outgoing link. The net effect is to remove the edges in the channel dependence graph that connect the node's input channels to the node's output channels. In Figure 2-23c, these edges are shown as dashed lines. This, in concert with a dimensioned-ordered routing or another in-network deadlock avoidance technique, removes the presence of cycles in the dependence graph, eliminating the possibility of deadlock.

Practically speaking, this means that the receiving node needs to be able to consume incoming values without producing new values on output channels, or if the incoming values do produce outgoing values, the node needs to have enough buffering to buffer the incoming values (and remove them from the channel) until the outgoing channel is freed up. To apply this discipline, the designer first quantifies the worst-case behaviors of the protocols running on the trusted network and then ensures that the node's local buffering is sufficient. Then, the system is designed to ensure that incoming messages never arrive at times when the node is unable to receive them[10].

In addition to prevent deadlock, this *message sinkability* discipline incurs the benefit that it naturally clears the networks, which eliminates network congestion that is caused by clients that run more slowly than the network. For instance, in the absence of receive buffering, if the hard disk in Figure 2-18 had a backlog of messages to write to disk, messages would back up into the network, clogging network channels. As a result, unrelated messages in the network could be blocked until the hard disk finished its transaction, which could be millions of cycles. However, if the I/O port (or hard-drive controller) connected to the hard drive has resources to sink the message, the system will be able to operate in parallel.

Analyzing the effects of employing the message sinkability discipline is a worthwhile task. Perhaps the most important analysis is the size of the node buffering that will be required. Obviously, every node must have some limit on the number of messages it can send; otherwise, any finite buffer at the receiving node could be exceeded. In practice, limiting the number of messages that a sender can issue is relatively easy; and in fact, it need not be node based; it could well be based on some other system resource. In practice, we often would like the amount of buffer per node to be proportional to the bandwidth-delay product of the round-trip path of a request-reply pair – this allows us to hide the communication latency of the system. In a mesh network of n nodes with Manhattan communication latencies and devices that rate-match the interface, this would be $\theta(\sqrt{n})$. The bandwidth-delay product for high-bandwidth, high-latency devices, such as a RAID

[10]For instance, if a tile's compute processor has executed a blocking message send, then it may be unable to execute a message receive to pull in the packet, because the send is blocked and may be indirectly dependent on the corresponding receive. On the other hand, it is acceptable if the computer processor cannot receive the value immediately, but is provably able to do so after the passage of some time.

system with high bandwidth and high latencies (e.g., 10 GB/s and 10 ms) could be even higher, however it is reasonable to expect that devices in general have enough buffering to handle their own internal bandwidth-delay products due to internal mechanisms.

More difficult is the possibility of a "flash mob", in which a large number of the nodes decide to send messages to a single node. In this case, a single node may need buffer space that is proportional to the number of nodes in the system, i.e. $\Omega(n)$. Moreover, if we want to be able to allow every tile to issue $\Omega(\sqrt{n})$ messages (for latency hiding) to a single target, we arrive at a total of $\Omega(n\sqrt{n})$ buffer space per tile. In practice, we may be able to decouple these two factors, by allowing the software (or the receiving node) to configure how many outstanding messages any sender-receiver pair can sustain, and to reserve the corresponding buffers from a pool of available buffers. But, in practice, requiring even $\Omega(n)$ buffer space per node is a troubling scalability issue – it means the system grows asymptotically as $\Omega(n^2)$.

In the context of the ATM, one solution is to disallow general messaging among tiles using the trusted network – instead inter-tile messaging is performed using an "untrusted network". Then the trusted network is used only for communication among I/O ports, as well as among I/O ports and tiles. This way, assuming that I/O ports scale with the perimeter of the chip, the n tiles will each require only $\theta(\sqrt{n})$ messages ($O(1)$ per port), while I/O ports will require $\theta(n)$ space ($O(1)$ per tile and port). Again assuming that I/O ports scale with the perimeter of the chip, the total buffering area will be $n * \theta(\sqrt{n}) + \sqrt{n} * \theta(n) = \theta(n * \sqrt{n})$, which may be more reasonable given the constant factors.

2.8.1.2 Discipline II: Employing Logical Channels

The emergence of the possibility of a non-scalable aspect of the ATM is somewhat disturbing at this late state in the analysis. We are motivated to find the ever elusive $\theta(n)$. The buffer requirements of providing arbitrary message sinkability are much greater than those needed in the common case. Further, even in the worst case scenario, the destination node is presumably so backlogged with data to process that the extra buffers are not of much use. A key observation is that while the message sinkability discipline eliminates the harmful side-effects of flow control (i.e., deadlock), it also eliminates the beneficial ones (e.g., telling sender nodes to "hold up").

If instead, we could find a way to reintroduce flow-control on all of the arcs from a transaction's initiator (e.g., the node for which no incoming messages required the outgoing message to be produced) to the final consumer (e.g., the node for which no outgoing messages are required to finalize the message), then we would be able to use flow-control to applying back-pressure on the network and avoid the need to buffer all incoming traffic. Then, the only party that is required to sink incoming messages is the final node, where it is easily done because there are not outgoing dependencies.

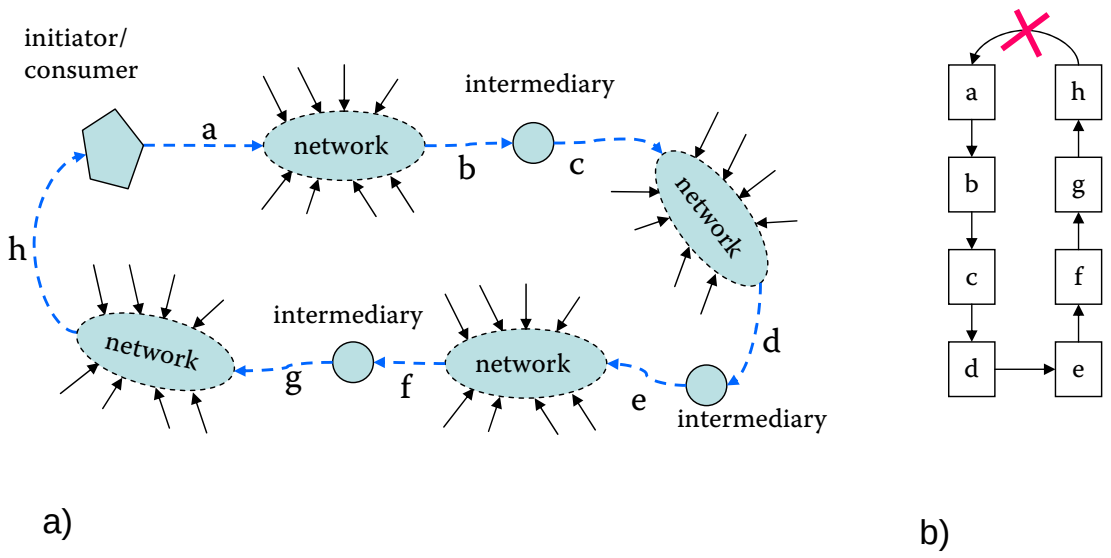


Figure 2-24: Request-reply communication flow and dependence graph. Each circle represents a node which is processing the input message and outputting a separate output message. The hexagon is both the initiator of the initial request and the consumer of the final reply. A different logical network is used between each pair of participants. The "X" denotes the edge of the dependence that has been removed by the guarantee that the final node be able to consume incoming messages.

Figure 2-24 shows this scenario. A request is sent out by an initiator (which also happens to be the consumer of the final reply message), processed by three intermediaries in succession, and then returned back to the consumer. The corresponding channel dependence graph is shown – it is complete, except for the arc that has been eliminated (marked by the "X") because of the requirement that the final node sink the message. As we can see, the dependences are held intact from the start to the end of the message. If any of the intermediate nodes are short on input buffer space, they are free to stall the nodes that feed into them without consequence. Because the final link does not finish the cycle, deadlock is not possible in the system. However, there is one detail which is overlooked – the network segments in-between intermediaries are represented as a single-acyclic edge. How can we guarantee that these network segments do not introduce cycles, especially since there may be unrelated messages flowing through the networks? That's the key: each network is a separate *logical network*, which employs a in-network deadlock avoidance algorithm. Furthermore, the networks are ranked according to their priorities. The intermediate nodes are only allowed to block lower-priority networks if waiting on higher-priority networks. Because blocking higher-priority networks while waiting on lower-priority networks is disallowed, and because each logical network is in-network deadlock free, we can prove that there are no cycles in the channel dependence graph and thus that the end-to-end protocol is deadlock free. This approach can extended to an arbitrary DAG-structured-protocol. Figure 2-25 shows the use of logical channels to eliminate the deadlock scenario of Figure 2-21.

What then, are these logical networks? The essential element is that they be able to act, in terms

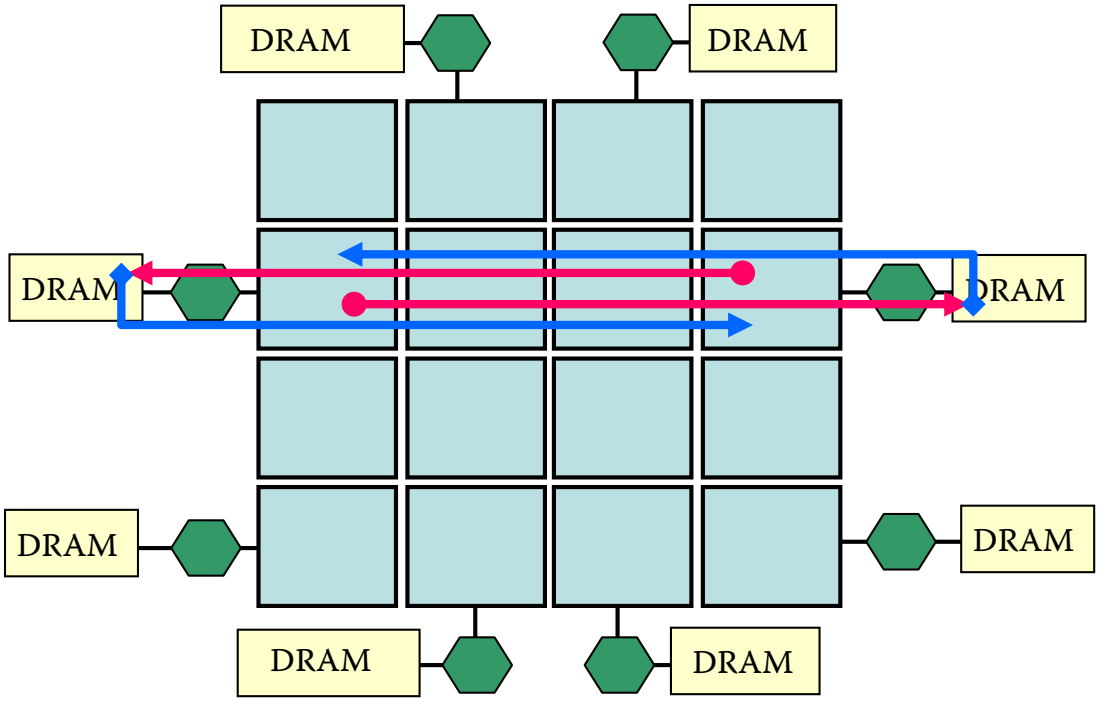


Figure 2-25: Deadlock elimination of example in Figure 2-21 using two logical channels – one for requests and one for replies.

of network blockages and flow-control, as if they were completely separate networks. In practice, we may implement them as several physical networks, or we may multiplex the data wires but employ a separate set of FIFOs. The relative constraints of the implementation technology will determine what is practical.

With this sketch in mind, we can evaluate the scalability impact of this organization. The number of logical networks is determined by the complexity of the protocols that are run on the trusted core – essentially, it is proportional to the maximum number of intermediaries that must be passed through on the longest trek related to a single request. Since this number depends on the protocols that we choose to use in the trusted core, rather than the number of tiles, we have effectively exchanged the unscalable buffer demands of the message sinkability discipline for a constant factor overhead. The constant factor comes from the fact that we limit our protocols on the trusted core so they require only a fixed number of logical networks. Of course, it will still be desirable to maintain some buffers at the receive side to reduce congestion in the network – but the sizes will be determined by performance constraints rather than the desire to avoid deadlock.

2.8.1.3 The ATM's trusted core

For the purposes of the ATM, we will assume the logical channels discipline for the trusted core transport network. In practice, we may well do with a mix of the two approaches. For smaller networks,

the message sinkability approach may be the most efficient, since it minimizes tile size. However, as the system is scaled up, logical networks (or another approach[11]) provide more scalability.

2.8.2 The Untrusted Core

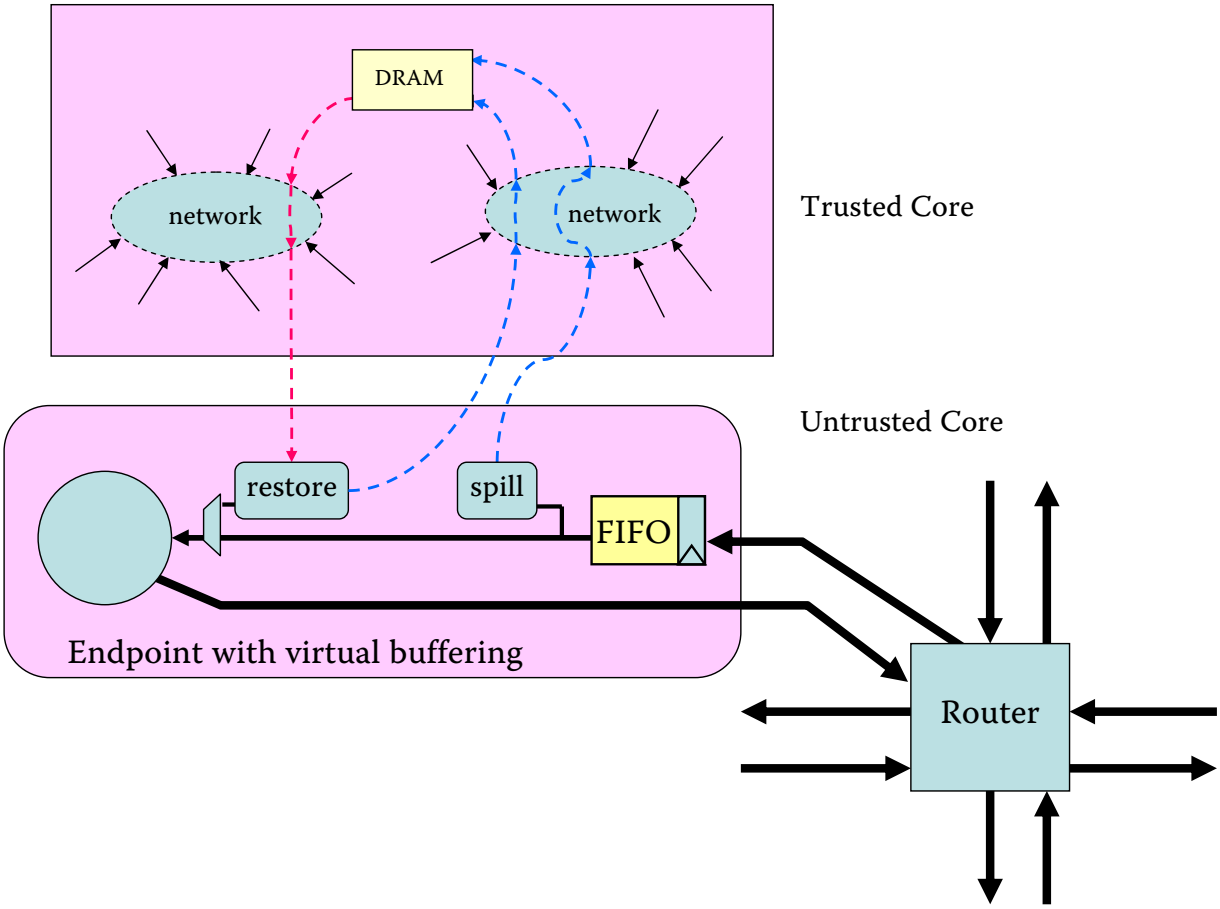


Figure 2-26: Using virtual buffering in the untrusted core to virtualize buffers and reduce potential deadlock. Only two links of the untrusted network are shown, but the topology could be arbitrary.

The trusted core discipline has two significant limitations. First, it requires that the users prove that their protocols are deadlock-free. This makes the trusted core difficult and error prone to modify. Second, the discipline limits the diversity of protocols that can be used on the trusted network. In the case of the logical channel discipline, the protocols are limited by the number of available logical channels. In the case of the message sinkability discipline, protocols are limited by the amount of buffer space available at the nodes.

The role of the untrusted core is to provide a communication mechanism for users that does not

[11] An alternative discipline is to do what is frequently done in larger-scale networks: drop packets. Dropping packets is another way of removing dependence edges in the network without restricting the routing function. Although this technique may indeed be applicable to on-chip networks, we leave investigation of these issues (and subsequent proofs about the absence of livelock and forward progress!) to future work.

require them to write proofs about deadlock, and further does not limit the complexity of protocols that are run. The key idea is that the trusted core can be relied upon to provide a path to copious memory (e.g., external DRAM) in order to virtualize the buffering resources in the untrusted core[12]. Effectively, this *virtual buffering* [76] implements a caching hierarchy for message buffers at each message end-point, providing the abstraction of nearly-unbounded message sinkability for nodes. Figure 2-26 shows an instance of virtual buffering on an untrusted core that employs the trusted core to virtualize buffer space. Each endpoint has a "spill unit" that can dequeue elements from the incoming FIFO and send them to the DRAM using the trusted core. The "restore unit" can then retrieve elements from DRAM on a demand basis. In the common case, the spill and restore units are not employed; however, if the data has been waiting on the input for a long time, then the spill unit will start buffering data in DRAM. The restore unit then intercepts requests to the incoming FIFO and substitutes the buffered data.

2.8.3 Deadlock Summary

The most challenging type of deadlock in tiled microprocessors is endpoint deadlock. In the preceding section, we proposed the technique of subdividing generalized transport network functionality into two parts – the trusted core and the untrusted core. The trusted core is designed to avoid deadlock using either the message sinkability or logical channel based disciplines. To do so requires that the system designers limit the set of protocols that run on the network, and that, for any incremental functionality, that they extend the existing proof of deadlock-freedom to encompass the new functionality and its interactions with the existing functionality. Since the burden of such proofs is too high to place on ordinary users, the ATM also provides the untrusted core, which does not require user proofs. The untrusted core uses the trusted core to implement virtual buffering (the trusted core is used as transport to pools of buffering in a deep memory such as DRAM), which delays the onset of deadlock much in the way that virtual memory delays the onset of out-of-memory and out-of-address space errors.

2.9 Exceptional Events - Especially Interrupts (C7)

Although we have already examined many types of exceptional events that microprocessors need to deal with – such as branches, cache misses, memory requests, memory dependencies, and I/O – the issue of *interrupt handling* remains. In a tiled microprocessor, we want interrupt handling to inherit the same scalability properties as the rest of this system. As a result, it is desirable to have interrupts (issued by periphery devices, or by tiles) implemented in a distributed and scalable

[12] Although there are parallels to the concept of *escape channels* [28], where an acyclic subset of network links guarantees a deadlock-free path from sender to receiver, this approach is subtly different. The deadlock-free subset in this case is used to redirect messages through a virtual buffer rather than to their final destination.

fashion. Generally speaking, the deadlock issues involved in processing interrupts can be solved by applying the same concepts that were used to define the trusted and untrusted cores. However, because interrupts are infrequent events, our willingness to expend on-chip resources to overcome deadlock is limited.

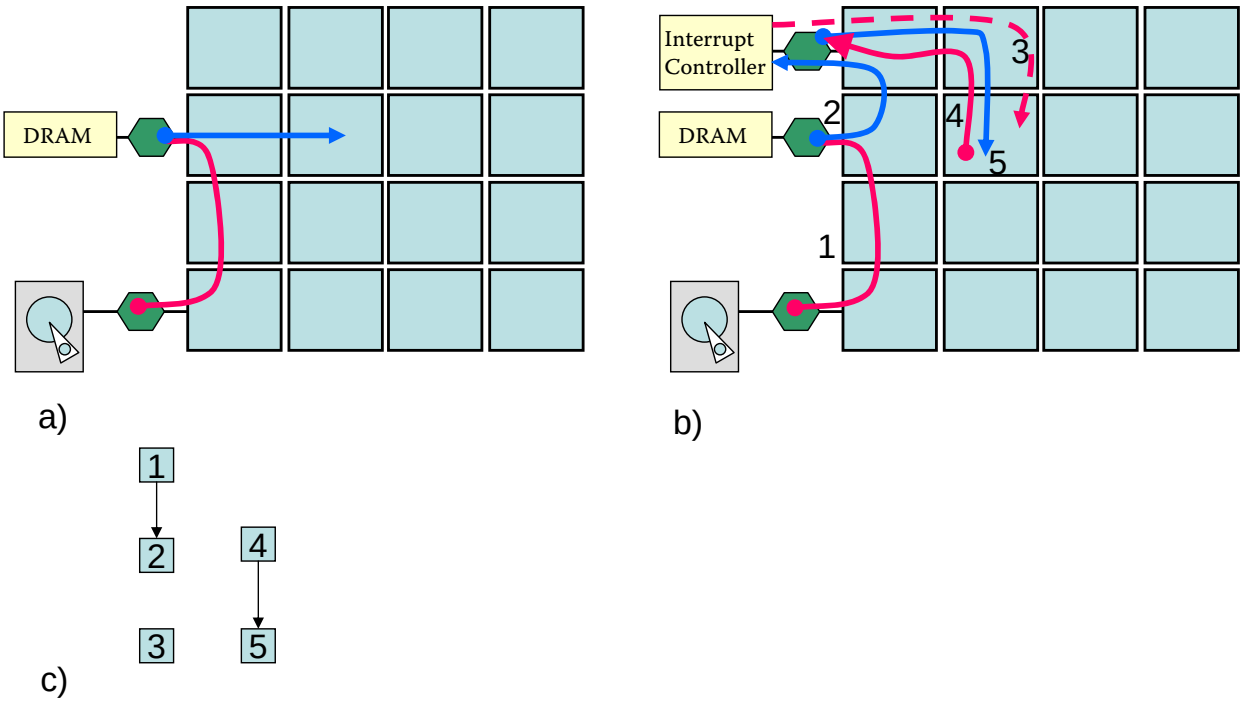


Figure 2-27: Part a) shows an interrupt being sent from an I/O device to a tile via "port bouncing". This configuration creates deadlock and scalability issues for the tile. Part b) show an alternative implementation, which makes use of an interrupt controller. The I/O device sends its interrupt to the designated interrupt controller via the DRAM (1). The interrupt controller records the per-tile interrupt information, sinking all messages (2). For each tile assigned to the interrupt controller, the interrupt controller maintains a bit indicating whether it has sent a notification message to a tile indicating the presence of available interrupts, and whether the tile has responded. If there are outstanding interrupts for a tile, and no notification message is outstanding, the interrupt controller will send a message (3) to the tile indicating the presence of waiting interrupts. Note however, that the interrupt controller can always sink incoming requests, which eliminates the corresponding arc from the channel dependence graph. The tile contains a single bit or word indicating the existence of a pending interrupt. A message from the interrupt controller will automatically be sunk and set this bit, clearing the channel. When the tile is free to service the interrupt, it will send a series of requests (4) to the interrupt controller, which will provide more information (5) about each interrupt. The coloring of the arcs indicates that this can be done with two logical networks (e.g., a request-reply network). Part c) shows the channel dependence graph, which is free of cycles. In many cases, the interrupt controller will be implemented in DRAM, since that is a cheap source of storage.

How does an interrupt occur in a tiled microprocessor? Typically, an interrupt will occur when some device has completed a task; for instance, when a hard drive has finished transferring data to DRAM over the on-chip interconnect. At this point, the hard drive needs to notify a tile that the data has finished transferring. On first impression, we may imagine that the hard drive would send a message directly to the tile. But in order to avoid race conditions, the hard drive must

ensure that its data has indeed reached the DRAM before informing the tile that that is the case. This is because it's feasible that the tile may receive the drive's notification message and access the DRAM before the final writes have been posted to DRAM. In many cases, we can rely upon the ordering guarantees of the network and DRAM system (e.g., messages sent from the same source to the same destination on the same logical network in an obliviously routed network are guaranteed to arrive in order) and simply "bounce" the notification message off of the DRAM port in question (Figure 2-27a). In more sophisticated systems, it may be necessary to perform a more comprehensive completion check.

The major challenge arises when considering the arrival of interrupt message at the destination tile. Once again, the *flash mob* issue arises – what if a flood of devices all decide to contact the same tile? We need to be able to avoid deadlock in this case. Logically, interrupts are higher priority than almost all messages except memory messages. They are lower priority than memory messages because the servicing of interrupt messages requires the use of memory messages. If interrupt and memory messages are intermixed on the same logical network, then we will be forced to buffer incoming interrupt messages because the memory messages stuck behind them are necessary (perhaps because of a inter-dependency with other tiles that are trying to complete their own interrupts and are performing cache-coherency requests to the current tile) for completing the current interrupt message and then continuing on to the next interrupt message. Conversely, interrupt messages are higher priority than user-level messages because we do not want interrupt processing to be delayed if the user is slow to process incoming messages and the interrupt message is stuck behind a number of incoming messages.

Of course, our first impulse is to employ the standard arsenal of deadlock management techniques. Indeed, both of our canonical solutions can be made to work. We could provide tiled-local memory buffering proportional to the number of requesters, or we could provide a separate logical network for delivering interrupts. However, both of these solutions are quite costly (and the first unscalable) considering the relatively low frequency of interrupts.

An alternative (shown in Figure 2-27b) is to make use of off-chip interrupt controllers. Each tile is assigned to one of the interrupt controllers. The job of the off-chip interrupt controller is to buffer interrupt information on behalf of tiles. It in turn notifies tiles when they have pending interrupts (setting as little as a single bit on the tile). The tile can then message, as its leisure, the interrupt controller to find out about and process the interrupts. This protocol can be integrated into the trusted core (essentially behaving as a series of memory requests) with little impact on logical network count or buffering requirements. In many cases, the interrupt controllers will be integrated with the DRAM controllers, providing convenient access to a cheap memory. Although the introduction of interrupt controllers can help reduce the effective cost of interrupt support, ultimately the system must ensure that the number of pending interrupts is bounded in some way.

Typically, this is done by the operating system, application (in an embedded system), and/or device drivers, by limiting the number of outstanding I/O operations that can be initiated.

2.9.0.1 Interrupt Controller Scalability Analysis

The scalability analysis of the off-chip interrupt controller is somewhat ambiguous, since it is difficult to estimate how the number of outstanding I/O requests may change as the system is scaled up. In general, we can expect there to be a number of interrupt controllers proportional to the number of I/O ports[13], $\theta(\sqrt{A})$. Each interrupt controller can receive messages from $\theta(\sqrt{A})$ I/O ports. If the interrupt controller uses a linked-list based representation for interrupt lists, we have a net asymptotic growth of $\theta(\sqrt{A} + \sqrt{A}) = \theta(\sqrt{A})$. If the interrupt controller uses a fixed array representation for interrupt lists, we have a net asymptotic growth of $\theta(\sqrt{A} * \sqrt{A}) = \theta(A)$. Since the density of each DRAM increases as $\theta(A)$ with successive generations of Moore's Law, the system is relatively scalable with respect to technology scaling. However, if a system is scaled up through an increase in real die area (e.g., the use of many silicon die) rather than increased transistor density, the burden on an individual interrupt controller becomes more appreciable, and the linked-listed representation may become increasingly more applicable. Practically speaking, however, the greater issue in such a system is the decreasing amount of DRAM available per tile rather than the relatively small amount that is needed for interrupts.

2.10 ATM Summary

This chapter used the ATM , an archetypal tiled microprocessor, to discuss the essential architectural components of a tiled microprocessor. To do so, it examined seven criteria for physical scalability. Criterion 1, *frequency scalability*, allows us to build larger and larger systems without suffering a decrease in operating frequency. Criterion 2, *bandwidth scalability*, keeps the system efficient as it scales up. Criterion 3, *usage-proportional resource latencies*, ensures that the cost of accessing resources (in cycles) does not increase unnecessarily as the system grows. Criterion 4, *exploitation of locality*, circumvents the otherwise unavoidable reduction in system efficiency caused by global communication. Criterion 5, *efficient operation-operand matching*, enables the execution of programs over distributed ALUs. Criterion 6, *deadlock management*, is attained through either the message sinkability or logical channels discipline, and is most frequently applicable to the I/O and memory systems. Finally, Criterion 7, *handling exceptional events*, is a necessary requirement for usable microprocessors.

The chapter also introduced Scalar Operand Networks, a class of low-latency, low-occupancy network responsible for the transport of operands between remote functional units. We developed a

[13]This analysis assumes, as in previous sections, that I/O ports scale with the perimeter, $\theta(\sqrt{A})$, of the die area, A.

metric for SONs, the 5-tuple, as well as a classification system, the AsTrO taxonomy, and applied them to existing SONs.

Finally, the chapter discussed the challenges of mapping programs to tiled microprocessors, which includes placement, routing, and scheduling in the presence of control flow as well as methods for enhancing and exploiting memory parallelism in programs.

With the more abstract foundations of tiled microprocessors in hand, we are now prepared for the next chapter, which examines the architecture of the Raw tiled microprocessor prototype.

Chapter 3

Architecture of the Raw Tiled Microprocessor

The Archetypal Tiled Microprocessor described in the previous chapter represents an abstract ideal. To evaluate the ideas inherent in the ATM, we must concretize this ideal into an architecture, render the architecture into silicon, and scrutinize the resulting artifact. To that end, we created the Raw Architecture, which is the culmination of the design proposed in [121, 106, 107]. This chapter overviews the Raw architecture, the next chapter examines the Raw implementation, and the following chapter, Chapter 5, evaluates its performance. Chapter A and Chapter B can be referred to for concrete details on the architecture and instruction set. Generally speaking, this chapter explains the *why* and Appendices A and B explain the *what*.

Every implementation of the Raw architecture will have a number of parameters that are implementation specific, such as the number of tiles and I/O ports. We include the parameters for the Raw microprocessor in this description so as to provide a complete picture; since Raw is a tiled microprocessor, scaling the system is relatively straight-forward.

3.1 Architectural Overview

Much like the ATM, the Raw architecture divides the usable silicon area into a mesh array of identical *tiles*. Each of these tiles is connected to its neighbors via four point-to-point, pipelined on-chip mesh inter-tile *networks*. Two of these networks form part of the scalar operand network of the processor. The other two networks correspond to the generalized transport networks of the ATM. At the periphery of the mesh, i.e, the edge of the VLSI chip, *I/O ports* connect the inter-tile network links to the pins. Figure 3-1 shows the Raw microprocessor, which is a 16 tile, 16 I/O port implementation of the Raw architecture. The Raw architecture is designed to scale to 1024 tiles.

As a tiled microprocessor, the Raw architecture exhibits the same scalability properties as the ATM. It is by construction physically scalable, and meets the seven criteria for physically scalable

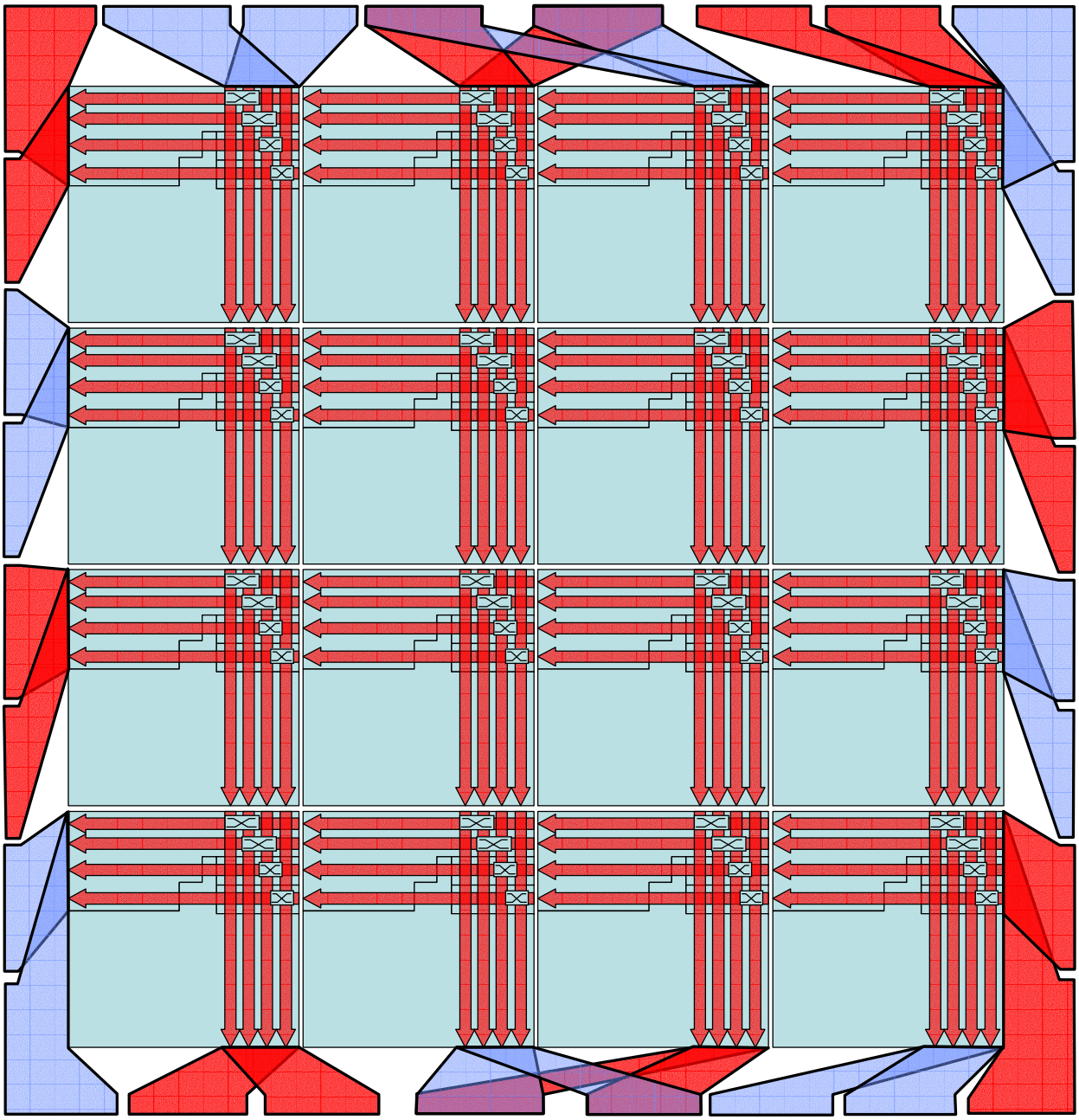


Figure 3-1: The Raw Microprocessor. The diagram shows the layout of the 16-tile, 16 I/O port Raw microprocessor on a VLSI die. The tiles, the I/O ports, and the networks are all visible. The sixteen replicated squares are the tiles. The arrows indicate the location of the network wires for the four physical networks. Each arrow corresponds to 34 wires running in each direction. The pin connections are located around the perimeter of the die. The shaded regions emanating from the tiles correspond to I/O ports; they show the connection from each edge tile's network links to the pins of an I/O port. Each I/O port is full duplex and has two sets of pins; 37 pins for incoming data, and 37 pins for outgoing data. The ports are shaded in alternating colors for better visibility. Note that two pairs of I/O ports share pins.

microprocessors. It does not employ broadcasts for its basic mechanisms, and its resources are fully distributed in order to exploit locality. It supports efficient operation-operation matching through the use of an **SSS** scalar operand network – i.e., assignment, transport, and ordering decisions are all performed at compile time. It attains an aggressive 5-tuple of <0,0,1,2,0>. The Raw architecture employs two wormhole-routed generalized transport networks. The first network, called the *memory dynamic network* ("MDN"), is used to implement a *trusted core* using the *message sinkability* deadlock-avoidance discipline. The second network, called the *general dynamic network* ("GDN"), is part of the user-level *untrusted core*, which provides *virtual buffering* through DRAM accessed over the trusted core. Finally, the Raw architecture employs interrupt controllers in order to eliminate the need for a third logical interrupt network. In practice, the message sinkability discipline works up to 1024-tile systems, but with the significant caveat that inter-tile messaging over the trusted core is disallowed.

Raw tiles each contain local data and instruction caches, which employ a configurable memory hash function[1] to wrap the address space around the I/O ports, according to the number of I/O ports, and according to which I/O ports are connected to DRAM. Although we've demonstrated *virtual caching* and *shared memory* in the Raw system on the untrusted core through the use of software run-time system assistance, Raw would likely benefit from greater hardware support of these mechanisms, and from the use of a logical-channel based deadlock avoidance system to allow flow-controlled inter-tile messaging on the trusted core. To an extent, our experience designing and implementing Raw motivated the addition of these entities to the ATM. Because *shared memory* has only limited hardware support in Raw, the compiler has the responsibility of maintaining not only memory dependences in sequential programs, but also cache coherence. Although in practice this is not a huge burden[2], it puts a moderately high price on the mobility of memory objects between phases of the computation, and thus results in relatively constrained movement of memory objects between caches in the system.

3.2 The Raw Tile

Logically, the components of a Raw tile, shown in Figure 3-2, are more or less the same as those of the ATM's tile. A tile contains a fetch unit and instruction cache to sequence instructions, functional units and a data cache for processing data, an SON for routing operands, and a trusted and untrusted core for free-form messaging, including cache misses. The Raw design divides the SON into two portions: an intra-tile SON and an inter-tile SON. This division optimizes the back-to-back execution of instructions in mostly-serial computations. The intra-tile SON has zero cost

[1] See the description of the `OHDR` instruction to see the mapping. Examples of the hash function in use are given in Chapters 9 and 10 of the Raw Specification [110].

[2] Raw has a selection of caching instructions that allow it to efficiently manipulate large address ranges, which can be used for efficient coarse-grained software-managed coherence.

(i.e., <0,0,0,0,0>) for communicating results between functional units that are local to the tile, while the inter-tile SON incurs the full <0,0,1,2,0> 5-tuple cost. The intra-tile SON is responsible for routing operands among the local functional units, the local data cache, and the interfaces to the inter-tile SON and the generalized transport networks. The local instruction and data caches employ the trusted core in order to message lower levels of the memory hierarchy.

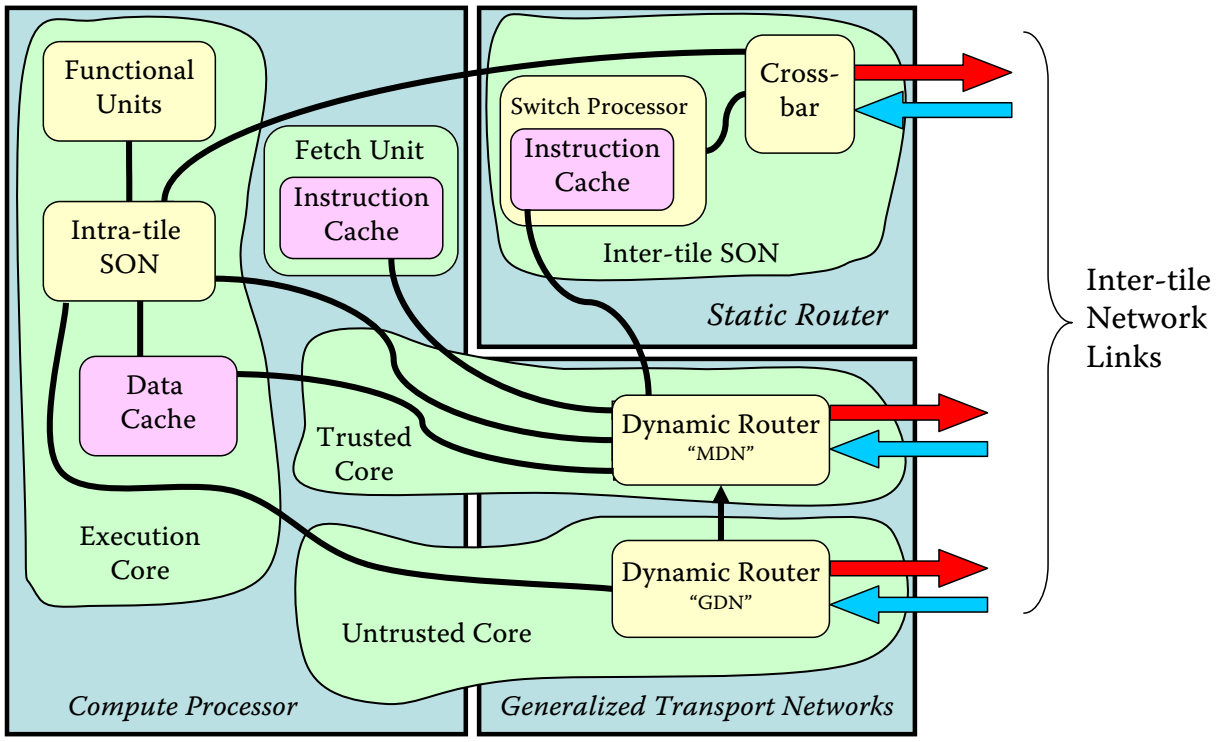


Figure 3-2: Logical anatomy of a tile in the Raw architecture.

Structurally, the Raw tile is divided into three components: the *compute processor*, the *static router*, and the *generalized transport networks*. The compute processor contains the functional units, the intra-tile SON, the data cache, and the fetch unit. The static router, which implements the inter-tile SON, uses a switch processor to control the inter-tile SON crossbar. This switch processor sequences through a dedicated instruction cache, which relies upon the trusted core for access to DRAM. The trusted and untrusted core are implemented jointly by the compute processor and the generalized transport networks, which provide the two physical wormhole-routed networks (called the "MDN" and the "GDN") for data transport. The untrusted core uses the trusted core (and its underlying network, the MDN) to implement virtual buffering.

3.2.1 The Raw Tile's Execution Core

The Raw tile execution core, shown in Figure 3-3, contains a number of pipelined functional units connected by the intra-tile SON. The intra-tile SON also connects the functional units to three

classes of inter-tile networks. First, it connects functional units to the inter-tile SON through the network input blocks ("NIB"s) labeled csti, csti_2, and csto. Second, the intra-tile SON connects the functional units to the trusted core generalized transport (through cmni and cmno). Third, the intra-tile SON connects the functional units to the untrusted core generalized transport (through cgni and cgno). The intra-tile SON contains a 32-element register file ("RF") with 2 read ports and 1 write port ("2R-1W") which is used to name, time-delay and reorder operands. The timing properties of the functional units and intra-tile SON are indicated with the flip-flop symbols in the figure; for instance, back-to-back dependent ALU instructions can occur on consecutive cycles, while back-to-back dependent FPU instructions can occur every four cycles.

The instruction set that controls the pipeline shown in Figure 3-3 is much like a conventional 32-bit MIPS-style instruction set, and the control logic is very similar to the canonical 5-stage MIPS pipeline (except with more pipeline stages). Figure 3-4 shows a more traditional pipeline representation for the tile's execution core and fetch unit. Register numbers are used to specify the communication of operands between functional units, and the typical RAW, WAW, and WAR hazards must be obeyed. Since the issue logic can dispatch one instruction per cycle to the functional units, the one write port on the register file is sufficient to manage the output operand traffic. However, because the functional units have different latencies, it is necessary to buffer their outputs so that writes to the register file can be performed in order. For instance, an ALU instruction that follows an FPU instruction but targets the same output register must wait for the FPU instruction to write its result before writing its own result. This buffering, shown in the left portion of Figure 3-3, in turn motivates the need for the bypass crossbar, which is used to forward the most recent value corresponding to a register name to the functional units.

3.2.1.1 Accessing the inter-tile networks

To specify that an instruction produces or consumes a value to or from a NIB rather than the register file, a number of register names have been reserved. When these register names are employed for an input operand, the value is taken from the corresponding input NIB instead of from the register-file. If the NIB is empty, then the instruction stalls in the IS stage. When one of the reserved register names is employed for an output operand, the value is sent to the corresponding output NIB. Much like writes to the register file, the microarchitecture must perform writes to a given NIB (in Raw's case, one per cycle) in the appropriate order. However, because one of the goals is to minimize latency through the inter-tile networks, writes to a NIB are allowed to occur as soon as all preceding writes to the NIB have completed. As a result, the pipeline employs a sort of "inverse-bypass" logic which, in contrast to conventional bypass logic, selects the *oldest* value destined for the NIB rather than the newest. If the oldest value is not ready, then all newer writes to the NIB must wait, ensuring in-order writes to the NIB. One consequence of this inverse-bypass scheme is that the commit point

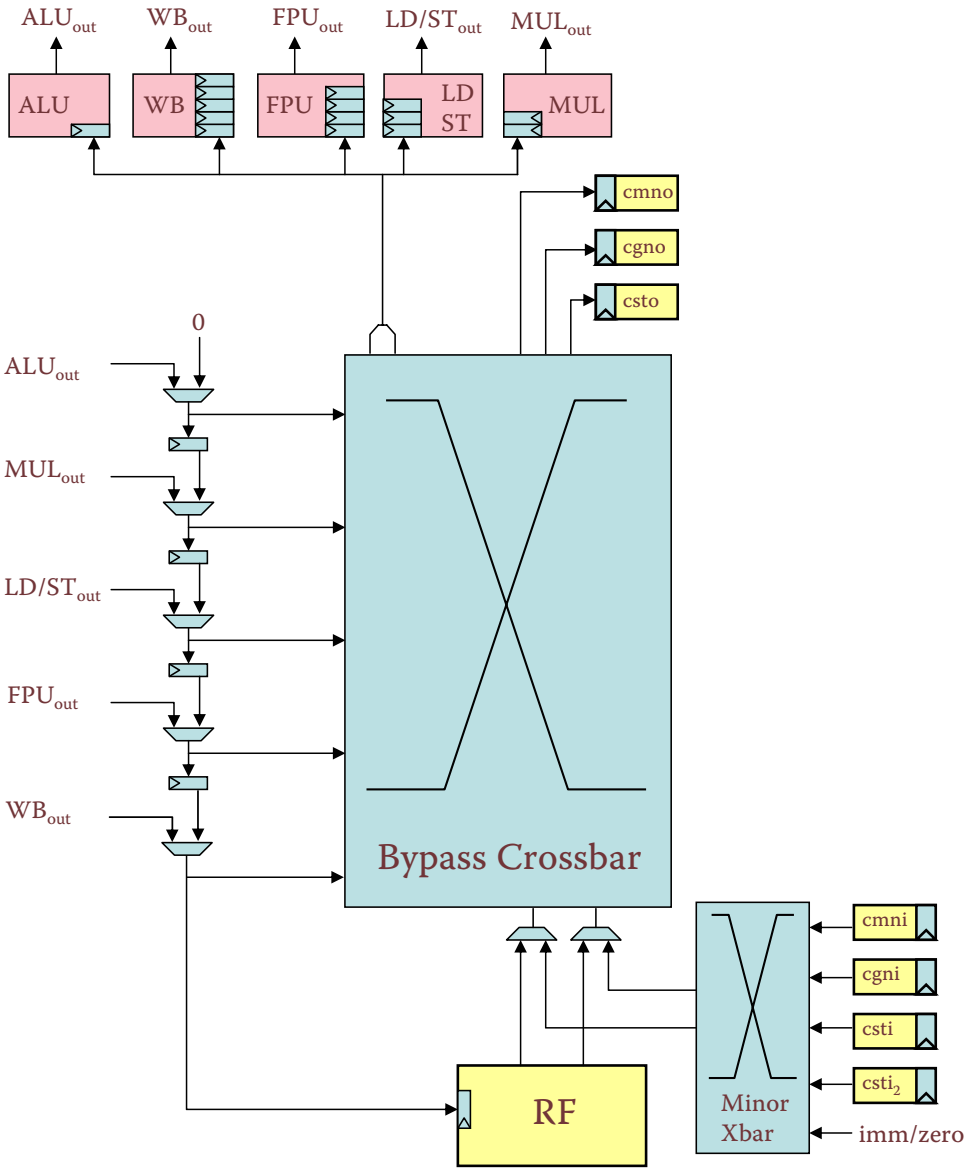


Figure 3-3: Execution Core of the Raw compute processor, comprising the intra-tile SON, functional units, and network input blocks ("NIB"s) that connect to the external networks. With the exception of the functional units, all of the logic is part of Raw's SON.

of the pipeline (i.e., the point at which the instruction can no longer be squashed) is quite early – as soon as an instruction passes the IS stage, the instruction is considered to have completed. This is the case for two reasons. First, instruction may have already dequeued elements from the input NIBs. Second, the inverse-bypass logic may have already forwarded the operand out into the inter-tile networks, at which point the side-effects of the instruction are no longer containable.

When an instruction targets an output NIB, the output NIB must have sufficient room to receive the operand when the operand becomes available. On the surface, it seems like this problem could be solved either by squashing the appropriate instructions, or by stalling the pipeline until the NIB

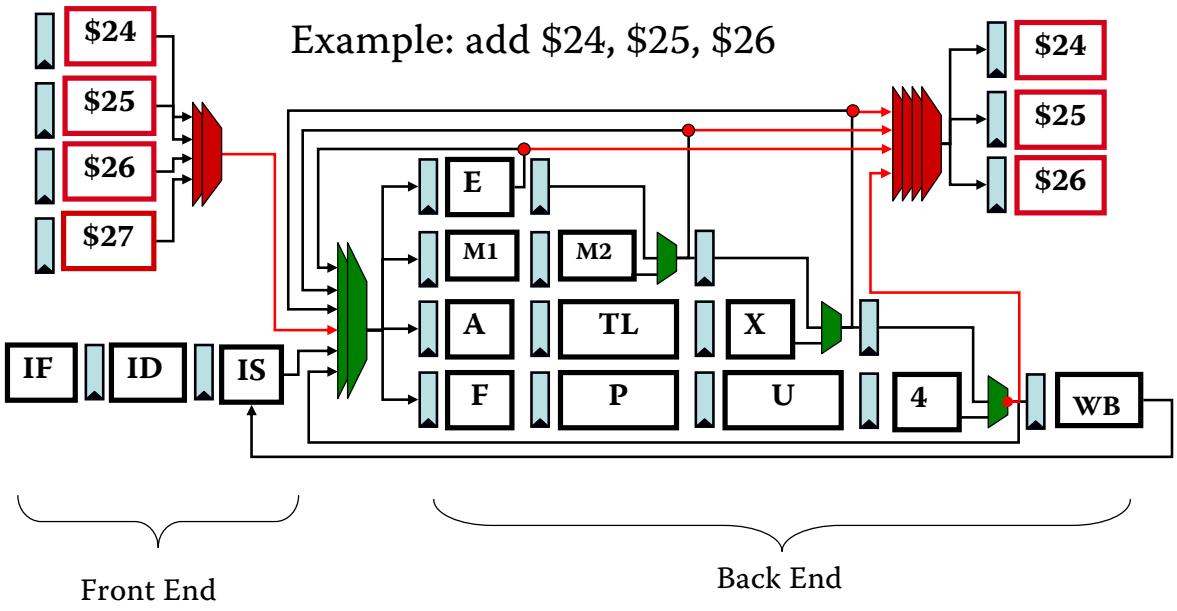


Figure 3-4: Traditional 8-stage pipeline representation of the tile's execution core and fetch unit (collectively called the tile's "compute processor"). The reserved register numbers are shown instead of NIB names. The front-end pipeline stages are instruction fetch (IF), instruction decode (ID), and instruction stall (IS). The back end of the pipeline is comprised of the various pipelines stages of the functional units. The passage of an instruction from the front-end of the pipeline to the back-end marks the point of commit. The intra-tile SON is shown in the form of the more traditional bypass network representation. An example instruction, `add $24,$25,$26`, which reads from two NIBs and writes to a third, is shown.

empties. However, the instructions in the back-end have already technically committed, so squashing is not a possibility. Furthermore, because there is no upper-bound on how long the NIB may remain full, stalling would make it unwieldy to implement context-switching in the pipeline – essentially, the entire state of the pipeline (including whether each instruction has yet to transmit its outputs to the inter-tile networks) would need to be saved. Instead, we would prefer to summarize the state of the pipeline with the PC of a single instruction to execute, in addition to the program's register file and NIB state.

To address this problem, the issue logic conservatively checks that there will be free space in the output NIB before allowing the instruction to leave the IS stage (and commit). It assumes pessimistically that nothing will be removed from the NIB in the intervening cycles, and accounts for operands that are in-flight in the pipeline but have not been written. The sizes of the corresponding output NIBs are increased by the bandwidth-latency product of the backend (i.e., 5 elements) so that the conservative nature of this check does not impact performance in the common case. Using this system, a context switch cleanly interrupts the execution of the program at the IS stage.

3.2.1.2 Arbitration for the trusted core network

The trusted core network presents a challenge because three parties can access it at once: the instruction cache, the data cache, and the instruction stream. Because messages on the trusted network are multi-word, there is a need to ensure that one party does not try to write to the network before another party has finished its message. Although it is easy for the hardware to ensure that the two caches "take turns", the interactions with the programmer-specified instruction stream are more complex. The *memory lock* (`mlk`) and *memory unlock* (`munlk`) instructions are used to indicate to the fetch unit that it must prefetch a region of code before executing it in order to ensure that the instruction cache misses do not occur while the program is in the middle of transmitting messages to the trusted network. Further, code that accesses the trusted network must make sure that memory instructions do not cache miss while in the middle of composing a message. Finally, the (`mlk`/`munlk`) instructions disable and enable interrupts in order to ensure that interrupts do not occur in the middle of message composition.

A few interlocks have been put into place to facilitate cleaner sharing of the trusted core network. The inverse bypass logic conservatively assumes that load/store instructions may access the trusted core network, until those instructions pass the stage at which they may cache miss (TL). Further, the cache miss state machine waits for the older instructions in the pipeline to drain (and all messages to enter the network) before initiating cache messages on the trusted core.

3.2.2 Raw's Scalar Operand Network

Raw's inter-tile SON is implemented using a *static router*, which is a input-buffered (i.e. with NIBs) crossbar controlled by a simple processor, the *switch processor*. The switch processor has a local instruction cache from which it sequences a series of wide instructions. Each switch instruction specifies a simple operation (flow control and data movement) and a number of routes. These routes control the static router's crossbars, which route values from its receiving NIBs to the inter-tile network links and/or the compute processor. The static router executes each instruction atomically. As a result, it must verify that every specified route has data available in the appropriate NIB, and that the destination NIB on the remote tile has sufficient buffer space to receive the element.

To determine if there is space available in the remote NIB, the tile maintains an internal counter which conservatively estimates the amount of free space. Every time the tile transmits a value to a remote tile's NIB, it decrements the counter. Every time the remote tile dequeues a value from the corresponding NIB, it sends a *credit* back to the original sender, which signals the sender to increment the counter. As long as the counter is non-zero, it is safe to send.

If these checks are successful, then all of the side-effects of the instruction occur atomically – all values are simultaneously dequeued from their respective NIBs (sending a credit to the neighboring

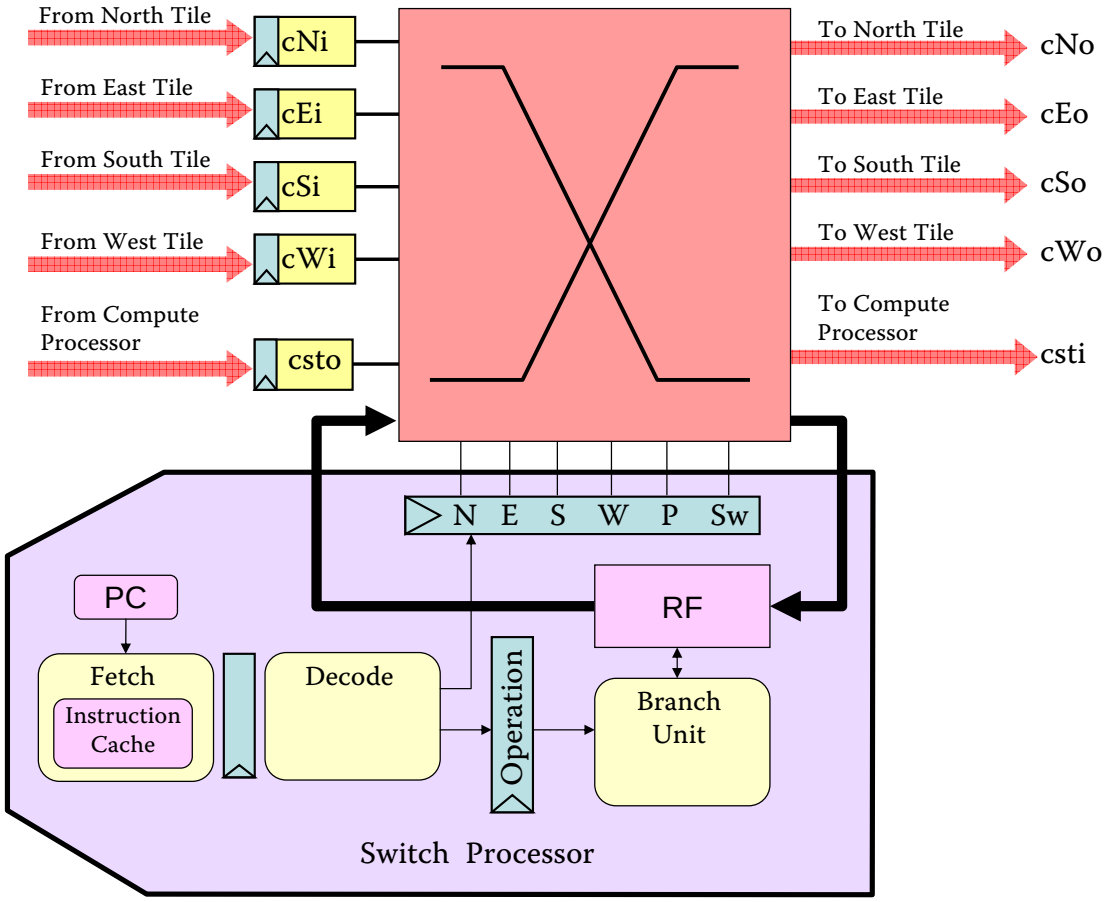


Figure 3-5: Basic Static Router Architecture. Raw's static router is similar, except it has two such crossbars and sets of network connections. See Appendix A and, in particular, Figure A-4 for more detail on the crossbars and connections.

tile indicating that one element has been freed up), and routed through the crossbar to the neighboring tile. Typically a single *valid* bit is routed along with the operand in order to indicate that a value has indeed been transmitted.

The basic static router architecture is shown in Figure 3-5. Raw's static router, described in more detail in Section A.2.2, extends this basic static router architecture by doubling the number of crossbars and inter-tile networks, thereby providing twice the network bandwidth (these two physical inter-tile networks are two of the four networks shown in Figure 3-1.)

Although the static router has a memory-backed instruction cache, this alone is not sufficient to completely inform the static router on the routing patterns it needs to take. It also needs to be informed of branch-condition values that control the control-flow of the program at hand. For this purpose, the crossbar can also route values to and from the switch processor. Thus, the crossbar routes not only between the north, east, south, west directions and the compute processor, it also routes to and from the local switch processor. These values are typically branch conditions and/or loop counts.

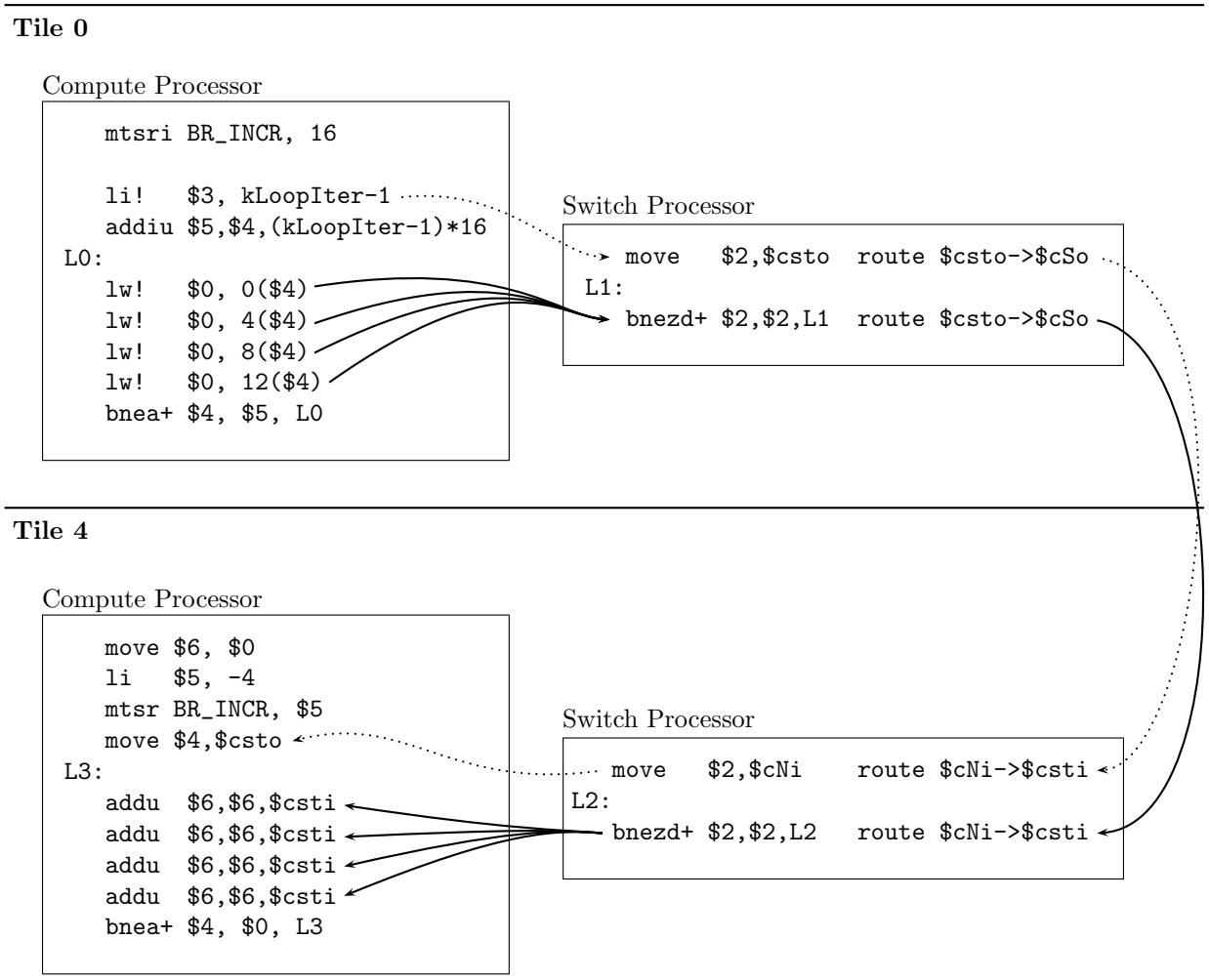


Figure 3-6: An example of two tiles using the inter-tile SON to communicate. The ! symbol indicates that the value should be transmitted from the output of the tile's functional unit to the csto NIB, which is the entry point to the inter-tile SON. The inter-tile SON is composed of a mesh network of static routers, which each employ a *switch processor* that controls the networking resources. The switch processor sequences an instruction stream that contains flow control instructions (including a branch-and-decrement), move operations (for accessing the switch's local register file) and route operations (which send values to other tiles, or to the tile's local compute processor). The arrows indicate the compile-time specified paths of operands over the inter-tile SON. This example is explained in more detail in Section A.2.3.

