A Superscalar Out-of-Order x86 Soft Processor for FPGA

by

Henry Ting-Hei Wong

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

A Superscalar Out-of-Order x86 Soft Processor for FPGA

Henry Ting-Hei Wong
Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto
2017

Although FPGAs continue to grow in capacity, FPGA-based soft processors have grown very little because of the difficulty of achieving higher performance in exchange for area. Superscalar out-of-order processor microarchitectures have been used successfully for hard processors for many years, and promise large performance gains if they can also be used for FPGA soft processors. Out-of-order soft processor microarchitectures have so far been avoided due to the area increase and the expectation that a loss in clock frequency would more than offset the instructions-per-cycle (IPC) gains.

This thesis presents the design of the microarchitecture and circuits of a two-issue (superscalar) out-of-order x86 FPGA soft processor.

Our microarchitecture achieves 2.7 times the per-clock performance of a performance-tuned Nios II/f, Altera's fastest (RISC-like, single-issue, pipelined) soft processor, and 0.8 times the frequency, for a total performance improvement of 2.2 times. The processor is projected to use around 28 700 Stratix IV Adaptive Logic Modules (ALMs), which is 6.5 times the area of the Nios II/f, but still only a small fraction of a modern FPGA.

In addition to performance improvements, our microarchitecture design is sufficiently complete and correct to boot most 32-bit x86 operating systems unmodified.

We also design circuits for most of the components in the processor. These highly-optimized circuits are key to achieving high operating frequency despite the increased complexity of out-of-order execution.

Through the design of our processor, we demonstrate that a high-performance processor microarchitecture can be implemented successfully on FPGAs. Beyond the proposed microarchitecture, the processor circuits presented in this thesis will enable new out-of-order soft processor microarchitectures of varying performance, cost, and instruction set, created from variations of our circuits.

Acknowledgements

I am grateful for my supervisors, Jonathan Rose and Vaughn Betz, for giving me the freedom to pursue my own ideas, while ensuring this project remained finite in size and produced useful results.

I also thank my parents for the support over the many years of growing up, and for encouraging my curiosity at a young age.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A Field-Programmable Gate Array (FPGA) is a programmable integrated circuit that can be configured to implement nearly any digital circuit. Processors have been implemented on FPGAs soon after FPGAs grew large enough to contain them. Having a soft processor in an FPGA system allows parts of the system to be implemented using easier-to-write software, reducing the amount of hardware that must be designed.

Since that time, FPGA logic capacity has continued to grow exponentially along with improved circuit fabrication technology. However, soft processor microarchitectures have progressed little, occupying an ever-decreasing fraction of modern FPGAs. Soft processors typically still use a single-issue in-order microarchitecture, while hard processors have continued to progress to increasingly aggressive superscalar and out-of-order designs.

Although modern FPGAs now often contain hard processors, soft processors continue to be useful. Like other hard vs. soft trade-offs, a hard processor is smaller and faster than an equivalent soft processor if the hardened processor is precisely what your system wants. Soft processors can be useful in cases such as needing more processors, needing processors with a different instruction set, using soft processors to ease migration between FPGA families (where the hard processor system may change), or needing the ability to customize the processor.

The ability to trade logic resource usage (area) for performance in a soft processor beyond current performance levels would make soft processors more useful, by allowing designers to implement more functionality in an easier-to-use software environment. The ever-increasing logic capacity of FPGAs has made area much cheaper, and new soft processor microarchitectures are needed to convert logic resources into greater performance.

This work sets out to design a higher-performance soft processor microarchitecture, using a fairly conventional superscalar out-of-order microarchitecture. In the interest of ease-of-use, we chose to use the popular (32-bit) x86 instruction set [1, 2], allowing existing software, operating systems, software development tools, and programmer expertise to be reused. This is in direct contrast to some prior work where even the programming model and instruction set were allowed to be redesigned to suit the FPGA substrate [3]. We aim to run existing operating systems without modification.

There are two major challenges to producing a “good” processor design within these constraints:

First, it is challenging to design a microarchitecture that is both high performance and correctly implements a complex instruction set architecture, while being aware of the strengths and limitations of the underlying FPGA substrate. Correctness constraints come not only from the instruction set manual, but also from the *de facto* specification implied by being binary compatible with several decades of software. While existing hard x86 processor designs do correctly implement the instruction set, the details of the mechanisms used are largely trade secrets.

Second, the microarchitecture needs to be mapped to carefully-designed circuits. Insufficient attention to circuit design can easily negate any microarchitecture-level performance improvements that are achieved. The FPGA substrate performance characteristics differ from hard processors and influence circuit design. Also, FPGA lookup tables (LUTs) are relatively coarse-grained, so carefully packing circuits into LUTs produce significant circuit speed improvements.

These challenges have sometimes led to out-of-order microarchitectures being considered as unsuitable or inefficient for FPGA soft processors, leading to many soft processor designs that abandon the easy-to-use single-threaded programming model to gain performance via data-level or thread-level parallelism. Instead, we aim to make superscalar out-of-order processors practical on FPGAs. We do this by careful design of both the microarchitecture and circuits: We design a microarchitecture that is aware of the characteristics of circuit designs while spending considerable effort designing faster circuits that enable a more aggressive microarchitecture. We believe that a well-designed out-of-order soft processor can give significant performance gains compared to existing in-order soft processors, while still using a modest amount of FPGA resources.

1.1 Why a Faster Single-Threaded Soft Processor?

While FPGA soft processors are never more area, power, or performance efficient than custom FPGA hardware (or ASICs), they remain useful due to their ease of use: Writing software (especially single-threaded) requires far less effort than hardware design. In systems with FPGA soft processors, the soft processor could be part of a larger system where most of the (usually parallelizable) computation is offloaded to other hardware on the FPGA, or used in some applications as the main method of computation (e.g., to run an operating system, a network software stack, coordinate I/O devices, and control a user interface, such as in an entertainment system). In both cases, as increasing system performance is required, the performance demand on the sequential soft processor also increases, unless greater effort is expended to parallelize the software or offload more of the computation onto hardware.

Faster single-threaded processors are useful because they increase performance without needing to rewrite software, as would be required for processors that gain performance by extracting data-level parallelism (e.g., vector or SIMD) or thread-level parallelism. Also, relying on data- or thread-level parallelism requires the software task to be parallelizable, and most real-world tasks are not perfectly parallelizable. Thus, simply replicating simple soft processors (or using specialized data-parallel processors) will not always be a sufficient solution. Higher sequential performance is still needed so the non-parallelizable portions of the computation do not end up becoming the bottleneck [4]. Indeed, even the VectorBlox vector soft processor, which provides high speedups for data-parallel workloads, still relies on a scalar Nios II/f processor to perform the non-vector operations [5]. The Nios II/f would likely become a bottleneck for workloads that consist of more than just vectorizable kernels. Thus, assuming one wants to avoid the development effort of custom hardware, increasing (single-threaded) soft processor performance is still useful.

As hard processors are more efficient and perform better than soft processors of a similar microarchitecture, one might ask whether hard processors should always be used whenever higher single-threaded performance is required. Hard processor do not suit every need. The traditional advantages of soft processors still apply: soft processors give the flexibility of choosing how many and what type of processor to use, the ability to more tightly couple the processor to other hardware on the FPGA, and allow compatibility across FPGA families, generations, and vendors.

1.2 Why Out-of-Order?

Out-of-order processors contain hardware that computes the data dependencies between instructions in a program, and are able to schedule instructions to execute whenever an instruction's operands are ready, while still computing the same result as though the program's instructions executed in their original order. Out-of-order execution improves processor performance by finding instruction-level parallelism to execute independent instructions in parallel and to tolerate instructions with higher latency, such as memory operations. The aggressiveness of the design (e.g., how far ahead in the instruction stream to search for independent instructions) allows trading increased logic area for increased processor performance.

As current commercial soft processors already achieve (for FPGAs) high clock frequencies, future sequential performance gains must come from instructions-per-cycle (IPC) increases. There is compelling data from the evolution of hard processor microarchitectures that there are significant performance gains to be had when moving from in-order to out-of-order processors, which we show in Table 1.1. Each row in Table 1.1 is a specific microarchitecture/vendor, and illustrates the net performance benefit of the transition from in-order to out-of-order. As these processors are built in different process technologies and the processors run at different frequencies, the performance increases due to frequency increases are factored out. Hence, the table shows the IPC improvement due to the transition, and this is significant in every case — ranging from 1.6 to 2 times on SPECint (95 or 2006).

1.3 Why x86?

Although the choice of instruction set has little impact on performance and power for all but very low-end processors [8], the choice is hugely important in determining ease of use. Using a popular instruction set is important as it allows reusing existing software, operating systems (OS), tools, and expertise.

There are few instruction sets that are widely used. Currently, the x86 instruction set dominates on desktops and laptops, and are used in some mobile devices (mainly tablets), while variants of ARM (v7

Vendor	SPECint Version	In-order		Out-of-Order		IPC Ratio
		Processor	Score	Processor	Score	
MIPS [6]	95	R5000 180 MHz	4.8	R10000 195 MHz	11.0	2.1
Alpha [6]	95	21164 500 MHz	15.0	21264 500 MHz	27.7	1.9
Intel [6]	95	Pentium 200 MHz	5.5	Pentium Pro 200 MHz	8.7	1.6
Intel [7]	2006	Atom S1260 2 GHz	7.4	Atom C2730 2.6 GHz	15.7	1.6

Table 1.1: Comparison of SPECint scores between in-order and out-of-order processors and frequency-normalized ratio

and v8) dominate on mobile devices (tablets and smartphones) and are also used in many embedded devices. x86 systems are unique in being largely binary compatible even for operating systems, due to the x86 PC being a *de facto* standard that defines not only the CPU instruction set, but a basic set of hardware devices and the interface (BIOS or UEFI) between firmware and the OS.

The RISC-V instruction set [9], while currently much less popular than many other instruction sets (including open instruction sets such as SPARC or Power), deserves mention as it is a new instruction set (created several years after our project started) that appears have generated significant interest in the research community. Despite the interest, given our goal of minimizing software development effort (thus, binary compatibility), we believe RISC-V is not ideal for several reasons. At this time, the RISC-V instruction set is not yet stable. The privileged ISA (for OSes) is still a draft [10] and future changes are expected to the memory consistency model [11]. Also, like all new instruction sets, there is a lack of existing software, including limited OS and compiler support, and the need to spend effort to port existing code to the new architecture. Despite its young age, the RISC-V instruction set already appears to be fragmented, creating potential portability difficulties. To target a large design space of implementations, RISC-V is designed around a small base instruction set with many optional extensions. Similarly, the current privileged ISA draft specifies a base privilege mode, and three more optional modes. It appears OS-level binary compatibility is not a design goal, as there seems to be an intention to use incompatible privileged ISAs in the future (while preserving the user-mode ISA) [9, 10]. While x86, too, has been extended many times, each extension is almost always a superset of the previous implementation, which preserves backwards compatibility even for OS code.

Using an instruction set with legacy compatibility of course comes with the cost of supporting decades of legacy. The x86 instruction set is large and complex, especially when compared to simple RISC instructions sets like MIPS. Complex behaviours (such as string repeat operations and operations that change privilege levels) require some microcode to implement. The need to be binary-compatible with existing operating systems increases processor design effort because even rarely-used behaviours are well-defined (earlier implementations are a *de facto* behavioural specification), leaving fewer truly-undefined behaviours that could be exploited to simplify the microarchitecture. Not only must the processor work for an operating system that has been modified to run on this implementation, it must also work with many unmodified operating systems, including those that may have made assumptions about how earlier processors handled supposedly-undefined behaviours.

Surprisingly, modern ARM instruction sets are not much simpler than x86, despite being called "RISC". The ARM instruction set also needs decoding into micro-ops. The ARM memory system is similar to x86, supporting unaligned data accesses and hardware TLB miss handling. Unlike x86, ARMv7 also allows predication of most instructions and an implicit barrel shift of one operand for most arithmetic instructions, which are features that are difficult to implement in out-of-order processors. On the other hand, one particularly complex-to-implement feature of x86 is variable instruction length where the length is non-trivial to compute (1 to 15 bytes), while ARM Thumb's variable-length instructions only have two possible lengths (2 or 4 bytes).

Due to our preference for minimizing software development effort, we wish to use a popular instruction set to reuse existing applications, tools, and operating systems. We chose x86 because of its binary compatibility even between operating systems. While x86 is more complex, the extra complexity due specifically to x86 is modest — much of the complexity of supporting an operating system (supporting exceptions, paging, privilege levels, etc.) is common to most instruction sets.

1.4 Contributions

This thesis documents the design of a superscalar out-of-order x86 FPGA soft processor, from a study of the FPGA substrate to the design of circuits. In this thesis, we make the following contributions:

- We study how the FPGA substrate affects circuits and out-of-order processor microarchitecture
- We design a microarchitecture that implements the x86 instruction set sufficiently well to boot modern operating systems. The microarchitecture is verified using a detailed cycle-level software simulation of the processor pipeline, successfully booting 16 different operating systems (See Chapter 4). As the simulator is written at an abstraction level not much higher than digital logic, we have confidence that the resulting microarchitecture has reasonable circuit implementations.
- We design FPGA circuits for many of the processor components that implement the out-of-order microarchitecture. Many of the critical components are implemented, which allows reasonable area and frequency estimates. However, as some components are not yet implemented, the processor circuit is not functional (See Table 13.5 on page 217 for implementation status.)
- We evaluate the performance and costs of an out-of-order soft processor compared to a commercial in-order soft processor and x86 hard processors

1.5 Organization

The rest of the thesis starts with a background on out-of-order processor microarchitecture and a review of FPGA soft processors, in Chapter 2.

Before we start designing our soft processor microarchitecture, we need to first understand how the FPGA substrate differs from the custom CMOS substrate usually assumed by hard processor microarchitectures. Thus, Chapter 3 presents a study of how various common subcircuits differ when implemented on an FPGA substrate, and their impact on some processor microarchitecture design decisions.

We begin our processor design journey in Chapter 4, where we present our methodology for modelling, evaluating, and verifying our processor microarchitecture. Chapter 5 presents a high-level description of our processor design and our approach to implementing the x86 instruction set. Chapters 6 through 12 then present the design of each processor component of our microarchitecture in more detail, including design decisions, performance evaluation, circuit designs, and hardware speed and cost.

Finally, in Chapter 13, we evaluate the performance, area, and frequency of the entire processor design, and compare these to the commercially-available Nios II/f FPGA soft processor. We also compare performance-per-clock with x86 hard processors to show how our design compares to the large design space of past and current x86 processors.

Chapter 2

Background

2.1 Out-of-Order Processors

In the single-threaded sequential programming model, programs consist of instructions that are sequentially executed. However, instructions often do not depend on the result of the immediately preceding instruction. Out-of-order processors use hardware mechanisms to dynamically track data dependencies between instructions, execute instructions whenever their dependencies are available, and create the illusion of sequential execution when necessary (e.g., at branch mispredictions or interrupts) [12].

A typical out-of-order processor microarchitecture is shown in Figure 2.1. Instruction fetch, decode, register renaming, and instruction commit are performed in program order, similar to pipelined in-order processors. The reorder buffer (ROB) tracks instructions as they progress through the out-of-order section of the processor, to allow recovery from a pipeline flush by tracking the program-order of the instructions. The instruction scheduler tracks the availability of operands and is the hardware unit for scheduling the instructions' out-of-order behaviour. When instructions are completed, they are committed in program order by dequeuing them from the ROB.

The in-order portion from instruction fetch through rename is often called the *front-end*, with the rest of the pipeline called the *back-end*. This section briefly describes the components; more detail will appear later.

2.1.1 Front End

The front-end of the processor is responsible for fetching and decoding instructions. The instruction fetch unit fetches a block of instructions (or bytes, for variable-length instruction sets such as x86 or ARM Thumb) each clock cycle. Branch predictors are included so that instruction fetch can continue along the correct (but non-contiguous) path with minimal extra delay when a taken branch is encountered, at least when correctly predicted. Branch prediction is particularly challenging because, at this stage, the branch predictor has no knowledge of the branch target address, or even if there actually is a branch instruction encoded in the fetched instruction bytes. The fetched instruction bytes are then queued and sent for decoding.

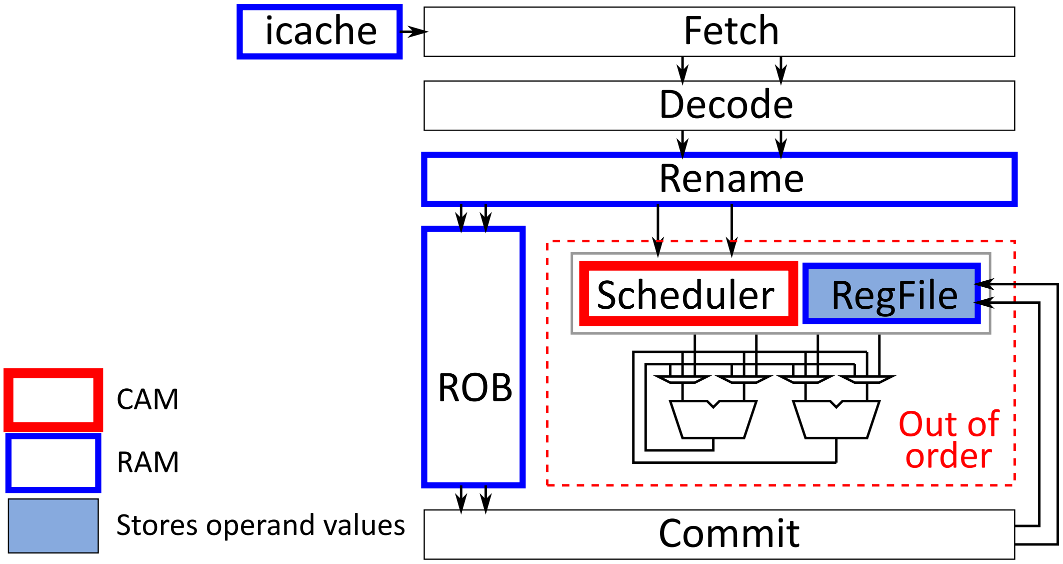


Figure 2.1: A typical out-of-order processor microarchitecture

The instruction decoders parse the instruction bytes to determine what each instruction does and what operands it consumes or produces. For an instruction set with variable-length instructions, decoding is particularly difficult, first requiring a *length decoding* stage to find instruction boundaries in the byte stream before sending each instruction (of variable length) to a decoder. Decoded instructions are then sent to the register renamer.

Register renaming maps each instruction's operand registers from a set of *architectural* registers to a larger set of *physical* registers. The main purpose is to remove false dependencies (write-after-write and write-after-read) for improved performance, but the same mechanism is often used to allow rolling back the processor state during a pipeline flush. Rollback is accomplished by speculatively mapping architectural registers to new physical registers, but not freeing the old value of the architectural registers until instructions are committed, and recovering the old renamer mapping table on a pipeline flush [13, 14]. After register renaming, the instructions now encode operations with *physical* register source and destination operands. These are then enqueued into the instruction scheduler where they can be executed in dataflow order.

2.1.2 Back End

The instruction scheduler holds instructions along with the status of their source operands. It tracks the execution of instructions to determine when operands are available. An instruction waiting in the scheduler becomes *ready* when its source operands are available. The scheduler selects some ready instructions each cycle and sends them to the execution units for execution.

There are two major variations of instruction scheduling: Data-capturing, where an instruction's source operand values are stored in the scheduler, and non data-capturing, where only the availability of source operands are tracked [15]. Both styles have been used in commercial processors. Data-capturing schedulers are larger due to the storage space needed to hold operand values, but non data-capturing schedulers often need one extra cycle to read operands from the register file between scheduling and execution. Data-capturing schedulers are also sometimes called *reservation stations* [13].

After execution, an instruction is considered *complete*. Completed instructions are dequeued from the ROB in program order and committed if there is no branch misprediction or exception condition detected. A branch misprediction or exception causes a pipeline flush, discarding subsequent ROB entries (instructions on the wrong path), recovering the register renamer state to undo any speculative changes, and then restarting the processor along the correct path.

2.2 Soft Processors

Soft processors are commonly used as part of an FPGA-based system. Current commercially-available single-threaded soft processors tend to be small, single-issue, in-order, pipelined processors [16–20]. Table 2.1 lists some relevant single-threaded soft processors.

Out-of-order processors have been synthesized for FPGAs, but are often using an FPGA solely as an ASIC prototyping tool rather than designing the processor microarchitecture targeting the FPGA. Intel's out-of-order x86 Nehalem is split into five FPGAs and runs at 520 kHz [21]. RISC-V BOOM is an out-of-order RISC-V processor that runs on an FPGA, but is aimed at an ASIC implementation and not optimized for FPGAs [22]. The OpenRISC OPA is a work in progress that has the basics of an out-of-order OpenRISC processor, designed for FPGA implementation [23].

Perhaps due to the complexity and high hardware costs of out-of-order microarchitectures, many soft processors choose to exploit data-level (vector [24–26], VLIW [27], SIMD [28, 29]) or thread-level [3, 29–31] parallelism instead of instruction-level parallelism. There have also been efforts to create soft processor architectures that are custom-designed to match the FPGA substrate [3, 32]. However, all of these methods require rewriting software to accommodate the new programming model, along with new compilers and development tools.

One notable example of a soft processor that increased performance without changing the programming model is SPREX [33], which uses runahead execution with an in-order processor to create and exploit memory-level parallelism. Runahead execution continues executing past a cache miss in an effort to discover more cache misses that could be serviced concurrently with the first cache miss.

Processor	Microarchitecture	LUTs	MHz	FPGA
Nios II/f [17]	6-stage pipelined	1100 ALUT	240	Stratix IV
MicroBlaze [16]	5-stage pipelined	2100 LUT	246	Virtex-7
Leon 3 [20]	7-stage pipelined	5600 ALUT	150	Stratix IV
OpenRISC OR1200 [34]	5-stage pipelined	3500 ALUT	130	Stratix IV
RISC-V BOOM [35]	2-wide out-of-order	—	50	Zync-7000
OpenRISC OPA [23]	3-wide out-of-order	12600 ALUT	215	Stratix IV
SPREX [33]	5-stage pipelined with runahead	1800 ALUT	150	Stratix III

Table 2.1: Summary of single-threaded soft processors

2.2.1 Soft Out-of-Order Components

In addition to the out-of-order processor projects mentioned above (OpenRISC OPA and RISC-V BOOM), there has been prior work on building various components of out-of-order soft processors. Prior work has focused on out-of-order instruction scheduling [36–39], and there has also been other work on register renaming [40] and non-blocking caches [41].

Perhaps this illustrates the amount of effort required to design a highly-tuned out-of-order processor: Projects that have carefully designed one block of an out-of-order processor have not yet progressed to a complete or nearly-complete out-of-order processor.

2.2.2 Soft x86 Processors

In addition to the Intel Nehalem mentioned in the previous section, we know of several other x86-based FPGA soft processor cores. Table 2.2 summarizes these processors.

Some FPGA x86 processors are based on RTL of a commercial x86 processor that has been synthesized for an FPGA. In addition to the Nehalem [21], the Atom (Bonnell) [42] and Pentium (P54C) [43] have been synthesized. Huang et al. [44] synthesized a single-issue 8-stage in-order processor based on the AMD (National Semiconductor) Geode GX2 CPU.

Several other projects designed their x86 processors from scratch. These are generally designed for correct functionality rather than performance, and often only support the older 16-bit instruction set. For example, Zet [45] and CPU86 [46] both support the 8086 instruction set. The ao486 [47, 48] supports the 32-bit 80486 instruction set, and is complete enough to run Windows 95. It uses a 4-stage pipeline, but achieves only about a quarter of the per-clock performance of a 5-stage pipelined Intel 80486 (on Dhrystone).

These processors have all booted up operating systems in hardware, which our hardware implementation is not yet complete enough to do. However, all of these projects either did not need to design the CPU core from scratch, or had low complexity and performance, making the processor core design less challenging.

Processor	Microarchitecture	LUTs	MHz	FPGA
Intel Nehalem [21]	64 bit 4-wide out-of-order	∼670K 6-LUT	0.52	Virtex-5
Intel Atom [42]	64 bit 2-wide in-order	176K 6-LUT	50	Virtex-5
Intel Pentium [43]	32 bit 2-wide in-order	66K 4-LUT	25	Virtex-4
AMD Geode GX2 [44]	32 bit 1-wide in-order, out-of-order FPU	130K 4-LUT	33	Virtex-4
ao486 [47]	32 bit 1-wide in-order	24K ALUT	100	Stratix IV
Zet [45]	16 bit multi-cycle	3.1K ALUT	63	Stratix IV
CPU86 [46]	16 bit multi-cycle	3.5K ALUT	105	Stratix IV

Table 2.2: Summary of existing x86 soft processors on FPGA

2.3 FPGA Architecture and Circuit Design

Many of the pieces of our soft processor have carefully-designed circuits tuned for speed. This circuit tuning relies on a knowledge of the underlying FPGA architecture and a sense of how each component performs. This section provides a very brief overview of the portions of our target Stratix IV FPGA architecture that were useful for our circuit design and tuning. More details on how the FPGA substrate affects particular circuits will be presented in Chapter 3 (using a Stratix III, with a nearly identical FPGA architecture as a Stratix IV).

The FPGA core consists of an array of fracturable LUTs called Adaptive Logic Modules (ALM), block SRAMs, and DSP blocks (containing multipliers, adders, and a few other functions). There is considerable complexity in the periphery as well (I/O pins, transceivers), but these are not relevant for a processor core until it needs to interface to external devices, so we do not discuss them further.

2.3.1 Adaptive Logic Modules

ALMs are essentially fracturable 6-input LUTs with 8 input pins [49]. A 6-LUT behaves like a truth table and can implement any 6-input, 1 output logic function. The 8 input pins are used when the 6-LUT is fractured into two smaller LUTs, for example, when implementing the classical two independent 4-LUTs.

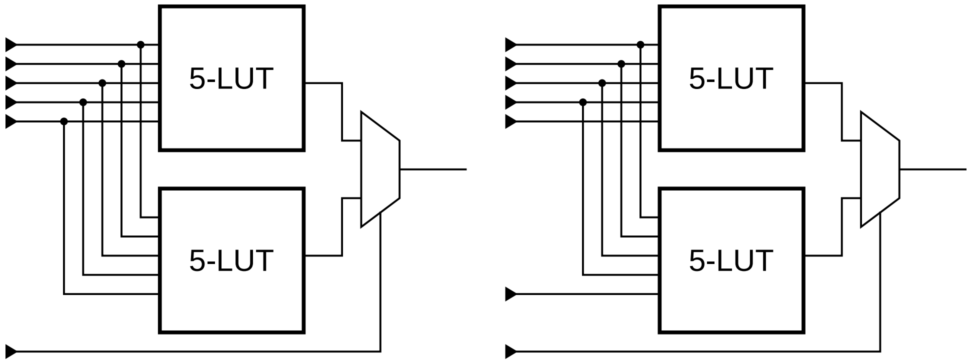
As our circuits are tuned for speed, we generally tried to pack functionality into the LUTs with as many inputs as possible, to reduce the number of logic levels. Although fracturing the ALM into two LUTs saves area by allowing two logic functions to fit in an ALM, this tends to increase the logic depth, so we usually do not deliberately aim for the 3-input or 4-input modes. At the other extreme, the 7-input mode is particularly useful, though difficult to use because it is not a true 7-LUT. The ALM's 6-LUT mode can be viewed as two 5-LUTs sharing all inputs selected by a 2-to-1 multiplexer. The 7-input mode is a straightforward extension of this by unsharing one of the inputs. The 7-input mode implements any function that can be expressed as a 2-to-1 multiplexer selecting between two 5-LUTs that share 4 inputs. Figure 2.2 illustrates this mode.

On the fastest speed grade, the delay per logic level tends to range between 300 ps to 800 ps, depending on the complexity of the circuit and routing. Trying to pack logic into LUTs with more inputs tends to create more routing complexity. Despite this, we have found that our design heuristic of minimizing logic depth works well. Deliberately increasing logic depth in the hope of reducing routing complexity enough to compensate for the extra logic level tends to result in a net delay increase.

The soft logic also contains hard carry chains for implementing addition. Although the ripple carry delay is quite fast (under 20 ps per bit), the delay to get on the carry chain exceeds one LUT delay. This overhead makes carry chains largely useless for functions other than addition, such as for wide AND or OR gates.

2.3.2 Clusters: Logic Array Blocks

ALMs are clustered into LABs (Logic Array Blocks) of 10 ALMs each. Since we have no easy way of influencing how the packing algorithm clusters ALMs into LABs, the impact of LABs were not considered



(a) 6-LUT

(b) 7-input function

Figure 2.2: Altera ALMs implementing 6-LUT and some 7-input functions

during circuit design.

2.3.3 Block RAM

Although block RAMs have good area efficiency and run at high frequency (as would be expected for a hard block), they actually have surprisingly high latency. The high frequencies are only achieved if the block RAM has both input and output registers located *in* the block RAMs, resulting in the block RAMs essentially borrowing about 1 ns (or about 30% of the cycle time of our 300 MHz target frequency) from the pipeline stage before and 400 ps after the block RAM, compared to about 70 ps before and 300 ps after a regular flip-flop.

As single-threaded processors are highly sensitive to latency (unlike typical FPGA applications that tolerate very deep pipelines), and our frequency target is quite high, we try to avoid using block RAMs in speed-critical portions of the processor core. Small LUT RAMs (MLAB), despite lower frequency, have much lower read latency (one LUT delay) than block RAMs.

2.3.4 DSP Blocks

In a processor, we make use of only the hard multiplier portion of the DSP block, while paying the delay overhead of all the other DSP block functionality. Like block RAMs, DSP blocks have high frequency only if the pipeline registers inside the DSP block are used. In this case, we have no reasonable alternative method of implementing multiplication, as LUT-based multipliers are much larger and likely not any faster.

2.4 Summary

This chapter presented background on out-of-order processor microarchitecture and existing soft processors. Current soft processors from the FPGA vendors use simple in-order microarchitectures and optimized for frequency. When more performance is required, soft processors have usually chosen to exploit data or thread-level parallelism rather than instruction-level parallelism.

We also presented a brief overview of FPGA architecture, and how it affected some of our circuit designs. The details of our target FPGA's logic elements (ALMs) are relevant because many of our circuits are hand-mapped to LUTs for minimum logic depth. Hardened circuit elements (adders, multipliers, and SRAM) are not fast enough to significantly affect circuit design. These hard blocks are used when an adder, multiplier, or SRAM block, respectively, are needed, but they are not fast enough to be able to replace logic elements for other (related) circuits. The next chapter will discuss in more detail the impact of FPGA architecture on circuit design and processor microarchitecture.

Chapter 3

Comparing FPGA vs. Custom CMOS Circuits

Before we begin designing an FPGA-tuned soft processor, we need to understand how the FPGA substrate differs from the custom CMOS substrate usually assumed by processor microarchitecture research. This chapter presents comparisons of delay and area of a set of processor building block circuits when implemented on custom CMOS and FPGA substrates, then uses these results to show how soft processor microarchitectures should be different from those of hard processors.

3.1 Introduction

The area, speed, and energy consumption of a digital circuit will differ when it is implemented on different substrates such as custom CMOS, standard cell ASICs, and FPGAs. Those differences will also change based on the nature of the digital circuit itself. Having different cost ratios for different circuit types implies that systems built using a range of different circuit types must be tuned for each substrate. In this chapter, we compare the custom CMOS and FPGA substrates with a focus on implementing instruction-set processors — we examine both full processors and subcircuits commonly used by processors, and explore the microarchitecture trade-off space of soft processors in light of these differences.

Previous studies have measured the average delay and area of FPGA, standard cell, and custom CMOS substrates across a large set of benchmark circuits [52,53]. While these earlier results are useful in determining an estimate of the size and speed of the full system that can be implemented on FPGAs, it is often necessary to compare the relative performance of specific types of "building block" circuits in order to have enough detail to guide processor microarchitecture design decisions.

This chapter makes two contributions:

1. We compare the delay and area of custom CMOS and FPGA implementations of a specific set of building block circuits typically used in processors.
2. Based on these measured delay and area ratios, and prior custom CMOS processor microarchitecture knowledge, we discuss how processor microarchitecture design trade-offs should change on an FPGA substrate.

This chapter has appeared at FPGA 2011 and in TVLSI [50,51].

We begin with a survey of prior work in Section 3.2 and describe our methodology in Section 3.3. We then present the building block comparisons in Section 3.4 and their impact on microarchitecture in Section 3.5, and conclude in Section 3.6.

3.2 Background

3.2.1 Technology Impact on Microarchitecture

One of the goals in processor microarchitecture design is to make use of circuit structures that are best suited to the underlying implementation technology. Thus, studies on how process technology trends impact microarchitecture are essential for designing effective microarchitectures that best fit the ever-changing process characteristics. Issues currently facing CMOS technology include poor wire delay scaling, high power consumption, and more recently, process variation. Microarchitectural techniques that respond to these challenges include clustered processor microarchitectures and chip multiprocessors [54, 55].

Circuits implemented on an FPGA substrate face a very different set of constraints from custom CMOS. Although power consumption is important, it is not currently the dominant design constraint for FPGA designs. FPGA designs run at lower clock speeds and the architectures of FPGAs are already designed to give reasonable power consumption across the vast majority of FPGA user designs. Interestingly, area is often the primary constraint due to high area overhead of the programmability endemic to FPGAs. This different perspective, combined with the fact that different structures have varying area, delay, and power characteristics between different implementation technologies mean that understanding and measuring these differences is required to make good microarchitecture choices to suit the FPGA substrate. Characteristics such as inefficient multiplexers and the need to map RAM structures into FPGA hard SRAM blocks are known and are generally adjusted for by modifying circuit-level, but not microarchitecture-level, design [42, 43, 56, 57].

3.2.2 Measurement of FPGAs

Kuon and Rose have measured the area, delay, and power overheads of FPGAs compared to a standard cell ASIC flow on 90 nm processes [52]. They used a benchmark set of *complete* circuits to measure the overall impact of using FPGAs compared to ASICs and the effect of FPGA hard blocks. They found that circuits implemented on FPGAs consumed 35× more area than on standard cell ASIC for circuits that did not use hard memory or multiplier blocks, to a low of 18× for those that used both types. The minimum cycle time (their measure of speed) of the FPGA circuits ranged from 3.0 to 3.5× greater than that of the ASIC implementations, and were not significantly affected by hard blocks. Chinnery and Keutzer [53] made similar comparisons between standard cell and custom CMOS and reported a delay ratio of 3 to 8×. Combined, these reports suggest that the delay of circuits implemented on an FPGA would be 9 to 28× greater than on custom CMOS. However, data for full circuits are insufficiently detailed to guide microarchitecture-level decisions, which is the focus of this chapter.

3.3 Methodology

We seek to measure the delay and area of FPGA building block circuits and compare them against their custom CMOS counterparts, resulting in *area ratios* and *delay ratios*. We define these ratios to be the area or delay of an FPGA circuit divided by the area or delay of the custom CMOS circuit. A higher ratio means the FPGA implementation is worse. We compare several complete processor cores and a set of building block circuits against their custom CMOS implementations, then observe which types of building block circuits have particularly high or low overhead on an FPGA.

As we do not have the expertise to implement highly-optimized custom CMOS circuits, most of our building block circuit comparisons use data from custom CMOS implementations found in the literature. We focus mainly on custom CMOS designs built in 65 nm processes, because it is the most recent process where design examples are readily available in the literature. The custom CMOS data is compared to an Altera Stratix III 65 nm FPGA. In most cases, the equivalent FPGA circuits were implemented on an FPGA using the standard FPGA CAD flows. Power consumption is not compared due to the scarcity of data in the literature and the difficulty in standardizing testing conditions such as test vectors, voltage, and temperature.

We normalize area measurements to a 65 nm process using an ideal scale factor of 0.5× area between process nodes. We normalize delay using published ring oscillator data, with the understanding that these reflect gate delay scaling more than interconnect scaling. Intel reports 29% fanout-of-one (FO1) delay improvement between 90 nm and 65 nm, and 23% FO2 delay improvement between 65 nm and 45 nm [58, 59]. The area and delay scaling factors used are summarized in Table 3.1.

Delay is measured as the longest register to register path (sequential) or input to output path (combinational) in a circuit. In papers that describe CMOS circuits embedded in a larger unit (e.g., a shifter inside an ALU), we conservatively assume that the subcircuit has the same cycle time as the larger unit. In FPGA circuits, delay is measured using register to register paths, with the register delay subtracted out when comparing subcircuits that do not include a register (e.g., wire delay).

To measure FPGA resource usage, we use the "logic utilization" metric as reported by Quartus rather than raw LUT count, as it includes an estimate of how often a partially used fracturable logic element can be shared with other logic. We count partially-used memory and multiplier blocks as entirely used since it is unlikely another part of the design can use a partially-used memory or multiplier block. Table 3.2 shows the areas of the Stratix III FPGA resources. The FPGA tile areas include the area used by the FPGA routing network so we do not track routing resource use separately. The core of the largest Stratix III (EP3LS340) FPGA contains 13 500 clusters (Logic Array Block, LAB) of 10 logic elements (Adaptive Logic Module, ALM) each, 1040 9-kbit (M9K) memories, 48 144-kbit (M144K) memories, and 72 DSP blocks, for a total of 18 621 LABs equivalent area and 412 mm^2 core area.

We implemented FPGA circuits using Altera Quartus II 10.0 SP1 CAD flow and employed the fastest speed grade of the largest Stratix III device. We set timing constraints to maximize clock speed, reflecting the use of these circuits as part of a larger circuit in the FPGA core, such as a soft processor. We note that the custom CMOS design effort is likely to be much higher than for FPGA designs because there is more potential gain for optimization and much of the design process is automated for FPGA designs.

	90 nm	65 nm	45 nm
Area	0.5	1.0	2.0
Delay	0.78	1.0	1.23

Table 3.1: Normalization factors between processes

Resource	Relative Area (Equiv. LABs)	Tile Area (mm^2)
LAB	1	0.0221
ALUT (half-ALM)	0.05	0.0011
M9K memory	2.87	0.0635
M144K memory	26.7	0.5897
DSP block	11.9	0.2623
Total core area	18 621	412

Table 3.2: Stratix III FPGA resource area usage

3.4 Custom CMOS vs. FPGA

3.4.1 Complete Processor Cores

We begin by comparing complete processor cores implemented on an FPGA vs. custom CMOS to provide context for the subsequent building block measurements. Table 3.3 shows a comparison of the area and delay of four commercial processors that have both custom CMOS and FPGA implementations, including in-order, multithreaded, and out-of-order processors. The FPGA implementations are synthesized from RTL code for the custom CMOS processors, with some FPGA-specific circuit-level optimizations. However, the FPGA-specific optimization effort is smaller than for custom CMOS designs and could inflate the area and delay ratios slightly.

The OpenSPARC T1 and T2 cores are derived from the in-order multithreaded UltraSPARC T1 and T2, respectively [64]. The OpenSPARC T2 processor core includes a floating-point unit. We synthesized one processor core for the Stratix III FPGA, with some debug features unnecessary in FPGA designs removed, such as register scan chains and SRAM redundancy in the caches.

The Intel Atom is a dual-issue in-order 64-bit x86 processor with two-way multithreading. Our Atom

Processor	Custom CMOS		FPGA		Ratios		FPGA Resource Utilization				
	f_{max} (MHz)	Area (mm^2)	f_{max} (MHz)	Area (mm^2)	f_{max}	Area	Utilization (ALUT)	ALUT	Registers	M9K	DSP
SPARC T1 (90 nm) [60]	1800	6.0	79	100	23	17	86 597	54 745	54 950	66	1
SPARC T2 (65 nm) [61]	1600	11.7	88	294	18	25	250 235	163 524	116 085	275	0
Atom (45 nm) [42, 62]	>1300	12.8	50	350	26	27	— 85% of Virtex-5 LX330 (176K LUT) —				
Nehalem (45 nm) [21, 63]	3000	51	–	1320	–	26	— 320% of Virtex-5 LX330 (670K LUT) —				
Geometric Mean					22	23					

Table 3.3: Complete processor cores. Area and delay normalized to 65 nm.

processor comparisons use published FPGA synthesis results by Wang et al. [42], which includes only the processor core without L2 cache, and occupies 85% of the LUTs of the largest 65 nm Virtex-5 FPGA (XC5VLX330). They do not publish a detailed breakdown of the FPGA resource utilization, so the FPGA area is estimated by assuming the core area of the largest Virtex-5 is the same as the largest Stratix III and treating the LUT usage as the fraction of the die used.

The Intel Nehalem is an out-of-order 64-bit x86 processor with two-way multithreading. The FPGA synthesis by Schelle et al. [21] includes the processor core and does not include the per-core L2 cache, with an area utilization of roughly 320% of the largest Virtex-5 FPGA. They partitioned the processor core across five FPGAs and time-multiplexed the communication between FPGAs, so the resulting clock speed (520 kHz) is not useful for estimating delay ratio.

Table 3.3 compares the four processors' speed and area. For custom CMOS processors, the highest commercially-available speed is listed, scaled to a 65 nm process using linear delay scaling as described in Section 3.3. The area of a custom CMOS processor is measured from die photos, including only the area of the processor core that the FPGA version implements, again scaled using ideal area scaling to a 65 nm process. The sixth and seventh columns contain the speed and area ratio between custom CMOS and FPGA, with higher ratios meaning the FPGA is worse.

For the two OpenSPARC processors, FPGA area is measured using the resource usage (ALUT logic utilization, DSP blocks, and RAM) of the design reported by Quartus multiplied by the area of each resource in Table 3.2. The FPGA synthesis of the Intel processors use data from the literature which only gave approximate area usage, so we list the logic utilization as a fraction of the FPGA chip.

Overall, custom processors have delay ratios of 18–26× and area ratios of 17–27×. We use these processor core area and delay ratios as a reference point for the building block circuit comparisons in the remainder of this chapter. For each building block circuit, we compare the FPGA vs custom CMOS area and delay ratios for the building block circuit to the corresponding ratios for processor cores to judge whether a building block circuit's area and delay are better or worse than the overall ratios for a processor core.

Interestingly, there is no obvious area ratio trend with processor complexity — for example, we might expect that an out-of-order processor synthesized for an FPGA, such as the Nehalem, to have a particularly high area ratio, but it does not. We speculate that this is because expensive CAMs are only a small portion of the hardware added by a high-performance microarchitecture. The added hardware includes a considerable amount of RAM and other logic, since modern processor designs already seek to minimize the use of CAMs due to their high power consumption. On the FPGA-synthesized Nehalem, the hardware structures commonly associated with out-of-order execution (reorder buffer, reservation stations, register renaming) consume around 45% of the processor core's LUT usage [21].

3.4.2 SRAM Blocks (Low Port Count)

SRAM blocks are commonly used in processors for building caches and register files. SRAM performance can be characterized by latency and throughput. Custom CMOS SRAM designs can trade latency and throughput by pipelining, while FPGA designs are limited to the prefabricated SRAM blocks on the

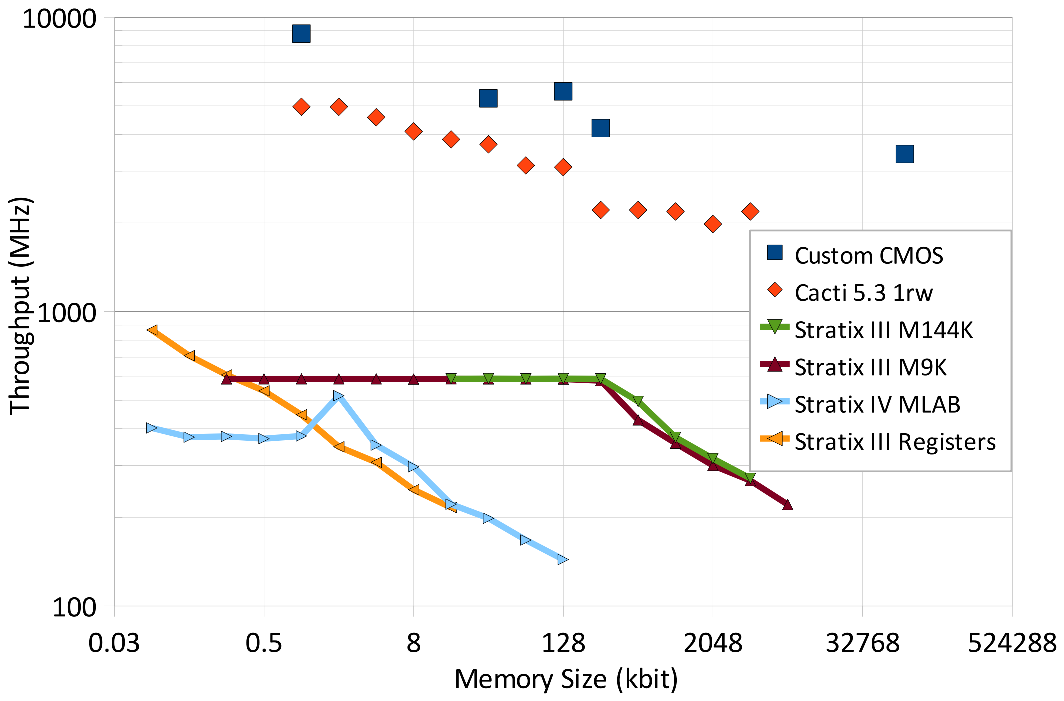


Figure 3.1: SRAM throughput

FPGA.

Logical SRAMs targeting the Stratix III FPGA can be implemented in four different ways: using one of the two physical sizes of hard block SRAM, using the LUT RAMs in Memory LABs (MLABs allow the lookup tables in a LAB to be converted into a small RAM), or in registers and LUTs. The throughput and density of the four methods of implementing RAM storage are compared in Table 3.4 to five high-performance custom SRAMs in 65 nm processes. In this section, we focus on RAMs with one read-write port (which we will refer to as 1rw), as it is a commonly-used configuration in larger caches in processors, but some custom CMOS SRAMs have unusual port configurations, such as being able to do two reads *or* one write [65]. The size column lists the size of the SRAM block. For MLAB (LUT RAM, 640 bit), M9K (block RAM, 9 kbit), and M144K (block RAM, 144 kbit) FPGA memories, memory size indicates the capacity of the memory block type. The f_{max} and area columns list the maximum clock speed and area of the SRAM block. Because of the large variety of SRAM block sizes, it is more useful to compare bit *density* than area. The last two columns of the table list f_{max} and bit density ratios between custom CMOS SRAM blocks and an FPGA implementation of the same block size on an FPGA. Higher density ratios indicate worse density on FPGA.

The density and throughput of custom CMOS and FPGA SRAMs listed in Table 3.4 are plotted against memory size in Figures 3.1 and 3.2. The plots include data from CACTI 5.3, a CMOS memory performance and area model [71]. There is good agreement between the CACTI models and the design examples from the literature, although CACTI appears to be slightly more conservative.

The throughput ratio between FPGA memories and custom is 7–10×, lower than the overall delay

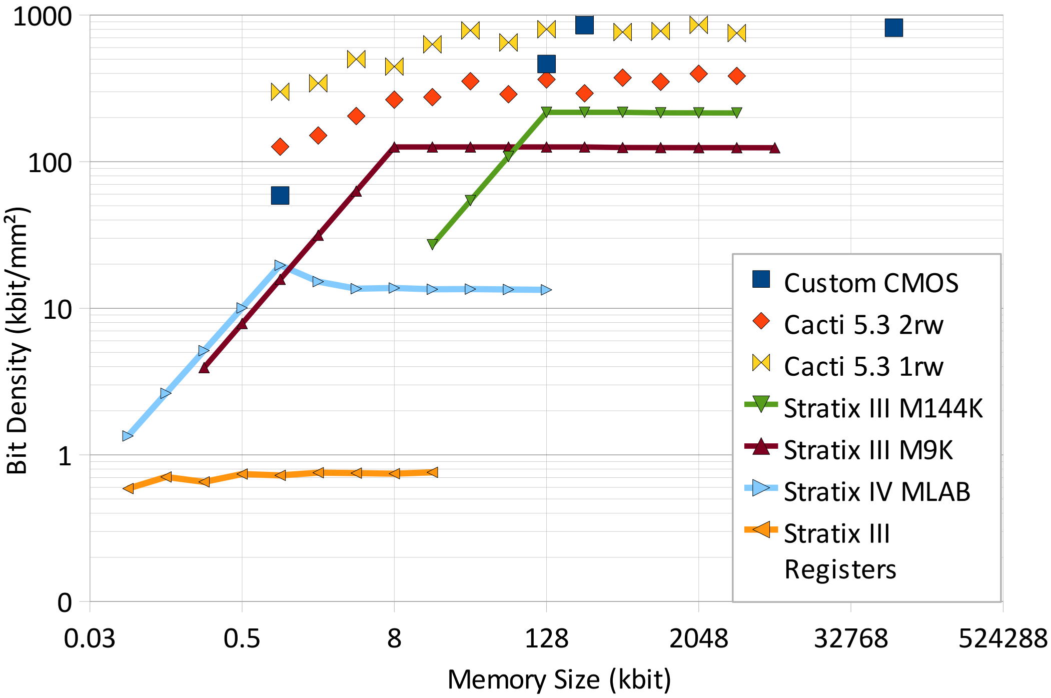


Figure 3.2: SRAM density

ratio of 18–26×, showing that SRAMs are relatively fast on FPGAs. It is surprising that this ratio is not even lower because FPGA SRAM blocks have little programmability. The 2 kbit MLAB (64×32) memory has a particularly low delay because its 64-entry depth uses the 64×10 mode of the MLAB, allowing both its input and output registers to be packed into the same LAB as the memory itself (each LAB has 20 registers), yet it does not need external multiplexers to stitch together multiple MLABs.

The FPGA data above use 32-bit wide data ports (often the width of a register on 32-bit processors) that slightly underutilize the native FPGA 36-bit ports. The raw density of a fully-utilized FPGA SRAM block is listed in Table 3.4. Below 9 kbit, the bit density of FPGA RAMs falls off nearly linearly with reducing RAM size because M9Ks are underutilized. The MLABs use 20-bit wide ports, so a 32-bit wide memory block always uses at least two MLABs, utilizing 80% of their capacity. The MLAB bit density (25 kbit/mm^2) is low, although it is still much better than using registers and LUTs (0.76 kbit/mm^2). For larger arrays with good utilization, FPGA SRAM arrays have a density ratio of only 2–5× vs. single read-write port (1rw)[1] CMOS (and CACTI) SRAMs, far below the full processor area ratio of 17–27×.

As FPGA SRAMs use dual-ported (2rw) arrays, we also plotted CACTI's 2rw model for comparison. For arrays of similar size, the bit density of CACTI's 2rw models are 1.9× and 1.5× the raw bit density of fully-utilized M9K and M144K memory blocks, respectively. This suggests that half of the bit density gap between custom CMOS and FPGA SRAMs in our single-ported test is due to FPGA memories paying the overhead of dual ports.

[1] There are three basic types of memory ports: read (r), write (w) and read-write (rw). A read-write port can read or write, but only to one location each cycle.

Design	Ports	Size	f_{max}	Area	Bit Density	Ratios	
		(kbit)	(MHz)	(mm^2)	($kbit/mm^2$)	f_{max}	Density
IBM 6T 65 nm [65]	2r or 1w	128	5600	0.276	464	9.5	2.1
Intel 6T 65 nm [66]	1rw	256	4200	0.3	853	7.1	3.9
Intel 6T 65 nm [67]	1rw	70 Mb	3430	110 [68]	820	–	–
IBM 8T 65 nm SOI [69]	1r1w	32	5300	–	–	9.0	–
Intel 65 nm Regfile [70]	1r1w	1	8800	0.017	59	15	3.7
Stratix III FPGA							
Registers	1rw	–	–	–	0.76		
MLAB	1rw	0.625	450	0.025	25		
M9K	1rw	9	590	0.064	142		
M144K	1rw	144	590	0.59	244		

Table 3.4: Custom CMOS and FPGA SRAM blocks

	CACTI 5.3		FPGA		Ratios	
Ports	f_{max} (MHz)	Density ($\frac{kbit}{mm^2}$)	f_{max} (MHz)	Density ($\frac{kbit}{mm^2}$)	f_{max}	Density
2r1w	3750	177	497	63	7.6	2.8
4r2w	3430	45	228	0.25	15	179
6r3w	3270	27	214	0.20	15	140
8r4w	2950	17	178	0.15	17	109
10r5w	2680	11	168	0.11	16	104
12r6w	2450	8.0	140	0.091	18	87
14r7w	2250	6.1	130	0.080	17	75
16r8w	2070	4.8	126	0.064	16	74
Live Value Table (LVT)						
4r2w			375	3.9	9.2	12
8r4w			280	0.98	11	17

Table 3.5: Multiported 8 kbit SRAM. LVT data from [72]

For register file use where latency may be more important than memory density, custom processors have the option of trading throughput for area and power by using faster and larger storage cells. The 65 nm Pentium 4 register file trades decreased bit density for 9 GHz single-cycle performance [70]. FPGA RAMs lack this flexibility, and the delay ratio is even greater (15×) for this specific use.

3.4.3 Multiported SRAM Blocks

FPGA hard SRAM blocks can typically implement up to two read-write ports (2rw). Implementing more read ports on an FPGA can be achieved reasonably efficiently by replicating the memory blocks, but increasing the number of write ports is more difficult. A multiple write port RAM can be implemented using registers for storage and LUTs for multiplexing and address decoding, but is inefficient. A more efficient method using hard RAM blocks for most of the storage replicates memory blocks for each write and read port and uses a live value table (LVT) to indicate for each word which of the replicated memories holds the most recent copy [72].

We present data for multiported RAMs implemented using registers, LVT-based multiported memories from [72], and CACTI 5.3 models of custom CMOS multiported RAMs. Like for single-ported SRAMs (Section 3.4.2), we report the random cycle time of a pipelined custom CMOS memory. We focus on a 256×32-bit (8 kbit) memory block with twice as many read ports as write ports ($2N$ read, N write) because it is a port configuration often used in register files in processors and the size fits well into an M9K memory block. Table 3.5 shows the throughput and density comparisons.

The custom CMOS vs. FPGA bit density ratio is 2.8× for 2r1w, and increases to 12× and 179× for 4r2w LVT- and register-based memories, respectively. When only one write port is needed (2r1w), the increased area needed for duplicating the FPGA memory block to provide a second read port is less than the area increase for tripling the number of ports from 1rw to 2r1w of a custom CMOS RAM (445 kbit/mm^2 1rw from Section 3.4.2 to 177 kbit/mm^2 2r1w). LVT-based memories improve in density on register-based memories, but both are worse than simple replication used for memories with one write port and multiple read ports.

The delay ratio is 7.6× for 2r1w, and increases to 9× and 15× for 4r2w LVT- and register-based memories, respectively, a smaller impact than the area ratio increase. The delay ratios when using registers to implement memories (15–18×) are higher than those for single-ported RAMs using hard RAM blocks, but still slightly lower than the overall processor core delay ratios.

3.4.4 Content-Addressable Memories

A Content-Addressable Memory (CAM) is a logic circuit that allows associative searches of its stored contents. Custom CMOS CAMs are typically implemented as dense arrays of cells using 9-transistor (9T) to 11T cells compared to 6T used in SRAM and are typically 2–3× less dense than custom SRAMs. Ternary CAMs use two storage cells per "bit" to store three states (0, 1, and don't-care). In processors, CAMs are used in tag arrays for high-associativity caches and translation lookaside buffers (TLBs). CAM-like structures are also used in out-of-order instruction schedulers. CAMs in processors require both frequent read and write capability, but not large capacities. Pagiamtzis and Sheikholeslami give a good overview of the CAM design space [73].

There are several methods of implementing CAM functionality on FPGAs that do not have hard CAM blocks [74]. CAMs implemented in soft logic use registers for storage and LUTs to read, write, and search the stored bits. Another proposal, which we will refer to as BRAM-CAM, stores one-hot encoded match-line values in block RAM to provide the functionality of a $w \times b$-bit CAM using a $2^b \times w$-bit block RAM [75]. The soft logic CAM is the only design that provides one-cycle writes. The BRAM-CAM offers improved bit density but requires two-cycle writes — one cycle each to erase then add an entry. We do not consider FPGA CAM implementations with even longer write times that are only useful in applications where modifying the contents of the CAM is a rare event, such as modifying a network routing table.

Table 3.6 shows a variety of custom CMOS and FPGA CAM designs. Search time indicates the time needed to perform an unpipelined CAM lookup operation. The FPGA vs. custom CMOS ratios compare the delay (search time) and density between each custom CMOS design example and an FPGA soft logic implementation of a CAM of the same size. Figures 3.3 and 3.4 plot these and also 8-bit wide and 128-bit

	Size	Search	Bit	Ratios vs.	
		Time	Density	Soft Logic	
	(bits)	(ns)	($\frac{\text{kbit}}{\text{mm}^2}$)	Delay	Density
Ternary CAMs (65 nm)					
IBM 64×72 [76]	4608	0.6	–	5.4	–
IBM 64×240 [76]	15360	2.2	167	1.8	519
Binary CAMs (65 nm)					
POWER6 8×60 [77]	480	<0.2	–	14	–
Godson-3 64×64 [78]	4096	0.55	76	5	99
Intel 64×128 [79]	8192	0.25	167	14	209
FPGA Ternary CAMs					
Soft logic 64×72	4608	3.2	0.40		
Soft logic 64×240	15360	4.0	0.32		
FPGA Binary CAMs					
Soft logic 8×60	480	2.1	0.83		
Soft logic 64×64	4096	2.9	0.77		
Soft logic 64×128	8192	3.4	0.80		
MLAB-CAM 64×20	1280	4.5	1.0		
M9K-CAM 64×16	1024	2.0	2.0		

Table 3.6: CAM designs

wide soft logic CAMs of varying depth.

CAMs can achieve delay comparable to SRAMs but at a high cost in power. For example, Intel's 64×128 BCAM achieves 4 GHz using 13 fJ/bit/search, while IBM's 450 MHz 64×240 ternary CAM uses 1 fJ/bit/search.

As shown in Table 3.6, soft logic binary CAMs have poor bit density ratios vs. custom CMOS CAMs — from 100 to 210 times worse. We included ternary CAM examples in the table for completeness, but since they are generally not used inside processors, we do not include them when summarizing CAM density ratios. Despite the poor density of soft logic CAMs, the delay ratio is only 14 times worse. BRAM-CAMs built from M9Ks can offer 2.4× better density than soft logic CAMs but needs two cycles per write. The halved write bandwidth of BRAM-CAMs make them unsuitable for performance-critical uses, such as tag matching in instruction schedulers and L1 caches.

We observe that the bit density of soft logic CAMs is nearly the same as using registers to implement RAM (Table 3.4), suggesting that most of the area inefficiency comes from using registers for storage, not the added logic to perform associative searching.

3.4.5 Multipliers

Multiplication is an operation performed frequently in signal processing applications, but not used as often in processors. In a processor, only a few multipliers would be found in ALUs to perform multiplication instructions. Multiplier blocks can also be used to inefficiently implement shifters and multiplexers [83].

Figure 3.5 shows the latency of multiplier circuits on custom CMOS and on FPGA using hard DSP

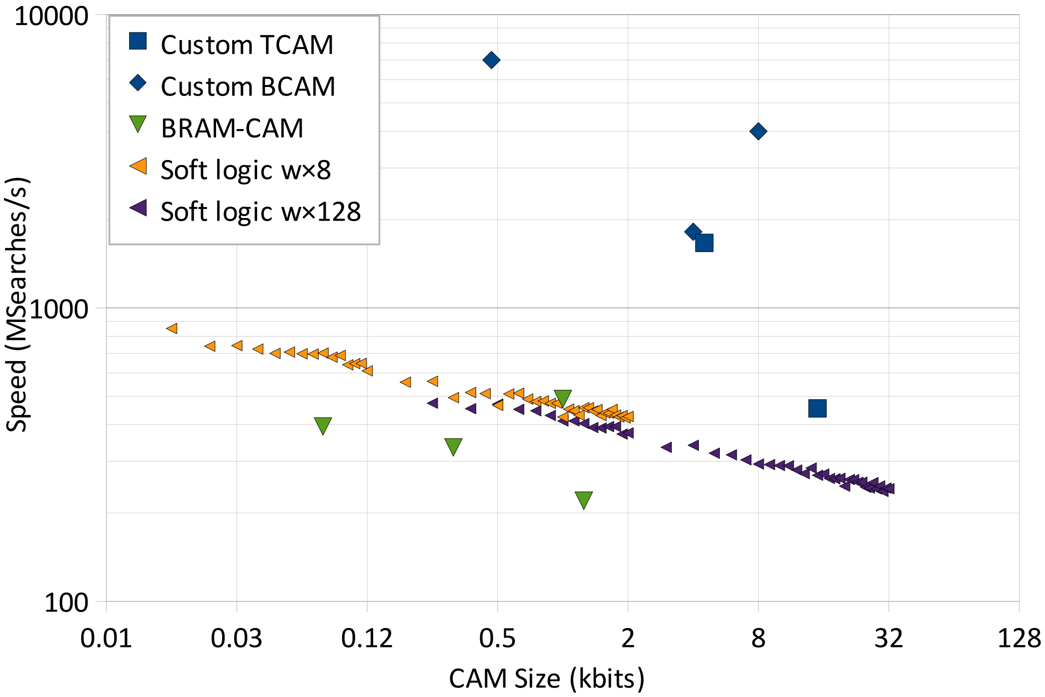


Figure 3.3: CAM search speed

blocks. Latency is the product of the cycle time and the number of pipeline stages, and does not adjust for unbalanced pipeline stages or pipeline latch overheads. Table 3.7 shows details of the design examples.

The two IBM multipliers have latency ratios comparable to full processor cores. Intel's 16-bit multiplier design has much lower latency ratios as it appears to target low power instead of delay. In designs where multiplier throughput is more important than latency, multipliers can be made more deeply pipelined (3 and 4 stages in these examples) than the hard multipliers on FPGAs (2 stages), and throughput ratios can be even higher than the latency ratios.

The area of the custom CMOS and FPGA multipliers are plotted in Figure 3.6. FPGA multipliers are relatively area-efficient. The area ratios for multipliers of 4.5–7.0× are much lower than for full processor cores (17–27×, Section 3.4.1).

Design	Size	Stages	Latency (ns)	Area (mm^2)	Ratios Latency	Ratios Area
Intel 90 nm 1.3 V [80]	16×16	1	0.81	0.014	3.4	4.7
IBM 90 nm SOI 1.4 V [81]	54×54	4	0.41	0.062	22	7.0
IBM 90 nm SOI 1.3 V [82]	53×53	3	0.51	0.095	17	4.5
Stratix III	16×16	1	2.8	0.066		
Stratix III	54×54	1	8.8	0.43		

Table 3.7: Multiplier area and delay, normalized to 65 nm process. Unpipelined latency is pipelined cycle time×stages.

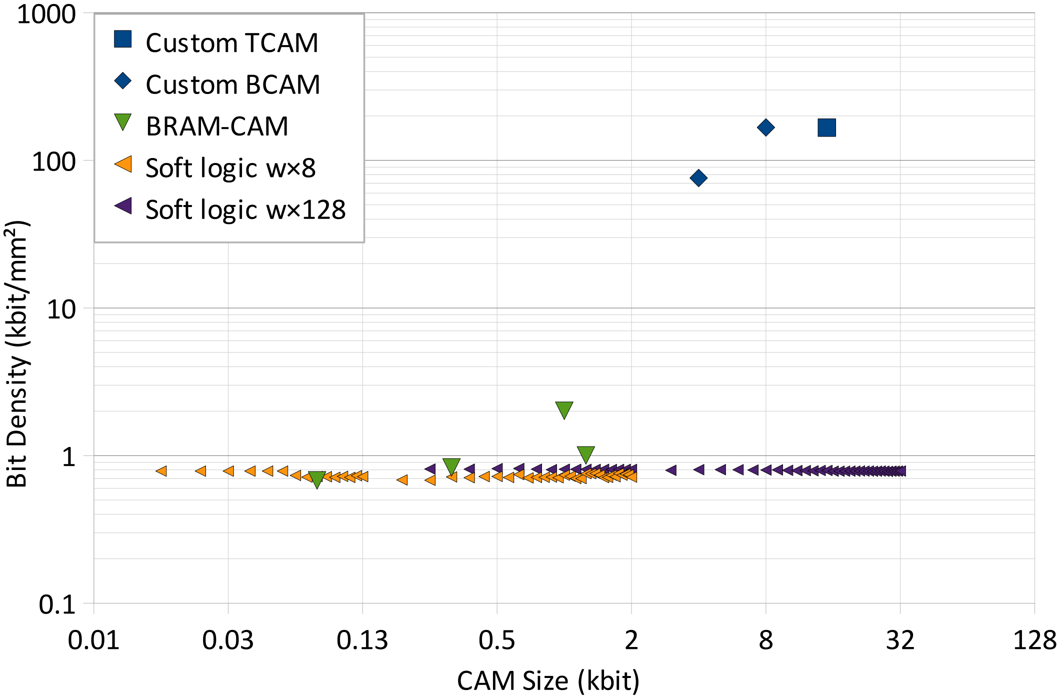


Figure 3.4: CAM bit density

3.4.6 Adders

Custom CMOS adder circuit designs can span the area-delay trade-off space from slow ripple-carry adders to logarithmic-depth fast adders. On an FPGA, adders are usually implemented using hard carry chains that implement variations of the ripple-carry adder, although carry-select adders have been also been used. Although fast adders can be implemented on FPGAs with soft logic and routing, the lack of dedicated circuitry means fast adders are bigger and usually slower than the ripple-carry adder with hard carry chains [88].

Figure 3.7 plots a comparison of adder delay, with details in Table 3.8. The Pentium 4 delay is conservative as the delay given is for the full integer ALU. FPGA adders achieve delay ratios of 15–20× and

Design	Size (bit)	f_{max} (MHz)	Area (mm^2)	Delay Ratio	Area Ratio
Agah 1.3 V [84]	32	12000	–	20	–
Kao 90 nm 1.3 V [85]	64	7100	0.016	19	4.5
Pentium 4 1.3 V [86]	32	9000	–	16	–
IBM 1.0 V [87]	108	3700	0.017	15	6.9
Stratix III	32	593	0.035		
	64	374	0.071		
	108	242	0.119		

Table 3.8: Adder area and delay, normalized to 65 nm process

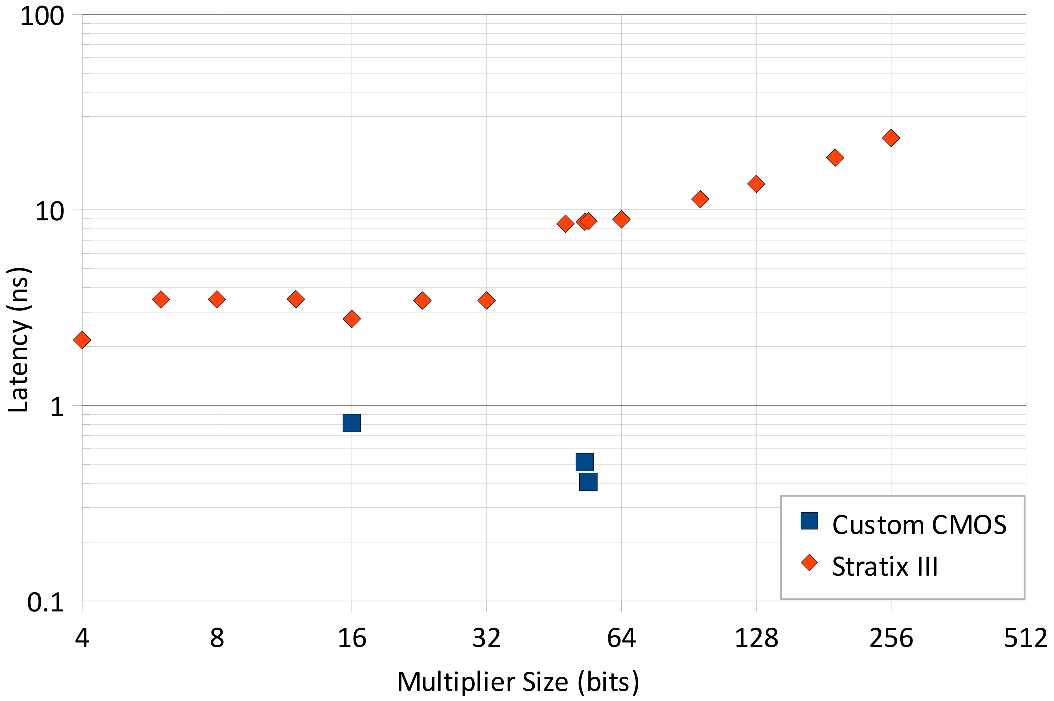


Figure 3.5: Multiplier latency

a low area ratio of around 4.5–7×. Despite the use of dedicated carry chains on the FPGA, the delay ratios are fairly high because we compare FPGA adders to high-performance custom CMOS adders. For high-performance applications, such as in processors, FPGAs offer little flexibility in trading area for even more performance by using a faster circuit-level design.

3.4.7 Multiplexers

Multiplexers are found in many circuits, yet we have found little literature that provides their area and delay in custom CMOS. Instead, we estimate delays of small multiplexers using a resistor-capacitor (RC) analytical model, the delays of the Pentium 4 shifter unit, and the delays of the Stratix III ALM. Our area

Mux	FPGA		Custom CMOS	
Inputs	Area (mm^2)	Delay (ps)	Delay (ps)	Delay Ratio
2	0.0011	210	2.8	74
4	0.0011	260	4.9	53
8	0.0022	500	9.1	54
16	0.0055	680	18	37
32	0.0100	940	29	32
64	0.0232	1200	54	21

Table 3.9: Analytical model of transmission gate or pass transistor tree multiplexers [89] normalized to 65 nm process.

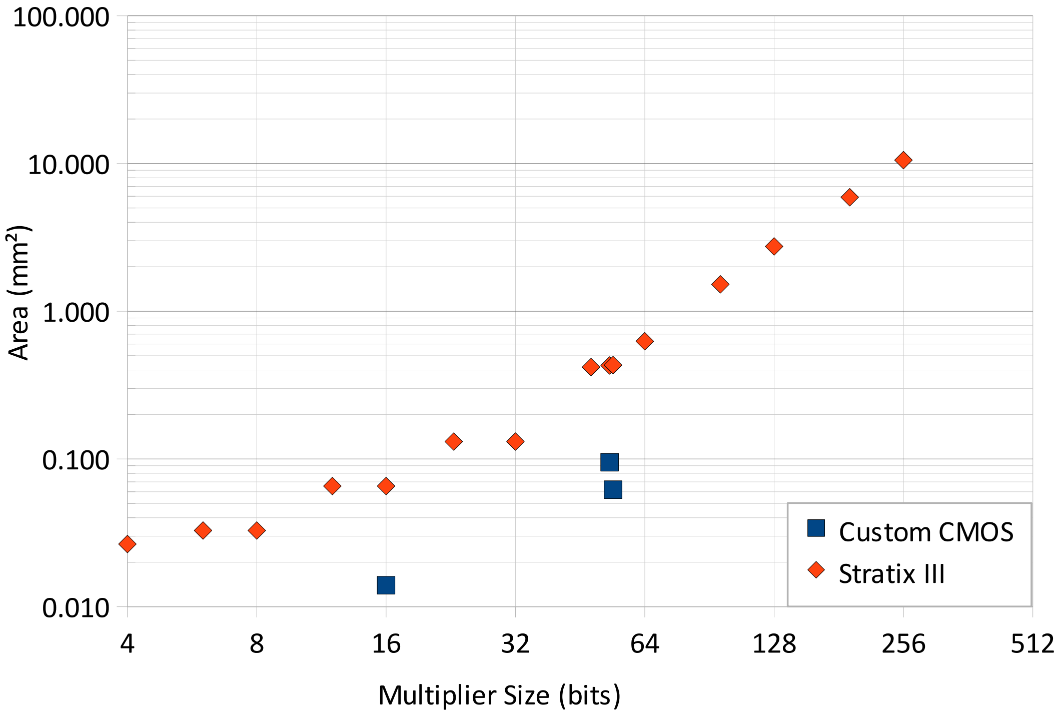


Figure 3.6: Multiplier area

Circuit	FPGA Delay (ps)	Custom CMOS Delay (ps)	Ratio
65 nm Pentium 4 Shifter	2260	111	20
Stratix III ALM			
Long path	2500	350	7.1
Short path	800	68	11.7

Table 3.10: Delay of multiplexer-dominated circuits

ratio estimate comes from an indirect measurement using an ALM.