Figure 3-6 shows an example program that employs the SON to communicate between tiles. The arrows indicate the flow of operands over the SON. Since Raw is an SSS architecture, the *assignment* of operations to tiles, the *transport* of operands between tiles, and the *ordering* of operations on the tiles are all specified by the compiler. In order to specify the transfer of an operand from one tile to a remote tile, the compiler must ensure that there is a corresponding route instruction on every tile along the path that the operand takes.

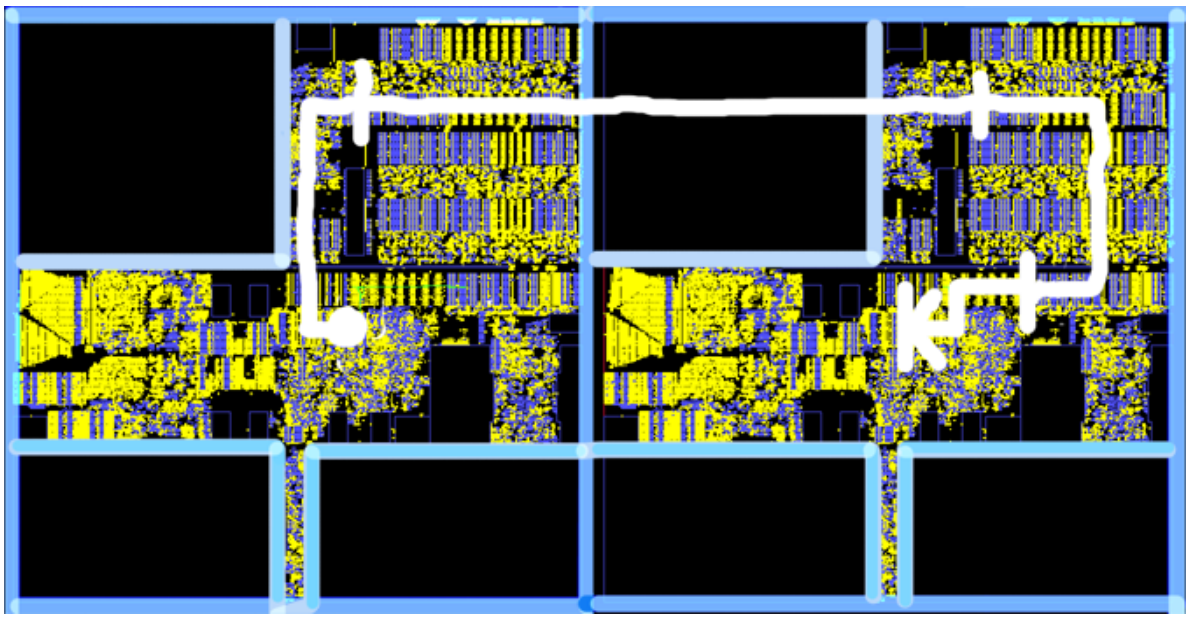


Figure 3-7: The path taken by an operand when it is routed from the output of an ALU on one tile to the input of an ALU on another tile. The picture shows a computer-generated layout of two Raw tiles. The white line is the path taken by the operand. The tick marks indicate cycle boundaries. The floorplan in Figure 4-10 may be used to correlate features in the picture.

3.2.2.1 Raw's 5-tuple Derivation

In this subsection, we examine the 5-tuple derivation for Raw's inter-tile SON. Figure 3-7 shows a computer-generated layout (refer to Figure 4-10 for a tile floorplan) of two neighbor tiles on the Raw chip. The white line is the actual physical path taken by the operand through the silicon as it leaves the ALU of one tile and is routed to the ALU of another tile. The hash marks indicate cycle boundaries. The operand is computed by the sender's ALU, and is reverse-bypassed out to the input of the csto NIB in the same cycle (0 cycles total send latency "SL"). In the next cycle, it is routed from the sender's static router to the east neighbor's static router (1 cycle network hop latency "NHL"). In the following cycle, it is routed from the east neighbor's static router to the east neighbor's compute processor csti NIB. Then, in the next cycle, the east neighbor's compute processor recognizes the arrival of the operand ("wakes up") and dispatches the instruction (2 cycles total receive latency "RL"). Because Raw employs statically-ordered, register-mapped, flow-controlled communication, there are no additional instructions required to send or receive operands (0 cycles send occupancy "SO" and 0 cycles receive occupancy "RO"). Thus, Raw's 5-tuple in total is <0,0,1,2,0>.

Analysis of the Raw design suggests that the last cycle of latency could most likely be eliminated by sending wakeup logic between tiles earlier, yielding a 5-tuple of <0,0,1,1,0>. Approximately 40% of the network hop latency is due to wire-related (including signal buffering with inverters)

delay [112].

3.2.3 Raw's Trusted Core

The principle purpose of Raw's trusted core is to provide deadlock-free communication between tiles and external I/O and memory devices. It employs a single physical wormhole-routed generalized transport network, called the memory dynamic network ("MDN") as transport, and uses the *message sinkability* deadlock-avoidance discipline. Thus, a message can only be transmitted on the network if it is known that the recipient will be able to remove it from the network to prevent cycles in the channel dependence graph. We selected this discipline over using multiple logical networks because it can be implemented with less area, and as a result, minimizes tile sizes.

Since the tiles' data caches rely upon the MDN network to access external DRAM, the tiles themselves do not have enough buffer space to handle the *flash mob* problem, especially for 1024 tiles systems. The solution we selected to address this issue is to forbid the sending of "unsolicited" messages to tiles on the MDN. With two exceptions, only messages which are in response to a tile's request may be sent to a tile over the MDN. The two exceptions are for two special classes of messages. The first class, interrupt messages, are sent to a tile on behalf of the interrupt controller, and serve simply to set a bit in the tile's interrupt status register. The second class of message, store acknowledgment messages, increments one of two counters in the tile, and thus is also always sinkable. Since, in both cases, there is no resource that needs to be reserved inside the tile to process the message, the message is always sinkable.

Although disallowing unsolicited messages handles the flash mob problem for tiles, tiles still must make sure that their requests do not result in a flurry of reply messages that they do not have space to store. In the case of the data cache, this is generally not a problem, because the data cache can control how many outstanding requests it has, and ensure that it has the necessary resources to dequeue or buffer the reply messages. In the Raw implementation, the data cache can have only one outstanding miss request, for which it has evicted the corresponding cache line to create the space for[3]. For programmer-based communication with I/O devices and DRAM over the trusted network, the user must be able to consume the messages as they arrive, or to reserve space in the cache (and ensure those lines are resident so touching them does not cause cache misses) to buffer replies.

For devices connected to I/O ports, buffer space is more readily accessible, and so communication is less restricted. Nonetheless, implementing the *message sinkability* discipline requires careful accounting. The system needs a facility for tracking buffer space. For protocols that have a request

[3]Technically, the miss request is sent out before the corresponding line is evicted, so it's reasonable to imagine a case where the outgoing network link is blocked, preventing the line from being evicted, which in turn prevents the miss reply from being consumed. However, the cmno NIB is sized to 16 elements so that it can hold a complete eviction message (10 words). We can prove that this is sufficient to avoid deadlock because in order for a miss response to be received by the tile, the miss request must have been processed by a remote device, which means that it has left cmno completely. Since there are no intervening messages between the request and evict messages, cmno will always have space to hold the entire evict request – 10 words.

packet for every reply packet, it is easy enough for the client to count how many outstanding requests have been issued. However, some protocol do not inherently have replies – in these cases, the Raw system uses a *store acknowledgment* message as a synthetic reply. One such case occurs in the tiles' data caches. Although a cache line fill request is a request-reply message, a cache line evict request is not – it simply specifies that cache line should be written to a DRAM. As a result, the number of fill requests is not inherently rate-limited by the system. To address this issue, Raw tiles maintain a store acknowledgment counter ("store meter"), which maintains the number of outstanding evict requests that each tile is allowed to have. When an evict request is injected into the network, this count is decremented; when the DRAM responds with the store acknowledgment message, the count is incremented. If the count is zero, then the tile will wait for a store acknowledgment to arrive before issuing more evict requests (and any dependent fill requests!). This system allows the tile to have multiple parallel pending evict packets, which improves performance when transferring streams of data. The store acknowledgment system is also used for direct communication between I/O devices – for instance between a video camera and the DRAM that it is streaming data to. The ability to have multiple outstanding packets (and to not wait for reply messages) is an important capability for high-throughput communication between devices.

The use of store meters creates some scalability concerns. First, we would like to avoid the need for every node to have a meter for every other node. One technique is to use a single store meter to bound space usage across multiple destination nodes. For example, if two destination nodes each have m messages worth of buffer space reserved for the sender node, the sender's meter would be initialized with the value of m, indicating that it may have at most m outstanding messages to any combination of the destination nodes. This conservative approach trades off store meters for buffer space.

The store meter technique can be extended by associating with each store meter a tag which indicates the specific ports for which it is associated. This allows the system to configure the store meter usage. Some meters may be set to a low value but with a tag that covers a large number of nodes, providing basic access to a large number of nodes. Others may have a high value but a tag that covers a small number of nodes, providing high bandwidth to a selected subset of nodes.

The Raw tile's data cache supports two store meters. The first store meter contains a tag which allows a "partner" I/O port to be selected for high-bandwidth access. The second store meter is tagless and counts storage acknowledges for all ports that are not handled by the partner store meter. These store meters are described further in Section B.4. In the Raw system, the memory management system strives to allocate a tile's memory objects to those ports that would provide the highest bandwidth.

In the Raw I/O system, buffers are allocated and de-allocated according to the I/O transactions at hand. To do this, chipset designers may be required to supplement each I/O device with a number

of configurable store meters, which can be configured by the OS or device drivers.

3.2.3.1 Privileged access to Trusted Core's MDN

As pictured in Figure 3-3 and Figure 3-4, the trusted core's generalized transport network is accessible via a register-mapped interface. Thus, instructions such as `move $cmno, $csti` are legal instructions. Although Raw does not implement a hardware protection model, access to the MDN is intended for privileged users (e.g., system and device driver-level programmers) rather than for end-users. When extending the trusted core, it is the privileged users' duty to ensure that the conglomeration of protocols running on the trusted generalized transport network continues to adhere to the *message sinkability* deadlock-avoidance discipline.

3.2.4 Raw's Untrusted Core

Extending Raw's trusted core with additional protocols requires users to prove deadlock properties. In many cases, the burden of formulating deadlock avoidance proofs is too great to place upon most users. For this reason, the Raw architecture provides the untrusted core, which employs the GDN wormhole-routed network in conjunction with a spill/restore mechanism like that shown in Figure 2-26 to implement virtual buffering. Much like the MDN, the GDN is accessible through a register mapped interface (`$cgno` and `$cgni`), which provides low-latency communication between nodes. When using the GDN, the user does not have to worry about deadlock, or about exceeding the buffer space in the system.

In the Raw design, we opted to implement the spill/restore virtual buffering mechanism through a combination of hardware and software. Inspired by the Alewife scheme [76], the Raw tile uses a *deadlock detection* heuristic to conservatively[4] estimate when deadlock has occurred in the network. When the heuristic believes a deadlock has occurred, it signals an interrupt. The corresponding interrupt handler then removes data from the `cgno` NIB and stores it in a software buffer. This software buffer resides in the cache, but may be flushed out to DRAM through the normal operation of the cache using the trusted core). It then enables the `GDN_REFILL` control register bit, which enables the restore mechanism. The restore mechanism maintains a control word, called the `GDN_REFILL_VAL`, which is the value that the next read from `$cgni` will return. After this word is read, a `GDN_REFILL` interrupt is fired. The corresponding interrupt handler will then load the next value into `GDN_REFILL_VAL`. If no more values are available in the local buffer, then the `GDN_REFILL` control register is cleared, and reads from `$cgni` will once again return values from `cgni`. More detail is given in Section B.4.

The deadlock detection heuristic in the Raw machine is implemented using per-tile watchdog

[4]Conservative in the sense that it triggers upon a condition which is a necessary but not sufficient condition for deadlock.

timers[5]. The watchdog timer can be programmed to fire an interrupt based on the number of cycles that data has sitting untouched in cgni. It also can be set to fire an interrupt based on the number of cycles that the processor has been blocked on an instruction. Each tile may use its watchdog timer in isolation from other tiles, or as a building block for a more sophisticated inter-tile deadlock detection algorithm that synchronizes using interrupts over the trusted core.

3.2.4.1 Context switching on the untrusted core

Since the untrusted core is a user primitive, we would like to be able to context switch the network, i.e., save and restore the state of the network. Fortunately, the spill/refill mechanism used for virtual buffering can be easily adapted for this purpose. Upon a context switch interrupt, the operating system can interrupt all of the tiles in a process and drain their cgni NIBs into the corresponding software buffers. Upon returning, the operating system can enable the refill mechanism so that the tile reads from the software buffers instead of cgni.

There is, however, one minor catch, which is that tiles may be in the middle of transmitting a message. In order to clear the network for the next process to run, it is necessary to complete the corresponding messages and free up the corresponding wormhole channels. To handle this case, the system provides a counter, `GDN_PENDING` which tracks the number of remaining words require to finish the current message. Upon a context-switching interrupt, the OS checks to see if the thread is in the middle of a message send. If it is, then it enables a `GDN_COMPLETE` interrupt, which fires when the message is completed. It also enables a timer which is used to bound the number of cycles that the thread has to finish the message. When the thread completes its message, context switching can proceed.

In order for this mechanism to work, user threads have to be constructed such that they can eventually complete a pending message even if other tiles have been interrupted for context switch. If the user process does not make sufficient progress in completing a pending GDN message, it can be killed, and the OS will send the appropriate number of words to clear the network. As a result, the necessary condition is that completion of a GDN message, once started, cannot depend on other tiles' forward progress. A simple exception to this case, which is useful in practice, is that a tile may depend on the completion of a GDN message that is already in progress.

An alternative to this approach, which is somewhat cleaner, is to employ a commit buffer, which buffers the words of a message until it has been finished, and then allows it to enter the network. Although this alternative is more user-friendly (because it does not place restrictions upon the users), it increases the latency of messages in comparison to the approach used in Raw.

[5]The watchdog timer is controlled by the `WATCH_VAL`, `WATCH_MAX`, and `WATCH_SET` registers described in B.4. These determine the conditions that trigger the `TIMER` interrupt.

3.3 The Raw I/O System

The Raw I/O system is implemented as a series of I/O ports, which connect the edges of the on-chip networks to the external pins. Since the full exposure of the on-chip networks would require 16 *links* $\times$ 4 *networks* $\times$ (32 *data* + 1 *valid* + 1 *credit*) $\times$ 2 *directions* = 4352 pins (see Section 3.2.2 for descriptions of valid and credit lines), and the Raw package only has 1052 available pins, some mechanism is required to address the mismatch. There are two basic approaches, both of which are employed in Raw. The first approach is simply to drop connectivity; to eliminate networks links. In the case of Raw, we also wanted Raw chips to be gluelessly connectible to form larger virtual Raw chips. This eliminates the possibility of dropping links in the dimension-ordered wormhole routed networks (i.e., the MDN and GDN), because the absence of links can prevent tiles on different chips from messaging each other. However, in the case of the static router, this was more tolerable, because the compiler can simply chose not to use the corresponding links[6]. The second approach, shown in Figure 3-8, is to multiplex networks over the physical links. On the sender side, a round-robin arbiter can alternate between networks with are ready to transmit, forwarding each available word with a tag that indicates which network it originates from. On the receiver side, the tag is examined and the data placed in the appropriate input NIB. It is the responsibility of the arbiter and demux logic to manage the credit-based flow-control system to prevent overflow of buffers.

The final Raw design does not expose the second physical static network to the periphery I/O, relying upon the compiler to avoid the use of these links. It furthermore multiplexes the three networks - GDN, MDN, and static network down onto one set of physical pins. This brings the pin requirements down to 16 *links* $\times$ (32 *data* + 2 *valid* + 3 *credit*) $\times$ 2 *directions* = 1184 pins. In order to bridge the remain shortage of pins, the chip further multiplexes two such multiplexed links, effectively creating two 6:1 multiplexed links on the top and bottom center tiles. This sharing is depicted in Figure 3-1. The final pin requirements of this configuration are 12 *links* $\times$ (32 *data* + 2 *valid* + 3 *credit*) $\times$ 2 *directions* +
2 *links* $\times$ (32 *data* + 3 *valid* + 6 *credit*) $\times$ 2 *directions* = 1052 pins, which was within the package capabilities.

Generally, because of the ability to drop connections, tiled microprocessors are fairly flexible in terms of the number of I/O ports they support, and thus scale relatively well with varying technology parameters. However, as tiled microprocessor are scaled up through Moore's Law, glueless interconnection of separate chips may require higher and higher levels of multiplexing which eventually may motivate more sophisticated dynamic networks that allow partial connectivity between chips.

[6]The resulting non-uniformity of the multi-chip array does create some limitations for an OS which is trying to map applications to a subset of tiles on the array.

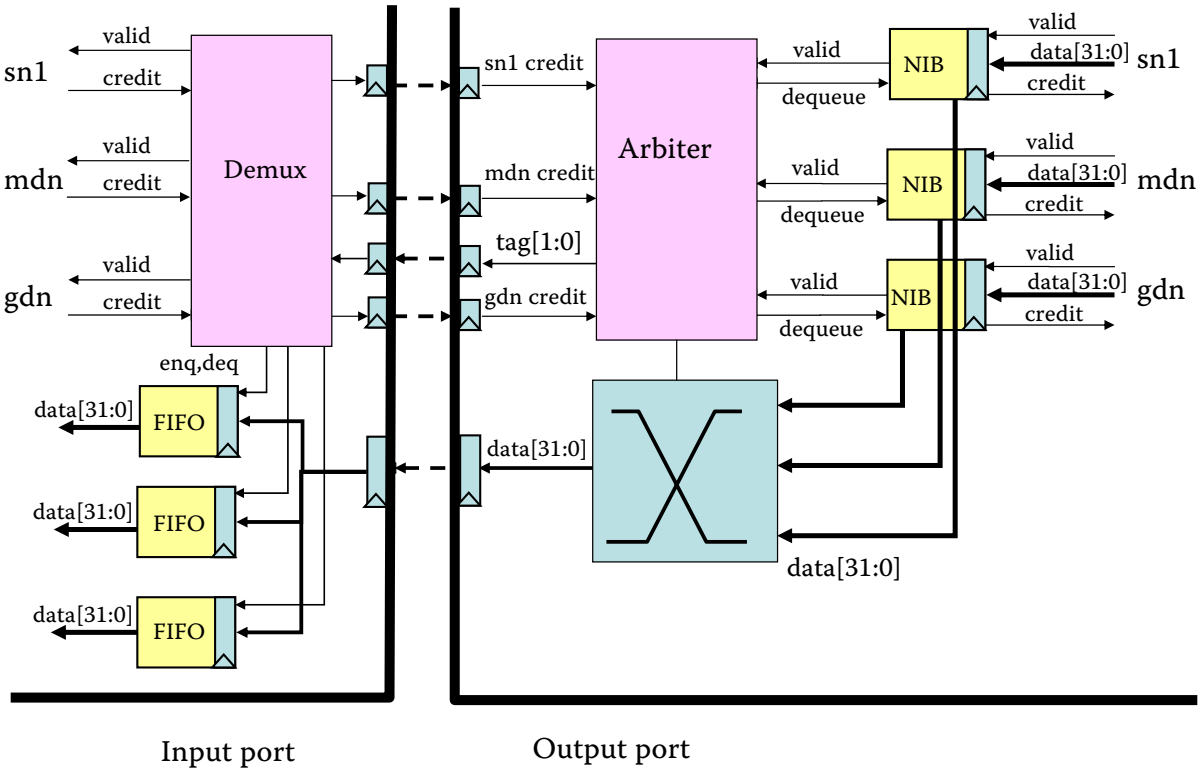


Figure 3-8: Multiplexing of I/O ports. A 3:1 output port and 1:3 input port are depicted, communicating across the pins between two chips. The 3:1 output port's job is to multiplex three networks across one physical set of pins, using a tag to indicate which network the current data word corresponds to. The role of the 1:3 input port is to demultiplex the three networks from the one set of physical pins. The arbiter and demux are responsible for managing the credit-based flow control system to prevent overflow of buffers.

3.3.1 Raw I/O Programming

The Raw system controls off-chip I/O devices by sending them messages. Request messages travel through the on-chip networks, through the I/O ports, and over the pins to the I/O device. Reply messages travel in the opposite order. Each I/O device has the support logic to interpret these messages, turn them into the appropriate device behavior, and emit the reply message.

In the Raw system, I/O devices may receive data via the first static network, via the GDN, and via the MDN. Which network is used when? Typically, any message which is not-compile predictable (such as a cache-miss) is transmitted via one of the dynamic networks. The static network is best suited for compile-time predictable data streams. This is because the static network guarantees the arrival order and routing paths of messages. As a result, the static network can be scheduled to provide high levels of network utilization (relative to networks with dimension-ordered wormhole routing, which have suboptimal routing, and tend to have poor fairness properties under heavy load) and tiles do not have to execute code to de-multiplex incoming packets as they arrive. Furthermore, the use of the message sinkability discipline and virtual-buffering provide challenges for streaming

models.

In practice, we've found that the MDN and GDN are useful for controlling I/O devices and routing non-compile time predictable requests. The static network has been most useful for routing compile-time predictable streams of data that the devices emit or consume. The MDN network is used for high-priority traffic, such as cache misses. An example device that we employed is a *streaming DRAM controller*, which accepts packets over the GDN that specify a starting address, a stride, and a number of words. The program can queue these packets up in the DRAM controller's queue, and then it can branch to the portion of the code that routes the resulting words in over the static network, and consumes and computes on them. An important constraint is that the tile may have instruction or data cache misses that are required to process the incoming streams. In this case, it is vital to ensure that these misses are not stuck behind the streaming memory requests; otherwise deadlock will occur. This could in theory be solved by implementing separate queues at the I/O controller for each level of priority (cache misses being of highest-priority), but in practice it is easier to rely on the fact that cache misses are routed over the MDN network. Messages that arrive at I/O devices over the MDN are of higher-priority than those that arrive over the GDN. I/O devices thus must be able to guarantee that MDN requests will not be blocked behind GDN requests.

3.3.1.1 Avoiding deadlock with I/O devices

For communication with I/O devices over the MDN, the message sinkability discipline must be obeyed, as with all parts of the trusted core. For communication with I/O devices over the GDN, there are two choices. The device itself may choose to implement *virtual buffering* – this is in a sense the most consistent with the philosophy of the untrusted core. However, in some cases, implementing virtual buffering may be a large burden to place on some I/O devices. In these cases, it may be more practical for the system to require that I/O programmers employ the message sinkability discipline with respect to the I/O devices themselves. In this case, virtual buffering will apply to all messages sent to tiles, but message sinkability applies for all messages sent to I/O device. This hybrid scheme ensures deadlock-avoidance by ensuring that all endpoints are able to sink incoming messages (through whichever mechanism.) Over the static network, the occurrence of deadlock is a deterministic affair and the result of incorrect scheduling.

3.3.1.2 User-level I/O Devices

The use of I/O devices over the untrusted core raises some complex questions about the system stability guarantees that we intend to provide. If a user process is killed in the middle of a series of I/O requests (over the GDN or static network) to devices, the devices would need to support a mechanism through which these requests can be aborted or completed. Further, in a system that

is expected to support context switching, we would need to be able to interrupt and resume these I/O requests. The Raw system optimizes performance over some of these usability constraints (as might be appropriate for a high-performance embedded processor); in some cases context-switching programs that employ direct I/O may be a possibility; but generally we expect that these applications will be mapped to the Raw array until they complete.

3.4 Summary

In this chapter, we examined the Raw, the tiled architecture that we implemented at MIT. We examined how Raw addresses the seven criteria of physical scalability to scale to systems with 1024 tiles. Raw's Scalar Operand Network attains an 5-tuple of <0,0,1,2,0> through aggressive architectural optimization. According to the AsTrO taxonmoy, Raw employs an SSS SON; thus it uses the compiler to determine instruction assignment, operand routing and instruction ordering. More details on the Raw architecture can found in Appendices A and B. In the next chapter, we examine Raw's 180 nm VLSI implementation.

Chapter 4

The Raw Implementation

This chapter examines the Raw VLSI implementation. It begins with an analysis of the 180 nm VLSI process in which the Raw microprocessor chip was implemented (Section 4.1). After this analysis, the chapter describes the Raw chip (Section 4.2), as well as the Raw systems that are constructed using the Raw chip (Section 4.3). The chapter concludes by quantifying, relative to the Intel Pentium 4, the benefits that tiled microprocessors carry for design, implementation, and verification (Section 4.4).

4.1 The Building Materials

The properties of the underlying building materials – propagation delays, area, and power – ultimately determine many of the properties of a microprocessor. The first step in building a novel system out of silicon is to examine the available resources and estimate their impact on the architectural and micro-architectural design. In the following subsection, we briefly overview the basics of the standard-cell abstraction, a common VLSI design style that we employed in constructing the Raw prototype.

4.1.1 The Standard-Cell Abstraction

Basic digital design classes typically teach the *gate-level* abstraction - that is, the representation of a logic function as a graph (or *netlist*) of *gates*. Each gate implements a simple logic function. Figure 4-1 shows an example of a netlist of gates. AO22 is a *compound gate* composed of several basic AND and OR gates. DFF is an edge-triggered flip-flop, used to hold state.

The gate-level abstraction is useful in representing the logical structure of a digital circuit. However, it is limited in its ability to express physical properties of a digital circuit. To effectively design a VLSI chip, we need to specify where gates are physically located on a VLSI chip, the size and

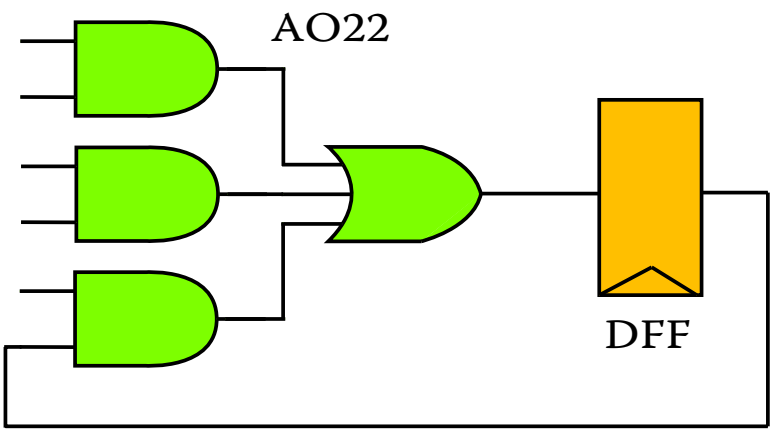


Figure 4-1: An example of some digital logic represented in the gate-level abstraction.

electrical drive strength of the gates, and the topology of the wires that connect the gates. Chip designers use the *standard-cell* abstraction to express this.

Figure 4-2 shows the same AO22 and DFF gates, represented in the standard cell abstraction. Unlike in the gate-level abstraction, the dimensions and position of the gates actually reflects their relative sizes and positions in the final VLSI design. As is evident from the diagram, the two gates (called *standard cells* in this context) have significantly different sizes. The sizes are measured in terms of *wiring tracks*. A *wiring track* is a space wide enough to contain a minimum width wire and the spacing around it that separates it from other wires. In the 180 nm IBM SA-27E process that the Raw microprocessor was implemented in, a wiring track is .56 microns wide. With a few exceptions, standard cells in the IBM SA-27E are 12 wiring tracks high. Thus, the dimensions of IBM's standard cells vary principally in their width. The AO22 standard cell shown in Figure 4-2 is 13 wiring tracks wide. On the other hand, the DFF standard cell is 24 tracks wide. In the standard cell process, wires are aligned to wiring tracks, and standard cells are aligned to *circuit rows*, which are groups of 12 horizontal wiring tracks.

Each standard cell has a number of pins, which correspond to the gate's inputs and outputs. These pins have fixed locations on the standard cell[1]. To connect these pins, it is necessary to route wires between the pins. These wires conform to the boundaries of the wiring tracks. A wire is often said to be composed of *wire segments*, portions of the wire that are straight.

VLSI processes typically offer a number of wiring (or "interconnect") layers. These layers, separated by an insulator, increase the number of wiring tracks that an integrate circuit can employ. Figure 4-3 shows an example of wire segments on 3 metal layers with alternating directions. Along with these layers, VLSI processes also provide metal *vias* that are used to connect wires segments on different layers.

[1]Many processes allow standard cells to be reflected across the Y axis, so this would change the effective locations of the pins.

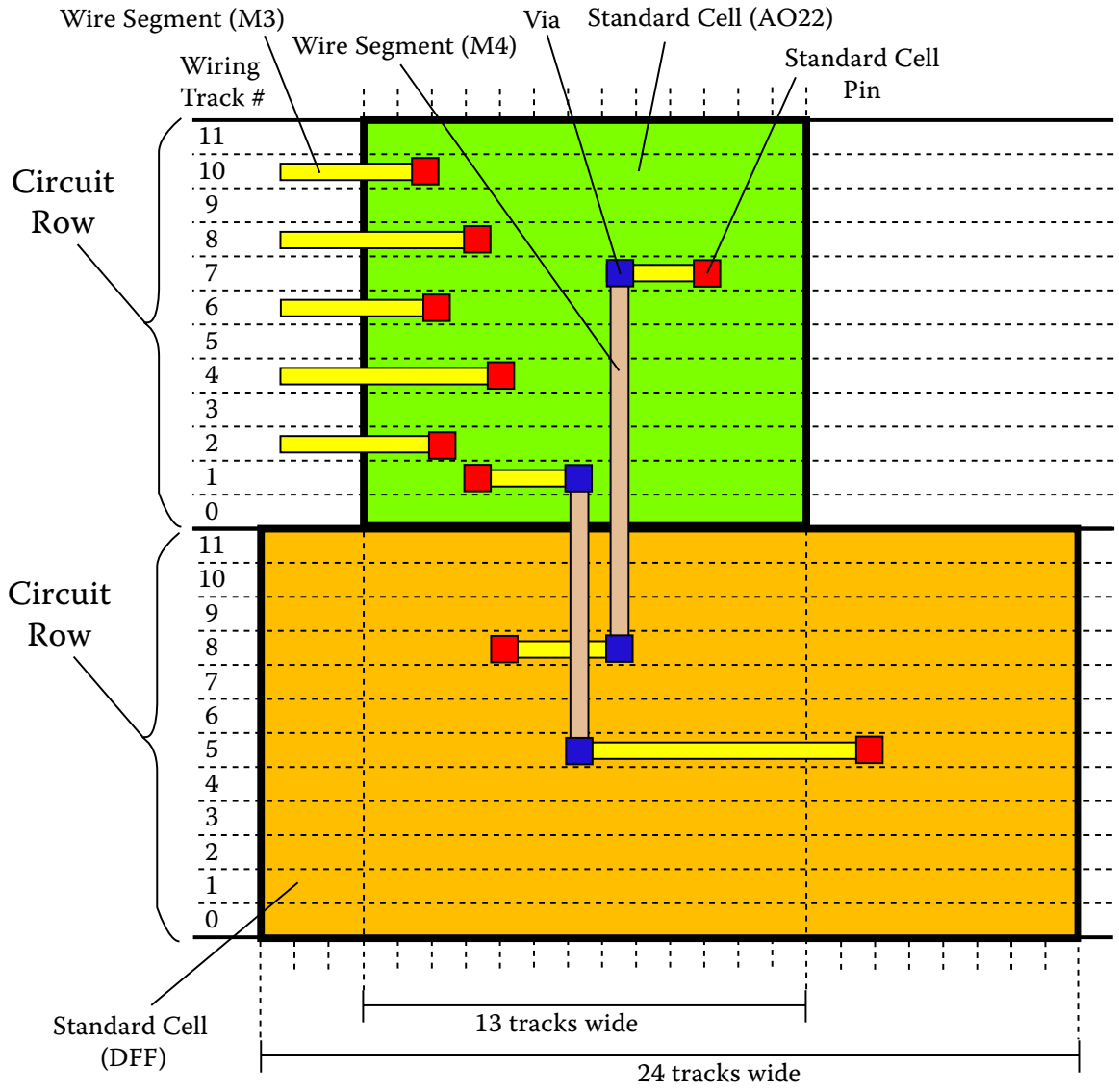


Figure 4-2: The AO22 and DFF gates, represented in the standard-cell abstraction.

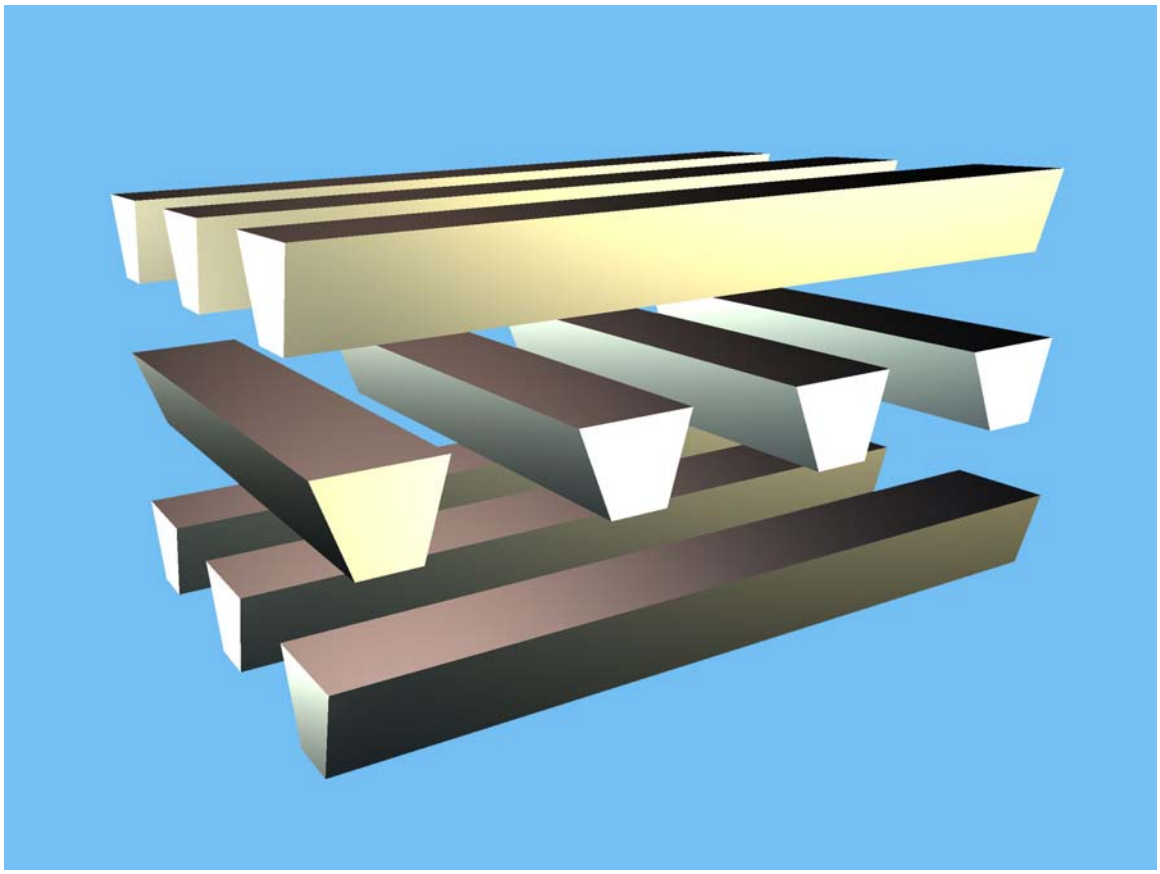


Figure 4-3: 3-D representation of wire segments at different interconnect layers. Alternate metal layers are typically run perpendicularly. The insulator between the metal layers is not depicted.

4.1.2 Examining the Raw Microprocessor's "Raw" Resources

The resources of a VLSI process can be expressed in terms of wiring, gate, and pin resources. Accordingly, in this subsection, we examine Raw's VLSI process in terms of these three classes of resources. Table 4.1 shows the resources available on the Raw die and package. Raw is implemented using an 18.2 x 18.2 mm, 331 mm^2 die in IBM's 180 nm 6-layer copper SA-27E ASIC process. Much as Intel targets large die sizes for new designs in anticipation of future size reductions due to Moore's law[2], we selected SA-27E's largest die size in order to demonstrate the applicability of tiled microprocessors in future process generations. We also selected SA-27E's largest pin count package – a 1657 pin CCGA (ceramic column grid array) package – for the same reason. In this section, we analyze the implications of these parameters on the design space.

[2]anecdotally, 100 mm^2 has been suggested as the ideal die size for desktop microprocessor economics

Chip Parameters	
Die Dimensions	18.2 mm x 18.2 mm
Metal Layers	6
Package Pins	1657
Gates	
Chip Dimensions	18.2 mm x 18.2 mm
Logic Area (Chip area minus perimeter I/O buffers)	17.56 mm x 17.56 mm
Available Circuit Rows	2612
DFF Cells per Circuit Row	1306
$\therefore$ Chip's Maximum Gate Capacity (measured in DFF)	**3.4 Million**
Wires	
Total Metal Layers	6
Available Wiring Tracks (per Metal Layer)	31,350
Free Horizontal Layers (M3, M5, $\frac{1}{2}$ M1)	2+
Free Vertical Layers (M2, M4)	2
$\therefore$ Chip's Free Wire Capacity (measured in 4 mm wire segments)	**501,600**
Pins	
Total Package Pins	1657
Used for GND	-204
Used for Vdd	-119
Used for I/O Vdd	-92
Dummy	-90
Net Usable Signal I/Os	1152
Used for testing, I/O reference, JTAG, etc.	-100
$\therefore$ Available Pins	**1052**

Table 4.1: Resources available on the Raw die and package. Each subsection "Gates", "Wires", and "Pins" shows some of the overheads associated with the resource, and the net available resource to the end chip designer.

4.1.2.1 Wires

The Raw Microprocessor employs 6 metal layers, the most offered through SA-27E, because we wanted to most closely match the number of interconnect layers found in future generations. Overall, the number of metal layers available in VLSI processes has been increasing with subsequent process generations. For example, 65 nm processes are forecasted to have up to 10 layers.

The use of SA-27E's metal layers comes with two classes of restrictions. The first class of restriction deals with the directionality of wire segments on a given metal layer. Of the six metal layers, the routing tools reserve the odd-numbered layers (M1, M3 and M5) for horizontal wires, and the even-numbered layers (M2, M4 and M6) for vertical wires. Since metal layers are typically dedicated to either horizontal or vertical wiring direction, wires with turns are composed of multiple wire segments spanning two or more metal layers along with the vias necessary to connected these segments. Figure 4-2 shows a few examples of wires that require multiple segments and vias.

The second class of restrictions emerges from wiring resources that are implicitly consumed by various functions of the VLSI chip. In the SA-27E process, standard cells utilize much of M1, so wires passing over standard cells have limited ability to take advantage of M1. Memories utilize all of M1 and M2 and approximately half of M3. Finally, M6, which is typically thicker than the others, is largely reserved for wide, vertical, power and ground buses [14, 10]. Additionally, M6 is used to route from the I/O buffers located on the perimeter of the die to the C4's ("Controlled Collapse Chip Connection") that are used to connect to the flip-chip package the encloses the die. Figure 4-4 shows a picture of the surface of the Raw die, in which M6 is clearly visible.

4.1.2.2 Gates

The SA-27E process also offers a menu of die sizes. The die size determines how many wiring tracks and standard cells could theoretically fit on the chip. The Raw die size was selected as 18.2 mm x 18.2 mm, which, after subtracting perimeter area for I/O buffers (.645 mm), is approximately 31,350 x 31,350 wiring tracks. This is enough for approximately 2612 circuit rows. If filled with the DFF's shown above, we could fit 1306 per row, for a total of 3.4 Million DFF. Furthermore, we could route in excess of 501,600 4 millimeter wire segments – the length of a Raw tile – assuming 4 of the 6 metal layers were free. In practice, such a dense design would be so atypical that it would evoke many implementation and testing issues in the ASIC flow.

4.1.2.3 Pins

Along with the choice of die sizes comes the choice of packages. A package is the interface between the silicon die and the motherboard it connects to. It serves two purposes. first it protects the silicon die from damage. Second, it connects the C4's ("Controlled Collapse Chip Connections") on the top-most layer of the silicon die to a set of solder columns. These solder columns are in turn

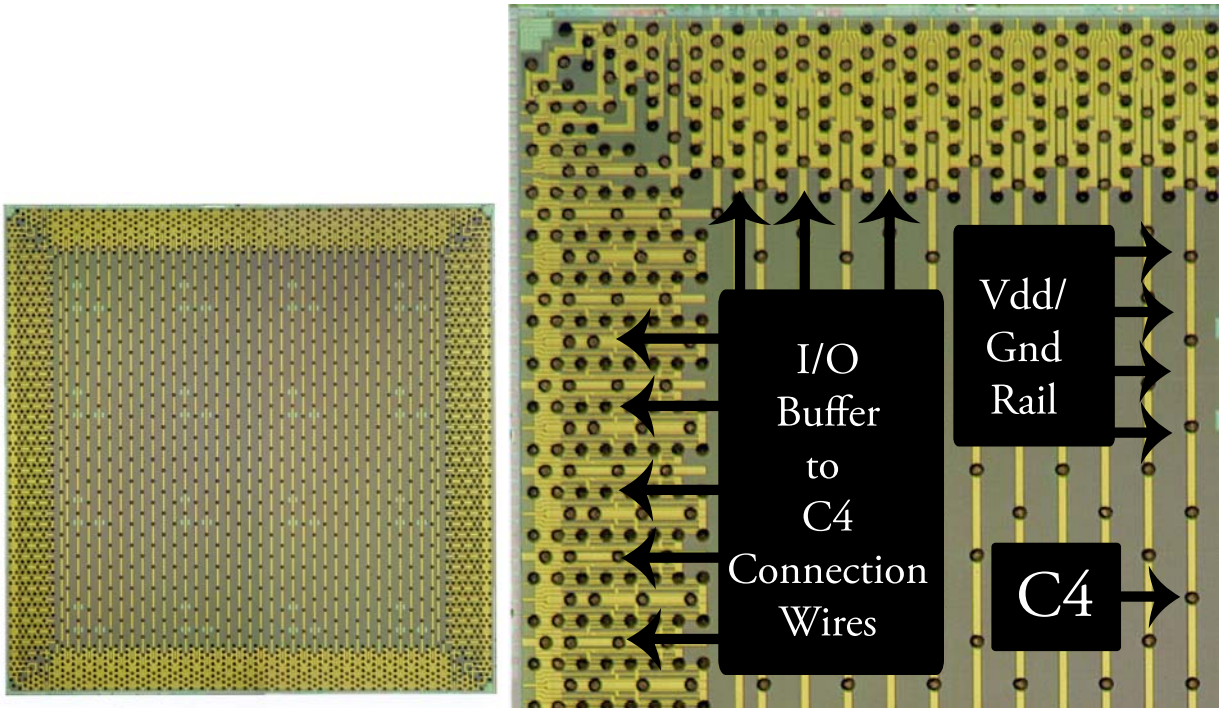


Figure 4-4: The left image is a photograph of the Raw die. Metal Layer 6 is visible, as are the C4's. The image on the right is the upper left hand corner of the same image. The Vdd/Gnd rails, C4's and I/O-Buffer-to-C4 wiring are indicated. The Vdd/Gnd rails are the vertical wires that run the length of the chip. The C4's are the small dots. The I/O-Buffer-to-C4 connection wires are the mass of wires that run from the edges of the chip to the C4's located near the periphery of the chip. Note that they do not appear to adhere to the M6-is-vertical discipline.

Figure 4-5: Ten Raw microprocessors in a chip carrier tray. Photograph: Jason Miller.

soldered to the motherboard which is used to connect the different chips. Inherently, it is performing a widening function; the pitch of solder columns on the package is 1 mm, while the spacing between C4's is as little a .25 mm. To perform this, Raw's package itself is composed of 19 ceramic layers, much like a PC board.

4.2 The Raw Chip

Figure 4-5 shows a picture of a chip carrier tray containing ten Raw tiled microprocessors. Each Raw microprocessor consists of a 18.2 mm x 18.2 mm silicon die enclosed in a 1657-pin 42.5 mm x 42.5 mm CCGA package, as detailed in the previous section.

A computer generated layout of the Raw chip is shown in Figure 4-6. Clearly visible are the sixteen 4 mm x 4 mm tiles. Around the periphery of the chip are the NIBs for the I/O ports and multiplexers, the PLL, and the I/O buffers for the HSTL (High Speed Transceiver Logic). Metal 6 is used to route from these buffers on the periphery to the C4's (such as in Figure 4-4) that connect to the page.

Gate Type	Quantity	Percentage
flip-flops	20,385	26.08%
muxes	16,608	21.25%
2-input nand	10,874	13.91%
inverters	10,345	13.24%
clk-related	4,913	6.29%
2-input and	3,956	5.06%
3-input nand	3,348	4.28%
buffers	2,911	3.72%
i/o related	1,346	1.72%
other	3,475	4.44%
Total	78,161	100%

Table 4.2: Gate types in the Raw "top-level"; includes all logic not included inside tiles, such as I/O ports, wire buffers, and other miscellaneous (such as clock and PLL) logic. The ports themselves comprise 45,118 instances, including 17,770 flip-flops, 13,836 muxes, 6,644 inverters, 2,754 2-input and gates, 911 buffers, and 529 2-input nand gates. The origins of the additional 10,345 2-input nand gates, which were automatically inserted by the ASIC toolchain, are somewhat mysterious.

There is a fair amount of "dust" logic around and in-between the tiles. This logic includes the clock tree, which is composed of a tree of SCBs (structured clock buffers), which are long, narrow arrays of wired-input wired-output inverters, and a variety of inverters and other logic for buffering signals, and for implementing scan and JTAG. Table 4.2 shows the break-down of gate types at the top-level of the design. This includes all standard cells not found inside the tiles themselves. 74% of the cells at the top-level are either flip-flops, muxes or 2-input nand gates. 57% of the gate

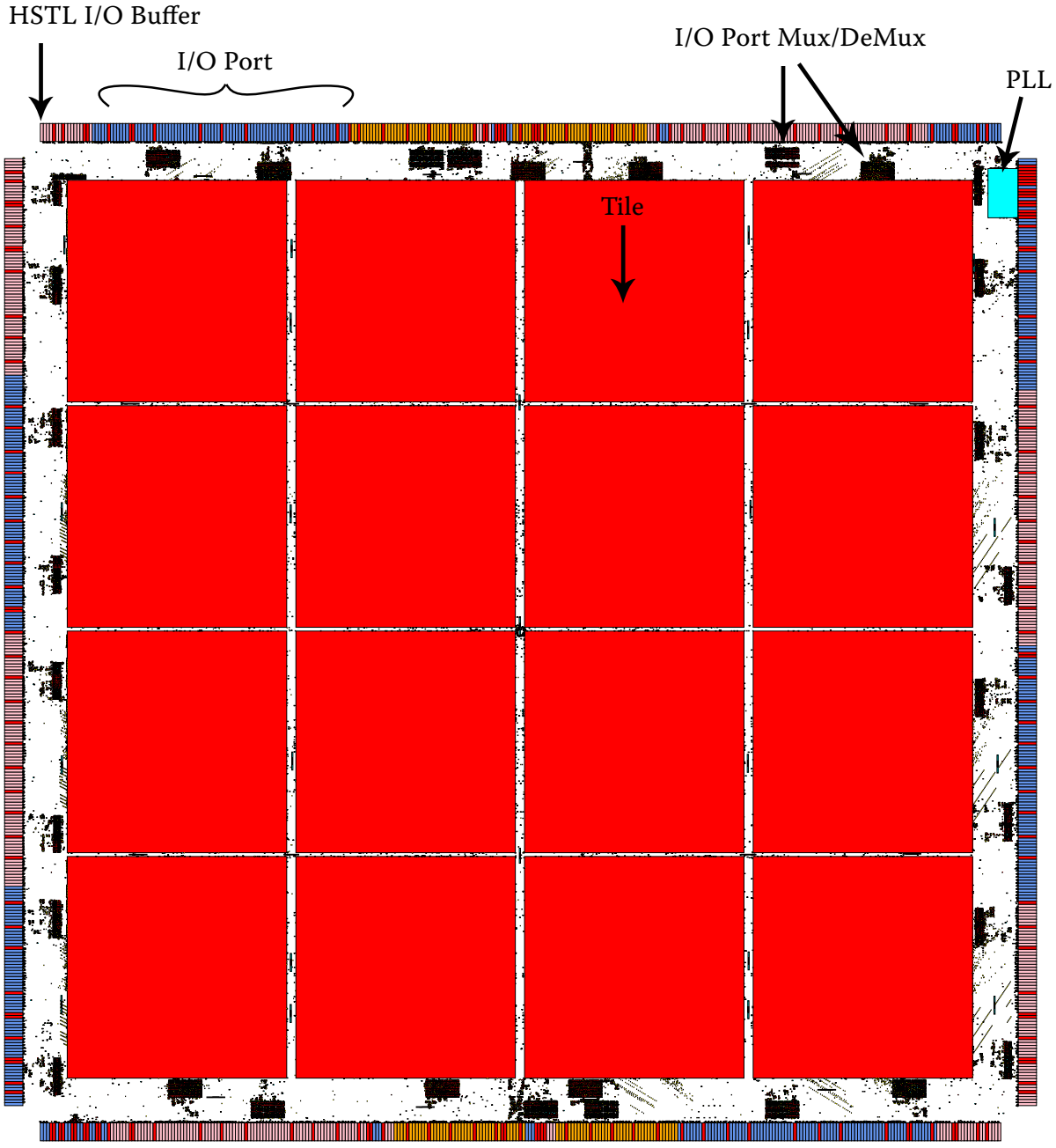


Figure 4-6: Computer-generated layout of Raw chip. The major components are 16 tiles, 16 I/O port multiplexers, 16 I/O port demultiplexers and a PLL. The remaining "dust" logic implements the clock tree, JTAG, LSSD scan and buffering of long wires.

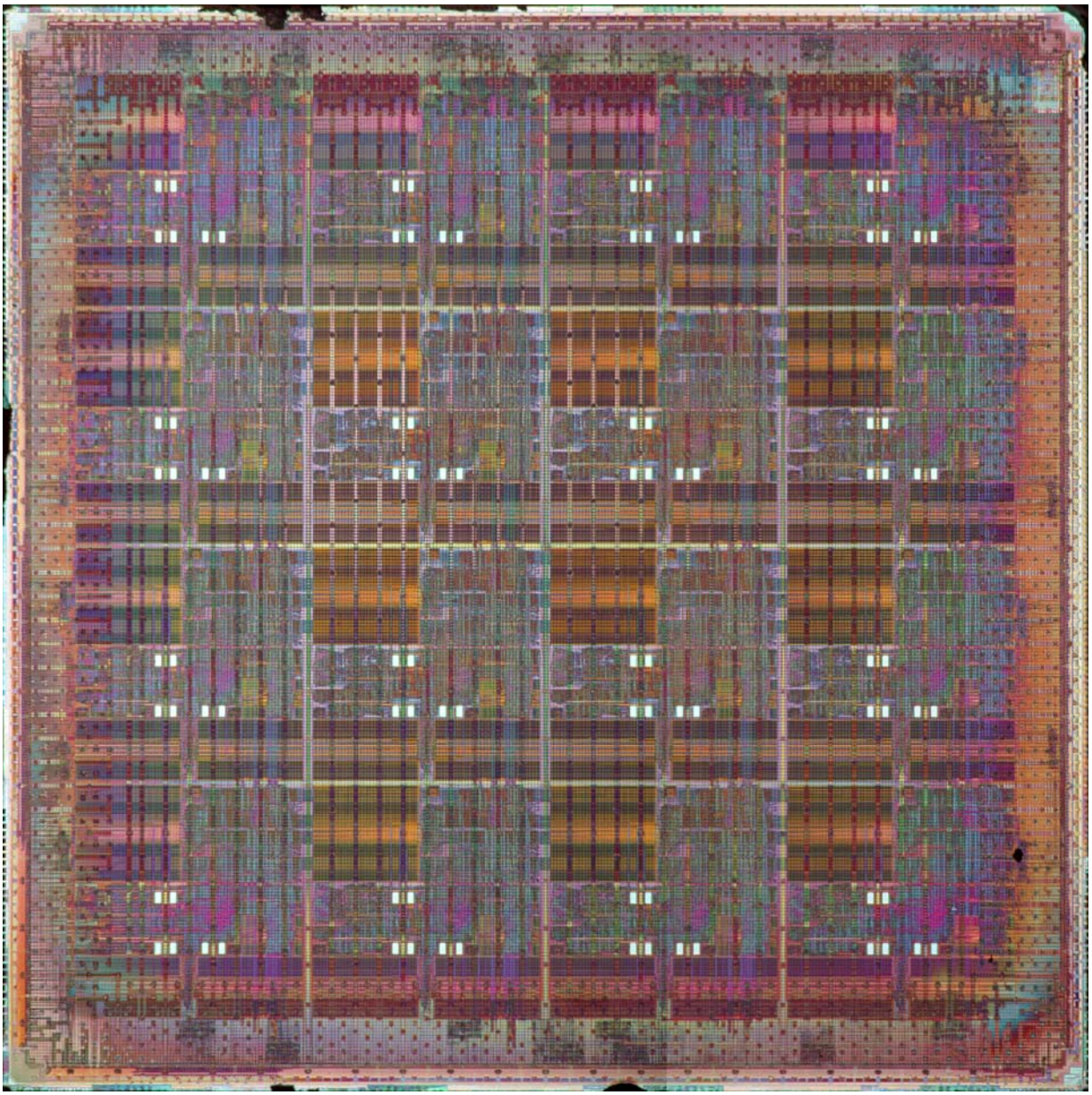


Figure 4-7: Photomicrograph of the 18.2 mm x 18.2 mm 16-tile Raw chip, after the top layers have been removed via a wet etch (i.e., acid.) The etch is somewhat imperfect and left some traces of the upper levels of metal (especially metal 6). The resolution of the apparatus (3000 pixels x 3000 pixels) at this magnification is approximately 11 wiring tracks per pixel. The 16 tiles are clearly visible. Around the periphery, the multiplexing logic for the I/O ports is also visible.

count comes from I/O port logic. Note that most of the perimeter around the tile array is empty. In retrospect, given that the die size could not be reduced because the design was pin- rather than gate-limited, it would have made sense to increase the size of the I/O port's FIFOs (from 4 elements) to reduce the burden on the I/O ports of the external chipset (implemented by FPGAs) to process credit signals at chip speed.

One surprise that arose out of the generation of the clock tree for Raw is that although the array of tiles lends itself to a structure clock, the I/O buffers around them also need a clock, and are not quite as regular.

After the chips returned from IBM's fab in October 2002, we received a few samples of unpackaged Raw die, which allowed us to produce die photomicrographs. We quickly found however that the resulting photos (such as in Figure 4-4) did not reveal much about the structure of the chip. As a result, we contracted a lab to remove successive layers of the chip via a wet etch and to photograph the die using a microscope and a 4x5 inch large format analog camera. The negative was then scanned using a film scanner to produce a digital image. This image is shown in Figure 4-7.

Pictured in Figure 4-9 is a computer generated layout of a single tile. Most of the Raw design was done hierarchically, i.e., only a single tile was coded, synthesized to gates and placed. Then, at final stages of the backend, wiring, and signal buffering passes were done "flat", treating each tile as an independent and non-identical entity. The final stages were done flat because this was the standard flow that IBM ASIC engineers were accustomed to. Not surprisingly, we found that the compute times and memory footprints of the flat runs were an order of magnitude larger than then the hierarchical ones, and since Raw was one of IBM's largest ASIC chips at the time, it strained the infrastructure. For larger tiled designs, it seems likely that tiled microprocessors, combined with a hierarchical backend flow, will produce great labor savings in the backend.

Source code statistics for the Raw design are shown in Figure 4-8. Generally, the quantities counted are lines of verilog. Overall, Raw was implemented with 33,981 lines of source code, plus 12,331 lines of TCL code. These counts do not include IBM-provided back-end TCL scripts, or IBM-provided structural verilog used to implemented most datapath operators such as adders. Overall, this is significantly less verilog than contemporary industrial designs. The Pentium 4 design, for instance, consisted of 1 million lines [12] – over 30x as many as the Raw design – even though the Pentium 4 chip has less than half the number of transistors (42 Million) as the Raw chip (over 100 Million). This is a testament to the significantly reduced design complexity and validation effort of tiled microprocessors relative to traditional superscalars. Raw was also designed with a small team of graduate students turned first-time microprocessor designers, significantly less than the Pentium 4's 500 engineers [12].

Basic gate type counts are shown in Table 4.3. Somewhat surprisingly, 60% of a Raw tile is flip-flops, muxes, or inverters. Approximately 24% of a tile's gate count is in FIFOs (network input

Component	Lines	Words	Characters	% (by words)
Main processor - Control	8998	28538	321422	28.50%
FPU	3104	11349	92806	11.33%
Static Network	3175	9433	108366	9.42%
Main processor - Datapath	2590	7731	94774	7.72%
Data Cache	1947	5954	69429	5.95%
I/O Ports	2039	5701	57262	5.69%
Dynamic Network	1693	5163	64131	5.16%
Test Network	703	1998	17463	2.00%
Integer Divider	457	1101	10700	1.10%
Datapath Module Wrappers	3526	11936	134185	11.92%
Top-level Glue (Signal Routing)	5749	11244	244708	11.23%
Total - Logic generation	33981	100148	1215246	100.00%
Placement Scripts (tcl)	12331	50279	386899	
Total	46312	150427	1602145	

Figure 4-8: Lines of source (generally, verilog) to describe Raw processor. Does not include testing code. The datapath was generally described with structural verilog, while control used higher-level RTL and synthesis. Wide-word Datapath operators (such as multipliers, adders and multiplexers) were instantiated from an IBM library, so although these items were specified at the gate-level (by IBM developers), they are not include in the counts. The entry *Datapath Module Wrappers* corresponds to the source code used to wrap these components for portability purposes. The entry *Top-level Glue* corresponds to the source code used to instantiate and connect the base components into tiles and then into an array of tiles and I/O ports.

Gate Type	Quantity	Percentage
muxes	16,094	24.41%
flip-flops	12,023	18.24%
inverters	11,796	17.90%
2-input nand, nor, xnor	11,277	17.11%
2-input and, or, xor	3,970	6.02%
buffers	2,702	4.10%
3-input and, nor, or, nand, xor	2,414	3.66%
clk-related	1,166	1.77%
4-input and, nand, nor, or	877	1.33%
other	3,591	5.45%
Total	65,911	100%

Table 4.3: Gate types in the Raw tile. $>$ 90% of the inverters are high-drive inverters, for driving long signals. The 26 NIBs (containing space for 128 elements) alone comprise 15,719 instances (23.85% of the gates), including 5,315 flip-flops, 5,519 2- or 3-input muxes, 2,156 inverters and 435 buffers.

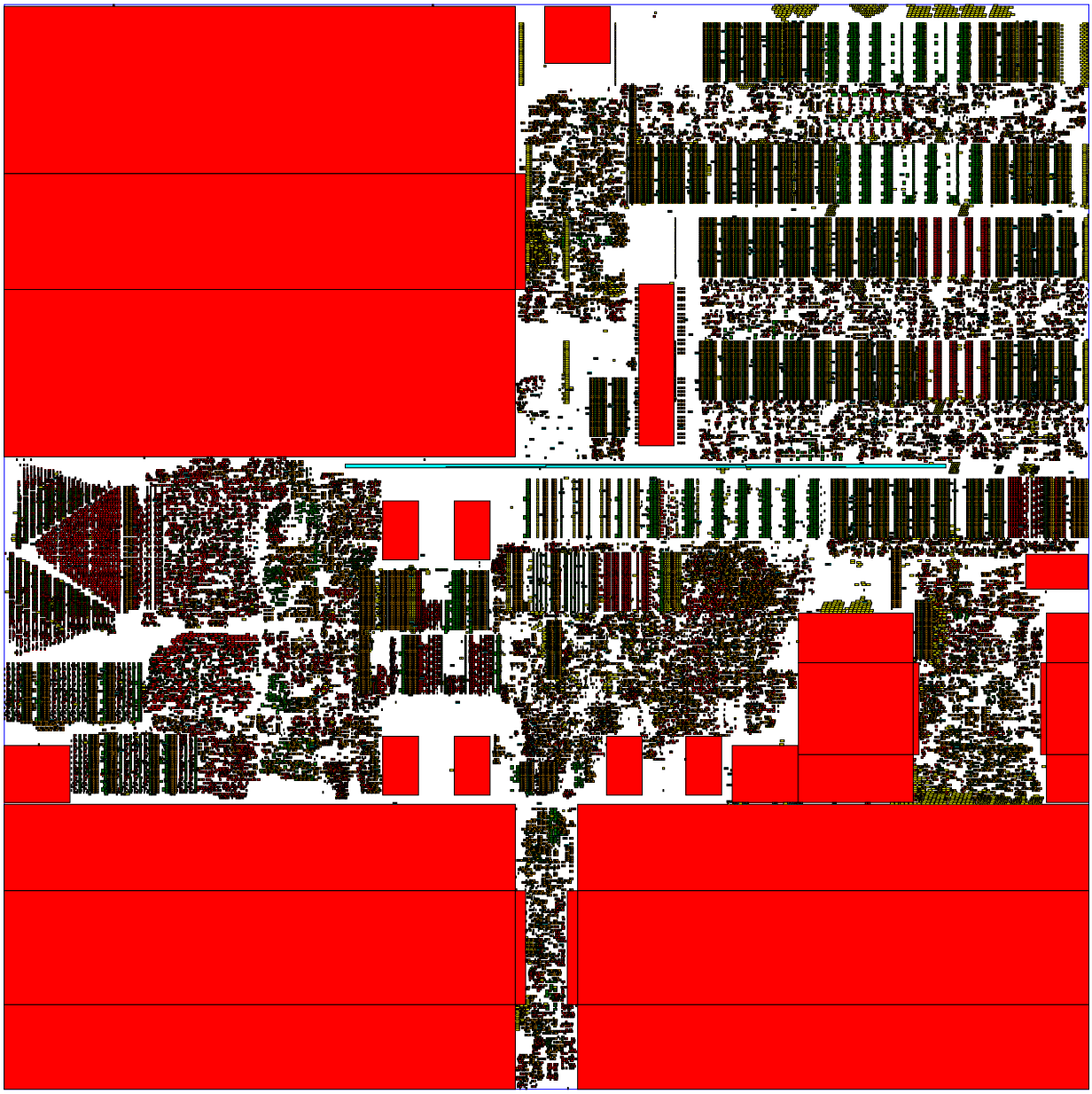


Figure 4-9: Computer-generated layout of 4 mm x 4 mm Raw tile. For color renderings, green cells are multiplexers, orange cells are registers, cyan cells are clock tree, and yellow cells are buffers. In the `.pdf` version of this thesis, the reader may zoom in up to 64x, examine individual cells, and print close-up views. In some cases it may be convenient to select "Print as Image".

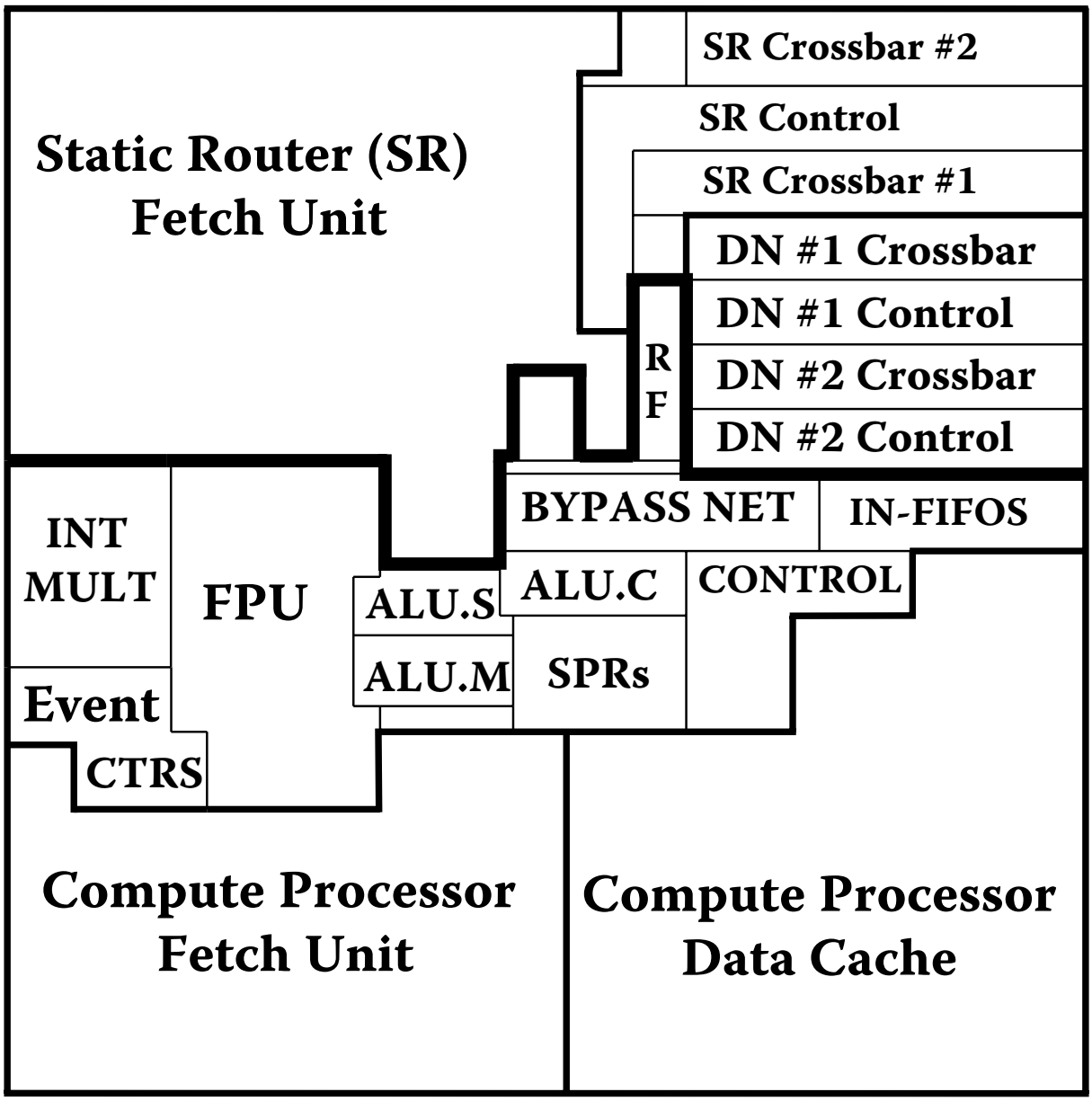


Figure 4-10: Floorplan of a Raw tile. The networks, including the static router ("SR") and dynamic networks ("DN"), comprise approximately 40% of the tile area. The register file ("RF") is quite small relative to conventional superscalar designs. ALU.S, ALU.C, and ALU.M implement the single-cycle instructions and are grouped according to criticality (C = critical, M = moderately critical, S = simple, not critical). The bypass network is approximately the same area as one of the network datapaths. SPRs is the unstructured logic that implements Raw's status and control registers. The Raw tile has 16 event counters (Event CTRS) for detecting and summarizing network and compute processor events.

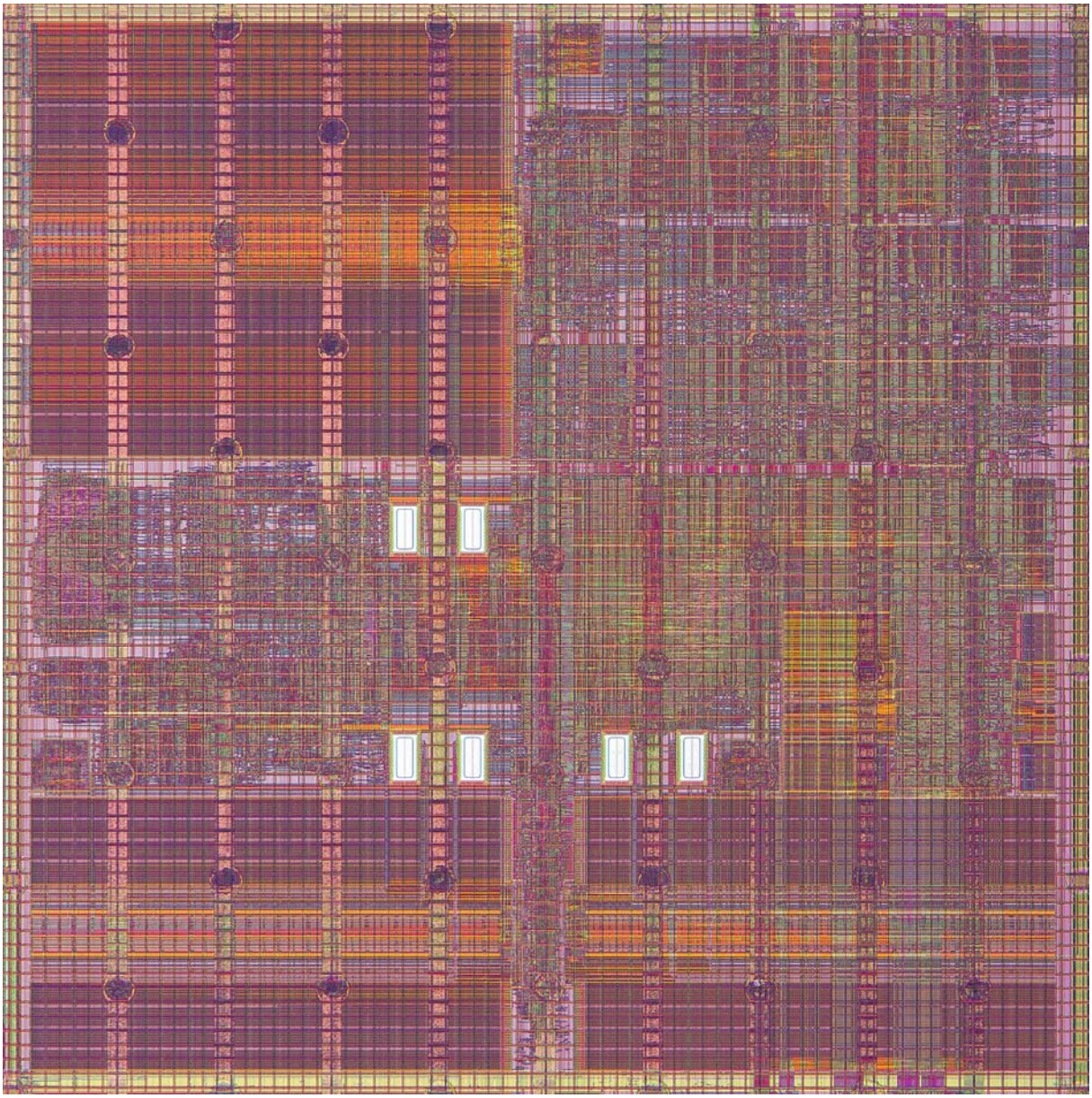


Figure 4-11: Photomicrograph of a 4 mm x 4 mm Raw tile. The top layers have been removed. The resolution of the apparatus (3000 pixel x 3000 pixel) at this magnification is approximately 1.3 microns per pixel, or around 2 wiring tracks per pixel. This is approaching, but not reaching the limits, imposed by the wavelength of light. Much of the detail of the tile is readily correlated against the tile floorplan diagram.

blocks) alone. Much of this overhead could be reduced if full-custom design were employed, or if the ASIC vendor provided custom FIFO macros.

Figure 4-10 shows a floorplan of the Raw tile, while Figure 4-11 shows a photomicrograph of a single tile. Both of these pictures can be readily correlated against the computer-generated layout in Figure 4-9. The Scalar Operand Networks (including the bypass networks) and Generalized Transport Networks (labeled "DN" for dynamic network) take up approximately 50% of the chip. Approximately 40% of the chip is 6-transistor SRAM. Generally, the placement of objects relative to the bypass network reflects their relative criticality. Blocks that are less critical may be placed further away. Thus, from the layout, we can see that the ALU.C ("critical ALU") and dynamic networks were the most critical. The 32-element, 2 read port, 1 write port register file of the system is a relatively small fraction of the total area[3]. Overall, the critical path was designed in to be the path through the fetch unit's SRAM, through a 2-input mux, and back into the SRAM – a typical path found in next-line predictors of superscalars. A higher frequency design would have to make this path take two cycles[4]. The FPU and Integer Multiplier ("INT-MULT") were non-critical because they were multicycle functional units and the wire delay was incorporated as part of the latency. Another indicator of relative criticality in terms of performance or density (or at perhaps research interest) is visible in Figure 4-9. Areas that appear regular are either SRAMS (the large red blocks) or were placed using a series of placement libraries that members of the Raw team wrote. In particular, the networks, ALU, and control logic for branch prediction and static router stall determination were placed in a data-path style. The IBM CAD flow's automatic placer was congestion-driven rather than timing-driven, which made this work necessary to meet timing. It is quite possible that more recent placement tools are now sufficient for the job, given reasonable floorplanning. Based on our flow, a synthesized and placed Raw chip could be automatically generated from source Verilog in 6 hours: 4 hours of synthesis on a Pentium-4 2 GHz, and 2 hours of placement (via IBM's ChipBench, which did not run on x86) on a 750 MHz UlraSparc IV.

4.3 The Raw Systems

The Raw chip was designed to be used in conjunction with two types of systems. The first system employs a single Raw chip and a custom-built workstation-style motherboard (the "Raw motherboard"). The other system integrates multiple Raw chips onto a single board (the Raw "array board"), which can be connected to other array boards to form supercomputer-style Raw systems with up to 1024 tiles.

Figure 4-12 shows a picture of the Raw motherboard. In the center of the board is the Raw

[3]One of the P4 architects, upon inspecting the Raw tile, asked surprised, "Where is the register file?!" – he had forgetten that since Raw wasn't wide issue, it would be tiny.

[4]A surprising result, as at least some high-frequency superscalars appear to be able to do this in one, indicating full-custom SRAMs (as measured in FO4) are faster.

chip. It is a surrounded by a number of FPGAs which implement the chipset, the glue-logic that interprets messages coming off the on-chip networks over the pins and converts them into control signals for peripherals. The motherboard has a number of standard SDRAM slots, PCI slots, a USB card, and various other peripherals[5]. The system is standalone; however, we typically attach a PC to the USB cable, which runs code to simulate I/O devices, or to interface to framebuffer boards for input or output video streams[6]. We ran a number of applications on the system, including real-time video applications including audio beamforming on a 1020-microphone array, and blue-screen video substitution.

As shown in Figure 4-13, the system fits in a standard PC case and runs off of a standard PC power supply. A heat-sink is necessary as the chip consumes approximately 17 W during average load (at 425 MHz) and up to 38 W with pathological programs [62]. Approximately 10 W of power is burned in the clock alone. We spent very little effort on optimizing power in the Raw system because the IBM tools generally did not support it, and because we were already within our thermal envelope, 80 W. Nonetheless, tiled microprocessor designs are easy to optimize for power, as detailed in [62].

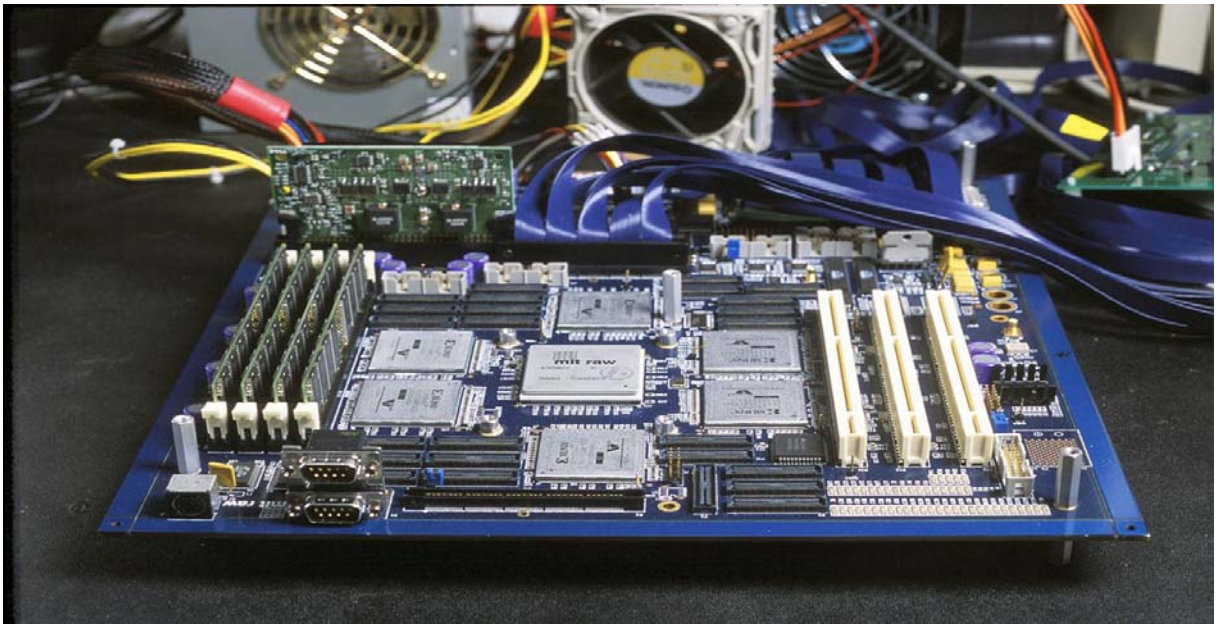


Figure 4-12: The Raw Motherboard.

Figure 4-14 depicts the Raw array board. The array board leverages the Raw chip's ability to gluelessly interconnect with other Raw chips. When the chips are connected, they act as a larger

[5]When the USB board is plugged into a windows machine, it says "Windows has detected a Raw Architecture Workstation"!

[6]In general, the use of PCI cards in the Raw system was difficult because of the lack of open standards and the cards' general use of embedded x86 BIOS code to initialize undocumented I/O registers. Finally, the latest generation of FPGA device does not allow 5 V signaling, which further reduced the selection of candidate cards.

Figure 4-13: The Raw Workstation.

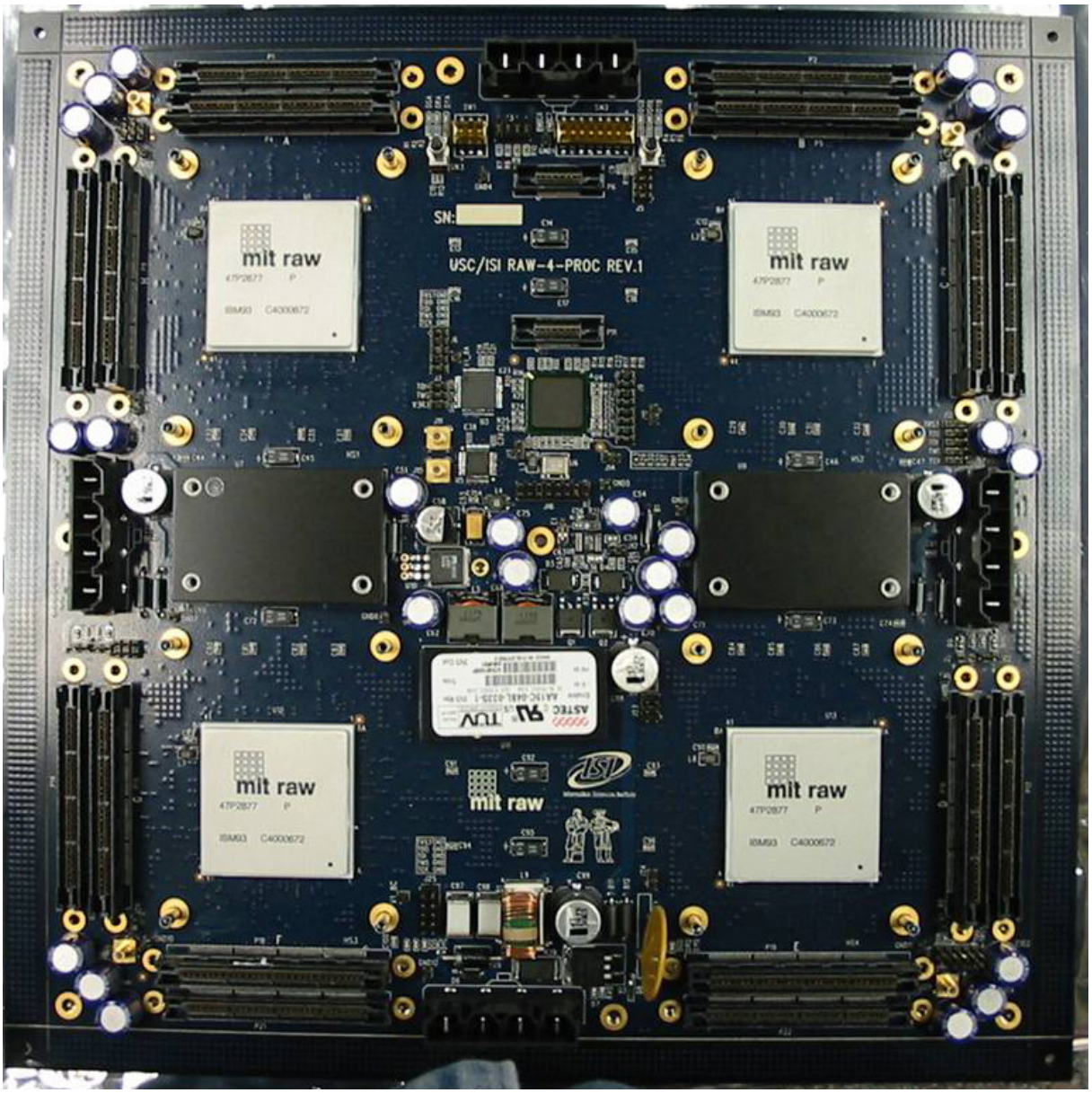


Figure 4-14: The Raw Array Board

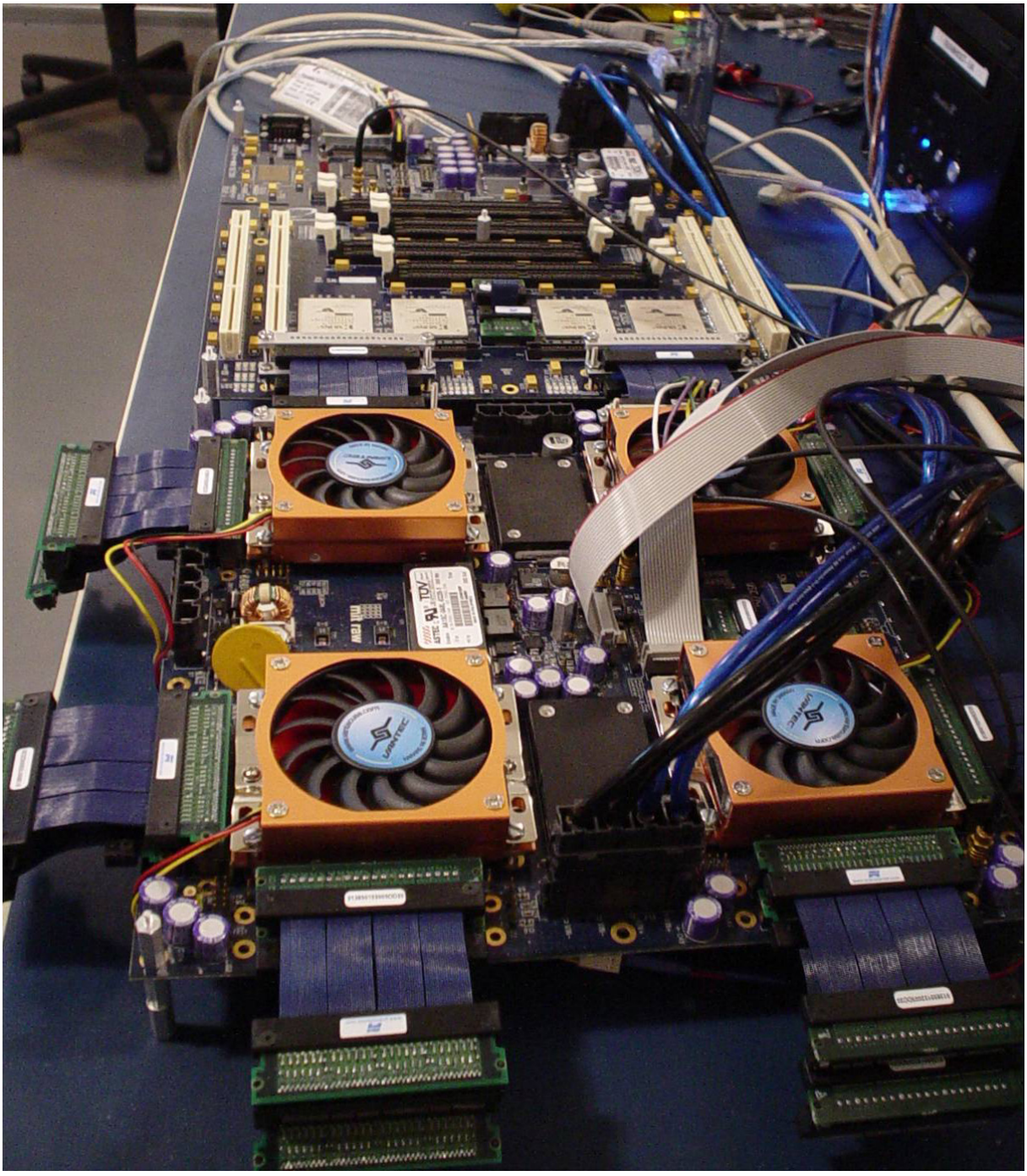


Figure 4-15: The Raw Array Board, connected to an I/O board for DRAM and PCI support. Visible in the picture are the fansinks for cooling the Raw chips, and a number of high-speed cables, which are used to connect the array boards to other array boards and I/O boards.

virtual Raw chip – with one exception: the second static network does not cross chip boundaries. Thus, a single array board implements a 64 tile system. The boards themselves can be tiled using high-speed ribbon cables to create systems of up to 1024 tiles[7]. In addition to the array board, the Raw group has created I/O boards, which support DRAM and PCI devices. Figure 4-15 shows the Raw array board connected to one such I/O board. Collectively, a 1024 tile system would with little doubt make Raw the widest-issue ILP-based microprocessor ever built.

Voltage (V) \ Frequency (MHz)	420	430	440	450	460	470	480	490	500
2.20	OK	OK	OK	OK	OK	OK	OK	OK	OK
2.15	OK	OK	OK	OK	OK	OK	OK	OK	FAIL
2.10	OK	OK	OK	OK	OK	OK	OK	FAIL	FAIL
2.05	OK	OK	OK	OK	OK	OK	FAIL	FAIL	FAIL
2.00	OK	OK	OK	OK	OK	FAIL	FAIL	FAIL	FAIL
1.95	OK	OK	OK	OK	FAIL	FAIL	FAIL	FAIL	FAIL
1.90	OK	OK	OK	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
1.85	OK	OK	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
1.80	OK	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL

Figure 4-16: Shmoo plot of 16-tile Raw core. Plot shows frequency-voltage relationship during core operation at an ambient temperature of 20° C. The Raw core operates without error at 420 MHz at the nominal voltage of the IBM process, 1.8 Volts, and as high as 500 MHz at 2.2 Volts.

Figure 4-16 depicts a Shmoo plot, which shows the relationship between voltage and maximum possible frequency of operation of the Raw chip. The chip reaches 425 MHz at 1.8 V and 500 MHz at 2.2 V[8].

4.4 Conclusions from Building the Raw System

Design, implementing and booting the Raw system was a rewarding experience. Figure 4-17 shows the timeline of the effort. On the right are given design milestones and dates, while on the right are given verification milestones. Much of the time between the first behavioural model and the final netlist was bottlenecked by our knowledge of the tools. Our verification effort was extensive and

[7]The principal impediments to scalability in the system beyond 1024 tiles are the use of ten-bit addresses for dynamic network packets, the use of 32-bit address (limiting the system to 4GB of address space), and increasing ratio of tiles to I/O ports.

[8]The estimate lifetime of a Raw chip is approximately 6 months at this voltage. Other academic groups [66, 93] have proven more daring at increasing voltages to higher levels; however because the Raw chips are attached with solder columns, rather than with sockets, replacing a chip is a more costly proposition.

consisted of many months of simulation time and hundreds of thousands of lines of code, including randomly generated tests. Our bug-finding and bug-fixing efforts are detailed in Figure 4-18. In all, we found less than 150 bugs before tapeout, significantly less than the 7,855 bugs found in the P4 before it taped out [11]. After tapeout, we found three bugs, none of them show-stoppers[9], in contrast to the P4's 101 errata in the released product. Although the P4 is a commercial system and has some necessary complexity over the larger Raw design, the vast difference in lines of verilog (30x), design team size (∼100x), pre-tapeout bugs (50x), and post-tapeout bugs (33x) is consistent with our claim that tiled microprocessors are easy to design, build, and verify.

[9]For instance, our first bug resulted from a test mode ("extra power burn mode") which disables gating of functional units so that we can measure the power impact. We left this untested and only discovered the night of a paper deadline that in this mode, during a cache miss, instructions would continue to march down the pipeline and fall off the end!

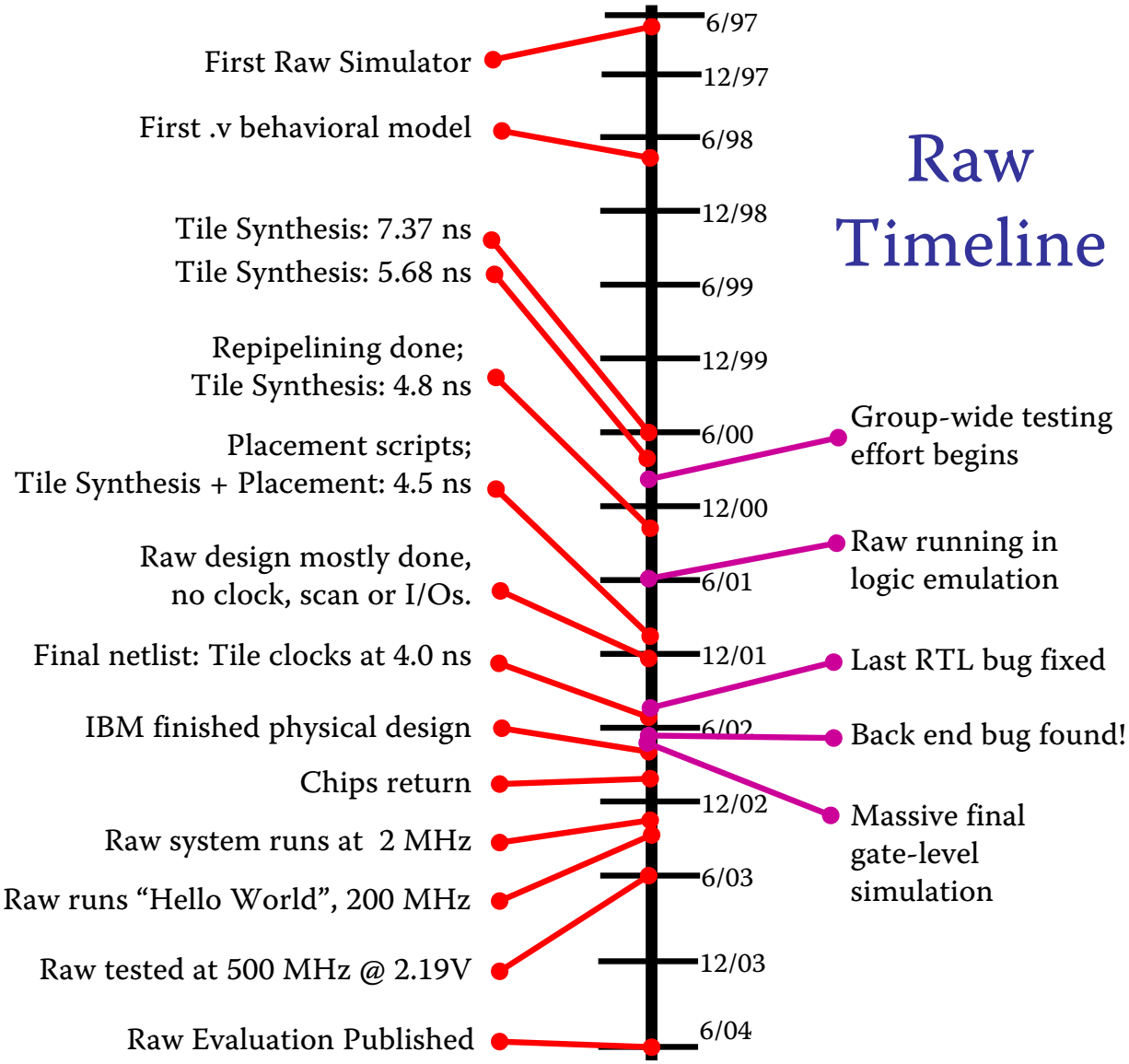


Figure 4-17: Raw timeline

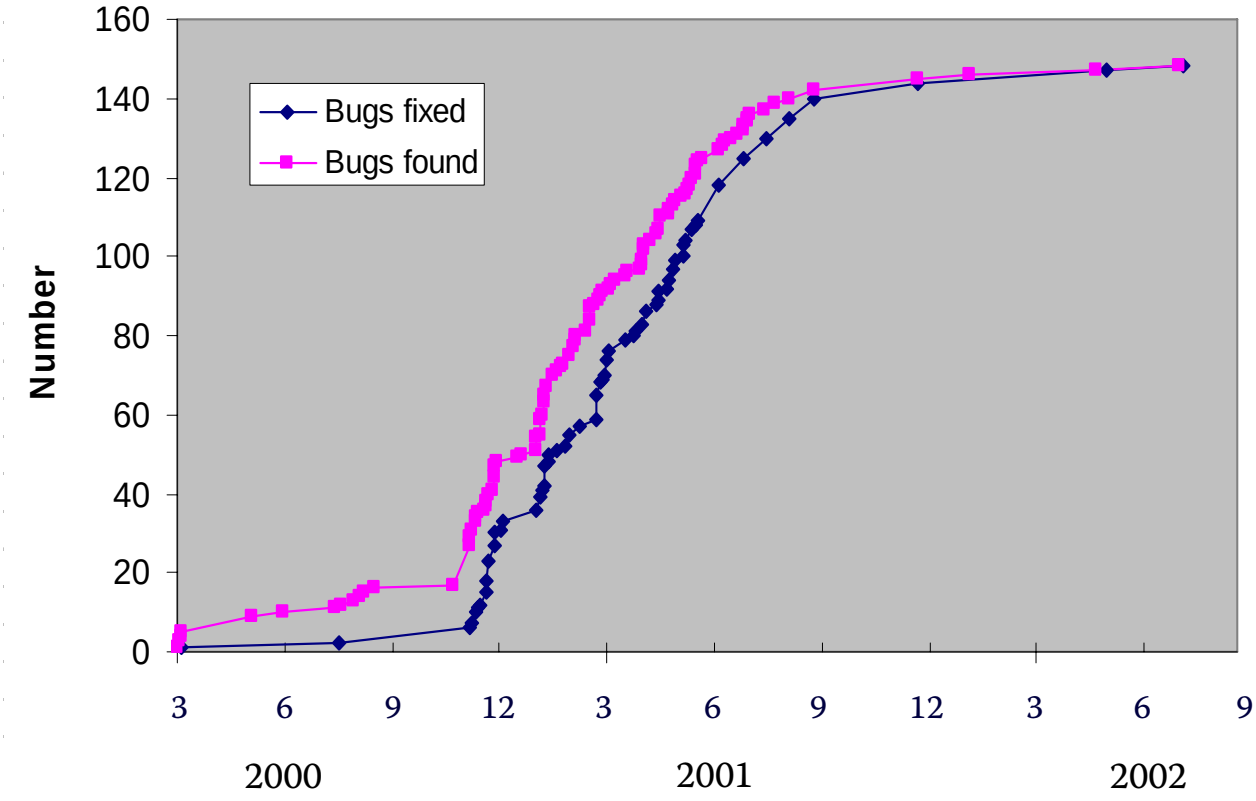


Figure 4-18: Bug finding and fixing timeline.

Chapter 5

Performance Evaluation

This chapter evaluates the tiled microprocessor design paradigm by analyzing the Raw microprocessor. Sections 5.1 introduces the methodology that we use to evaluate the performance of the Raw design. Our methodology compares the Raw system to the Intel Pentium 3, which was implemented in the same (180 nm) process generation. In general, we organize the results according to the compilation methodology, and describe the compilers, algorithms or languages that were employed along with the performance results.

The results are given in the following order. We first examine Raw's single tile performance on conventional desktop applications in Section 5.2. This establishes the baseline performance of a single tile versus a conventional superscalar like the Pentium 3. We continue by examining Raw's performance as a server-farm on a chip, using the same set of benchmarks. Then, in Section 5.4, we evaluate Raw's performance using multiple tiles to execute automatically-parallelized sequential C and Fortrain programs. Section 5.5 explores Raw's performance on multiple tiles using StreamIt, an parallel language for expressing computations. Section 5.6 examines Raw's performance on hand-coded streaming applications. Finally, in a what-if exercise, Section 5.7 examines Raw's performance on sequential single tile applications if it had been implemented with 2-way issue tiles. Section 5.8 concludes.

5.1 Evaluation Methodology

The following two sections examine the methodology that this dissertation uses to evaluate the performance of the Raw microprocessor. Section 5.1.1 explains the two major options in evaluation metholodogy, and our reasons for choosing the second option, a comparison with a third-party system. Section 5.1.2 examines our decision to compare against the Intel Pentium 3 processor. Section 5.1.3 describes how this methodology was put into practice.

5.1.1 Challenges in Evaluation Methodology

Self-speedups vs. Comparison against 3rd-party System One of the great challenges in microprocessor design research is the issue of performance evaluation. The traditional methodology in parallel processing is to express the performance of a system in terms of "self-speedups" – that is, by dividing the runtime of the system in a baseline configuration (for instance, on one node) by the run-time of the system in the proposed configuration (for instance, on 16 nodes.) This is by far the easiest methodology, because it carries the advantage that it does not require examination of other reference systems unrelated to one's own system.

Unfortunately, this advantage is also one of the major limitations of the approach - self-speedups do not show how the approach compares to other approaches. Self-speedups do not expose performance problems that may be systematic of the research and common to both research configurations. Worse, self-speedups lead to moral hazard - they penalize the system implementor for performing due diligence. For instance, many compiler optimizations act to reduce the effective amount of work that needs to be performed (such as common sub-expression elimination and register allocation.) Unfortunately, reducing the effective amount of work also reduces the opportunities for self-speedup. Thus, researchers are put in the unenviable position where the most moral action will entail additional work and worsen their performance numbers.

An alternative choice is to compare against existing systems implemented by 3rd parties that have an interest in creating the best system. In this case, inherent inefficencies in the proposed design will be exposed. This approach is, however, more difficult to carry out. First, there is the challenge that the systems will have non-identical aspects (for instance, differing VLSI processes and logic families) that will require normalization. Second, it places a greater burden on the completeness of the experimental prototype. Non-research aspects of the system may need complete industrial-strength implementations, even though the implementations themselves are not part of the research agenda. For instance, FPU architecture is not a central component of this dissertation's focus; however an inefficient FPU would have significantly hurt the outcome of performance results in the system, and could have obscured otherwise positive findings of the research.

This dissertation takes the 3rd-party-comparison approach, despite the inherent difficulty in doing so. The comparison system, the Intel Pentium III ("P3"), was selected for its wide-spread availability, and for the relative closeness of its implementation parameters to the Raw design. Although in an ideal world, the P3 and Raw implementation parameters could be closer, this approach exposes the research "to the open light" better than self-speedup numbers alone. Furthermore, for those who feel comparison with another system is more appropriate, it gives them the means to do so: they can compare the run-times on another system to the P3 and by extension, to Raw.

5.1.2 Developing a Comparison with the Pentium III

In order to implement the methodology of comparing Raw to a third party system, two key steps were taken. First, a comparison system, the P3, was selected based on an evaluation of its implementation parameters. Second, the Raw and P3 systems were configured so as to match as much as possible. The evaluation further employs BTL, the Raw cycle accurate simulator, in order to perform normalizations that would not be possible in hardware.

Selection of a Reference Processor The selection of a comparison (or reference) third-party system led us to reflect upon the properties that such a comparison system would have. For fairness, the comparison system must be implemented in a process that uses the same lithography generation, 180 nm. Furthermore, the reference microprocessor needs to be measured at a similar point in its lifecycle, i.e., as close to first silicon as possible. This is because most commercial systems are speedpath or process tuned after first silicon is created [17]. For instance, the 180nm P3 initial production silicon was released at 500-733 MHz and gradually was tuned until it reached a final production silicon frequency of 1 GHz. The first silicon value for the P3 is not publicly known. However, the frequencies of first-silicon and initial production silicon have been known to differ by as much as 2x.

The P3 is especially ideal for comparison with Raw because it is in common use, because its fabrication process is well documented, and because the common-case functional unit latencies are almost identical. The back ends of the processors share a similar level of pipelining, which means that relative cycle-counts carry some significance. Conventional VLSI wisdom suggests that, when normalized for process, Raw's single-ported L1 data cache should have approximately the same area and delay as the P3's two-ported L1 data cache of half the size. For sequential codes with working sets that fit in the L1 caches, the cycle counts should be quite similar. And given that the fortunes of Intel have rested (and continue to rest, with the Pentium-M reincarnation) upon this architecture for almost ten years, there is reason to believe that the implementation is reasonable. In fact, the P3, upon release in 4Q'99, had the highest SpecInt95 value of any processor [41].

Itanium and Pentium 4 (P4) came as close seconds in the final choice. The selection of the P3 over these possibilities stemmed from the need to match the lifecycle of the reference system to Raw's. Intel's market pressures cause it to delay the release of new processors such as P4 or Itanium until they have been tuned enough to compete with the existing Pentium product line. Consequently, when these processors are released, they may be closer to final-silicon than first-silicon. For example, it is documented in the press that Itanium I was delayed for two years between first-silicon announcement and initial production silicon availability. Finally, Raw's implementation complexity is more similar to that of the P3 than to that of the P4 or Itanium.

5.1.2.1 Comparing Raw and P3 Coppermine VLSI Characteristics

Table 5.1 compares the two chips and their fabrication processes, IBM's CMOS 7SF [77, 94] and Intel's P858 [125]. CMOS 7SF has denser SRAM cells and less interconnect resistivity, due to copper metalization. P858, on the other hand, compensates for aluminum metalization by using a lower-k dielectric, SiOF, and by increasing the dimensions of wires at highers level of metal.

Parameter	**Raw (IBM ASIC)**	**P3 (Intel)**
Lithography Generation	180 nm	180 nm
Process Name	CMOS 7SF (SA-27E)	P858
Metal Layers	Cu 6	Al 6
Dielectric Material	SiO_2	SiOF
Oxide Thickness (T_{ox})	3.5 nm	3.0 nm
SRAM Cell Size	4.8 μm^2	5.6 μm^2
Dielectric k	4.1	3.55
Ring Oscillator Stage (FO1)	23 ps	11 ps
Dynamic Logic, Custom Macros (SRAMs, RFs)	no	yes
Speedpath Tuning since First Silicon	no	yes
Initial Frequency	425 MHz	500-733 MHz
Die Area[1]	331 mm^2	106 mm^2
Signal Pins	$\sim$ 1100	$\sim$ 190
Vdd used	1.8 V	1.65 V
Nominal Process Vdd	1.8 V	1.5 V

Table 5.1: Comparison of Implementation Parameters for Raw and P3-Coppermine.

The Ring Oscillator metric measures the delay of a fanout-of-1 (FO1) inverter. It has been suggested that an approximate FO4 delay can be found by multiplying the FO1 delay by 3 [46]. Thus, P858 gates appear to be significantly (2.1x) faster than the CMOS 7SF gates. This is to be expected, as IBM terms CMOS 7SF a "value" process. IBM's non-ASIC, high-performance, 180 nm process, CMOS 8S, is competitive with P858 [27], and has ring oscillator delays of 11 ps and better. Furthermore, production 180 nm P3's have their voltages set 10% higher than the nominal process voltage, which typically improves frequency by 10% or more. Overall, the P858 is a significantly higher-performance process than CMOS 7SF, with speed advantages for both transistors and long-haul wires. However, CMOS 7SF has some advantages in terms of density.

A recent book, [17], lists a number of limitations that ASIC processor implementations face versus full-custom implementations. We mention some applicable ones here. First, because the ASIC flow predetermines aspects of a chip, basic overheads are relatively high in comparison to full-custom designs. Two of Raw's largest overheads were the mandatory scan flip-flops (18%), and clock skew and jitter (13%). Second, ASIC flows tend to produce logic that is significantly less dense

[1]Note that despite the area penalty for an ASIC implementation, it is almost certain that the Raw processor is a bigger design than the P3. Our evaluation does not aim to make a cost-normalized comparison, but rather seeks to demonstrate the scalability of our approach for future microprocessor designs.

Operation	Latency		Occupancy	
	1 Raw Tile	P3	1 Raw Tile	P3
ALU	1	1	1	1
Load (hit)	3	3	1	1
Store (hit)	-	-	1	1
FP Add	4	3	1	1
FP Mul	4	5	1	2
Mul	2	4	1	1
Div	42	26	42	26
FP Div	10	18	10	18
SSE FP 4-Add	-	4	-	2
SSE FP 4-Mul	-	5	-	2
SSE FP 4-Div	-	36	-	36

Table 5.2: Raw versus P3 Functional unit timings. Commonly executed instructions appear first. FP operations are single precision.

than corresponding custom flows. Third, ASIC flows prevent use of custom or dynamic logic, except for a limited menu (up to 2 read ports and 2 write ports) of fixed pipeline-depth register files and SRAMs, which are machine-generated. A 40-80% improvement in frequency often is attributed to the use of dynamic logic. Process and speedpath tuning account for 35%. Finally, speed-binning yields approximately 20%.

In order to compensates for the last two factors, we selected as reference processor the 600 MHz P3, which was released prior to process tuning, and after limited speedpath tuning. Thus, it is solidly in the middle of the P3 initial production frequency range, presumably representing an average-speedbin part.

A Raw implementation with the same engineering effort and process technology as the Intel P3 would be smaller and significantly faster. Because the results are typically favorable towards Raw, we do not generally try to adjust the Raw results to normalize for the Intel P3's advantages. In some cases, however, we show the simulation results for Raw running at the higher frequency of the P3 to show the relative impact of greater cache miss times.

5.1.2.2 Comparing Raw and P3 Microarchitectural Parameters

One of the reasons that we selected the P3 as the comparison processor is because the functional unit latencies and occupancies were quite similar to those found in Raw. These latencies and occupancies are shown in Table 5.2. Most operations except divides are pipelined on both processors. Unique to P3 are the SIMD instructions ("SSE") that execute four single-precision floating point operations every two cycles. Whenever possible, we enable the use of SSE in the P3 benchmarks that are run. The closeness of these latencies suggests a similar level of aggressiveness applied to pipelining the two chips. Of course, the P3 has more total pipeline stages due to the extra logic levels incurred by implementing out-of-order execution and implementing a more complex instruction set. These

additional front-end pipeline stages are evident in the substantially larger misprediction penalty (10 to 15 cycles average case) of the P3. This larger misprediction penalty creates the need for the P3's 512-entry BTB and dynamic branch prediction hardware, in contrast to Raw's small and simple static branch predictor.

The difference in functional unit latencies was one reason why the P4 was not selected as a comparison system. The P4 is much more aggressively pipelined and has greater functional unit latencies, despite the use of advanced dynamic logic circuit techniques to reduce datapath circuit delay.

Table 5.3 compares the salient execution and memory system parameters of the two systems. The Raw system runs at a lower frequency, due to slower transistors, less aggressive static-logic circuit implementation technology, and lack of speedpath tuning. The P3 manifests its microarchitectural approach to microprocessor scalability via its out-of-order 3-operation-per-cycle implementation. The Raw tile on the other hand, sticks to a more spartan single-issue in-order pipeline.

The L1 caches of the system are comparable, with the P3 opting for more ports (via banking) and the Raw system opting for more capacity. Surprisingly, examination of the two processors' die-photos indicates that the P3's L1 SRAM cache banks are almost twice as large as the Raw Tile's L1 data cache. Anecdotal evidence suggests that the L1 cache was ultimately a major speed-path limiter in the P3 design.

The P3 holds over the Raw tile the advantage of having a 7-cycle latency 256 KB L2 cache. A single Raw tile has the edge over the P3 for working sets between 16 and 32 KB, while the P3 has the edge for between 32 KB and 256 KB. Of course, when multiple tiles work in concert, their collective L1 capacity can be much larger.

As will be seen in the next section, the DRAM timing parameters of the Raw system have been normalized through the BTL simulation to match those of the Dell P3 system in terms of time.

5.1.3 Normalization with the P3

5.1.3.1 Validated, Cycle Accurate Simulator

The evaluation in this chapter uses the BTL validated cycle-accurate simulator of the Raw chip. Using the validated simulator as opposed to actual hardware allows us to better normalize differences with a reference system, e.g., DRAM memory latency, and instruction cache configuration. It also allows us to explore alternative motherboard configurations. We verified that the simulator and the gate-level RTL netlist have *exactly* the same timing and data values for all 200,000 lines of our hand-written assembly test suite, as well as for a number of C applications and randomly generated tests. Every stall signal, register file write, SRAM write, on-chip network wire, cache state machine transition, interrupt signal, and chip signal pin matches in value on every cycle between the two.

	1 Raw Tile	P3
CPU Frequency	425 MHz	600 MHz
Sustained Issue Width	1 in-order	3 out-of-order
Mispredict Penalty	3	10-15 ave.
Branch Predictor	Static	512-entry BTB
DRAM Freq (RawPC)	100 MHz	100 MHz
DRAM Freq (RawStreams)	2 x 213 MHz	-
DRAM Access Width	8 bytes	8 bytes
L1 D cache size	32K	16K
L1 D cache ports	1	(1 Load, 1 Store) 2
L1 I cache size	32K	16K
L1 miss latency	54 cycles	7 cycles
L1 fill width	4 bytes	32 bytes
L1 / L2 line sizes	32 bytes	32 bytes
L1 associativities	2-way	4-way
L2 size	-	256K
L2 associativity	-	8-way
L2 miss latency	-	79 cycles
L2 fill width	-	8 bytes

Table 5.3: Comparison of execution resources in a Raw tile and the P3.

This gate-level RTL netlist was then shipped to IBM for manufacturing. Upon receipt of the chip, we compared a subset of the tests on the actual hardware to verify that the chip was manufactured according to spec.

5.1.3.2 Instruction Caching Normalization

We observed that Raw's software-managed instruction-caching system introduced significant differences between the two systems. To enable comparisons with the P3, the cycle-accurate simulator was augmented so that compute and switch processors employed conventional 2-way associative hardware instruction caches. These instruction caches are modeled cycle-by-cycle in the same manner as the rest of the hardware. Like the data caches, they service misses over the memory dynamic network. Resource contention between the caches is modeled accordingly. Of all of the differences in the system, this was the one which mostly greatly motivated the use of the validate cycle-accurate simulator for the evaluation study.

5.1.3.3 Motherboard Normalization

With the selection of a reference CPU implementation comes a selection of an enclosing computer. We used a pair of 600 MHz Dell Precision 410's to run our reference benchmarks. We outfitted these machines with identical 100 MHz 2-2-2 PC100 256 MB DRAMs, and wrote several microbenchmarks to verify that the memory system timings matched.

Although the Raw system has a complete motherboard implementation, that motherboard's

chipset is implemented using FPGAs to reduce cost in the system. The Dell motherboard chipset, on the other hand, is implemented using custom ASICs. As a result, the memory access times on the Dell motherboard are faster than the Raw motherboard. Although this difference is not enormous, we decided that it was relatively easy to normalize. To compare the Raw and Dell systems more equally, we used the Raw simulator's extension language to implement a cycle-matched PC100 DRAM model and a chipset[3]. This model has the same wall-clock latency and bandwidth as the Dell 410. However, since Raw runs at a slower frequency than the P3, the latency, measured in cycles, is less. We use the term **RawPC** to describe a simulation which uses 8 PC100 DRAMs, occupying 4 ports on the left hand side of the chip, and 4 on the right hand side.

Because Raw is also designed for streaming applications, it was desirable to examine applications that use the full pin bandwidth of the chip. In this case, the results employ a simulation of CL2 PC 3500 DDR DRAM, which provides enough bandwidth to saturate both directions of a Raw port. This configuration includes 16 PC 3500 DRAMs, attached to all 16 logical ports on the chip, in conjunction with a memory controller, implemented in the chipset, that supports a number of stream requests. A Raw tile can send a message over the general dynamic network to the chipset to initiate large bulk transfers from the DRAMs into and out of the static network. Simple interleaving and striding is supported, subject to the underlying access and timing constraints of the DRAM. We call this configuration **RawStreams**.

The placement of a DRAM on a Raw port does not exclude the use of other devices on that port – the chipsets have a simple demultiplexing mechanism that allows multiple devices to connect to a single port.

5.1.3.4 Operating System and I/O Hardware Normalization

One of the sources of deviation between two systems is the operating system and underlying I/O hardware. To normalize these factors, two approaches were taken.

C standard I/O For C or Fortran stdio call, the same version of the C standard library routines, newlib 1.9.0, was employed for Raw and P3. Furthermore, to eliminate the impact of differing file systems and operating system layers, the results of I/O system calls for Spec benchmarks were captured and embedded into the binaries as static data using a tool that I developed called the Deionizer [109]. In this system, two runs of the application are performed. In the first run, the I/O calls are logged into a file. Then, a new version of the application is created, in which the POSIX system calls like `open`, `read`, `write`, and `close` are replaced with a customized version that has the "answers" to the particular I/O calls embedded as static data in the binary. When this application is run, no I/O calls are actually performed, just accesses to DRAM. This substitution eliminates an important difference between the Raw and P3 systems.

[3]The support chips typically used to interface a processor to its memory system and I/O peripherals.

Streaming I/O In other cases, the applications model a series of streams coming in from a data source, such as a video camera. On Raw, these streams arrive via an I/O port, while in the P3, the conventional I/O mechanism is via direct memory access (DMA) through DRAM. To compare these systems, we place the input data set in DRAM before running the benchmark. Thus, the P3 results are in a sense optimistic for this case, because no DRAM contention or I/O system overhead is simulated.

5.1.3.5 Compiler and Custom Library Normalization

Another source of variation between the two systems are the compiler and customized libraries. The P3 system has more mature compilers and libraries than the Raw system. On the other hand, many of our applications have been optimized by Raw graduate students.

For standard C and Fortran single-tile programs, we normalize by using the same compiler and standard libraries. In this case, C and Fortran code were compiled with gcc 3.3 -O3 for both Raw[4] and the P3[5]. We chose against employing the Intel C compiler for this set of comparisons because its performance on the Spec benchmark suite is not representative of typical performance [34]. The Intel compiler is heavily tuned for Spec, due to the commercial importance of Spec numbers.

In cases where more customized code was run on Raw, we sought out the most optimized versions available for the P3. For example, we used the tuned ATLAS [124] SSE-based implementation, and custom convolution codes from the Intel Integrated Performance Primitives (IPP). We also hand-tweaked the P3 benchmarks to improve their performance; for instance, we modified the STREAM benchmark to use the SSE instruction set, which greatly improved performance.

5.2 Evaluation of a Single Tile using SpecInt and SpecFP

To establish the baseline performance of a single Raw tile, we ran a selection of Spec2000 benchmarks, compiled with gcc 3.3, on both Raw and the P3. For each benchmark, two Raw simulations were performed. One simulation ("425") uses Raw's actual frequency, and the other uses the assumption that Raw runs at the same frequency (600 MHz) as the P3. Both simulations use the same **RawPC** motherboard timings; thus, cache-miss latencies, when measured in cycles, are greater in the 600 MHz simulation.

As shown in Table 5.4, one 16 mm^2 Raw tile is surprisingly close to the 106 mm^2 P3. At 425 MHz, Raw has 52% of the performance of the P3. At 600 MHz, Raw has 65% of the performance. The impact of cache misses is evident: a 41% increase in frequency results in only a 25% increase in performance. The ability to use other tiles as a L2 cache could serve to reduce this penalty, especially

[4]The Raw gcc backend, based on the MIPS backend, targets a single tile's compute and network resources.
[5]For P3, we added -march=pentium3 -mfpmath=sse

Benchmark	Source	# Raw Tiles	Cycles on Raw (425 MHz)	Speedup vs P3-600 Time	Cycles on Raw (600 MHz)	Speedup vs P3-600 Time
171.swim	SPECfp	1	1.27B	0.87	1.49B	1.05
172.mgrid	SPECfp	1	.240B	0.69	.263B	0.89
173.applu	SPECfp	1	.324B	0.65	.359B	0.82
177.mesa	SPECfp	1	2.40B	0.53	2.42B	0.74
183.equake	SPECfp	1	.866B	0.69	.922B	0.91
188.ammp	SPECfp	1	7.16B	0.46	9.17B	0.51
301.apsi	SPECfp	1	1.05B	0.39	1.12B	0.52
175.vpr	SPECint	1	2.52B	0.49	2.70B	0.64
181.mcf	SPECint	1	4.31B	0.33	5.75B	0.34
197.parser	SPECint	1	6.23B	0.48	7.14B	0.60
256.bzip2	SPECint	1	3.10B	0.47	3.49B	0.59
300.twolf	SPECint	1	1.96B	0.41	2.23B	0.50
Geometric Mean				0.52		0.65

Table 5.4: Performance of SPEC2000 programs on one tile on Raw. MinneSPEC [63] LgRed data sets were used to reduce simulation time.

in memory-bound applications like mcf. Overall, these single-tile results demonstrate the degree to which the P3's microarchitectural approach to scalability has met with diminishing returns.

5.3 Multi-tile Performance as a "Server Farm On A Chip"

Tiled microprocessors can also serve as chip-multiprocessors. Like a chip-multiprocessor (CMP), fully tiled microprocessors, as exemplified by the ATM and Raw[6], are composed of autonomous and largely self-sufficient computing elements linked by one or more communication networks. This allows these fully-tiled microprocessors to efficiently execute many independent programs simultaneously, operating as a server-farm-on-a-chip ("SFOC"). This is an important advantage relative to more centralized designs like wide-issue superscalars and partially-tiled microprocessors.

As this chapter transitions from evaluating single-tile performance to evaluating multi-tile performance, we start by evaluating Raw's abilities as a server-farm-on-a-chip. We employ a SpecRate-like metric by running multiple copies of the same Spec application on all tiles. In this section, we report only numbers for Raw running at 425 MHz. Raw, running at 600 MHz, would have even better performance.

Table 5.5 shows the throughput of Raw running 16 copies of an application relative to the performance of the same application run 16 times in a row on a single P3. In the table, the column labeled "Efficiency" shows the ratio between the actual throughput and the ideal 16x speedup attainable with 16 tiles. The column labeled time incorporates the 425 versus 600 MHz clock difference.

[6]In partially-tiled architectures such as TRIPS [96], tiles have more limited ability to operate autonomously.

Benchmark	Cycles on Raw (425 MHz)	Speedup vs P3-600 Time (425 MHz)	Efficiency (425 MHz)
172.mgrid	.240B	10.6	96%
173.applu	.324B	9.9	96%
177.mesa	2.40B	8.4	99%
183.equake	.866B	10.7	97%
188.ammp	7.16B	6.5	87%
301.apsi	1.05B	6.0	96%
175.vpr	2.52B	7.7	98%
181.mcf	4.31B	3.9	74%
197.parser	6.23B	7.2	92%
256.bzip2	3.10B	7.1	94%
300.twolf	1.96B	6.1	94%
Geometric Mean		7.36	93%

Table 5.5: Performance of Raw on server-farm workloads relative to the P3.

Deviation from the ideal 100% efficiency is caused by interference of memory requests that are passing over the same network links, DRAM banks, and I/O ports. The efficiency averages 93%, which suggests that a microprocessor with more tiles and the same RawPC memory system, containing eight memory banks, would scale to even higher throughputs.

Eventually, as the number of tiles increases, one of the shared resources will become a bottleneck. This point can be deferred by using all 14 DRAM ports instead of the 8 used in the **RawPC** configuration. Eventually, as even those additional resources become insufficient, it makes sense to allocate more than one tile to a process. The memory resources of additional tiles can be used as caches to reduce the traffic to the external memory system.

Overall, Raw, even at 425 MHz, has a geometric mean of 7.36x improvement over the Spec throughput of the 600 MHz P3. Normalizing for Raw's larger die area, Raw's throughput per mm^2 is 2.3x better. These results suggest, in a parallel to [90], that fully-tiled architectures composed of simpler tiles can indeed offer higher threaded performance than conventional chip multiprocessors composed of wider-issue superscalars.

5.4 Multi-tile Performance on sequential C and Fortran applications

In contrast to current-day chip multiprocessors, tiled microprocessors focus on cheap communication between remote ALUs. To provide this kind of communication, it is necessary to provide fast, generalized transport networks and thin, tightly-coupled, low-overhead interfaces (as measured by

the 5-tuple, described in Section 2.4.4) for data transport and synchronization. In tiled microprocessors, this communication can facilitate both the traditional message passing and shared memory parallel programming models as well as finer grained parallel models, such as streaming and ILP (instruction-level parallelism), that are enable by an SON.

In this section, we continue the examination of multi-tile performance on Raw by considering traditional C and Fortran applications that have been automatically parallelized using the Rawcc compiler [72, 8, 7, 71]. Rawcc relies extensively on the low-cost communication provided by Raw's SON to exploit ILP. Rawcc takes as input a sequential C or Fortran program and compiles it so that it executes in a parallel fashion across the Raw tiles, using the inter-tile SON. Rawcc simultaneously tries to extract both instruction and memory parallelism from sequential programs.