Table 3.9 shows a delay comparison between an FPGA and an analytical model of transmission gate or pass gate tree multiplexers [89]. This unbuffered switch model is pessimistic for larger multiplexers, as active buffer elements can reduce delay. On an FPGA, small multiplexers can often be combined with other logic with minimal extra delay and area, so multiplexers measured in isolation are likely pessimistic. For small multiplexers, the delay ratio is high, roughly 40–75×. Larger multiplexers appear to have decreasing delay ratios, but we believe this is largely due to the unsuitability of the unbuffered designs to which we are comparing.

An estimate of the multiplexer delay ratio can also be made by comparing the delay of larger circuits that are composed mainly of multiplexers. The 65 nm Pentium 4 integer shifter datapath [86] is one such circuit, containing small multiplexers (sizes 3, 4, and 8). We implemented the same datapath excluding control logic on the Stratix III. A comparison of the critical path delay is shown in Table 3.10. The

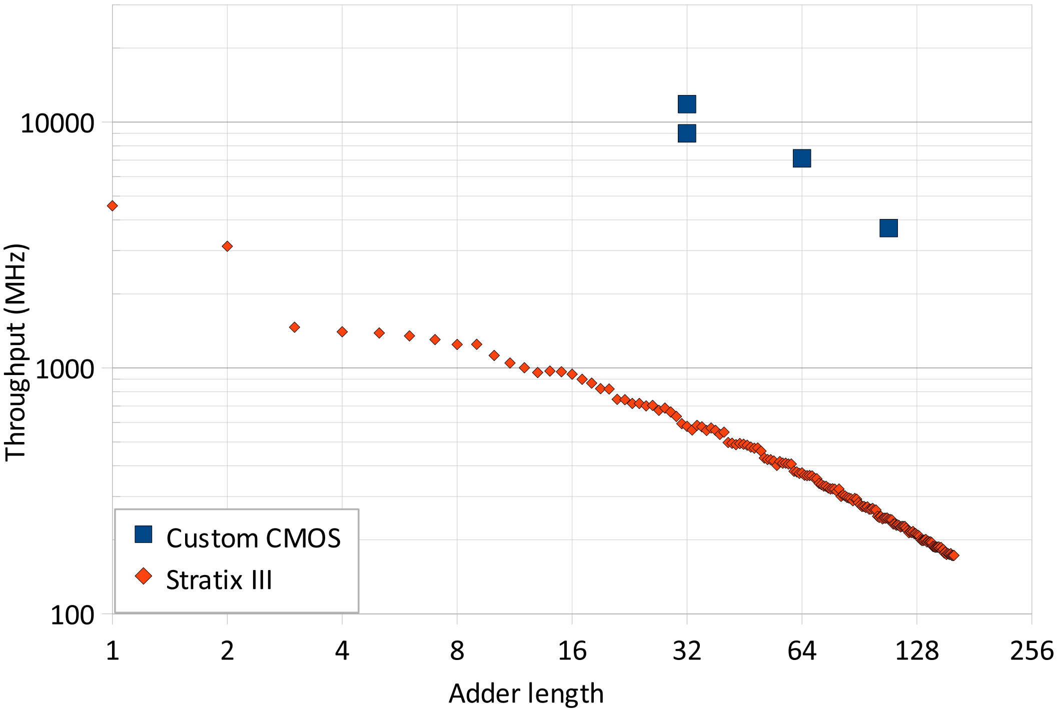


Figure 3.7: Adder delay

delay ratio of 20× is smaller than suggested by the isolated multiplexer comparison, but may be optimistic if Intel omitted details from their shifter circuit diagram causing our FPGA equivalent shifter to be oversimplified.

Another delay ratio estimate can be made by examining the Stratix III Adaptive Logic Module (ALM) itself, as its delay consists mainly of multiplexers. We implemented a circuit equivalent to an ALM as described in the Stratix III Handbook [90], comparing delays of the FPGA implementation to custom CMOS delays of the ALM given by the Quartus timing models. Internal LUTs are modelled as multiplexers that select between static configuration RAM bits. Each ALM input pin is modelled as a 21-to-1 multiplexer, as 21 to 30 are reasonable sizes according to Lewis et al. [49].

We examined one long path and one short path, from after the input multiplexers for pins datab and dataf0, respectively, terminating at the LUT register. Table 3.10 shows delay ratios of 7.1× and 11.7× for the long and short paths, respectively. These delay ratios are lower compared to previous examples due to the lower power and area budgets preventing custom FPGAs from being as aggressively delay-optimized as custom processors, and to extra circuit complexity not shown in the Stratix III Handbook.

We can also estimate a lower bound on the multiplexer area ratio by implementing only the multiplexers in our FPGA equivalent circuit of an ALM, knowing the original ALM contains more functionality than our equivalent circuit. Our equivalent ALM consumes 104 ALUTs, or roughly 52 ALMs, resulting in an estimated area ratio of 52×. However, the real ALM area ratio is substantially greater, as we implemented only the ALM's input and internal multiplexers and did not include global routing resources or configuration RAM. A rule of thumb is that half of an FPGA's core area is spent in the programmable

Design	Register Delay (ps)	Delay Ratio
Pentium 4 180 nm [91]	35 (90 ps in 180 nm)	12
Hartstein et al. [92]	32 (2.5 FO4)	14
Hrishikesh et al. [93]	23 (1.8 FO4)	19
Geometric Mean	29.5	15
Stratix III	436	–

Table 3.11: Pipeline latch delay

global routing network, doubling the area ratio estimate to 104× while still neglecting the configuration RAM.

In summary, groups of multiplexers (measured from the Pentium 4 shifter and ALM) have delay ratios below 20×, with small isolated multiplexers being worse (40–75×). However, multiplexers are particularly area-intensive with an area ratio greater than 100×. Thus we find that the intuition that multiplexers are expensive on FPGAs is justified, especially from an area perspective.

3.4.8 Pipeline Latches

In synchronous circuits, the maximum clock speed of a circuit is typically limited by a register-to-register delay path from a pipeline latch[2], through a pipeline stage's combinational logic, to the next set of pipeline latches. The delay of a pipeline latch (its setup and clock-to-output times) impacts the speed of a circuit and the clock speed improvement when increasing pipeline depth. Note that hold times do not directly impact the speed of a circuit, only correctness.

The "effective" cost in delay of inserting an extra pipeline register into LUT-based combinational pipeline logic is measured by observing the increase in delay as the number of LUTs between registers increases, then extrapolating the delay to zero LUTs. This method is different from, and more pessimistic than, simply summing the T_{co} (clock to output), T_{su} (setup), clock skew, and one extra LUT-to-register interconnect delay to reach a register, which is 260ps. This pessimism occurs because inserting a register also impacts the delay of the combinational portion of the delay path. The measured latch delay in Stratix III is 436 ps.

Table 3.11 shows estimates of the delay of a custom CMOS pipeline latch. The 180 nm Pentium 4 design assumed 90 ps of pipeline latch delays including clock skew [91], which we scaled according to the FO1 ring oscillator delays for Intel's processes (11 ps at 180 nm to 4.25 ps at 65 nm) [67]. Hartstein et al. and Hrishikesh et al. present estimates expressed in fanout-of-four (FO4) delays, which were scaled to an estimated FO4 delay of 12.8 ps for Intel's 65 nm process.

Thus, the delay ratio for a pipeline latch ranges from 10 to 15 times. Although we do not have area comparisons, registers are considered to occupy very little FPGA area because more LUTs are used than registers in most FPGA circuits, yet FPGA logic elements include at least one register for every LUT.

[2] Latch refers to pipeline storage elements. This can be a latch, flip-flop, or other implementation.

3.4.9 Interconnect Delays

Interconnect delay comprises a significant portion of the total delay in both FPGAs and modern CMOS processes. In this section we explore the point-to-point delay of these technologies, and include the effect of congestion on these results.

Point-to-Point Routing

In this section, we measure the wire delay of a point-to-point (single fanout) connection. In modern CMOS processes, there are multiple layers of interconnect wires, for dense local connections and faster global connections. On an FPGA, an automated router chooses a combination of faster long wires or more abundant short wires when making a routing connection.

For custom CMOS, we approximate the delay of a buffered wire using a lumped-capacitance model with interconnect and transistor parameters from the International Technology Roadmap for Semiconductors (ITRS) 2007 report [94]. The ITRS 2007 data could be pessimistic when applied to high-performance CMOS processes used in processors, as Intel's 65 nm process uses larger pitch and wire thicknesses than the ITRS parameters, and thus reports lower wire delays [67]. On the Stratix III FPGA, point-to-point delay is measured using the delay between two manually-placed registers with automated routing, with the delay of the register itself subtracted out. We assume that LABs on the Stratix III FPGA have an aspect ratio (the vertical/horizontal ratio of delay for each LAB) of 1.6 because it gives a good delay vs. manhattan distance fit.

Figure 3.8 plots the point-to-point wire delays for custom CMOS and FPGA wires versus the length of the wire. The delay for short wires (under 20 µm) is dominated by the delay of the driver and load buffers (i.e., one FO1 delay). These delays may be optimistic for global wires because we do not include the delay of the vias required to access the top layers of wiring. The FPGA point-to-point wire delays are plotted as "Stratix III". FPGA short local wires (100 µm) have a delay ratio around 9× compared to "local" wires of the same length. Long wire delay (above 10 000 µm) is quite close (2×) to CMOS for the same length of wire.

When trying to measure the impact of wire delays on a circuit, routing delays are more meaningful when "distance" is normalized to the amount of "logic" that can be reached. To approximate logic density-normalized routing delays, we adjust the FPGA routing distance by the square-root of the FPGA's overall area overhead vs. custom CMOS ($\sqrt{23\times} = 4.8\times$). That is, a circuit implemented on an FPGA will need to use wires that are 4.8 times longer than the equivalent circuit implemented in custom CMOS.

The logic density-normalized routing delays are plotted as "Stratix III Area Adjusted" in Figure 3.8. Short local FPGA wires (100 µm) have a logic density-normalized delay ratio of 20×, while long global wires (7 500 µm) have a delay ratio of only 9×. The short wire delay ratio is comparable to the overall delay ratio for full processors, but the long wire delay ratio is half that, suggesting that FPGAs are less affected by long wire delays than custom CMOS.

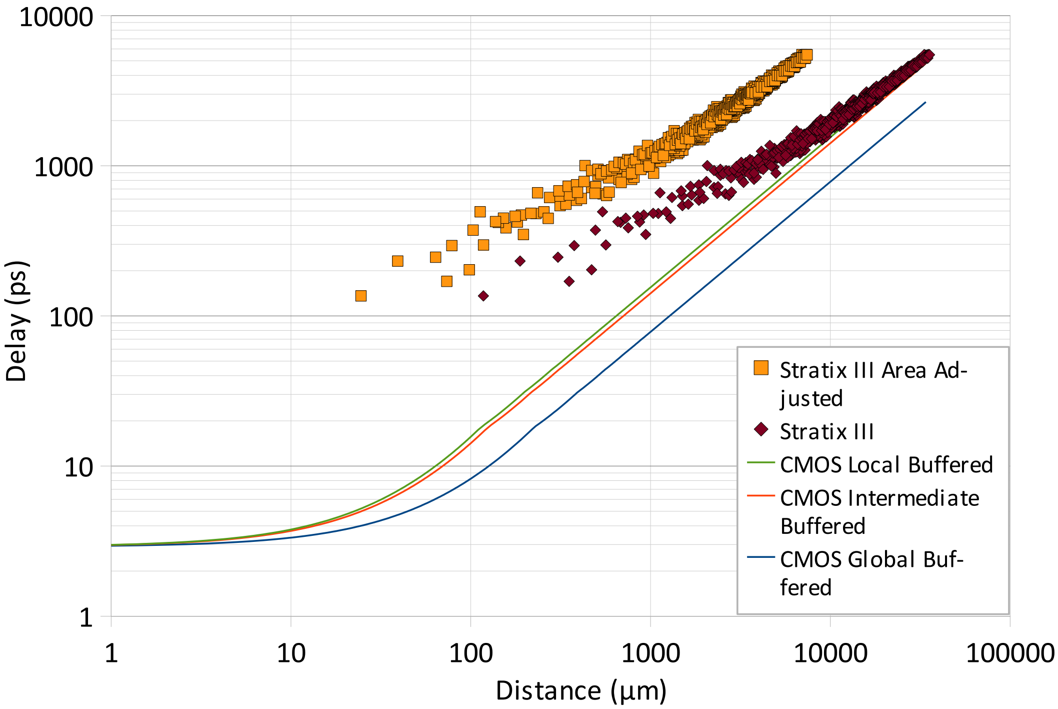


Figure 3.8: Point-to-point routing delay

FPGA Routing Congestion

The preceding section compared FPGA vs. custom CMOS point-to-point routing delays in an uncongested chip. These delays could be optimistic compared to routing delays in real circuits where congestion causes routes to take sub-optimal paths. This section shows how much FPGA routing delay changes from the ideal point-to-point delays due to congestion found in real FPGA designs.

To measure the impact of congestion, we compare the delay of route connections found on near-critical paths in a soft processor to the delay of routes travelling the same distance on an empty FPGA. We synthesized two soft processors for this measurement: The OpenSPARC T1, a large soft processor, and the Nios II/f, a small soft processor specifically designed for FPGA implementation. We extracted register-to-register timing paths that had delay greater than 90% of the critical path delay (i.e., the top 10% of near-critical paths). Timing paths are made up of one or more connections, where each connection is a block driving a net (routing wires) and terminating at another block's input. For each connection in the top 10% of paths, we observed its delay as reported by the Quartus timing analyzer and its manhattan distance calculated by placement locations of the source and destination blocks.

The resulting delay vs. distance plots are shown in Figure 3.9a for the OpenSPARC T1 and Figure 3.9b for the Nios II/f. The empty-chip measurements are the same as those from the preceding section (Figure 3.8). The larger size of the OpenSPARC T1 results in many longer-distance connections, while the longest connection within the top 10% of paths in the small Nios II/f has a distance of 1800 µm or about the width of 15 LAB columns. We see from these plots that the amount of congestion found in typical soft processors does not appreciably impact the routing delays for near-critical routes, and that routing

	Custom CMOS	FPGA [95]	Ratio
DDR2 Frequency (MHz)	533	400	1.3
DDR3 Frequency (MHz)	800	533	1.5
Read Latency (ns)	55–65	85	1.4

Table 3.12: Off-chip DRAM latency and throughput. Latency assumes closed-page random accesses.

Design	Delay Ratio	Area Ratio
Processor Cores	18 – 26	17 – 27
SRAM 1rw	7 – 10	2 – 5
SRAM 4r2w LUTs / LVT	15 / 9	179 / 12
CAM	14	100 – 210
Multiplier	17 – 22	4.5 – 7.0
Adder	15 – 20	4.5 – 7.0
Multiplexer	20 – 75	>100
Pipeline latch	12 – 19	—
Routing	9 – 20	—
Off-Chip Memory	1.3 – 1.5	—

Table 3.13: Delay and area ratio summary

congestion does not alter our conclusions in the preceding section that FPGA long wire routing delays are relatively low.

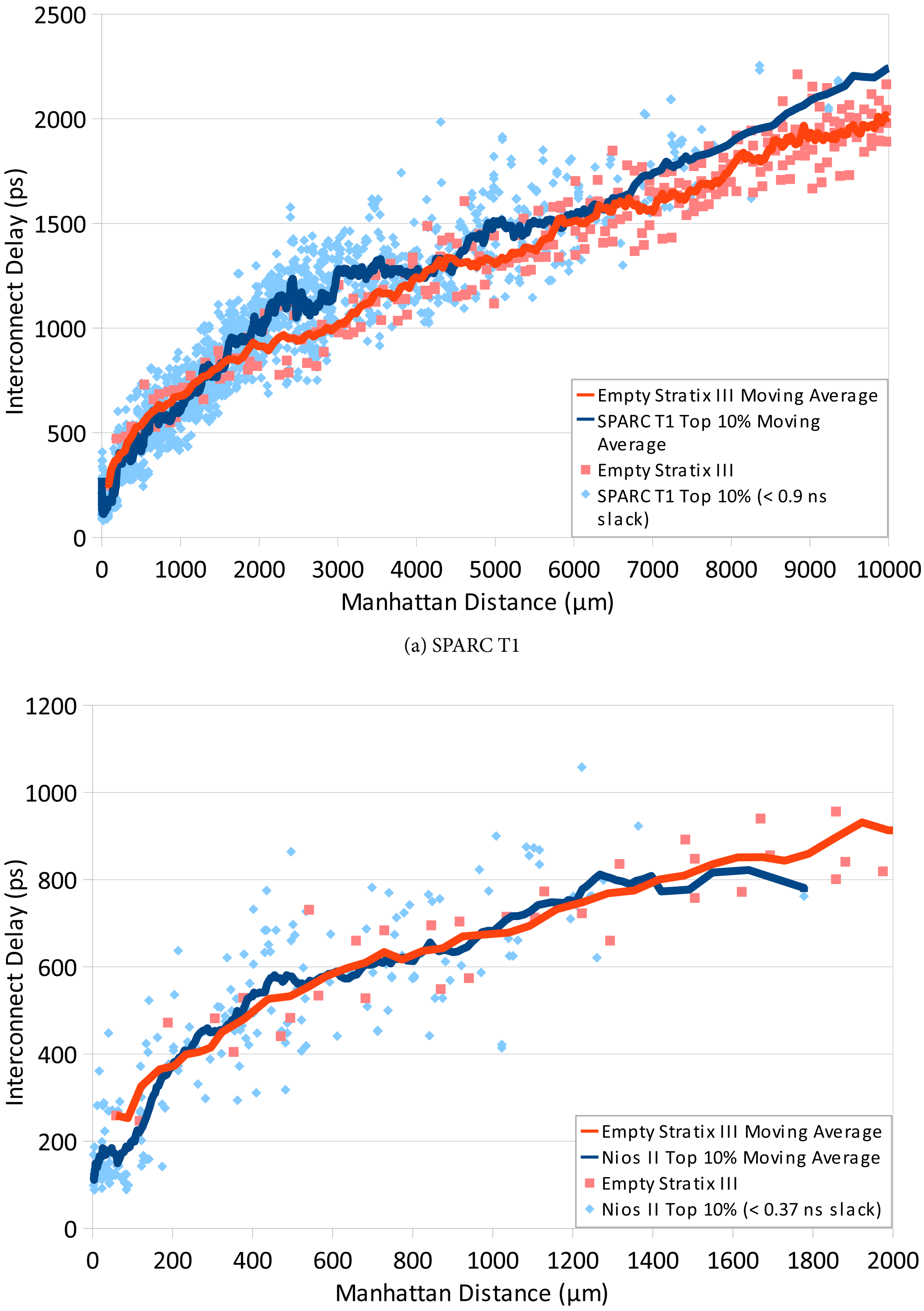
3.4.10 Off-Chip Large-Scale Memory

Table 3.12 gives a brief overview of off-chip DRAM latency and bandwidth as commonly used in processor systems. Random read latency is measured on Intel DDR2 and DDR3 systems with off-chip (65 ns) and on-die (55 ns) memory controllers. FPGA memory latency is calculated as the sum of the memory controller latency and closed-page DRAM access time [95]. While these estimates do not account for real access patterns, they are enough to show that off-chip latency and throughput ratios between custom CMOS and FPGA are far lower than for any of the in-core circuits discussed above.

3.4.11 Summary of Building Block Circuits

A summary of our estimates for the FPGA vs. custom CMOS delay and area ratios is given in Table 3.13. Note that the range of delay ratios (from 7–75×) is smaller than the range of area ratios (from 2–210×). The multiplexer circuit has the highest delay ratios. Hard blocks used to support specific circuit types have only a small impact on delay ratios, but they considerably impact the area-efficiency of SRAM, adders, and multiplier circuits. Multiplexers and CAMs are particularly area-inefficient.

Previous work [52] reported an average of 3.0–3.5× delay ratio and 18–35× area ratio for FPGA vs. standard cell ASIC for a set of complete circuits. Although we expect both ratios to be higher when comparing FPGA against custom CMOS, our processor core delay ratios are higher but area ratios are slightly



(a) SPARC T1

(b) Nios II/f

Figure 3.9: Comparing interconnect delays between an empty FPGA and soft processors

lower, which is initially surprising. We believe this is likely due to custom processors being optimized more for delay at the expense of area compared to typical standard cell circuits.

3.5 Impact on Processor Microarchitecture

Section 3.4 measured the area and delay differences between different circuit types targeting both custom CMOS and FPGAs. In this section we relate those differences to the microarchitectural design of circuits in the two technologies. It is important to note that *area* is often a primary concern in the FPGA space, given the high area cost of programmability, leading to lower logic densities and high relative costs of the devices. In addition, the results above show that the area ratios between different circuit types vary over a larger range (2–200×) than the delay ratios (7–75×). For both of these reasons, we expect that area considerations will have a stronger impact on microarchitecture than delay.

The building blocks we measured cover many of the circuit structures used in microprocessors:

- SRAMs are very common, but take on different forms. Caches are usually low port count and high density SRAMs. Register files use high port count, require higher speed, and are lower total capacity. RAM structures are also found in various predictors (branch direction and target, memory load dependence), and in various buffers and queues used in out-of-order microarchitectures (reorder buffer, register rename table, register free lists)

- CAMs can be found in high-associativity caches and TLBs. In out-of-order processors, CAMs can also be used for register renaming, memory store queue address matching, and instruction scheduling (in reservation stations). Most of these can be replaced by RAMs, although store queues and instruction scheduling are usually CAM-based.

- Multipliers are typically found only in ALUs (both integer and floating-point).

- Adders are also found in ALUs. Addition is also used for address generation (AGUs), and in miscellaneous places such as the branch target address computation.

- Small multiplexers are commonly scattered within random logic in a processor. Larger, wider multiplexers can be found in the bypass networks near the ALUs.

- Pipeline latches and registers delimit the pipeline stages (which are used to reduce the cycle time) in pipelined processors.

We begin with general suggestions applicable to all processors, then discuss issues specific to out-of-order processors. Our focus on out-of-order processors is driven by the desire to improve soft processor performance given the increasing logic capacity of new generations of FPGAs, while also preserving the ease of programmability of the familiar single-threaded programming model.

3.5.1 Pipeline Depth

Pipeline depth is one of the fundamental choices in the design of a processor microarchitecture. Increasing pipeline depth results in higher clock speeds, but with diminishing returns due to pipeline latch delays. Hartstein and Puzak [92] show that the optimal processor pipeline depth for performance is proportional to $\sqrt{\frac{t_p}{t_o}}$, where t_p is the total logic delay of the processor pipeline, and t_o is the delay overhead of a pipeline latch. Other properties of a processor design, such as branch prediction accuracy, the presence of out-of-order execution, or issue width, also affect the optimal pipeline depth, but these properties depend on *microarchitecture*, not *implementation technology*. The *implementation technology*-dependent parameters t_o and t_p have a similar effect on the optimal pipeline depth for different processor microarchitectures, and these are the only two parameters that change when comparing implementations of the same microarchitecture on two different implementation technologies (custom CMOS vs. FPGA).

Section 3.4.8 showed that the delay ratio of registers (which is the t_o of the FPGA vs. the t_o custom CMOS, measured as $\sim 15\times$) is lower than the delay ratio of a complete processor (which is roughly[3] the t_p of the processor on the FPGA vs. the t_p of a custom CMOS processor, $\sim 22\times$), increasing t_p/t_o on FPGA. The change in t_p/t_o is roughly (22/15), suggesting soft processors should have pipeline depths roughly 20% longer compared to an equivalent microarchitecture implemented in custom CMOS. In addition to performance, pipeline registers are nearly free in area in many FPGA designs because most designs consume more logic cells (LUTs) than registers, which reduces the cost of registers for deeper pipelines in soft processors.

That soft processors should have slightly deeper pipelines than an equivalent hard processor seems opposed to the observation that today's soft processors tend to use short pipelines [96]. Current (relatively simple) soft processors processors have short pipelines because they have low complexity (low t_p), which is a property of the processor microarchitecture, and not a property of the FPGA substrate.

3.5.2 Interconnect Delay and Partitioning of Structures

The portion of a chip that can be reached in a single clock cycle is decreasing with each newer process generation, while transistor switching speeds continue to improve. This leads to microarchitectures that partition large structures into smaller ones. This could be dividing the design into clusters (such as grouping a register file with ALUs into a cluster and requiring extra latency to communicate between clusters) or employing multiple cores to avoid global, one-cycle, communication [55].

In Section 3.4.9, we observed that after adjustment for the reduced logic density of FPGAs, long wires have a delay ratio roughly half that of a full processor core. The relatively faster long wires lessen the impact of global communication, reducing the need for aggressive partitioning of designs for FPGAs. Current FPGA processors have less logic complexity than high-performance custom processors, so there

[3] The value of t_p is the total propagation delay of a processor with the pipeline latches removed, and is not easily measured. It can be approximated by the product of the number of pipeline stages (N) and cycle time if we assume perfectly balanced stages. The cycle time includes both logic delay (t_p/N) and latch overhead (t_o) components for each pipeline stage, but since we know the custom CMOS vs. FPGA t_o ratio is smaller than the cycle time ratio, using the cycle time ratio as an estimate of the t_p ratio results in a slight underestimate of the t_p ratio.

is little need to partition. As FPGA processors grow in complexity, such as for the out-of-order processor in this work, the lower impact of long wires on FPGAs suggests that FPGA processors would have less difficulty with long-distance interconnect delay than hard processors.

3.5.3 ALUs and Bypassing

Multiplexers consume much more area (>100×) on FPGAs than custom CMOS (Section 3.4.7), making bypass networks that shuffle operands between functional units more expensive on FPGAs. On the other hand, the functional units themselves are often composed of adders and multipliers and have a lower 4.5–7× area ratio. The high cost of multiplexers reduces the area benefit of using multiplexers to share these functional units.

There are processor microarchitecture techniques that reduce the size of operand-shuffling networks relative to the number of ALUs. "Fused" ALUs that perform two or more dependent operations at a time increase the amount of computation relative to operand shuffling, such as the common fused multiply-accumulate unit and interlock collapsing ALUs [97, 98]. Other proposals cluster instructions together to reduce the communication of operand values to instructions outside the group [99, 100]. These techniques may benefit soft processors more than hard processors. However, we did not consider fused operations in our processor design due to the extra complexity (e.g., in instruction decoding) required to find suitable operations to fuse.

3.5.4 Cache Organization

Set-associative caches have two common implementation styles. Low associativity caches replicate the cache tag RAM and access them in parallel, while high associativity caches store tags in CAMs. High associativity caches are more expensive on FPGAs because of the high area cost of CAMs (100–210× bit density ratio). In addition, custom CMOS caches built from tag CAM and data RAM blocks can have the CAM's decoded match lines directly drive the RAM's word lines, while an FPGA CAM must produce encoded outputs that are then decoded by the SRAM, adding a redundant encode-decode operation that was not included in the FPGA circuits in Section 3.4.4 (we assumed CAMs with decoded outputs). In comparison, custom CMOS CAMs have minimal delay and 2–3× area overhead compared to RAM allowing for high-associativity caches (with a CAM tag array and RAM data array) to have an amortized area overhead of around 10%, with minimal change in delay compared to lower-associativity set-associative caches [101].

CAM-based high-associativity caches are not area efficient in FPGA soft processors and hence soft processor caches should have lower associativity than similar hard processors. Soft processor caches should also be of higher capacity than those of similar hard processors because of the good area efficiency of FPGA SRAMs (2–5× density ratio).

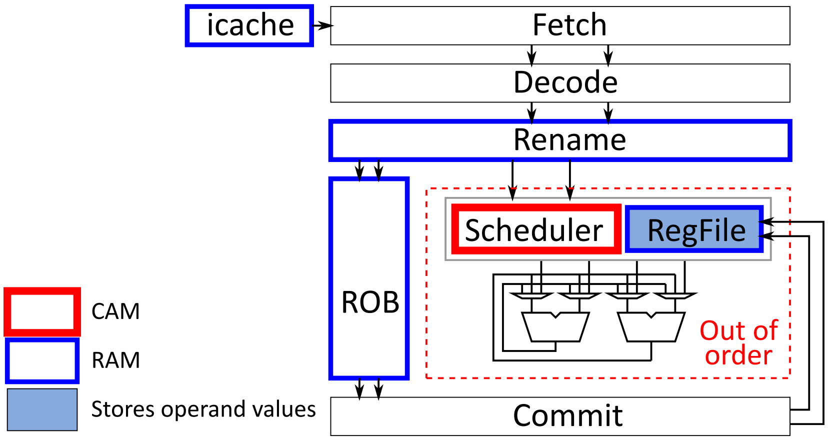


Figure 3.10: A typical out-of-order processor microarchitecture

3.5.5 Memory System Design

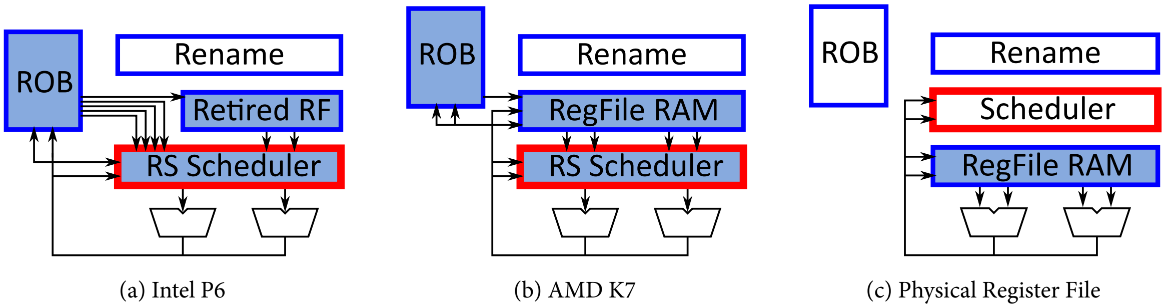
The lower area cost of block RAM encourages the use of larger caches, reducing cache miss rates and lowering the demand for off-chip DRAM bandwidth. The lower clock speeds of FPGA circuits further reduce off-chip bandwidth demand. The latency and bandwidth of off-chip memory is only slightly worse on FPGAs than on custom CMOS processors as they use essentially the same commodity DRAMs.

Hard processors use many techniques to improve memory system performance, such as DRAM access scheduling, non-blocking caches, prefetching, memory dependence speculation, and out of order memory accesses. The lower off-chip memory system demands on FPGA soft processors suggest that more resources should be dedicated to improving the performance of the processor core than to improving memory bandwidth or tolerating latency.

3.5.6 Out-of-Order Microarchitecture

Superscalar out-of-order processors are more complex than single-issue in-order processors. The larger number of instructions and operands in flight increase multiplexer and CAM use, leading to the common expectation that out-of-order processors would be disproportionately expensive on FPGAs and therefore not a suitable choice for use in soft processors. However, section 3.4.1 suggests that processor complexity does not have a strong correlation with FPGA vs. custom CMOS area ratio: even when not specifically FPGA-optimized, the multiple-issue out-of-order Nehalem processor has an area ratio similar to the three in-order designs, suggesting that out-of-order and in-order processor designs appear equally suited for FPGA implementation. One possible explanation is that, for issue widths found in current processors, most of the area in a complex out-of-order processor is not spent on the CAM-like schedulers and multiplexer-like bypass networks, even though these structures are often high power, timing critical, and scale poorly to very wide issue widths. The small size of the CAMs and multiplexers mean that even particularly high area ratios for CAMs and multiplexers cause only a small impact to the area of the whole processor core.

Figure 3.10 shows the high-level organization of a typical out-of-order processor. Fetch, decode,



(a) Intel P6 (b) AMD K7 (c) Physical Register File

Figure 3.11: Out-of-order processor microarchitecture variants

register rename, and instruction commit are done in program order. The reorder buffer (ROB) tracks instructions as they progress through the out-of-order section of the processor. Out-of-order execution usually includes a CAM-based instruction scheduler, a register file, some execution units (ALUs), and bypass networks. The memory load/store units and the memory hierarchy are not shown in this diagram.

There are several styles of microarchitectures commonly used to implement precise interrupt support in pipelined or out-of-order processors and many variations are used in modern processors [14,15,102]. The main variations between the microarchitecture styles concern the organization of the reorder buffer, register renaming logic, register file, and instruction scheduler and whether each component uses a RAM- or CAM-based implementation. Some common organizations used in recent out-of-order processors are shown in Figure 3.11. These organizations have important implications on the RAM and CAM size and port counts used by a processor.

The Intel P6-derived microarchitectures (from Pentium Pro to Nehalem) use reservation stations and a separate committed register file (Figure 3.11a) [103]. Operand values are stored in one of three places: retired register file, reorder buffer, or reservation stations. The retired register file stores register values that are already committed. The reorder buffer stores register values that are produced by completed, but not committed, instructions. When an instruction is dispatched, it reads any operands that are available from the retired register file (already committed) or reorder buffer (not committed), stores the values in the reservation station entry, and waits until the remaining operand values become available on the bypass networks. When an instruction commits, its result value is copied from the reorder buffer into the retired register file. This organization requires several multiported RAM structures (reorder buffer and retired register file) and a scheduler CAM that stores operand values (any number of waiting instructions may capture a previous instruction's result).

The organization used in the AMD K7 and derivatives (K7 through K10) unifies the speculative (future file) and retired register files into a single multiported RAM structure (labelled "RegFile RAM" in Figure 3.11b [104]). Like the P6, register values are stored in three places: reorder buffer, register file, and reservation stations. Unlike the P6, dispatching instructions only need to read the future file RAM but not from the reorder buffer. However, result values are still written into the reorder buffer, and, like the P6, are copied into the register file when an instruction commits. Using a combined future file and register file reduces the number of read ports required for the ROB (the ROB is read only for committing results), but increases the number of read ports for the register file. Like the P6, the K7 uses a reservation station scheduler that stores operand values in a CAM. For FPGA implementations, the K7 organization seems to be a slight improvement over the P6 because only the register file is highly multiported (the ROB only needs multiple write ports and sequential read for commit), and the total number of RAM ports is reduced slightly.

The physical register file organization (PRF, Figure 3.11c) has been used in many hard processor designs, such as the MIPS R10K, IBM Power 4, 5, 7, and 8, Intel Pentium 4 and Sandy Bridge, Alpha 21264, and AMD Bobcat and Bulldozer [105–112]. In a physical register file organization, operand values are stored in one central register file. Both speculative and committed register values are stored in the same structure. The register renamer explicitly renames architectural register numbers into indices into the physical register file, and must be able to track which physical registers are in use and roll back register mappings during a pipeline flush. After an instruction is dispatched into the scheduler, it waits until all of its operands are available. Once the instruction is chosen to be issued, it reads all of its operands from the physical register file RAM or bypass networks, normally taking one extra cycle compared to the P6 and K7. The instruction's result is written back into the register file RAM and bypass networks. When an instruction commits, only the state of the register renamer needs to be updated, and there is no copying of register values as in the previous two organizations.

The physical register file organization has several advantages that are particularly significant for FPGA implementations. Register values are only stored in one structure (the PRF), reducing the number of multiported structures required. Also, the scheduler's CAM does not store operand values, allowing the area-inefficient CAM to be smaller, with operand values stored in a more area-efficient register file RAM. This organization adds some complexity to track free physical registers and an extra pipeline stage to access the PRF. FPGA RAMs have particularly low area cost (Section 3.4.2), but CAMs are area expensive (Section 3.4.4). The benefits of reducing CAM size and multiported RAMs suggest that the PRF organization would be particularly preferred for FPGA implementations.

The delay ratio of CAMs (15×) is not particularly poor, so CAM-based schedulers are reasonable on FPGA soft processors. However, the high area cost of FPGA CAMs means scheduler capacity should be kept small. In addition to reducing the number of scheduler entries, reducing scheduler area can also be done by reducing the amount of storage required per entry. One method is to choose an organization that does not store operand values in the CAM, like the PRF organization (Figure 3.11c). Schedulers can be data-capturing where operand values are captured and stored in the scheduler, or non data-capturing where the scheduler tracks only the availability of operands, with values fetched from the register file or bypass networks when an instruction is finally issued. Non data-capturing schedulers reduce the amount of data that must be stored in each entry of a scheduler.