5.4.1 Memory Parallelism

Rawcc enhances memory parallelism by analyzing and transforming the high-level program in order to increase the number of *alias equivalence classes* ("AECs") [8, 9]. To do so, Rawcc operates on *memory objects*, which refer to either a static program object (such as a statically declared array), or the set of run-time objects created from the same program location (e.g., stack or heap objects). The AECs form a compile-time partitioning of memory objects into sets such that each memory access (load or store) instruction is proven to access memory objects from at most one set. To this end, Rawcc's *pointer analysis* allows Rawcc to accurately determine which load and store instructions may point to which memory objects. In order to increase the number of memory objects in the system, Rawcc subdivides array objects into multiple subarrays and divides structures (and arrays of structures) into their constituent fields. Because subdivided arrays often remain in the same AEC, array subdivision is performed simultaneously with *modulo unrolling*. *Modulo unrolling* selectively unrolls loops and replicates memory access instructions so that individual instructions can be proven to access different subarrays.

In some cases, the lack of memory dependence information for a particular memory access would require the compiler to assign many objects to the same AEC. For instance, a `void *` pointer which is returned from an unanalyzed routine(e.g., a library routine) may, from the compiler's perspective, be able to point to almost any memory object. Ordinarily, this would cause the collapse of all of the AEC partitions into a single partition, eliminating memory parallelism. To deal with these kinds of problems, Rawcc creates two types of AEC partitions. The first is the *common-case* AEC partition, which is formulated ignoring these problem accesses. The second, the *total* AEC partition, is formulated with the problem accesses[7]. The basic idea is to improve program performance by allowing the common-case to run more quickly at the expense of making a small number of problem

[7]Rawcc's "problem accesses" currently are selected only in the context of accesses from modulo-unrolled regions. The loop bodies are formulated to contain only disambiguated accesses (i.e., members of the common-case AEC), while the ramp-up and ramp-down code contains non-disambiguated accesses that require special treatment.

accesses run more slowly.

For each AEC in the common-case partition, Rawcc assigns the corresponding memory access instructions to the same tile. When the program executes, the corresponding memory objects will be accessed only through that particular tile's cache. This is an important property, both because the tiles are not cache-coherent and because accessing a memory object solely from a given tile's cache reduces penalties due to memory coherence traffic. Furthermore, because the tile's instructions have been compiled in an order that respects program dependences (and the tile implements in-order memory semantics for its local instruction stream), there is no need for explicit locking or synchronization with other instructions that may access the same memory location. This synchronization would be necessary even in a cache-coherent system (to respect memory dependences) if instructions in the same AEC were mapped to different tiles.

Common Case Accesses In the regions of the program that do not contain problem accesses, memory accesses occur with little overhead. Each tile issues the load and store instructions that were assigned to it in-order, based on the common-case AEC partition. The addresses and store-values may be specified by other tiles, in which case they will be routed over the inter-tile SON into the tile that was assigned to access that memory object. If the load-result-value is needed by another tile, the load is performed on the tile that owns the memory object, and the result is sent out over the inter-tile SON.

Problem Accesses In regions of the program that contain problem accesses, there are two types of issues that must be dealt with: enforcing memory dependences and enforcing cache coherence. These issues stem from the fact that we do not know at compile time which AEC the memory object referenced by a problem (or *non-disambiguated*) memory access belongs to. To correctly deal with these issues, the compiled program must employ the on-chip networks to ensure the enforcement of dependences and to avoid cache coherence hazards.

The enforcement of memory dependences is potentially the more difficult of the two. Because non-disambiguated memory accesses are not located on the tiles that own the memory objects in question, they are not automatically synchronized (through the semantics of sequential single-tile execution) with respect to other accesses to the same memory objects. Rawcc employs the *serial static ordering* technique, where all problem accesses belonging to the same AEC in the total AEC partition are assigned to the same tile, and then dispatched, in-order, over the GDN from the same tile. Because the GDN respects message ordering for messages with the same sources and destinations, these problem accesses are guaranteed to occur in order and will not violate dependences. Then, all that remains to be done is to obey dependence constraints between problem memory accesses and the common case accesses, which can be performed through compile-time multicast messaging over the inter-tile SON.

In order to avoid cache coherence hazards, it is necessary to transmit the request to the tile that owns the memory location so that the owner tile can process that request on the behalf of the sender. In Rawcc, this request is transmitted over the GDN in a point-to-point message destined only for the tile that owns that particular memory address.

Section 5.4.4 examines some modifications to the existing system that could improve the enhancement of memory parallelism versus the baseline Rawcc system.

5.4.2 Instruction Parallelism

In addition to trying to achieve parallelism during accesses to memory objects, Rawcc also strives to achieve parallel execution of instructions. To achieve this, the Rawcc backend, called Spats, performs a series of program transformations that map an input program across a set of tiles. We summarize these transformations here, and refer the reader to [71] for further detail.

Spats examines each procedure individually. Spats starts by performing SSA *renaming* across the procedure, in order to eliminate unnecessary anti- and output- dependences. It then divides the procedure into a group of *scheduling regions* through a process called *region identification*; each region is a single-entry, single-exit portion of the control flow graph that contains only forward control-flow edges. Finally, it inserts a number of "dummy" read and write instructions that are used to manage persistent scalar variables that must be communicated between scheduling regions. Figure 5-1a shows these steps.

Intra-Region code generation Afterwards, Spats builds a data dependence graph to represent the region, shown in Figure 5-1b. The nodes of the graph represent instructions, while the arrows represent data dependences. In the diagram, the numbers to the right of the instruction are the latency of the instruction, in cycles. The alphanumeric values to the left of the instruction are a unique identifier for the instruction, that allows the reader to track an instruction across subsequent phases of compilation.

Subsequently, Spats performs *instruction partitioning* and *scalar partitioning*, followed by *instruction assignment* and *scalar assignment*. These phases are collectively responsible for assigning each instructions in a scheduling region to a tile. This assignment tries to spread instructions across multiple tiles in order to increase parallelism. At the same time, it tries to allocate communicating instructions to the same tile or to a nearby tile to minimize the impact of communication on the critical path of the program. The assignment furthermore respects the load and store constraints determined by the AEC partitions, and makes sure that the dummy read and write instructions for a given scalar variable are mapped to the same "home" tile[8]. Figure 5-1c shows the assignment of both instructions and scalar values to tiles.

[8]Much like function linkages, the home location of a variable facilitates the "handoff" of scalar values that persist across regions.

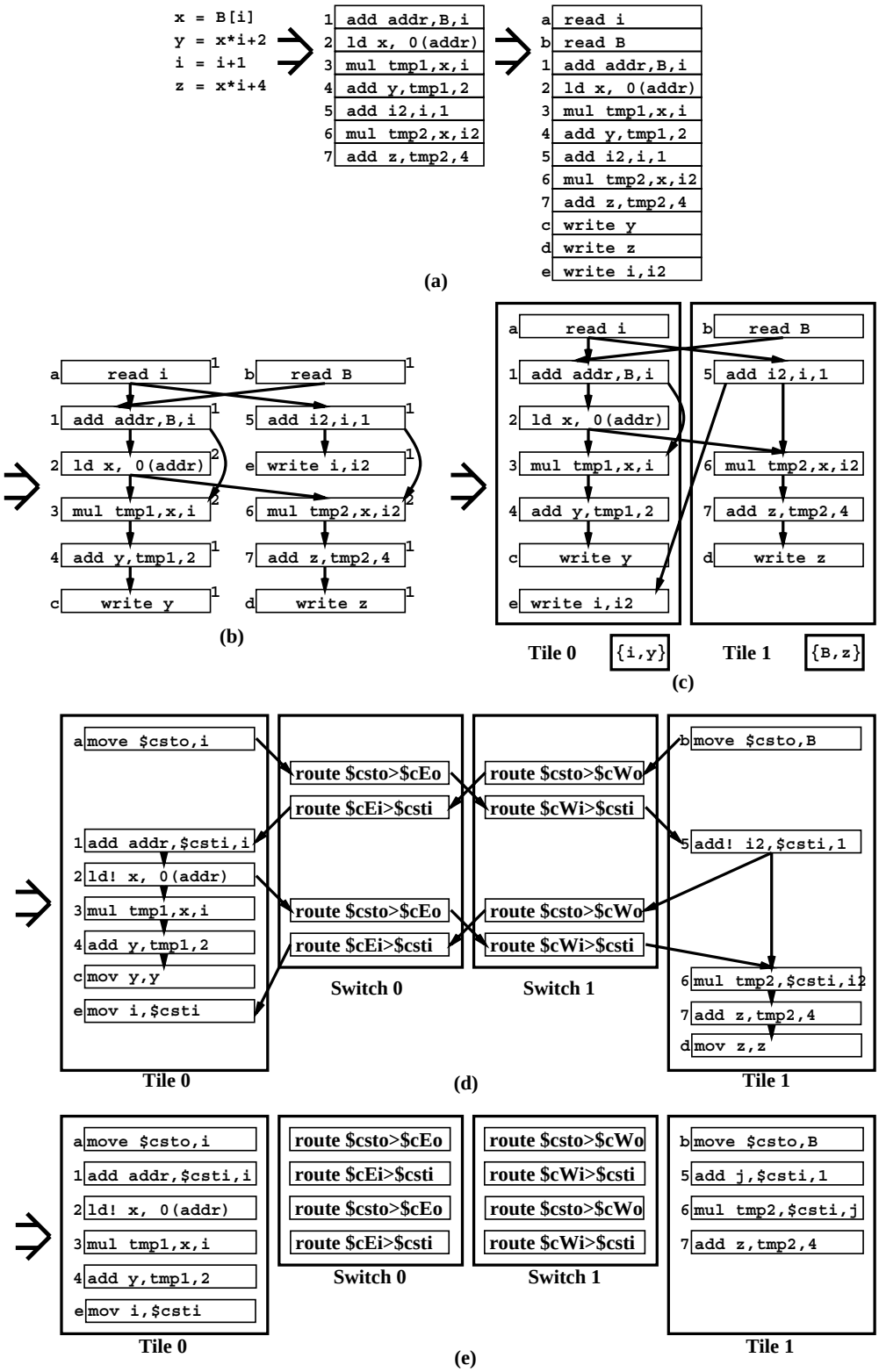


Figure 5-1: Sequence of transformations performed by Rawcc on a single region: a) The region undergoes *renaming* and *region identification*, b) A data dependence graph is constructed, c) *Instruction partition*, *scalar partition*, and *scalar and instruction assignment* are performed, d) *Route assignment* is performed and e) The instructions are *scheduled*. Source: [71].

At this point, the locations of objects in the scheduling region have been determined. Spats now can perform *route assignment.* Route assignment, shown in Figure 5-1d, is responsible for generating the switch processor communication instructions that route values between dependent compute processor instructions that reside on different tiles. Spats allocates one switch instruction to each tile along the path that the operand takes over the inter-tile SON. Because Spats allocates dependent compute processor instructions in a way that minimizes inter-tile communication, contention is seldom a problem Accordingly, Spats uses a simple dimensioned-ordered algorithm (e.g, perform all X routes, then all Y routes) for determining the path that operands take between distant tiles. The Spats algorithm also implements multicasting over the inter-tile SON. In this phase, Spats also converts dummy read and write instructions into corresponding `move` instructions.

After route assignment, the assignment of instructions (whether switch or compute processor) to tiles for the region has been almost completely determined. All that remains is to *schedule* them. Spats uses a list scheduling algorithm that maintains a list of instructions that are ready to be scheduled. It repeatedly chooses the highest priority instruction that is ready and assigns it an issue slot on the tile that it has been assigned to. In the case of switch processor routing instructions, Spats schedules all switch processor instructions for the entire route atomically. This guarantees that all routes can make forward progress, and avoid the need for the scheduler to backtrack.

Inter-Region code generation After the regions have been scheduled, the task remains to generate the code that transitions between regions. *Stitch code* is code that transfers scalar values from their home locations in one region to their home locations in another region. This stitch code consists of additional route or `move` instructions.

Finally, the register allocator is run on each procedure. The register allocator is designed to allocate the same physical register to the sources and destinations of move instructions in order to allow them to be eliminated. This is helpful in eliminating the `move` instructions that result from the dummy read and write instructions and stitch code.

5.4.3 Results

With an understanding of how Rawcc maps programs to the Raw microprocessor, we can look towards the performance that it is able to attain. The set of benchmarks used to evaluate Rawcc performance on Raw is shown in Figure 5.6. The benchmarks generally fit into two classes; the first class are the dense-matrix scientific applications, which have ample parallelism that can be revealed by loop unrolling. The second class of benchmarks consists of sparse matrix and irregular benchmarks which have more limited parallelism and require the compiler to manage locality-parallelism tradeoffs carefully. More details can be found in [71, 112]. Raw, at 425 MHz, attains a geometric mean speedup of 2.23x over the 600 MHz P3. This speedup comes from instruction level parallelism, and

Benchmark	**Origin**	Lines of Code	**Raw Tiles**	**Cycles on Raw**	**Speedup vs. P3-600**	**Cycles on Raw**	**Speedup vs. P3-600**
				425 MHz		600 MHz	
Dense-Matrix Scientific Applications							
Swim	Spec95	618	16	14.5M	2.9	16.2M	3.7
Tomcatv	Nasa7:Spec92	254	16	2.05M	1.3	2.17M	1.7
Btrix	Nasa7:Spec92	236	16	516K	4.3	545K	5.8
Cholesky	Nasa7:Spec92	126	16	3.09M	1.7	3.22M	2.3
Mxm	Nasa7:Spec92	64	16	247K	1.4	248K	2.0
Vpenta	Nasa7:Spec92	157	16	272K	6.4	305K	8.1
Jacobi	Raw bench. suite	59	16	40.6K	4.9	40.6K	6.9
Life	Raw bench. suite	118	16	332K	2.9	335K	4.1
Sparse-Matrix/Integer/Irregular Applications							
SHA	Perl Oasis	626	16	768K	1.3	778K	1.8
AES Decode	FIPS-197	102	16	292K	0.96	306K	1.3
Fpppp-kernel	Nasa7:Spec92	735	16	169K	3.4	181K	4.5
Unstructured	CHAOS	1030	16	5.81M	1.0	5.83M	1.4
Geometric Mean					2.23		3.01
Average		344					

Table 5.6: Performance of sequential programs on Raw and on a P3.

Rawcc's ability to exploit memory parallelism to increase the number of cache ports and effective L1 cache size (16 banks of 32KB each).

We can also examine how the speedups vary with the number of tiles in order to see how performance characteristics would change as Raw processors with more tiles are constructed. Figure 5.7 shows how application performance varies with tile count. The numbers are all normalized to the performance on one tile, compiled with Rawcc. Generally, the sparse matrix applications have mostly reached their maximum speedup by 16 tiles. On the other hand, many of the dense matrix applications still have increasing speedups as the number of tiles is increased. Many of the applications have super-linear speedups due to the increase in effective register and cache storage as the number of tiles increases.

We have also included the single tile performance using gcc 3.3. This effectively is a measurement of Rawcc's single-threaded code quality. It is evident from the data that gcc's code quality is generally much better than Rawcc's. In Rawcc, a number of compiler phases were omitted in the interests of focusing on the key research aspects of tiled microprocessor compilation. Despite this, Rawcc's ability to exploit parallelism to a large degree compensates for any losses in performance due to omitted passes. A commercial Rawcc implementation that implemented these passes would likely result in better speedups, or equal speedups with fewer tiles.

Benchmark	Number of tiles							
	gcc 3.3	**1**	**2**	**4**	**8**	**16**	**32**	**64**
Dense-Matrix Scientific Applications								
Swim	2.4	1.0	1.3	2.7	5.3	10.0	22.2	34.7
Tomcatv	1.9	1.0	1.3	3.2	5.7	8.4	10.2	10.9
Btrix	1.5	1.0	1.8	6.2	18.3	41.9	75.7	-
Cholesky	1.7	1.0	1.8	5.1	9.8	12.4	12.6	11.3
Mxm	1.3	1.0	1.6	4.8	6.9	8.7	9.9	10.5
Vpenta	1.5	1.0	2.8	12.0	34.5	63.8	132.0	-
Jacobi	1.1	1.0	3.0	6.3	14.4	24.8	45.4	62.9
Life	1.5	1.0	1.0	2.2	5.9	13.5	23.3	49.5
Sparse-Matrix/Integer/Irregular Applications								
SHA	.9	1.0	1.5	1.5	1.7	2.3	1.9	2.0
AES Decode	1.1	1.0	1.6	2.5	3.1	3.2	2.5	2.3
Fpppp-kernel	1.3	1.0	0.9	1.9	3.5	6.6	7.7	11.8
Unstructured	1.8	1.0	1.8	3.2	3.6	3.3	3.0	3.2

Table 5.7: Speedup of the ILP benchmarks relative to Rawcc targeting a single 600 MHz Raw tile. The following results are shown: gcc 3.3 on 1 tile, and Rawcc on 1, 2, 4, 8, 16, 32, and 64 tiles, in that order.

5.4.4 Future Improvements

Looking back, there are a few improvements of this compilation and execution model that would be interesting to examine. Many of these aspects result from the way in which the architecture has evolved since the compiler was initially designed.

Improving the AEC model Rawcc creates two partitions, the common case AEC partition and the total AEC partition. The "problem accesses" are determined only in the special case of unrolling. Thus, a single problem pointer that defies pointer analysis could cause AEC collapse relatively easily and greatly reduce memory parallelism. This makes the performance of the system more brittle.

A more general approach reuses the concept of a common-case AEC partition. However, instead of determining this partition by selecting problem accesses through solely modulo-unrolling, a more general analysis could be performed. By annotating the estimated number of times each instruction is executed, the compiler could determine those accesses that are relatively unimportant in the execution of the program. These accesses would be used to determine the set of problem accesses that would be excluded from the formulation of the "common-case" AEC. This set would be larger than that provide by modulo unrolling alone. Then, the common-case AEC could be used for determining the mapping of memory objects to tiles, and for most loads and stores.

Furthermore, there are a few alternatives that merit examination with respect to the way in which problem accesses are handled. Instead of using the software-serial-ordering ("SSO") technique in combination with the total AEC partition, the compiler can instead use a dependence-based

approach. Problem accesses are multicasted over the inter-tile SON to all possible tiles that may need to service the access. Then, the receiving tiles conditionally dispatch on the address – either ignoring the request, or processing the request. This check could be performed with a branch, predicted unlikely to minimize the cost to tiles that do not need to process the request. In the case of stores, the recipient tile performs the store and continues. In the case of loads, the recipient tile sends the result to a known tile over the GDN. That tile can then forward the result to wherever it needs to go. Because all loads and stores are still ordered on each tile, dependences are automatically respected. At first glance, the multicast of memory access requests to all possible recipient tiles seems like a potential disadvantage versus the serial static ordering technique. However, the multicast technique actually requires the same set of multicasts as the SSO technique because of the need to manage dependences. Also, the cost of dynamic network requests ends up being much higher than the corresponding requests over the SON due the cost of synchronization. The multicast approach also allows the possibility for more parallelism than the total AEC approach, because equivalence classes are a more conservative serialization than strict dependence ordering. This is because the total AEC incorporates all possible dependences in the entire program, while the dependence ordering only looks locally. For example, a pointer in the startup portion of a program may cause the total AEC to merge two AECs in the common-case partition. However, in a region of code where that pointer is not used, the total AEC will still cause accesses to the two AECs to be serialized where as the dependence approach would be able to process them independently.

There is one case in which the use of the GDN for sending requests could still improve performance. In this case, it is known that a set of memory accesses are all independent; however the destinations are unknown. In that case, multicast over the SON would be inefficient. Instead, it makes sense to issue these requests from arbitrary locations over the GDN, and to bookend the requests with a barrier over the inter-tile SON. These barriers are quite efficient in the Raw microprocessor, taking only the number of cycles for a message to travel from one corner of the Raw array to the opposite corner.

Incorporating Dynamic Memory Object Placement Another extension which merits future examination is the possibility of allowing dynamic memory object placement; in other words, allowing the memory object to tile mapping to vary over time. This allows the compiler to alter the memory assignment to improve performance, for instance in a case where two major loops have differing access patterns that merit different data layouts. It would seem that having static memory is ultimately a limiter to performance because as programs get larger, they may be more likely to have significant regions of execution that access memory in different ways.

In dynamic memory object placement, a global common-case AEC partition is no longer employed for assigning memory objects to tiles. Instead, a common-case AEC partition is created for each code region. When memory objects are migrated between regions, the compiler routes a value over the

inter-tile SON between the old and new owners of the memory objects to synchronize the transfer of ownership and ensure that memory dependences are respected[9] In this case, algorithms would need to be developed to introduce an affinity model that incorporates the notion that there is a penalty for changing the tile-memory object mapping. Or, the compiler could compile a few versions of the code that employ different mappings, so that it can decide using run-time information whether to perform migration.

Although this memory migration could be performed statically through an explicit copy or by flushing the cache of the owning tile, this may cause the system to a penalty higher than it needs to, especially in cases where most of the data may have already vacated the cache. An alternative which leverages prior multiprocessor research is to implement cache-coherence in the system. As long as dependences are respected in the program, using the techniques described in the previous paragraph, the system could allow the cache-coherence hardware to manage the migration of data automatically and with potentially lower cost. In the context of dynamic memory object placement, it may be necessary to have finer grained cache lines inside the Raw tile, so as to eliminate cache line ping-ponging that occurs when words in the same cache line are mapped to different AECs.

Control flow speculation One of the performance limiters in the Raw system is the multicast communication implicit in global control-flow operations. In some cases the control flow operations (such as the incrementing of a counter) can be distributed across the tiles, and they can operate mostly autonomously. However, in other cases, the control flow unfolds as part of the computation. In this case, the transmission of the branch condition amounts to a multicast to all of the tiles, which end up stalling. An alternative is still support a limited form of speculation that allows tiles to assume a particular direction of control flow, and continue execution until the branch condition arrives. If it turns out that predicted direction is incorrect, the tile can then undo the changes that it speculatively made.

In such a system, we need to examine all forms of state in the system and determine a way of rolling back for each one. This state includes register state, memory state, and network state. Many of the issues in this arena are similar to those in conventional VLIWs and out-of-order superscalars. Multiple simultaneous copies of variables stored in registers (one copy for each loop iteration) can be accomplished through hardware register renaming, or through unrolling/pipelining and code duplication, with fix up code on loop exit. In both cases, it would help to have a large number of registers available, even within a tile. Arrays of Raw tiles effectively have very long latencies to hide because of the cost of inter-tile communication. The longer these latencies, the more copies of each variable are needed to hide the latency, and the more registers would be helpful in hiding that latency[10].

[9] Or, if there are many such synchronizations, a barrier could be performed over the inter-tile SON.

[10] Ultimately, systems that have enough local state to perform perfect latency hiding are unscalable, but the technique would provide at least a constant factor improvement in performance.

Memory state would need non-excepting loads so that loads could be hoisted above the branches that guard them. In Raw, this might require some simple hardware that filters out cache misses to addresses that would cause messages to get sent to I/O ports that don't even have DRAMs on them. For stores, a store buffer with some form of commit mechanism would be useful. This change is not inordinately problematic in the current implementation, as the Raw compute processor already has a store buffer in order to allow stores to be pipelined. However, one significant difference is that Raw's store buffer is currently always written back before a cache miss is issued; thus the store buffer is designed with the invariant that the applicable cache lines are in the cache, and it never has to check cache tags. In the case where the store buffer contains speculative accesses, these values cannot be drained on a cache miss, because that would effectively cause them to be committed. Thus, the store buffer would need to be able to cause cache misses to occur.

Network state is also important issue that needs to be considered during speculative operation. For scheduling purposes, different tiles may schedule loop exit branches in different locations. As a result, a sender tile may transmit a value before evaluating the branch that it depends on. On the other hand, the receiver tile may schedule the receiving instruction after the same corresponding branch. Now, some method must be created to dequeue the spurious element from the network. One such approach would have the tile inject the results of the branches into the inter-tile SON, so that the switch processors can branch appropriately. However, this incurs additional complexity. A simpler solution is to require that all network sends and receives in the unrolled compute processor code be retained, and to keep one schedule for the static network. Thus loop exit fix up code will potentially contain sends and receives that are there just to satisfy the expectations of the static network.

SON Efficiency Improvements One of the tensions that became apparent as the Raw microprocessor and Rawcc compiler were developed is the issue of operand arrival ordering. In many cases the optimal time for producing a value at a sender node is independent of the optimal time for the receiver to receive it. We can take insight into this problem from the way that postal mail is delivered from sender to receiver. Most people would find it extremely disruptive to their lives if they had to be around to read their mail exactly as it comes off the airplane that transported it over the long haul. Similarly, although the compiler can strive to schedule code so that the receiver issues instructions in the same order that corresponding operands arrive, this reduces performance and increases the complexity of the scheduler. In many cases, the scheduler cannot match operands appropriately and is forced to introduce additional `move` instructions that transfer the value of `csti` or `csto` to the compute processor's register file in order to clear the FIFO for other operands to come in. We call this `move` instruction an *inter-tile SON spill*, in analogy with the traditional *intra-tile SON spill* (i.e. to the stack) which is also used to time-delay operands. Inter-tile SON spills also occur when the tile needs to use an operand that it dequeues from a NIB more than once.

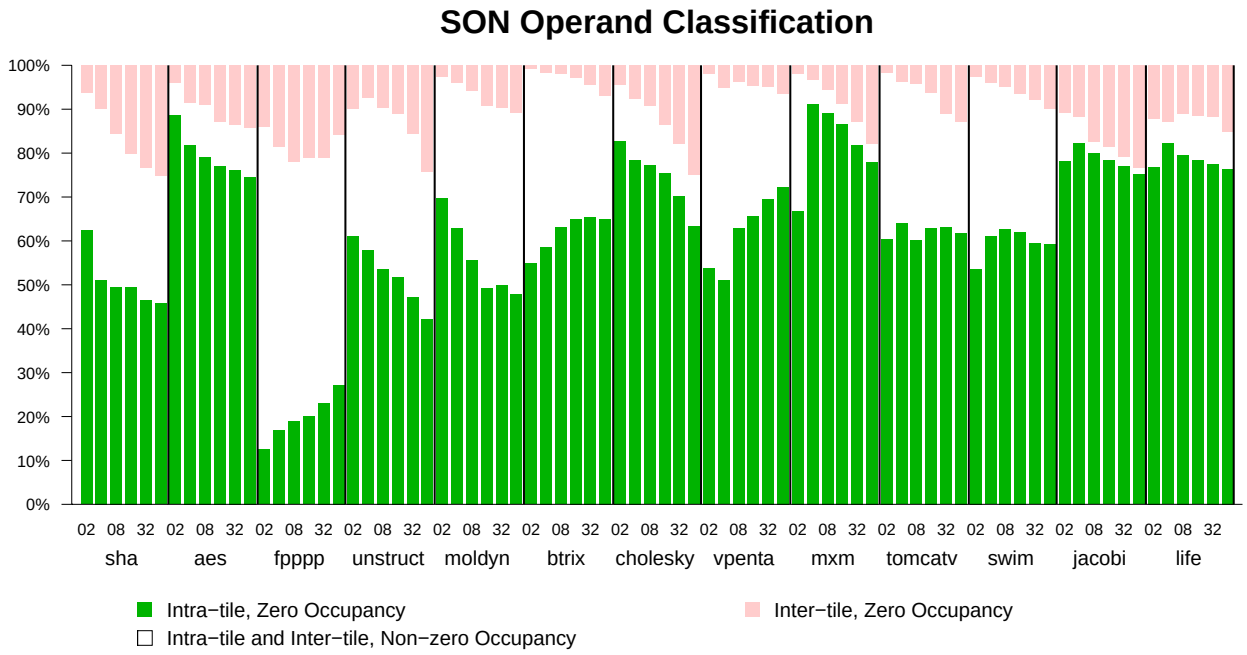


Figure 5-2: SON Operand classification. The shaded regions are operands that traveled through the system without incurring any occupancy. The top region is the percentage of operands that traveled between tiles over the inter-tile SON without a inter-tile SON spill. The bottom region is the percentage of operands that traveled only within a tile without an intra-tile SON spill. The white middle region is the percentage of operands that had some form of spill.

In order to get more insight into this issue, and in general, into Rawcc's use of the SON, we instrumented our simulation infrastructure to track operands on both the inter-tile and intra-tile SONs. In the data collected, each unique operand is counted only once per generating event (i.e., result from functional unit, arrival via transport network, or reload from cache). Figure 5-2 shows for each of the benchmarks, and varying from 2 to 64 tiles: 1) the percentage of operands that traveled between tiles without an inter-tile SON spill, 2) the percentage of operands that traveled only within a tile without an intra-tile SON spill, and 3) the percentage of operands that had some form of spill. Thus, the third category, in white, represents an opportunity for improvement. Spat's efforts to reduce inter-tile communication are evident here; the majority of operands travel only over the intra-tile SON, represented by the bottom part of the figure. We note that as the number of tiles increases, and the degree of parallelism also increases, the number of inter-tile operands rises.

To understand the impact of inter-tile SON spills, we look to Figure 5-3. In some cases, a significant percentage of operands required the insertion of `move` instructions, either because they needed to be time-delayed, or to be used multiple times.

A natural response to this inefficiency is to try to solve the problem by reserving encoding space in the compute processor instruction set. This space could implement free transfers from NIBs to

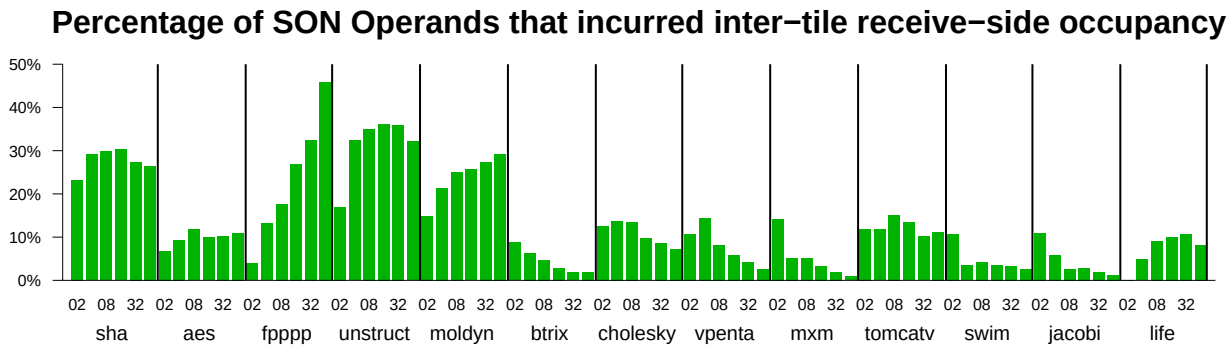


Figure 5-3: Percentage of all operands that both 1) traveled inter-tile and 2) were inter-tile SON spilled.

the register file, or to allow random access or selective dequeuing of NIB contents, or to specify one of a greater number of NIBs. For the Raw microprocessor, this would require increasing the compute processor instruction word size, which although not an insurmountable issue, is undesirable.

An alternative is to give the switch direct access to the compute processor's register file, so that it can deposit values directly there. Exploration of this scenario quickly reveals the need to synchronize the switch processor with the compute processor in order to preserve dependences.

In the Raw design, one promising solution is to use the existing switch processor's local register file. Each switch processor in the existing design has a 4-element 1R 1W register file for holding local loop counts and return addresses. The switch processor instruction set allows values to be routed from the networks into the register file, or out of the register file to any network output. Values that need to be time-delayed or multicasted can be held there, and then injected into the csti or csti_2 NIBs each time the compute processor needs it. A register allocator can determine which register file slots the switch processor uses. Because the NIBs are synchronized with Raw, dependences can be easily preserved.

Unfortunately, the limited number of read and write ports, and limited register file size caused us to doubt the utility of implementing this functionality in our existing infrastructure. Future Raw designs might consider increasing the size and number of ports in these switch processor register files.

Figure 5-4 shows the percentage of operands that required intra-tile SON spills. As can be seen, a relatively large number of spills occur in some of the benchmarks. These spills typically are a by-product of the compiler's efforts to exploit parallelism. The same transformations that increase parallelism in the program often increase the number of live values that need to be maintained in the computation. Future versions of Rawcc could try to incorporate the notion of register pressure into the instruction and scalar assignment and scheduling phases. However, as discussed in the section on Control Flow Speculation, the addition of more registers generally increases the architecture's

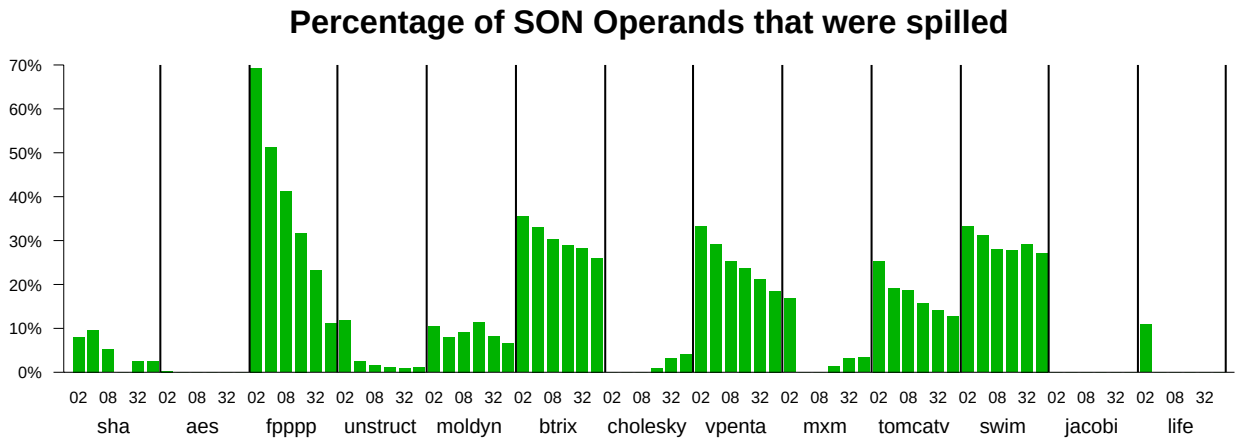


Figure 5-4: Percentage of all operands that both 1) traveled only intra-tile and 2) were intra-tile SON spilled.

ability to perform latency hiding, exploit more parallelism and alleviate phase-ordering problems in the compiler.

5.5 Multi-Tile Performance on Streaming Applications with StreamIt

Stream computations The previous section examined Raw's performance on traditional sequential applications written in C or Fortran, using a parallelizing compiler. We transition to examine performance characteristics on the class of *stream computations*. Stream computations typically operate on large or even infinite data sets. However, each output word (or group of words) typically depends on only a selected region of the input. Thus, typical stream programs need only buffer a certain portion of the input in local storage until the given output words have been computed; then the inputs can be discarded. For these types of applications, conventional cache heuristics often perform suboptimally, so it is frequently profitable to explicitly manage the transport of data as it flows over the on-chip networks and between ALUs and memories. Some computations may not be strictly stream computations, for instance, repeatedly rendering triangles to a frame buffer, but are close enough that the model works well [16]. Stream computations often occur naturally in embedded and real-time I/O applications.

5.5.1 The StreamIt Language

As in the previous section, this section examines Raw's performance when the computation is expressed in a high level language. In this case, the benchmarks employ StreamIt, a high-level,

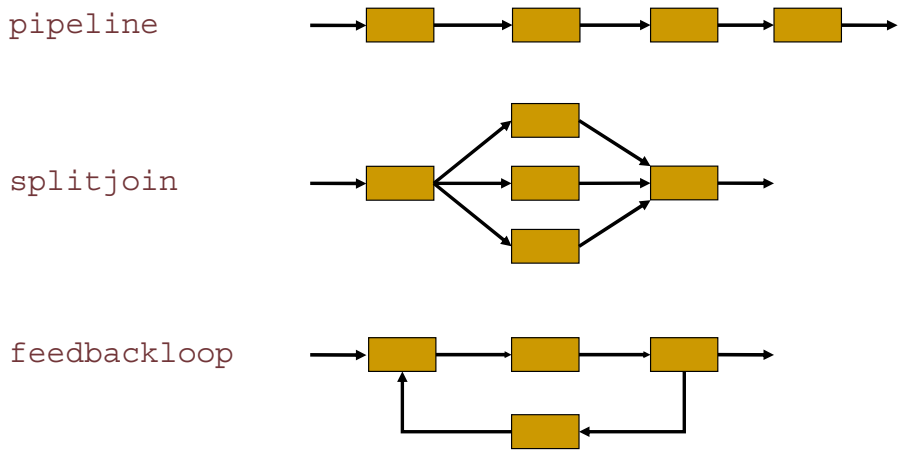


Figure 5-5: StreamIt filter composition. Filters can be combined linearly, in the form of a pipeline. Special splitter and joiner filters can be used to fanout and fanin the stream. Finally, feedback loops can be created for filter feedback. Source: MIT StreamIt Group.

architecture-independent language [115]. StreamIt allows the programmer to expression the application as a collection of filters that are connected by communication channels in a hierarchical graph. Each filter has its own control-flow, its own address space, and independent sense of time. Furthermore, filters have an *input stream* and an *output stream*, each of which consists of a sequence of typed records. The types of the input and output streams do not have to be the same.

The composition of filters into a stream graph is accomplished through three keywords: `pipeline`, `splitjoin`, and `feedbackloop`. Figure 5-5 shows these keywords and an example of a stream graph that uses that keyword. To provide hierarchy, the three keywords can also be applied to subgraphs in addition to filters. We use the term *node* to refer to an entity that is either a subgraph or a filter. The `pipeline` keyword connects a number of single-input single-output nodes serially. The `splitjoin` distributes the elements of a single input stream to a collection of nodes, and then merges the outputs of the nodes together into an output stream. For splitting the elements, `split duplicate` copies the input stream to all interior nodes, while `split roundrobin` spreads the elements across the nodes. The `join roundrobin` command is used to gather the elements of the interior `splitjoin` nodes. Finally, the `feedbackloop` construct can be used to create structured cycles in the stream graph.

Figure 5-6 shows one example filter that processes an input stream of floats and generates an output stream of floats. Each instantiation has its own independent copy of the local array `weights`. Unlike for Rawcc, the communication between program components is explicit, so StreamIt can assume that memory objects are not shared between filters. The `init` function is run once, when the filter starts. The `work` function is run repeatedly as the filter runs. Each filter examines its input stream via the `peek` command, which allows the filter to look a bounded depth into the input stream without dequeuing. In the case of Figure 5-6, the parameter `N` is specified at compile-time by

```
float->float filter LowPassFilter(int N) {
    float[N] weights;

    init {
        for (int i=0; i<N; i++)
            weights[i] = calcWeights(i);
    }

    work push 1 pop 1 peek N {
        float result = 0;
        for (int i=0; i<N; i++)
            result += weights[i] * peek(i);
        push(result);
        pop();
    }
}
```

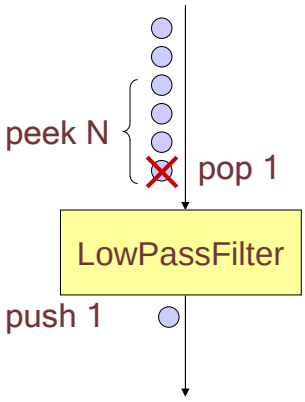


Figure 5-6: An example StreamIt filter. Each execution, it peeks at N elements of its input stream, enqueues one element onto the output stream and then dequeues one element from its input. Source: MIT StreamIt Group.

the code that instantiates the filter. StreamIt filters employ the `push` command to emit a value onto its output. Finally, StreamIt filters can use the `pop` command to dequeue a value from the input. The signature of the filter, `work push 1 pop 1 peek N` gives the StreamIt compiler the number of pops, pushes, and peeks that the filter `work` function performs on each execution. The StreamIt compiler uses this information to generate a compile-time schedule of filters according to their data rates and connectivity.

Figure 5-7 shows a hierarchically-implemented beamforming pipeline that employs the StreamIt `splitjoin` construct. The construction routine takes as a parameter the number of channels and beams, and generates a pipeline with the corresponding number of channels and beams.

5.5.2 StreamIt-on-Raw Compiler Backend

Along with the StreamIt language comes a StreamIt compiler that targets Raw. In addition to the functionality that maps StreamIt programs across a tiled microprocessors, the StreamIt compiler features a number of novel signal-processing-oriented features. For instance, *linear analysis* is used to reduce the number of floating point computations required [70, 3] and *teleport message* enables programmer-friendly stream control [116]. Because these components apply to both legacy and tiled microprocessors, we refer the reader to the original publications for descriptions of these features. Instead, this section focuses on the StreamIt backend [37], which shares with Rawcc the same basic *partitioning*, *assignment* and *routing* phases.

Partitioning The StreamIt `filter` construct simultaneously aids the programmer and the compiler. It provides the user with a convenient and natural way to express a stream computations. At the same time, it provides the compiler with a basic unit of parallelism with easy-to-analyze

```
complex->void pipeline BeamFormer(int numChannels, int numBeams)
{
    add splitjoin {
        split duplicate;
        for (int i=0; i<numChannels; i++) {
            add pipeline {
                add FIR1(N1);
                add FIR2(N2);
            };
        };
        join roundrobin;
    };

    add splitjoin {
        split duplicate;
        for (int i=0; i<numBeams; i++) {
            add pipeline {
                add VectorMult();
                add FIR3(N3);
                add Magnitude();
                add Detect();
            };
        };
        join roundrobin(0);
    };
}
```

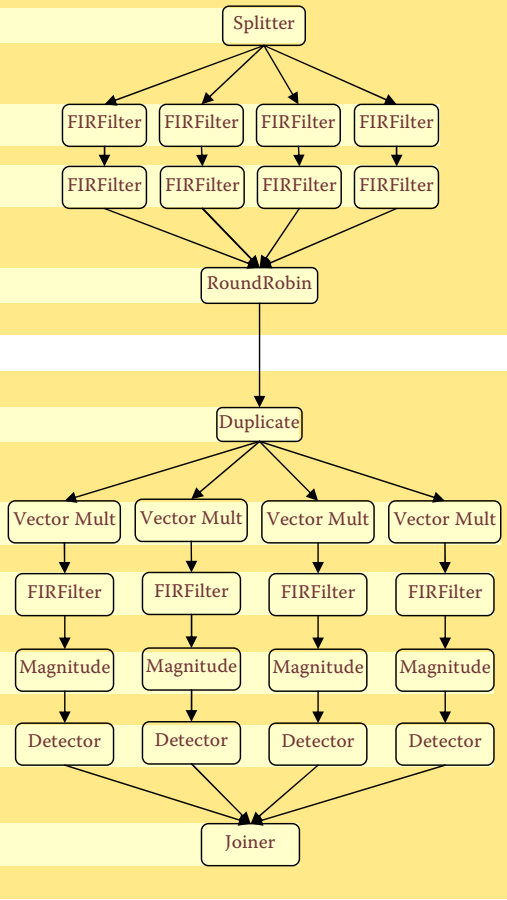


Figure 5-7: Implementation of a beamforming application. Demonstrated are the use of the `splitjoin`, `duplicate`, `pipeline`, `join`, `add`, and `roundrobin` keywords.

properties. Unlike for Rawcc, the dependences between filters are limited to those implied by the stream graph. StreamIt does not need to perform pointer analysis or modulo unrolling in order to distribute different filters to different tiles in order to preserve cache coherence and exploit memory parallelism. Furthermore, StreamIt can execute filters in parallel, which would be difficult with Rawcc because of the complexity of analyzing control dependences in a sequential representation of the program.

Although StreamIt has an easier time of discovering parallelism, there is still the need to divide the computation into equal portions, one portion per tile, so that the tiles are load balanced. The aim is to transform the stream graph so that it contains one node per tile, where each node has approximately the same amount of work. If one node has substantially more work than other tiles, then it would become the bottleneck in the pipeline. To this end, StreamIt's partitioner uses running time estimates of each filter's work function in order to guide a series of *filter fusion*, and *filter fission*

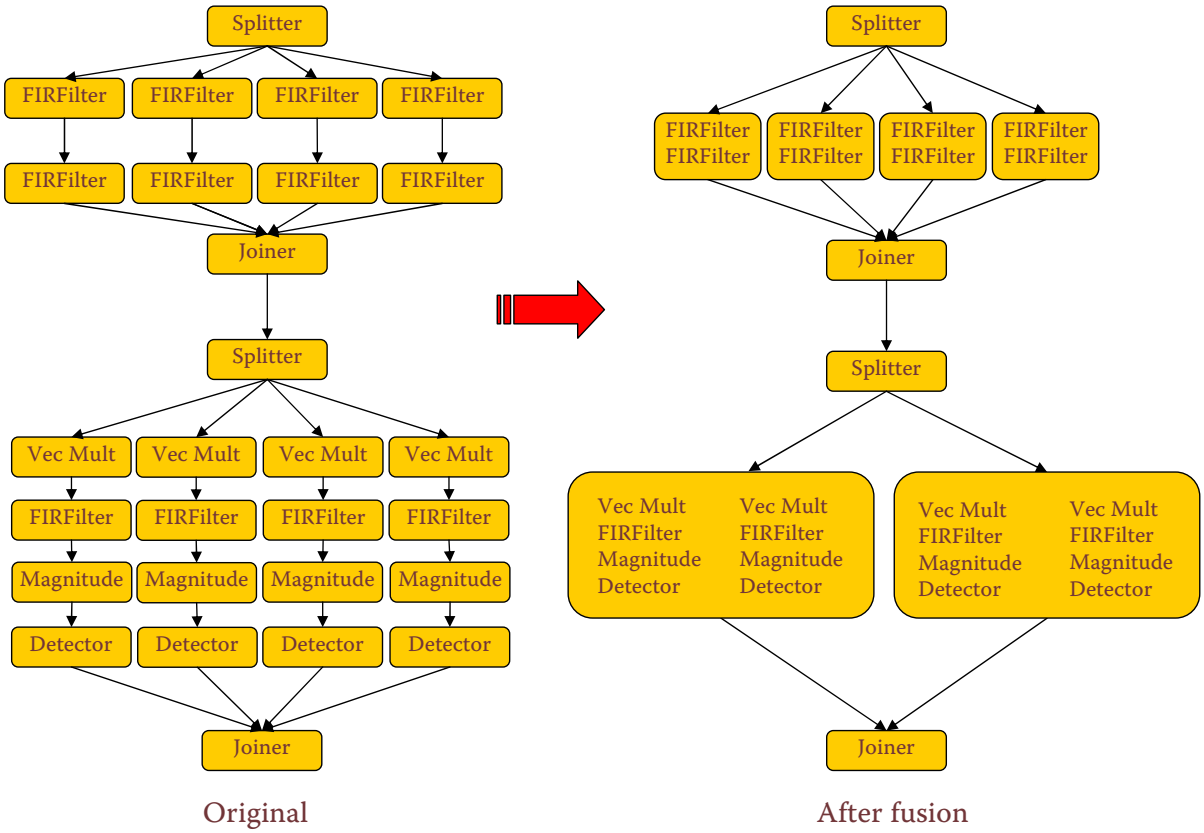


Figure 5-8: StreamIt Fusion transformation. The StreamIt compiler merges ("fusion") and splits ("fission") filters in order to partition the work evenly across the tiles.

transformations. Filter fusion, shown in Figure 5-8, merges together filters in order balance them. Fusion can be applied to nodes that are sequentially connected ("vertical fusion"), or to nodes that are connected through the `splitjoin` construct ("horizontal fusion"). Both cases are shown in Figure 5-8, which demonstrates the partitioner as it works to map the beam forming algorithm to eight tiles.

Filter fission, on the other hand, breaks a filter into multiple sub-filters. In the current StreamIt implementation, fission is performed under limited circumstances, such as for stateless filters, i.e., filters which do not maintain local state. Future research could apply the same algorithms found in Rawcc to parallelize individual filters.

At the end of the partitioning phase, the StreamIt compiler merges splitters into the upstream filter, and separated out joiners into their own node. Upon exiting from the partitioning phase, the stream graph now contains a number of nodes that is less than the number of tiles.

Assignment The next stage of the StreamIt compiler takes the filters and assigns each one to a tile. Joiners occupy their own tiles, while splitters are merged into the tile of the upstream filter. Each tile is assigned only one filter. The assignment phase chooses the mapping between filter

and tiles in way that reduces the communication distances between tiles, as well as the number of intersecting streams. Intersecting streams are undesirable because they lead to false synchronization between otherwise independent filters. To that end, the assignment algorithm performs a modified simulated annealing algorithm with a specialized `cost` function, which we describe below:

TYPES

e : (*pair* node-id node-id)	*stream graph edge pair (src, dest)*
eSet :	*set of edges in stream graph*
eWeight : e $\mapsto$ int	*number of values routed on stream graph edge per work fn execution*
a : node-id $\mapsto$ tile-id	*assignment of stream graph nodes to tiles*
p : list of tile-ids	*a path between two assigned nodes*

HELPER FUNCTIONS

(*path* e a) $\rightarrow$ list of tiles along XY dimension-ordered path from e.src to e.dst given a

(*sync* eSet a p) $\rightarrow |(\textit{range}\ \text{a}) \cap (\cup\ \text{p})| + \sum_{e \in eSet} |(\cup\ (\textit{path}\ \text{e a})) \cap (\cup\ \text{p})|$

MAIN FUNCTION

(*cost* eSet eWeight a) $\rightarrow \sum_{e \in eSet}$ (eWeight e) $\times$ |(*path* e a)| + 10 $\times$ (*sync* eSet a (path e a)))

The assignment phase assumes that routing is performed using XY dimension-ordered routing. The `cost` function, shown above, sums all of the inter-tile communication distances and uses the helper function `sync` to estimate interference with other streams. The `sync` function incorporates the number of active tiles that are crossed, and the number of intersections with other streams. As is evident from the 10x weighting of the `sync` function, the StreamIt compiler is generally more worried about inter-stream contention than routing distances.

The results of the application of the assignment phase to the beam forming stream graph are shown in Figure 5-9.

Scheduling and Code Generation After the nodes of the stream graph have been assigned to tiles, it remains to schedule and generate the code. Each StreamIt program has two phases that require scheduling and code generation: initialization and steady state. Steady state corresponds to the regime of execution where the filters in the pipeline are executing in parallel. The initialization code brings the program from the point where the pipeline is empty to the point where it is about to enter the steady state. In order to compute the schedules for the initialization and steady-state phases, the StreamIt compiler performs an internal, high-level simulation of the filters executing in the StreamIt graph. StreamIt employs a *push schedule* approach, wherein the schedule fires the furthest downstream filter in the graph that is ready to be fired[11]. The scheduler simulates and

[11] A filter is ready to fire if, among other things, it has received enough data that all `peek` commands can be processed.

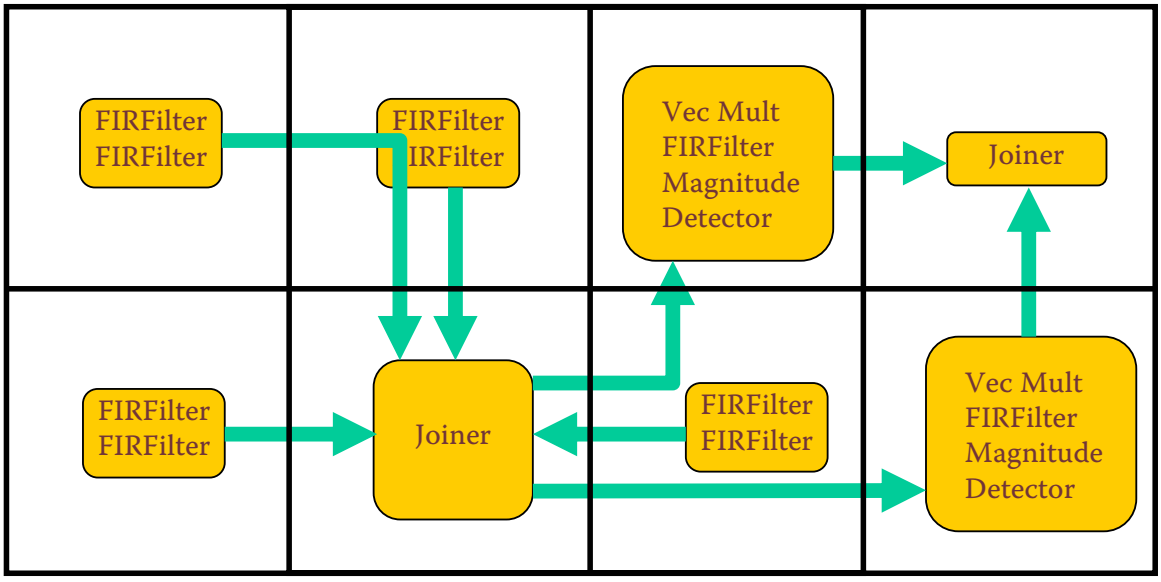


Figure 5-9: StreamIt filter assignment phase. The StreamIt compiler assigns filters to tiles, striving to minimize stream communication congestion and false synchronization between streams.

records the firing of filters until the executing stream graph repeats its state – that is, the number of elements in the buffers between filters is the same as at some previous time. This region of firings is the steady state. The firings that precede this region form the initialization phase of the program.

From there, it remains for the compiler to generate the actual code that corresponds to the schedule. This code includes compute processor code for filter execution and inter-tile SON code for communication between filters.

The StreamIt compiler emits C code for compute processors and uses GCC to compile it. Although most of the code is straightforward, the code generation of `peek`, `push`, and `pop` require special attention. First, each steady-state work function expects to receive *pop* elements before firing (for the first firing, it expects to receive *peek* elements). Thus, the beginning of the work function includes code which transfers these elements from the inter-tile SON to a software-managed circular buffer that is *peek* elements large. Then, `peek(index)` and `pop()` are translated into buffer accesses and updates. `push(value)` is translated into inline assembly code that injects data into the inter-tile SON. Finally, the corresponding switch processor code is generated to perform the inter-filter routes over the inter-tile SON.

5.5.3 StreamIt-on-Raw Results

In order to examine the performance of StreamIt applications compiled to Raw, a collection of benchmarks were written and run both on Raw and on the P3. Table 5.8 shows these benchmarks and the number of lines of code. The benchmarks are relatively small in size. This is due to the

Benchmark	Lines of Code	Cycles Per Output		Speedup vs P3 Time
		Raw 425 MHz	Pentium 3 600 MHz	
Beamformer	402	2074.5	15144	5.2
Bitonic Sort	266	11.6	57	3.5
FFT	169	16.4	110	4.8
Filterbank	156	305.6	4706	10.9
FIR	52	51.0	592	8.2
FMRadio	163	2614.0	23526	6.4
Geo. Mean				6.1

Table 5.8: StreamIt performance results.

lack of an existing code base for StreamIt and as well as the high expressiveness of the StreamIt language. These StreamIt benchmarks are refreshingly free of the ugliness often found in typical parallel programs. Furthermore, the StreamIt versions of these benchmarks on average exceed the performance of straight-forward hand-coded C versions on both P3 and single-tile Raw. This is due to aggressive signal-processing specific optimizations in the StreamIt compiler.

The results in Table 5.8 show the performance of the 425 MHz 16-tile Raw prototype relative to the Pentium 3 running at 600 MHz. Raw at 425 MHz has a geometric mean speedup of 6.1x versus the 600 MHz Pentium 3. A commercial version of Raw, implemented with the same level of design effort as the P3 and the same process would do even better. Because these applications do not experience many cache misses, the improvement would be approximately linear with frequency.

Benchmark	StreamIt on P3-600	StreamIt on Raw-425: n tiles				
		1	2	4	8	16
Beamformer	4.2	1.0	4.1	4.5	5.2	21.8
Bitonic Sort	1.8	1.0	1.9	3.4	4.7	6.3
FFT	1.6	1.0	1.6	3.5	4.8	7.3
Filterbank	2.1	1.0	3.3	3.3	11.0	23.4
FIR	3.7	1.0	2.3	5.5	12.9	30.1
FMRadio	1.7	1.0	1.0	1.2	4.0	10.9
Geo. Mean	2.3	1.0	2.1	3.2	6.4	14.1

Table 5.9: Speedup (in time) of StreamIt benchmarks relative to a 1-tile 425 MHz Raw configuration. From left, the columns indicate the StreamIt version on a 600 MHz P3, and on 425 MHz Raw configurations with one to 16 tiles.

We also examined the StreamIt performance as a function of tile count. Table 5.9 shows these results, normalized to performance on one tile. Many of the benchmarks continue to exhibit increasing speedups with increasing tile counts at 16 tiles. The left-most column shows the relative performance of the P3-600. As can be seen, the P3's performance lies somewhere between that of

two to four tiles, closer to two. Once again, Raw is more efficient per unit area than the P3 on this class of benchmarks. More details on these experiments can be found in [113].

5.5.4 Future Improvements

Examination of the StreamIt generated code reveals a number of areas of potential improvement. In many ways the usage model of the inter-tile SON is quite different for StreamIt than for Rawcc. StreamIt kernels tend to be courser grained and have independent control flow. They operate on data streams rather than individual operands. The performance results that we've attained demonstrate that the Raw is quite effective at performing both classes of applications. However, it is instructive to see what sources of overhead remain, especially in the management of operands that flow through inter-tile SON.

Space-Time Scheduling One of the possible improvements to the system is to integrate Rawcc-style parallelization into the filter fission passes. Individual filters can be processed by Rawcc to parallelize them across multiple tiles. This will allow for better load balancing, because the partitioner can start with finer-grained chunks. Additionally, Rawcc-style fine grained co-scheduling (instead of passing C code through to gcc) of compute processor and switch processor instructions could better overlap communication and computation.

Support of the Peek Construct Perhaps the greatest source of overhead is the buffering code that is inserted into each StreamIt filter. This code copies incoming inter-tile SON values into a local buffer so that the peek command can repeatedly select among a window of incoming values. This creates an effective receive-side occupancy for each value received. This is a typical operation in signal processing applications that are trying to extract information from a time-domain signal. Clearly, the existing inter-tile SON FIFO abstraction, which only allows the head value to be examined, does not perfectly match this usage pattern. This deficiency is similar but not the same to the challenges Rawcc experiences with arrival ordering of operands. However, StreamIt programs may peek hundreds of values into a stream, and may reuse a value hundreds of times, which is quite different from Rawcc usage, which typically reuses values a few times and does not have such a long window for most operands. On one hand, for such large numbers of operands, a typical microprocessor would also be forced to perform load and store operations. On the other hand, since the values are coming in via a stream, there is a sense that there is an opportunity to reduce this overhead. Certainly, one mechanism that would be useful is the enlarged switch processor register file ("ESPRF") proposed for improving Rawcc performance. However, this mechanism ties operands to register names, which would require a lot of code unrolling to implement the "sliding window" semantics of a queue.

In the StreamIt model, most of the peek window has already arrived at the destination. It is

merely waiting for the last *pop* values to arrive. These values, however, do not necessarily require sliding window semantics, since they are all dequeued and stored from the NIBs immediately when the filter executes.

Thus, in the common case where *peek* >> *pop*, the major performance overhead is not so much the storing of values to the buffer, but the repeated loading of values from the circular buffer. We could imagine implementing this functionality using a circular variant of a vector load. In this implementation, we would add a new NIB, ldsti to the processor pipeline. Then, a circular vector load could be issued to the load/store unit. These values would appear in the ldsti, to be accessed as register mapped values. Of course, this requires that values be stored in use-order, which is not uncommon in these classes of applications. This approach could also be used to load constant-valued taps used in local filters. Of course, more such input NIBs would allow more of these streams to be processed.

In the case where *peek* $\sim$ *pop*, the overhead of buffer stores becomes more important and the overhead of buffer loads less important. In this case, the ESPRF mechanism becomes more appropriate. However, if the number of values that need to be reordered exceeds the size of the ESPRF, it may still be necessary to buffer values in memory. In this case, a similar vector store mechanism could be implemented, with a similar register mapped NIB, ldsti to which instructions could write their outputs. However, since the values are actually coming in from csti and csti_2, to truly eliminate the overhead, we would either want to implement a NIB that connects the static router and the load-store unit, or to have some sort of way for the load store access to share the cst or csti_2 NIBs. One problem that would limit the performance benefits of this feature is if the ESPRF is insufficient to reorder the operands so that they can be stored in the order of their use via the vector store.

False Synchronization Another source of overhead which is somewhat specific to the use of a statically-routed SON is the issue of false synchronization. Crossing streams on the inter-tile SON create synchronization points between other-wise unrelated components of the program. If the compiler is unable to predict or compute the timing of filters, then there may be additional stalls that would not exist if the program were dynamically routed. This concern for false synchronization is evident in the *sync* function in the StreamIt's assignment phase, described in Section 5.5.2.

iWarp, another system optimized for streaming computation [38], addresses this system by allowing the route paths of streams to be determined at compile-time, but utilizes separate NIBs for separate streams. Values traveling through the network contain a tag that determines the NIB that they are placed into. At each hop, the values would be demultiplexed into the appropriate NIB. This system effectively removes the synchronization constraints. However, it carries with it the significant usability problem that it places a fixed upper-bound on the number of streams that can exist in the system at one time. Raw's SON, on the other-hand, interleaves streams, allowing an essentially unbounded number of streams in the system.

A recent experiment[35] measured the impact of eliminating false synchronization, load balancing and joiner overheads. It improved performance by 22%, which is relatively small in comparison to the overhead of buffering, and the magnitude of the improvements. Of course, as the system is optimized, this proportion may become more significant, but there is also the possibility of improving the communication scheduling algorithm in the compiler. At least theoretically, a set of filters with static data rates should be amenable to a static schedule.

An alternative is to use a dynamic transport SON that uses a receive-side hardware to demultiplex and reorder incoming messages, especially useful for `splitjoin` and `feedback` nodes. Generally, this will increase latency relative to a static transport SON; the latency generally will not impact performance except in short running filters, or filters with tight feedback loops. The use of a dynamic transport SON might also enable filters with variable rates. However, variable data rates are likely also to require filter migration in order to keep the system load balanced.

Another alternative is to provide a mechanism in the static transport SON by which certain streams can cross each other without incurring synchronization. For instance, a switch processor could be instructed to route any (or a certain number) incoming values from the east to the west. This essentially creates a local relaxation of the static ordering property. This will work for streams which cross but do not share channels. However, streams that share the same links ultimately must be synchronized in order to maintain the static ordering property. Ultimately, this may undermine the efficacy of such a mechanism.

5.6 Multi-Tile Performance on Hand-Coded Streaming Applications

We now transition from examining streaming programs that have been automatically parallelized to those that were hand-parallelized. Typically, these applications are mostly written in C, and are augmented with assembly language to program the switch processor and perhaps to optimize inner loops. Often the bulk of the execution time of these programs resides in a few loops, making the effort of hand-parallelization worthwhile. Ideally a compiler would be able to automatically produce code with the same level of quality as the hand-coded versions; continued compiler research will advance this goal. In the mean time, hand-parallelization for a tiled architecture remains a desirable alternative to implementing the same application in an FPGA or custom ASIC.

5.6.1 Linear Algebra Routines

Table 5.10 shows Raw's performance on a selection on linear algebra routines, implemented using the Stream Algorithms [47] pattern for tiled microprocessors. Stream Algorithms operate by streaming

Benchmark	Problem Size	MFlops on Raw	Speedup vs P3	
			Cycles	Time
Matrix Multiplication	256 x 256	6310	8.6	6.3
LU factorization	256 x 256	4300	12.9	9.2
Triangular solver	256 x 256	4910	12.2	8.6
QR factorization	256 x 256	5170	18.0	12.8
Convolution	256 x 16	4610	9.1	6.5

Table 5.10: Performance of linear algebra routines.

data from peripheral memories, processing it with the array of tiles, and streaming the data back into the peripheral memories. The tiles operate on the data as it comes in from the interconnect, using a small, bounded (with respect to the input size and number of tiles) amount of storage. Stream Algorithms achieve 100% compute efficiency asymptotically with increasing numbers of tiles.

With the exception of Convolution, we compare against the P3 running single precision Lapack (Linear Algebra Package). We use clapack version 3.0 [4] and a tuned BLAS implementation, ATLAS [124], version 3.4.2. We disassembled the ATLAS library to verify that it uses P3 SSE extensions appropriately to achieve high performance. Since Lapack does not provide a convolution, we compare against the Intel Integrated Performance Primitives (IPP).

As can be seen in Table 5.10, Raw performs significantly better than the P3 on these applications even with optimized P3 SSE code. Raw operates directly on values from the network and avoid loads and stores, thereby achieving higher utilization of parallel resources than the blocked code on the P3.

Raw is also able to reach its peak efficiency for much smaller data sizes than the P3. This is shown in Table 5-10 which shows the scaling trends for a Triangular Solver. Each line represents the MFLOPS performance relative to performance for each processor on the 2048x2048 configuration. Raw's speedup over the P3 increases as input sizes are decreased, because Raw achieves peak efficiency earlier.

5.6.2 The STREAM Benchmark

John McCalpin created the STREAM benchmark to measure sustainable memory bandwidth and the corresponding vector kernel computation rate [80]. This benchmark has been used to evaluate thousands of machines, including PCs and desktops as well as MPPs and other supercomputers.

The Raw implementation of the STREAMs benchmark is handcoded and employs the Raw-Streams configuration. To improve the quality of the comparison, the performance of the P3 was improved by adjusting compiler flags to use single precision SSE floating point. The Raw implementation employs 14 tiles and streams data between 14 compute processors and 14 memory ports through the static network. The floating point instructions on the compute processors access the network directly. For instance, the `add` portion of the benchmark is implemented by a unrolled

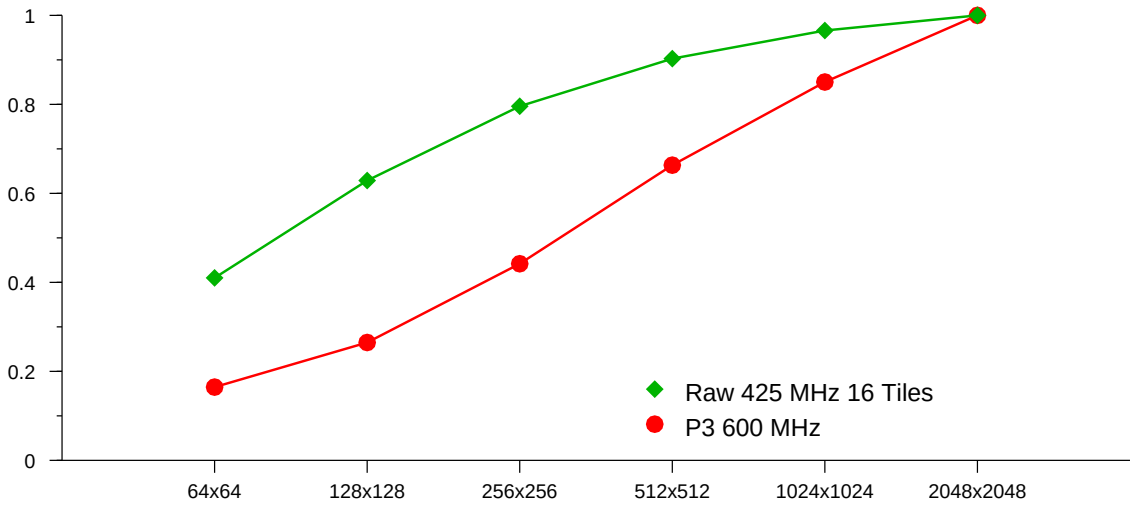


Figure 5-10: Triangle Solver Efficiency for Different Input Sizes. Performance is given relative to the MFLOPS with an input size of 2048x2048. In many linear algebra computations, larger input sizes are required in order to attain peak efficiency. The data shows that Raw is efficient even at smaller input sizes, in contrast to the P3.

	Bandwidth (GB/s)			
Problem Size	**P3-600**	**Raw-425**	**NEC SX-7**	**Raw/P3**
Copy	.567	47.6	35.1	84
Scale	.514	47.3	34.8	92
Add	.645	35.6	35.3	55
Scale & Add	.616	35.5	35.3	59

Table 5.11: Performance (by time) of STREAM benchmark.

loop containing "`add.s $csto, $csti2, $csti`" instructions. The local data memory is unused. Table 5.11 displays the results. Raw at 425 MHz is 55x-92x better than the P3 at 600 MHz. The table also includes the performance of STREAM on NEC SX-7 Supercomputer, which has the highest reported STREAM performance of any single-chip processor at the time the data was gathered. With the RawStreams configuration, Raw surpasses that performance. This attainment of this performance is achieved by taking advantage of three Raw architectural features: pin bandwidth, the ability to precisely route data values in and out of DRAMs with minimal overhead, and a good match of floating point and DRAM bandwidth.

5.6.3 Signal Processing Applications

Table 5.12 shows a selection of other signal processing applications implemented on Raw. Shown in the table are the application name, the memory system used, and the number of lines of code used in the two versions: the P3's version (in C), and Raw's version (in C, assembly, and/or other language). In some cases, the switch processor code had enough structure that it was most convenient to write

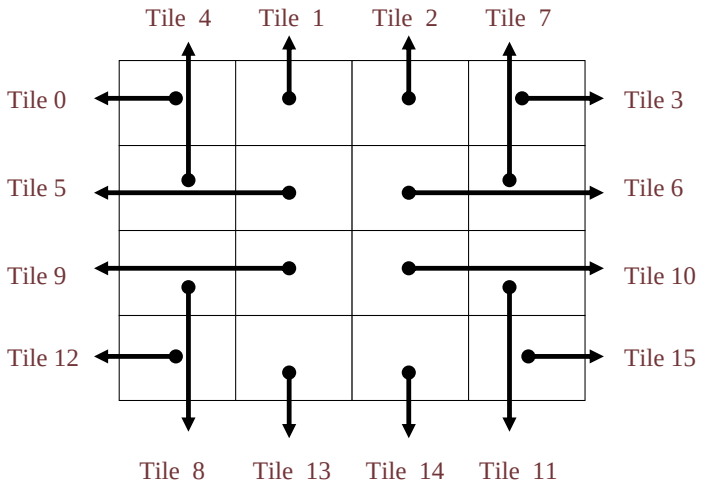


Figure 5-11: Tile-to-DRAM assignment for RawStreams configuration.

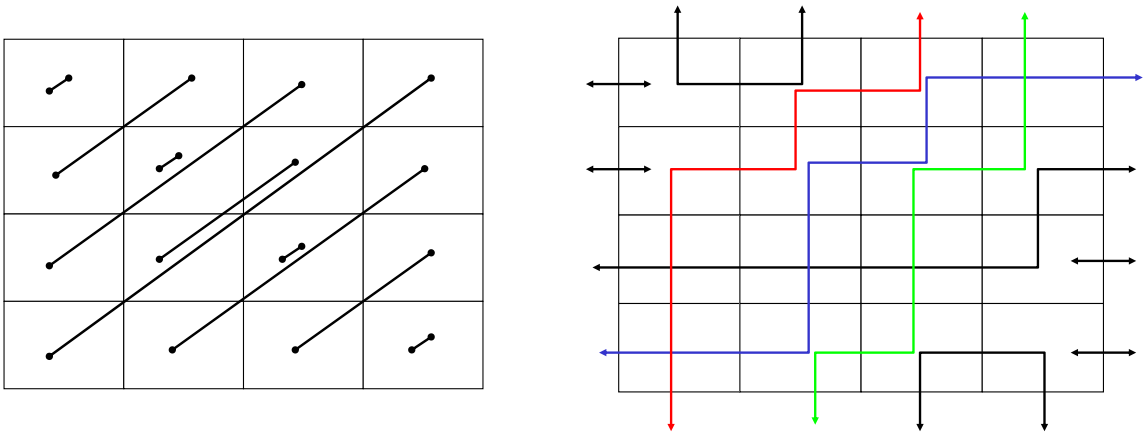


Figure 5-12: Corner Turn (Matrix Transpose) communication patterns. If the data were stored in the tiles, then the communication pattern would be that shown on the left. However, the array is so large that the bulk of it is stored in external DRAM. As an optimization, the static network is used to route data directly between the DRAMs that hold the data, resulting in the communication pattern shown on the right.

a "generator" program that generates the relevant code. In all cases, the performance gains far exceeded the increase in code size. This is in contrast to the P3, where the effort of improving performance (given the complexity of optimizing for the P3 microarchitecture) is high but the potential performance gains only moderate (given the limited issue width of the P3).

The Acoustic Beamforming application processes a stream of audio data coming in from an array of 1020 microphones. It uses this data to determine the sound signal that is emitted from a particular location. The FFT and FIR are hand-coded implementation of the classical algorithms.

CSLC, Beam Steering, and Corner Turn are signal processing algorithms selected and implemented in a third-party study of three architectures, including Raw [103]. All three of these benchmarks use the RawStreams memory configuration, shown in Figure 5-11, to maximize performance.

Beam Steering streams data in over the static network from external DRAM, processes it and streams it back. Thus, the communication patterns over the static network match those shown in Figure 5-11. Corner Turn implements a matrix transpose on a matrix that is too large to fit in the on-chip caches and is distributed as 16 sub-matrices across the DRAMs of the system. The implementation streams blocks of data over the static network from the source DRAMs to destination DRAMs, using the local memories of intermediate tiles to transpose the data blocks. The communication pattern is shown in 5-12. The implementation performs one load and one store for each data value transposed, and manages the simultaneous constraints of I/O ports, network bandwidth, and instruction bandwidth. CSLC has coarse-grained parallelism and does not require communication between tiles.

Benchmark	**Machine Configuration**	**Lines of Source**					**Cycles on Raw-425**	**Speedup vs P3-600 (Time)**
		P3	**Raw**					
		C	**C**	**Asm**	**Switch Asm**	**Generator**		
Acoustic Beamforming	RawStreams	1459	1709	166	0	435	7.83M	6.9
16-tap FIR	RawStreams	(lib)	0	34	116	0	548K	7.7
CSLC	RawPC	1809	2119	0	0	0	4.11M	12.0
Beam Steering	RawStreams	302	470	12	66	0	943K	46
Corner Turn	RawStreams	123	546	0	0	112	147K	174
512-pt Radix-2 FFT	RawPC						331K	3.3

Table 5.12: Performance of hand-written stream applications.

5.6.4 Bit-level Communications Applications

A final class of applications that were studied were bit-level communication applications [122, 123]. These applications were selected in an effort to understand how Raw performed on bit-level applications in comparison to FPGAs and ASICs. Like Raw, FPGAs and ASICs can also exploit fine-grained parallelism. However, Raw focuses on word-level computations. Thus, we expected that FPGAs and ASICs would hold an advantage for bit-level computations like the 802.11a convolutional encoder and the 8b/10b encoder.

Raw came unexpectedly close in performance, due in part to the anticipated benefits of direct I/O access, tile parallelism and fast communication. However, it was also due to the presence of rotate-and-mask bit-manipulation instructions (`rlm, rlmi, rlvm` (see Chapter B) that had been introduced to assist in bit-level operations. These instructions, combined with the use of 8-input lookup tables (which are equivalent to up to a large number of 4-input CLBs) reduced the impact of critical feedback loops in these computations. However, on the other hand, the ASIC and FPGA

	Problem Size	Lines of Code			Cycles on Raw	Speedup vs P3-600 Time		
		P3	Raw					
		C	C	Asm		Raw	FPGA	ASIC
802.11a ConvEnc	1024 bits	65	0	290	1048	7.8	6.8	24
	16408 bits	65	0	290	16408	12.7	11	38
	65536 bits	65	0	290	65560	23.2	20	68
8b/10b Encoder	1024 bytes	236	244	347	1054	5.8	3.9	12
	16408 bytes	236	244	347	16444	8.3	5.4	17
	65536 bytes	236	244	347	65695	14.1	9.1	29

Table 5.13: Performance of two bit-level applications: 802.11a Convolutional Encoder and 8b/10b Encoder. The hand coded Raw implementations are compared to reference sequential C implementations on the P3.

implementations generally occupied orders of magnitude less silicon area.

Table 5.13 compares the performance of Raw-425 (RawStreams memory configuration), an 180 nm SA-27E ASIC, and 180 nm Xilinx Virtex-II 3000-5 FPGA, relative to C written for a P3-600. The 180 ASIC was synthesized (but not placed) on the same ASIC process as Raw. For each benchmark, three problem sizes were used, selected to fit in the P3's three levels of memory hierarchy: the L1 cache, the L2 cache and DRAM. The input sequences are randomized. As is evident, the P3's performance relative to Raw drops because the P3 is pulling data across its memory hierarchy, rather than streaming it in directly from DRAM as in the case of Raw.

Table 5.14 shows the same applications, running in configuration that emulates a base-station processing many streams in parallel. For this test of throughput, we used versions of the algorithms that provided the most performance per tile, rather than the highest absolute performance.

Benchmark	Problem Size	Cycles on Raw	Speedup vs P3-600 Time
802.11a ConvEnc	16*64 bits	259	32
	16*1024 bits	4138	51
	16*4096 bits	16549	92
8b/10b Encoder	16*64 bytes	257	24
	16*1024 bytes	4097	33
	16*4096 bytes	16385	56

Table 5.14: Performance of two bit-level applications for 16 streams: 802.11a Convolutional Encoder and 8b/10b Encoder. This test simulates a possible workload for a base-station which processes multiple communication streams.

5.7 The Grain Size Question: Considering a 2-way Issue Compute Processor

One tension that arises in the design of a tile's compute processor is the question of *grain size*. How powerful should a tile be? We can employ a larger and more powerful tile, but have fewer of them, or we can have more simple tiles and have more of them. Programmatically, the idealized superscalar presents a desirable model because it offers free inter-instruction communication and few restrictions on instruction execution. However, implementation-wise, superscalars must expend transistors to overcome the challenges of multiple-issue - tracking branches when fetching multiple instructions per cycle, dealing with structural hazards involving register file ports, cache ports and functional units, and managing data dependences. Finding area- and frequency- efficient solutions to these challenges not only incurs transistor overhead but often results in constraints that deliver less than ideal performance transparency – for instance, using banking instead of multiporting, and imposing rules on the types of instructions that can issue together. For instance, the P3 does not allow two stores or two loads to execute in a cycle.

Considering a superscalar compute-processor However, a narrow-issue superscalar compute processor implementation is still an attractive possibility, because it would boost application performance on applications whose inter-instruction communication is too fine-grained to effectively utilize the inter-tile SON. Although an analysis of this could be arbitrarily detailed, we explore some of these issues here. To advance the discussion, we examine a 2-way issue compute-processor, because the performance advantages and efficiency of superscalars on Spec drops off quickly with higher issue rates.

Figure 5-13 shows the intra-tile SON of one such 2-way issue superscalar compute processor. The SON shown allows two instructions to be issued each cycle to different functional units. The number of functional units has not increased. In this example, we maintain the rate of operand delivery to csto, cgno and cmno to one per NIB per cycle.

The superscalar's intra-tile SON shares the same basic design and structure as the Raw compute processor SON, shown previously in Figure 3-3. However, it requires additional resources to support the ability to fetch and retire two instructions per cycle.

First, a more heavily ported 4R 2W (4-read ports, 2-write ports) register file is necessary to prevent register file structural hazards caused by the register file needs of the two instructions. In the IBM SA-27E process, a 4R 2W register file made from two 2R 2W register files is 2.9x the size of the 2R 1W register file in the Raw tile. Such a register file would have little impact on the existing tile design, because the register file occupies little area and poses little risk to the cycle time.

Additionally, a second operand pipeline is needed to manage the additional operands flowing in the system. This second operand pipeline is shown on the right hand side of the figure. The

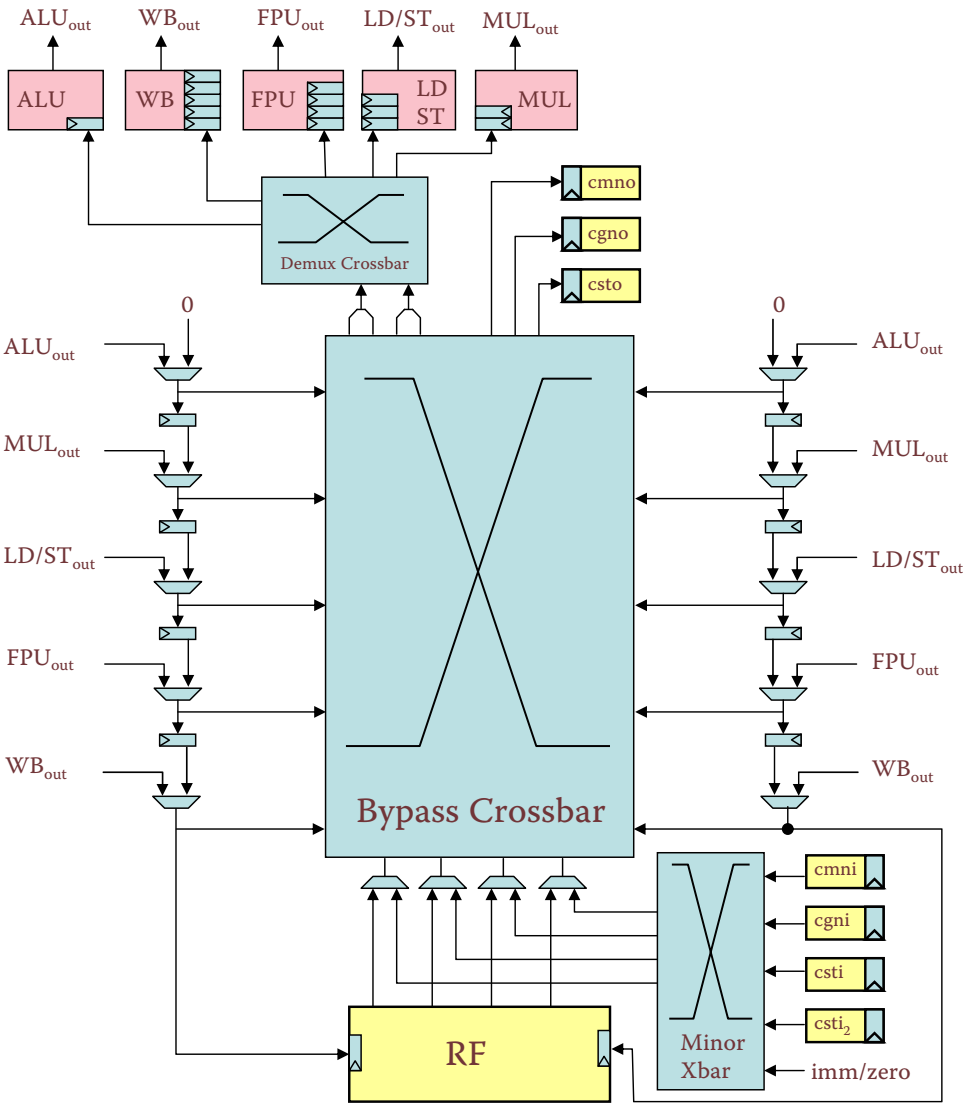


Figure 5-13: Intra-tile SON of a 2-issue superscalar tile. It performs bypassing and operand management on two instructions for each cycle. This version does not increase the number of functional units, but rather allows two instructions to be issued to different functional units each cycle.

operand pipeline gathers operands that are completing from the functional units, presents them to the bypass network for bypassing, and delivers them to the register file on instruction retirement.

The most significant impact on the intra-tile SON is the impact on the bypass crossbar, which went from being a 7-input, 5-output 32b crossbar to being a 14-input, 7-output 32b crossbar. This is actually less worse than it could be, because it allows only one operand to be delivered to csto, cgno and cmno per cycle. Assuming that crossbar area is related to the product of input and output ports, the bypass path area is 2.6x as large. Furthermore, the bypass paths are often on the critical path of the compute processor, so the additional delay through this structure could be a problem. One alternative to reduce the impact of this structure is to only allow long-latency

instructions to be paired with shorter latency ones, and to always assign the long-latency instruction to the second operand pipeline. Then, the first few stages of the second operand pipeline (and the corresponding inputs into the bypass network) could be eliminated. For instance, one bypass crossbar input could be eliminated by disallowing the simultaneous execution of ALU and multiply instructions. However, the decimation of crossbar inputs moves the design further from the goal of performance transparency, and bears the additional control complexity of tracking which instruction is the oldest or youngest at symmetric positions in the two operand pipelines.

The *demux crossbar* is necessary because each functional unit needs to choose between one of two issuing instructions to receive its operands from. This crossbar, although relatively small, is also on the critical path of the processor and has a high risk of impacting cycle time. If the functional units were duplicated rather than shared between the two operand pipelines, this demultiplexing crossbar would not be necessary.

Intra-tile SON Control and Fetch Unit Of course, there are also other parts of the system that will change. The control logic that manages the intra-tile SON will also have to be changed. However, most of these calculations can be performed soon after the instructions are fetched in the front end of the processor. Generally, front-end modifications (excepting the fetch unit) are relatively cheap because they can be pipelined, using branch prediction to mitigate performance effects.

The fetch unit would need more bandwidth to sustain delivery of two instructions per cycle. To smooth out performance artifacts due to cache line granularities and branch prediction bubbles, it would probably fetch four instructions at a time and buffer them up in a queue. A branch target buffer (BTB) would be essential for mitigating the cycle impact, by predicting the next address to fetch from.

Duplicating Functional Units Beyond these modifications lies the question of whether any of the functional units should be duplicated to allow instructions that use the same functional units to issue in parallel. The more duplicated functional units, the more close to the ideal superscalar model. In practice, functional duplication is expensive because it increases the area of the tile. Eventually, if everything is duplicated, with the overhead of superscalar execution, the one new tile likely to be larger than two original tiles. Generally, we look to duplicate the less-area intensive functional units, such as the ALU[12]. Duplicating the LD/ST unit is also desirable. However, it greatly complicates the data cache design, due to the interdependence of memory operations, and risks increasing cycle time. Such a LD/ST would have to be implemented using banking, because the cycle time and area (2x) penalty of a full 2-ported data cache are too high.

Cost Estimate By using the floorplan of the existing compute processor, we can estimate the

[12]Although, since the ALU is a one-cycle operation, it requires a maximum length operand pipeline, which requires the full-sized 14-by-7 bypass crossbar.

additional area cost of the 2-way issue superscalar. Given the general trend that structures for the 2-way superscalar are a little under 3x larger, we add overhead for wiring constraints and assume that the intra-tile SON, decode and fetch-related items (excluding SRAMs) triple in size. Including a smallish 512-entry BTB, we arrive at an estimated ∼20% increase in the area of a tile. The change in frequency is more difficult to predict. At this point, the addition of a simple ALU (∼2%) has neglible area and frequency impact.

Benchmark	**# Raw Tiles**	**Cycles on Raw-1I-1L** (600 MHz)	**Speedup Raw-2I-1L vs Raw-1I-1L** (600 MHz)	**Speedup Raw-2I-2L vs Raw-1I-1L** (600 MHz)	**Cache Miss Cycle % Raw-1I-1L** (600 MHz)
171.swim	1	1.49B	1.118	1.133	56%
172.mgrid	1	.263B	1.087	1.141	33%
173.applu	1	.359B	1.155	1.178	37%
177.mesa	1	2.42B	1.138	1.202	2%
183.equake	1	.922B	1.189	1.230	23%
188.ammp	1	9.17B	1.036	1.037	83%
301.apsi	1	1.12B	1.125	1.160	26%
Geometric Mean (SpecInt)			1.120	1.153	
175.vpr	1	2.70B	1.009	1.041	25%
181.mcf	1	5.75B	1.011	1.015	95%
197.parser	1	7.14B	1.084	1.096	48%
256.bzip2	1	3.49B	1.107	1.123	43%
300.twolf	1	2.23B	1.039	1.025	46%
Geometric Mean (SpecFP)			1.049	1.059	
Geometric Mean (All)			1.090	1.113	

Table 5.15: Performance of SPEC2000 programs on a 2-way issue superscalar compute processor. Raw-1I-1L is the baseline single-issue Raw compute processor, running at 600 MHz. Raw-2I-1L is a two-way issue compute processor that can issue two instructions per cycle, but only one load/store per cycle. Raw-2I-2L is a two-way issue computer processor that can issue two instructions per cycle, including two load/stores per cycle. In all cases, only one floating point divide and one integer divide can be sustained at a time. The estimated percentage of total cycles spent on cache misses (data and instruction) is shown in the last column.

Performance Estimate To estimate the performance improvement of two-way issue, we simulated the benchmarks in Table 5.4, on an augmented version of the BTL simulator. The simulation executes two instructions, of any type, per cycle. However, at most one branch will be executed per cycle. It has an idealized fetch unit, which can fetch instructions from different cache blocks in the same cycle, with the same 3-cycle mispredict penalty as in the Raw hardware. The application was compiled with gcc 3.3, which was modified to use the issue-width and number of load/store units as an input to the scheduler. Disassembly of the output confirmed that the intra-basic block scheduling was competitive with what could be achieved by hand. We simulated both a two-way issue compute processor with 1 load-store unit ("2I-1L") and a two-way issue compute processor

with 2 load/store units ("2I-2L"). As such, these results present a reasonable upper estimate of 2-way in-order performance. The results, shown in Table 5.15 suggest that the performance improvement (over the baseline Raw single-issue compute processor, 1I-1L) is around 11.3% for a 2I-2L and a 9.0% for a 2I-1L. A 1995 study [118] reports an improvement of 14% for a non-duplicated 2-way issue in-order superscalar, with an additional 7% for a second ALU[13]. The greater cost of cache misses (also shown in Table 5.15) relative to 1995-era technology is a likely culprit for the performance decrease. The implementation of virtual-caching using other tiles could yield a greater overall improvement on single-threaded performance.

Overall, a 2-way issue compute processor would add complexity to the system, and is likely to decrease efficiency slightly in terms of Spec throughput per mm^2. On the other hand, the efficiency drop is relatively low. Assuming that it can be implemented without a cycle time penalty, the improvement in peak performance and corresponding increases in hand-coded and streaming applications are probably an area-performance win – up to twice the performance at ∼22% cost[14].

Inter-tile SON System Balance Along with the decision to increase the grain size of the tile comes the need to rework the inter-tile properties of the system. First, bigger tiles will likely have larger inter-tile communication costs. Further, the tiles are now capable of generating more operands per cycle. Should the inter-tile network bandwidth be scaled up accordingly? Of course, if the bandwidth is scaled up, then the tile size gets even bigger. Clearly, it is a slippery slope.

The general question of the appropriate balance of resources and grain size is an interesting area of research. Some early work on this subject was performed by Moritz et al [84].

5.8 Conclusion

In this chapter, we evaluated the performance of the 16-tile Raw microprocessor versus a Pentium 3 implemented in the same process generation. Surprisingly, a single Raw tile, implemented in ASIC technology, comes within a factor of 2x of the P3, even though the Raw tile occupies only 16% of the area. Implemented in the same process, and with a more efficient caching system, the performance gap would be even less. This is a testament to the limitations of the microarchitectural approach to scaling microprocessor performance.

As a server-farm-on-a-chip, Raw exceeds the performance of the Pentium 3 by a factor of 7.36x, or a factor of 2.3x when normalized for area. On automatically parallelized codes, Raw beats the P3 by a factor of 2.23x. On streaming programs parallelized over the SON using StreamIt, Raw beats

[13]The paper also reports improvements of 46% and 64% when a rescheduling profile-based compiler is employed, with aggressive cross-block scheduling, on an architecture with non-blocking loads. Although profile-based compilation is used only infrequently for shipping software today, we should not discount the possibility that a better scheduling backend could allow greater benefits for 2-way issue.

[14]The ability to perform load and store instructions in parallel with FPU instructions, which is possible even with the non-duplicated superscalar, seems particularly useful for streaming applications.

the P3 by a factor of 6.1x. On hand-coded applications, the performance gains are even greater. Only modest benefits on single-threaded non-parallelized programs are observed when moving to a 2-way issue compute processor (9%), likely because of caching. Employing virtual caching could improve performance.

Overall, Raw excels over the P3 on those programs that have modest to large amounts of parallelism. As the support systems and compilation environments for tiled architectures mature, and as the amount of silicon area and number of tiles realizeable increases, we can expect that the scope of programs that achieve speedup will broaden, that the effort of parallelization will continue to fall, and that the performance benefits of tiled architectures over conventional superscalars will continue to increase.

Chapter 6

Related Work

In this section, we examine the relationship between the research described in this thesis and other efforts in three related research areas: microprocessor scalability, tiled microprocessors, and scalar operand networks.

6.1 Microprocessor Scalability

The challenge of microprocessor scalability given modern technology constraints (e.g., billion transistors on a single chip) is widespread and there are both industry and academic efforts.

Industrial superscalar efforts have striven to maintain backwards compatibility while increasing the issue width. However, the sustained issue rates of recent leading-edge designs like Alpha 21264 [59] (4-issue), Pentium 4 [44] (3-issue), Core 2 Duo (4-issue), and Power4 [114] (4-issue plus 1 branch) have stopped increasing, and hover around 3-5 issue. The Intel Itanium 2 [85, 81] is 6-issue but runs at half the frequency of the 3-issue Pentium 4 in the same lithography generation, clearly pushing against the limits of frequency scalability.

6.1.1 Decentralized Superscalars

A number of recent academic publications have explored approaches to extending the lifetime of superscalar microprocessor designs through decentralization – such as Kim's ILDP [61], Palacharla's Complexity Effective Superscalar Processors [91], Tseng's RingScalar [119], and Farkas's Multicluster Architecture [33]. Overall, these research efforts seek to extend the scalability of existing designs by reducing over-provisioned capabilities (such as widely-ported, centralized register files) that are not used in average-case programs. These ad-hoc approaches, while enabling backwards compatibility and increasing overall average performance (usually expressed as "a small reduction in IPC versus an ideal machine of the same width"), tend to decrease the performance transparency of the system

and increase the complexity of the microarchitecture. They offer limited design adaptivity and do not present a long term solution to the scalability problem. Tiled microprocessors, on the other hand, are regular and offer performance transparency and scalability at the cost of a richer (and non-backwards compatible) architectural interface.

6.1.2 Non-Tiled Parallel Microprocessors

In contrast to approaches which try to extend the life of superscalar designs, some recent projects have proposed alternatives to superscalar designs in order to utilize large quantities of VLSI resources. In this subsection, we examine those approaches which are generally not considered to be tiled, such as VIRAM [65], CODE [64], Stanford Imagine [93, 57], Tarantula [31], and GARP [42].

The VIRAM project proposed that large portions of the die area be dedicated to on-chip DRAM banks, replacing the traditional hierarchies of caches. The relatively long latency of the DRAM banks was to be hidden through the use of a vector architecture. Unlike the Raw architecture, VIRAM does not directly address the issue of scalability, instead focusing on consuming a fixed number – one billion – of transistors. A follow-on proposal, CODE, examines vector scalability issues, and in a fashion that parallels decentralized superscalar research efforts, applies decentralization techniques to vector ISAs, including the use of an on-chip communication network between clusters. CODE's scalability is ultimately bounded by its centralized instruction fetch mechanism and restrictive execution model due to its requirement that parallel computations be expressed as vector operations. Tiled architectures, on the other hand, can exploit any form of parallelism using either the SON or the dynamic transport networks.

The Stanford Imagine processor proposes an architecture which might be described as a SIMD-of-VLIWs. It includes the Stream Register File as a staging mechanism for streaming data. Like CODE and VIRAM, its focus is on stream and vector computations, and it employs a centralized fetch unit, which restricts its applicability for parallel models other than streaming or vector.

In many cases some of the previous projects could be scaled by composing systems out of multiple such vector or streaming processors. Tiled microprocessors benefit by offering a smooth continuum of communication costs (as measured by the 5-tuple), exposed through a uniform programming model as the system is scaled up. This simplifies the compilation model versus systems which are composed out of hierarchical out of components that use different modalities of communication at different grains (e.g., vector machines connected by shared memory or message passing).

The Tarantula research proposes a heterogeneous architecture composed of a wide-issue superscalar processor coupled with a vector unit. This allows it to exploit both ILP and vector computations. The GARP architecture integrated bit-level (FPGA-style) and ILP-style processing components. The logical limit of these forms of architecture is to compose processors out of many heterogeneous entities, each specialized for a class of processing. Tiled architectures, in contrast, seek

to reduce complexity (in the architecture, microarchitecture and compiler) by providing *modeless* primitives mechanisms (e.g., the scalar operand network and the dynamic transport network) that can address many types of computations rather than many specialized computational subsystems, each specifically designed for a particular class.

6.1.3 Classical Multiprocessors

A number of late 80's and early 90's parallel machines strived for and attained many of the same elements of physical scalability as the ATM and Raw. However, unlike tiled microprocessors, these systems could not offer low-latency, low-occupancy communication (i.e., Criterion 5: Efficient Internode Operation-Operand Matching) like that provided by the Scalar Operand Network found in tiled microprocessors. These systems include MIMD message passing systems like the Connection Machine CM-5 [73], and J-Machine [88], cache-coherent distributed shared memory systems like DASH [74], DMA/Rendezvous-based multicomputers like Transputer [1] and systolic systems like Warp [5] and iWarp [38]. The conventional wisdom that emerged from these efforts was that massively parallel systems were low-volume and only economically feasible if they were constructed out of commercially-available off-the-shelf microprocessors (for instance, as in the Cray T3E [98]). Now that it is possible to integrate many processors on a single die, most processors of the future are likely to be parallel processors; thus parallel processing is now a high-volume industry.

As a result, the requirement that parallel processors must be constructed out of largely unmodified superscalar processors is less important. This thesis proposes tiled microprocessors as a design choice in this new engineering regime. The principal difference between tiled microprocessors and hypothetical on-chip versions of these earlier parallel systems (including current-day shared-memory multicore microprocessors) is the presence of a scalar operand network to support low-latency and low-occupancy communication between nodes. Of these early systems, iWarp bears the most similarity and will be addressed in further depth.

6.1.4 Dataflow Machines

Some of the earliest investigations into distributed, large-scale, fine-grained ILP-style program execution can be found in the dataflow work of the 1970's and 1980's. Of these, one of the earliest was Jack Dennis's Static Dataflow [26]. The dataflow model can be viewed as an extension of Tomasulo's execution model [117] in that it also permits the instructions of a program to execute dynamically as their inputs arrive (i.e., dynamic ordering of instructions as in the AsTrO taxonomy). The dataflow work logically extends Tomasulo's algorithm by 1) eliminating the centralized front-end (e.g., PC, and issue logic), 2) allowing for many functional units to be used simultaneously, 3) replacing the CDB with a more general SON, 4) virtualizing the reservation stations – effectively creating one or more reservation stations per static instruction – and 5) replacing operand broadcasts with point-

to-point operand communication. Von Neumann-style tiled microprocessors, as discussed in this thesis, do not necessarily support out-of-order execution, but do share with the dataflow work the elimination of front-end scalability problems (but through the duplication of the front-ends rather than through elimination of program counters), the usage of many parallel functional units, the presence of a more parallel SON, and the employment of point-to-point operand communication.

Generally, tiled microprocessors inherit many of the scalability goals of the dataflow machines, but adhere more closely to conventional imperative programming languages, which 1) makes them more applicable with mainstream programming, 2) enables the compiler to control the execution of the program more tightly and 3) exploits the locality that is often implicit in imperative program representations. Von Neumann-style tiled microprocessor inter- and intra- tile SON latencies are generally lower than the corresponding dataflow latencies because they do not have large associative wakeup windows. These lower latencies allow tiled microprocessors to perform better in programs for which available parallelism is not much larger than the number of ALUs. On the other hand, like tiled microprocessors, the dataflow machines typically enjoy zero send and receive occupancy costs. Some follow-on dataflow research, such as [48], attempts to reduce the latency impact of dataflow by coarsening the model and employing threads of instructions with explicit communication instructions for inter-thread communication. This bears some resemblance to some of the recent dynamically-ordered tiled microprocessor work (especially TRIPS, and Wavescalar), which are dynamic but try to allocate dependent instructions on the same node to reduce latency. However, these more recent efforts are able to support low-cost back-to-back communication of dependent instructions at the microarchitectural rather than architectural level, which retains some of the cleanliness of the earlier dataflow models.

Both tiled microprocessors and dataflow processors need to attain *the seven criteria for physical scalability* described in Section 1.2.1 in order to be effective. In many ways, tiled microprocessor research distinguishes itself from previous dataflow efforts in the degree to which the fulfillment of these criterias is facilitated through at the architectural level rather than at the implementation level. Tiled microprocessors directly and explicitly address these requirements through the application of tiling and full distribution of architectural structures.

The WaveScalar tiled dataflow processor re-examines dataflow processors in the context of tiled microprocessors executing von-Neumann style programs. In particular, WaveScalar extends earlier dataflow work by adding support for von-Neumann style codes and by allowing back-to-back instructions at the same node to execute without paying a penalty for accessing the instruction window (e.g. 0-cycle SON for local bypass). A thorough examination of dataflow-related work can be found in [104].

6.1.5 Tiled or Partially-Tiled Microprocessors

Over the last few years, a number of research efforts have begun to examine tiled or partially-tiled microprocessors. These include Multiscalar [101], TRIPS [87, 96], WaveScalar [105], Scale [66], Synchroscalar [89] and Smart Memories [78].

Multiscalar might be considered to be one of the earliest precursors of a partially-tiled microprocessor. It featured a one-dimensional Scalar Operand Network that was used to forward operand values between ALUs.

TRIPS explores the benefits of a partially-tiled design in an effort to decrease the 5-tuple cost among a subset of ALUs. It employs an SDD (Static assignment, Dynamic transport and Dynamic ordering) Scalar Operand Network (according to the AsTrO taxonomy) to tolerate memory latency. Unlike the ATM and the Raw microprocessor, TRIPS has a centralized fetch unit, and a centralized load/store memory system, which makes it only partially tiled and thus only partially scalable. Beyond the 16 ALUs of a single core, TRIPS programs must employ shared memory or message passing for communication. WaveScalar explores the implications of dataflow execution in a fully-tiled design that employs a hierarchical SDD SON.

Scale, like TRIPS, is a partially-tiled microprocessor design. It employs the Vector-Thread ISA as the interface to an SSS SON-connected array of ALUs. At the cost of routing flexibility, and support for floating point, it offers extremely dense integer-only operation for low-power applications. The Synchroscalar architecture is a partially tiled architecture with a reconfigurable static SON that supports SIMD execution and voltage scaling on columns of tiles. The picoChip [30] architecture is special-purpose system composed of 430 16-bit MIMD processors for use in 3G base-stations. It employs a time-multiplexed, circuit-switched SON for communication. As a result of the lengthy connections in the systems formed by the circuit-switching, the frequency of the system is relatively low (160 MHz in 130 nm). The use of circuit switching provides low-latency communication but raises both frequency scalability and programmability concerns.

6.2 Scalar Operand Networks

Tiled microprocessors are distinguished from parallel systems by the presence of a scalar operand network, which provides low-latency, low-occupancy communication between dependent instructions. In this section, we examine related research along this dimension.

Seitz's 1983-era MOSAIC systems [75] can be considered to contain an early instance of an SSS scalar operand network. MOSAIC systems were composed of a number of MOSAIC chips, each containing a processor that had connections to four ports; each port was connected to one input and one output pin[1]. The pins could be connected at the system level to form arbitrary topologies. The

[1]MOSAIC employed bit-serial links due to a low number of pins in the package.

instruction set, predating the RISC efforts, offered a number of addressing modes, include those that allowed direct access to the input and output ports in addition to standard memory and register address modes. Move instructions could target one input and one output port, while computation instructions could employ at most one input or output port. This tightly coupled network access is quite similar to Raw's register-mapped network interface. Unlike Raw, MOSAIC systems did not employ separate switch processors; the compute processor was directly connected to its neighbor. Like Raw's SON, the SON offered blocking semantics, so if an output or input port were blocked, the instruction would stall. Tiled microprocessors such as Raw deploy MOSAIC-style SONs in the context of pipelined microprocessors that 1) implement complete memory systems with caching, 2) utilize generalize transport networks for more dynamic communication modes that can efficiently support non-statically analyzable communication patterns, and 3) provide support for automatic parallelization.

Corporaal's TTA [19, 20, 56] was an early effort that sought to address microprocessor register file scalability issues by making SON interconnect explicit, and allowing operand routing to occur directly between functional units. Tiled microprocessors inherit some of these basic ideas, but consider the challenge of extending scalability to all parts of the system, including the SON itself. Sankaralingam's Routed Inter-ALU Networks [95] examines tradeoffs between routed and broadcast SONs.

Both NuMesh [99] and iWarp [38] support fast, statically orchestrated communication, but focus on stream communication. In both cases, the architecture can switch between a fixed number of hardware determined communication patterns (in the form of streams) very quickly. The diversity of communication patterns is bounded by the number of hardware buffers (NuMesh) or logical channels (iWarp) that is contained in a node. In these systems, each stream is given a dedicated set of hardware buffers which serves to decouple the streams. In contrast, in the Raw system, there is only one set of buffers for all communication patterns; however, all communication patterns (and use of the buffers) is scheduled by the compiler, and specified collectively by the router instruction streams. As a result, there is no practical architectural limit on the number of communication patterns that can be supported; however it does require that the compiler be able to determine a statically ordered[2] communication schedule for all communications that cross. As a result, the architecture can cycle through new communication patterns very easily, which is necessary both for control-flow intensive code and for effective exploitation of the relatively unstructured communication patterns in ILP programs.

[2]Note that the routers support flow control and will stall if an expected words has not arrive. As a result, the schedule does not have to be cycle-accurate. This is important for handling unpredictable events like cache misses and interrupts.

6.3 Conclusion

Tiled microprocessor research efforts are positioned at the intersection of three concerns – scaling microprocessor systems, building parallel microprocessors that support versatile programming models – including steams, ILP, message passing and shared memory – and the desire to provide low-latency, low-occupancy communication through scalar operand networks.

Chapter 7

Conclusion

This thesis presents a new class of microprocessor design, the tiled microprocessor, which seeks to address the scalability concerns caused by advances in VLSI fabrication technology. Toward the goal of understanding the issue of scalability in the context of microprocessors, we formalized *seven criteria* for physically scalable microprocessors. These criteria included *frequency scalability*, *bandwidth scalability*, *usage-proportional resource latencies*, *exploitation of latency*, *efficient operation-operand matching*, *deadlock and starvation management*, and *support for exceptional events*.

In Chapter 2, we explored the ramifications of meeting these seven criteria through an archetypal tiled microprocessor, the ATM. In Chapter 3 and Appendices A and B, we turned the abstract conceptualization of the ATM into a concrete architecture, Raw, which is an 180 nm VLSI implementation of a tiled microprocessor.

The results of this research are compelling. With a team of a handful of graduate students, we were able to design and build a tiled microprocessor prototype (described in Chapter 4) that meets the seven criteria and scaled to larger issue widths (16-issue) than any current-day commercial superscalar. Although Raw has over 2x as many as the P4's 42 million transistors, the P4 required 30x times as many lines of Verilog, 100x the design team size, had 50x as many pre-tapeout bugs, and 33x times as many post-tapeout bugs. We believe this speaks strongly as to the benefits of tiled microprocessors when it comes to reducing complexity, design and verification cost, and risk of product recalls.

After describing the Raw artifact and surrounding systems, this thesis continued on to evaluation and performance analysis. In Chapter 5, we examined and analyzed Raw's performance on a variety of application classes, including ILP, stream, bit-level and server-farm-on-a-chip. In comparison to a Pentium 3 implemented in the same process generation, Raw's performance, facilitated by the presence of the Scalar Operand Network (SON), demonstrates its versatility across both sequential and parallel codes.

To a great extent, the recent widespread industrial adoption of multicore processors is a testament to the merits of our physically scalable approach. However, these multicore processors lack efficient inter-tile operation-operand matching and leave performance on the table as long as they do not possess a viable inter-tile Scalar Operand Network.

The Scalar Operand Network is a major contribution of this work. We examined this class of network along several axes. We introduced the 5-tuple performance metric, we implemented a high-frequency <0,0,1,2,0> network as a proof of concept, and we introduced the AsTrO taxonomy for categorizing them. It is our belief that competitive scalable microprocessor designs must possess inter-tile SONs. We hope that manufacturers will realize the compelling opportunities provided by low-latency, low-occupancy scalar transport and deploy them into their existing product lines in the not-too-distant future.

Appendix A

Compiler Writer's and Assembly Language Programmer's View of the Raw Architecture

This appendix describes the Raw architecture from the compiler-writer's and assembly language programmer's perspective. This section may be useful in resolving questions that arise in the reader's mind as they read the thesis. Some material from Chapter 3 is repeated so that this appendix is stand-alone. Encodings and instruction set specifics are given in Appendix B.

Every implementation of the Raw architecture will have a number of parameters that are implementation specific, such as the number of tiles and I/O ports. In some cases, we include the parameters for the Raw microprocessor in this description so as to provide a complete picture.

A.1 Architectural Overview

The Raw architecture divides the usable silicon area into a mesh array of identical *tiles*. Each of these tiles is connected to its neighbors via four point-to-point, pipelined on-chip mesh inter-tile *networks*. Two of these networks form part of the scalar operand network of the processor. At the periphery of the mesh, i.e, the edge of the VLSI chip, *I/O ports* connect the inter-tile network links to the pins. Figure A-1 shows the Raw microprocessor, which is a 16 tile, 16 I/O port implementation of the Raw microprocessor. Two pairs of these I/O ports share pins.

A.1.1 The Raw Tile

Each Raw tile contains a *compute processor*, one *static router*, and two *dynamic routers*. Figure A-2 shows the locations of these components in a Raw tile. The compute processor uses a 32-bit MIPS-style instruction set, with a number of modifications that make it more suitable for tile-based processing. The static router is part of the Raw microprocessor's scalar operand network, and

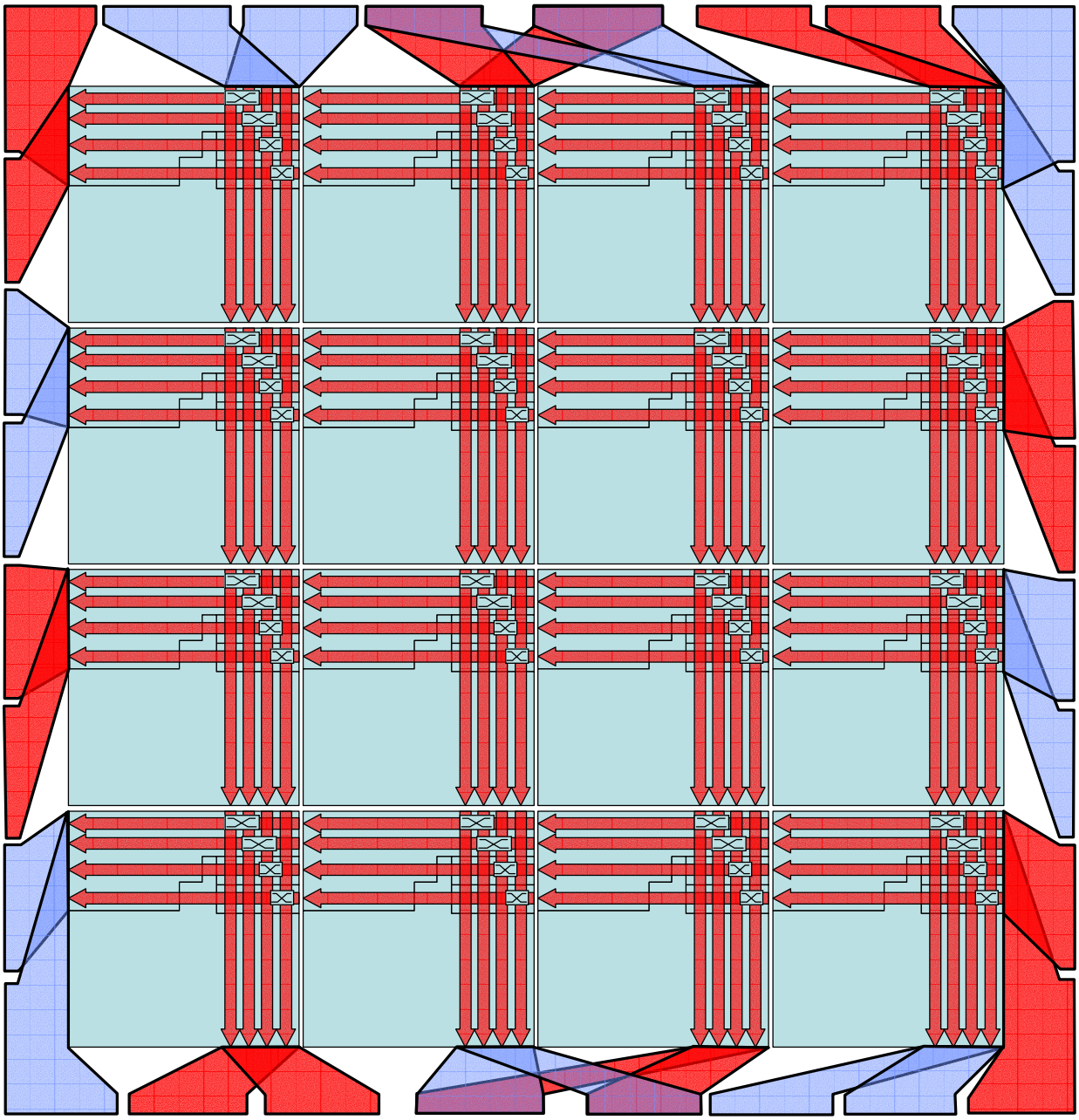


Figure A-1: The Raw Microprocessor. The diagram shows the layout of the 16-tile, 16 I/O port Raw microprocessor on a VLSI die. The tiles, the I/O ports, and the networks are all visible. The sixteen replicated squares are the tiles. The arrows indicate the location of the network wires for the four physical networks. Each arrow corresponds to 34 wires running in each direction. The pin connections are located around the perimeter of the die. The shaded regions emanating from the tiles correspond to I/O ports; they show the connection from each edge tile's network links to the pins of an I/O port. Each I/O port is full duplex and has two sets of pins; 37 pins for incoming data, and 37 pins for outgoing data. The ports are shaded in alternating colors for better visibility. Note that two pairs of I/O ports share pins.

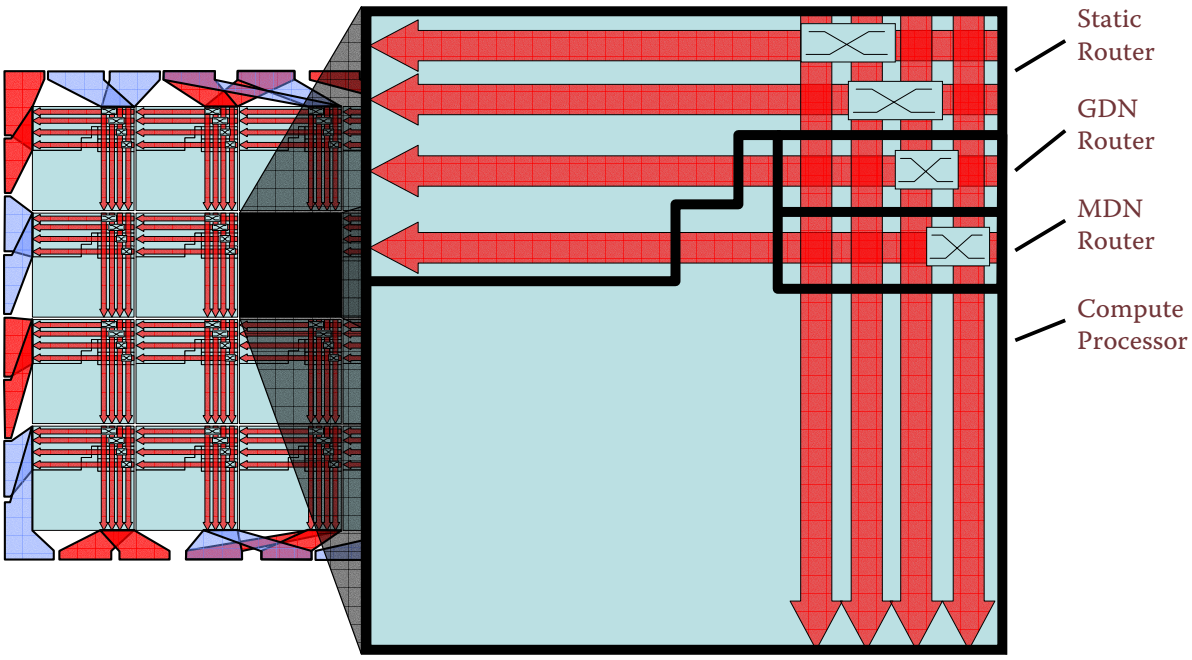


Figure A-2: Closer view of Raw Tile. The major components of the tile are the Static Router, the two dynamic routers (GDN and MDN) and the Compute Processor. The approximate positions and sizes of the crossbars, one per network per tile, are indicated using rectangles with X's on them.

manages the scalar operands flowing over two of the inter-tile networks (the "static networks"). The two dynamic routers, called the GDN router and the MDN router, respectively, are dimension-ordered, wormhole-routed [22] dynamic routers and each individually manage one of the remaining two inter-tile networks (the "dynamic" networks).

Compute Processor Each compute processor contains an 8-stage single-issue in-order 32-bit pipeline, a 32-bit ALU, a 32-bit pipelined single precision FPU, a 32-KB data cache, and a 32-KB software-managed[1] instruction cache. The compute processor uses a register-mapped interface to communicate with the three routers, although some status and configuration state is accessed via a special purpose register (SPR) interface.

Dynamic Routers The dynamic routers are responsible for managing inter-tile and off-chip communication that can not be predetermined at compile time. The two dynamic routers have identical architectures, shown in Figure A-3, but the networks they control have different purposes. The general dynamic network ("GDN") is for user-level messaging while the memory dynamic network ("MDN") is for system-level traffic, including memory and I/O traffic.

Figures A-3 shows the architecture of the dynamic routers. Data words arriving from the north,

[1]The software-managed instruction cache is an orthogonal piece of research performed in the Raw group. For the purposes of this thesis, it is largely valid to assume that a hardware instruction cache is being used. Indeed, some of the results presented in later sections employ a simulated hardware instruction cache in order to better normalize results with the current state of the art.

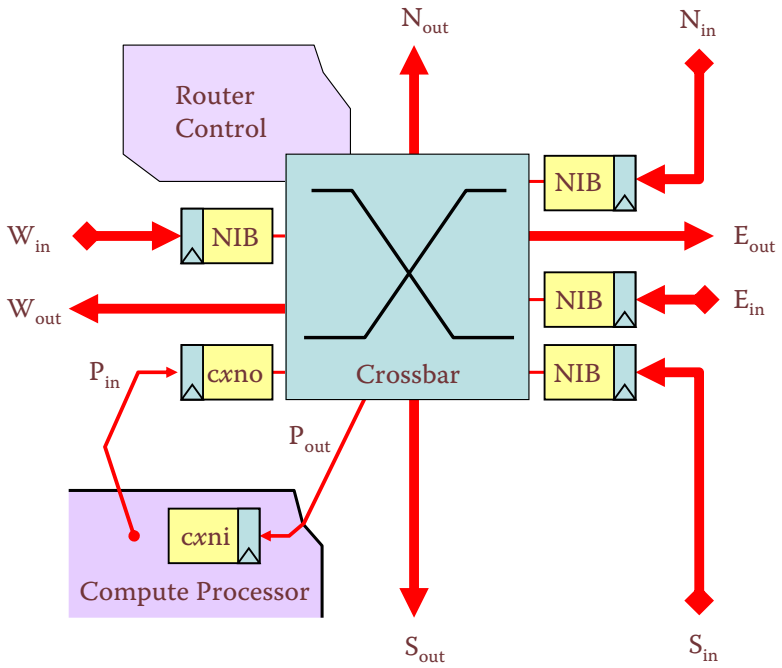


Figure A-3: Basic architecture of a dynamic router. Each dynamic router has its own set of inter-tile network links, NIBs and its own crossbar and control. The NIB marked cxno corresponds to the cgno and cmno NIBs on the GDN and MDN, respectively. The NIB marked cxni corresponds to cgni and cmni.

east, south and west (NESW) inter-tile network links and from the tile's local Compute Processor are captured by the input registers of the network input blocks (NIBs). Each NIB contains a small FIFO for buffering data. On a subsequent cycle, under direction of the router control logic, the crossbar routes the data from the NIBs to the outgoing inter-tile network links or to the tile processor's local NIB. Achieving single-cycle latency on this path from a NIB, through the crossbar, across a 4 *mm* inter-tile network link (the longest wires in the system, excluding the clock), and into the neighbor tile's NIB input register was a central Raw microprocessor design goal. The dynamic router control is an autonomous state machine that controls the passage of multi-word packets through a tile, using the first word of each packet (the packet *header*) to program the settings of the local crossbar to determine the packet's path.

Static Router The static router, shown in Figure A-4, is responsible for managing the transport of operands between separate tiles. The static router is controlled by the router's *switch processor*, a simple processor with a limited 64-bit wide instruction word and a four-element one-read-port one-write-port register file. Each instruction word contains a control-flow or register-file move operation (such as a conditional branch, branch-and-decrement or procedure call) and a number of densely packed route operations. The route operations determine the flow of operands through the crossbars of the two static networks. The static router has its own 64KB software-managed instruction cache. It has no data cache.

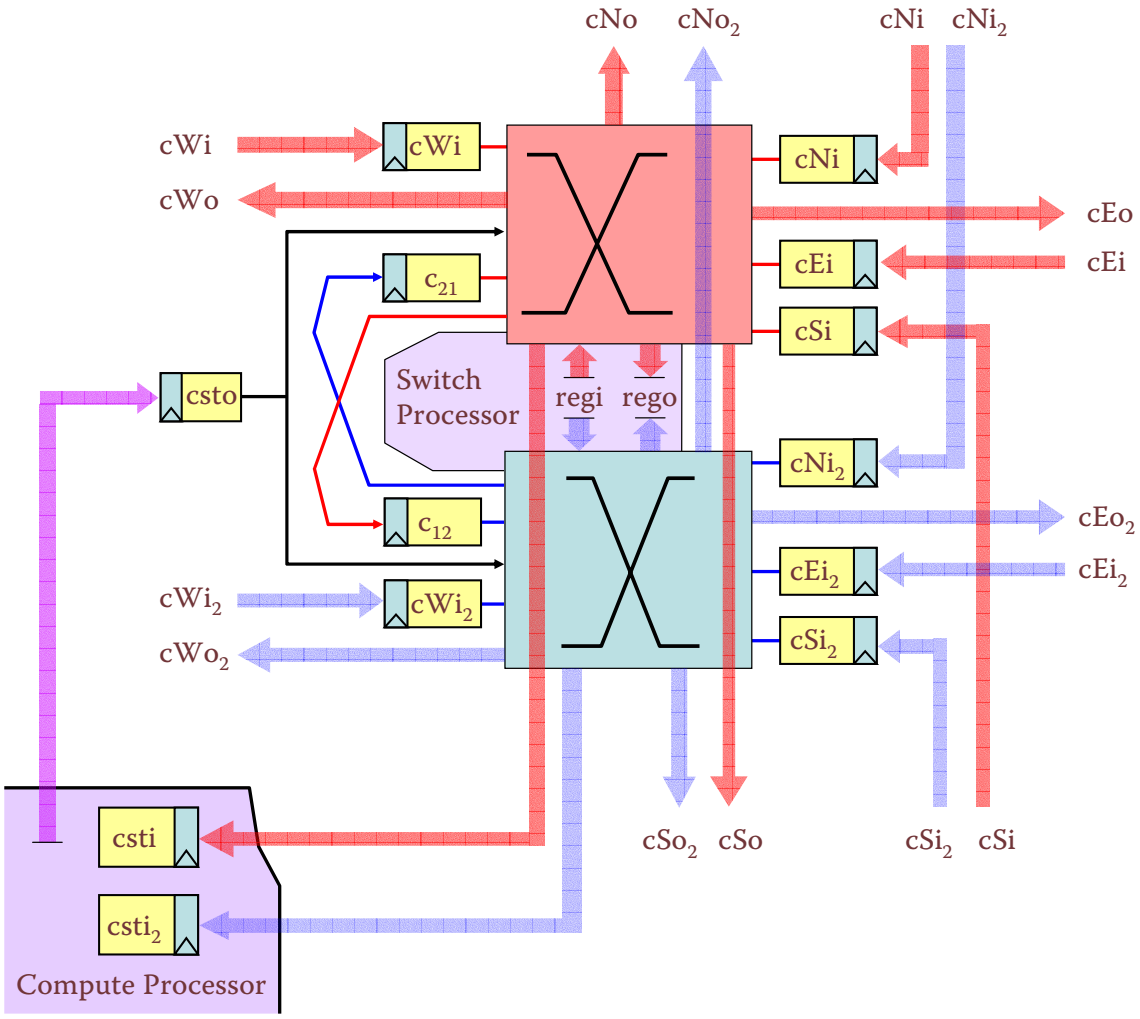


Figure A-4: Basic architecture of a static router. The static router is comprised of the switch processor, two crossbars, and a set of NIBs and inter-tile network links. Most NIBs are connected to exactly one crossbar. However, the csto NIB is accessible by both crossbars, and the c_{12} and c_{21} NIBs are used to route values between crossbars. The rego and regi ports, shared by both crossbars, are not NIBs, but are direct connections into the switch processor.