The processor organizations described above all use a CAM for instruction scheduling. It may be possible to further reduce the area cost by removing the CAM. There are CAM-free instruction scheduler techniques that are not widely implemented [37,54], but may become more favourable in soft processors. Reorder buffers, register renaming logic, and register files have occasionally been built using CAMs in earlier processors, but are commonly implemented without CAMs.

On FPGAs, block RAMs come in a limited selection of sizes, with the smallest block RAMs commonly being 4.5 kbit to 20 kbit. Reorder buffers and register files are usually even smaller in capacity but are limited by port width or port count so processors on FPGAs can have larger capacity ROBs, register files, and other port-limited RAM structures at little extra cost. In contrast, expensive CAMs limit soft processors to small scheduling windows (instruction scheduler size). Microarchitectures that address this particular problem of large instruction windows with small scheduling windows may be useful in soft processors [113].

3.6 Conclusion

We have presented area and delay comparisons of processors and their building block circuits implemented on custom CMOS and FPGA substrates. In 65 nm processes, we found FPGA implementations of processor cores have 18–26× greater delay and 17–27× greater area usage than the same processors in custom CMOS. The FPGA vs. custom CMOS delay ratios of most processor building block circuits fall within the relatively narrow delay ratio range for complete processor cores, but area ratios have much wider variation. Building blocks such as adders and SRAMs that have dedicated hardware support on FPGAs are particularly area-efficient, while multiplexers and CAMs are particularly area-inefficient.

In the second part of this chapter, we discussed the impact of these measurements on microarchitecture design choices: The FPGA substrate encourages soft processors to have larger, low-associativity caches, deeper pipelines, and fewer bypass networks than similar hard processors. Also, while current soft processors tend to be in-order, out-of-order execution is a valid design option for soft processors, although scheduling windows should be kept small and a physical register file (PRF) organization should be used to reduce the area impact of using a CAM-based instruction scheduler.

As we will see in the upcoming chapters, some of these design guidelines have influenced our processor design. We use a PRF organization to reduce the number of RAM ports and use a non data-capture scheduler to reduce CAM size (Section 3.5.6, Chapter 5). We also use set-associative caches (2-way L1, 4-way L2) and TLBs (2-way), with a high capacity for the TLBs (128 entries) (Section 3.5.4, Chapter 12).

As a processor design needs to consider constraints beyond just area and delay, some of the design suggestions made in this chapter (which considered only area and delay) were not used in our processor design. For example, fused ALU operations (Section 3.5.3) would greatly increase the complexity of the processor, and hence were not used, and our L1 cache size was small (8 KB) due to complexity related to address translation rather than area efficiency considerations (Section 3.5.4).

Chapter 4

CPU Design Methodology

Having some understanding of what circuits cost and how processor microarchitecture decisions are impacted by the FPGA substrate, we set out to design an out-of-order soft processor microarchitecture — one that maps reasonably well to an FPGA, yet maintains a fairly classical and aggressive microarchitecture that does not make large performance compromises to fit on the FPGA.

This chapter discusses our design goals, and the design process involved in designing, verifying, and optimizing our processor microarchitecture and circuits.

4.1 Design Goals

To be useful, a processor must run software correctly, fast, and not use too much area. Due to low clock speeds, FPGA processors are not yet power-limited, so the current design did not consider energy consumption to reduce design effort.

4.1.1 Instruction Set Compatibility

The primary design goal for the processor is a sufficiently-correct implementation of the x86 instruction set, which is a requirement for binary compatibility with existing software. As we are targeting compatibility with fairly modern operating systems, we have chosen the 32-bit variant of the instruction set. Floating-point x87 support is mandatory, as most operating systems now target at least the Pentium instruction set, which includes floating point.

Our target instruction set most closely matches the Intel P6 (Pentium Pro)'s user-mode instruction set, but the P5 (Pentium) system-mode features [1]. The primary addition in the P6's user mode are the CMOV (conditional move) instructions, which were straightforward to implement. We avoided the P6's new system-level features, as they were more complex to implement and not usually required to boot an operating system. These features include PAE (Physical Address Extensions) and MTRR (Memory Type Range Registers) [1].

We also chose to omit some rarely-used legacy features to reduce design effort, even though there are no particular technical challenges involved with implementing them. These included support for the x86 hardware task switching mechanism, 16-bit protected mode, and a few other features. This unfortunately means that OS/2 no longer runs, as OS/2 is unusual in making full use of the x86 architecture's features.

We chose to implement floating-point using a new software emulation mechanism that is transparent to the operating system, to reduce the amount of hardware that needed to be designed. The emulation mechanism traps to an emulation routine in firmware in response to a floating-point instruction, instead of taking a normal (OS-visible) exception. In retrospect, this was a poor design decision. Not only is the overhead of emulation very high (programs with more than a few percent floating-point instructions would spend nearly all of its time running floating-point emulation routines), but the complexity of bypassing segmentation and paging for memory operations added hardware complexity right in the critical path of the memory system. A hardware floating-point unit (of any performance level) would have been a better option. See Appendix A for a detailed description of the floating-point emulation mechanism.

4.1.2 Performance

One method to specify an approximate level of performance and complexity is to specify a processor's issue width (peak instructions per clock cycle). For a first implementation of an out-of-order x86 soft processor, we chose a modest two-way issue out-of-order design, which targets a performance level slightly lower than the Pentium Pro (3-issue). This design target is both superscalar and out of order, but due to risk and inexperience, we did not want to attempt an even more complex and higher-performance design.

Within the parameters of a two-way out-of-order design, we have tried to include aggressive microarchitecture features, even some that are not found on a Pentium Pro (e.g., memory dependence speculation). Part of the objective of this thesis is to find FPGA-compatible implementations of modern microarchitecture techniques, so our processor design choices are not purely about finding the easiest method to hit a particular performance target, and include some features that may seem somewhat excessive for a processor of this performance level.

We chose to target a 300 MHz frequency (on a Stratix IV FPGA) for each component in the processor. This target was chosen as it is somewhat higher than a Nios II/f on the same FPGA (240 MHz), and we felt it was aggressive yet plausible to achieve. While the amount of pipelining could potentially achieve even higher frequencies, we felt that a reasonable processor design would have a single-cycle ALU for simple operations, whose critical path would include at minimum a 32-bit adder and a set of operand forwarding multiplexers, where the multiplexers would likely have comparable delay to the adder. 300 MHz is approximately twice the delay of a 32-bit adder (See Table 3.8). We do not expect every component to achieve the target, however we also do not progressively lower the target as we discover components that miss the target. Not only would this result in a monotonically-decreasing frequency target, but if the target drops low enough, we would need to reconsider whether those components that did reach the frequency target should be re-pipelined to shorten the pipeline length.

4.1.3 Area

As FPGAs have grown exponentially for the 15 years since the 1700-LE 32-bit Nios soft processor was released [114], we now have much more area to spend on a soft processor core. At the time, 1700 LE was 3.3% of the then-largest Altera FPGA's 51 840 LEs (EP20K1500E). If we use that as a measure of "reasonable" resource usage, 3.3% of the largest Stratix 10 (SX 5500 with 1 867 680 ALMs) would be 60K ALMs.

Since performance is our primary goal, we do not specifically optimize for resource utilization. We merely aim for a "reasonable" resource utilization of several tens of thousands of ALMs, which is a substantial increase compared to current small soft processors, while still comfortably fitting into a modern FPGA.

4.2 Design Process

The design of a processor, like most systems, is a mostly top-down, but also iterative, process. We divided our process into roughly three parts.

- Define **architecture**: The behaviour of the processor — what instructions and functionality it supports and what it does.
- Design **microarchitecture**: A collection of hardware blocks and their cycle-by-cycle behaviour that together implement the architecture. The microarchitecture also determines the instructions-per-cycle performance of the processor.
- Design **circuits**: Circuits implement each hardware block of the microarchitecture. The resulting circuit determines the cycle time and area (resource usage) of the block of hardware.

Due to the complexity of the design, we rely heavily on intuition to reduce the amount of iteration, particularly for architecture and microarchitecture design, where the high abstraction level means circuit delay and cost cannot be directly measured. For example, we had to redesign the floating-point emulation mechanism (an architectural feature) once it became clear that an unnecessarily complex microarchitecture was needed to implement it, and we had to extend the pipeline length of the processor's memory system (a microarchitectural feature) once it was clear that circuit-level optimizations could not create a circuit fast enough to implement the microarchitecture.

The following sections discuss each stage in more detail.

4.3 Processor Architecture

The processor architecture (or instruction set architecture) is largely dictated by the x86 instruction set. The design work that remained involved choosing what x86 features were safe to omit, and designing a mechanism to emulate floating-point instructions.

We started our processor design based on Bochs 2.4.6, an x86 full-system simulator [115]. Bochs simulates the functionality of I/O devices and the CPU, but does not attempt to model the cycle-by-cycle performance. To test whether unused legacy features (e.g., 16-bit protected mode) are safe to remove, we instrumented Bochs to test whether the feature is ever used. We tested with many existing x86 operating systems, which we describe in more detail in Section 4.4.2. The floating-point emulation mechanism was tested by running both the emulation code and the native floating-point instruction and comparing the emulated floating-point unit state with the actual FPU state after each floating-point instruction.

4.4 Processor Microarchitecture

The processor's microarchitecture was designed by implementing a detailed cycle-by-cycle simulation model (in C++) of the CPU and replacing Bochs' CPU implementation. The end result is a detailed performance model for the CPU along with a functional model of the full system outside the CPU. The detailed simulator should behave identically to the functional simulator, and is used both for performance tuning and verifying the microarchitecture.

Our implementation consists of a set of two separate simulators, one using Bochs' CPU model modified to implement our architectural changes, and one using our detailed CPU model to evaluate the microarchitecture. We refer to these two simulators as the Bochs CPU and detailed CPU models, respectively, in later sections. Verification is done by printing out logs in the same format from both simulators and comparing them.

4.4.1 Detailed Performance Model

The detailed CPU model is structured similarly to the pipeline of an out-of-order x86 processor. This includes a front-end that fetches and decodes x86 instructions to micro-ops, then executes and commits the micro-ops. The goal is to produce a simulation model that is detailed enough that hardware modules could be designed based on the simulated behaviour of each hardware block.

The detailed CPU model is cycle-driven (not event-driven). For each simulated cycle, every simulated hardware unit or pipeline stage performs its work for that cycle. Cycle-driven was chosen because it more easily corresponds to hardware, making translation into a hardware design easier.

Due to the overall goal of building the processor in hardware, the detailed model has a higher level of detail than a typical microarchitecture-level performance simulator such as SimpleScalar or PTLsim [116, 117]. A performance simulator often decouples the functional simulation from the performance simulation, which allows the simulator to execute programs correctly while still allowing easy experimentation with the performance model, as the performance model only needs to be mostly correct to give sufficiently accurate performance results. In contrast, we rely on the behaviour of the simulated hardware blocks to be precisely correct, deliberately making it easy for a change to the simulated behaviour to cause a functional error. This "execute-in-execute" approach allows us to verify that the hardware design being simulated is functionally correct, but makes it harder to explore the design space due to the need to always maintain a functionally-correct hardware model.

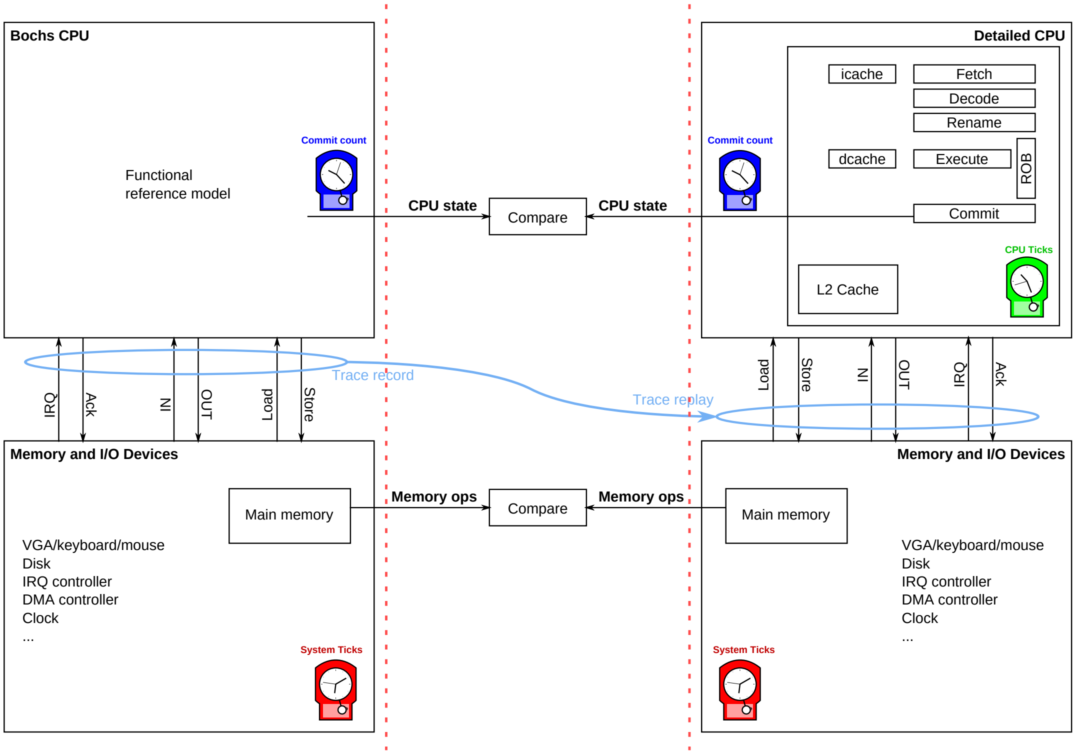


Figure 4.1: Microarchitecture simulation and verification flow

4.4.2 Verification

Correctness Verification

To verify the correctness of the microarchitecture, we need to compare its behaviour to a known-good reference model. We use the Bochs CPU as our reference model. A simple comparison of the final output of a test program is insufficient, as many test programs are not performing a computation with a well-defined output (e.g., booting an OS), and a simple mismatch is near-impossible to debug as it does not give information about when in the billions of cycles of simulation the first error occurred. Thus, we need to verify the behaviour near-continuously to detect as soon as an error occurs.

Continuously verifying the processor behaviour is non-trivial as we need to ensure that both the Bochs and detailed CPU models behave identically over the entire simulation. We need to remove all sources of non-determinism and timing-dependent behaviour so that both simulation models have identical behaviour regardless of the exact timing (i.e., IPC) of the detailed CPU model.

We verify our microarchitecture design by simulating a large set of workloads, described in the next section.

To compare correctness, both of our simulators print out the CPU state after each committed instruction. We compare the instruction pointer value, flags, architectural general-purpose registers, and whether an interrupt was pending. To improve simulation speed, we optionally compare a 32-bit hash of the above state instead of the entire state.

Early in the design process, we also tracked memory loads and stores. Tracking memory accesses allows detecting errors that only affect memory state when the error occurs, instead of detecting it (much) later once the erroneous memory value is consumed by the processor and shows up as an incorrect CPU state. However, once the detailed model implemented out-of-order execution and caches, the sequence of memory operation is expected to change and thus a mismatch can no longer be used to indicate incorrectness. Memory loads are expected to occur speculatively so their timing and loaded values can differ, and caches drastically altered the expected stream of memory accesses, by filtering, delaying, and coalescing memory accesses into cache-line sized accesses.

Although not verifying memory accesses does reduce coverage slightly, we believe that verifying the CPU state alone is sufficient. Verifying the CPU state already ensures the correctness of all computations and all intermediate values written to memory (those that are eventually read from memory again). The case that could be missed is if the correct final result of a computation is written to memory incorrectly and never read again, which is highly unlikely to occur given how many other memory accesses are verified.

Performance Verification

Performance verification aims to verify that the processor design performs at an expected level. Performance bugs are difficult to detect, as they usually show no obvious symptoms, other than executing code slower than expected, but correctly. Compounding the problem is that prior to completing the design, we do not know with certainty whether the expected performance level is achievable, which can make it difficult to distinguish between a performance bug and simply hitting a performance limit inherent in the chosen design.

Our main tool for performance verification is to instrument the detailed performance simulation to collect statistics that indicate how well each component of the processor performs. The simulator currently collects over 200 measurements. These measurements may be simple counts (e.g., counting the number of pipeline flushes), histograms (e.g., distribution of x86 instruction lengths), or even traces of interesting events (e.g., IPC at each cycle as the processor recovers from a pipeline flush). Where it makes sense, we compare some of these metrics to performance counter measurements of commercial x86 processors running the same workload. Appendix E shows some of these measurements and comparisons.

As a sanity check of the per-component measurements, we also compare overall IPC to other x86 processors. Chapter 13 presents some of these comparisons. Complex benchmark measurements (unlike microbenchmarks) serve mainly as a sanity check because they are usually too complex to understand which parts of the processor pipeline are exercised, and too complex to compute by hand what the expected performance *should* be. If a performance bug is suspected, it is usually debugged either by focusing on a small section of the benchmark that exercises the relevant portions of the processor, or using a microbenchmark that exercises the relevant components.

Tests and Benchmarks

We benchmarked and verified our processor design using a collection of workloads, shown in Table 4.1. Our test workloads consist of several user-mode benchmarking suites (run under Linux), a few other individual user-mode applications, and booting a large set of operating systems that tests both system mode and user mode behaviours. The benchmark suites range from SPECint2000 intended for high-performance systems to the very small CHStone suite intended to benchmark C-to-gates high-level synthesis. As much of x86's complexity is in its system-mode behaviours, we tried to test with many operating systems. We excluded benchmarks that made heavy use of floating-point code because our processor design uses slow floating-point emulation.

Due to the long runtime of our test workloads, we routinely benchmark with only a subset of these workloads. We did not develop a directed test suite, as developing a sufficiently-complete test suite on its own is a huge undertaking. We did use a few *ad hoc* microbenchmarks for performance verification and debugging.

Synchronization and Removing Non-Determinism

In order to compare the behaviour of our detailed CPU simulation to the Bochs CPU model, the simulation runs must be deterministic and produce the same results in both simulators. There are many sources of non-determinism that need to be removed, as many behaviours depend on time, and the passage of time varies depending on how fast the CPU executes instructions.

To make "time" independent of CPU execution speed, we divide the simulation model into three clock domains: CPU internal clock (detailed model only), committed instruction count ("commit count"), and system (outside the CPU) tick count. The CPU internal clock (detailed model only) is visible only within the CPU. Its purpose is to count CPU cycles to determine IPC. The commit count is used as our primary measure of time. It is simply a count of the number of committed instructions since power-on. Since all correct processors regardless of IPC should execute the same program with precisely the same results if the initial conditions are the same (ignoring asynchronous events like interrupts for now), the committed instruction count can be used as a measure of time that has the same definition for both the functional and detailed simulators. The system tick domain is a feature of Bochs that is used to allow simulation of the processor HLT state, where the CPU sleeps until the next interrupt with no instructions committed while the rest of the system continues running. We did not change this behaviour. Care was needed to ensure the implementation of the processor entering and leaving HLT state occurred at precisely the same time in both models, as the system tick is what determines the timing of hardware interrupts.

Using commit count as a measure of time, the functional and detailed simulators with identical initial state should perform the same sequence of actions and remain synchronized unless a non-deterministic asynchronous event occurs differently in one simulator than the other. Sources of non-determinism include user interaction with the simulated input devices (keyboard, mouse), interrupts, and memory accesses.

We address user-generated non-determinism by requiring the user to avoid interacting with the simulation GUI. All of our test workloads are designed run without user interaction.

Workload	**x86 Inst.** (10^6)	**Description**
SPECint2000	20 000	gzip, gcc, mcf, crafty, parser, gap, vortex, bzip2 **Excluded:** eon, vpr, twolf, perlbmk
MiBench [118]	2 568	**Automotive:** bitcount, qsort, susan **Consumer:** jpeg, mad, tiff2bw, tiffdither, diffmedian, typeset **Office:** ghostscript, ispell, stringsearch **Network:** patricia, dijkstra **Security:** blowfish, pgp, rijndael, sha **Telecom:** adpcm, crc32, gsm **Excluded:** basicmath, lame, tiff2rgba, fft
Stanford[1] version 4.2	13.5	Perm, Towers, Queens, Intmm, Puzzle, Quick, Bubble, Tree **Excluded:** Mm, FFT
CHStone 1.9 [119]	3.3	dfadd, dfmul, dfdiv, dfsin, mips, adpcm, gsm, jpeg, motion, aes, blowfish, sha
VPR 7.0 [120]	54 316	FPGA pack, place, and route, in Linux
Quartus II 6.0[2]	54 158	FPGA synthesis, pack, place, and route, in Windows XP
Dhrystone	260	200 000 iterations
CoreMark	247	600 iterations
Doom 1.9s	1 938	-timedemo demo3
MS-DOS 6.22	500	DOS and a bunch of drivers
Windows 3.1	300	DOS 6.22 + Windows 3.1
Windows 95	330	DOS + 9x kernel
Windows 98	600	DOS + 9x kernel
Windows 98 SE	600	DOS + 9x kernel
Windows Me	500	DOS + 9x kernel
Windows NT 4.0	1 400	32-bit NT kernel
Windows XP	5 200	32-bit NT kernel
Windows 7	16 000	32-bit NT kernel
Mandriva Linux 2010.2	15 500	Desktop Linux OS (kernel 2.6.33)
Android 3.0.1	8 000	Linux-based OS
FreeBSD 9.3	4 200	UNIX-like OS (no GUI)
FreeBSD 10.1	4 000	UNIX-like OS (no GUI)
ReactOS 0.3.14	1 600	Windows NT clone
Syllable Desktop 0.6.7	4 000	Desktop operating system
Haiku R1/Alpha 4.1	22 600	Desktop operating system
OS/2 2.1	33	Not supported: run until first unsupported feature

[1] The “Stanford” benchmarks by John Hennessy and Peter Nye do not appear to have been published, yet have been used to benchmark real machines [121].

[2] Quartus II 6.0 was the latest version that did not require SSE instruction set extensions.

Table 4.1: Benchmarks and x86 instruction counts

Interrupt non-determinism comes from two sources: whether the right interrupt is signalled at the right moment in time, and whether a pending interrupt is serviced at the correct time. The former problem is addressed by keeping simulator time synchronized: if requests *to* I/O devices happen at the same time, then any interrupts *from* I/O devices as a consequence would be triggered at the same time. The latter problem is currently solved by disallowing instruction commit past an instruction boundary in a single clock cycle when doing verification (thus limiting x86 IPC to 1), allowing pending interrupts to be checked and handled after every instruction. Simulation runs for benchmarking allow committing two instructions per cycle, which can cause minor changes in behaviour by sometimes taking interrupts one instruction later.

Memory access timing non-determinism occurs because in a speculative out-of-order processor, memory accesses occur speculatively, earlier than they would have occurred if the processor executed each instruction to completion sequentially. Caches also change the number and timing of memory accesses. Fortunately, accessing regular memory has no side effects (if caches are coherent with DMA, which is true in x86), so changing the timing of memory accesses has an impact only for memory that performs memory-mapped I/O. Since memory-mapped I/O should always be uncacheable (UC) accesses, the simulator can, when used for verification, optionally perform uncacheable stores at the "correct" time, instead of the more realistic behaviour of deferring stores for a few cycles after instruction commit.

Another source of easily-solved timing-dependent behaviours involve the software observing the various system clocks. The system clock was set to start at the same time at power-on for every simulation run; the system clock was set to depend solely on system ticks; and RDTSC (read timestamp counter) was changed to report commit counts instead of CPU cycles.

Simulation Speed and Parallelization

Like all detailed CPU simulators, our detailed simulator is slow, on the order of 0.25 MIPS (simulated millions of instructions per second) when running on a 4 GHz Intel Haswell system. For long simulations (billions of instructions), running the entire simulation serially is prohibitively slow (e.g., 18 hours simulation time to boot Windows 7). Thus, we need to partition long simulations into shorter sections (typically 10^8 – 10^9 instructions per section), and simulate each section in parallel. This requires that the simulator allow simulations to begin at a point other than system power-on.

Simulators often use checkpoints or fast-forwarding to allow beginning simulation at an arbitrary point in time. Checkpoints capture the complete system state at an instant in time. This state is restored, and simulation continues starting at this point. Fast-forwarding is similar, except that instead of skipping over some initial portion of the simulation, there is a method to quickly execute that initial portion (at low detail). For a full-system simulator such as Bochs, a checkpoint must include not only the state of CPU and memory, but also the internal state of every I/O device that could be attached to the system, which makes checkpointing intractable.

We use fast forward instead, by generating a trace of memory accesses and interrupt events using the functional Bochs simulator, then reading the trace in the detailed simulator and performing this sequence of operations at precisely the right moments in time. Since the processor interacts with the rest of the system solely by memory accesses, I/O accesses, and interrupts, performing a sequence of these actions — without actually simulating the operation of the CPU — is sufficient to put the all of the system except the CPU into the correct state. We then use a checkpoint of only the CPU state to restore the CPU state to begin detailed simulation. The trace record and replay mechanism is illustrated in Figure 4.1. Events generated when simulating with the Bochs CPU are recorded to a trace, which is replayed in a detailed CPU simulation with the CPU disabled to fast forward the non-CPU state. The fast-forwarding mechanism is effective: since none of the instructions are simulated during fast forward, the simulator fast forwards at about 55 MIPS, more than 200 times faster than simulation.

The trace is carefully encoded, and consumes an average of about one byte per instruction. This trace can be stored on disk or generated on-the-fly by running both simulators concurrently and sending the trace through a named pipe (FIFO). The trace also contains CPU state snapshots after every million instruction commits (and one at the end of the trace) to allow detailed simulation to start at any multiple of one million commit counts, allowing the same trace to be used to fast forward by a variable amount. In practical use, we often generate the trace on-the-fly. This is slower, as a Bochs CPU (functional-only) simulation only runs at 15 MIPS. This only has a small effect on total simulation time as the time spent in the detailed CPU portion still dominates, but avoids consuming large amounts of disk bandwidth to read a trace at 55 MB/s per simulation, typically with many simulations running in parallel.

4.5 Circuit Design

Unlike many FPGA systems, processors are latency sensitive and deep pipelines have a large performance cost. We must spend effort on creating fast circuits, as simply pipelining mediocre circuits to get high f_{max} would result in unnecessarily long pipelines.

The processor microarchitecture was designed with intuition on how each hardware block could be mapped into FPGA hardware (usually LUTs). Our circuit designs usually started out with a rough idea of the LUT circuit structure. We implemented circuits in SystemVerilog, then optimized the RTL code until the resulting circuit was fast enough, or could not be made faster. Optimization typically involved looking at the LUT-level circuit of the near-critical paths and comparing them with a hand-drawn sketch of the circuit, and either correcting the sketch due to missing details, or changing the RTL code until the critical path matched the sketch.

In most cases, we spent considerable effort in optimizing the RTL code, as we found that a straightforward behavioural coding style rarely produced a circuit that was as fast as our hand-designed circuits, especially for datapath circuits that are more structured. An initial straightforward behavioural implementation often had cycle times 50% longer than our final tuned designs. A particularly frustrating challenge was the amount of extra effort needed to prevent logic optimization algorithms from making the circuit worse, even when the desired circuit structure is already coded to be obvious. Because most of our circuits needed manual design, many of the subsequent thesis chapters describing the detailed processor design contain LUT-level circuit designs.

In most cases, we did not enable physical synthesis or register retiming optimization algorithms, as they were not effective. Because of the effort spent designing at the LUT level, there is little room for optimization algorithms to improve on our circuit design.

We did not attempt to do any manual placement or routing optimizations. There is less potential for improvement compared to optimizing logic synthesis and technology mapping. Even if manual placement or routing were worth doing, they would have to be done after the processor design is complete and verified.

4.5.1 Verification

Circuit designs were generally verified by simulating the circuit Verilog in ModelSim. We used a small number of hand-crafted test cases that test both common and corner cases, and ensured that they behaved according to their microarchitectural specification.

Future work should test using more test vectors and compare a larger number of test vectors to the software detailed microarchitecture simulation.

4.6 Summary

This chapter discussed our processor's design goals, and the method by which be designed and verified the processor instruction set architecture, microarchitecture, and circuit designs.

For this processor design, we aim to increase performance and reduce cost (area), while being sufficiently 32-bit x86 compatible. Due to low clock speeds, FPGA processors are not yet power-limited, so we did not consider energy consumption, to reduce design effort. Instruction-set level changes (such as FPU emulation) were verified by modifying an existing x86 functional simulator (Bochs). The microarchitecture design was verified by comparing the behaviour of a detailed pipeline model of our processor design with the functional simulation, running 16 operating systems and many more user-level workloads. Eliminating non-determinism caused by simulating an out-of-order processor was essential for debugging, by allowing the behaviour of the out-of-order model to match the functional-only model. As not all of the circuit designs for the processor are complete, circuits have been less thoroughly tested than the microarchitecture.

Chapter 5

Proposed Processor Microarchitecture

After having defined our processor's instruction set architecture in Section 4.3 and an approximate performance target of two-way issue in Section 4.1.2, we now design a microarchitecture that correctly implements the architecture and achieves reasonable efficiency on an FPGA. This chapter describes the end result of the microarchitecture design, while more detailed trade-offs are presented in subsequent chapters.

Our microarchitecture is shown in Figure 5.1. The processor is designed to sustain a peak performance of close to two micro-ops per cycle. The front-end is 10 stages. Each cycle, the processor can fetch 8 bytes, decode two instructions into two micro-ops each (four micro-ops total), and rename and issue two micro-ops. The out-of-order execution core uses four separate scheduler queues and can dispatch up to 4 operations per cycle, one from each queue. Once instructions are completed, two micro-ops are dequeued from the reorder buffer and committed per cycle. Committed stores then write back to the L1 cache, handling cache misses if necessary.

5.1 Clustered PRF Organization

We chose a Physical Register File (PRF) organization, which is advantageous for reducing register file read/write ports, as discussed in Section 3.5.6. Because the PRF organization stores both speculative and non-speculative (committed) values in the same unified register file, extra register file read and write ports are not needed to copy values during instruction commit.

We chose to organize the processor's execution units into clusters, where each cluster contains a scheduler, its own register file, and the attached execution unit. During instruction execution, each cluster can read any register from any other cluster with no penalty, but can only write to its own cluster's register file. The main advantage is that the register file can be implemented without multiple write ports, using three 7r1w (7 read port, 1 write port) clusters. On FPGAs with dual-ported block RAMs, increasing the number of read ports can be done reasonably efficiently by replicating the RAM, but increasing the number of write ports is much more difficult, either by implementing the entire RAM in flip-flops, or using extra logic such as a live value table (LVT) [72]. Instead, we use the register renamer to track in which cluster source registers are located, and ensure writes always go to a physical register in the same

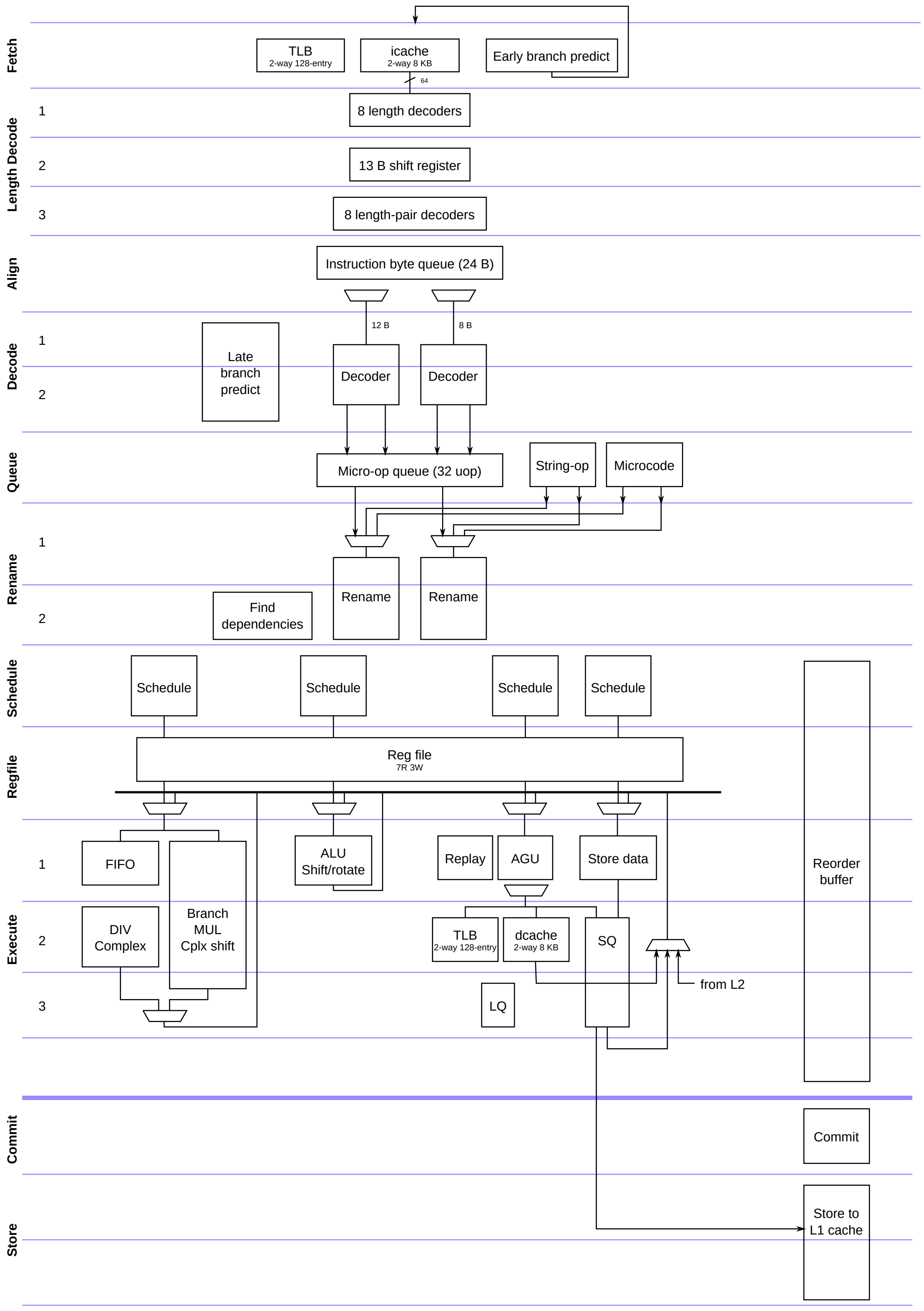


Figure 5.1: Proposed processor core microarchitecture

cluster. Conceptually, this scheme resembles a LVT-based memory, except the function of the LVT is subsumed by the register renamer, and the cluster choice is made in advance, instead of in the same cycle as the register file read operation.

Using the register renamer to track register file clusters has a disadvantage. An instruction must be assigned to a cluster by rename and cannot be changed later in the pipeline. This reduced flexibility may make less efficient use of the instruction schedulers and execution units due to the loss of flexibility to schedule an operation on any free execution unit. However, our design does not have replicated execution units (all four units are distinct), so this loss of flexibility would only affect future higher-performance designs with replicated execution units.

5.2 Micro-ops

Our micro-ops are structured quite similarly to an x86 instruction, and is each able to encode a load, computation, and store. The motivation of this design is to allow most x86 instructions to be translated into one micro-op, regardless of which addressing mode is used. This reduces the complexity of instruction decoding, as it reduces the variation in the number of micro-ops into which an instruction could decode.

Like x86 instructions and unlike most RISC-style instructions, micro-ops use a destructive source operand where the destination operand overwrites one of the source operands. Although non-destructive operands are more flexible, there is still a cost (in register renaming) to support them, and non-destructive source operands are rarely needed because micro-ops are decoded from destructive two-operand x86 instructions.

An example of a complex read-modify-write instruction that can be encoded by a single micro-op is the following:

`add [eax + ecx*4 + 12], edx` (A four-byte instruction: `01 54 88 0c`)

This instruction performs a memory load from `[eax + ecx*4 + 12]`, adds `edx` to it, then stores the result back to the same memory location (`[eax + ecx*4 + 12]`). All of this in encoded in one micro-op.