The static router design instantiates many of the same data routing component types found in the dynamic router: NIBs, crossbars, and network links. The static router design shared the same design goal of single-cycle inter-tile latency.

A.2 Programming the Networks

From an assembly-language perspective, programming Raw requires the specification of three entities for each tile: the compute processor's instruction stream, the compute processor's data segment, and the switch processor's instruction stream. This section develops the reader's understanding of the Raw architecture by focusing on the way in which inter-tile communication through the dynamic

Register Number	NIB dequeued from on read	Network	NIB enqueued onto on write	Network
$24	csti	Static Network 1 (SN1)	csto	SN1 and SN2
$25	cgni	General Dynamic Network (GDN)	cgno	GDN
$26	cmni	Memory Dynamic Network (MDN)	cmno	MDN
$27	$csti_2$	Static Network 2 (SN2)	———	———

Table A.1: Mapping of Compute Processor register space onto network input blocks (NIBs).

and static routers is specified.

A.2.1 Compute Processor Programming

The compute processor communicates with both types of routers through a register-mapped interface. Instruction reads from registers $24, $25, $26, or $27 indicate that the operands are to be taken from the incoming NIBs of the first static network ("SN1"), the GDN, the second static network ("SN2"), and the MDN, respectively. Instructions that write to these register numbers send the results out to the corresponding outgoing NIB[2]. Table A.1 shows the correspondence between register number and NIBs.

Most instructions also have a "bang" variant (i.e., the mnemonic has an exclamation point at the end, e.g., xor!) that specifies that the output should be sent to csto in addition to the register or NIB specified in the destination field of the instruction. Thus, a single instruction can send its output to the static router *and* the dynamic router or register of its choice.

Accesses to register-mapped NIBs (including those indicated via bang instruction variants) have blocking semantics. When the compute processor's program counter (PC) encounters an instruction that is trying to read from an empty NIB, it will stall until the NIB receives a value. When the value becomes available, the instruction is dispatched, and one element is dequeued from the NIB. Similarly, the PC of the compute processor will stall on an instruction that writes to one or more NIBs if one or more of those NIBs is full, or more precisely, if one or more of those NIBs *could be full* after all of the instructions currently in the pipeline have retired.

Figure A-5 shows some example compute processor assembly instructions, demonstrating the use of register-mapped NIBs. Note that the assembly code allows the use of symbolic NIB names, e.g., $csti for the csti NIB, and $csti2 for the $csti_2$ nib.

A.2.2 Switch Processor Programming

Static router instructions consist of a "general-purpose" control-oriented operation and a number of parallel route operations. The general-purpose operations are limited in variety: moves to and

[2]SN1 and SN2 share the same outgoing NIB, labeled csto in the diagram. This is so that the "bang" functionality can send to both static networks.

```
# XOR register 2 with value 15
# place result in register 31

  xor $31, $2, 15

# dequeue 'A' from csti2 NIB
# dequeue 'B' from csti NIB
# place result (A-B) in register 9

  subu $9, $csti2, $csti

# subtract register 2 from register 3
# send result to outgoing csto NIB
# no registers are written --
#  (register 0 is the null register)

  subu! $0, $3, $2

# dequeue 'A' from csti NIB
# load byte from address A+25
# sign extend
# send value to cgno NIB
# send value to csto NIB

  lb! $cgno, 25($csti)
```

```
# dequeue 'A' from csti2 NIB
# jump to that address, A.

  jr $csti2

# dequeue 'A' from cgni NIB
# dequeue 'B' from csti NIB
# store A to address 24+B

  sw $cgni, 24($csti)

# dequeue 'A' from csti NIB
# dequeue 'B' from csti2 NIB
# if (A != B) goto LABEL

  bne $csti, $csti2, LABEL

# dequeue 'A' from cmni NIB
# dequeue 'B' from csti2 NIB
# add A and B
# send value to cmno NIB
# send value to csto NIB

  addu! $cmno, $cmni, $csti2
```

Figure A-5: Selection of compute processor assembly language instructions, showing the use of register-mapped NIBs.

from the local register file, conditional branches (with or without decrement), jumps, and nops ("no operation"). The destinations of all general-purpose operations must be registers, but the sources can be NIBs or registers.

The route operations specify the cycle-by-cycle configuration of the crossbars in Figure A-4. Switch processor instructions contain one route field for every crossbar output, including the rego output, which is used by the switch processor when a general-purpose operation uses a NIB as a source. Each route field can select from any of the crossbar inputs, including regi, which allows the crossbar to route values out of the switch processor's local register file. The rego and regi connections allow data (addresses and loop counts, mostly) to be routed to and from the switch processor, and are the means by which the general-purpose operations communicate with the rest of the system. A *multicast* occurs any time multiple route fields in an instruction make use of the same crossbar input (e.g. the instruction specifies routes from cNi to rego and from cNi to cSo). In a multicast, the same single element is copied to all specified outputs, and if the common input is a NIB, only a single element is dequeued.

Atomicity Semantically, static router instructions occur atomically. To achieve instruction atom-

```
# if register 1 is 0,
#    branch to LABEL

  beqz $1, LABEL

# dequeue 'A' from csto NIB
# (A is routed to switch-
#    processor through rego)
# if A is zero, branch to LABEL

  beqz $csto, LABEL

# dequeue 'A' from cNi2 NIB
# if A is zero, branch to LABEL

  beqz $cNi2, LABEL

# let A = register 2
# store (A-1) into register 2
# if A is zero, branch to LABEL

  beqzd- $2, $2, LABEL

# dequeue 'A', from cWi NIB
# jump to A
# register 2 = return address

  jalr $2, $cWi
```

```
# dequeue 'A' from cSi2 NIB
# (A is routed to switch-processor
#    through rego)
# store into local register 3

  move $3, $cSi2

# same as above, but also:
# route A west on SN2
# route register 2, via regi, to cEo2

  move $3, $cSi2 route  $cSi2 -> $cWo2,
                        $2    -> $cEo2

# performs jalr operation and
# many routes in parallel,
# with multicasts from csto, cWi and $3

  jalr $2, $cNi  route  $3    -> $cNo,
                        $cWi  -> $cEo,
                        $cWi  -> $cSo,
                        $csto -> $cWo,
                        $3    -> $cNo2,
                        $c12  -> $cEo2,
                        $csto -> $cSo2,
                        $cWi2 -> $cWo2,
                        $cEi  -> $csti,
                        $cSi2 -> $csti2,
                        $cNi  -> $c12,
                        $cEi2 -> $c21
```

Figure A-6: Selection of static router assembly language instructions.

icity, the static router must perform a number of checks before executing an instruction. First, it must verify that every source NIB has a data value available. Second, the static router must ensure that neighboring tiles have the corresponding space in their NIBs to receive the data. If these checks pass, the appropriate NIB values are dequeued and routed and the general-purpose operation is executed – all in a single cycle. Otherwise, the static router waits another cycle before trying to execute all components of the instruction again.

Switch Processor Assembly Language Figure A-6 shows some example static router assembly instructions. The assembly language hides the presence of rego and regi; these routes are implicitly generated by the assembler. In addition to checking for simple syntax and semantic errors, the assembler also verifies that the input program does not try to specify an operation that violates the underlying constraints of the architecture (and consequently has no corresponding encoding in the switch processor machine language). Figure A-7 shows two types of violating instructions that are

```
(1) nop         route $3 -> $cWo, $2 -> $cEo
(2) nop         route $cNi -> $cNo2
```

Figure A-7: Switch-specific illegal instruction patterns.

unique to the switch processor. The first instruction calls for two register-file reads, exceeding the capabilities of the one-write-port one-read-port register file. The second instruction specifies a route between entities that are not located on the same crossbar.

Routing between static networks Since the static networks carry the same type of data, it is often useful to move an operand from one network to the other. This can be done in three ways -

1. by routing through the inter-crossbar NIBs, c_{12} and c_{21}, or
2. by writing and then reading the switch processor's local register file (which implicitly uses the rego and regi crossbar ports), or
3. by routing through the compute processor in two steps:
   (a) first, routing the operand to csti, and then
   (b) second, executing a `move $csto, $csti` on the compute processor, so that the operand is available to both crossbars via csto.

A.2.3 Inter-tile Static Network Programming

Figure A-8 shows the code for an array summation operation being performed with two tiles. The north tile's compute processor first transmits a loop count to the north tile's switch processor via csto. This switch processor routes the value southward, storing a local copy in $2. The south tile's switch processor then saves a copy and forwards it to the south tile's compute processor. This communication of the loop count is indicated by the dotted arcs.

As each of these compute processors and switch processor receives its values, it commences a loop[3]. The north tile's compute processor loop repeatedly reads values from an array and sends them to the local static router via csto. The north tile's switch processor loops repeatedly, forwarding values from csto to cSo. The south tile's switch processor forwards values from cNi to csti. The south tile's compute processor then repeatedly adds the values coming in from csti. These communications are indicated by the solid arcs.

These two communication arcs are indicative of two of the typical communication patterns found on the static network. In the first case, a *scalar operand* is transmitted, in this case a loop count; in the second case, a *stream* of values is transmitted. The RawCC parallelizing compiler uses the first class of communication (only, with hundreds or thousands such communications and producing and

[3] The loops are implemented using the compute processor's `bnea` instruction and the switch processor's `bnezd` instruction. These instructions perform a conditional branch and a loop counter update in a single cycle. More details can be found in Chapter B.

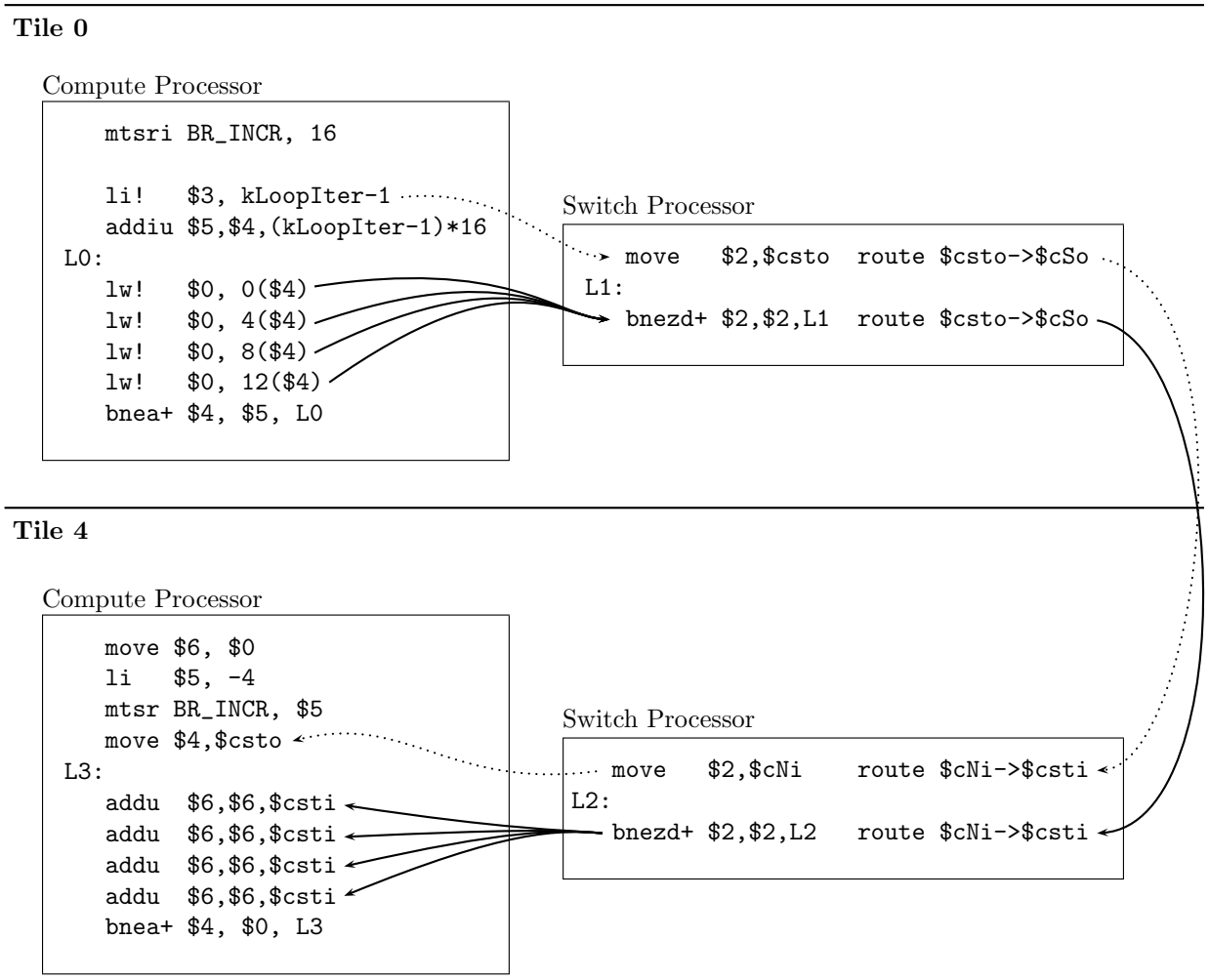


Figure A-8: Inter-tile static network programming. Shown above are two neighboring tiles, one to the north of the other. The north tile is loading elements from an array, and the south tile is summing them up in a reduction operation. The tiles' local static routers are routing the data elements between tiles. The dotted arrows show the transmission of a scalar that specifies the number of elements in the loop. The non-dotted arrows show the transmission of the stream of values originating from the array.

consuming operations) to exploit *instruction-level-parallelism* ("ILP"). It spreads the often loop-unrolled input program's "rat's nest" of scalar data-flow operations across the array of tiles, in order to execute them in parallel. The compute processors execute the operations that they have been assigned, while the switch processors route the operands between the operations, according to the dependence edges of the program graph.

The StreamIt language and compiler, on the other hand, utilizes the second communication pattern, in which kernels repeatedly execute the same computation and relay the results in stream fashion via the static network to a consuming kernel.

As it turns out, many of our hand-optimized program sequences for Raw use a streaming model,

due to the fact that this code structure creates programs that are small enough to merit hand-programming. RawCC-style ILP compilation, which involves many, many operations, is generally too involved to do by hand, unless it is for a small kernel. The output of RawCC is often difficult to decipher due to the complexity and sophistication of the schedules that it can generate.

A.2.4 Dynamic Router Programming

Uses for the dynamic routers The dynamic routers are used to transmit messages that are inappropriate for the statically-ordered (i.e, the ordering can be represented in closed form at compile time) communication model of the static network. Experience using the Raw microprocessor has shown that these messages often fit into one of three classes:

1. messages that are difficult or impossible to predict at compile time (as is often the case for cache miss messages, which may or may not be sent depending on the current state of the cache),

2. messages involving an independent, non-synchronized process (such as asynchronous interrupts and DMA responses coming in off-chip from I/O devices, or messages that must travel across a region of the Raw chip which is running an independent computation), and

3. communications that are statically-ordered but are too costly to represent in closed form (for instance, when loop bodies with different initiation intervals are communicating, the degree of necessary unrolling and/or compiler sophistication required to create the schedule may be unwieldy).

Sending a dynamic message to another tile The process of sending an inter-tile message is relatively simple: the sender constructs a header word using a special instruction, `ihdr`[4], and writes it to the `$cgno` register-mapped NIB. The parameters to `ihdr` include the number of payload data words (between 0 and 31) and the destination tile number[5]. After writing the header, the sender then writes the payload words, in order, to `cgno`.

When the header word arrives at the head of the queue in the `cgno` NIB, the router control examines it and determines which direction it will forward the packet in. The format of the inter-tile header, shown in Figure A-9, contains a destination X and Y coordinate. The control compares these coordinates to the coordinates of the local router and chooses a direction that will bring the packet one hop closer, in Manhattan distance, to the final destination. It then signals to the dynamic router's round-robin arbiter that it wishes to send a packet in that direction. When the

[4]The `ihdr` instruction provides a fast way to create headers that fit the common case of a compile-time known length and a run-time known destination tile. The user can write their own assembly or C routines to create the appropriate headers in other circumstances.

[5]Tiles are numbered in row-major order, with tile 0 at coordinates $(X, Y) = (0, 0)$.

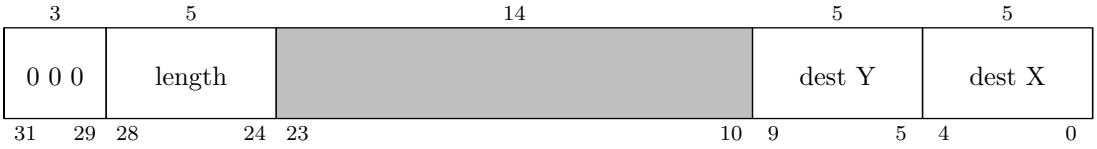


Figure A-9: Inter-tile dynamic router header format. Grayed regions are ignored by routers. The header contains a length (the number of non-header words in the packet), and a destination, specified in terms of X and Y coordinates.

arbiter indicates that the packet's turn has come (which may be on the next cycle, if no other local NIBs are currently sending in that direction), it will forward the entire packet, one word per cycle, through the crossbar, to the neighbor dynamic router, stalling to prevent overflow of the neighbor's NIB. This process is repeated until the message arrives at its destination tile, where the message header is discarded and the payload words are routed, one by one, into the receiving `cgni` NIB. The receiving compute processor can consume the words as they arrive via `$cgni`. Figure A-10 displays a pair of assembly sequences (one for each tile) in which two tiles communicate over the general dynamic network.

Dynamic network routing The dynamic networks use *wormhole routing* [22], meaning that routers forward the individual words of the packet on to the next tile as space in the neighbor becomes available, rather than waiting for the entire packet to accumulate in the local NIB. This allows the NIBs to be quite small in size (4 elements each for the inter-tile NIBs), smaller than the size of many of the packets that travel over the routers. Furthermore, to avoid in-network deadlock, the routers perform deterministic *dimensioned-ordered* routing, performing all necessary X-dimension routes before performing any Y-dimension routes.

Dynamic network message arrival order Unlike for static network messages, the arrival ordering of messages at a destination tile or I/O port is generally guaranteed only in one limited circumstance: if the messages were sent from the same tile. This is because two such messages will always take the same path and will always maintain the same ordering across the NIBs they encounter. The lack of knowledge about arriving messages is one reason that dynamic network messages are multi-word – the message almost always needs to communicate additional words describing the data and the action to be taken on it. One efficient way of doing this is through the use of *active messages* [120], where the first payload word of the message is a code pointer that the receiving tile jumps to via a `jr` instruction. The code pointer points to a handler routine customized for the action that needs to be taken[6].

This limited arrival ordering capability has proven useful in the Raw microprocessor, for instance, for maintaining the ordering of memory and I/O transactions. Without it, messages would require

[6]A retrospective definition of an active message is that it involves the transmission of an instance of an object-oriented class with a single virtual "run" method stored at the beginning of the instance.

Tile 7 Compute Processor

```
##  Tile 7 wants tile 13 to do a memory access on its behalf.
##  It sends a message to tile 13 with a return tile number and the address
##  in question. It then waits on the network port for the response.

  # load destination tile number
  li $5, 13

  # create and send outgoing header (recipient in $5, payload: 2 words)
  ihdr $cgno, CREATE_IHDR_IMM(2)($5)

  # send my tile number
  li $cgno, 7

  # send address, this completes the message
  move $cgno, $4

  # < time passes ... >

  # receive response message from tile 13
  move $2, $cgni
```

Tile 13 Compute Processor

```
##  Tile 13 is processing memory requests on behalf of other tiles.
##  It does not know which tiles will request accesses, but it does
##  know that only that type of access will be requested.

  # wait for incoming message, then create and send outgoing header
  #    (recipient tile is value dequeued from $cgni, payload: 1 word)
  ihdr $cgno, CREATE_IHDR_IMM(1)($cgni)

  # load from address specified by second word of incoming message,
  # result goes out to cgno and completes the outgoing message.

  lw $cgno, 0($cgni)
```

Figure A-10: Inter-tile dynamic router communication using the `ihdr` instruction to create headers. Tile 7 sends a message to Tile 13, which consists of a header (which is discarded automatically by Tile 13's compute processor), a return tile number, and a memory address. Tile 13 receives the message and then sends another message back to Tile 7 with the contents of that memory address. `ihdr` takes one cycle to execute. `CREATE_IHDR_IMM` is a helper macro that helps the user fill in the length field of the `ihdr` instruction.

a sequence number, and the message recipients (hardware or software) would need to reorder and buffer the messages upon arrival. In fact, a common programming pattern has been to restructure the computation so that ordered requests originate from the same tile.

Sending a dynamic message to an external port Dynamic messages can also be sent to I/O ports on the edge of the array. Internal to the Raw microprocessor, each I/O port is identified by a 3-tuple, <X, Y, fbits>. X and Y are the coordinates of the tile that the I/O port is connected to, and *fbits* (the "final route bits") is the position of the I/O port relative to that tile. Thus, when the routers route a packet, they route until the X and Y coordinates of the packet match that of the local tile, and then they perform a final route in the direction specified by the *fbits*. The encoding of the fbits directions is shown in Table A.2. The inter-tile header is in fact just an instance of the generalized dynamic header, shown in Figure A-11, with the fbits value set to the direction of the local compute processor.

Direction	fbits value
North	101
East	100
South	011
West	010
Compute Processor	000

Table A.2: Final Route (fbits) values for each of the output ports in a dynamic router. 001, 110, and 111 are reserved.

Unlike in the case of the compute processor, clients of I/O ports see the headers of message packets. As a result, it makes sense to use some of the free space to encode useful information. To that end, the generalized header contains three additional fields: *src X* and *src Y*, indicating the coordinates of the tile that originated the message, and a *user* field, which describes the nature of the request. Of course, four bits is not enough to describe all possible requests, so subsequent words of the message payload are often used for this purpose as well. The meaning of the user bits is assigned by the hardware and software systems built on top of the Raw microprocessor, however a few of the values have been pre-allocated in the Raw architecture to indicate messages for cache miss requests, interrupts, and buffer management.

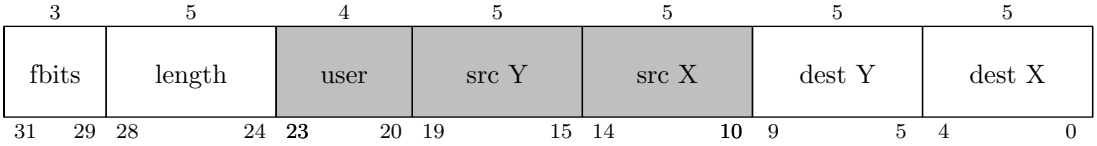


Figure A-11: Generalized dynamic router header format. The grayed regions (user, src Y, and src X) are ignored by routers, but can be used by devices connected to the I/O ports. The *fbits* field specifies a final route to be performed after the packet has reached the tile specified by dest Y and dest X.

Figure A-12 shows an example of dynamic network communication between a tile's compute processor and a device attached to an I/O port. The compute processor sends a request message to the I/O port, which travels to the I/O port and is read by the device. The device uses the field of the received header to create a return header and relays all words of the original message's payload (using the length field of the original message) back to the sender. The sender then receives the message, minus the header, which has been dequeued automatically as the message enters the cgni.

Compute Processor

```
##  Code assumes compile-time-known payload size (kMessageLength = 1),
##  user field value (kUserField), and port location (kFbits, kDestY, kDestX).

  # $3 = tuple <kDestY:5, kDestX:5, kFbitsSouth:3>
  ori  $3, $0, (((kDestY << 5) + kDestX) << 3) + kFbits

  # $4 = header word <XXX:3, length:5, user:4, srcY:5, srcX:5, XXXXX:5, XXXXX:5>
  ohdr $4, CREATE_OHDR_IMM(kMessageLength, kUserField) ($0)

  rrmi $4, $3, 3, 0xE00003FF   # replaces X's in header ($4) with
                               # corresponding bits of ($3 rotated right by 3).

  move $cgno, $4               # send the header
  li   $cgno, 42               # send the data, completing message
                               # < time passes ... >
  move $4, $cgni               # receive response, header is dropped by hardware
```

Code that injects a dynamic message destined for an I/O port, and receives a reply. No particular instruction exists for creating headers that specify an I/O port as destination. This example uses the `ohdr` and `rrmi` instructions because they are efficient, but a standard C routine suffices.

I/O port client

```
while (1) {
   /* receive header from I/O port, blocks until value arrives */
   int inHeader = gdn_io_receive(machine, ioPort);
   yield;

   /* decode header */
   int fbits, payloadSize, userField, senderY, senderX, myY, myX;
   DecodeDynHdr(inHeader, &fbits, &payloadSize, &userField,
                &senderY, &senderX, &myY, &myY);

   /* send reply header and echo payload */
   gdn_io_send(machine, ioPort, ConstructDynHdr(kFbitsProc, payloadSize,
                                                userField, myY, myX,
                                                senderY, senderX));
   yield;

   while (payloadSize-- != 0) {
        gdn_io_send(machine, ioPort, gdn_io_receive(machine, ioPort));
        yield;
   }
}
```

An example I/O port device that sends packets back to their senders. The device is expressed in the *bC* modeling language, a superset of C that includes the keyword `yield`, indicating the end of a clock cycle. The `gdn_io_receive` and `gdn_io_send` are used to send and receive data on a Raw I/O port. Our infrastructure allows bC devices to be used across the spectrum of Raw realizations, from software simulation to rtl simulation to the actual hardware prototype. Devices that require high performance (such as the DRAM controllers) were later re-implemented in hardware.

Figure A-12: Communication between a compute processor and an example device attached to an I/O port. A message is sent from a compute processor to the I/O device via the I/O port. The device sends a copy of the message back to the compute processor, via the same I/O port.

General versus Memory Dynamic Network Although the implementations of the underlying networks are identical, the purpose of the two dynamic networks in the Raw system differ significantly. The MDN is accessed directly only by operating-systems or device-drivers, while the GDN is available for general user use. The restrictions on MDN use arise from the need to make sure that the user does not interfere 1) with the compute processor hardware as it performs data and instruction cache misses using the MDN, 2) with the operating system hardware as it performs system-critical operations, and 3) with the communication of I/O devices with tiles and other I/O devices through the MDN. The GDN examples shown in Figures A-10 and Figures A-12 will work on the MDN if cmno and cmni are substituted for cgno and cgni, and a few additional locking and prefetching instructions are inserted.

Appendix B

The Raw Architecture Instruction Set

B.1 Compute Processor Instruction Set

Refer to Section B.2 for concrete definitions of semantic helper functions such as *sign-extend-16-to-32* and *rotate-left*.

B.1.1 Register Conventions

The compute processor uses register conventions similar to those used in MIPS microprocessors. Procedures cannot rely on *caller-saved* registers retaining their values upon a procedure call. Procedures must restore the initial values of *callee-saved* registers before returning to their caller. The conventions are shown below.

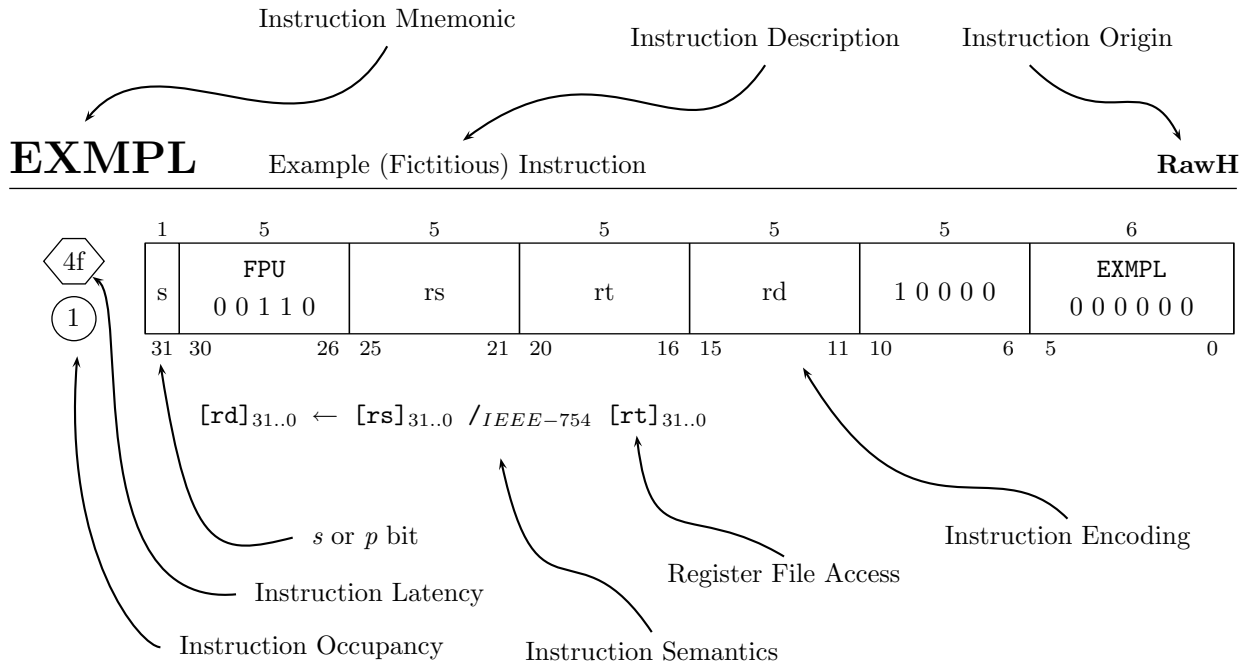
Register Number	Assembly Alias	Saved by	Description
$0		n/a	Always has value zero.
$1	$at	caller	Assembler temporary clobbered by some assembler operations.
$2..$3		caller	First and second words of return value, respectively.
$4..$7		caller	First 4 arguments of function.
$8..$15		caller	General registers.
$16..$23		callee	General registers.
$24	$cst[i/o]	n/a	Static Network input/output port.
$25	$cgn[i/o]	n/a	General Dynamic Network input/output port.
$26	$csti2	n/a	Static Network input port #2.
$27	$cmn[i/o]	n/a	Memory Dynamic Network input/output port.
$28	$gp	callee	Global pointer. Points to start of tile's code and static data.
$29	$sp	callee	Stack pointer. Stack grows towards lower addresses.
$30		callee	General register.
$31		caller	Link register. Saves return address for function call.

B.1.2 Compute Processor Instruction Template

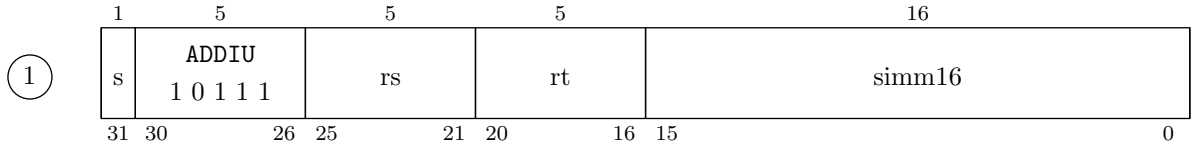
Shown below is an example instruction listing. The *instruction occupancy* is the number of compute processor issue cycles that are occupied by the instruction. Subsequent instructions must wait for this number of cycles before issuing. The *instruction latency* is the total number of cycles that must pass before a subsequent dependent instruction can issue. Suffixes of d and f indicate that an instruction uses the integer and floating-point divide units, respectively, for that number of cycles. Subsequent instructions that require a particular unit will stall until that unit is free. A suffix of b means that the instruction has an additional 3 cycles of occupancy on a branch misprediction. An occupancy of c means that the instruction takes at least 13 cycles if a cache line is evicted, 5 cycles if only an invalidation occurs, otherwise 1 cycle.

The s or p bits in the instruction encoding specify respectively whether 1) an instruction's output will be copied to `csto` in addition to the destination register, or 2) whether a branch is predicted taken or not.

Generally, the Raw compute processor attempts to inherit the MIPS instruction set mnemonics to the extent that it reduces the learning curve for new users of the system. However, the underlying instruction semantics have been "cleaned up"; for instance, interlocks have been added for load, branch, multiply and divide instructions (reducing the need to insert `nop`s), and the FPU uses the same register set as the ALU. To this end, the *instruction origin* specifies whether the instruction semantics are very similar the MIPS instruction of the same name ("MIPS"), whether they are specific to the Raw architecture ("" or Raw), or to the Raw architecture extended with hardware instruction caching ("RawH"). Of course, the instruction encodings (including the presence of s and p bits) are completely different from MIPS.

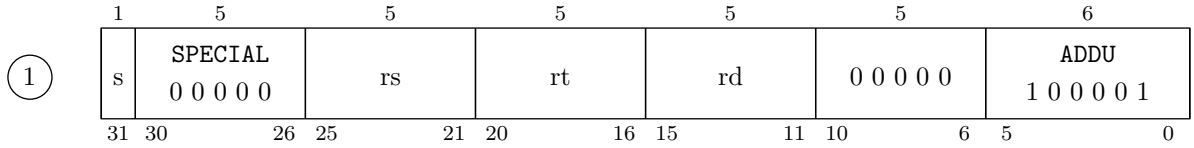


ADDIU Add Immediate **MIPS**



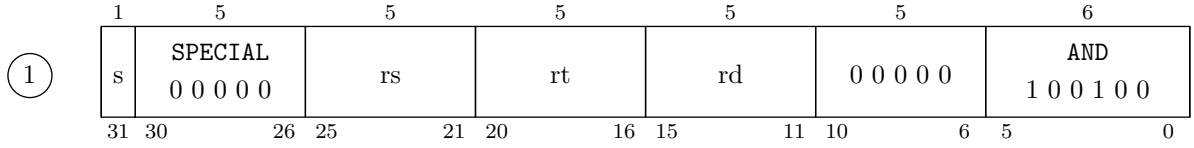

```
simm32[31..0] ← (sign-extend-16-to-32 simm16)
[rt]31..0     ← { [rs]31..0 + simm32 }31..0
```

ADDU Add **MIPS**



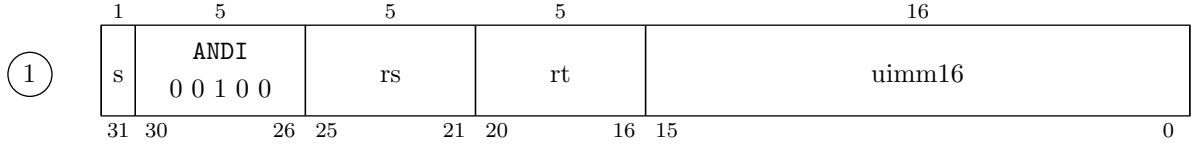

```
[rd]31..00 ← { [rs]31..00 + [rt]31..00 }31..0
```

AND And Bitwise **MIPS**




```
[rd]31..00 ← [rs]31..00 & [rt]31..00
```

ANDI And Bitwise Immediate **MIPS**

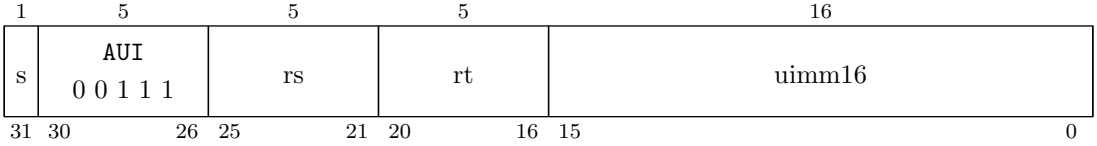



```
[rt]31..16 ← [rs]31..16
[rt]15..00 ← [rs]15..00 & uimm16
```

AUI

Add Upper Immediate

(1)




```
[rt]31..16 ← { [rs]31..16 + uimm16 }15..0
[rt]15..00 ← [rs]15..00
```

B

Branch Unconditional (Assembly Macro)

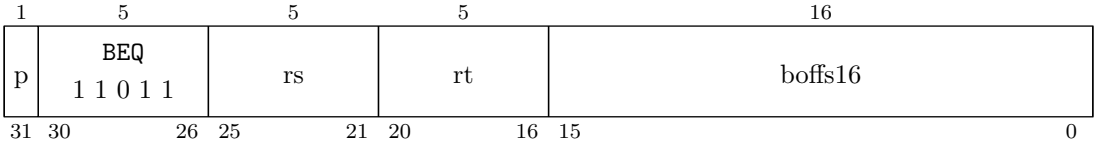
(1) *b <label>*

```
PC31..00 ← <label>
```

BEQ

Branch if equal

(1b)

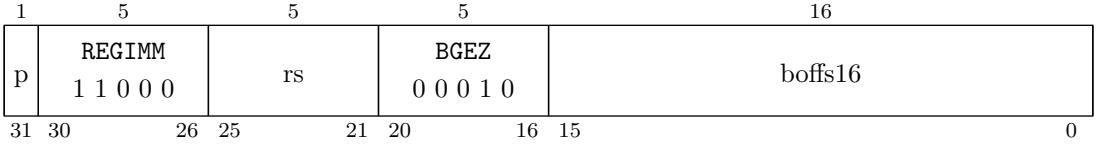



```
if ([rs] == [rt])
   PC31..02 ← { PC31..02 + (sign-extend-16-to-30 boffs16) }29..00
   PC01..00 ← 0
```

BGEZ

Branch if greater than or equal to zero (signed)

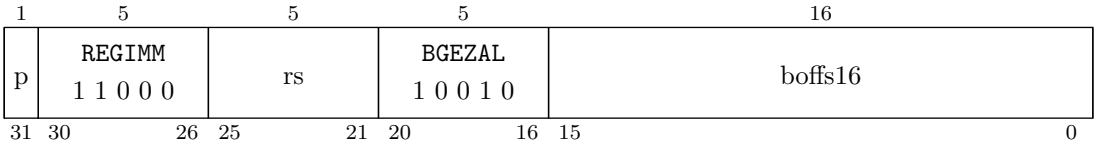
(1b)




```
if (![rs]31)
   PC31..02 ← { PC31..02 + (sign-extend-16-to-30 boffs16) }29..00
   PC01..00 ← 0
```

BGEZAL Branch if greater than or equal to zero and link (signed)

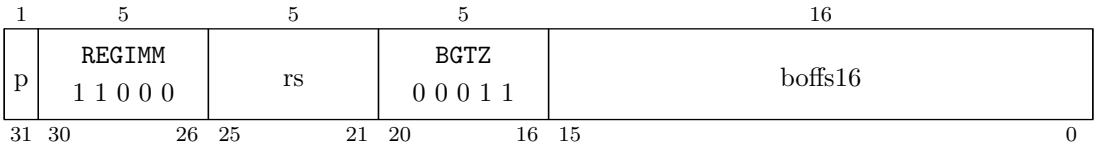
(1b)




```
if (![rs]31)
   [31]    ← { PC + 4 }31..00
   PC31..02 ← { PC31..02 + (sign-extend-16-to-30 boffs16) }29..00
   PC01..00 ← 0
else
   [31]    ← { PC + 4 }31..00
```

BGTZ Branch if greater than zero (signed)

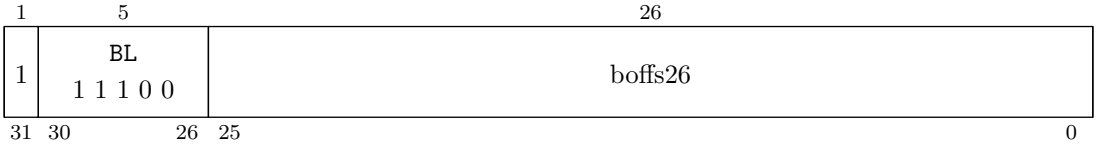
(1b)




```
if (![rs]31 && ([rs] != 0))
   PC31..02 ← { PC31..02 + (sign-extend-16-to-30 boffs16) }29..00
   PC01..00 ← 0
```

BL Branch Long **RawH**

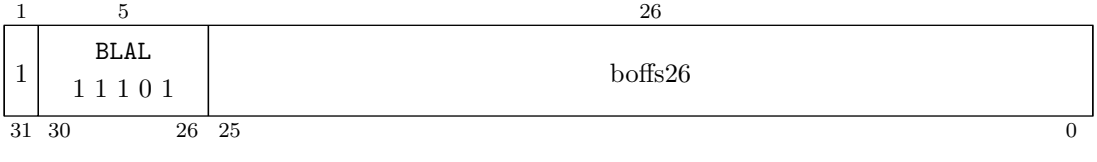
(1)




```
PC31..02 ← { PC31..02 + (sign-extend-26-to-30 boffs26) }29..00
PC01..00 ← 0
```

BLAL Branch Long and Link **RawH**

(1)

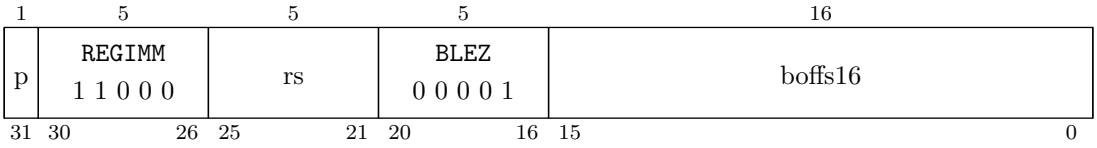



```
[31]    ← { PC + 4 }31..00
PC31..02 ← { PC31..02 + (sign-extend-26-to-30 boffs26) }29..00
PC01..00 ← 0
```

BLEZ

Branch if less than or equal to zero (signed)

(1b)

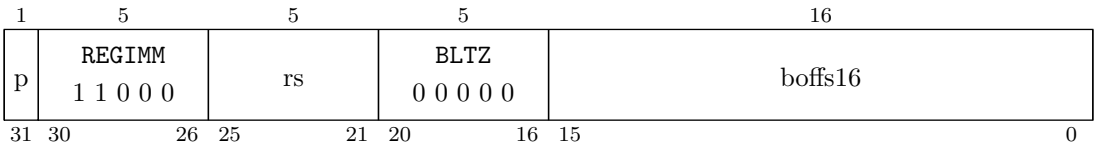



```
if ([rs]31 || ([rs] == 0))
   PC31..02 ← { PC31..02 + (sign-extend-16-to-30 boffs16) }29..00
   PC01..00 ← 0
```

BLTZ

Branch if less than zero (signed)

(1b)

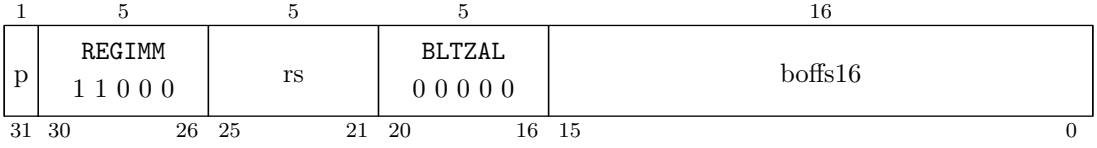



```
if ([rs]31)
   PC31..02 ← { PC31..02 + (sign-extend-16-to-30 boffs16) }29..00
   PC01..00 ← 0
```

BLTZAL

Branch if less than zero and link (signed)

(1b)

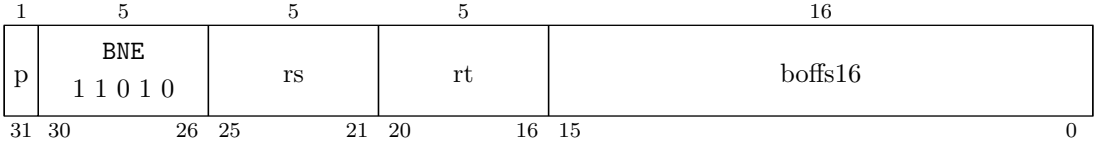



```
if ([rs]31)
   [31]    ← { PC + 4 }31..00
   PC31..02 ← { PC31..02 + (sign-extend-16-to-30 boffs16) }29..00
   PC01..00 ← 0
else
   [31]    ← { PC + 4 }31..00
```

BNE

Branch if not equal

(1b)

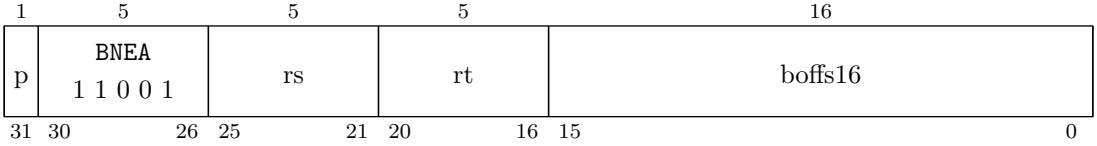



```
if ([rs] != [rt])
   PC31..02 ← { PC31..02 + (sign-extend-16-to-30 boffs16) }29..00
   PC01..00 ← 0
```

BNEA

Branch if not equal and add

(1b)

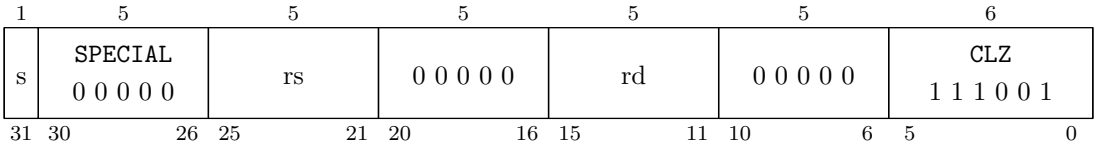



```
if ([rs] != [rt])
   PC31..02 ← { PC31..02 + (sign-extend-16-to-30 boffs16) }29..00
   PC01..00 ← 0
[rs] = [rs] + SR[BR_INC]
```

CLZ

Count Leading Zero

(1)



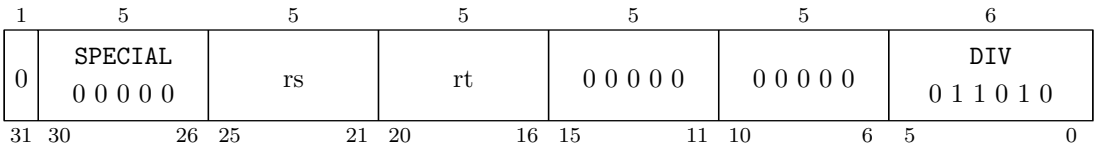
$$[\texttt{rd}]_{05..00} \leftarrow \sum_{i=0}^{31} ([\texttt{rs}]_{31..i} \texttt{ ? 0 : 1})$$

$$[\texttt{rd}]_{31..06} \leftarrow 0$$

DIV

Divide Signed

(42d)
(1)



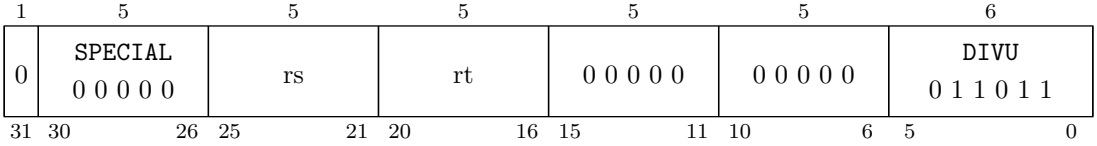
$$\texttt{LO} \leftarrow \{ \texttt{ [rs] } /_{signed} \texttt{ [rt] } \}_{31..0} \}_{31..0}$$
$$\texttt{HI} \leftarrow \{ \texttt{ [rs] } \%_{signed} \texttt{ [rt] } \}_{31..0} \}_{63..32}$$

```
if ([rt] == 0)
   HI ← [rs]
   if ([rs]31)
      LO ← 1
   else
      LO ← -1
```

DIVU Divide Unsigned

42d

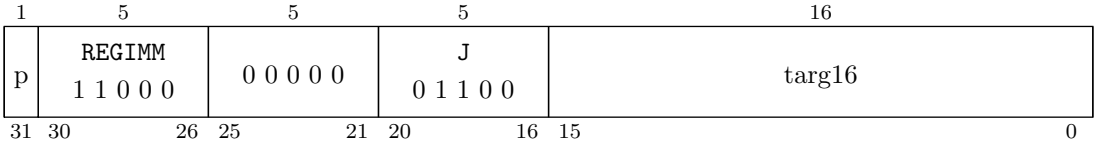
1




```
LO ← { [rs] /unsigned [rt] }31..0 }31..0
HI ← { [rs] %unsigned [rt] }31..0 }63..32

if ([rt] == 0)
   HI ← [rs]
   LO ← -1
```

J Jump

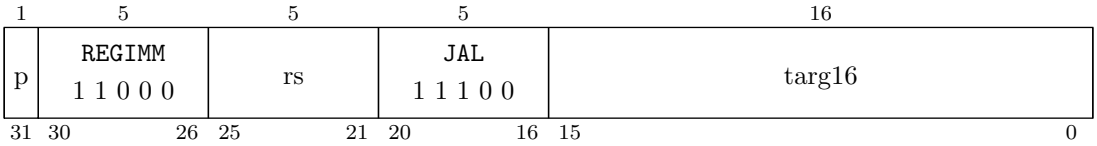
1




```
PC31..02 ← (zero-extend-16-to-30 targ16)
PC01..00 ← 0
```

JAL Jump and link

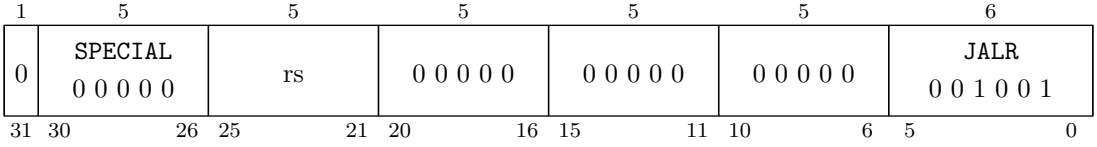
1




```
[31]    ← { PC + 4 }31..00
PC31..02 ← (zero-extend-16-to-30 targ16)
PC01..00 ← 0
```

JALR Jump and link through Register

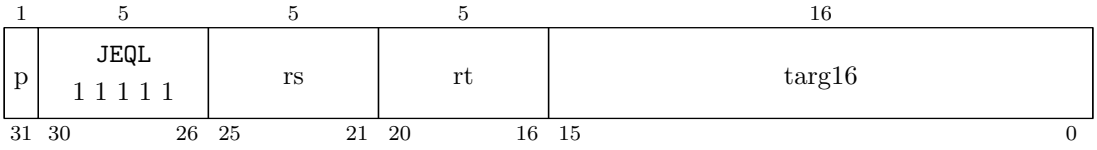
4




```
[31]    ← { PC + 4 }31..00
PC31..02 ← [rs]31..02
PC01..00 ← 0
```

JEQL — Jump if not equal and link

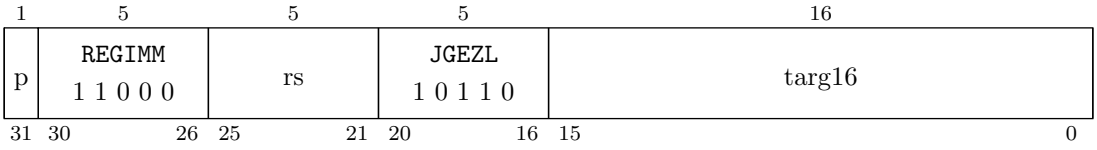
(1b)




```
if ([rs] == [rt])
   [31]    ← { PC + 4 }31..00
   PC31..02 ← (zero-extend-16-to-30 targ16)
   PC01..00 ← 0
else
   [31]    ← { PC + 4 }31..00
```

JGEZL — Jump if greater than or equal to zero and link (signed)

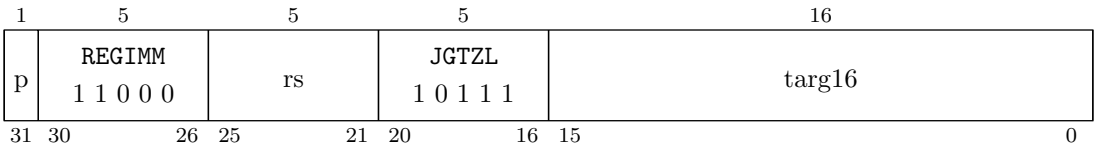
(1b)




```
if (![rs]31)
   [31]    ← { PC + 4 }31..00
   PC31..02 ← (zero-extend-16-to-30 targ16)
   PC01..00 ← 0
else
   [31]    ← { PC + 4 }31..00
```

JGTZL — Jump if greater than zero and link (signed)

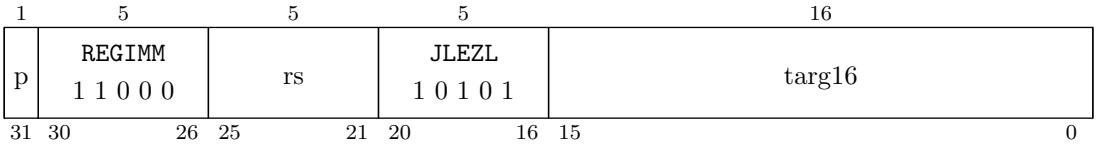
(1b)




```
if (![rs]31 && ([rs] != 0))
   [31]    ← { PC + 4 }31..00
   PC31..02 ← (zero-extend-16-to-30 targ16)
   PC01..00 ← 0
else
   [31]    ← { PC + 4 }31..00
```

JLEZL — Jump if less than or equal to zero and link (signed)

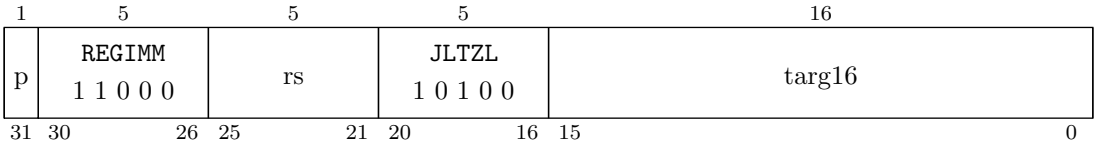
(1b)




```
if ([rs]31 || ([rs] == 0))
   [31]    ← { PC + 4 }31..00
   PC31..02 ← (zero-extend-16-to-30 targ16)
   PC01..00 ← 0
else
   [31]    ← { PC + 4 }31..00
```

JLTZL — Jump if less than zero and link (signed)

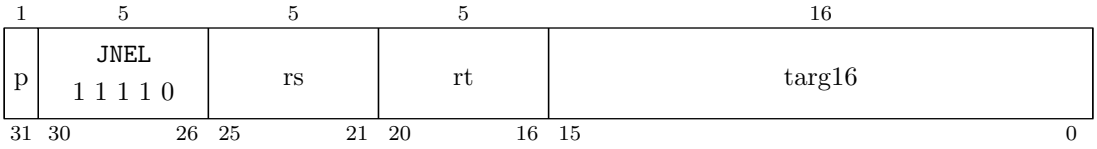
(1b)




```
if ([rs]31)
   [31]    ← { PC + 4 }31..00
   PC31..02 ← (zero-extend-16-to-30 targ16)
   PC01..00 ← 0
else
   [31]    ← { PC + 4 }31..00
```

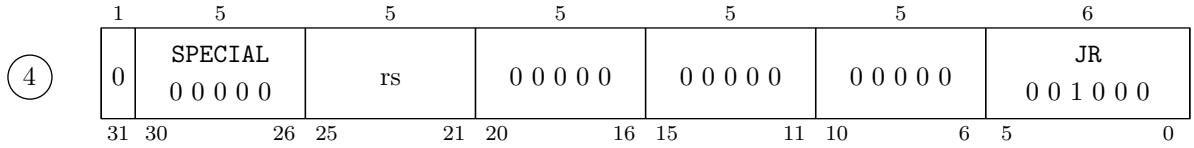
JNEL — Jump if not equal and link

(1b)



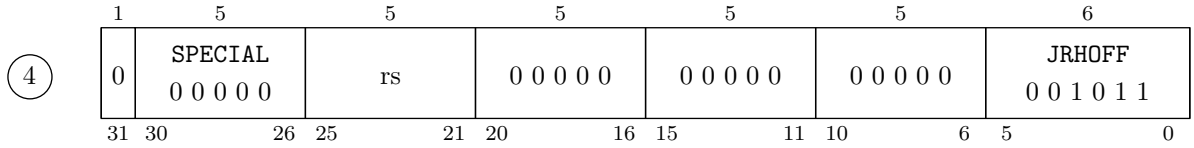

```
if ([rs] != [rt])
   [31]    ← { PC + 4 }31..00
   PC31..02 ← (zero-extend-16-to-30 targ16)
   PC01..00 ← 0
else
   [31]    ← { PC + 4 }31..00
```

JR — Jump through Register



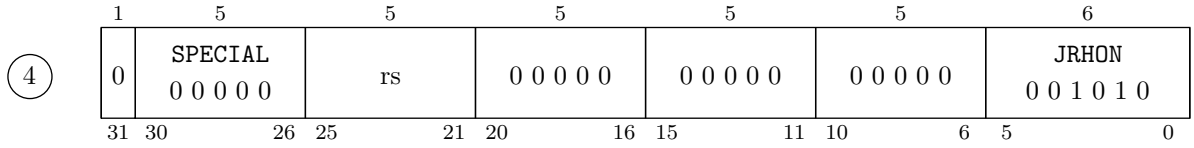
$PC_{31..02} \leftarrow [rs]_{31..02}$
$PC_{01..00} \leftarrow 0$

JRHOFF — Jump through Register and Disable Hardware ICaching **RawH**



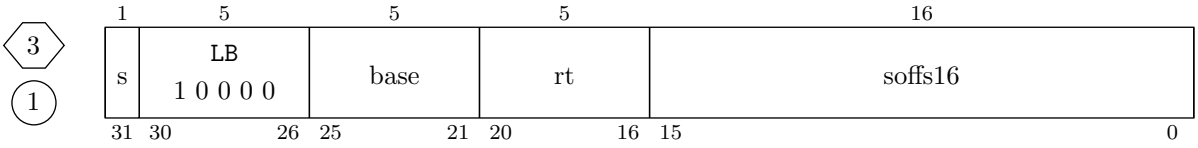
$PC_{31..02} \leftarrow [rs]_{31..02}$
$PC_{01..00} \leftarrow 0$

JRHON — Jump through Register and Enable Hardware ICaching **RawH**



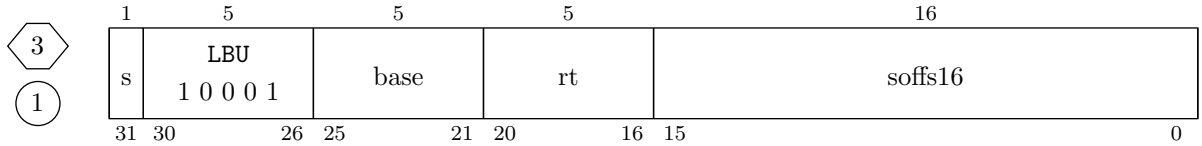
$PC_{31..02} \leftarrow [rs]_{31..02}$
$PC_{01..00} \leftarrow 0$

LB — Load Byte



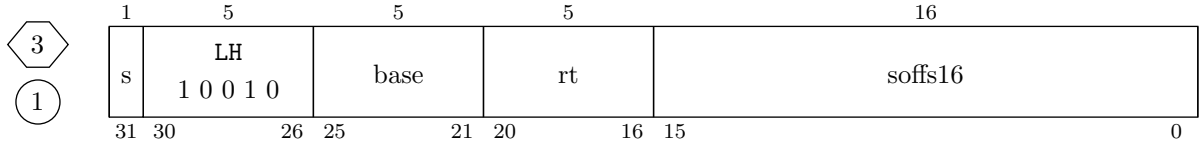
ea $\leftarrow$ { [base] + (*sign-extend-16-to-32* soffs16) }$_{31..0}$
[rt] $\leftarrow$ (*sign-extend-8-to-32* (*cache-read-byte* ea))

LBU — Load Byte Unsigned



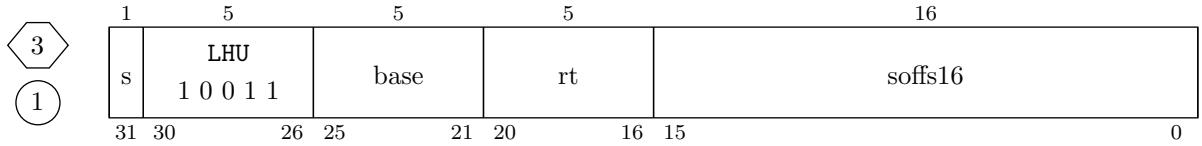

```
ea        ← { [base] + (sign-extend-16-to-32 soffs16) }31..0
[rt]31..8 ← 0
[rt]7..0  ← (cache-read-byte ea)
```

LH — Load Halfword




```
ea    ← { [base] + (sign-extend-16-to-32 soffs16) }31..0
[rt]  ← (sign-extend-16-to-32 (cache-read-half-word ea))
```

LHU — Load Halfword Unsigned




```
ea         ← { [base] + (sign-extend-16-to-32 soffs16) }31..0
[rt]31..16 ← 0
[rt]15..0  ← (cache-read-half-word ea)
```

LI — Load Immediate (Assembly Macro) **MIPS**

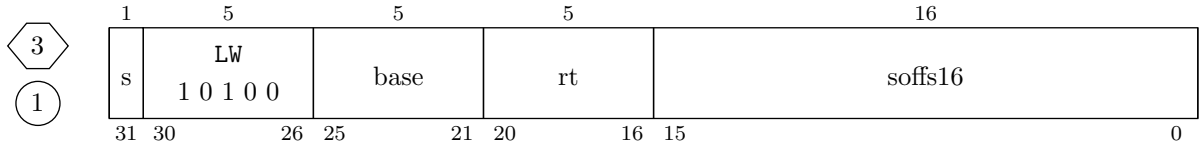
1-2

li rd, uimm32
li! rd, uimm32

```
[rd]31..0 ← uimm3231..0
```

LW

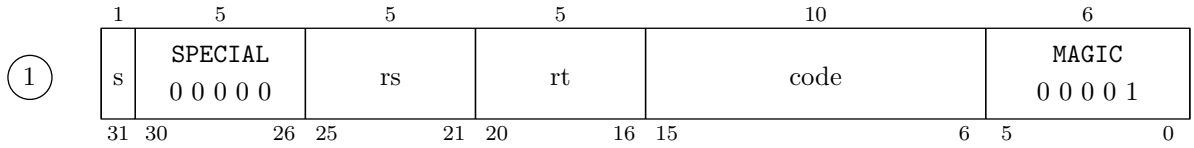
Load Word




```
ea     ← { [base] + (sign-extend-16-to-32 soffs16) }31..0
[rt]   ← (cache-read-word ea)
```

MAGIC

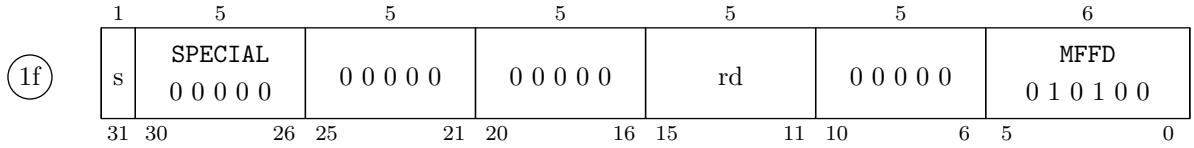
User-specified simulator function




```
[rt]31..00 ←  (user_function code [rs]) - On BTL simulator

[rt]31..00 ←  unspecified value         - On RTL and hardware
```

MFFD

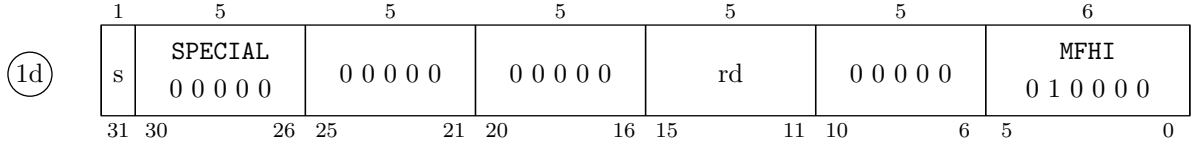
Move from FD




```
[rd]31..00 ←  FD31..00
```

MFHI

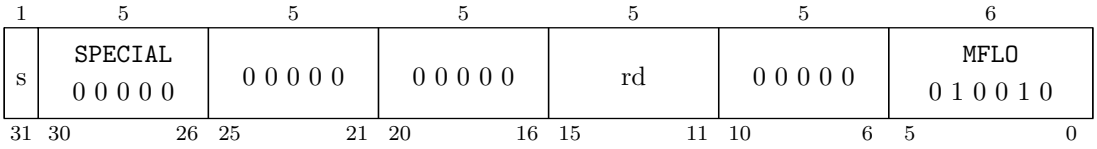
Move from HI




```
[rd]31..00 ←  HI31..00
```

MFLO Move from LO

(1d)



$[\texttt{rd}]_{31..00} \leftarrow \texttt{LO}_{31..00}$

MOVE MOVE (Assembly Macro) **MIPS**

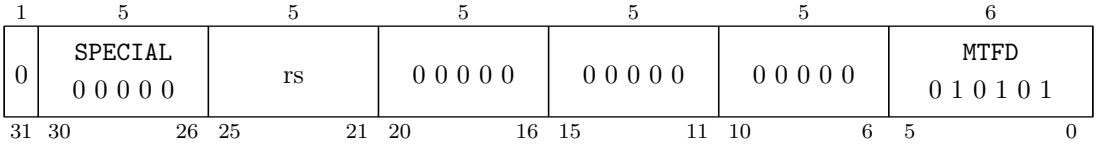
(1)

move rd, rt
move! rd, rt

$[\texttt{rd}]_{31..0} \leftarrow [\texttt{rt}]_{31..0}$

MTFD Move to FD

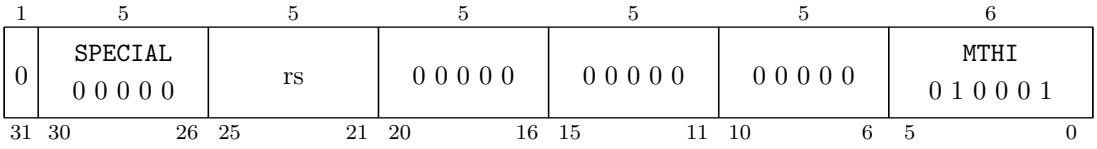
(1f)



$\texttt{FD}_{31..00} \leftarrow [\texttt{rs}]_{31..00}$

MTHI Move to HI

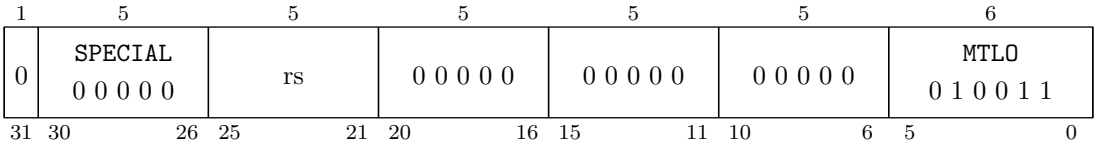
(1d)



$\texttt{HI}_{31..00} \leftarrow [\texttt{rs}]_{31..00}$

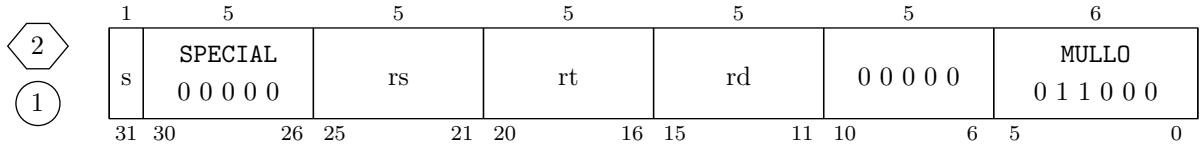
MTLO Move to LO

(1d)



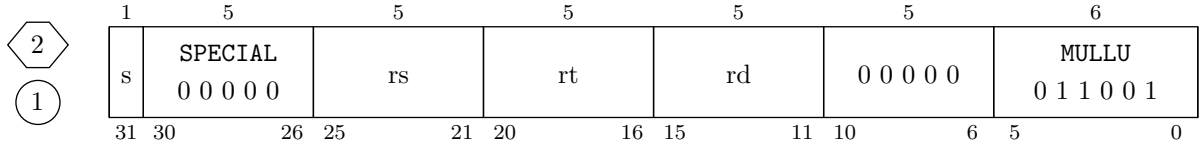
$\texttt{LO}_{31..00} \leftarrow [\texttt{rs}]_{31..00}$

MULLO Multiply Low Signed



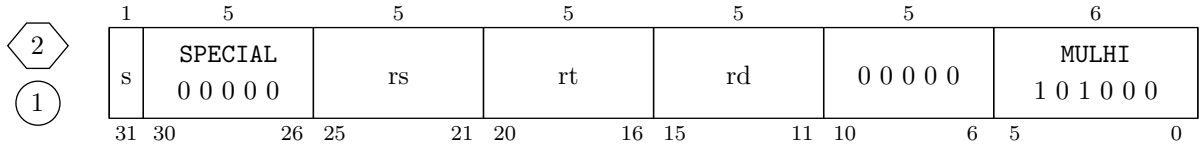
$[\texttt{rd}]_{31..00} \leftarrow \{\ [\texttt{rs}]_{31..00}\ *_{signed}\ [\texttt{rt}]_{31..00})\ \}_{31..0}$

MULLU Multiply Low Unsigned



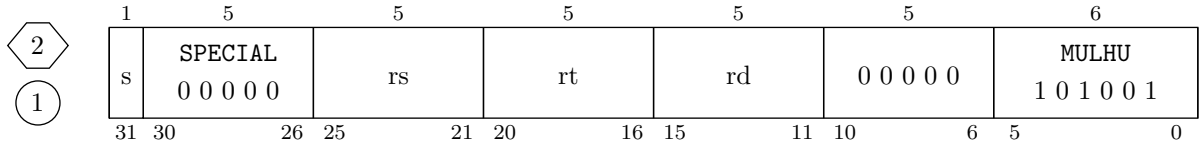
$[\texttt{rd}]_{31..00} \leftarrow \{\ [\texttt{rs}]_{31..00}\ *_{unsigned}\ [\texttt{rt}]_{31..00})\ \}_{31..0}$

MULHI Multiply High Signed



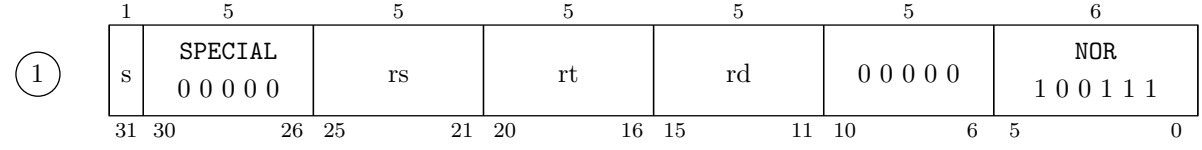
$[\texttt{rd}]_{31..00} \leftarrow \{\ [\texttt{rs}]_{31..00}\ *_{signed}\ [\texttt{rt}]_{31..00})\ \}_{63..32}$

MULHU Multiply High Unsigned



$[\texttt{rd}]_{31..00} \leftarrow \{\ [\texttt{rs}]_{31..00}\ *_{unsigned}\ [\texttt{rt}]_{31..00})\ \}_{63..32}$

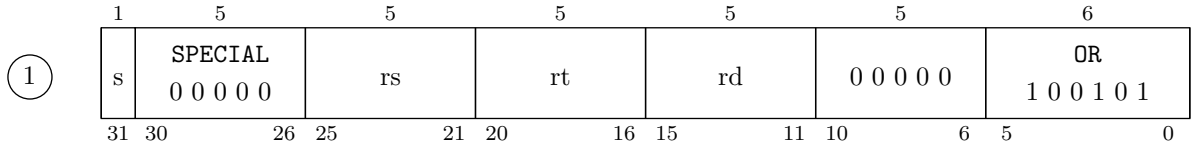
NOR Nor Bitwise **MIPS**



$[\texttt{rd}]_{31..00} \leftarrow \texttt{~}([\texttt{rs}]_{31..00}\ \texttt{|}\ [\texttt{rt}]_{31..00})$

OR

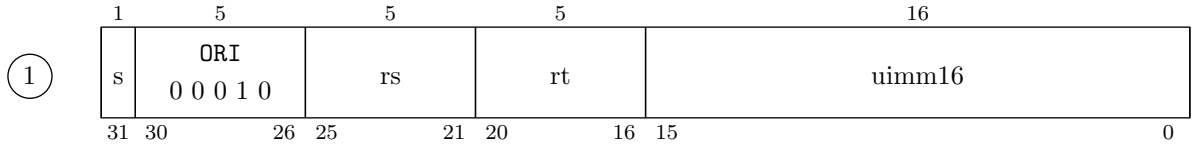
Or Bitwise **MIPS**



$[\texttt{rd}]_{31..00} \leftarrow [\texttt{rs}]_{31..00}$ | $[\texttt{rt}]_{31..00}$

ORI

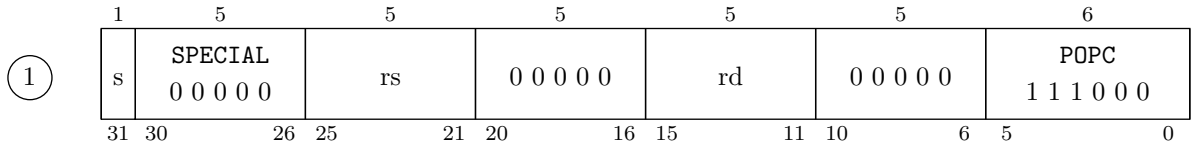
Or Bitwise Immediate **MIPS**



$[\texttt{rt}]_{31..16} \leftarrow [\texttt{rs}]_{31..16}$
$[\texttt{rt}]_{15..00} \leftarrow [\texttt{rs}]_{15..00}$ | uimm16

POPC

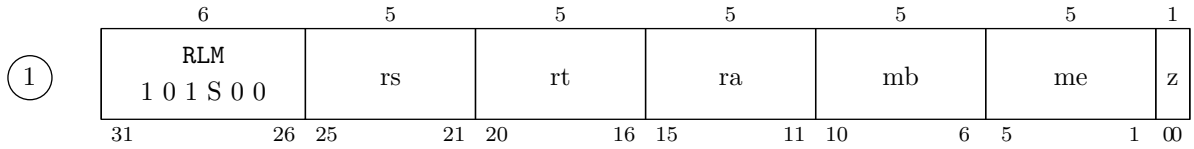
Population Count



$$[\texttt{rd}]_{04..00} \leftarrow \sum_{i=0}^{31} [\texttt{rs}]_i$$
$[\texttt{rd}]_{31..05} \leftarrow 0$

RLM

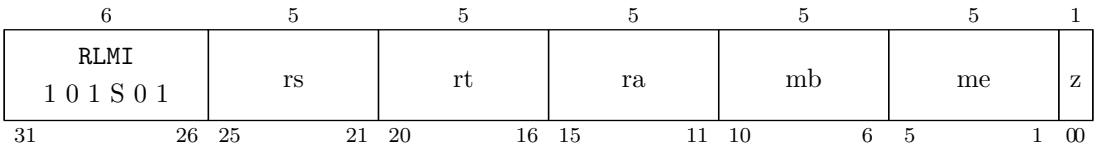
Rotate Left and Mask



mask ← (*create-mask* mb me z)
$[\texttt{rt}]_{31..0} \leftarrow$ (*left-rotate* $[\texttt{rs}]_{31..0}$ ra) & mask

RLMI Rotate Left and Masked Insert

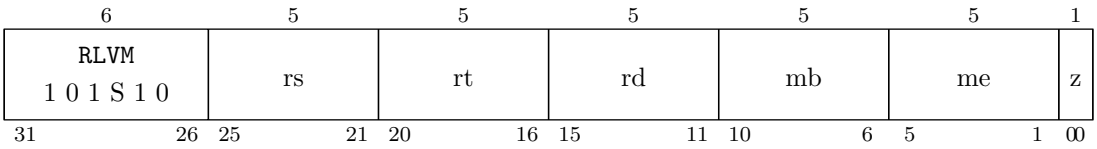
(1)




```
mask    ← (create-mask mb me z)
[rt]31..0 ← ((left-rotate [rs]31..0 ra) & mask) | ([rt]31..0 & ~mask)
```

RLVM Rotate Left Variable and Mask

(1)




```
mask    ← (create-mask mb me z)
ra      ← [rt]31..0
[rd]31..0 ← (left-rotate [rs]31..0 ra) & mask
```

RRM Rotate Right and Mask (Assembly Macro)

(1)

rrm rt, rs, ra, mask
rrm! rt, rs, ra, mask

```
[rt]31..0 ← (right-rotate [rs]31..0 ra) & mask
```

(instruction is implemented using RLM; same set of masks are valid)

RRMI Rotate Right and Mask (Assembly Macro)

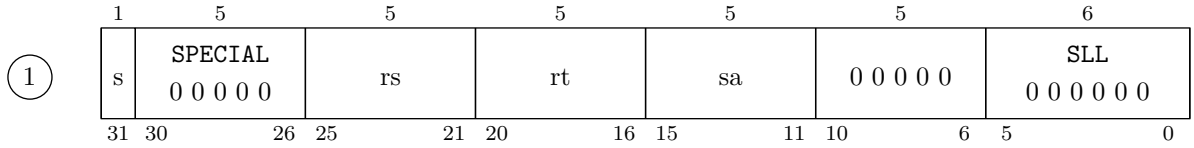
(1)

rrmi rt, rs, ra, mask
rrmi! rt, rs, ra, mask

```
[rt]31..0 ← ((right-rotate [rs]31..0 ra) & mask) | ([rt]31..0 & ~mask)
```

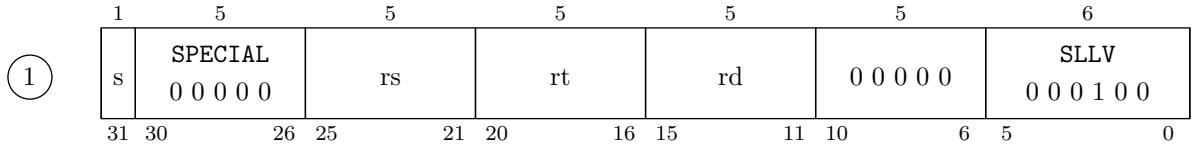
(instruction is implemented using RLMI; same set of masks are valid)

SLL — Shift Left Logical — MIPS



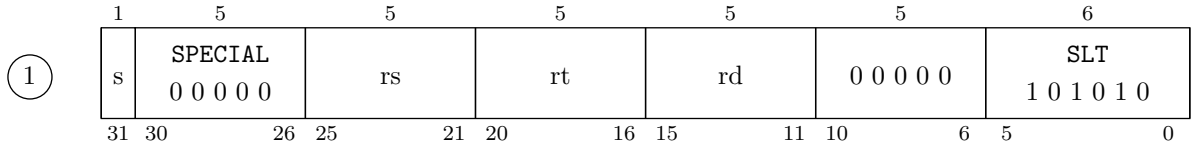
$[\texttt{rt}]_{31..sa} \leftarrow [\texttt{rs}]_{(31-sa)..0}$
$[\texttt{rt}]_{(sa-1)..0} \leftarrow \texttt{0}$

SLLV — Shift Left Logical Variable — MIPS



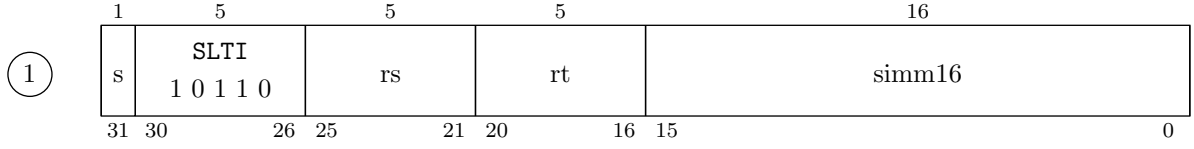
$\texttt{sa} \leftarrow [\texttt{rt}]_{4..0}$
$[\texttt{rd}]_{31..sa} \leftarrow [\texttt{rs}]_{(31-sa)..0}$
$[\texttt{rd}]_{(sa-1)..0} \leftarrow \texttt{0}$

SLT — Set Less Than Signed — MIPS



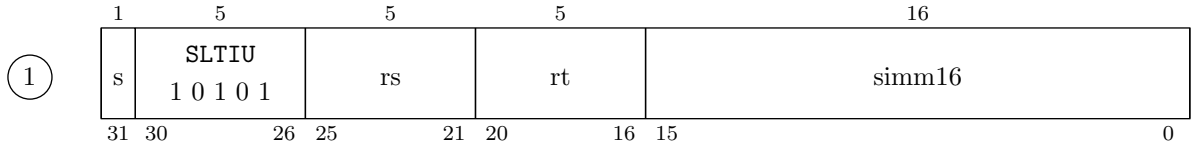
$[\texttt{rd}]_{00} \leftarrow ([\texttt{rs}]_{31..00} <_{signed} [\texttt{rt}]_{31..00})\ \texttt{?}\ \texttt{1}\ \texttt{:}\ \texttt{0}$
$[\texttt{rd}]_{31..01} \leftarrow \texttt{0}$

SLTI — Set Less Than Immediate Signed — MIPS



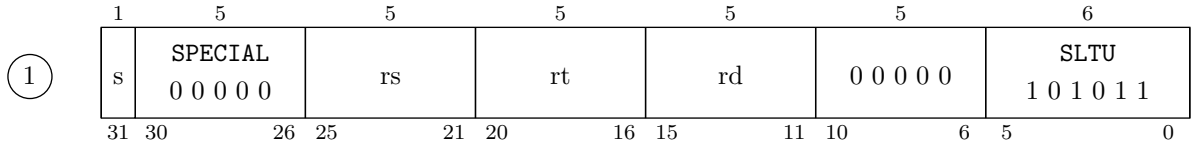
$\texttt{simm32} \leftarrow (\textit{sign-extend-16-to-32}\ \texttt{simm16})$
$[\texttt{rt}]_{00} \leftarrow ([\texttt{rs}]_{31..00} <_{signed} \texttt{simm32}_{31..00})\ \texttt{?}\ \texttt{1}\ \texttt{:}\ \texttt{0}$
$[\texttt{rt}]_{31..01} \leftarrow \texttt{0}$

SLTIU — Set Less Than Immediate Unsigned — MIPS



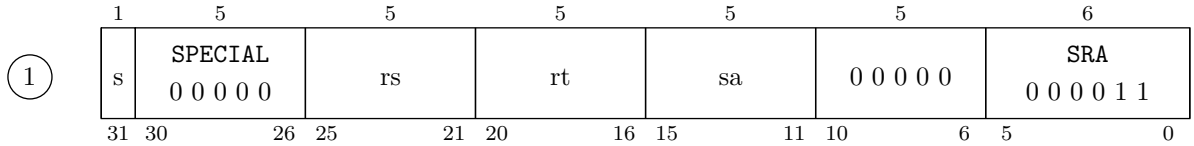

```
uimm32    ← (sign-extend-16-to-32 simm16)
[rt]00    ← ([rs]31..00 <unsigned uimm3231..00) ? 1 : 0
[rt]31..01 ← 0
```

SLTU — Set Less Than Unsigned — MIPS



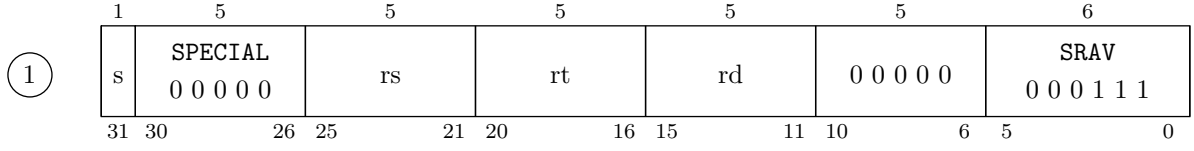

```
[rd]00    ← ([rs]31..00 <unsigned [rt]31..00) ? 1 : 0
[rd]31..01 ← 0
```

SRA — Shift Right Arithmetic — MIPS




```
[rt](31−sa)..00 ← [rs]31..sa
[rt]31..(31−sa) ← [rs]31
```

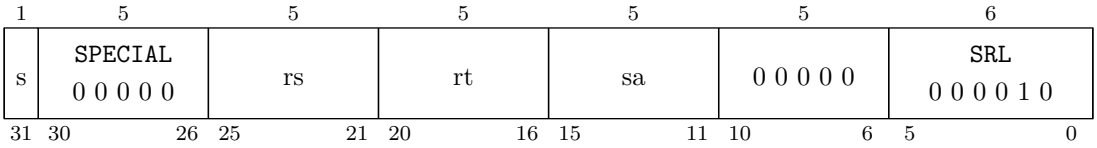
SRAV — Shift Right Arithmetic Variable — MIPS




```
sa             ← [rt]4..0
[rd](31−sa)..00 ← [rs]31..sa
[rd]31..(31−sa) ← [rs]31
```

SRL — Shift Right Logical — MIPS

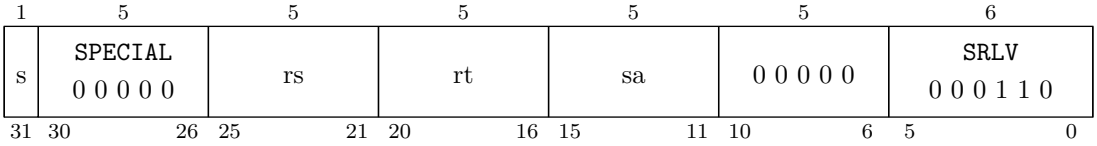
(1)



$[rt]_{(31-sa)..0} \leftarrow [rs]_{31..sa}$
$[rt]_{(31..(32-sa)} \leftarrow 0$

SRLV — Shift Right Logical Variable — MIPS

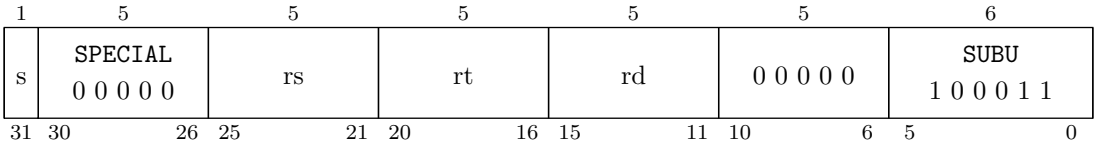
(1)



$sa \leftarrow [rt]_{4..0}$
$[rd]_{(31-sa)..00} \leftarrow [rs]_{31..sa}$
$[rd]_{31..(32-sa)} \leftarrow 0$

SUBU — Subtract — MIPS

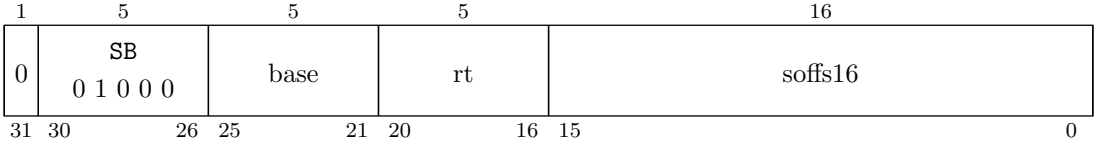
(1)



$[rd]_{31..00} \leftarrow \{\ [rs]_{31..00}\ -\ [rt]_{31..00}\ \}_{31..0}$

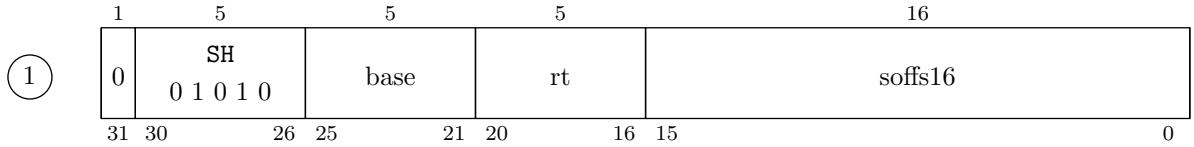
SB — Store Byte — MIPS

(1)



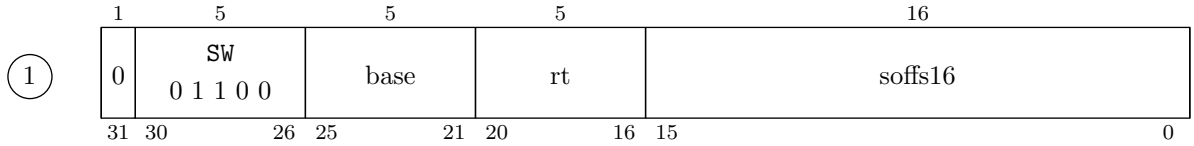
$ea \leftarrow \{$ [base] + (*sign-extend-16-to-32* soffs16) $\}_{31..0}$
(*cache-write-byte* ea [rt])

SH — Store Halfword — MIPS



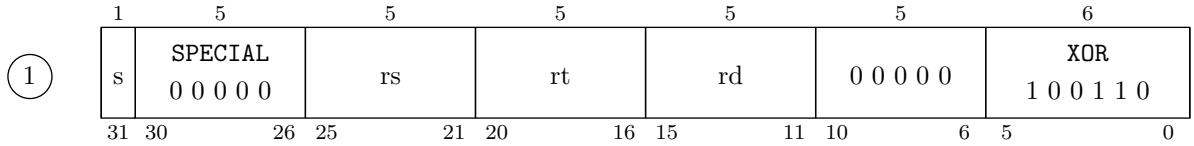

```
ea    ← { [base] + (sign-extend-16-to-32 soffs16) }31..0
(cache-write-half-word ea [rt])
```

SW — Store Word — MIPS



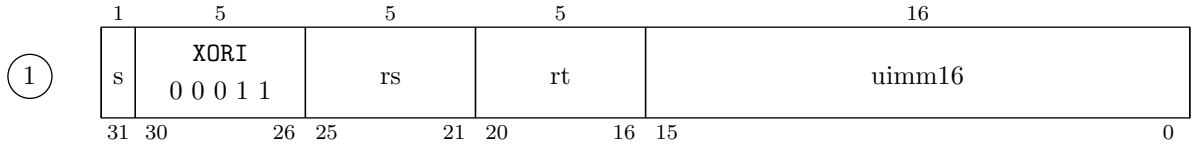

```
ea    ← { [base] + (sign-extend-16-to-32 soffs16) }31..0
(cache-write-word ea [rt])
```

XOR — Exclusive-Or Bitwise — MIPS



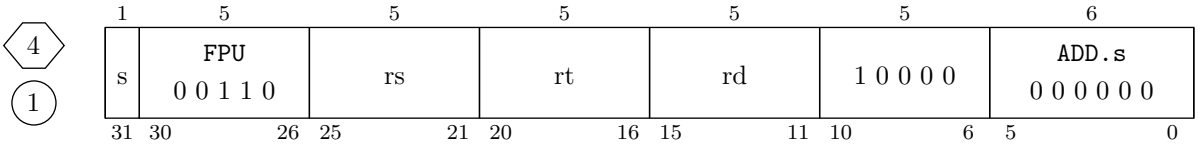
$[rd]_{31..0} \leftarrow [rs]_{31..00}$ ^ $[rt]_{31..00}$

XORI — Exclusive-Or Bitwise Immediate — MIPS



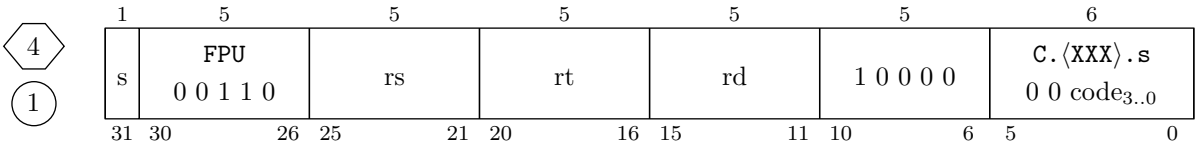
$[rt]_{31..16} \leftarrow [rs]_{31..16}$
$[rt]_{15..00} \leftarrow [rs]_{15..00}$ ^ uimm16

ADD.s Add (single precision floating point)



$[rd]_{31..0} \leftarrow [rs]_{31..0} +_{IEEE-754} [rt]_{31..0}$

C.⟨XXX⟩.s Compare (single precision floating point)



$\{invalid_0\ result_0\} \leftarrow (\textit{floating-point-compare}\ \langle XXX \rangle\ [rs]_{31..0}\ [rt]_{31..0})$
$[rd]_0 \leftarrow result_0$
$[rd]_{31..1} \leftarrow 0$
$SR[FPSR]_4 \leftarrow SR[FPSR]_4\ |\ invalid_0$

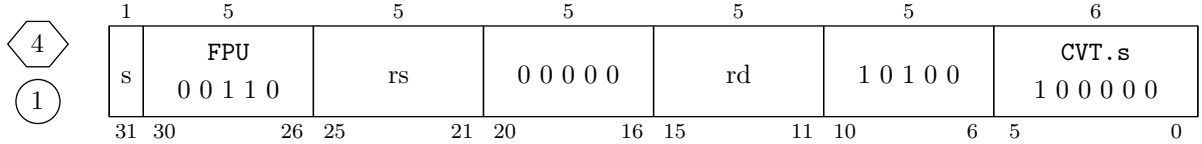
e.g.,

```
c.ult.s $4, $5, $7
```

The code values of 8..15 correspond to instructions that set the `invalid` bit of the floating point status register (`FPSR`) when an unordered comparison occurs. The behavior of the helper function *floating-point-compare* matches the MIPS ISA [43] and is shown in the following table:

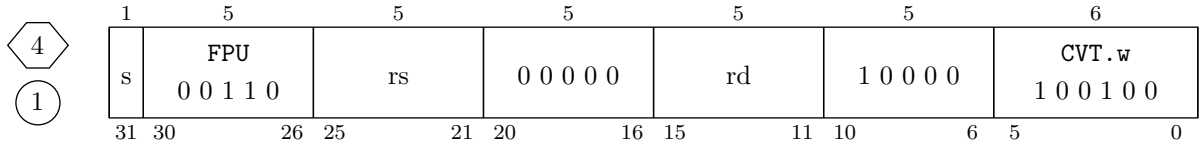
Predicate			*floating-point-compare* outputs for each comparison outcome					
			$result_0$				$invalid_0$	
code	Mnemonic ⟨XXX⟩	Description	>	<	==	unordered	>, <, ==	unordered
0	F	False	↑	0	0	0	↑	↑
1	UN	Unordered		0	0	1		
2	EQ	Equal		0	1	0		
3	UEQ	Unordered ==		0	1	1		0
4	OLT	Ordered <		1	0	0		
5	ULT	Unordered or <		1	0	1		
6	OLE	Ordered ≤		1	1	0		↓
7	ULE	Unordered or ≤		1	1	1		
			0				0	
8	SF	Signaling False		0	0	0		↑
9	NGLE	Not (> or ≤)		0	0	1		
10	SEQ	Signaling ==		0	1	0		
11	NGL	Not (< or >)		0	1	1		1
12	LT	<		1	0	0		
13	NGE	Not ≥		1	0	1		
14	LE	≤		1	1	0		↓
15	NGT	Not >	↓	1	1	1	↓	

CVT.s — Convert from integer to float



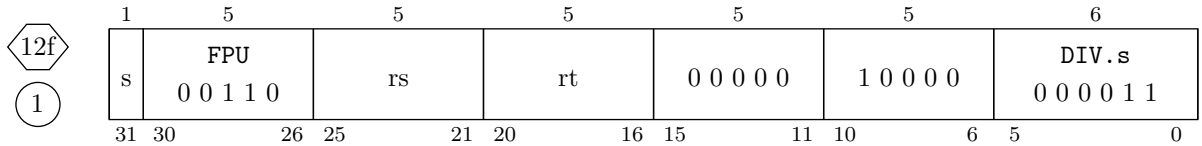
$[\texttt{rd}]_{31..0} \leftarrow (\textit{convert-from-integer-to-float}\ [\texttt{rs}]_{31..0})$

CVT.w — Convert from float to integer, with round to nearest even



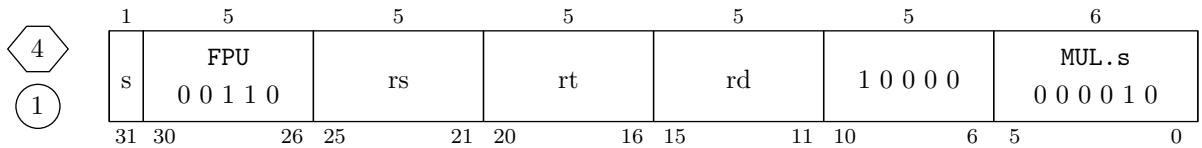
$[\texttt{rd}]_{31..0} \leftarrow (\textit{convert-from-float-to-integer-round-nearest-even}\ [\texttt{rs}]_{31..0})$

DIV.s — Divide (single precision floating point)



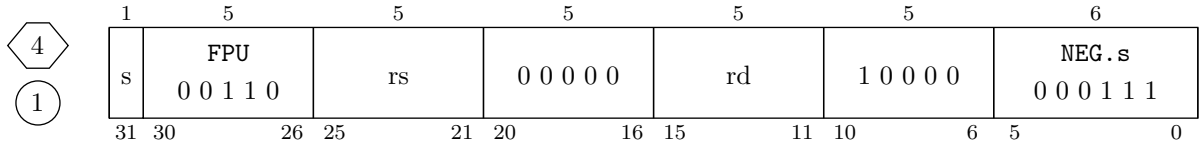
$\texttt{FD}_{31..0} \leftarrow [\texttt{rs}]_{31..0}\ /_{IEEE-754}\ [\texttt{rt}]_{31..0}$

MUL.s — Multiply (single precision floating point)



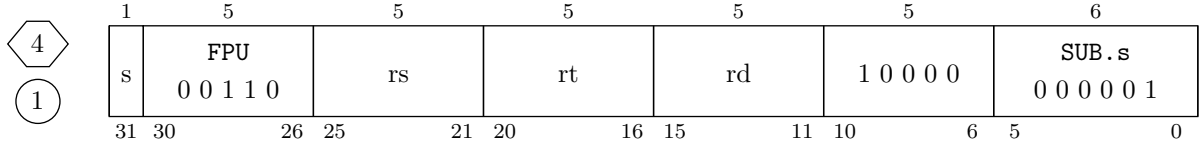
$[\texttt{rd}]_{31..0} \leftarrow [\texttt{rs}]_{31..0}\ *_{IEEE-754}\ [\texttt{rt}]_{31..0}$

NEG.s — Negate (single precision floating point)



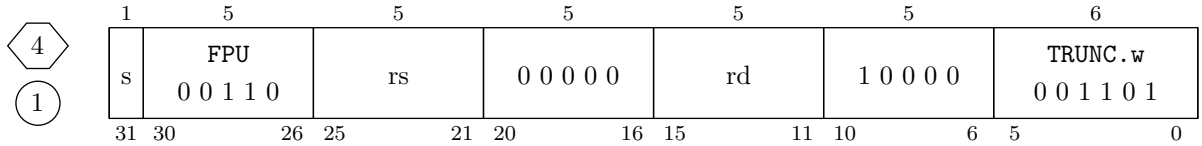
$[\texttt{rd}]_{31..0} \leftarrow \texttt{-}_{IEEE-754}\ [\texttt{rs}]_{31..0}$

SUB.s — Subtract (single precision floating point)



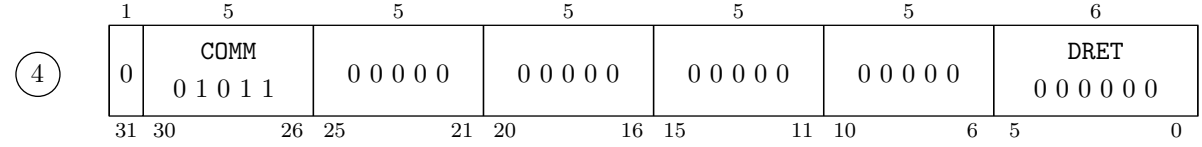
$[\texttt{rd}]_{31..0} \leftarrow [\texttt{rs}]_{31..0}\ \texttt{-}_{IEEE-754}\ [\texttt{rt}]_{31..0}$

TRUNC.w — Convert from float to integer, with truncation



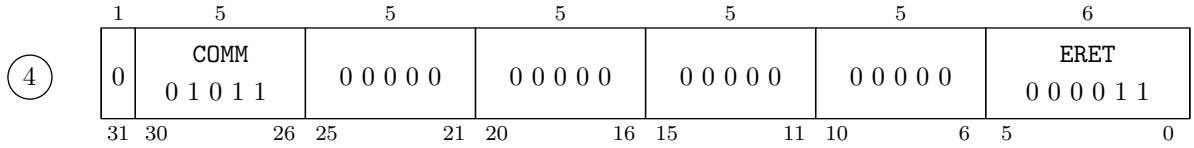
$[\texttt{rd}]_{31..0} \leftarrow (\textit{convert-from-float-to-integer-truncate}\ [\texttt{rs}]_{31..0})$

DRET — Return from User Interrupt



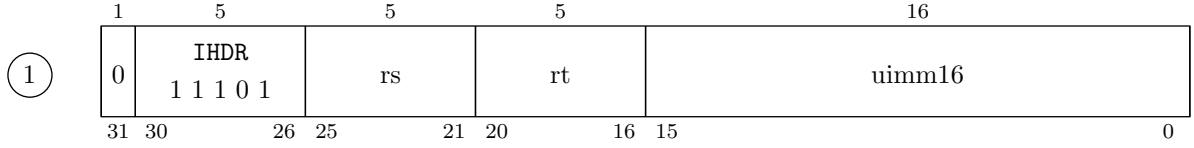
$\texttt{PC}_{31..02} \leftarrow \texttt{SR[EX\_UPC]}_{31..02}$
$\texttt{PC}_{01..00} \leftarrow \texttt{0}$
$\texttt{SR[EX\_BITS]}_{31} \leftarrow \texttt{1'b1}$

ERET Return from System Interrupt

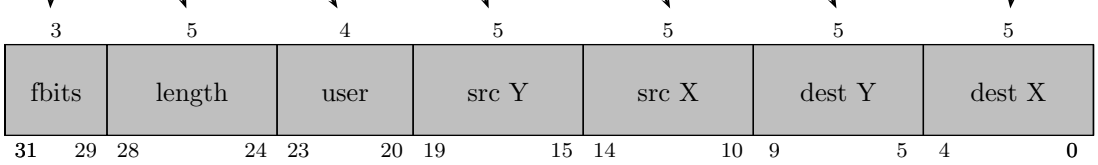


$PC_{31..02} \leftarrow SR[EX\_PC]_{31..02}$
$PC_{01..00} \leftarrow 0$
$SR[EX\_BITS]_{30} \leftarrow 1'b1$

IHDR Create Internal Header



$[rt]_{31..29} \leftarrow uimm16_{15..13}$
$[rt]_{28..24} \leftarrow uimm16_{12..8}$
$[rt]_{23..20} \leftarrow uimm16_{7..4}$
$[rt]_{19..15} \leftarrow DN\_YPOS_{4..0}$
$[rt]_{14..10} \leftarrow DN\_XPOS_{4..0}$
$[rt]_{09..05} \leftarrow ([rs]_{4..0}$ & $GDN\_XMASK_{4..0}) + GDN\_XADJ_{4..0}$
$[rt]_{04..00} \leftarrow (([rs]_{11..0} >> GDN\_YSHIFT_{2..0})$
& $GDN\_YMASK_{4..0})$
$+ GDN\_YADJ_{4..0}$

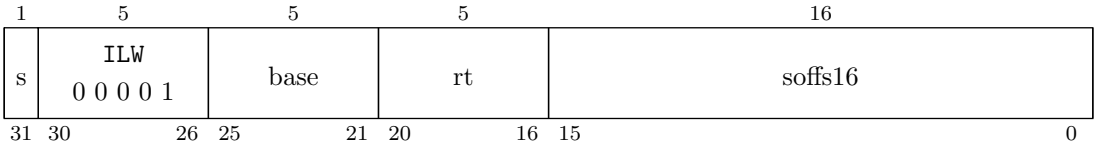


ILW

Instruction Load Word

5

2



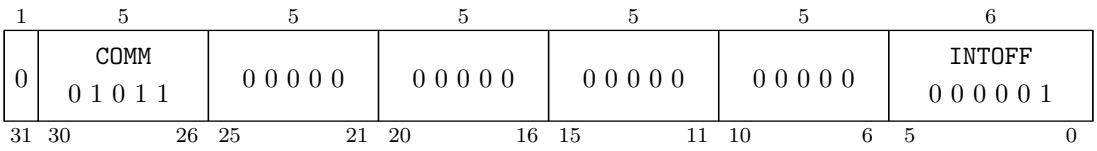

```
ea31..2  ← { [base] + (sign-extend-16-to-32 soffs16) }31..2
ea1..0   ← 0
[rt]     ← (proc-imem-load ea)
```

The additional cycle of occupancy is a cycle stolen from the fetch unit on access.

INTOFF

Disable System Interrupts

1

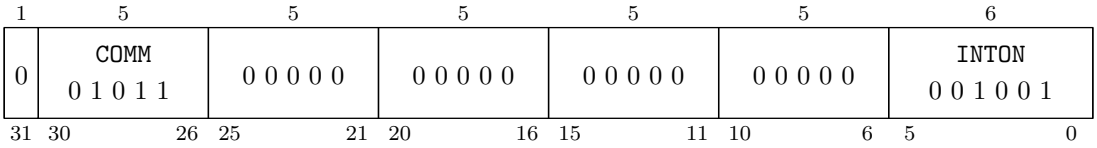



```
SR[EX_BITS]30 ← 1'b0
```

INTON

Enable System Interrupts

1

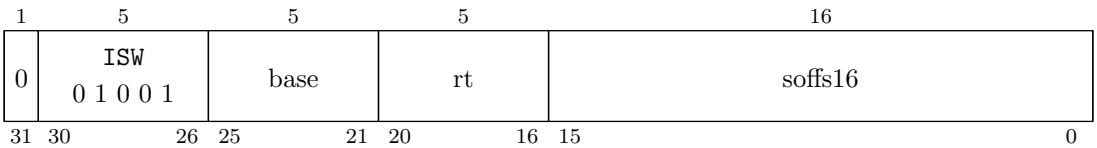



```
SR[EX_BITS]30 ← 1'b1
```

ISW

Instruction Store Word

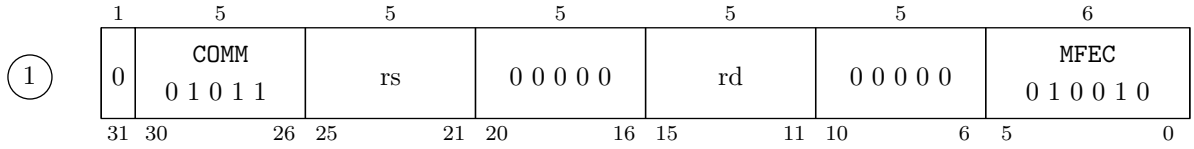
2




```
ea31..2  ← { [base] + (sign-extend-16-to-32 soffs16) }31..2
ea1..0   ← 0
(proc-imem-store ea [rt])
```

Steals one fetch cycle from compute processor fetch unit.

MFEC Move From Event Counter



(1)

$[rd]_{31..00} \leftarrow$ EC[rs]

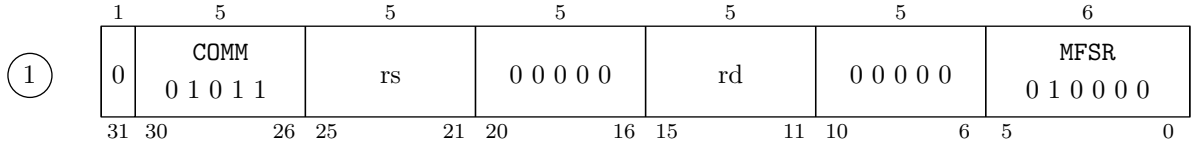
Note: MFEC captures its value in the RF stage. This is because the event counters are located physically quite distant from the bypass paths of the processor, so the address is transmitted in RF, and the output given in EXE. For example,

```
lw $0,4($0)             # cache miss in TV stage, pipeline frozen
nop                     # occupies TL stage
mfec $4, EC_CACHE_MISS  # EXE stage -- will not register cache miss
mfec $4, EC_CACHE_MISS  # RF        -- will register cache miss
```

Additionally, there is one cycle of lag between when the event actually occurs and when the event counter is updated. For example, assuming no outside stalls like cache misses or interrupts,

```
mtec EC_xxx, $4         # write an event counter
mfec $5, EC_xxx         # reads old value
mfec $5, EC_xxx         # reads new value
```

MFSR Move From Status / Control Register

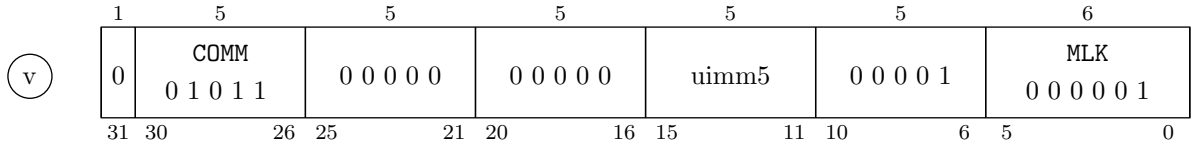


(1)

$[rd]_{31..00} \leftarrow$ SR[rs]

Section B.4 describes the status registers.

MLK MDN Lock **RawH**

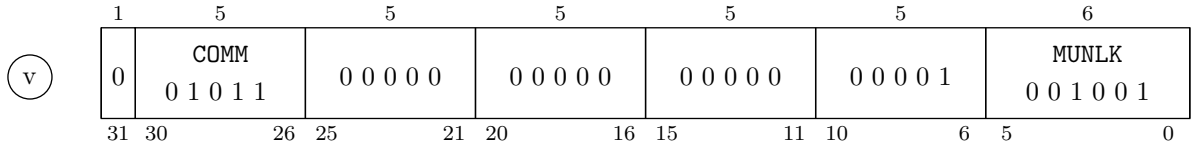


(v)

SR[EX_BITS]$_{30} \leftarrow$ 1'b0;
(*icache-prefetch* PC uimm5)

Signals to hardware or software caching system that the following `uimm5` cache lines needs to be resident in the instruction cache for correct execution to occur. Disables interrupts. This allows instruction sequences to access the memory network without concern that the i-caching system will also access it.

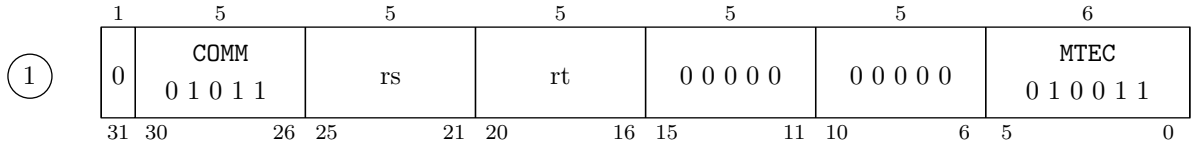
MUNLK MDN Unlock **RawH**



$\texttt{SR[EX\_BITS]}_{30} \leftarrow \texttt{1'b1}$

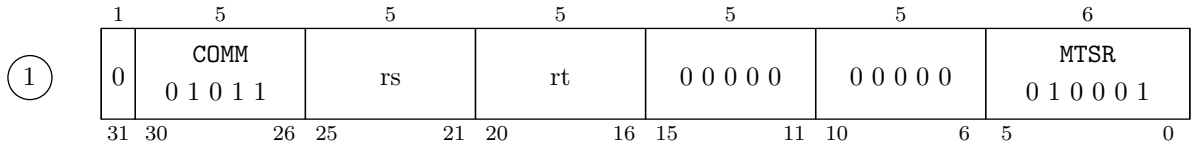
Marks end of MDN-locked region. Enables interrupts.

MTEC Move To Event Counter



$\texttt{EC[rt]} \leftarrow \texttt{[rs]}_{31..00}$

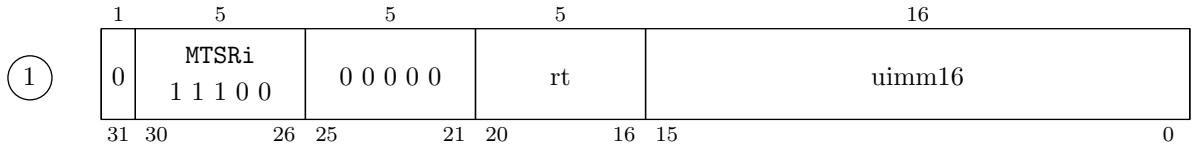
MTSR Move To Status / Control Register



$\texttt{SR[rt]} \leftarrow \texttt{[rs]}_{31..00}$

Section B.4 describes the status registers. Note that not all status register bits are fully writable, so some bits may not be updated as a result of an `MTSR` instruction.

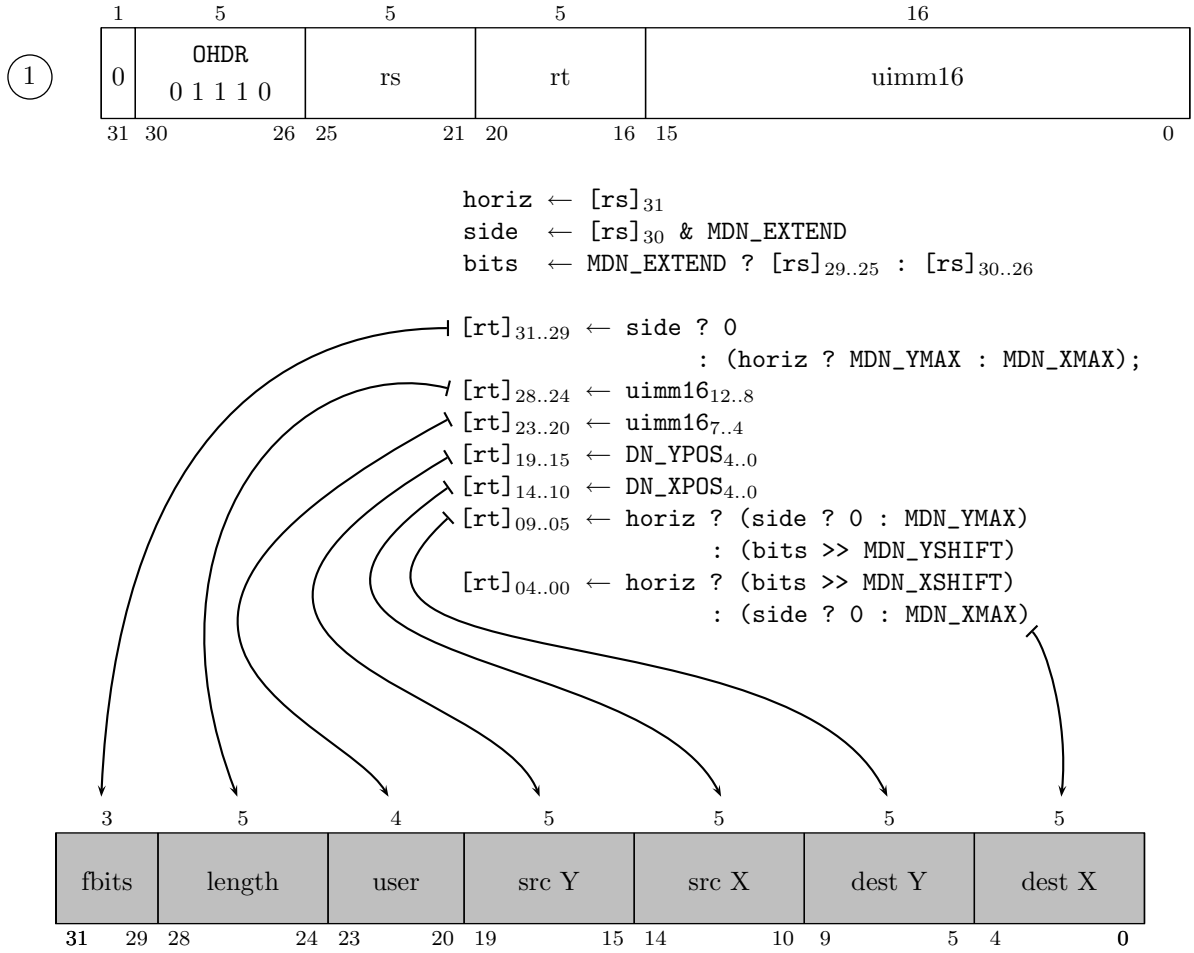
MTSRi Move to Status / Control Immediate



$\texttt{SR[rt]}_{31..16} \leftarrow \texttt{0;}$
$\texttt{SR[rt]}_{15..00} \leftarrow \texttt{uimm16;}$

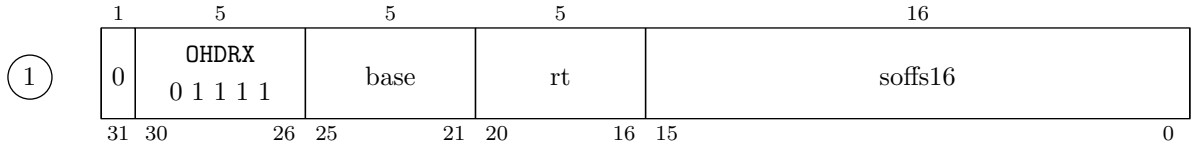
Section B.4 describes the status registers. Note that not all status register bits are fully writable, so some bits may not be updated as a result of an `MTSR` instruction.

OHDR Create Outside Header

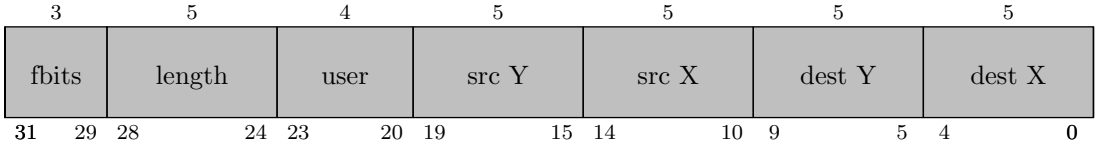


`OHDR` takes an address and an immediate field and produces a header suitable for injecting into the MDN. The immediate field specifies the `user` and `length` fields of the message header. `OHDR` maps the address to an I/O port, which effectively wraps the address space around the periphery of the chip. Raw's hardware data cache uses a private copy of this logic to implement Raw's *memory hash function*. (See Section 2.6.)

OHDRX Create Outside Header; Disable System Interrupts

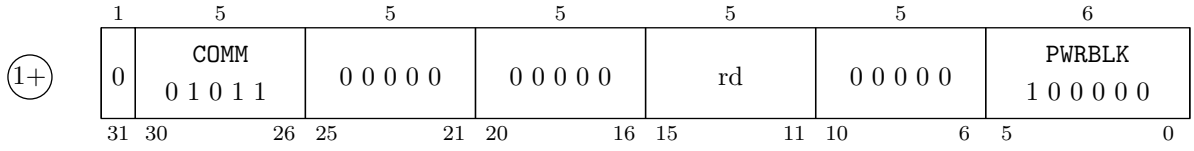



```
horiz ← [rs]31
side  ← [rs]30 & MDN_EXTEND
bits  ← MDN_EXTEND ? [rs]29..25 : [rs]30..26

SR[EX_BITS]30 ← 1'b0

[rt]31..29 ← side ? 0
                  : (horiz ? MDN_YMAX : MDN_XMAX);
[rt]28..24 ← uimm16 12..8
[rt]23..20 ← uimm16 7..4
[rt]19..15 ← DN_YPOS4..0
[rt]14..10 ← DN_XPOS4..0
[rt]09..05 ← horiz ? (side ? 0 : MDN_YMAX)
                   : (bits >> MDN_YSHIFT)
[rt]04..00 ← horiz ? (bits >> MDN_XSHIFT)
                   : (side ? 0 : MDN_XMAX)
```



OHDRX takes an address and an immediate field and produces a header suitable for injecting into the MDN. The immediate field specifies the `user` and `length` fields of the message header. OHDRX maps the address to an I/O port, which effectively wraps the address space around the periphery of the chip. Since the MDN must be accessed with interrupts disabled, OHDRX provides a cheap way of doing this. Raw's hardware data cache uses a private copy of this logic to implement Raw's *memory hash function*. (See Section 2.6.)

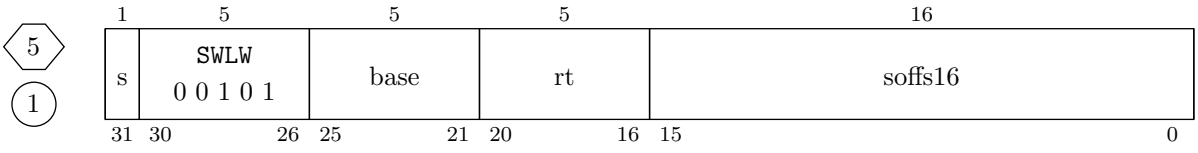
PWRBLK Power Block



$[rd]_{31}$ ← data available in cgni
$[rd]_{30..13}$ ← 0;
$[rd]_{12}$ ← data available in cNi
$[rd]_{11}$ ← data available in cEi
$[rd]_{10}$ ← data available in cSi
$[rd]_{9}$ ← data available in cWi
$[rd]_{8}$ ← data available in csti
$[rd]_{7}$ ← data available in cNi2
$[rd]_{6}$ ← data available in cEi2
$[rd]_{5}$ ← data available in cSi2
$[rd]_{4}$ ← data available in cWi2
$[rd]_{3}$ ← data available in csti2
$[rd]_{2}$ ← data available in cmni
$[rd]_{1}$ ← timer interrupt went off
$[rd]_{0}$ ← external interrupt went off

Stalls in RF stage until output is non-zero.

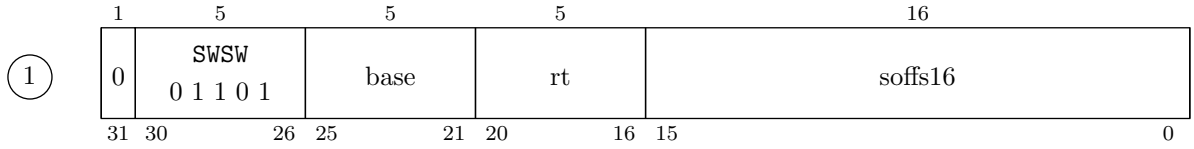
SWLW Switch Load Word



$ea_{31..2}$ ← { [base] + (*sign-extend-16-to-32* soffs16) }$_{31..2}$
$ea_{1..0}$ ← 0
[rt] ← (*static-router-imem-load* ea)

Steals one fetch cycle from static router.

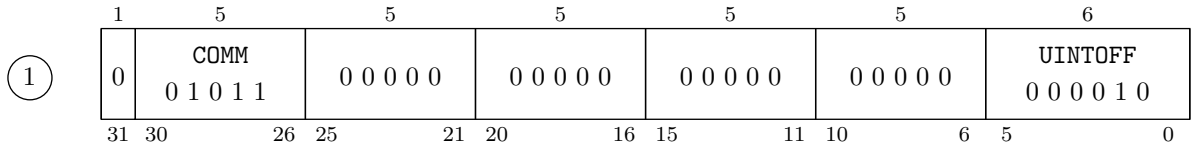
SWSW Switch Store Word




```
ea31..2  ← { [base] + (sign-extend-16-to-32 soffs16) }31..2
ea1..0   ← 0
(static-router-imem-store ea [rt])
```

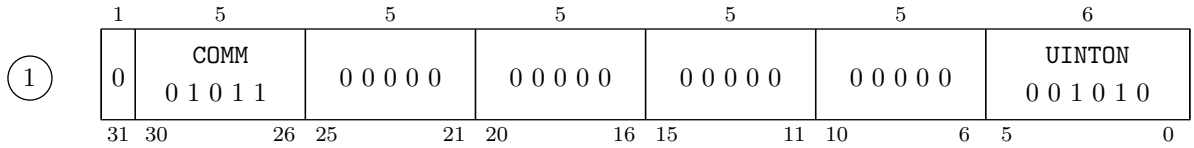
Steals one fetch cycle from static router.