For execution, a micro-op is treated as up to three separate operations: An address generation that provides the memory address (AGU), a computation, and a store-data operation. Data dependencies between the three sub-operations within a micro-op are communicated through registers. For example, the above instruction would have three sub-operations (AGU, ALU, and store-data), three source operands (`eax`, `ecx`, and `edx`), and two destination operands (both unnamed architecturally-invisible registers, one for the load result and one for the addition result).

This micro-op design is often (but not necessarily) used with address generation units (AGU) that can handle both loads and stores rather than specialized load-only and store-only AGUs. As our design uses only one AGU, the AGU of course handles both loads and stores. In future designs, using dedicated load AGUs and store AGUs may be preferable because store AGUs do not need to write back to the register file or bypass networks, allowing extra AGU execution bandwidth without requiring extra writeback bandwidth.

We chose to use complex (load-operate-store) micro-ops (as shown above) instead of simpler single-operation (load, operate, *or* store) micro-ops. Using complex micro-ops reduces the number of micro-ops (and thus, the required micro-ops per cycle throughput) at the expense of larger micro-ops. Complex micro-ops can reduce multiplexing because each portion of the wide micro-op is steered to the appropriate execution cluster, as the load/store and arithmetic portions of the micro-op are actually independent. We speculate that the benefit of reduced multiplexing from using fewer micro-ops outweighs the cost of larger micro-ops with fields that are often left unused by simpler operations that do not make full use of the complex micro-op.

5.3 Instruction Set Specific Considerations

Many processors, including those using x86 and many RISC instruction set architectures (ISAs), share many features not usually associated with simple RISC processors, such as variable-length instructions (ARM Thumb-2 [122], x86 [1]), the use of microcode for complex operations [109, 110, 123, 124], decoding instructions into micro-ops [109, 110, 123–126], serializing (pipeline flush) operations [1, 122, 127], and even string-like operations that load or store a variable number of bytes (ARM `ldm/stm` [122], Power `lswx/stswx` [127], x86 `rep` instructions [1]).

This section describes how we implemented some of these features in the context of our x86 processor. Features that are neatly contained within one particular unit will be discussed in their respective chapters.

5.3.1 Variable-Length Instructions

The x86 instruction encoding is variable-length. However, the bulk of the difficulty is not that instructions have 15 legal lengths, but that the instruction length is not easily computed. In the worst case, one cannot ascertain the length of an instruction until all bytes of the instruction have been examined. In our design, we compute the lengths of instruction pairs in the three-stage length decoder, before sending instruction bytes to the instruction decoders. Instruction length decoding is covered in more detail in Section 6.4.

5.3.2 Atomic Instruction Commit

For processors with precise interrupts (i.e., all practical processors) where instructions can be broken into multiple micro-ops for execution, the processor must always commit either none or all of the micro-ops associated with an instruction.

In a straightforward instruction commit design, the commit stage examines the reorder buffer (ROB) entry at the ROB's tail pointer, and commits instructions while the tail instruction is "completed", with no knowledge of instruction boundaries. To support atomic instruction commit, we continue to use the tail pointer in this manner, but instead of committing the tail micro-op, the commit unit only searches for the next instruction boundary to compute a `commit_to` pointer that then permits a second tail pointer to commit micro-ops up to that point. The `commit_to` pointer is updated when the next completed *instruction* (not micro-op) is found, and indicates that the micro-ops belonging to all instructions up to that point have completed. Section 8.2.1 in Chapter 8 discusses atomic instruction commit in more detail.

5.3.3 Serializing Operations

A serializing instruction causes all preceding instructions to complete before the serializing instruction is executed, and waits until it is committed before later instructions are fetched and executed.

Our implementation uses three mechanisms to implement serializing behaviour: Two kinds of instruction pausing (pause-before and pause-after) located after the decoded micro-op queue immediately before register renaming, and an unconditional branch micro-op that guarantees a branch misprediction.

When a serializing instruction is decoded, its first micro-op is marked as "pause before", and its last micro-op is marked as "pause after". Pause before causes the pipeline to stall before register renaming until both the ROB and store queue are empty, guaranteeing all earlier instructions have committed and stored their results to cache. A micro-op marked with pause after causes the pipeline to stall before register renaming until that micro-op is committed (thus emptying the pipeline). The combination of these two pauses ensure that the serializing instruction is not reordered before any earlier instructions, nor reordered after any later instructions.

An unconditional-mispredicted-branch micro-op can be used to force the next instruction to be re-fetched after the serializing instruction executes. Our processor design currently uses this micro-op for the `IRET` instruction, but not for other serializing instructions. The x86 specifications require that the instruction following a serializing instruction is *fetched* after the serializing instruction has committed, so our processor is not entirely compliant with the x86 specifications. Re-fetching after an IRET was necessary because IRET changes the current code segment, so the speculatively fetched instructions can be from the wrong segment and need to be re-fetched after the code segment change. We have not seen any adverse effects for other serializing instructions, and we expect this only has the potential to affect *cross*-modifying code in cases where the instructions immediately following a serializing instruction are fetched and decoded, then subsequently modified by a *different* processor. Self-modifying code is handled with a different mechanism, described later in this section.

5.3.4 Microcode and String Operations

While complex instructions can be decoded into a sequence of multiple micro-ops, instructions whose behaviour depends on the value of registers (not only on the opcode itself) cannot be translated into micro-ops until the register value is known. An additional complication is that registers are renamed, so there is no way to retrieve the value of a register without first renaming it. For example, `mov ds, [eax]` (load a segment register) requires generating additional micro-ops to read the segment descriptor from either the global descriptor table (GDT) or local descriptor table (LDT), but which table to read from and the offset depend on the new segment selector that is loaded from memory (from `[eax]`). These instructions are handled by first executing a micro-op to read the register value, then triggering the string-op (for string operations) or microcode units to generate micro-ops in response.

To ensure correct operation, an instruction that will use the string or microcode units is decoded into a micro-op with the “pause after” flag set. This causes the pipeline to stall before register renaming to wait for all of the micro-ops to be generated by the string or microcode unit before continuing with the next instruction, as micro-ops belonging to different instructions can not be reordered while still in the in-order front-end portion of the processor.

There are many corner cases involving exceptions occurring within string-op or microcode streams. Each iteration of a string operation is committed atomically, as allowed by x86. For microcode sequences, the entire sequence is committed atomically. This can be challenging, as the longest such sequence is currently 42 micro-ops for taking an exception from virtual 8086 mode into protected mode.

5.3.5 Floating-Point Emulation

The floating-point emulation mechanism is different from the usual undefined instruction exception due to the need for the emulation routine to be invisible to the operating system. Floating-point instructions cause the processor state to be saved and a branch into a hard-coded location in firmware code, very similar to what would be done for an exception. In the current implementation, floating-point exceptions and exceptions caused by floating-point memory accesses are handled in hardware (without emulation) because there is no OS-transparent method to handle an exception inside the emulation routine.

In order to be invisible to the OS, the floating-point emulation code and data are located at a fixed physical memory address, but floating-point loads and stores must perform their memory accesses in the context of the current processor state, which may or may not have segmentation and paging applied. Thus, instruction fetch bypasses segmentation and paging during emulation mode, and the memory system must allow per-instruction bypassing of segmentation and paging for data loads and stores. More details of the floating-point emulation mechanism are in Appendix A.

5.3.6 Self-Modifying Code

Unlike most other instruction sets, self-modifying code is supported by hardware in the x86 instruction set. This not only means that data and instruction caches are coherent, but that modifying an instruction will have an immediate effect, even if the modified instruction is already fetched and in the pipeline, with no serializing or cache flush operation required. This even includes corner cases such as a string instruction (using the REP prefix) overwriting itself and having its new opcode take effect immediately after the iteration that overwrites the opcode.

Detecting self-modifying code that has not been fetched is done through standard cache coherence. Two sets of filters are used to check for conflicts between fetched instructions already in the pipeline and store addresses. One filter tracks instructions that have been fetched (checked by stores when executing), and the other tracks store addresses (checked by instruction fetches). The need for two sets of filters is because stores do not modify the cache until some time after the store commits, but the instruction being modified could potentially be fetched in the vulnerability window after the store is committed but before the change is applied to the cache.

Both filters are cleared after a pipeline flush. The fetched-instructions filter is cleared immediately, but the store-address filter is cleared once the store buffer is drained. This portion of the design is not well-optimized, so there may be better solutions than our current implementation.

5.4 Summary

This chapter described the high-level microarchitecture of the processor. Detailed descriptions of each portion of the processor will be presented in the upcoming chapters. We used a clustered physical register file (PRF) organization (as recommended in Chapter 3) as it better suits the FPGA substrate. Using a PRF reduces register file (memory) ports by avoiding copying register values on commit, reduces CAM size by allowing the use of a data-less instruction scheduler. Dividing the physical register file into clusters reduces the number of register file write ports to one per cluster. These design choices reduce the use of circuit structures that are particularly expensive on FPGAs.

Like most out-of-order x86 designs, we decode instructions into micro-ops. We chose to use wider, complex (load-operate-store) micro-ops instead of simpler single-operation (load, operate, or store) micro-op, as we expect that using a lower throughput (micro-ops per cycle) of complex micro-ops would be less expensive in multiplexers than using a higher throughput of simpler micro-ops, for equivalent processor performance.

This chapter also discussed mechanisms to handle various behaviours required by the x86 instruction set. Many of these requirements are not unique to x86: Variable-length instructions, floating-point, and atomic commit of micro-ops also appear in some RISC-like instruction sets, while serializing operations are required when the processor supports some form of supervisor mode to support operating systems. Since our processor needs to run unmodified x86 operating systems correctly, a large amount of design effort went into designing and verifying that supervisor mode and various infrequent behaviours were handled correctly. This is particularly evident in the memory system (Chapter 12) where a large number of behavioural requirements needed to be satisfied (e.g., segmentation, paging, and its privilege checks, memory ordering rules, and cache coherency).

Chapter 6

Instruction Fetch and Decode

In a processor with out-of-order execution, the processor's front end (fetch, decode, and register renaming) still operate in program order. The instruction fetch and decode stages fetch instructions from the instruction cache and then decode the instructions into a stream of micro-ops that will be sent to the register renamer and execution. The front end fetches a stream of instructions that follow the predicted correct path, and thus includes branch predictors.

As our processor aims to sustain a peak throughput of two micro-ops per cycle, the front-end must also be able to sustain this throughput. The complex x86 instruction encoding makes this more challenging. The lengths of the variable-length instructions are not easy to compute, particularly because of the use of prefix bytes in the encoding, where the instruction length is affected by the number of prefix bytes used. Yet, the lengths of two instructions must be computed every clock cycle. Instructions are then decoded into a variable number of micro-ops (most often 1 per instruction, but can be up to 15 in our implementation), requiring the decoder to have a mechanism to handle the worst-case behaviour.

This chapter describes the design of our instruction fetch (excluding the instruction cache) and decode stages. We look at the properties of a typical x86 instruction stream, which informs the instruction fetch bandwidth requirement and the design of the instruction length decoder. We also look at the hardware design of the instruction decoders and how the branch predictors interact with the different parts of the fetch and decode pipeline.

6.1 Workloads

In this chapter, measurements of the behaviour of workloads use the following subset of workloads from Table 4.1:

- SPECint2000
- MiBench
- Dhrystone
- CoreMark
- Doom
- Windows XP

	Prefixes 0 or more bytes	Opcode 1 or 2 bytes	ModR/M 1 byte	SIB 1 byte	Operands 0 or more bytes	Length	
1:		90				1	nop
2:	66	0f 84			10 00	5	jz +0x0010
3:		01	c0			2	add eax, eax
4:		01	84	41	78 56 34 12	7	add [ecx+eax*2+0x12345678], eax
5:		81	84	41	78 56 34 12 dd cc bb aa	11	add [ecx+eax*2+0x12345678], 0xaabbccdd
6:	66 66 66	81	84	41	78 56 34 12 bb aa	12	add [ecx+eax*2+0x12345678], 0xaabb

Figure 6.1: Instruction encoding and examples. 1: One-byte instruction. 2: Two-byte opcode, length-changing prefix, and 16-bit immediate operand. 3: With ModR/M byte. 4: With ModR/M, SIB, and 32-bit immediate offset. 5: With ModR/M, SIB, 32-bit immediate offset and 32-bit immediate operand. 6: Same as 5 but with a (redundant) length-changing operand size prefix.

This subset of the workload has about 30.2 billion instructions.

6.2 x86 Instruction Encoding

An x86 instruction consists of zero or more prefix bytes, followed by a one- or two-byte opcode, bytes that specify addressing mode if necessary, and operands if necessary. Figure 6.1 shows the general format of a 32-bit x86 instruction and a few examples. Recent instruction set extensions that we chose not to support, such as SSE and AVX, have further extended the encoding complexity beyond what we describe here.

Every x86 instruction must have an opcode. Opcodes are one byte or two bytes if the first byte is `0f`, for an opcode space of 512 opcodes, minus bytes reserved for prefixes. When designing length decoding, it is sometimes convenient to consider two-byte opcodes as a `0f` prefix followed by a one-byte opcode. Taking this view, an instruction consists of any number of prefix bytes followed by a one-byte opcode. Including `0f`, there are 12 different prefix bytes (thus, bytes of those values are not used for opcodes). The initial version of the instruction set used on the Intel 8086 did not have a limit to the instruction length. A limit of 10 bytes was added on the Intel 286 [128], and increased to 15 bytes since the Intel 386 [129], which we support.

Instruction length computation starts by scanning for the instruction opcode (the first byte that is not a prefix). The 9-bit opcode (8-bit opcode and whether a `0f` prefix was used) then determines the number of operand bytes and whether a ModR/M byte is used to specify the addressing mode. If a ModR/M byte is used to specify the addressing mode, the ModR/M byte will specify whether a SIB (Scale Index Base) byte follows the ModR/M byte. In the worst case, prefix bytes, the opcode, ModR/M, and SIB bytes will completely specify the instruction length.

Figure 6.1 shows several examples of 32-bit instruction encodings. The simplest instruction is a single opcode byte. The second example shows a prefix (`66`) and a two-byte opcode `0f 84` that specifies an immediate operand and no ModR/M byte. The operand-size prefix (`66`) not only increases the instruction length by one byte, it also modifies the size of the immediate operand (the operand here would be 4 bytes long without the `66` prefix). A prefix that changes the size of the instruction operands is known as a *length-changing prefix*, and can increase length decoding difficulty depending on the circuit design.

The third example shows a straightforward instruction that uses a ModR/M byte. The fourth example is the same instruction, but now the ModR/M byte specifies both the use of a SIB byte and an immediate offset operand. The fifth example shows the longest possible 386 instruction with a one-byte opcode that uses no prefixes (including `0f`), with two four-byte immediate operands. The last example illustrates the use of a length-changing operand-size prefix, where *redundant* use of a prefix is legal and has no special meaning. While the need to examine up to three bytes (opcode, ModR/M, SIB) to determine instruction length is already complicated, the potentially large number of redundant prefixes (up to 14 prefix bytes for a 15-byte instruction) is even more difficult.

Because the worst-case encoding of an x86 instruction is difficult to handle (15 bytes, up to 14 prefixes, decoding into up to 15 micro-ops each), we need to characterize a realistic instruction mix and design the decoders for typical instruction encodings rather than worst case. Behaviours that are typical need to be decoded quickly (two per cycle), while rare worst-case behaviours only need to be handled correctly, with a performance penalty. Our design took advantage of three properties of realistic instructions: instruction lengths tend to be much shorter than the 15-byte maximum, very few prefix bytes are typically used, and instructions typically decode into very few micro-ops.

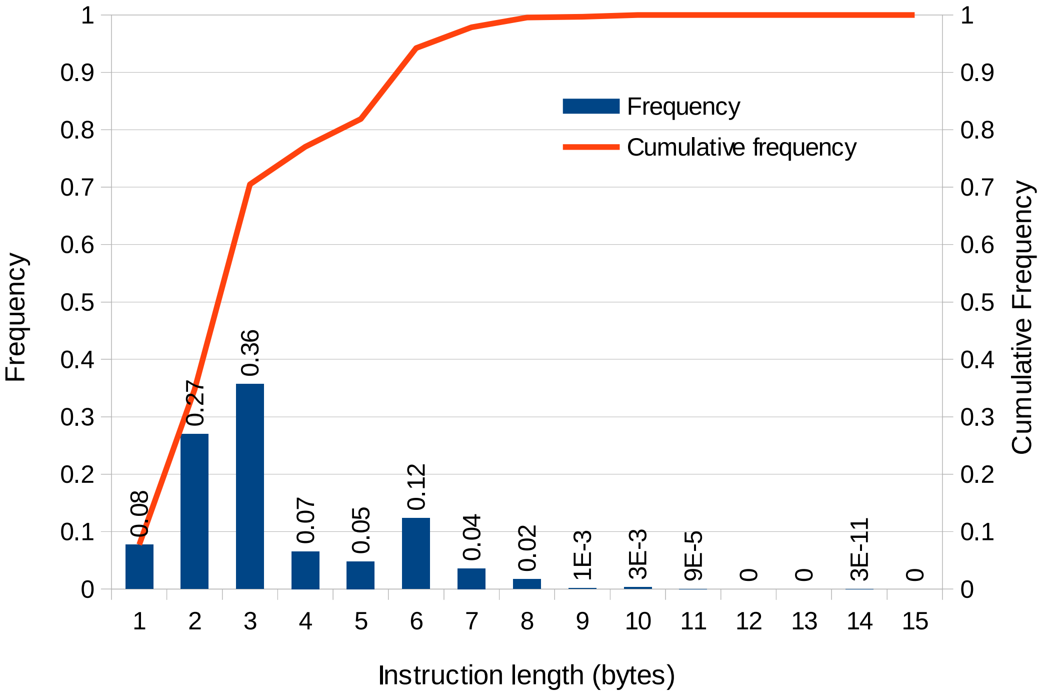


Figure 6.2: Instruction length distribution

Figure 6.2 shows the distribution of instruction lengths over the 35-billion instruction workload from Section 6.1. About 70% of instructions are 3 bytes or shorter, but there is a cluster around 6 bytes. While full-length instructions still need to be decoded, disallowing dual-issue of long instructions will have a minimal performance impact if long instructions are rare. We use this observation to simplify the second instruction decoder.

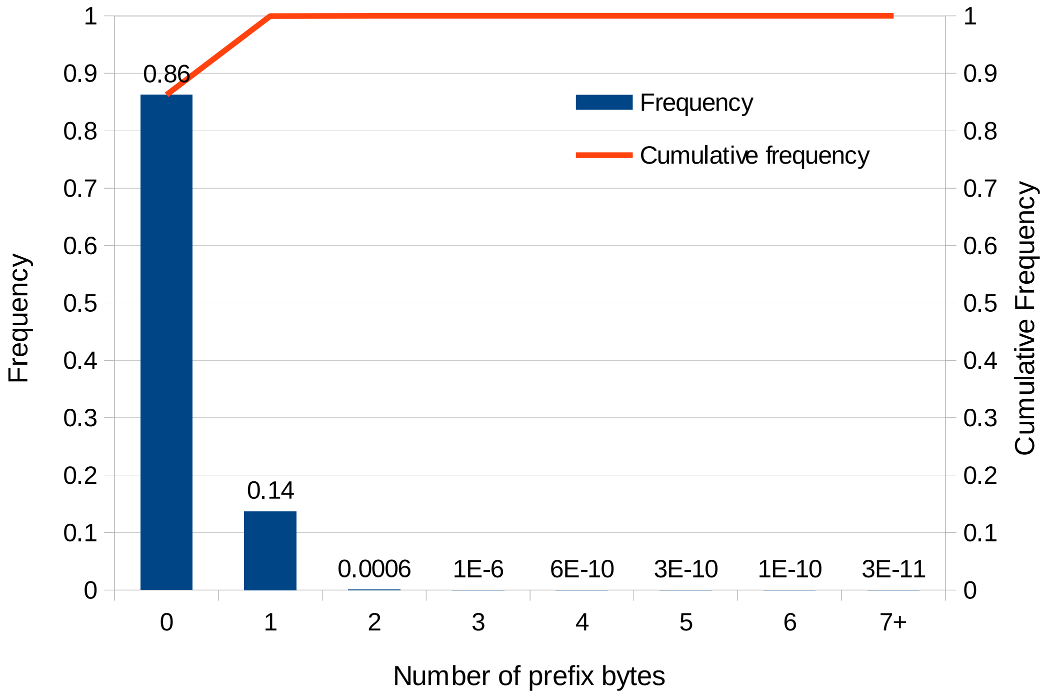


Figure 6.3: Prefix byte usage

Figure 6.3 shows the distribution of prefix byte use. In this chart, we classified the first byte (`0f`) of a two-byte opcode as a prefix byte. Most instructions (86%) do not use a prefix byte at all, and 14% of instructions use one prefix byte (this includes two-byte opcodes with no other prefixes). Only 0.06% of instructions use two or more prefix bytes. We take advantage of this distribution by designing our length encoders to only handle up to one prefix byte (serving 99.94% of instructions), and serially process the prefixes for instructions that use more prefixes. We note that this result does not hold when newer instruction set extensions such as SSE or x86-64 are used. SSE instructions use many three-byte opcodes, often with an operand-size prefix, and x86-64 adds an extra prefix byte to many instructions.

Figure 6.4 shows how many micro-ops each instruction requires. Most instructions (94%) decode into one micro-op, while those requiring 4 or more micro-ops are rare (0.08%). The average is 1.09 micro-ops per instruction decoded[1]. Thus, we can design the instruction decoders to handle quickly only instructions with few micro-ops while processing the rest slowly over multiple cycles.

6.3 Instruction Fetch

The fetch unit fetches 8 aligned bytes per cycle. This is sufficient for a two-way issue processor because the average instruction length is around 3.4 bytes. Sustaining two instructions per cycle requires an average fetch bandwidth of 6.8 bytes per cycle. To confirm this rough bandwidth calculation, we simulated the

[1] This increases to 1.12 micro-ops per instruction (measured at commit) when including micro-ops that did not come from the instruction decoder, such as string operations and microcode.

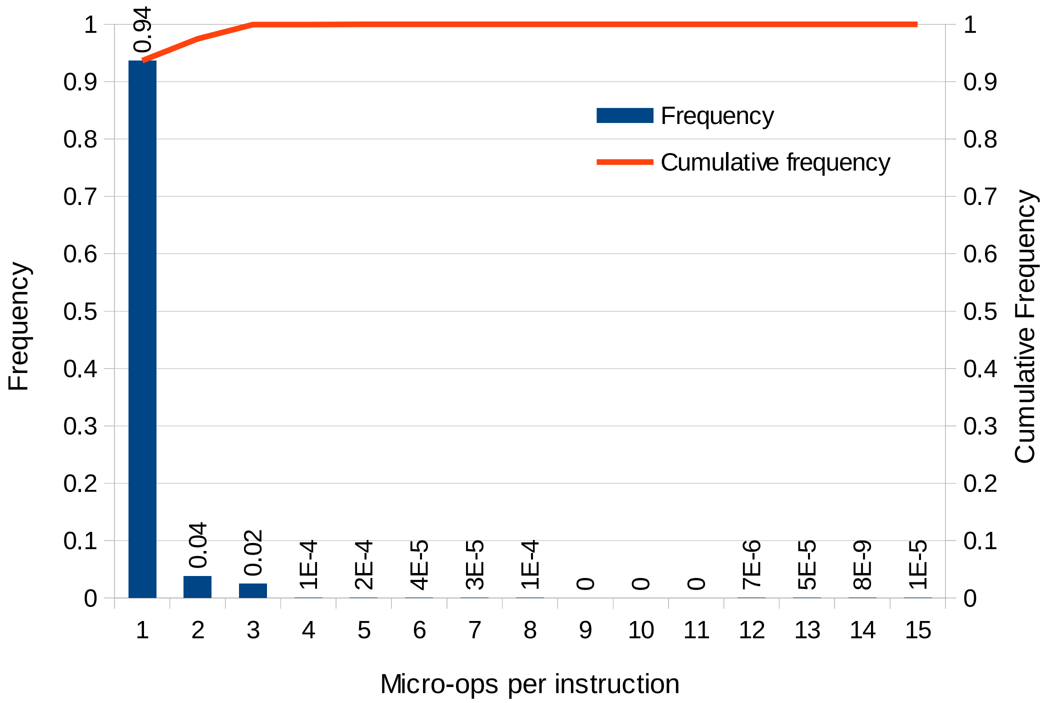


Figure 6.4: Micro-ops per instruction distribution

processor with varying fetch bandwidths from 1 to 32 bytes per cycle, and measured the average instructions per cycle on a set of workloads. Figure 6.5 shows that for our processor design, fetch bandwidth does not significantly limit performance at 8 or more bytes per cycle.

The design of the instruction caches are presented along with the rest of the memory and cache system in Chapter 12. We use a standard instruction (opcode) cache and did not consider caching decoded micro-ops due to the increased complexity that would be required to manage two fetch/decode pipelines.

6.4 Length Decoding

After being fetched, the instruction bytes go through three pipeline stages of length decoding. Instruction bytes remain 8-byte aligned throughout these stages. The purpose of these stages is to compute instruction lengths, which are used by the align stage to pick out instructions from the byte stream.

The first length decode stage contains eight instruction length decoders that decode in parallel the length of the instruction that begins at each byte position. One length decoder is shown in Figure 6.6. As discussed in Section 6.2, the use of more than one prefix byte is rare (0.06%). Thus, the length decoder is only capable of computing the length of an instruction that has at most one prefix byte, and needs to observe at most four bytes to do so. Instructions that have two or more prefix bytes are treated by the length decoder as though they were instructions with two-byte length, which the decoder will accumulate (to remove redundant prefixes) at a rate of two prefix bytes per cycle until the opcode is found.

Since computing the instruction length requires examining four instruction bytes, the last three po-

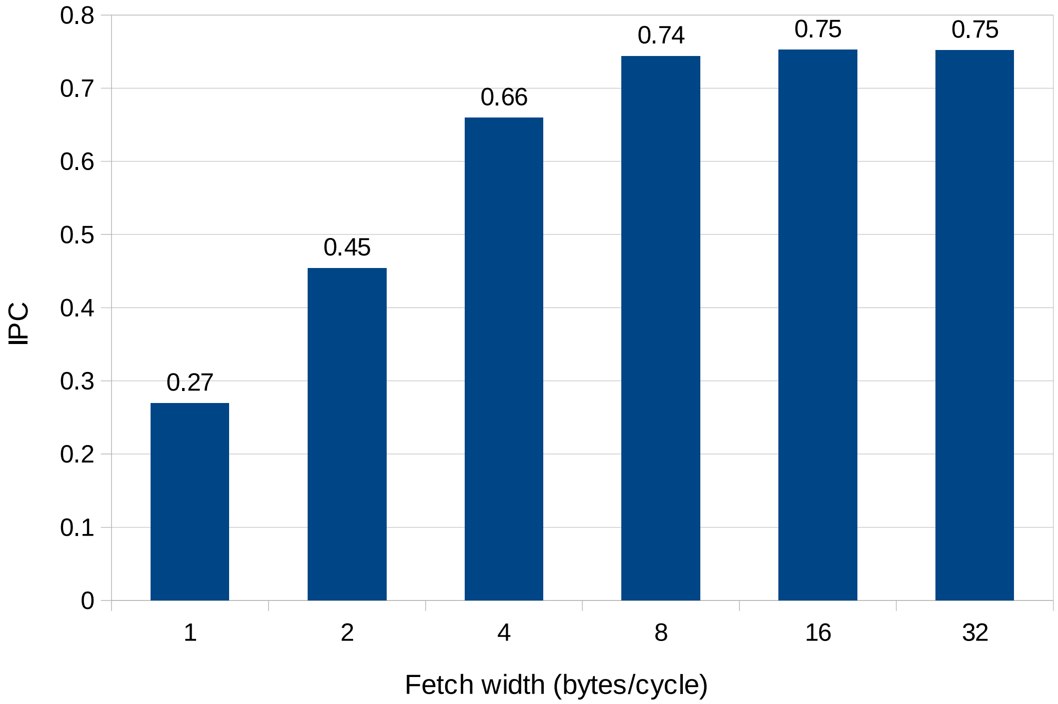


Figure 6.5: IPC vs. fetch bandwidth

sitions (bytes 5–7) of each eight-byte fetch block cannot be computed until the next block of instruction bytes are fetched. Thus, at each cycle, the eight length decoders produce eight lengths per cycle: the final three lengths for the previous fetch block and the first five lengths for the current fetch block.

The second length decode stage adds one cycle of latency to accumulate a larger window of instruction bytes before computing the lengths of instruction *pairs* in the third stage. Knowing the length of each instruction is not enough information to dual-issue instructions within our cycle time target. In the align stage, which extracts the instruction bytes for each instruction, knowing only the instruction lengths produces a critical path that must traverse two sets of multiplexers: One to select the start of the first instruction, retrieving its length, which then drives a second multiplexer to select the length of the second instruction in order to know how many bytes to advance in a cycle. To reduce this critical path, we use an additional pipeline stage to compute the lengths of a dual-issued instruction *pair* starting at each position. With length pairs computed, the align stage can dequeue two instructions per cycle with only one set of multiplexers on the critical path.

Figure 6.7 illustrates how instruction length pairs are computed when given the lengths of instructions that start at each byte position. To remove the need for addition, instruction lengths are represented by a number indicating the start position of the next instruction, rather than the length itself. In each cycle, a 13-byte window is available (the final three bytes of the second fetch window are not yet available, as explained above). Eight instruction pair lengths are computed by selecting the length of the second instruction when given the length of the first. For clarity, only positions 0 and 7 are shown in the figure. Only having a 13-byte window restricts dual issue to cases where the second instruction in a pair starts

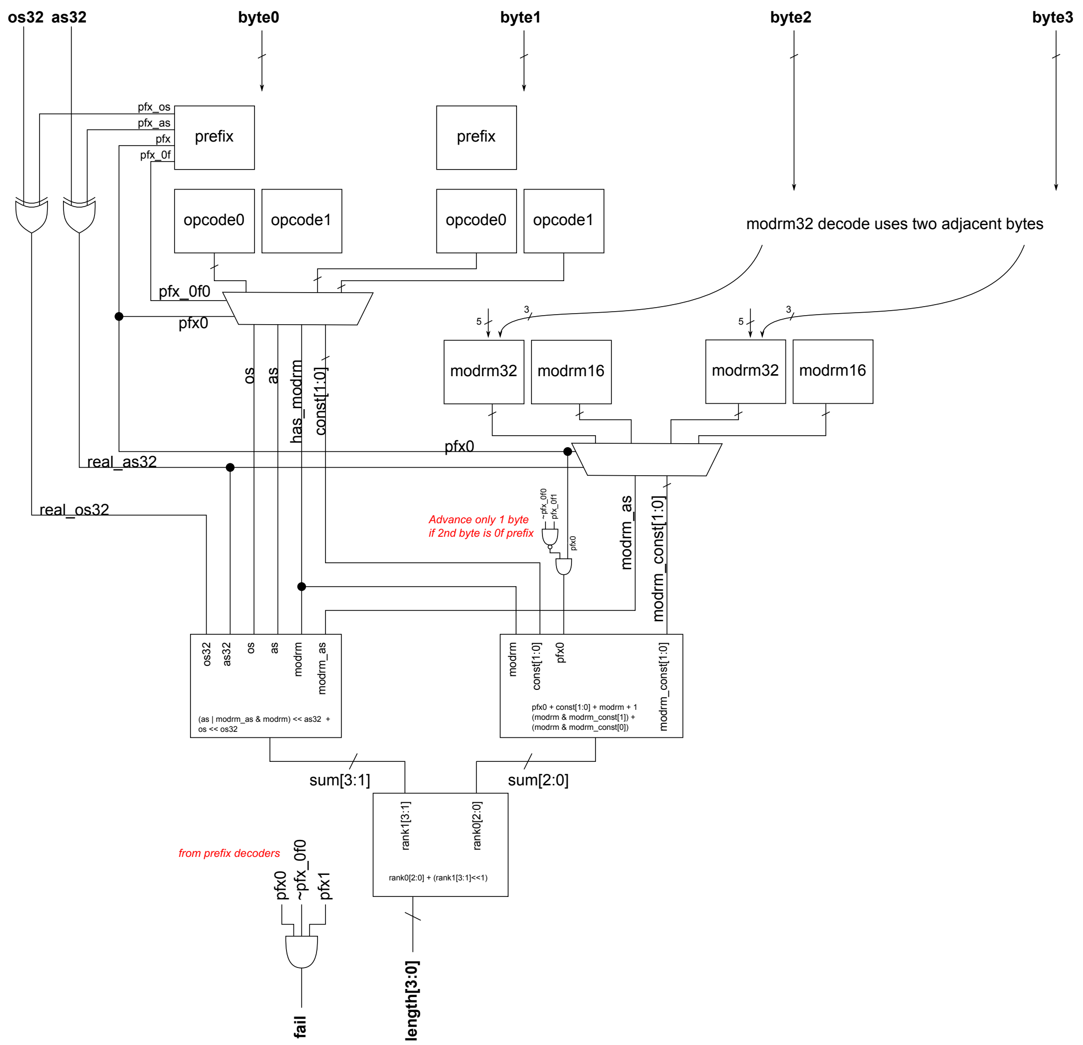


Figure 6.6: One instance of a length decode circuit

(but not necessarily ends) no later than byte position 12. For example, in the worst case where the first instruction starts at byte position 7, dual issue can occur only if the first instruction is at most 5 bytes long (placing the start of the second instruction at byte 12). We expect this limitation to reduce dual issue by around 2.4%[2], since this limitation affects only instructions that start late in the fetch window and most instructions are short. To reduce multiplexer size, we disallowed dual-issue if the first instruction's length exceeds 7 bytes. We expect this to have a minimal effect as only 2.2% of instructions are longer than 7 bytes.

Branches that are predicted taken need special handling. In general, branch instructions do not end at 8-byte fetch block boundaries, and branch targets are not 8-byte aligned. Thus, there may be trailing bytes

[2] This is computed by using the distribution of instruction lengths in Figure 6.2. Instructions start at random positions within a fetch block, and this limitation affects length 7 instructions that start at position 6, and length 6 and 7 instructions starting at position 7.

at the end of a taken branch instruction's fetch block and leading bytes before the branch target that need to be discarded. To avoid an extra cycle of overhead for every taken branch, we override the computed instruction length for predicted-taken branches and pretend it is a longer instruction that consumes all of the discarded bytes. To the length decoder and align stages, a taken branch appears as an unusually long instruction, while the instruction decoder is not affected by extra trailing bytes after an instruction.

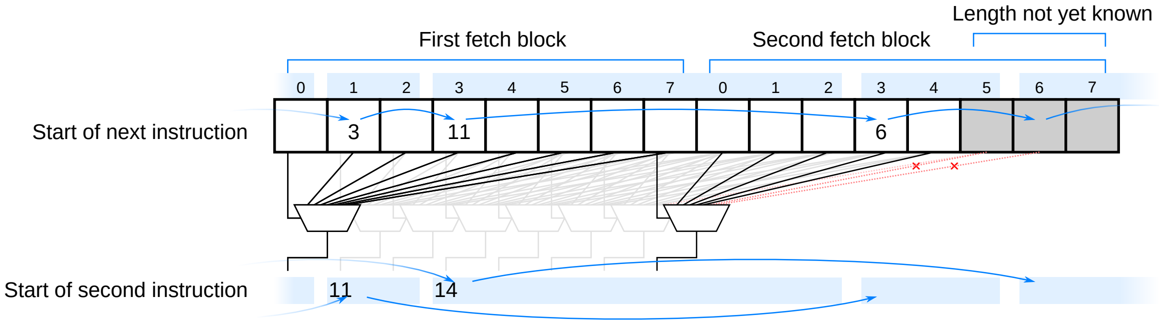


Figure 6.7: Computing instruction lengths for dual issue. Instruction lengths are encoded as a pointer to the beginning of the next instruction or instruction group. Instruction lengths are first computed and marked, then a set of multiplexers compute lengths of pairs of dual-issued instructions. This diagram shows an example with three instructions of length 2, 8, and 3 bytes starting at offset 1 of the first fetch block.

6.5 Align

The align stage receives one aligned 8-byte chunk of instruction bytes each cycle, and must pick out up to two adjacent instructions, align them to instruction boundaries, and send them to the instruction decoders.

The align stage contains a three-fetch-block (24-byte) queue. This is the minimum queue size, as in the worst case, a long instruction (10 bytes or greater) could begin on the last byte of the first 8-byte block and span three fetch blocks. Two instructions are processed if the queue contains enough bytes for both instructions. Otherwise, one or even zero instructions are processed.

Two sets of multiplexers steer the instruction bytes for each instruction to the two instruction decoders. These two sets of multiplexers are 12 bytes and 8 bytes for the first and second decoders, respectively. The first decoder is designed to handle the worst-case 12-byte instruction, while the second is designed to handle only the common case. The worst-case instruction length is only 12 bytes here because instructions only contain at most one prefix. Instructions that contain two or more prefix bytes are treated as though they were two-byte instructions that cannot be dual-issued, which causes two prefix bytes to be sent each cycle to the first decoder until the number of prefixes remaining is one or zero. The first decoder contains logic that accumulates prefixes sent over multiple cycles. We limited the second decoder to handle only 8-byte instructions because instructions with 9 or more bytes are rare (0.45%, Figure 6.2), and the penalty is that they cannot be the second instruction in a dual-issued pair (i.e., less than one cycle penalty).

6.6 Instruction Decode

Our processor's two instruction decoders take two pipeline stages. Each decoder receives instruction bytes and produces up to two micro-ops that implement the instruction. As seen in Figure 6.4, supporting two micro-ops handles about 98% of instructions. Instructions that decode into 3 or more micro-ops cannot be dual-issued. Instructions with 3 or 4 micro-ops use *both* decoders to produce up to 4 micro-ops in a single cycle (with the same instruction bytes steered to both decoders). Instructions using more than 4 micro-ops (0.06%) are decoded at a rate of 4 micro-ops per cycle.

6.6.1 Zeroing Idioms

Zeroing idioms are instructions are commonly used to set a register to zero. In x86, these are the `xor` and `sub` instructions where the source and destination register are the same. For these instructions, the result is always zero regardless of the register's previous value, yet the instructions imply a data dependence on the earlier value.

Most out-of-order x86 processors, including ours, will recognize these zeroing idioms and break the data dependence, which exposes more instruction-level parallelism. On our design, recognizing zeroing idioms improved IPC by an insignificant 0.05%. The small impact is likely because our processor is not starved for instruction-level parallelism, as it has few ALUs (one for common-case arithmetic and one for load/stores) and the arithmetic unit has low latency (1 cycle).

6.7 Branch Prediction

The front-end is designed with two branch predictors: one located at the instruction fetch stage, and one at the instruction decoders. We refer to them in Figure 5.1 as the early and late branch predictors, respectively. Table 6.1 lists the parameters for the two branch predictors. The branch predictors have not been thoroughly tuned, so they still use fairly simple prediction algorithms and large table sizes.

The early branch predictor is intended to guide instruction fetch to fetch the correct sequence of 8-byte blocks, with no penalty for taken branches (beyond discarded instruction bytes from misaligned branch instructions and targets). In this implementation, we use a fairly standard combination of branch target buffer, two-bit counters for direction predictor, and a 16-entry return address stack (RAS). We leave the design of an improved branch predictor for future work.

The early branch predictor is a block-level predictor: given the current fetch block, it predicts the next fetch block. At the instruction fetch stage, the early branch predictor has no information on the instructions being executed, so it must not only predict direction, but also whether there is a branch within the fetch block and the branch target, thus, using the BTB for all branches. To handle fetch blocks containing multiple branches, the BTB index hash function includes the low-order bits of the fetch address. The low-order bits are usually zero, but are non-zero when the fetch block is a branch target that is not aligned to a fetch block. This method avoids the need to make multiple independent branch predictions per fetch block.

	Early	Late
Branch target buffer (BTB)	16K entries, 4-way associativity	512 entries
Return address stack (RAS)	16 entries	16 entries
Direction predictor	Two-bit counters in BTB	8K entries

Table 6.1: Simulated branch predictor parameters for each component of the early and late predictors.

Configuration	IPC	Pipeline Flushes (Millions)
Default	0.744	227
Late predictor disabled	0.722	293
Early predictor disabled	0.698	226
No branch predictor (not-taken)	0.376	2668

Table 6.2: Performance impact of branch predictors. (30.2 billion committed instructions)

The late branch predictor has a similar structure as the early branch predictor (BTB, direction, and RAS). However, the late branch predictor is more accurate despite using smaller table sizes as it has more information than the early branch predictor. The late branch predictor knows with certainty which instructions are branches. Since the majority of branches are direct branches (91%) and the branch targets of direct branches can be calculated, the BTB is only used for a small fraction of the branches.

Additional complexity in the early branch predictor results from not knowing the instructions during prediction. The early predictor can incorrectly predict the existence of a branch. The case where the predictor predicts a taken branch when no branch instruction exists needs special handling. Normally, incorrect predictions are corrected when a branch instruction is executed. However, when no branch instruction exists, there is no micro-op that will verify the branch prediction. Thus, we ensure that an instruction that is predicted to be a taken branch must always decode into a branch instruction, or the processor front end will be flushed. Also, the early predictor uses path history instead of branch (taken/not-taken) history because it predicts using fetch blocks and has no knowledge of whether or how many branches are contained in the fetch block.

We use the late branch predictor as an overriding predictor: its predictions are assumed to be more accurate than the early predictor and overrides the earlier prediction if the two predictors disagree. Due to its location in the pipeline, an override results in flushing the front end of the pipeline, causing a 7-cycle pipeline bubble. In ideal cases, most of this bubble can be hidden by the 32-entry micro-op queue. Otherwise, branch overrides cost front-end bandwidth, but do not cause pipeline flushes of the out-of-order portion of the processor.

Table 6.2 evaluates the effectiveness of the two branch predictors. By default, we use both predictors. We also show the impact of disabling the early branch predictor (predict not taken), disabling the late branch predictor (always agree with early predictor), and disabling both. The results show that the late branch predictor is more accurate, as there are fewer pipeline flushes when only the late branch predictor is used compared to when only the early predictor is used, but due to the high latency of the late predictor, IPC is actually lower. Omitting the early branch predictor costs 6% IPC loss even though the number of pipeline flushes actually decreases.

Despite the generous branch predictor table capacities, our branch predictor does not match the accuracy of modern branch predictors. Recent desktop processors can achieve 100% branch prediction accuracy on simple benchmarks such as Dhrystone, while our design still has a 3% branch prediction miss rate on Dhrystone.

6.8 Hardware

We have implemented most of the instruction decode stages on a Stratix IV FPGA, with two major exceptions. Our decoder circuit[3] currently supports approximately 100 of 350 opcodes, and none of the branch predictors are implemented. The overall f_{max} for the decode stages is 247 MHz (median of 31 random seeds). The critical path is currently in the first stage of the length decoder.

Figure 6.8 shows our circuit design for the instruction decoder. The length decoder occupies three pipeline stages. The first stage performs length decoding, the second stage accumulates two fetch blocks of results (instruction lengths), and the third stage computes the length of instruction pairs. This is followed by the align stage with a 24-byte instruction queue that picks two instructions per cycle to decode. Decoding aligned instructions is split into two independent tasks: cracking the x86 instruction into micro-ops, and extracting the operands from the instruction (immediate values, displacement, and decoding the addressing mode). The decoded micro-ops may then substitute the *instruction's* operand values into the *micro-op's* immediate value or addressing mode fields as necessary. The final stage is the micro-op queue, which behaves as one pipeline stage despite the two-cycle-latency RAM blocks, due to the use of forwarding multiplexers.

Further pipelining of the decode stages is possible, because most of the pipeline stages are feed-forward stages with no loops. The important timing *loop* is in the align stage, where two variable-length instructions are dequeued each clock cycle, and the next dequeue cannot begin until the previous dequeue has completed. The align stage in isolation currently runs at around 330 MHz.

Table 6.3 shows a breakdown of resource utilization by component. Unsurprisingly, components with large shifters and multiplexers (align and micro-op queue) consume a fairly large amount of resources. The 32-entry micro-op queue is more complex than its name suggests. It contains wide $\sim$165-bit micro-ops and allows up to 4 enqueues per cycle. Its inputs contain multiplexers to select from the variable number of micro-ops (1 to 4) that may be produced from the decoders and pack them into contiguous positions in the queue. Due to the slow MLAB RAMs, there are also two cycles of bypass multiplexers to achieve single-cycle FIFO latency.

[3] Note that the decoder's *microarchitecture* design is complete, including the branch predictor design and micro-op sequences for nearly every x86 instruction and behaviour. Our *circuit* implementation is less complete than our *microarchitecture* design (implemented as a detailed pipeline simulation).

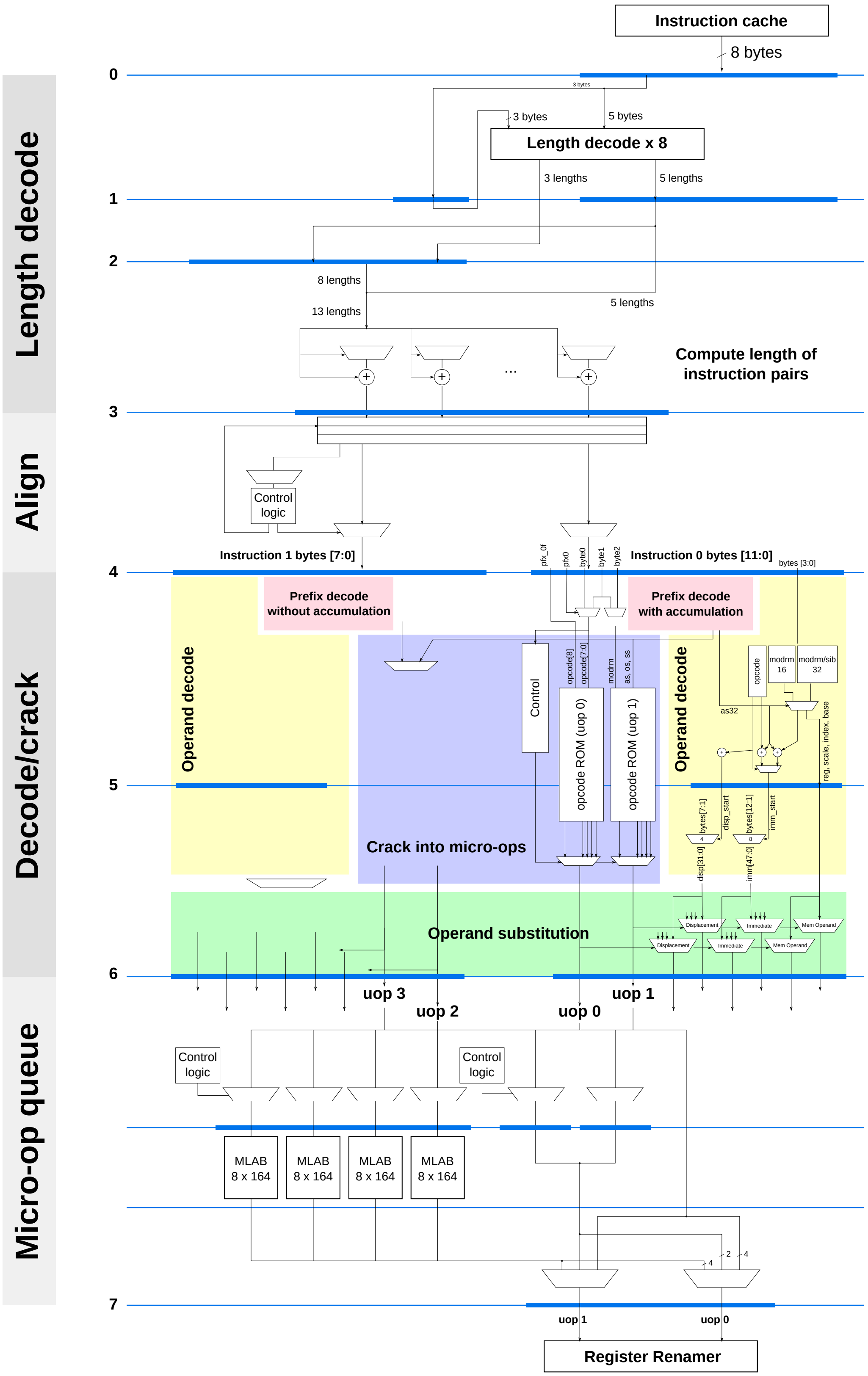


Figure 6.8: Length decode, decode, and micro-op queue. Details for decoding instruction 1 (left side) are omitted as it is almost identical to decoding instruction 0.

Unit	ALUT	ALM	Status
Fetch	–	(∼300)	Fetch is part of the memory system (See Chapter 12).
Length decode	1055	780	Complete
Align	931	851	Complete
Decode	1090	941	Partial, supports about 100 opcodes of about 350
32-entry micro-op queue	2625	1807	Complete
Branch predictors	—	—	Not implemented
Total	5833	4281	No branch predictors; Supports approx 100/350 opcodes.

Table 6.3: Resource consumption and status of hardware implementation of decoder sub-blocks

6.9 Future Work

To complete the decoding stages, branch predictors need to be added and the rest of the instruction set needs to be implemented in hardware.

The current branch predictor design (simulated, not currently implemented in hardware) has not been thoroughly tuned. The most obvious improvement is to use a more accurate prediction algorithm than a simple global predictor (two-bit counters indexed with a hash of the instruction pointer and global branch history). One possibility is a TAGE-like predictor [130], which would be particularly useful as the late predictor because the prediction does not need to complete in less than one clock cycle.

Another direction to explore is whether the overriding late predictor is in the optimal location in the pipeline. Although we have shown that using both early and late predictors outperforms using either one alone, it may still be possible to improve the accuracy of the early predictor or reduce the latency of the late predictor by moving the late predictor earlier in the pipeline. If the early predictor becomes sufficiently accurate, it may even be possible to replace the late predictor with a branch target address calculator that only corrects the branch targets (but not direction) of direct branches.

To improve cycle time, the decoder can be further pipelined without much difficulty until limited by the timing loop in the align stage, for a potential improvement of perhaps 20 – 30%. Handling for the corner case where instructions decode into more than four micro-ops can also be moved out of the decoder to simplify the near-critical decode stage, likely in exchange for an extra cycle or two of overhead for those instructions (0.08% of instructions).

6.10 Summary

This chapter discussed the microarchitecture and circuit design of the front-end of the processor, starting with fetching instruction bytes, until decoding into micro-ops. The fetch, length decode, and decode units are the most exposed to the complexity of the x86 instruction set. While x86 instructions are not particularly complex *on average* (e.g., short opcodes, one micro-op per instruction), part of the design challenge is to design a circuit that handles the common cases at a sufficient throughput and the uncommon cases correctly, and to minimize the performance loss from the slower handling of the uncommon cases.

Like much of the rest of the processor, many of the circuits were manually mapped to logic elements to improve delay. More than half of the logic resources are used for queues (the align and micro-op queues in Table 6.3), which is related to the high cost of multiplexing on FPGAs. These queues are not specific to x86: The align queue would still be required (but possibly smaller) for any variable-length instruction set, including RISC instruction sets with instruction compression, and the micro-op queue is always required if the front-end needs to be decoupled from the register renamer.

Chapter 7

Register Renaming

After instructions have been fetched and decoded, we now have a stream of micro-ops. Before they can be executed, the register operands of each micro-op needs to be renamed, to remove false register dependencies. This is done by the register renamer.

After decoding, micro-ops specify operations using *logical* register numbers. Logical registers consists of architectural registers and a few other registers used by micro-ops but not directly by x86 instructions (See Section D.2 in Appendix D for a complete list of logical registers). The micro-op stream contains false data dependencies (write-after-write and write-after-read) that reduce the amount of instruction-level parallelism that can be extracted. Register renaming removes these false dependencies by dynamically mapping logical registers to a larger set of physical registers.

Due to speculative execution (from branch prediction, the possibility of exceptions, and a few other reasons), a repair mechanism is needed so that the processor state can be restored whenever a misspeculation occurs. Register renaming serves an important role in implementing this repair mechanism, thus the design of the register renaming mechanism impacts the design of many other areas of the processor. The renaming algorithm affects the structure of the register files, the reorder buffer (ROB), mechanisms for recovering from branch mispredictions and exceptions, and even instruction scheduling.

This chapter begins by giving a high-level description of the physical register file (PRF) renaming scheme used in our design. We then compare it to an alternative scheme, reorder-buffer-based renaming, which has been used successfully in some x86 processors. We then discuss several design decisions used in our implementation and its FPGA hardware implementation results.

7.1 Physical Register File Organization

The register renamer renames logical register numbers into physical register numbers. Our design uses a physical register file (PRF) organization, which was discussed briefly in Sections 5.1 and 3.5.6. The PRF organization is not new, and has been used on many processors, including some x86 processors[1]. A PRF organization uses a single register file to hold both committed and speculative registers, and no

[1] Examples of processors that use a PRF organization, from Section 3.5.6: MIPS R10K, IBM Power 4, 5, 7, and 8, Intel Pentium 4 and Sandy Bridge derivatives, Alpha 21264, AMD Bobcat and Bulldozer derivatives

other structure in the processor holds register values. Register mapping tables (also called Register Alias Tables, or RAT) track which physical register holds the value for each logical register, and whether a physical register holds committed or speculative state. Figure 7.1a illustrates the structure of the integer register file in our design.

Our design uses two mapping tables: a speculative mapping table that maps logical registers to the most speculative (i.e., most recently allocated) physical register holding each logical register's value, and a committed mapping table holding register mappings corresponding to the committed state of the processor. The committed mapping table is used for recovery from branch mispredictions and exceptions, and will be discussed in more detail in Section 7.2.2.

In the register renaming pipeline stage, each source operand reads the register mapping table to map its logical register number to a physical register number. Each destination operand is allocated a new physical register from the *free list*, which holds a list of physical registers that are currently unused. The new register mapping is then written back to the mapping table to reflect the new mapping.

When a micro-op reaches the commit stage, there are two possibilities: the micro-op is either successfully committed or discarded. Each successfully-committed micro-op updates the committed mapping table with the same mapping as it did for the speculative mapping table, then discards the physical register that held the *previous* value of the destination logical register, by enqueuing the discarded physical register number on the free list. Discarded micro-ops do the opposite: a discarded micro-op does not update the committed mapping table and discards the physical register that was allocated to hold the *new* (now discarded) destination logical register. Pipeline flushes cause the speculative mapping table to be rolled back to the committed state by copying from the contents of the committed mapping table.

7.1.1 Comparison to Reorder Buffer Organization

There are many variations in the organization of the register files and register renaming used in previous processors. This section will briefly compare the PRF scheme to reorder-buffer-based renaming, a scheme that uses a reorder buffer (ROB) that holds speculative register values. While many alternatives and variations exist, the ROB-based renaming scheme is notable for being used on a series of Intel P6-based processors from P6 through Nehalem [103, 123, 131].

Figure 7.1b illustrates the ROB-based renaming scheme. For ROB-based renaming, register values are held in three structures: Speculative register values are stored in the ROB, committed register values are stored in the committed register file (which Intel calls the Architectural Register File, or ARF), and register values used as source operands are stored in reservation stations while micro-ops await execution.

The ROB-based scheme is simpler in several aspects. There is only one (speculative) register mapping table, and pipeline flush recovery is done by resetting the mapping table to point every entry to the committed register file without copying table contents. In contrast, our PRF design requires copying the committed mapping table to the speculative mapping table. The ROB is a circular queue holding speculative micro-ops and their results in program order, so entries (thus, speculative registers) are also allocated and deallocated sequentially, which is simpler than managing free registers using a register free list. A PRF organization must use a free list because any physical register may become committed state and persist indefinitely, leaving occupied "holes" in the register file that cannot be used for new speculative state, while a ROB entry is always freed when a micro-op is committed or discarded.

However, it is this free-on-commit behaviour that makes the ROB organization unattractive for FPGA implementations, for (at least) two reasons. First, the destination register for every micro-op needs to be read out of the ROB and written into the committed register file when it is committed, which requires an extra two read and two write ports for a two-issue design, increasing the number of ports in *two* multiported RAM structures (the ROB and committed register file). Second, the possibility of a register value changing its physical location from the ROB into the committed register file during commit — *while* a consumer micro-op may want to read that same operand from the ROB entry — strongly encourages a data-capturing reservation station scheduler design. When using reservation stations, register values in the ROB or committed register file are copied to reservation stations (a CAM) during register renaming, ensuring later changes to the physical location of a register value do not affect micro-ops already waiting in reservation stations. Reservation stations that hold register values in a CAM are undesirable because CAMs are area-inefficient on FPGAs. Alternatives to using reservation stations include finding all consumer micro-ops waiting in the instruction schedulers and updating their source register to point to the committed register file whenever a micro-op commits, or stalling instruction commit while there are still pending consumers of a ROB entry, both of which are complex to implement. A PRF organization allows more-efficient data-less instruction schedulers that are followed by reading the source operands from a physical register file (RAM) in the next cycle.

Although slightly more complex in design, we prefer a PRF-based scheme because it reduces the number of ports in the multiported register files and reduces the size of the instruction scheduling CAMs.

7.2 Proposed Renamer and Register File Design

This section discusses further design features and optimizations beyond the basic PRF scheme described above.

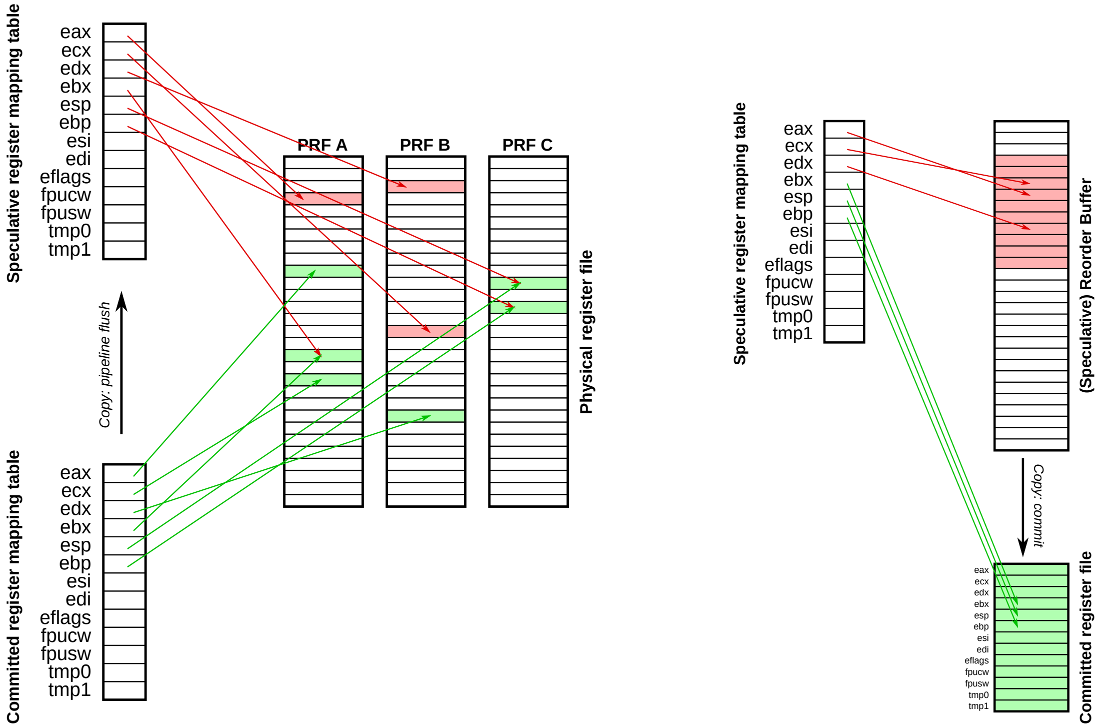
The microarchitecture simulation experiments in this chapter uses the same workloads as in the previous chapter (See Section 6.1): SPECint2000, MiBench, Dhrystone, CoreMark, Doom, and Windows XP. This subset of the workload has about 30.2 billion instructions.

7.2.1 Using Renaming to Simplify Multiported Register Files

In a multi-issue processor, multiple instructions are executed each clock cycle, requiring multiported register files. We can extend the register renaming mechanism to further reduce the complexity of the multiported physical register files by tracking multiple pools of registers instead of one unified pool.

In the simplest case, physical register numbers represent entries in a single multiported register file. Alternatively, a non-uniform physical register file can be used to allow flexibility in how the physical register file is structured. One obvious way is to split the register files by data type: integer and floating-point registers are often implemented as two separate register files, and the renamer tracks separate pools of registers. In our x86 implementation (which has no floating point support), we rename three separate types of registers: 32-bit integer registers, 6-bit condition codes, and segment registers. Rarely-modified registers such as control and debug registers are not renamed.

Allowing the renamer to track multiple pools of registers can also be used to reduce the complexity of the multiported physical register files. In the simplest case, a register file for one data type (such as integer registers) is treated as a single multiported structure, which requires multiple read and write ports for multiple execution units executing multiple instructions per cycle. As we saw from Section 3.4.3, using multiple write ports is far more expensive than using multiple read ports. While we cannot easily reduce the number of register file read ports, our design uses register renaming to eliminate multiple write ports by giving each execution unit its own physical register file and requiring the register renamer to track multiple pools of registers (one per execution unit). The cost of this scheme is a larger physical register number to indicate the register file (two extra bits for 3 execution units), the need for multiple physical register free lists, and the need to assign micro-ops to register files at the rename stage of the pipeline, removing the possibility of execution unit load balancing later in the pipeline. Conceptually, using the renamer to track multiple register files behaves similarly to a live value table multiported memory [72] but with the live value table functionality merged with the register renamer and accessed many cycles earlier than the data access.



(a) Physical Register File (b) Reorder Buffer

Figure 7.1: Physical register file (a) and reorder buffer (b) renaming schemes. PRF stores both speculative and committed values in a (banked) register file. ROB renaming uses the ROB to hold speculative values, and a separate committed register file. We omitted some arrows for clarity.

7.2.2 Misspeculation Recovery

Due to the possibility of misspeculation, there is a need for a mechanism to recover the renamer state to an earlier point in time. There are many methods to do this recovery, trading off the time needed to recover with hardware complexity. Common methods of recovering from a misspeculation include reconstructing the speculative register mapping state by traversing the reorder buffer in reverse order to undo speculative mappings; maintaining a committed mapping table and copying it to the speculative renamer table (effectively traversing the reorder buffer in the forward direction to construct the committed state); and maintaining several checkpoints of the mapping table's state to allow instantaneous (single-cycle) recovery from a checkpoint. As mentioned earlier, we use the second method of copying committed register mappings.

Due to its ability to do instantaneous recovery, checkpointing has been used in commercial processors [105], studied [132, 133], and even proposed for use in an out-of-order FPGA soft processor [40]. Our design uses a committed mapping table for recovery rather than checkpoints, as checkpoints are complex and expensive to implement and, at least in our design, has a limited potential for improved performance. However, checkpointing deserves discussion as earlier work often considers checkpoints, and increasing the number of checkpoints, to be useful.

In a checkpointing-based system, checkpoints of the register renamer tables are made periodically, but doing so for every instruction or micro-op has an excessive storage overhead. Thus, checkpoints are only used to recover from branch mispredictions (which are relatively frequent) but not exceptions. This adds complexity, as there always needs to be another recovery mechanism to handle recovery for exceptions. In a classical checkpointing system, such as the one used in the MIPS R10000 [105], a complete copy of the register renamer state is made at every branch instruction. The finite number of checkpoints available imposes a limit on the number of unresolved branches that can be supported (4 branches in the R10000), which causes performance loss when there are too few checkpoints. More recent work improves on this restriction by making checkpoints less frequently than at every branch, such as only for low-confidence branches that are more likely to need the checkpoint [132, 133]. In contrast, doing recovery by maintaining a committed mapping table imposes no limit on the number of outstanding branches, which avoids the performance loss associated with having too few checkpoints, in exchange for the possibility of recovery taking multiple cycles.

Figure 7.2 shows a plot of the cycle-cumulative behaviour of outstanding branches on the 30-billion instruction workload used in this chapter. A cycle-cumulative plot shows the cumulative number of cycles using no more than the amount of the resource shown on the horizontal axis. For example, Figure 7.2 shows that 82% of processor cycles are spent with 5 or fewer branches in between the rename and commit stages (i.e., in the ROB). This plot shows that it will likely take around 6 (88% of cycles) to 8 (97% of cycles) checkpoints at branches (assuming without the extra complexity of branch confidence prediction) to make the maximum-outstanding-branches limit have a negligible performance impact, if used on our design. This result corresponds well with the number of checkpoints used in previous work [40].

The two main advantages of using a committed mapping table for recovery rather than checkpoints are that a single mechanism can be used for pipeline flushes caused by both branch mispredictions and

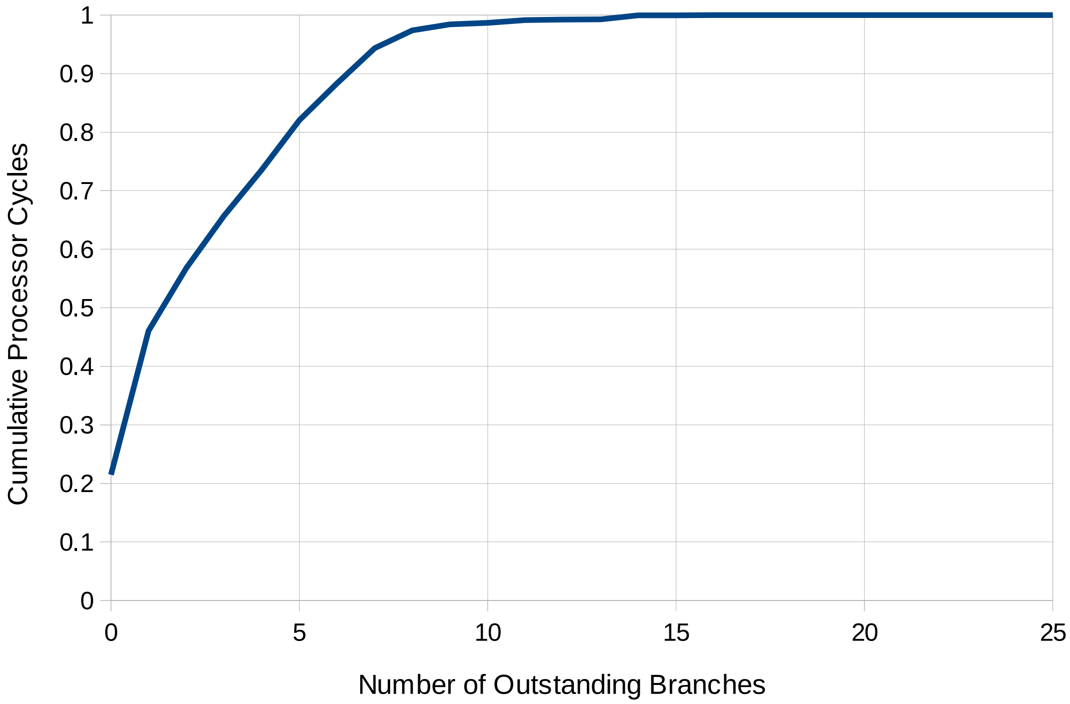


Figure 7.2: Cycle-cumulative behaviour of outstanding branches.

exceptions, and it does not impose a limit on the number of outstanding branches. Its main drawback is the inability to guarantee instantaneous recovery, as the misspeculated instruction (usually a branch) needs to be committed before the register mapping table can be recovered. This delay in renamer recovery can impact performance if corrected-path instructions are fetched, decoded, and are then stalled at rename because mapping table recovery has not yet occurred. In our design, the number of cycles stalled at rename while waiting for a misspeculated branch to commit is 2.3% (or 4.2 clocks per pipeline flush)[2], which is an upper bound on the amount of performance that could be gained if recovery were instantaneous. This is an upper bound because the processor is not necessarily idle in those 2.3% of cycles, as it can still be doing useful work for instructions before the mispredicted branch that have not yet executed.

We speculate two reasons for the small performance impact for not having instantaneous checkpoint recovery. First, to avoid the complexity of repeatedly flushing the pipeline when a series of mispredicted branches are executed out of order, we execute branches in program order. This means that branch mispredictions tend to be detected later (closer to commit) than if full out-of-order branch execution were permitted. Second, our processor, like many modern processors, has a long front-end pipeline (about 9 cycles from fetch to rename), which delays the arrival of corrected-path instructions. A delay in mapping table recovery has an effect only if the delay between a mispredicted branch's execution and commit is longer than fetching and decoding new instructions along the corrected path — otherwise, the speculative mapping table would be recovered earlier than the arrival of new instructions, resulting in no extra

[2] This stall while waiting for mapping table recovery appears to be the same stall measured by the INT_MISC.RECOVERY_CYCLES performance counter on Intel Haswell, Broadwell, and Skylake processors. Haswell measures around 4 to 6 clock cycles per pipeline flush. See Appendix E for a comparison using CoreMark.

stalls.

The extra complexity and the small potential performance improvement of using checkpoints for faster branch mispredict recovery leads to our choice to not use renamer checkpoints. In addition, actually taking advantage of the faster recovery requires further complexity in other parts of the processor. If the processor allows new corrected-path instructions to enter the out-of-order portion of the processor (instruction schedulers) while micro-ops before the mispredicted branch are still in-flight, the processor must support *partial* flash-clearing of the instruction schedulers to selectively remove wrong-path micro-ops without removing correct-path micro-ops — in other words, correct-path micro-ops before the branch, wrong-path micro-ops after the branch, and new *corrected*-path micro-ops must coexist in the instruction scheduler, with only the wrong-path micro-ops flash-cleared from the schedulers. The MIPS R10000 appears to use this method [105]. In our current design, we flash-clear[3] the entire instruction scheduler when the branch commits, as it can be guaranteed that immediately after the mispredicted branch commits, no corrected-path new micro-ops have yet entered and all committed micro-ops have already left the out-of-order portion of the pipeline.

7.2.3 Sizing the Physical Register Files

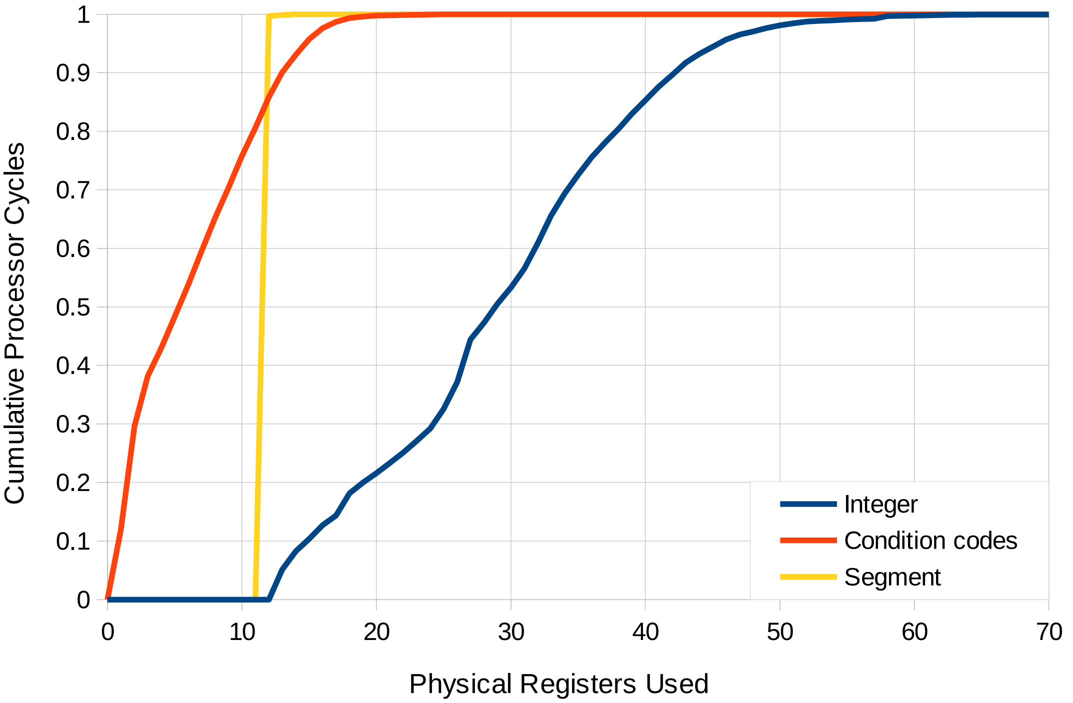


Figure 7.3: Cycle-cumulative behaviour of physical register usage, for three types of registers.