UINTOFF Disable User Interrupts




```
SR[EX_BITS]31 ← 1'b0;
```

UINTON Enable User Interrupts




```
SR[EX_BITS]31 ← 1'b1;
```

B.1.3 Cache Management in Raw and RawH

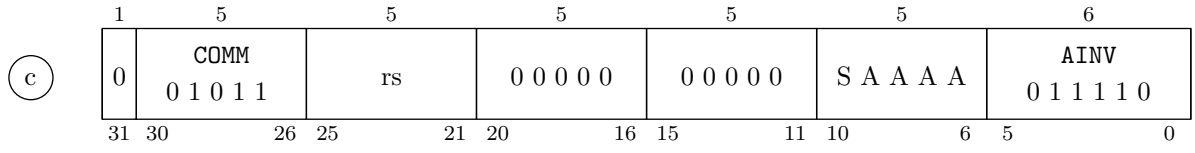
Because Raw's memory model is shared memory but not hardware cache-coherent, effective and fast software cache management instructions are essential. One tile may modify a data structure through its caching system, and then want to make it available to a consuming tile or I/O device. To accomplish this in a cache-coherent way, the sender tile must explicitly flush and/or invalidate the data, and then send an MDN Relay message that bounces off the relevant DRAM I/O Port (indicating that all of the memory accesses have reached the DRAM) to the consumer. The consumer then knows that the DRAM has been updated with the correct values.

To provide effective cache management, there are two series of cache management instructions. Both series allow cache lines to be flushed and/or invalidated. The first series, `ainv`, `afl`, and `aflinv`, takes as input a data address. This address, if it is resident in the cache, is translated into a <set, line> which is used to identify the physical cache line. The second series of instructions, `tagsw`, `taglv`, `tagla`, and `tagfl`, takes a <set, line> pair directly.

The address-based instructions are most effective when the range of addresses residing in the cache is relatively small. If $|A|$ is the size of the address range that needs to be flushed, this series can flush the range in time $\theta(|A|)$.

The tag-based instructions are most effective when the processor needs to invalidate or flush large ranges of address space that exceed the cache size. In this case, the address range can be manipulated faster by using the tag-based instructions to scan the tags of the cache and selectively invalidate and/or flush the contents. In this case, the operations can occur in $\theta(|C|)$, where $|C|$ is the size of the cache. The `tagla` and `taglv` operations allow the cache line tags to be inspected, `tagfl` can be used to flush the contents, and `tagsw` can be used to rewrite (or zero) the tags. Of course, the `tag`xxx series of instructions can accomplish more than simply flushing or invalidating. They provide an easy way to manipulate the cache state directly for verification purposes and boot-time cache initialization.

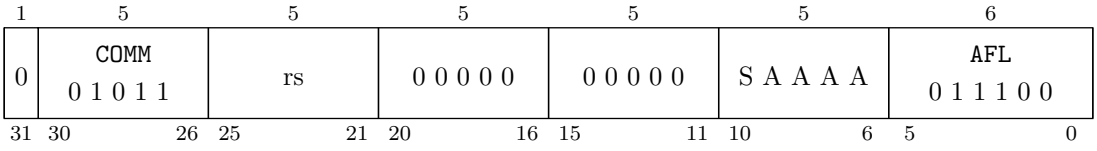
AINV Address Invalidate




```
ea ← [rs] + (S << 14) + (AAAA << kDataCacheLineSize)

if (cache-contains ea)
   TAGS[(cache-get-tag ea)].valid ← 0    # stall 4 cycles
```

AFL Address Flush

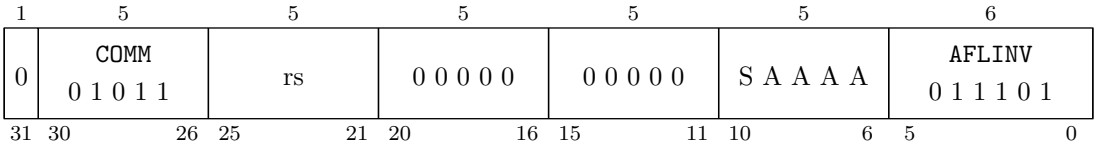
(c)




```
ea ← [rs] + (S << 14) + (AAAA << kDataCacheLineSize)

if (cache-contains ea)
{
  <set,line> ← (cache-get-tag ea)
  TAGS[<set,line>].mru ← !set

  if (TAGS[<set,line>].dirty)
  {
     TAGS[<set,line>].dirty ← 0
     (cache-copy-back <set,line>)    # stall >= 13 cycles
  }
  else
     # stall 5 cycles
}
```

AFLINV Address Flush and Invalidate

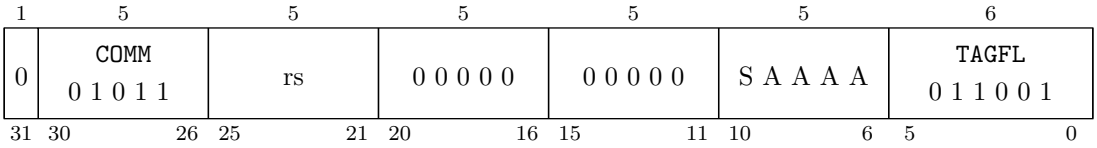
(c)




```
ea ← [rs] + (S << 14) + (AAAA << kDataCacheLineSize)

if (cache-contains ea)
{
  <set,line> ← (cache-get-tag ea)
  if (TAGS[<set,line>].dirty)
  {
     TAGS[<set,line>].dirty ← 0
     TAGS[<set,line>].valid ← 0
     (cache-copy-back ea)    # stall >= 13 cycles
  }
  else
     # stall 5 cycles
}
```

TAGFL Tag Flush

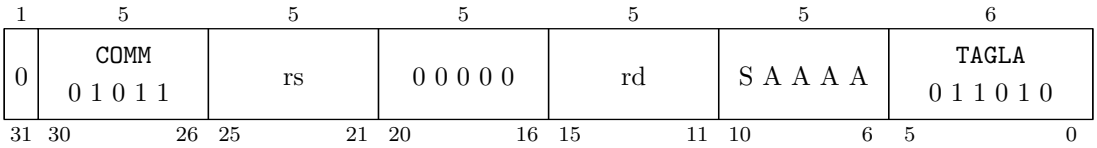
c




```
set      ← [rs]14 ^ S
line8..0 ← [rs]13..5 + AAAA

if (TAGS[<set,line>].valid)
{
  TAGS[<set,line>].mru ← !set
  if (TAGS[<set,line>].dirty)
  {
     TAGS[<set,line>].dirty ← 0
     (cache-copy-back <set,line>)    # stall >= 13 cycles
  }
  else
     # stall 5 cycles
}
```

TAGLA Tag Load Address

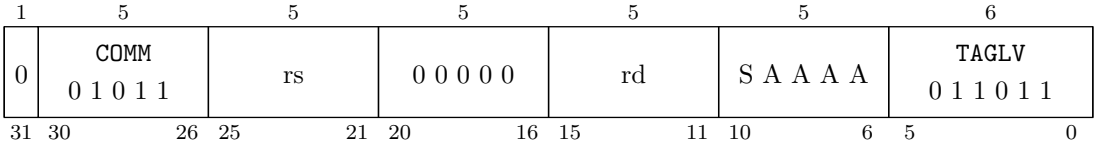
3
1




```
set      ← [rs]14 ^ S
line8..0 ← [rs]13..5 + AAAA

[rd] ← { TAGS[<set,line>].addr17..00 line8..0 [rs]4..0 }
```

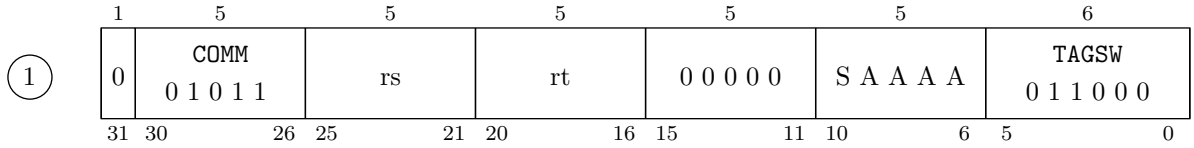
TAGLV Tag Load Valid

3
1




```
set     ← [rs]14 ^ S
line8..0 ← [rs]13..5 + AAAA

[rd] ← TAGS[<set,line>].valid
```

TAGSW Tag Store Word




```
set     ← [rs]14 ^ S
line8..0 ← [rs]13..5 + AAAA

TAGS[<set,line>].valid0     ← [rt]18
TAGS[<set,line>].addr17..00 ← [rt]17..00
TAGS[<set,line>].dirty0     ← 0

Should not be issued the cycle after a load or store instruction
because of write-after-write hazards on the tag memory.
```

B.2 Semantic Helper Functions

This section gives the semantics of the helper-functions used in the previous section. This thesis uses little-endian bit-ordering exclusively.

$\texttt{w}_{x..y}$	$\longrightarrow$	Bits `x..y`, inclusive, of `w`. If (`x < y`), the empty string.
`{ w z }`	$\longrightarrow$	Concatenate the bits of `w` and `z` together. `w` will occupy the more significant bits.
$\texttt{z}^{n}$	$\longrightarrow$	Concatenate `n` copies of `z` together.
(*sign-extend-16-to-32* simm16)	$\longrightarrow$	$\{\ (\texttt{simm16}_{15})^{16}\ \texttt{simm16}_{15..00}\ \}$
(*sign-extend-26-to-30* simm26)	$\longrightarrow$	$\{\ (\texttt{simm26}_{25})^{4}\ \ \texttt{simm26}_{25..00}\ \}$
(*sign-extend-16-to-30* simm16)	$\longrightarrow$	$\{\ (\texttt{simm16}_{15})^{14}\ \texttt{simm16}_{15..00}\ \}$
(*zero-extend-16-to-32* uimm16)	$\longrightarrow$	$\{\ 0^{15..0}\ \texttt{uimm16}_{15..00}\ \}$
(*left-rotate* uimm32 ra)	$\longrightarrow$	$\{\ \texttt{uimm32}_{(31-ra)..0}\ \texttt{uimm32}_{31..(32-ra)}\ \}$
(*right-rotate* uimm32 ra)	$\longrightarrow$	$\{\ \texttt{uimm32}_{(ra-1)..0}\ \ \texttt{uimm32}_{31..ra}\ \}$
(*cache-contains* addr)	$\longrightarrow$	Returns 1 if valid cache line corresponding to `addr` is in cache, otherwise 0.
(*cache-get-tag* addr)	$\longrightarrow$	Returns `<set,line>` pair corresponding to `addr` in cache.
(*cache-copy-back* tagid)	$\longrightarrow$	Sends update message containing data corresponding to `tagid` to owner DRAM.
(*cache-read-byte* addr)	$\longrightarrow$	Ensure cache line corresponding to `addr` is in cache, return byte at `addr`.
(*cache-read-half-word* addr)	$\longrightarrow$	Ensure cache line corresponding to `addr` is in cache, return half-word at $\{\ \texttt{addr}_{31..1}\ 0^{1}\ \}$.
(*cache-read-word* addr)	$\longrightarrow$	Ensure cache line corresponding to `addr` is in cache, return word at $\{\ \texttt{addr}_{31..2}\ 0^{2}\ \}$.
(*cache-write-byte* addr val)	$\longrightarrow$	Ensure cache line corresponding to `addr` is in cache, write $\texttt{val}_{7..0}$ to `addr`.
(*cache-write-half-word* addr val)	$\longrightarrow$	Ensure cache line corresponding to `addr` is in cache, write $\texttt{val}_{15..0}$ to $\{\ \texttt{addr}_{31..1}\ 0^{1}\ \}$.
(*cache-write-word* addr val)	$\longrightarrow$	Ensure cache line corresponding to `addr` is in cache, write $\texttt{val}_{31..0}$ to $\{\ \texttt{addr}_{31..2}\ 0^{2}\ \}$.

(*create-mask* mb me z)	$\longrightarrow$	`if (z)`
		`  if (me`$_{1..0}$ `== 0b00)` $\{\ \texttt{mb}_{4..0}\ \texttt{me}_{4..2}\ \}^{4}$
		`  if (me`$_{1..0}$ `== 0b11)`
		$\{\ \texttt{mb}_{4}{}^{4}\ \texttt{mb}_{3}{}^{4}\ \texttt{mb}_{2}{}^{4}\ \texttt{mb}_{1}{}^{4}\ \texttt{mb}_{0}{}^{4}\ \texttt{me}_{4}{}^{4}\ \texttt{me}_{3}{}^{4}\ \texttt{me}_{2}{}^{4}\ \}$
		`else`
		`  if (mb <=`$_{unsigned}$ `me)`
		$\{\ 0^{31..(me+1)}\ 1^{me..mb}\ 0^{(mb-1)..0}\ \}$

```
else
    { 1^{31..(mb+1)} 0^{mb..me} 1^{(me-1)..0} }
```

The last line was a specification bug as it does not generate every mask with a single zero. A better version is:

```
    { 1^{31..(mb+1)} 0^{mb..me+1} 1^{(me)..0} }
```

(*icache-prefetch* `addr lines`) ⟶ Ensure `lines` instruction cache lines following cache line containing `addr` are resident in instruction cache.

(*static-router-imem-store* `addr data`) ⟶ Writes 32-bit value `data` into static router instruction cache at location `addr`.

(*static-router-imem-load* `addr data`) ⟶ Loads 32-bit value `data` from static router instruction cache at location `addr`.

(*proc-imem-store* `addr data`) ⟶ Writes 32-bit value `data` into static router instruction cache at location `addr`.

(*proc-imem-load* `addr data`) ⟶ Loads 32-bit value `data` from static router instruction cache at location `addr`.

B.3 Opcode Maps

Below are opcode maps which document the allocation of instruction encoding space.

B.3.1 High-Level ("Opcode") Map

(Instructions with bits 31..29 set to 1 are predicted taken.) Bang instructions all have bit 31 set. however RLM, RLMI and RLVM use bit 28 to indicate bang.

bits	bits 28..26							
31..29	000	001	010	011	100	101	110	111
111	**REGIMM+**	BNEA+	BNE+	BEQ+	BL	BLAL	JNEL+	JEQL+
110	LB!	LBU!	LH!	LHU!	LW!	SLTIU!	SLTI!	ADDIU!
101	RLM	RLMI	RLVM		RLM!	RLMI!	RLVM!	
100	**SPECIAL!**	ILW!	ORI!	XORI!	ANDI!	SWLW!	**FPU!**	AUI!
011	**REGIMM-**	BNEA-	BNE-	BEQ-	MTSRI	IHDR	JNEL-	JEQL-
010	LB	LBU	LH	LHU	LW	SLTIU	SLTI	ADDIU
001	SB	ISW	SH	**COM**	SW	SWSW	OHDR	OHDRX
000	**SPECIAL**	ILW	ORI	XORI	ANDI	SWLW	**FPU**	AUI

B.3.2 SPECIAL Submap

(Applies when bits 31..26 are SPECIAL or SPECIAL!)

bits	bits 2..0							
5..3	000	001	010	011	100	101	110	111
000	SLL	MAGIC	SRL	SRA	SLLV		SRLV	SRAV
001	JR	JALR	JRHON	JRHOFF				
010	MFHI	MTHI	MFLO	MTLO	MFFD	MTFD		
011	MULLO	MULLU	DIV	DIVU				
100		ADDU		SUBU	AND	OR	XOR	NOR
101	MULHI	MULHU	SLT	SLTU				
110								
111	POPC	CLZ						

B.3.3 FPU Submap

(Applies when bits 31..26 are FPU or FPU!)

bits	bits 2..0							
5..3	000	001	010	011	100	101	110	111
000	ADD.s	SUB.s	MUL.s	DIV.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

B.3.4 COM Submap

(Applies when bits 31..26 are COM)

bits	bits 2..0							
5..3	000	001	010	011	100	101	110	111
000	DRET	INTOFF	UINTOFF	ERET				
001		INTON	UINTON					
010	MFSR	MTSR	MFEC	MTEC				
011	TAGSW	TAGFL	TAGLA	TAGLV	AFL	AFLINV	AINV	
100	PWRBLK							
101								
110								
111								

B.3.5 REGIMM Submap

(Applies when bits 31..26 are REGIMM+ or REGIMM-.) Bit 20 indicates a link instruction, and bit 18 indicates an absolute jump. The conditions are mirrored across these axes when appropriate.

bits	bits 18..16							
20..19	000	001	010	011	100	101	110	111
00	BLTZ	BLEZ	BGEZ	BGTZ				
01					J			
10	BLTZAL		BGEZAL		JLTZL	JLEZL	JGEZL	JGTZL
11					JAL			

B.4 Status and Control Registers

#	Status Reg Name	R/W	Purpose
0	SW_FREEZE	RW	Switch Processor is Frozen. [00] ← (1 Frozen) (0 Running)
1	SW_BUF1	R	# of elements in static router crossbar 1 NIBs [22:20] number of elements in c21 (≤ 4) [19:17] number of elements in cNi (≤ 4) [16:14] number of elements in cEi (≤ 4) [13:11] number of elements in cSi (≤ 4) [10:08] number of elements in cWi (≤ 4) [07:05] number of elements in csti (≤ 4) [04:00] number of elements in csto (≤ 8)
2	SW_BUF2	R	# of elements in static router crossbar 2 NIBs [22:20] number of elements in c12 (≤ 4) [19:17] number of elements in cNi_2 (≤ 4) [16:14] number of elements in cEi_2 (≤ 4) [13:11] number of elements in cSi_2 (≤ 4) [10:08] number of elements in cWi_2 (≤ 4) [07:05] number of elements in csti_2 (≤ 4) [04:00] number of elements in csto (≤ 8)
3	MDN_BUF	R	# of elements in MDN router NIBs [19:17] number of elements in cNi (≤ 4) [16:14] number of elements in cEi (≤ 4) [13:11] number of elements in cSi (≤ 4) [10:08] number of elements in cWi (≤ 4) [07:05] number of elements in cmni (≤ 4) [04:00] number of elements in cmno (≤ 16)
4	SW_PC	RW	Current PC of switch processor. Byte address aligned to eight-byte boundaries. Used primarily for context switching. Generally, writing to this register is used for context-switching purposes. It should only be performed when the switch is FROZEN or if the compute processor program knows absolutely that the switch is stalled at a known PC. Otherwise, the program can no longer assume the static ordering of operands on the SON. Writing to this register causes a branch misprediction in the switch. Allow at least three cycles for corresponding instruction to be executed.
5	BR_INCR	RW	Signed 32-bit increment value for BNEA instruction. Caller-saved.

#	Status Reg Name		Purpose
6	EC_DYN_CFG	RW	Configuration for Event Counting of Dynamic Network events [30:28] Memory Network North (D=N) Configuration [27:25] Memory Network East (D=E) Configuration [24:22] Memory Network South (D=S) Configuration [21:19] Memory Network West (D=W) Configuration [18:16] Memory Network Proc (D=P) Configuration [14:12] Memory Network North (D=N) Configuration [11:09] Memory Network East (D=E) Configuration [08:06] Memory Network South (D=S) Configuration [05:03] Memory Network West (D=W) Configuration [02:00] Memory Network Proc (D=P) Configuration
	Settings: 0 # of cycles output port D wants to transmit but could not because neighbor tile's input buffer is full. 1 # of words transmitted from input port D to output port P 2 # of words transmitted from input port D to output port W 3 # of words transmitted from input port D to output port S 4 # of words transmitted from input port D to output port E 5 # of words transmitted from input port D to output port N 6 # of words transmitted from input port D 7 # cycles input port D had data to transmit but was not able to		
7	WATCH_VAL	RW	[31:00] 32-bit timer; increments each cycle
8	WATCH_MAX	RW	[31:00] value to fire timer interrupt and then zero WATCH_VAL
9	WATCH_SET	RW	[00] zero WATCH_VAL if cgno is empty or a value was dequeued [01] zero WATCH_VAL if processor issues an instruction
10	CYCLE_HI	RW	[31:00] high 32-bits of cycle counter
11	CYCLE_LO	RW	[31:00] low 32-bits of cycle counter Note: To read the cycle counter efficiently, read CYCLE_HI, then CYCLE_LO, then subtract one from CYCLE_LO. Cycle counters are writable to make tests reproducible.
12	EVENT_CFG2	RW	[24:0] configures the set of events that causes c_trigger event counters to be incremented. See B.5.
13	GDN_RF_VAL	RW	[31:00] GDN refill value When EX_MASK[GDN_REFILL] is enabled, a read from $cgno will return GDN_RF_VAL, signal an interrupt by setting EX_BITS[GDN_REFILL], and leave cgno unchanged. This allows cgno to be virtualized, e.g. for context switches and deadlock recovery.

#	Status Reg Name		Purpose
14	GDN_REMAIN	RW	[04:00] Number of words remaining to be sent to complete current message on cgno. GDN_COMPLETE interrupt fires when value transitions to zero. OS typically initializes this with GDN_PENDING value to allow GDN messages to complete when context switching.
15	EX_BASE_ADDR	RW	[31:00] Pointer to beginning of exception vector table. Set to zero at boot time. Applies to **RawH**.
16	GDN_BUF	R	# of elements in GDN router NIBs [24:20] GDN_PENDING number of elements ($\leq$ 31) that need to be sent to cgno from processor pipeline to complete current message. Note this count does not include those instructions currently in the pipeline; the operating system should flush the pipeline before reading this value. The OS loads this value into the GDN_REMAIN SPR for the GDN_PENDING interrupt to trigger on. [19:17] number of elements ($\leq$ 4) in cNi [16:14] number of elements ($\leq$ 4) in cEi [13:11] number of elements ($\leq$ 4) in cSi [10:08] number of elements ($\leq$ 4) in cWi [07:05] number of elements ($\leq$ 4) in cgni [04:00] number of elements ($\leq$ 16) in cgno
17	GDN_CFG	RW	General Dynamic Network Configuration [31:27] GDN_XMASK - Masks X bits from an address [26:22] GDN_YMASK - Masks Y bits from an address [21:17] GDN_XADJ - Adjusts from local to global X address [16:12] GDN_YADJ - Adjusts from local to global Y address [11:09] GDN_YSHIFT - Gets Y bits from an address See `IHDR` instruction.

#	Status Reg Name		Purpose
18	STORE_METER	RW	STORE_ACK counters [31:27] PARTNER_Y - Y location of partner port [26:22] PARTNER_X - X location of partner port [21] ENABLE - enable store meter-based stalls [10] DECREMENT_MODE (see below; reads always zero) [9:5] COUNT_PARTNER - # of partner accesses left [4:0] COUNT_NON_PARTNER - # of non-partner accesses left Since the counts are updated as STORE_ACK messages are received over the MDN, care must be taken to update STORE_METER in a way that avoids race conditions. Ordinarily, the only way to do this is to modify the register only when all store-acks have been received. Alternatively, the user may write to the register with DECREMENT_MODE set; in this case the COUNT_NON_PARTNER will be decremented if bit 0 is set, and COUNT_PARTNER will be decremented if bit 5 is set. No other bits are changed. This handles the case where the user is directly transmitting memory packets over the MDN using explicit accesses to cmno, and needs to update the the STORE_ACK counters to reflect this.
19	MDN_CFG	RW	Memory Dynamic Network Configuration [31:27] DN_XPOS - Absolute X position of tile in array [26:22] DN_YPOS - Absolute Y position of tile in array [21:17] MDN_XMAX - X Coord of East-Most Tiles [16:12] MDN_YMAX - Y Coord of South-Most Tiles [11:09] MDN_XSHIFT - Shift Amount X [08:06] MDN_YSHIFT - Shift Amount Y [00:00] MDN_EXTEND - Use all four edge of chip. These SPRs are used to determine Raw's *memory hash function* as described in Section 2.6. This function determines where the data caches send their messages for cache fills and evictions. It also determines the functionality of the OHDR and OHDRX instructions.
20	EX_PC	RW	PC where system-level exception occurred.
21	EX_UPC	RW	PC where user-level exception occurred. (GDN_AVAIL is the only user-level exception)

#	Status Reg Name		Purpose
22	FPSR	RW	Floating Point Status Register [5] Unimplemented [4] Invalid [3] Divide by Zero [2] Overflow [1] Underflow [0] Inexact operation These bits are sticky; i.e. floating point operations can set but cannot clear these bits. However, the user can freely change the bits via `MTSR` or `MFSR`. These flags are set the cycle after the floating point instruction finishes execution; i.e., you need three nops inbetween the last floating point operation and a `MFSR` to read the correct value.
23	EVENT_BITS	R	[15:0] the list of events that have triggered
24	EX_BITS	R	Interrupt Status [31] USER - all user interrupts masked if 0 [30] SYSTEM - all interrupts masked if 0 The above can be set/cleared using `inton`, `intoff`, `uinton`, `uintoff`. [6] EVENT_COUNTER [5] GDN_AVAIL [4] TIMER [3] EXTERNAL [2] TRACE [1] GDN_COMPLETE [0] GDN_REFILL For bits 0..6, a “1” indicates a request for a given interrupt occurred but that it has not yet been serviced.
25	EX_MASK	RW	Interrupt Mask [6] EVENT_COUNTER [5] GDN_AVAIL [4] TIMER [3] EXTERNAL [2] TRACE [1] GDN_COMPLETE [0] GDN_REFILL A ”0” indicates that the exception is suppressed.

#	Status Reg Name		Purpose
26	EVENT_CFG	RW	Event Counter Configuration [31:16] Enables for events 16..0 [15:01] PC to profile (omit low two bits) for single mode [00] ← (1 Single Instruction Mode) (0 Global Instruction Mode)
27	POWER_CFG	RW	Power Saving Configuration [00] Disable comparator toggle-suppression [01] Disable ALU toggle-suppression [02] Disable FPU toggle-suppression [03] Disable Multiplier toggle-suppression [04] Disable Divider toggle-suppression [05] Disable Data Cache toggle-suppression [06] Enable Instruction Memory power saving [07] Enable Data Memory power saving [08] Enable Static Router Memory power saving [09] Disable `pwrblk` wake up after TIMER interrupt [10] Disable `pwrblk` wake up after EXTERNAL interrupt [11] Timer wakeup pending on return to `pwrblk` [12] External wakeup pending on return to `pwrblk` At reset, POWER_CFG is set to zero. Bits 11 and 12 are set by the processor if the corresponding interrupt is taken while waiting on a `pwrblk`.
28	TN_CFG	W	Test Network Configuration
29	TN_DONE	W	Signal “DONE” on Test Network with value [31:0]
30	TN_PASS	W	Signal “PASS” on Test Network with value [31:0]
31	TN_FAIL	W	Signal “FAIL” on Test Network with value [31:0]

B.5 Event Counting Support

The event counters provide a facility to monitor, profile, and respond to events on a Raw tile. Each tile has a bank of 16 c_trigger modules. Each c_trigger has a 32-bit counter. These counters count down every time a particular event occurs. The EVENT_CFG2 register is used to determine which events each c_trigger responds to. When the counter transitions from 0 to -1, it will assert a line (the "trigger") which will hold steady until the user writes a new value into the counter. These triggers are visible in the EVENT_BITS register, and are OR'd together to form the EX_BITS EVENT_COUNTER bit, which can cause an interrupt. When the trigger is asserted, the c_trigger module latches the PC (without the low zero bits) of the instruction that caused the event into bits [31:16] of the counter (the `rlm` instruction can be used to extract them efficiently). The c_trigger module will continue to count down regardless of the setting of the trigger. Because the PC is stored in the high bits, there is a window of time in which subsequent events will not corrupt the captured PC. Note that if the event is not instruction related, the setting of the PC in the c_trigger is undefined. The event counters can be both read and written by the user. There is typically a one cycle delay between when an event occurs and when an `mfec` instruction will observe it; there is also a delay of two cycles before an event trigger interrupt will fire.

c_trigger #	EVENT_CFG2	Stage	Function	Notes
0	[25] ← 0	@	Cycle Count	So handler can bound sampling window.
0	[25] ← 1	F	Write Over Read	For poor man's shared memory support. Detects when a resident cache line is marked dirty by a `sw` to an odd address for the first time. Note: If the `sw` is preceded by a `lw` / `sw` / `flush` this mechanism does not have the bandwidth to verify the previous state of the bits. It will conservatively count it as an event.
1		M	Cache Writebacks	Includes flushes.
2		M	Cache Fills	
3		M	Cache Stall Cycles	Total number of cycles that the backend of the pipeline is frozen by the cache state machine. Includes write-back and fill time, as well as time stolen by non-dirty `flush` instructions.
4	[0] ← 0	E	Cache Miss Ops	Number of `flush`, `lw`, `sw` instructions issued.
4	[0] ← 1	E	FPU Ops	Number of FPU instructions issued. Includes `.s` and `.w` instructions.

c_trigger #	EVENT_CFG2	Stage	Function	Notes
5	[1] ← 0	E	Possible Mispredicts	Conditional Jumps and Branches, ERET, DRET, JR, JALR.
5	[1] ← 1	E	Possible Mispredicts	Possible mispredicts due to wrong SBIT (i.e., only conditional jumps and branches)
6	[2] ← 0	E	Actual Mispredicts	Branch mispredictions.
6	[2] ← 1	E	Actual Mispredicts	Mispredictions due to wrong SBIT
7		@	Switch Stalls	On static router (Trigger captures static router PC)
8		@	Possible Mispredicts	On static router (Trigger captures static router PC)
9		@	Actual Mispredicts	On static router (Trigger captures static router PC)
10		@	Pseudo Random LFSR	X_next = (X >> 1) | (xor(X[31,30,10,0]) << 31) Note: Sampling this more than once per 32 cycles produces highly correlated numbers.
11	[3]	R	Functional Unit Stalls	Stalls due to bypassing (e.g., the output of a preceding instruction is not available yet) or because of interlocks on the fp/int dividers.
11	[4]	@	GP	GDN Processor Port Counting
11	[5]	@	MP	MDN Processor Port Counting
11	[23]	@	Instructions Issued	# of instructions that enter Execute stage.
12	[6]	R	Non-cache stalls	# of stalls not due to cache misses. Includes `ilw/isw`; if trigger fires on isw/ilw PC will be the PC of the instruction in the RF stage, rather than the `ilw/isw` instruction.
12	[7]	@	GW	GDN West Port Counting
12	[8]	@	MW	MDN West Port Counting
13	[9]	R	`ilw/isw`	# of `ilw/isw` instructions issued.
13	[10]	@	GS	GDN South Port Counting
13	[11]	@	MS	MDN South Port Counting
13	[24]	@	Instructions Issued	# of instructions that enter Execute stage.
14	[12]	R	`$csto` stalls	Instruction issue blocked on `$csto` full
14	[13]	R	`$cgno` stalls	Instruction issue blocked on `$cgno` full
14	[14]	R	`$cmno` stalls	Instruction issue blocked on `$cmno` full
14	[15]	@	GE	GDN East Port Counting
14	[16]	@	ME	MDN East Port Counting
15	[17]	R	`$csti` stalls	Instruction issue blocked on `$csti` empty
15	[18]	R	`$csti2` stalls	Instruction issue blocked on `$csti2` empty
15	[19]	R	`$cgni` stalls	Instruction issue blocked on `$cgni` empty
15	[20]	R	`$cmni` stalls	Instruction issue blocked on `$cmni` empty
15	[15]	@	GN	GDN North Port Counting
15	[16]	@	MN	MDN North Port Counting

The previous table describes the events that the c_trigger modules can be configured to count. The EVENT_CFG2 column specifies the bit number of EVENT_CFG2 that must be set in order to enable counting of that event.

The low bits of EVENT_CFG allow the user to count events that occurs on a particular instruction at a particular PC instead of across all PCs. For this "single instruction mode", EVENT_CFG[0] is set to 1, and the PC to sample is placed into EVENT_CFG[15:1]. In cases where the event does not have an associated main processor PC (marked with the "@" in the table), the EVENT_CFG single instruction mode setting is ignored. The high bits of EVENT_CFG selectively enable counting on a per event basis, but do not suppress existing triggers.

The EVENT_CFG2 SPR allows the user to configure the events that a particular c_trigger module counts. In some cases multiple enabled events may be connected to the same trigger. In that case, the counters increments each cycle if any such enabled events has occurred. In some cases, there are nonsensical combinations that can be enabled (say GE and `$csto` stalls).

The meaning of the GN, GE, GS, GW, GP, MN, ME, MS, MW, and MP events are configured by the EC_DYN_CFG status/control register. Each event corresponds to a network N (G = general, M = memory) and a direction D (N=north, E=east, ...). The encodings are shown in the table in Section B.4.

B.6 Exception Vectors

#	Name	Offset	Purpose
0	VEC_GDN_REFILL	0x00	Dynamic Refill Exception
1	VEC_GDN_COMPLETE	0x10	GDN Send Is Complete
2	VEC_TRACE	0x20	Trace Interrupt
3	VEC_EXTERN	0x30	External Interrupt (MDN)
4	VEC_TIMER	0x40	Timer Exception
5	VEC_GDN_AVAIL	0x50	Data Avail on GDN
6	VEC_EVENT_COUNTERS	0x60	Event Counter Interrupt

In the Raw architecture, the exception vectors are stored starting at offset zero in instruction memory. In RawH, the exception vectors are stored relative to `SR[EX_BASE_ADDR]`. When an exception occurs, the processor starts fetching from the corresponding exception location. Thus, a TIMER exception would start fetching at address `SR[EX_BASE_ADDR] + 0x40`.

Each exception has 4 contiguous instructions; this is enough to do a small amount of work; such as save a register, load a jump address, and branch there:

```
sw $3, interrupt_save($gp)
lw $3, gdn_vec($gp)
jr $3
```

B.7 Switch Processor Instruction Set

The switch processor ("the switch"), pictured in Figure B-1, controls the static router that serves as Raw's inter-tile SON. Architecturally, it looks like a VLIW processor which can execute a large 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. See Sections A.1 and A.2.2 for more details.

Programmatically, the static router has three structural components:

1. A sequencer (part of the switch processor) which executes a very basic instruction set.
2. A one read port, one write port (1R-1W) 4-element register file.
3. A pair of crossbars, which is responsible for routing values to neighboring switches. The inputs of the crossbars are network input blocks (NIBs), while the outputs are wires, typically to neighbor tiles.

A switch instruction consists of a small processor instruction and a list of routes for the two crossbars. 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 a 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.
5. Routes between different domains (i.e., cN, cS, cE, cW, cst_1 and cN_2, cS_2, cE_2, cW_2, cst_2) are forbidden. To switch domains, one should route the word through c_{12} or c_{21} (or the register file). It will be available in c_{12} or c_{21} on the next cycle.

B.7.1 Switch Processor Instruction Set Examples

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 (2) and write one register (3).

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

is illegal because the ROUTE instruction is trying to use register 2 as a destination.

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

is illegal because two different reads are being initiated to the register file (2,3).

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

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

B.7.2 Switch Instruction Format

bit	name	Function
63	pr	If 1, predict instruction is a taken branch/jump.
62..59	Op	The operation, see "Switch High-Level Opcode Map".
58	R	Relative bit. R=1 for branches.
57	0	Reserved; allows expansion of switch memory.
56..44	imm	Immediate field, used for branches and jumps.
43..42	rdst	Register number to write to.
41..40	rsrc	Register number to read from.
39	w	"Which" bit. Whether rego is from 1st or 2nd xbar.
38..36	rego	Which port to route into the switch processor.
35..33	cNo	Which port to route out to cNo.
32..30	cEo	Which port to route out to cEo.
29..27	cSo	Which port to route out to cSo.
26..24	cWo	Which port to route out to cWo.
23..21	csti	Which port to route out to csti.
20..18	c12	Which port to route out to c_{12}.
17..15	cNo2	Which port to route out to cNo_2.
14..12	cEo2	Which port to route out to cEo_2.
11..9	cSo2	Which port to route out to cSo_2.
8..6	cWo2	Which port to route out to cWo_2.
5..3	csti2	Which port to route out to $csti_2$.
2..0	c21	Which port to route out to c_{21}.

B.7.2.1 Switch High-Level Opcode Map

bits	bits 60..58							
62..61	000 (R=0)	010 (R=0)	100 (R=0)	110 (R=0)	001 (R=1)	011 (R=1)	101 (R=1)	111 (R=1)
00	ExOp	JLTZ	JNEZ	JGEZ		BLTZ	BNEZ	BGEZ
01	JAL	J	JGTZ	JEQZ	BAL	B	BGTZ	BEQZ
10								
11	JLEZ	JNEZD	JEQZD	DEBUG	BLEZ	BNEZD	BEQZD	

B.7.3 Switch Instruction Format (Op == ExOp)

bit	name	Function
63	0	Always 0.
62..59	0	Always 0 (indicates ExOp).
58..57	0	Always 0.
56..48	9'b0	Expansion space.
47..44	ExOp	The operation, see "Switch ExOp Map".
43..42	rdst	Register number to write to.
41..40	rsrc	Register number to read from.
39	w	"Which" bit. Whether rego is from 1st or 2nd xbar
38..36	rego	Which port to route into the switch processor.
35..33	cNo	Which port to route out to cNo.
32..30	cEo	Which port to route out to cEo.
29..27	cSo	Which port to route out to cSo.
26..24	cWo	Which port to route out to cWo.
23..21	csti	Which port to route out to csti.
20..18	c12	Which port to route out to c_{12}.
17..15	cNo2	Which port to route out to cNo_2.
14..12	cEo2	Which port to route out to cEo_2.
11..9	cSo2	Which port to route out to cSo_2.
8..6	cWo2	Which port to route out to cWo_2.
5..3	csti2	Which port to route out to $csti_2$.
2..0	c21	Which port to route out to c_{21}.

B.7.3.1 Switch ExOp Map

bits	bits 45..44			
47..46	00	01	10	11
00	NOP	MOVE	JR	JALR
01				
10				
11				

B.7.4 Switch Port Name Map (Primary Crossbar)

	000	001	010	011	100	101	110	111
Port	none	csto	cWi	cSi	cEi	cNi	swi1	regi

B.7.5 Switch Port Name Map (Secondary Crossbar)

	000	001	010	011	100	101	110	111
Port	none	csto	cWi2	cSi2	cEi2	cNi2	swi2	regi

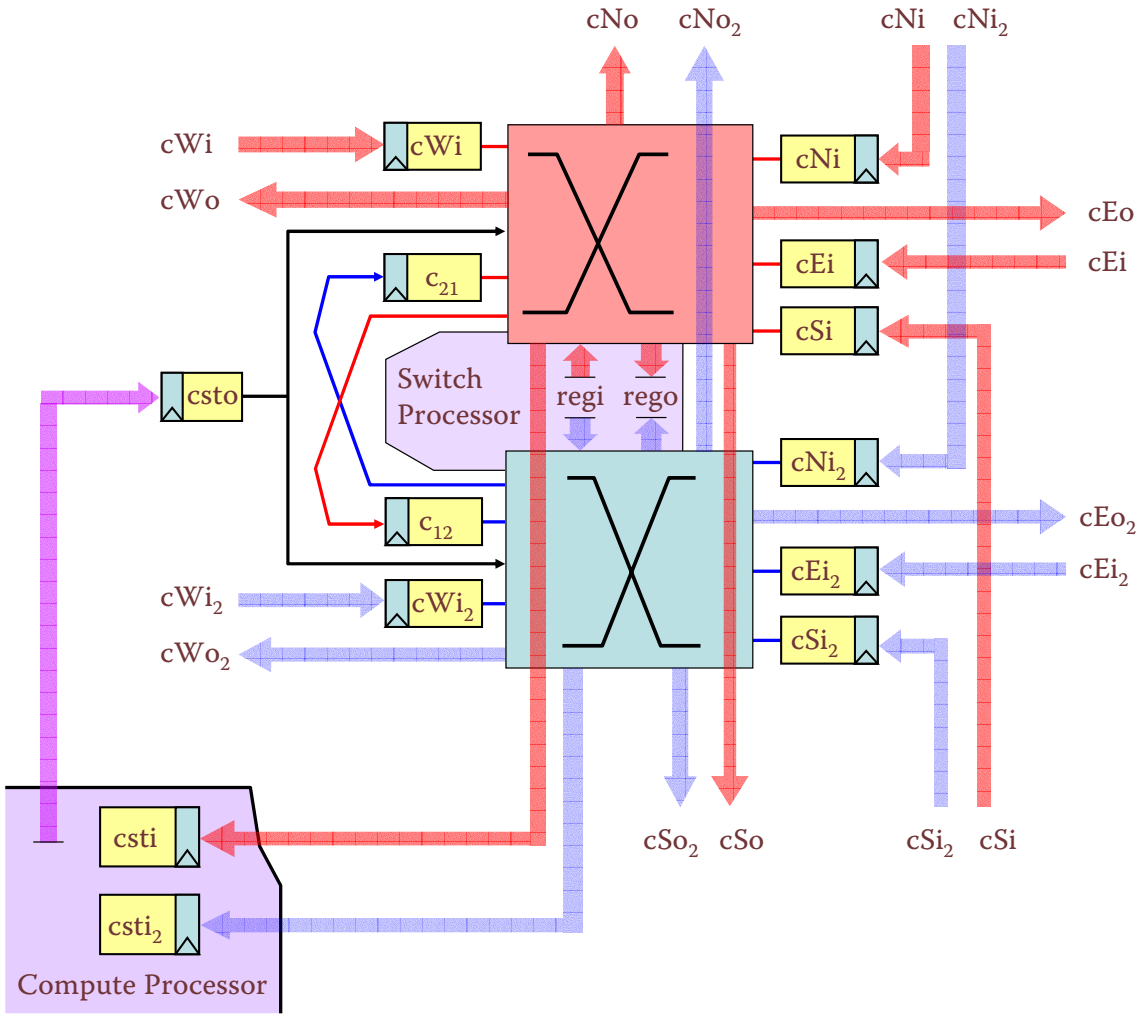


Figure B-1: Basic architecture of a static router. The static router is comprised of the switch processor, two crossbars, and a set of NIBs and inter-tile network links. Most NIBs are connected to exactly one crossbar. However, the csto NIB is accessible by both crossbars, and the c_{12} and c_{21} NIBs are used to route values between crossbars. The rego and regi ports, shared by both crossbars, are not NIBs, but are direct connections into the switch processor.

Bibliography

[1] *The Transputer Databook (2nd Edition).* Inmos Limited, 1989.

[2] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *International Symposium on Computer Architecture*, pages 248–259, June 2000.

[3] S. Agrawal, W. Thies, and S. Amarasinghe. Optimizing Stream Programs Using Linear State Space Analysis. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 126–136, September 2005.

[4] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A Portable Linear Algebra Library for High-Performance Computers. In *International Conference on Supercomputing*, pages 2–11, November 1990.

[5] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilicioglu, and J. A. Webb. The Warp Computer: Architecture, Implementation and Performance. *IEEE Transactions on Computers*, 36(12):1523–1538, December 1987.

[6] Arvind and S. Brobst. The Evolution of Dataflow Architectures from Static Dataflow to P-RISC. *International Journal of High Speed Computing*, pages 125–153, June 1993.

[7] R. Barua. *Maps: A Compiler-Managed Memory System for Software-Exposed Architectures.* PhD thesis, Massachusetts Institute of Technology, January 2000.

[8] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: A Compiler-Managed Memory System for Raw Machines. In *International Symposium on Computer Architecture*, pages 4–15, May 1999.

[9] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Compiler Support for Scalable and Efficient Memory Systems. In *IEEE Transactions on Computers*, pages 1234–1247, November 2001.

[10] T. Bednar, R. Piro, D. Stout, L. Wissel, and P. Zuchowski. Technology-migratable ASIC library design. In *IBM Journal of Research and Development*, pages 377–386, July 1996.

[11] B. Bentley. Validating the Intel Pentium 4 Microprocessor. In *Design Automation Conference*, pages 244–248, June 2001.

[12] B. Bentley. Keynote: Validating a Modern Microprocessor. In *International Conference on Computer Aided Verification*, June 2005.

[13] M. Bohr. Interconnect Scaling - The Real Limiter to High Performance ULSI. In *International Electron Device Meeting*, pages 241–244, December 1995.

[14] P. Buffet, J. Natonio, R. Proctor, Y. Sun, and G. Yasar. Methodology for I/O Cell Placement and Checking in ASIC Designs Using Area-Array Power Grid. In *Q3 2000 IBM MicroNews*, pages 7–11, Q3 2000.

[15] D. Carmean. The Intel Pentium 4 Processor. In *University of California at San Diego CSE Talk*, Spring 2002.

[16] J. Chen, M. Gordon, W. Thies, M. Zwicker, K. Pulli, and F. Durand. A Reconfigurable Architecture for Load-Balanced Rendering. In *Proceedings of SIGGRAPH/Eurographics Graphics Hardware*, pages 71–80, July 2005.

[17] D. Chinnery and K. Keutzer. *Closing the Gap Between ASIC & Custom*. Kluwer Academic Publishers, 2002.

[18] K. E. Coons, X. Chen, D. Burger, K. S. McKinley, and S. K. Kushwaha. A spatial path scheduling algorithm for EDGE architectures. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 129–140, New York, NY, USA, 2006. ACM Press.

[19] H. Corporaal. Move: A framework for high-performance processor design. In *Supercomputing '91*, November 1991.

[20] H. Corporaal. *Transport Triggered Architectures, Design and Evaluation*. PhD thesis, Delft University of Technology, 1995.

[21] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: towards a realistic model of parallel computation. In *Principles and Practice of Parallel Programming*, pages 1–12, May 1993.

[22] W. J. Dally. *A VLSI Architecture for Concurrent Data Structures*. Kluwer Academic Publishers, 1987.

[23] W. J. Dally and B. Towles. *Principles and Practice of Interconnection Networks*. Elsevier, Inc., 2004.

[24] B. Davari, R. Dennard, and G. Shahidi. CMOS Scaling for High Performance and Low Power – The Next Ten Years. In *Proceedings of the IEEE*, pages 595–606, April 1995.

[25] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. In *Journal of Solid-State Circuits*, pages 256–268, October 1974.

[26] J. Dennis and D. Misunas. A prelinary architecture for a basic data-flow processor. In *International Symposium on Computer Architecture*, pages 125–131, 1975.

[27] K. Diefendorff. Intel Raises the Ante With P858. *Microprocessor Report*, pages 22–25, January 1999.

[28] J. Duato. A Necessary and Sufficient Condition for Deadlock- Free Adaptive Routing in Wormhole Networks. In *IEEE Transactions on Parallel and Distributed Systems*, October 1995.

[29] J. Duato and T. M. Pinkston. A General Theory for Deadlock-Free Adaptive Routing using a Mixed Set of Resources. *IEEE Transactions on Parallel and Distributed Systems*, 12(12):1–16, December 2001.

[30] A. Duller, G. Panesar, and D. Towner. Parallel processing – the picochip way! In *Communicating Process Architectures*. IOS Press, 2003.

[31] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: A Vector Extension to the Alpha Architecture. In *International Symposium on Computer Architecture*, pages 281–292, May 2002.

[32] F. Faggin, J. Hoff, M.E., S. Mazor, and M. Shima. The history of the 4004. In *Proceedings of the IEEE*, pages 10–20, December 1996.

[33] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing Cycle Time through Partitioning. In *International Symposium on Microarchitecture (MICRO)*, pages 149–159, Washington, DC, USA, December 1997. IEEE Computer Society.

[34] I. Fried. Apple's benchmarks put to the test. In *CNET News.com http://news.com.com/Apples+benchmarks+put+to+the+test/2100-1042_3-1020631.html*, June 24, 2003.

[35] M. Gordon. A Stream-Aware Compiler for Communication-Exposed Architectures. Master's thesis, MIT Laboratory for Computer Science, August 2002.

[36] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[37] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffman, D. Z. Maze, and S. Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October, 2002.

[38] T. Gross and D. R. O'Halloron. *iWarp, Anatomy of a Parallel Computing System*. The MIT Press, Cambridge, MA, 1998.

[39] S. Gunther, F. Binns, D. Carmean, and J. Hall. Managing the Impact of Increasing Microprocessor Power Consumption. In *Intel Technology Journal*, Q1 2001.

[40] L. Gwennap. Comparing RISC microprocessors. In *Proceedings of the Microprocessor Forum*, October 1994.

[41] L. Gwennap. Coppermine Outruns Athlon. *Microprocessor Report*, page 1, October 1999.

[42] J. R. Hauser and J. Wawrzynek. Garp: A MIPS Processor with Reconfigurable Coprocessor. In *1997 FCCM*, pages 12–21, 1997.

[43] J. Heinrich. *MIPS R4000 Microprocessor User's Manual*. MIPS Technologies, second edition, 1994.

[44] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. In *Intel Technology Journal*, Q1 2001.

[45] R. Ho. *On-Chip Wires: Scaling and Efficiency*. PhD thesis, Stanford, August 2003.

[46] R. Ho, K. W. Mai, and M. A. Horowitz. The Future of Wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001.

[47] H. Hoffmann, V. Strumpen, A. Agarwal, and H. Hoffmann. Stream Algorithms and Architecture. Technical Memo MIT-LCS-TM-636, LCS, MIT, 2003.

[48] R. A. Iannucci. Toward a dataflow / von neumann hybrid architecture. In *International Symposium on Computer Architecture*, pages 131–140, 1988.

[49] Intel. Pentium Processor Family Developer's Manual. 1997.

[50] Intel. Intel Pentium 4 Processor Optimization Reference Manual. In *Intel Order Number 24896*, 2001.

[51] Intel. Intel Microprocessor Quick Reference Guide . In *http://www.intel.com/pressroom/kits/quickreffam.htm* , December 15, 2005.

[52] Intel. Intel Pentium III Processor Specification Update. In *Intel Document 244453-055*, May 2005.

[53] Intel. Intel Pentium 4 Processor Specification Update. In *Intel Document 249199-061*, October 2005.

[54] Intel Corporation. *MCS-4 Micro Computer Set Users Manual*. Revision 4 edition, February 1973.

[55] Intel Corporation. *MCS-4 Micro Computer Set Spec Sheet*. November 1971.

[56] J. Janssen and H. Corporaal. Partitioned Register File for TTAs. In *1996 MICRO*, pages 303–312, 1996.

[57] U. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany. The Imagine Stream Processor. In *2002 ICCD*, pages 282–288, 2002.

[58] M. Karczmarek, W. Thies, and S. Amarasinghe. Phased scheduling of stream programs. In *Languages, Compilers, and Tools for Embedded Systems*, San Diego, CA, June 2003.

[59] R. E. Kessler. The alpha 21264 microprocessor. In *IEEE Micro*, pages 24–36, Los Alamitos, CA, USA, March 1999. IEEE Computer Society Press.

[60] H.-S. Kim and J. E. Smith. An Instruction Set Architecture and Microarchitecture for Instruction Level Distributed Processing. In *International Symposium on Computer Architecture*, pages 71–81, 2002.

[61] H.-S. Kim and J. E. Smith. An ISA and Microarchitecture for Instruction Level Distributed Processing. In *International Symposium on Computer Architecture*, pages 71–81, May 2002.

[62] J. S. Kim, M. B. Taylor, J. Miller, and D. Wentzlaff. Energy Characterization of a Tiled Architecture Processor with On-Chip Networks. In *International Symposium on Low Power Electronics and Design*, August 2003.

[63] A. KleinOsowski and D. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *Computer Architecture Letters*, June 2002.

[64] C. Kozyrakis and D. Patterson. Overcoming the limitations of conventional vector processors. In *International Symposium on Computer Architecture*, June 2003.

[65] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Trewhaft, and K. Yelick. Scalable Processors in the Billion-Transistor Era: IRAM. In *IEEE Computer*, pages 75–78, September 1997.

[66] R. Krashinsky, C. Batten, S. Gerding, M. Hampton, B. Pharris, J. Casper, and K. Asanovic. The Vector-Thread Architecture. In *International Symposium on Computer Architecture*, June 2004.

[67] J. Kubiatowicz. *Integrated Shared-Memory and Message-Passing Communication in the Alewife Multiprocessor*. PhD thesis, MIT, 1998.

[68] J. Kubiatowicz and A. Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *International Supercomputing Conference*, pages 195–206, 1993.

[69] N. Kushman. Performance Nonmonotonicities: A Case Study of the UltraSPARC Processor. Master's thesis, Massachusetts Institute of Technology, June 1998.

[70] A. A. Lamb, W. Thies, and S. Amarasinghe. Linear Analysis and Optimization of Stream Programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 12–25, San Diego, CA, June 2003.

[71] W. Lee. *Software Orchestration of Instruction Level Parallelism on Tiled Processor Architectures*. PhD thesis, Massachusetts Institute of Technology, May 2005.

[72] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–54, October 1998.

[73] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, D. Hillis, B. C. Kuszmaul, M. A. S. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak. The network architecture of the connection machine cm-5. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, New York, NY, USA, 1992. ACM Press.

[74] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[75] C. Lutz, S. Rabin, C. Seitz, and D. Speck. Design of the mosaic element. In *CalTech Computer Science Tech Report 5093:TR:83*, 1983.

[76] K. Mackenzie, J. Kubiatowicz, M. Frank, W. Lee, V. Lee, A. Agarwal, and M. F. Kaashoek. Exploiting Two-Case Delivery for Fast Protected Messaging. In *Symposium on High-Performance Computer Architecture*, February 1998.

[77] R. Mahnkopf, K.-H. Allers, M. Armacost, A. Augustin, J. Barth, G. Brase, R. Busch, E. Demm, G. Dietz, B. Flietner, G. Friese, F. Grellner, K. Han, R. Hannon, H. Ho, M. Hoinkis, K. Holloway, T. Hook, S. Iyer, P. Kim, G. Knoblinger, B. Lemaitre, C. Lin, R. Mih, W. Neumueller, J. Pape, O. Prigge, N. Robson, N. Rovedo, T. Schafbauer, T. Schiml, K. Schruefer, S. Srinivasan, M. Stetter, F. Towler, P. Wensley, C. Wann, R. Wong, R. Zoeller, and B. Chen. 'System on a Chip' Technology Platform for 0.18 micron Digital, Mixed Signal and eDRAM Applications. In *International Electron Devices Meeting*, pages 849–852, December 1999.

[78] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *International Symposium on Computer Architecture*, pages 161–171, 2000.

[79] D. Matzke. Will Physical Scalability Sabotage Performance Gains? *IEEE Computer*, 30(9):37–39, September 1997.

[80] J. McCalpin. STREAM: Sustainable Memory Bandwidth in High Perf. Computers. http://www.cs.virginia.edu/stream.

[81] C. McNairy and D. Soltis. Itanium 2 Processor Microarchitecture. In *IEEE Micro*, pages 44–55, March 2003.

[82] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. Eggers. Instruction Scheduling for Tiled Dataflow Architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[83] G. Moore. Cramming more components onto integrated circuits. In *Electronics Magazine*, April 19, 1965.

[84] C. A. Moritz, D. Yeung, and A. Agarwal. SimpleFit: A Framework for Analyzing Design Tradeoffs in Raw Architectures. *IEEE Transactions on Parallel and Distributed Systems*, pages 730–742, July 2001.

[85] S. Naffziger, G. Colon-Bonet, T. Fischer, R. Riedlinger, T. J. Sullivan, and T. Grutkowski. The Implementation of the Itanium 2 Microprocessor. In *IEEE Journal of Solid-State Circuits*, November 2002.

[86] R. Nagarajan, S. Kushwa, D. Burger, K. McKinley, C. Lin, and S. Keckler. Static Placement, Dynamic Issue (SPDI) Scheduling for EDGE Architectures. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2004.

[87] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A Design Space Evaluation of Grid Processor Architectures. In *2001 MICRO*, pages 40–51, 2001.

[88] M. Noakes, D.A.Wallach, and W. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *International Symposium on Computer Architecture*, pages 224–235, San Diego, CA, May 1993. ACM.

[89] J. Oliver, R. Rao, P. Sultana, J. Crandall, E. Czernikowski, L. W. J. IV, D. Franklin, V. Akella, and F. T. Chong. Synchroscalar: A multiple clock domain, power-aware, tile-based embedded processor. In *Proceedings of the International Symposium on Computer Architecture*. IEEE Computer Society, June 2004.

[90] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, New York, NY, USA, 1996. ACM Press.

[91] S. Palacharla. *Complexity-Effective Superscalar Processors*. PhD thesis, University of Wisconsin–Madison, 1998.

[92] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-Effective Superscalar Processors. In *International Symposium on Computer Architecture*, pages 206–218, 1997.

[93] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. Mattson, and J. D. Owens. A Bandwidth-Efficient Architecture for Media Processing. In *Proceedings of the International Symposium on Microarchitecture*, pages 3–13, December 1998.

[94] N. Rovedo, T. Hook, M. Armacost, G. Hueckel, D. Kemerer, and J. Lukaitis. Introducing IBM's First Copper Wiring Foundry Technology: Design, Development, and Qualification of CMOS 7SF, a .18 micron Dual-Oxide Technology for SRAM, ASICs, and Embedded DRAM. In *Q4 2000 IBM MicroNews*, pages 34–38, Q4 2000.

[95] Sankaralingam, Singh, Keckler, and Burger. Routed Inter-ALU Networks for ILP Scalability and Performance. In *International Conference on Computer Design*, 2003.

[96] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *International Symposium on Computer Architecture*, pages 422–433, June 2003.

[97] T. Schubert. High Level Formal Verification of Next-Generation Microprocessors. In *Design Automation Conference*, 2003.

[98] S. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Symposium on Architectural Support for Programming Languages and Operating Systems*, 1996.

[99] D. Shoemaker, F. Honore, C. Metcalf, and S. Ward. NuMesh: An Architecture Optimized for Scheduled Communication. *Journal of Supercomputing*, 10(3):285–302, 1996.

[100] A. Smith, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, K. S. McKinle, and J. Burrill. Compiling for EDGE Architectures. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 185–195, Washington, DC, USA, 2006. IEEE Computer Society.

[101] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *International Symposium on Computer Architecture*, pages 414–425, June 1995.

[102] Y. H. Song and T. M. Pinkston. A Progressive Approach to Handling Message Dependent Deadlocks in Parallel Computer Systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):259–275, March 2003.

[103] J. Suh, E.-G. Kim, S. P. Crago, L. Srinivasan, and M. C. French. A Performance Analysis of PIM, Stream Processing, and Tiled Processing on Memory-Intensive Signal Processing Kernels. In *International Symposium on Computer Architecture*, pages 410–419, June 2003.

[104] S. Swanson. *The WaveScalar Architecture*. PhD thesis, University of Washington, 2006.

[105] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *International Symposium on Microarchitecture*, December 2003.

[106] M. Taylor. Design Decisions in the Implementation of a Raw Architecture Workstation. Master's thesis, Massachusetts Institute of Technology, September 1999.

[107] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw Microprocessor: A Computation Fabric for Software Circuits and General-Purpose Programs. In *IEEE Micro*, March-April 2002.

[108] M. Taylor, W. Lee, M. Frank, S. Amarasinghe, and A. Agarwal. How to Build Scalable On-Chip ILP Networks for a Decentralized Architecture. Technical Report 628, MIT Laboratory for Computer Science (now CSAIL), April 2000.

[109] M. B. Taylor. Deionizer: A Tool For Capturing And Embedding I/O Calls. Technical Memo 644, CSAIL/Laboratory for Computer Science, MIT, 2004. http://cag.csail.mit.edu/∼mtaylor/deionizer.html.

[110] M. B. Taylor. The Raw Processor Specification, Continuously Updated 2005. ftp://ftp.cag.lcs.mit.edu/pub/raw/documents/RawSpec99.pdf.

[111] M. B. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal. Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures. In *International Symposium on High Performance Computer Architecture*, pages 341–353, February 2003.

[112] M. B. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal. Scalar Operand Networks. In *IEEE Transactions on Parallel and Distributed Systems*, pages 145–162, February 2005.

[113] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *Proceedings of the International Symposium on Computer Architecture*, pages 2–13, June 2004.

[114] J. M. Tendler, J. Dodson, J. J.S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. In *IBM Journal of Research and Development*, January 2002.

[115] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *International Conference on Compiler Construction*, pages 179–196, 2002.

[116] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe. Teleport Messaging for Distributed Stream Programs. In *Symposium on Principles and Practice of Parallel Programming*, pages 224–235, Chicago, Illinois, June 2005.

[117] R. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In *IBM Journal of Research and Development*, 1967.

[118] M. Tremblay, D. Greenley, and K. Normoyle. The Design of the Microarchitecture of UltraSPARC-1. In *Proceedings of the IEEE*, December 1995.

[119] J. H.-C. Tseng. *Banked Microarchitectures for Complexity-Effective Superscalar Microprocessors*. PhD thesis, Massachusetts Institute of Technology, May 2006.

[120] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *International Symposium on Computer Architecture*, May 1992.

[121] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, pages 86–93, September 1997.

[122] D. Wentzlaff. Architectural Implications of Bit-level Computation in Communication Applications. Master's thesis, Massachusetts Institute of Technology, 2002.

[123] D. Wentzlaff and A. Agarwal. A Quantitative Comparison of Reconfigurable, Tiled, and Conventional Architectures on Bit-level Computation. In *Symposium on Field-Programmable Custom Computing Machines*, pages 289–290, April 2004.

[124] R. Whaley, A. Petitet, J. J. Dongarra, and Whaley. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–35, 2001.

[125] S. Yang, S. Ahmed, B. Arcot, R. Arghavani, P. Bai, S. Chambers, P. Charvat, R. Cotner, R. Gasser, T. Ghani, M. Hussein, C. Jan, C. Kardas, J. Maiz, P. McGregor, B. McIntyre, P. Nguyen, P. Packan, I. Post, S. Sivakumar, J. Steigerwald, M. Taylor, B. Tufts, S. Tyagi, and M. Bohr. A High Performance 180 nm Generation Logic Technology. In *International Electron Devices Meeting*, pages 197–200, 1998.