Part of the register renaming design problem includes choosing the sizes of the physical register files. The physical register files must always be larger than the logical register file, as every logical register oc-

[3] Not supporting flash-clearing of schedulers results in schedulers being clogged with wrong-path micro-ops, which greatly increases the effective branch misprediction penalty, especially for larger schedulers that take longer to drain.

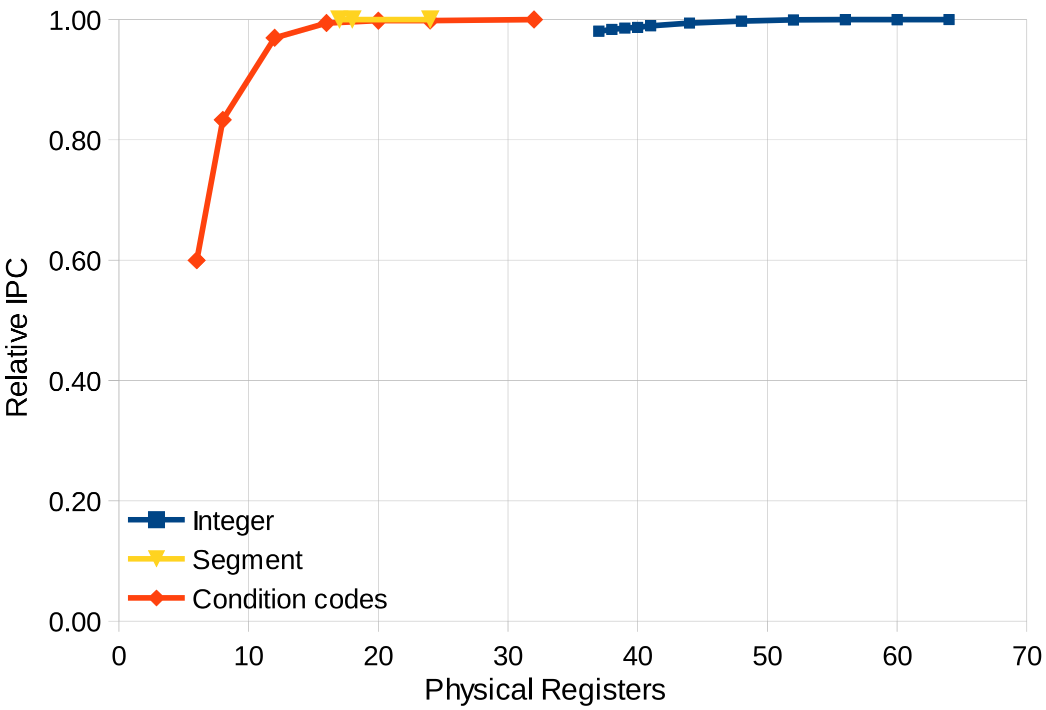


Figure 7.4: IPC impact vs. physical register file capacity, for three types of registers.

cupies one physical register to hold its committed state, in addition to any speculative instances. In our design, the minimum size of the integer general-purpose physical register file is 41, to hold 13 committed registers and 28 speculative registers used by the worst-case micro-op sequence that needs to be atomically committed (42 micro-ops). Figure 7.3 shows the cycle-cumulative behaviour of physical register use, when the number of in-flight micro-ops is limited only by the reorder buffer size (64 outstanding micro-ops). Integer registers, unsurprisingly, have high demand (51% of cycles are spent using at most 29 integer registers, and 99.9% of cycles using 64 or less). There are 13 logical integer registers, and most instructions write to the integer registers and create speculative instances. Condition codes are also frequently used (98% of cycles using at most 16), despite there being only one committed logical register. Segment registers are rarely modified, so although there are 12 physical segment registers used to hold committed state, very few speculative registers are used (99.7% of cycles have exactly 12 segment registers used). Based on this data, we believe 64 integer registers, approximately 16 segment registers, and 16 condition code registers is a good design point.

Figure 7.4 shows how cycle-cumulative register use translates into IPC when each register file is made smaller. Limiting condition codes to 16 entries reduces IPC by 0.6%. The general-purpose integer and segment register files have almost no IPC impact, as the minimum size to avoid deadlock is higher than the size at which there is appreciable IPC loss. This experiment used the workload from Section 6.1 with Windows XP omitted (25.0 billion instructions). Our processor stalls whenever there are inadequate free registers to serve the worst-case pair of micro-ops that could be renamed in a cycle, without examining whether the micro-ops being renamed actually require the registers. This simplification slightly increases physical register demand, but simplifies the stall logic.

7.2.4 x86-Specific Design Issues

In our design, x86 instructions decode into fairly complex micro-ops. A single micro-op can encode x86-style load-modify-store operations. As a result, a micro-op can have a large number of operands. A further complexity arises from partial register and flag writes because many instructions modify only some of the bits of a register (e.g., arithmetic instructions with 8- or 16-bit operand size).

In our design, micro-ops are capable of encoding a load, arithmetic operation, and a store, which would usually require at least 3 RISC-style instructions (depending on addressing mode). Thus, there can be a large number of renamed operands: the most complex micro-op can use 3 source and 2 destination integer registers (general-purpose registers, or GPRs), 2 source and 1 destination segment register, and 1 source and 1 destination condition code register. For load-modify, modify-store, and load-modify-store micro-ops, the load, modify, and store operations execute independently in different execution units. For example, a load-modify-store micro-op will be broken into three operations: An address computation and load sent to the AGU (address generation unit), an arithmetic operation sent to the ALU that consumes the loaded value, and a store-data operation sent to the store data unit that consumes the ALU result. The data dependencies are communicated through two registers: one for the load result, and one for the ALU result. The renamer is responsible for rewriting the intra-micro-op data dependency to communicate the data value through a register. The renamer is designed to rename two micro-ops every cycle with no restrictions on complexity.

While it would appear that renaming two micro-ops each cycle would require a GPR mapping table to have four write ports (because there are two micro-ops with two destination registers each), only two write ports are actually needed. Complex micro-ops contain *internal* data dependencies that cannot be seen by other instructions, and a micro-op can never have two architecturally-visible destination operands. Since register mapping table write ports are needed only for destination registers that are architecturally visible, two micro-ops only need two mapping table write ports, despite using up to 4 destination registers. Indeed, the most complex load-modify-store type micro-op has two destination operands but does not modify the mapping table at all, because a "memory" destination operand does not modify any architectural registers.

Each micro-op may need up to two segment register source operands. Every memory access uses a segment register (to get the segment base and limit used for the memory access), and micro-ops that manipulate the segment registers may need to access a second segment register. There are 12 segment registers mapped to 16 physical segment registers, leading to a mapping table of 12×4 bits.

Condition codes are a subset of bits from the 32-bit EFLAGS register (We also call the condition codes OCZAPS, named for the 6 flags). For performance reasons, EFLAGS cannot be treated as a single register, as the condition codes are performance-sensitive, but some of the other bits are necessarily slow because they have side effects with global impact (e.g., I/O privilege level). We split the 32-bit EFLAGS register into a 32-bit GPR and 6 bits of condition codes. Operations that read or write to the entire EFLAGS use a micro-op to merge the condition codes into the 32-bit GPR before reading it, or set the condition codes after modifying EFLAGS. For all other operations, these two parts are not kept synchronized. As there is only one condition code register, the mapping table contains only one 5-bit entry (two banks of 16 physical registers). Although conceptually the mapping table should have two read and two write ports, since it has only one entry, it simplifies to one set of flip-flops with arbitration to decide which micro-op should write to the entry.

One design issue that is unique to x86 is the handling of partial register writes. A partial write occurs when a narrow 8-bit or 16-bit instruction writes to part of a register and leaves the rest of the register unchanged. This can cause problems if the full-length register is consumed by a later instruction, as a single register would contains bits that were produced by two different instructions. There are several options to handle this, and we chose the simplest (but lowest-performing) option. The simplest method is to merge the unmodified bits of the register with the computation result after each micro-op *executes*, which serializes (often unnecessarily) chains of narrow instructions that access the same register. Other more complex solutions require the involvement of the register renamer. Instead of merging partial writes during execution, they can be merged at *commit* as on the Intel P6 (causing a "partial register stall" if the merged value is consumed too soon), merged when necessary with an extra micro-op as on newer Intel processors, or merged immediately *before* execution by treating each source register as three source operands (very expensive). An analogous issue occurs with condition codes ("partial flag write"), as not all 6 condition codes are modified by all instructions.

We also made the register renamer handle tracking the Resume Flag (RF) state, as our method to track the RF state behaves like a one-bit flag that is recovered along with the register mapping tables during pipeline flushes. The RF flag is used by debuggers to suppress a debug fault that would otherwise be generated by single stepping or instruction breakpoints. The RF flag is set by software and cleared by the processor after every instruction is executed, except for a few cases (e.g., an IRET instruction).

7.3 Detailed Hardware Design

Since our processor is designed for a peak throughput of two micro-ops per cycle, the renamer is designed for the same bandwidth. To verify this choice, we simulate the processor's IPC performance when varying the renamer throughput from 1 to 4 micro-ops per cycle. Figure 7.5 shows that reducing the renamer throughput to 1 micro-op per cycle incurs a 19% drop in IPC, but there is no gain when increasing throughput to 3 or more micro-ops per cycle. Thus, for our current processor design, renaming two micro-ops per cycle is a good design point. In this experiment, the renamer throughput affects the rate at which micro-ops are dequeued from the post-decode micro-op queue (but leaves decode width unchanged at 2 instructions/cycle), and also the rate at which instructions are issued from rename into the schedulers.

Figure 7.6 shows the design of our two-pipeline-stage register renamer. The first cycle selects the micro-op source (up to two micro-ops from regular decode, string operation, or microcode unit), then computes what operands need renaming and the dependencies within the two micro-op group. This stage generates all of the multiplexer select signals used in the second stage, including deciding whether

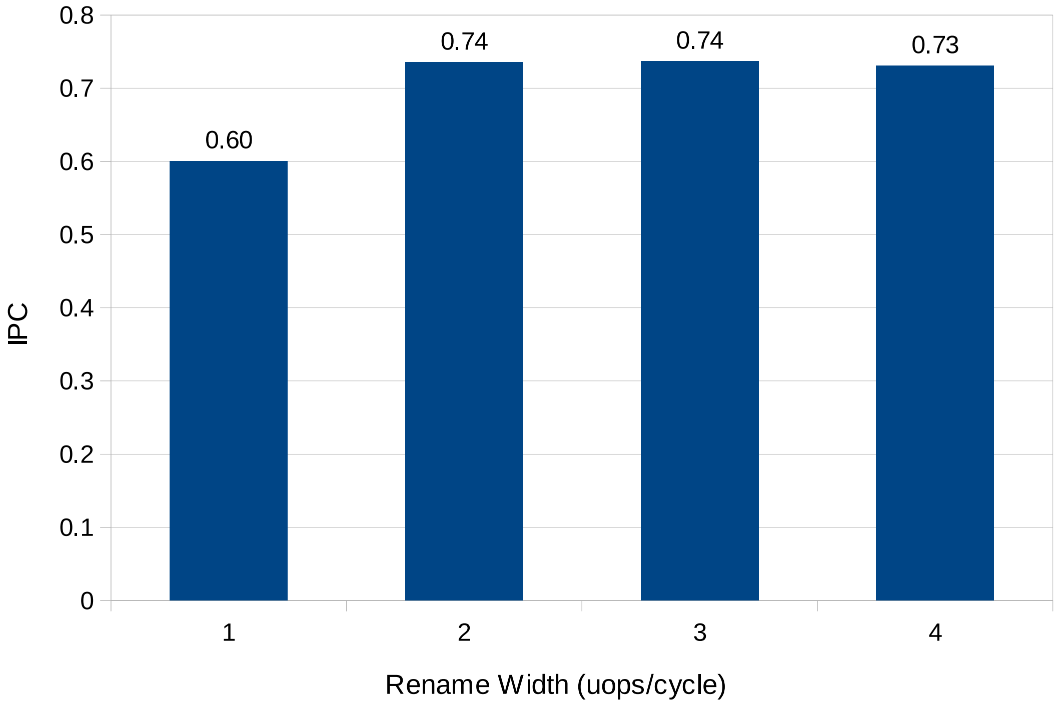


Figure 7.5: Performance at varying renamer throughput

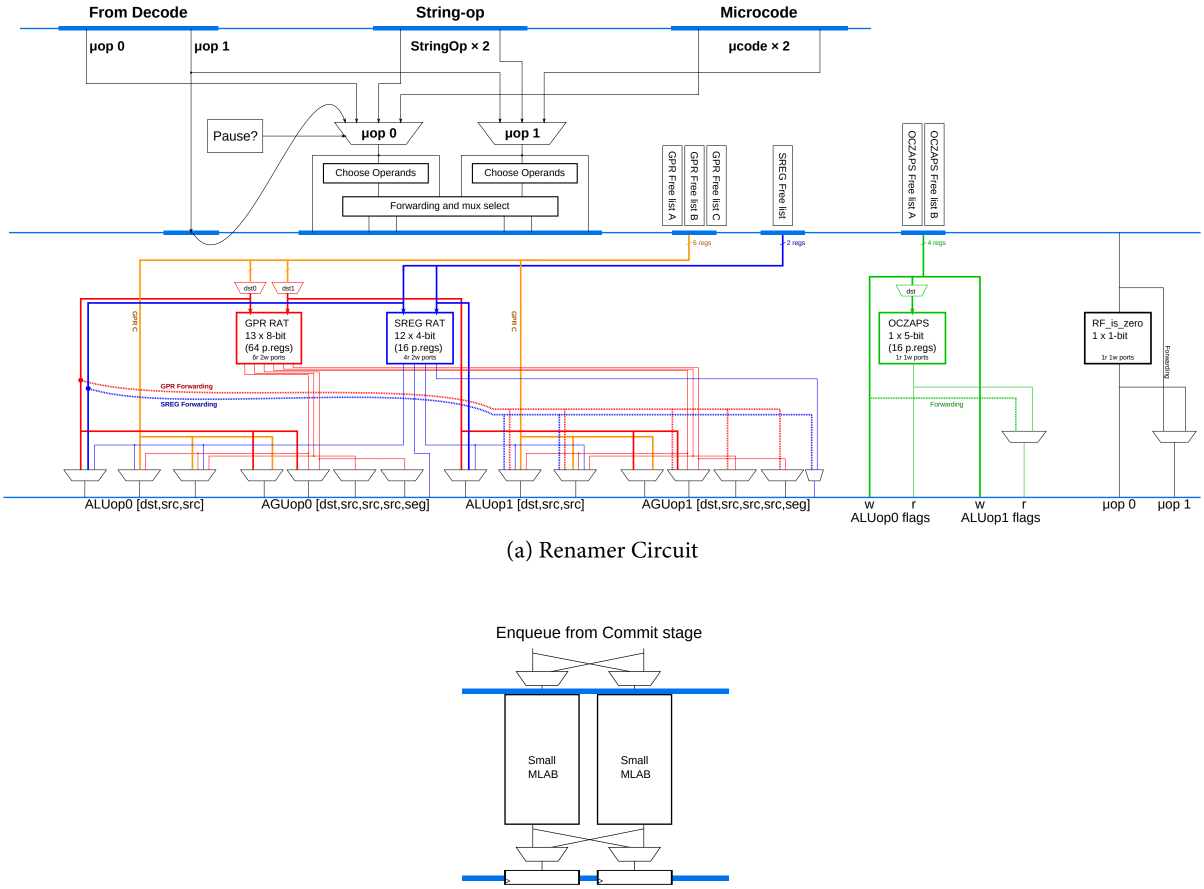
to forward the first micro-op's destination to the second micro-op's source(s) if there is a data dependency.

The second stage reads the register mapping tables, acquires new registers from the free lists, routes the output of each read port of each RAT to the right operands, and performs forwarding if there are dependencies between the two micro-ops renamed this cycle. To support renaming (and committing) two micro-ops per cycle, each free list is a (not necessarily FIFO-ordered) queue that supports enqueuing and dequeuing two registers per cycle, composed of two simple FIFO queues and steering logic (Figure 7.6b). The outputs of the second stage are separate ALU and AGU operations and their physical register numbers, with any internal data dependencies being represented by having the producer operation write to a physical register and reading it as a source operand in the consumer.

Given the large number of ports (10r 4w for just the GPR and segment registers alone) and the need for low-latency single-cycle updates, the register mapping tables are implemented in soft logic (LUTs and flip-flops). Fortunately, x86 has a fairly small register context (in our microarchitecture, only 13 logical GPRs and 12 logical segment registers), so the total mapping table state is only about 160 bits.

7.4 Hardware Frequency and Area

We have implemented the register renaming unit for a Stratix IV FPGA. It achieves 317 MHz (median of 31 random seeds), exceeding our target of 300 MHz. It uses 1892 ALMs (Table 7.1 shows a resource usage breakdown). The renamer is unusual in using more ALMs than ALUTs, because of the large number of pipeline registers used in the design (2747). Registers are included in the ALM count, but not ALUTs,



(a) Renamer Circuit

(b) Free List Circuit

Figure 7.6: Two-stage renamer circuit. Each free list is a two-enqueue two-dequeue queue.

Unit	ALUT	Registers	ALM
Free lists	521	409	398
First stage	550	167	542
Second stage	721	157	558
Total	1813	2747	1892

Table 7.1: Resource consumption of hardware implementation of register renamer

which counts only logic functions.

The hardware includes everything in Figure 7.6. It does not include the committed mapping table, which is located with the instruction commit hardware. The committed mapping table is a simplified version of the speculative mapping table, containing only the write ports and none of the read ports, so we expect it to be much smaller and somewhat faster than the renamer's second stage. Logic to determine the readiness of each operand is also not included, as it is part of the instruction scheduler, and the design of the readiness check logic depends on what type of instruction scheduler is used. For example, a matrix-based scheduler not only needs to know whether a source operand is ready, but also which entry in the scheduler will produce the result. We discuss instruction scheduling in Chapter 9.

7.5 Summary

The register renamer used in our processor is designed to rename two micro-ops per cycle, with no corner cases even for complex micro-ops such as load-modify-store operations. The two pipeline-stage circuit achieves 317 MHz, which exceeds our target frequency of 300 MHz, and uses under 2000 ALMs. We use a physical register file organization to reduce the number of ports used by multiported RAMs, and use one bank of registers per execution unit to eliminate multiple write ports.

After renaming, the micro-ops are ready to be executed, in dataflow order. Renamed micro-ops are sent to the instruction schedulers (Chapter 9) to be scheduled for execution, and also to the reorder buffer (Chapter 8) that tracks the status of each micro-op to enable in-program-order instruction commit.

Chapter 8

Reorder Buffer and Instruction Commit

In an out of order processor, instruction execution occurs in dataflow order, but both the front-end and instruction commit occurs in program order. To keep track of the program order of instructions while they execute out-of-order, a reorder buffer (ROB) is used. A ROB is a circular queue that holds micro-ops in program order and tracks their status while they execute so that they can be committed in-order once completed. Micro-ops are inserted in program order into the reorder buffer after register renaming, and completed micro-ops are dequeued in program order to be committed.

This chapter discusses both the ROB and instruction commit. Although these seem far apart in pipeline stages, they are related in function. The ROB serves to preserve micro-op ordering to serve the in-order instruction commit unit.

Figure 8.1 illustrates the ROB and commit unit in the context of the processor pipeline. The ROB is a fairly straightforward circular queue of micro-ops. Renamed micro-ops are enqueued into the ROB after register renaming. Micro-ops are also simultaneously enqueued in the instruction schedulers so the micro-ops can be scheduled and executed. The status of each micro-op after execution (e.g., whether there are any exceptions) is written back to the ROB. Instruction commit is more complex. Committing micro-ops involves updating the committed register renamer mapping, freeing resources (e.g., physical registers), and writing the data associated with store micro-ops to cache. In addition, it must handle atomic commit (committing none or all of the micro-ops belonging to an instruction), and detect and handle several different types of pipeline flushes.

The following sections will describe these functions in more detail. We will then present some performance results related to pipeline flushes.

8.1 Reorder Buffer

The reorder buffer (ROB) is a circular queue that holds micro-ops in program order, enqueued after renaming, and dequeued for commit. Because we use a physical register file organization (Chapter 7), the ROB does not hold speculative register values used for instruction execution. Each ROB entry holds information necessary for the micro-op to be committed. This includes fields such as the old and new destination register physical numbers (to free when committed/discarded), instruction pointer (for ex-

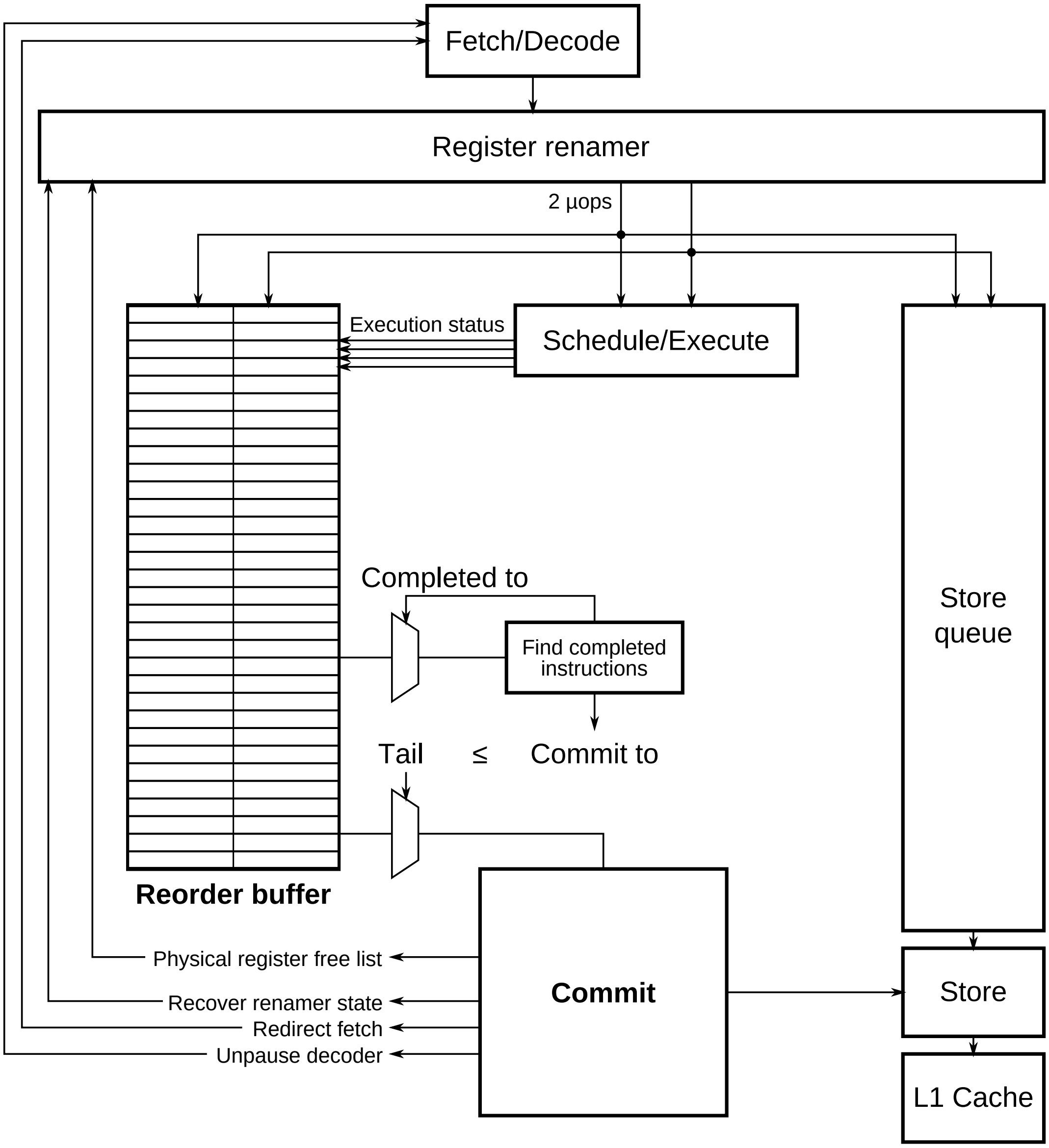


Figure 8.1: Reorder buffer and commit. A ROB read port searches completed micro-ops for instruction boundaries to compute a commit_to pointer that controls micro-op commit. Commit frees resources, flushes various parts of the pipeline, and controls when stores are written to cache.

ceptions), and the execution status of the micro-op (whether is has executed, and whether it is successful or caused an exception). Out of the many ROB entry fields, only the execution status field is modified during execution, thus most of the ROB storage can be implemented in a RAM as a simple FIFO queue, with only the status bits implemented in soft logic.

The RAM portion of the ROB needs to support enqueuing and dequeuing two micro-ops per cycle in FIFO order. This can be implemented as a pair of simple one-enqueue-one-dequeue FIFOs with some steering multiplexers. The steering multiplexers can even be omitted if the ROB is allowed to have unused holes, trading less efficient utilization of the ROB for simpler logic. This has the potential to work well because for cycles where one or more micro-ops are renamed, 93% of these cycles have two renamed micro-ops (i.e., 7% of cycles produce one hole, leading to an average 96.5% utilization of the ROB, reducing the effective ROB size from 64 to 61.8 entries). We have not investigated this design further.

The status fields require more read and write ports. Status bits are enqueued and dequeued along with the rest of the ROB entry. They are also updated when each sub-operation (address generation/load, modify, or store data) of each micro-op is executed, requiring 4 updates per cycle (one for each of the four execution units in our processor). The status bits are also examined by the commit unit when searching the ROB for completed instructions (by looking for instruction boundaries) to determine which micro-ops that can be committed, and examined by the commit unit when committing a micro-op.

The next section discusses the complexities of committing micro-ops once they leave the reorder buffer.

8.2 Committing Micro-Ops

Micro-ops are dequeued in program order from the reorder buffer when they have completed and are ready to be committed. Conceptually, the oldest micro-op in the ROB can be committed whenever it has completed executing. Each micro-op can have two outcomes: either it successfully executed and needs to be committed, or the micro-op has been squashed and its effects discarded. The commit unit updates the committed state of the processor (if the micro-op is not squashed), and frees resources associated with each micro-op. For micro-ops that caused a branch misprediction or exception, the commit unit is also responsible for orchestrating the pipeline flush: redirecting the fetch unit, rolling back the register renamer state, and clearing out subsequent wrong-path instructions from the instruction schedulers.

The commit unit turns out to be a surprisingly difficult piece of hardware to design. One source of complexity lies in the need to commit instructions atomically: either none or all of the micro-ops for an instruction are committed. The more difficult task is orchestrating a pipeline flush, due to the explosion of interactions between the several different types of pipeline flush and the state of each processor component that needs resetting when the pipeline flush occurs. Any error easily results in squashed micro-ops being incorrectly committed, non-squashed micro-ops incorrectly discarded, new corrected-path micro-ops being discarded, old micro-ops being trapped in the pipeline (waiting forever to execute), new and old micro-ops being reordered, and many other types of deadlocks and failures. We discuss some of these issues in the following subsections.

8.2.1 Atomic Commit

For processors that use precise interrupts (i.e., essentially all general-purpose processors), instructions are considered atomic: an instruction is either entirely completed or discarded. When implementing instructions using micro-ops, this implies that the entire micro-op sequence for an instruction must also be completely committed or discarded. As mentioned in Section 5.3, breaking instructions into multiple micro-ops is not x86-specific, and is also used in ARM and Power processors.

The simple rule mentioned above of committing all completed micro-ops without regard to instruction boundaries does not satisfy the atomicity requirement if an instruction containing multiple micro-ops successfully completes at least one micro-op but has an exception on a subsequent micro-op. An exception should cause the entire instruction to be discarded, but the exception may not be discovered until some micro-ops from that instruction have already been committed, preventing those micro-ops from being discarded.

We handle atomic commit by first scanning the completed micro-ops in the reorder buffer to look for instruction boundaries, but without committing those micro-ops yet. Only once *all* of the micro-ops for an instruction are completed will permission be granted for the entire instruction to be committed. As the commit unit scans the completed micro-ops, it increments the "completed to" pointer (See Figure 8.1). The "commit to" pointer points to the most recent instruction boundary seen, which guarantees that partial instructions are never allowed to commit. The commit_to pointer limits how far micro-op commit may proceed (the tail pointer must not exceed the commit_to pointer). It is the micro-ops at the tail pointer that are committed, one or two micro-ops at a time.

Figure 8.2 illustrates how the commit_to and completed_to pointers are used to ensure atomic instruction commit. In this example, the `call` and `ret` instructions are instructions that use multiple micro-ops, and must be atomically committed. The completed_to pointer advances until it points to the first incomplete micro-op. In this example, the `call` instruction is partially completed (two of three micro-ops complete). The commit_to pointer points immediately after the instruction boundary preceding the completed_to pointer. This is accomplished by setting commit_to pointer to the current completed_to pointer value whenever the completed_to pointer crosses an instruction boundary. The commit_to pointer then ensures that none of the `call`'s micro-ops can commit until all of its micro-ops have completed and the completed_to and commit_to pointers reach the next instruction.

To ensure exceptions are detected, micro-ops that cause an exception update the exception status for the *first* micro-op of its instruction rather than for the micro-op itself. This combination of using pointers to delay commit and setting exception status for the first micro-op of an instruction guarantees that all micro-ops for an instruction have completed before the first micro-op is examined for commit, and that if any micro-op in an instruction causes an exception, it is detected when examining the first micro-op. This results in discarding either none of an instruction's micro-ops, or all of an instruction's micro-ops (if any micro-op causes an exception).

When using atomic commit, the ROB must be large enough to hold all of the micro-ops for an atomically-committed micro-op sequence. This can be problematic for long sequences. The longest such sequence in our current design is 42 micro-ops (to handle an interrupt from virtual-8086 mode

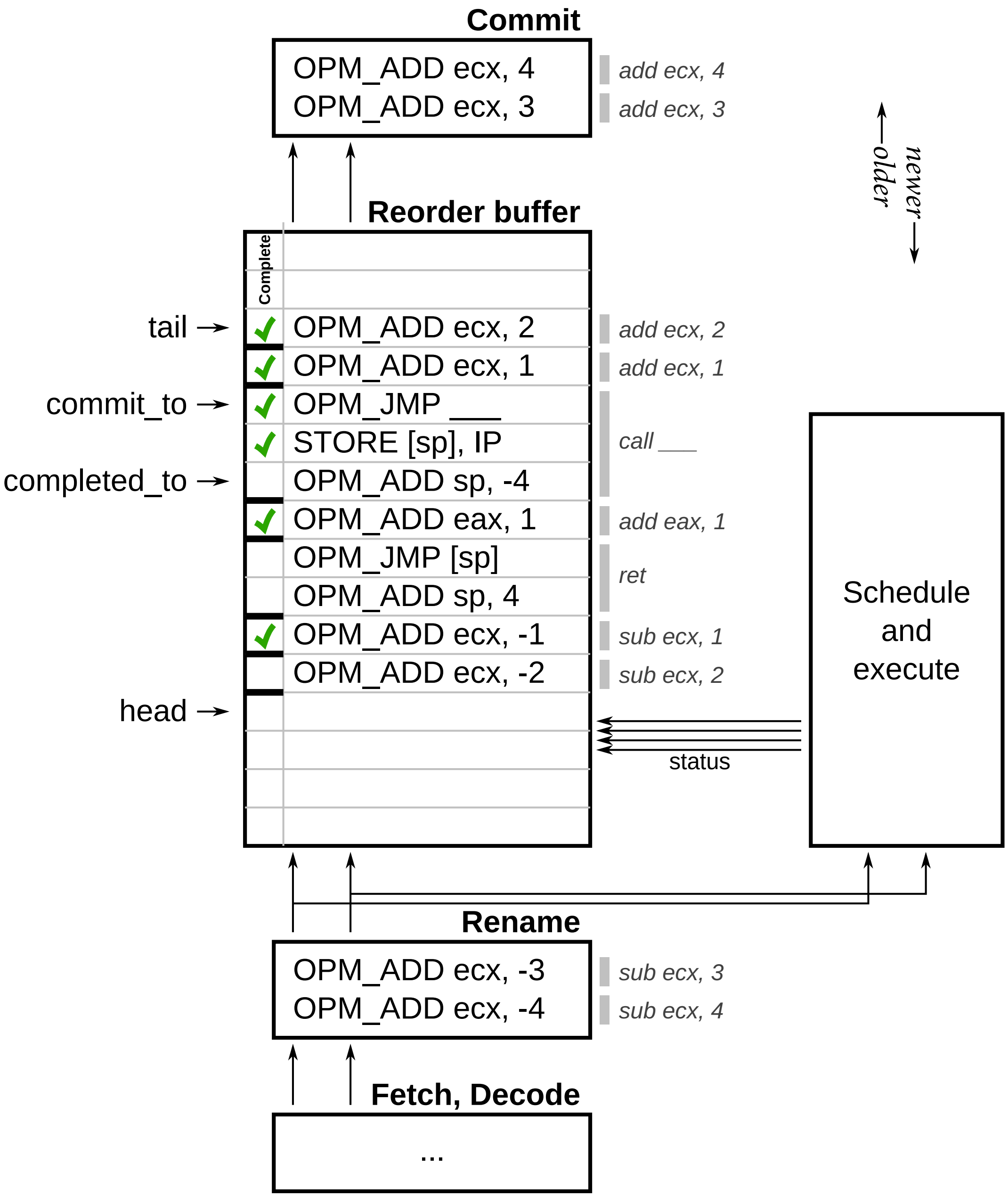


Figure 8.2: Implementing atomic commit: commit_to pointer demarcates completed *instructions*, while completed_to pointer demarcates completed *micro-ops*. In this example, the *call* instruction is partially complete, so cannot be committed yet.

that changes privilege levels), which also mandates that the ROB capacity must be at least 42 micro-ops to avoid deadlock. The deadlock occurs because the first micro-op cannot commit until the last micro-op has executed, but the last micro-op cannot execute until some earlier micro-ops have committed to make space in the ROB for it. This is not an issue from a performance perspective, as these very long micro-op sequences are rare (From Section 6.2, the average micro-ops per instruction is 1.12). The minimum ROB size requirement also did not affect our design because we were targeting 64 ROB entries for performance reasons anyway, but it may become a significant problem for a smaller, lower-performance processor design.

An alternative method to handle atomic commit is to design all micro-op sequences so that all possible exception conditions are checked for before any processor state changes occur, so that partially committing an instruction results in no visible effects. We did not attempt this method for the current design because designing micro-op sequences (especially for handling exceptions) is already complicated enough without the added requirement of anticipating all possible exception conditions.

The hardware cost of this atomic-commit scheme is fairly small. The main cost is in providing a read port to read the "completed" status bit out of the ROB (a one-bit wide 64-to-1 multiplexer).

8.2.2 Handling Pipeline Flushes

If an exceptional condition is encountered during instruction commit, the commit unit is responsible for performing the pipeline flush. The complexity comes from having several types of exception-like conditions that are handled differently, and the large number of processor components that need resetting. Any error in any of the many corner cases can result in deadlock or committing or discarding the incorrect micro-ops.

Pipeline flushes have many causes. In addition to branch mispredictions, interrupts, and exceptions (faults and traps), pipeline flushes also occur after some instructions that change non-renamed processor state such as control registers, if a string instruction terminates earlier than the expected count (for string compare and scan), if self-modifying code is detected, and when one of the other predictions fail (memory ordering violation, descriptor table predictor). If more than one of these conditions are true for an instruction, there is a prescribed priority defining which event should be handled. Each one of these causes is handled slightly differently.

For all pipeline flushes, the register renamer's committed mapping table is copied to the speculative mapping table to roll back the register renamer state. Some of these events will redirect the front-end to begin fetching from a new location (usually to re-fetch the same instruction), while some do not (branches, interrupts, faults, traps). Branches do not redirect the front-end because, unlike all of the other events, redirecting the fetch unit was done earlier during branch execution to reduce the branch misprediction penalty. Interrupts, faults, and traps do not redirect the front-end. Instead, they trigger a microcode sequence to process the exception, and the microcode sequence contains a branch micro-op that branches to the start of the exception handler. A pipeline flush requires discarding subsequent wrong-path micro-ops, but some events discard the current instruction too (interrupts, faults) while others discard starting with the next instruction (branches, traps).

In addition to all of the ways a pipeline flush can be triggered and ways it may need to be handled, processor components can be in different states when the pipeline flush occurs. One example corner case involves an exception occurring within a microcode sequence (e.g., processing an exception). A microcode sequence is atomically committed, so none of the micro-ops in that sequence are committed (or checked for exceptions) until all micro-ops in the sequence have completed. However, an exception within the sequence may cause the microcode sequence to be aborted before the entire microcode stream is generated, leading to deadlock unless this case is specially handled[1]. These corner cases are rare (this example occurred 26 times in 30 billion instructions), but must be handled correctly.

8.3 Performance

8.3.1 Commit Throughput

As our processor is designed for a peak throughput of two micro-ops per cycle, the reorder buffer and commit unit are designed for this bandwidth. To verify this choice, we simulate the IPC of the processor while varying the throughput of the commit unit from 1 to 4 micro-ops per cycle, using the same subset of the workloads used in Chapters 6 and 7 (user-mode benchmarks and booting an OS, 30 billion instructions). Figure 8.3 shows that two micro-ops per cycle is a good design point: committing only one micro-op per cycle causes a 18% loss in IPC, while increasing the commit throughput beyond two per cycle (draining the ROB faster after a pipeline flush) only gains up to 1% more IPC. It is interesting to note that lowering the peak throughput of the commit unit to 1 micro-op/cycle has a large performance penalty even though the average IPC (and micro-ops per cycle) is below 1.

8.3.2 Pipeline Flushes

While the throughput of the commit unit affects the processor performance in the common case where instructions are executed and committed without misspeculations, the frequency of pipeline flushes and how quickly pipeline flushes occur also have an impact on performance. We will analyze pipeline flushes here, as the commit unit is responsible for handling them, even though much of the responsibility for causing them are scattered around various other parts of the processor design (such as branch predictors).

Table 8.1 shows the overall impact of speculation and pipeline flushes for our 30-billion instruction workload. Pipeline flushes occur every 133 instructions on average, and cause 12% of all executed micro-ops to be discarded, discarding around 23 micro-ops per pipeline flush. We believe these numbers indicate that the processor has a reasonable design without obvious performance bugs. For comparison, one example system using a simulated 8-wide processor showed squash rates between 3 and 35% on the SPECint2000 benchmark suite [134]. We also compared the squash rate with the Intel Haswell processor using the CoreMark benchmark (See Appendix E), showing that the fraction of wasted micro-ops (17% vs. 3.4% for Haswell) is reasonable after accounting for Haswell's superior branch prediction. While this

[1] The rule we use in this case is to commit the atomic "instruction" anyway if there is exactly one "instruction" in the ROB, all of its micro-ops in the ROB are complete, and there is already an exception pending.

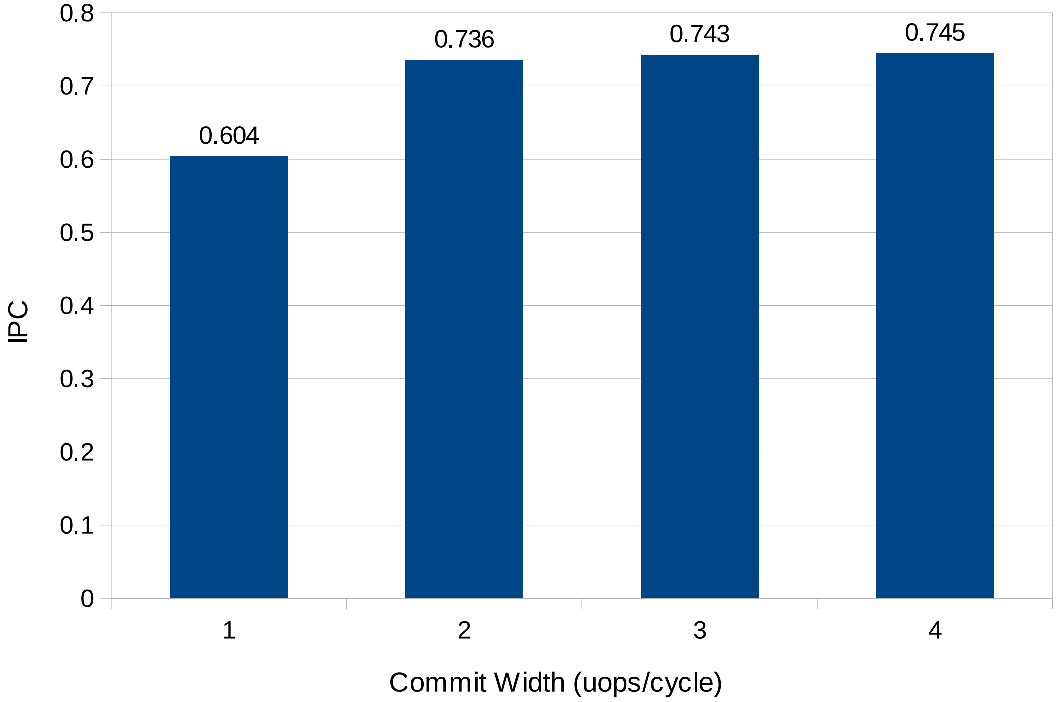


Figure 8.3: Commit width impact on IPC. Committing micro-ops faster than two per cycle improves performance by about 1%.

Metric	Value ($\times 10^6$)	
Instructions committed	30 226	
Pipeline flushes	227	
µops committed	33 845	(88% of µops)
µops discarded	4 610	(12% of µops)
µops total	38 454	

Table 8.1: Pipeline flush impact on instruction and micro-op counts

number shows that the overall pipeline flush impact is reasonable, breaking down the number of pipeline flushes by cause gives further insight into the processor design.

Table 8.2 shows a breakdown of the 227 million pipeline flushes by cause. Unsurprisingly, branch mispredictions cause nearly all (97.7%) of the pipeline flushes, confirming yet again the well-known fact that branch prediction is important for processor performance. The next most frequent cause is string-op aborts. These occur for string compare and scan operations where the string operation stops before reaching the end of the string (e.g., string compare found two strings to be different). These involve a flush of the out-of-order portion of the processor, but not the in-order front-end. Self-modifying code flushes occur if the (conservative) filter mechanism suspects the possibility of self-modifying code. The relative rarity (0.7% of flushes) suggest there is potential to trade an increase in false positive detections for further simplification of the self-modifying code detection logic. Overall, the relative rarity of pipeline flushes caused by reasons other than branch mispredictions indicate that those components of the processor

Metric	Value ($\times 10^3$)	
Pipeline flushes	227 413	(100%)
Branch mispredictions	222 224	(97.7%)
String-op aborts	2 686	(1.2%)
Self-modifying code detected	1 705	(0.7%)
Interrupts and exceptions	440	(0.2%)
Memory ordering violation	184	(0.1%)
Flush after changing global machine state	175	(0.1%)

Table 8.2: Breakdown of pipeline flushes by cause

were of reasonable design.

Now that we know the number of pipeline flushes and the reasons that cause it are consistent with a reasonable processor design, we can then ask whether each pipeline flush itself behaves correctly and has reasonable performance. The next subsection looks at the cycle-by-cycle behaviour of pipeline flushes.

8.3.3 Analysis of a Pipeline Flush

When describing a processor microarchitecture, the cost of a pipeline flush is often summarized as a single number, the branch misprediction latency (also often called the "pipeline length"[2]). While this may be a good approximation metric, the actual behaviour of the processor is rather more complex than just flushing the pipeline and sitting idle for a few cycles, as describing the pipeline flush cost as a single number might suggest.

Figure 8.4 shows a plot of how instruction execution and commit behaves immediately following a pipeline flush caused by a branch misprediction. A pipeline flush (mispredicted branch being committed) occurs in cycle 0. The chart plots how the pipeline recovers until it reaches steady state again. For instruction execution, we plot the occupancy of the instruction scheduler and also the execution throughput in number of micro-ops per cycle. For instruction commit, we plot the occupancy of the reorder buffer and the commit throughput in number of micro-ops (excluding those that are discarded) per cycle. Execution and commit throughputs are 3-point moving averages to produce a smoother plot. In the steady state (far right), the ROB and schedulers hold about 30 and 24 micro-ops, respectively, and $\frac{4}{3}$ micro-ops are executed and committed each cycle (The 4 micro-op loop takes 3 cycles per iteration).

We can roughly divide the process into 5 phases. At the start of phase 1, the schedulers are flash-cleared, leaving the schedulers empty (The two micro-ops in the schedulers at cycle 1 are stray squashed micro-ops inserted after clearing the schedulers, which ideally should be discarded earlier). In this phase, there are no micro-ops to execute, leaving the ALUs idle. While waiting for corrected-path instructions to be fetched and decoded, the commit unit drains the ROB of old wrong-path instructions at a rate of two per cycle. The ROB cannot be flash-cleared because it must free resources such as physical registers. For this workload, phase 2 begins at around cycle 12 when corrected-path micro-ops are inserted into the schedulers and begin executing. At this point, the ROB has not finished draining old micro-ops, so

[2] The true pipeline length of a processor is rarely mentioned, because stages that are outside the branch misprediction loop tend to have minimal impact on performance.

new micro-ops cannot be committed. As the fetch/decode front-end usually runs faster than execution, the scheduler begins filling up. The ROB occupancy is flat because the rate at which old micro-ops are drained matches the rate at which new micro-ops are inserted (2 micro-ops per cycle). In phase 3, the ROB has been drained of old wrong-path micro-ops, and new micro-ops begin committing. In phase 4, instruction execution has reached its steady state of $\frac{4}{3}$ micro-ops per cycle. In phase 5, instruction commit also reaches its steady state behaviour.

It may seem as though the delay in committing new instructions during phase 2 would lower performance. However, delays in committing instructions have minimal impact on performance unless the ROB is full (the ROB is not full because phase 1 drained away many micro-ops). The commit unit can usually catch up on backlogged commits because commit typically has higher throughput than execution. In this chart, we see commit throughput approach 2 micro-ops per cycle as it catches up in phases 3 and 4, committing at a higher rate than execution until the processor reaches steady state.

The workload used for Figure 8.4 is the first component of the bitcount benchmark in the MiBench benchmark suite [118]. The inner loop of this workload counts the number of 1 bits in an integer:

```
while ((x = x&(x-1)) != 0)
    n++;
```

This translates into a loop of 4 instructions (also 4 micro-ops in our implementation) that runs in 3 cycles per iteration, with one unpredictable branch when the while-loop terminates. This workload is short, easy-to-analyze, and has one branch, which allows us to probe the behaviour of one branch with a minimum of other confounding issues. Using a more complex workload results in a qualitatively similar plot, but with the features less clearly visible because some branch mispredictions recover quicker than others. In our simulator, we capture the relevant statistics at every cycle for the first few hundred cycles after every pipeline flush. The bitcount algorithm is run on 75 000 random numbers, which produced 76 000 branch mispredictions (including some code outside of the inner loop). We then average the 76 000 cycle-by-cycle traces over the 76 000 pipeline flushes to produce a trace of the average behaviour, which is then plotted.

The plot in Figure 8.4 has been useful in our design process to debug performance issues involving the pipeline flush mechanism. Unfortunately, we cannot create such a plot for a processor using performance counters, and thus cannot compare against an existing processor to sanity check our design. However, comparing the features in the plot with expected behaviour has been useful. For example, this plot allowed us to debug a lower than expected IPC on this workload (bitcount), where larger instruction schedulers and ROB performed even worse, and discovered the importance of flash-clearing instruction schedulers during pipeline flushes. Not flash-clearing instruction schedulers increased the pipeline flush penalty by spending cycles slowly draining the instruction schedulers and delaying execution of new instructions after they entered the scheduler, leading to long drain/idle period that gets worse with larger schedulers.

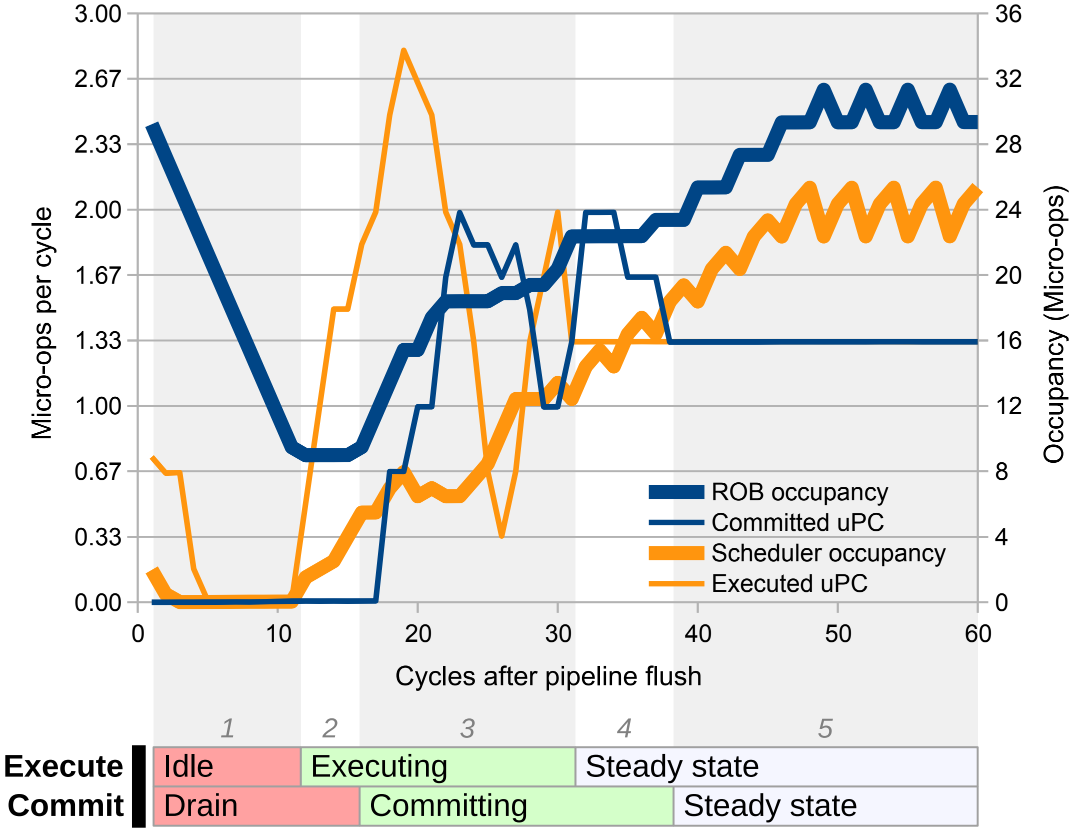


Figure 8.4: Pipeline flush behaviour: Cycle-by-cycle recovery of micro-op execution and commit.

8.4 Hardware

We have not yet designed the hardware for the commit unit. We believe it can be implemented to meet our timing constraints, as the commit unit pipeline length can be increased to satisfy cycle time requirements with only a small IPC performance impact. Due to its location in the pipeline, increasing the instruction commit latency only increases the pipeline flush (including branch misprediction) latency. Interrupt and exception latency has a minimal impact on performance because they are infrequent and already have an unavoidable large cost (for context/privilege changes). Increasing the branch misprediction latency has a larger performance impact, but this latency affects only the delay to recover the register renamer state (not redirecting fetch/decode, which is done during branch execution), and much of that impact is mitigated by branch prediction. The IPC impact of extending the commit pipeline length is a loss of approximately 1% IPC for each extra cycle for the first few cycles of extra latency, growing to 11% at 10 extra cycles.

8.5 Summary

This chapter described the microarchitecture of the reorder buffer and instruction commit hardware. Although the commit unit is outside the most performance-critical execution portions of the processor, pipeline flushes and how they are handled still impact performance. When there are no pipeline flushes, the commit bandwidth needs to be high enough to not become a bottleneck (which is two micro-ops per cycle in our case). Not surprisingly, nearly all pipeline flushes are due to branch mispredictions (98%). When handling pipeline flushes, it is important to be able to instantaneously empty the instruction schedulers of wrong-path instructions, as schedulers filled with wrong-path instructions delay execution of new corrected-path instructions, increasing the effective pipeline flush latency.

Due to the many different causes of pipeline flushes, requiring different handling methods, and various other corner cases, much effort was required to design the commit logic to correctly handle every corner case. Although we have not designed circuits for the commit unit, we believe it is feasible to design correct hardware without too much timing closure difficulty, because our microarchitectural simulation of the commit unit was written at an abstraction level not much higher than logic equations, and the commit unit can be pipelined without a large loss in IPC (1% per cycle),

Chapter 9

Instruction Scheduling

In an out-of-order processor, instruction scheduling is the hardware structure that creates the out-of-order execution behaviour. Instruction schedulers hold a small pool of instructions that are awaiting execution. The instruction scheduler tracks data dependencies and determines which instructions are ready to execute. It then selects some of these ready instructions for execution. While instructions enter into the instruction scheduler in program order, these instructions become ready and are selected for execution as soon as their data dependencies are satisfied, and not necessarily in program order. Being able to examine a pool of instructions allows an out-of-order processor to avoid unnecessary stalls and find instruction-level parallelism to improve processor performance.

This chapter focuses on the design of instruction schedulers for FPGA processors, and explores the microarchitecture and design of scheduler circuits that yield high IPC and large gains in f_{max} operating frequency. Prior work has often failed to achieve reasonable f_{max} [21, 37, 38], although at least one prior work has shown single-issue out-of-order instruction schedulers on an FPGA at reasonable f_{max} [36]. Our new circuit designs improve on these earlier results, achieving 60% greater f_{max} for the same size of scheduler. We then use these scheduler circuit structures to build *multi-issue* distributed schedulers that have slightly higher f_{max} than single-issue scheduler circuits of the same total scheduler capacity.

9.1 Review of Instruction Scheduling in Out-of-Order Processors

The key attribute of out-of-order processors is that they execute instructions in *dataflow* order (based on data dependencies) rather than program order. In typical processor pipelines, this dataflow ordering occurs after the instructions are fetched, decoded, and register renamed in program order. They are then inserted into the instruction scheduler, which executes instructions as they become ready. Instructions leave the scheduler when completed. Finally, completed instructions are committed in program order.

The instruction scheduler is responsible for tracking the readiness of every not-yet-completed instruction and for choosing which ready instruction should be executed each cycle. An instruction is ready to execute when all of its source operands are available, having been computed by previously exe-

Portions of this chapter have appeared at FCCM 2016 [135] and accepted to ACM TRETS cuted instructions.

An instruction scheduler holds a pool of instructions that have not yet executed which are waiting to be executed. The *wakeup* portion of the scheduler is responsible for determining when a waiting instruction is ready for execution. It does this by observing which instructions are completing in each cycle and comparing their outputs with the required inputs for each waiting instruction. The *selection* logic is responsible for selecting one of the ready instructions for execution.

Multi-issue schedulers are built around the same wakeup and select circuits. The wakeup and select logic can be extended to handle multiple operations per cycle, or more commonly, the wakeup and select logic can be replicated to achieve the needed instruction execution throughput. In a superscalar processor, the peak throughput of the instruction schedulers usually exceeds that of the overall processor (fetch, decode, and commit), as instruction execution tends to have lower utilization due to data dependencies and the execution units being specialized for particular types of operations (e.g., branches, memory, or integer).

9.1.1 Scheduler Trade-offs

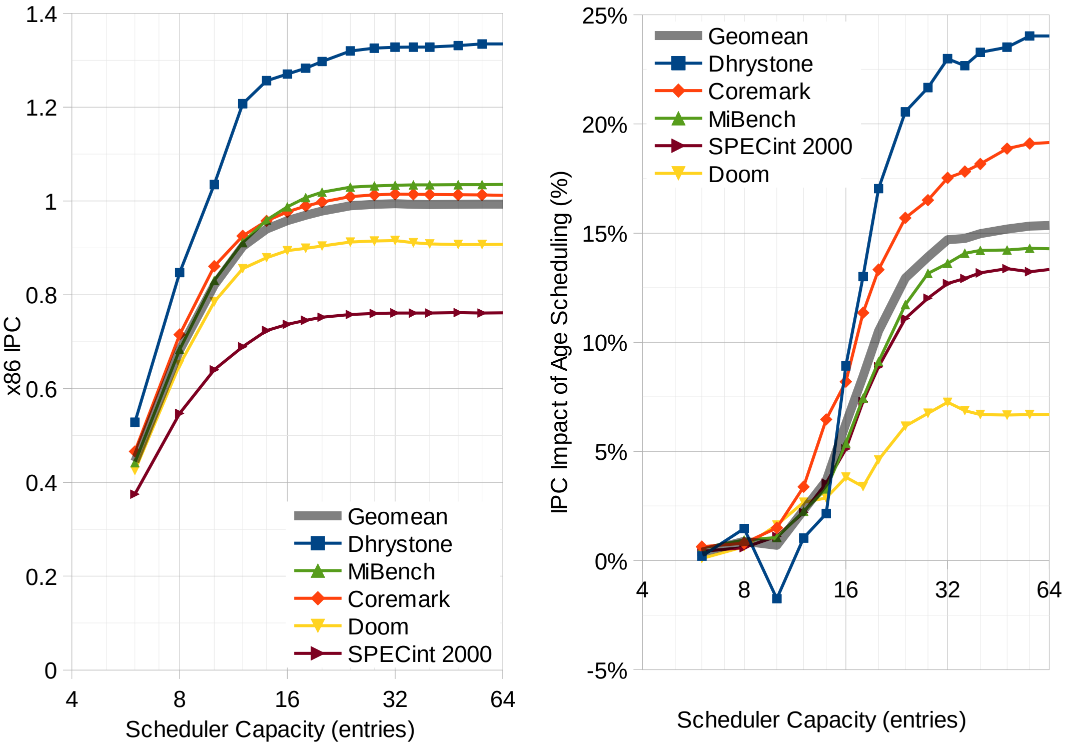
Processor design is about trading off IPC, f_{max}, and design complexity. Here we discuss three major design decisions that affect this trade-off.

First, the number of scheduler entries affects how far ahead in the instruction stream instructions can be considered for execution. A small number of entries limits the ability to extract instruction-level parallelism (ILP) whereas larger schedulers (with more entries) increase ILP and IPC, but require more area and tend to have lower f_{max}. For example, Figure 9.1a shows how IPC improves with scheduler size on our two-issue superscalar system. It shows that more scheduler entries eventually give diminishing returns — this is because other parts of our processor limit the number of in-flight instructions to 64 (reorder buffer size). The figure also shows that there is severe IPC loss with small schedulers of less than 16 entries.

Second, the selection policy — how to decide which of several ready instructions should execute — has an impact on IPC: choosing the oldest instruction first is a known good heuristic as it is more likely that an older instruction blocks execution of later dependent operations. However, an oldest-first heuristic requires tracking the age of entries in the scheduler, which has a hardware cost. Figure 9.1b shows the impact of an oldest-first selection policy compared to random selection. The IPC impact is small for small schedulers because the chance of having more than one ready instruction is lower, but the impact grows to over 15% for large schedulers. Prior out-of-order processors have mostly employed age-based selection [112, 136–138].

The third key decision is whether wakeup and selection operations complete in a single cycle, which allows execution of dependent operations in consecutive cycles. Without back-to-back execution of dependent instructions, a processor will suffer a roughly 10% IPC penalty for adding just one extra cycle [139, 140]. Back-to-back execution of dependent instructions does make circuit timing challenging, however.

The trade-offs in multi-issue schedulers are the same, but have many more degrees of freedom. In



(a) IPC vs. Scheduler Capacity

(b) Benefit of Age Priority vs. Random

Figure 9.1: IPC sensitivity to scheduler capacity and age-based selection policy. The simulated processor has 1 each of branch, ALU, AGU, and store-data execution units, 64 reorder buffer entries, and a peak IPC of 2.

a distributed multi-issue scheduler (where each execution unit has its own scheduler), each cluster can have its size, selection policy and latency chosen independently from the others.

Because we focus on improving processor performance, we make the following two up-front design decisions: First, a requirement of single cycle wakeup and second, an oldest-first selection policy (although we will measure the impact of omitting this for one case). For all scheduler designs we explore, we will measure the impact of a wide range of the number of entries.

9.2 Background on Scheduler Circuits

As described above, schedulers have two key components: wakeup logic to determine which instructions are ready and selection logic to choose among the ones that are ready to execute in the next cycle. In this section we describe how classical hard processor *CAM*-based and *matrix* [141] schedulers perform these two functions.

9.2.1 Wakeup Logic

CAM-based schedulers track operand dependencies using physical register numbers (after register renaming). Each entry in the scheduler's wakeup array holds an instruction's two source operand register numbers and two comparators that compare them to the destination register number of instructions completing each cycle. A source operand is available after its register number has been broadcast on a result bus, and an instruction is ready when all source operands are ready.

Matrix-based schedulers track dependencies by the position of producer instructions in the scheduler. Each entry (row) of the wakeup array contains a bit vector indicating which *instructions* in the scheduler will produce the source operands. The result bus bit vector indicates which *instructions* are granted execution each cycle, and an instruction is ready when all producer *instructions* have completed. This arrangement uses wired-OR gates instead of comparators to compute when each entry is ready. Grant signals broadcasting vertically to the horizontal wired-OR gates with an SRAM cell at each intersection results in a circuit that resembles a matrix.

9.2.2 Selection Logic

The selection logic is responsible for choosing one instruction for execution from a set of ready instructions. The simplest and fastest selection logic uses fixed priority, prioritizing instructions based only on an instruction's position in the scheduler. However, age-based selection heuristics are better than random selection for IPC (Section 9.1.1). Age-based selection can be achieved by maintaining age ordering of scheduler entries and compacting holes so that scheduler position corresponds to age, or allowing random ordering of instructions in the scheduler and augmenting the selection logic with age information.

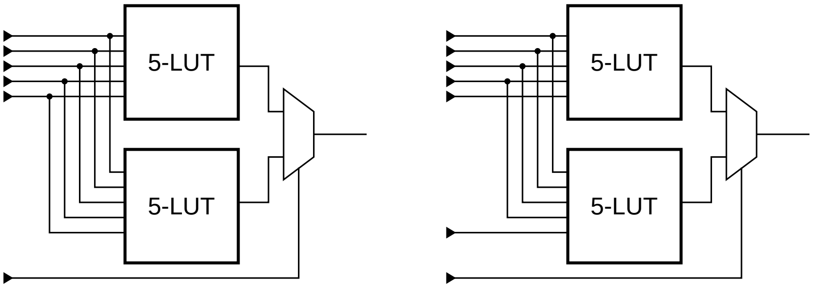
Compacting schedulers insert new instructions at the top, and shift scheduler entries down to fill holes left behind by instructions that have completed execution. Compaction allows a fast fixed-priority selection circuit to be used. The main drawback is in the power consumption of shifting the scheduler entries and the delay of the multiplexer required for shifting.

The alternative of explicitly tracking instruction age makes selection logic more complicated due to dynamic priority. There are many methods to track age, including precise and approximate methods (e.g., [112, 137]). Our matrix scheduler uses age matrices, a precise method that uses a matrix where the bits in each row indicate which instructions are older than the instruction occupying the row [142].

Hybrids approaches have also been used, such as the Alpha 21264 that uses a compacting scheduler that tracks register numbers, but uses wired-OR dynamic logic instead of comparators [136].

9.3 Scheduler Circuits on FPGAs

FPGA logic is composed of LUTs and wires, while custom CMOS has much more flexibility in implementation. Unfortunately, matrix schedulers rely heavily on dynamic logic and wired-OR circuits with dense, regular layouts, so matrix-style schedulers become less appealing on FPGAs. However, LUT-based matrix circuits can still be optimized.



(a) 6-LUT (b) 7-input function

Figure 9.2: Altera ALMs can implement some 7-input functions

Instruction schedulers are usually implemented with separate wakeup and select circuits, performed sequentially. For matrix schedulers on FPGAs, the wakeup logic's wide OR gate reductions and the selection logic's (conceptually) linear pick-first-ready scan logic are both implemented as trees of LUTs. In some circuits, it is possible to reformulate the logic function to combine two reduction or scan operations into one, improving delay. The sum-addressed decoder is one well-known example of this kind of transformation [143].

Inspired by this strategy, we present a new scheduler circuit, the fused-logic matrix scheduler, that combines both the wakeup wide-OR and select linear scan operations into a single tree of LUTs. This circuit is faster than both the CAM and age-based matrix schedulers for most scheduler sizes.

9.4 Detailed Circuit Designs

This section discusses the circuit designs of the three scheduler circuits we implemented on the Stratix IV FPGA: a compacting CAM scheduler, a non-compacting matrix scheduler, and our new fused-logic matrix.

Before we discuss detailed circuit implementations, we first explain the 7-input mode of Altera Adaptive Logic Modules (ALMs), which we use in several of our circuits [144]. The Stratix IV ALM contains an 8-input fracturable 6-LUT. Although it is mainly intended to allow fracturing into two smaller LUTs (e.g., two independent 4-LUTs), the ALM can also implement 7-input functions that can be expressed as a 2-to-1 multiplexer selecting between two 5-input LUTs sharing 4 inputs (Figure 9.2). Having LUTs that implement logic functions with more inputs can reduce logic depth. Priority multiplexers and our new fused select-and-wakeup logic are mapped to 7-input functions that fit into an ALM.

9.4.1 CAM

Our CAM scheduler implementation uses compaction to maintain age ordering and allows back-to-back scheduling of dependent operations. In each cycle, ready bits are used to select an instruction for execution. The selected instruction's destination tag is then broadcast on the result bus, and consumers of the newly-produced register are woken up.

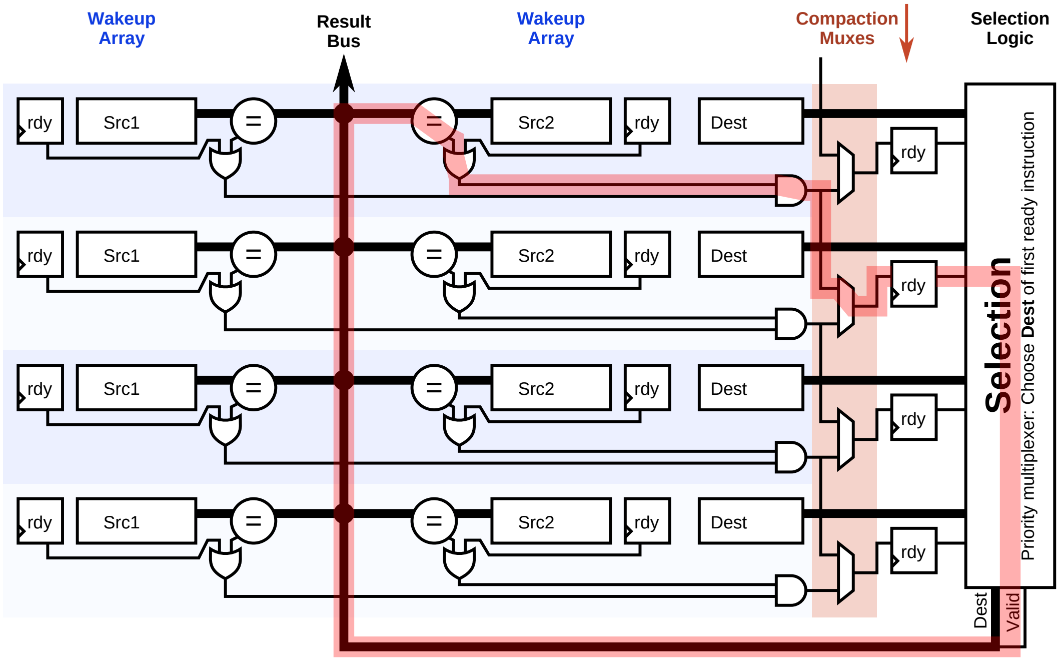


Figure 9.3: CAM-based wakeup circuit. Entries compact downwards. An example critical path is highlighted in red.

Wakeup

Each entry in the CAM wakeup logic has two source operand tags and an associated pair of comparators. The comparators monitor the result bus for a physical register number that indicates when an operand becomes available. An instruction is ready when all operands are available and has not already been selected for execution. The register number is assumed to be large enough to hold at least twice the scheduler capacity, so comparators compare two $log_2 N + 1$ bit numbers. Each 6-LUT can do three bits of a comparison, which is followed by an AND tree, so the total logic depth for two comparators is roughly $log_6(4(log_2 N + 1))$. The ready bit of every entry, forming a *ready vector*, is sent to the selection logic.

Compaction

Our CAM wakeup logic can shift down to eliminate up to one hole per clock cycle. This is enough because only one instruction can be selected for execution each cycle, so new holes are created no faster than one per cycle. Compacting by one position occurs through 2-to-1 multiplexers immediately before the set of pipeline registers.

The control logic to decide whether each entry should shift down is a prefix OR operation, computing for each entry whether there is a vacant entry at or below the current position. This prefix OR function is implemented using a tree of LUTs with logic depth $log_6(N)$ using a radix-6 Han-Carlson prefix tree with sparsity 6 [145]. The radix and sparsity were chosen to suit a 6-LUT FPGA architecture, rather than the more typical radix-2 used in custom CMOS designs. This is much faster than a naïve implementation

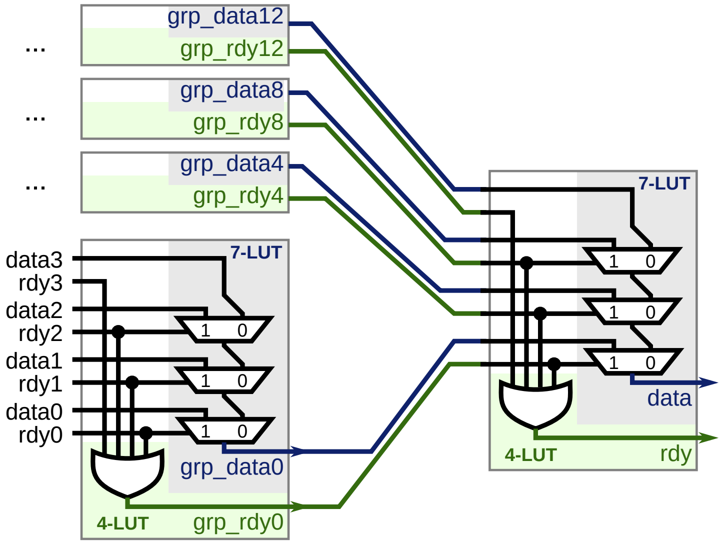


Figure 9.4: Priority multiplexer built from radix-4 blocks, each of which is a size 4 priority multiplexer. This figure shows how a depth-2 tree can implement a 16-entry priority multiplexer. **grp_data** and **grp_rdy** fit in a single 7-input and 4-input LUT, respectively. Used for CAM selection and fused-logic matrix selection.

that uses a linear chain of 6-input OR gates with depth $(N-1)/5$.

Selection

The CAM scheduler's selection logic performs two functions. It must grant execution to the oldest ready instruction, and it must also select that instruction's destination register number and broadcast it on the result bus to wake up dependent operations.

One grant signal per entry indicates whether that entry has been selected for execution. Oldest-ready grant logic is implemented using the same radix-6 Han-Carlson prefix tree used for computing the wakeup compaction multiplexer control signals.

Generating the destination register is done with a priority multiplexer that selects the destination register number field of the oldest ready instruction. The priority multiplexer has a logic depth of $log_4 N$ LUTs, implemented as a radix-4 tree using 7-input ALMs. Figure 9.4 shows this circuit.

9.4.2 Matrix

The matrix scheduler implementation tracks dependencies of instructions using a wakeup matrix of dependency bits. We evaluated the matrix scheduler both with and without age-based selection. Age-based selection tracks the age of each entry using an age matrix, without compaction. In each cycle, the ready bits (and age matrix) are used to select an instruction for execution, and grant signals are broadcast into the wakeup matrix to wake up dependent instructions.

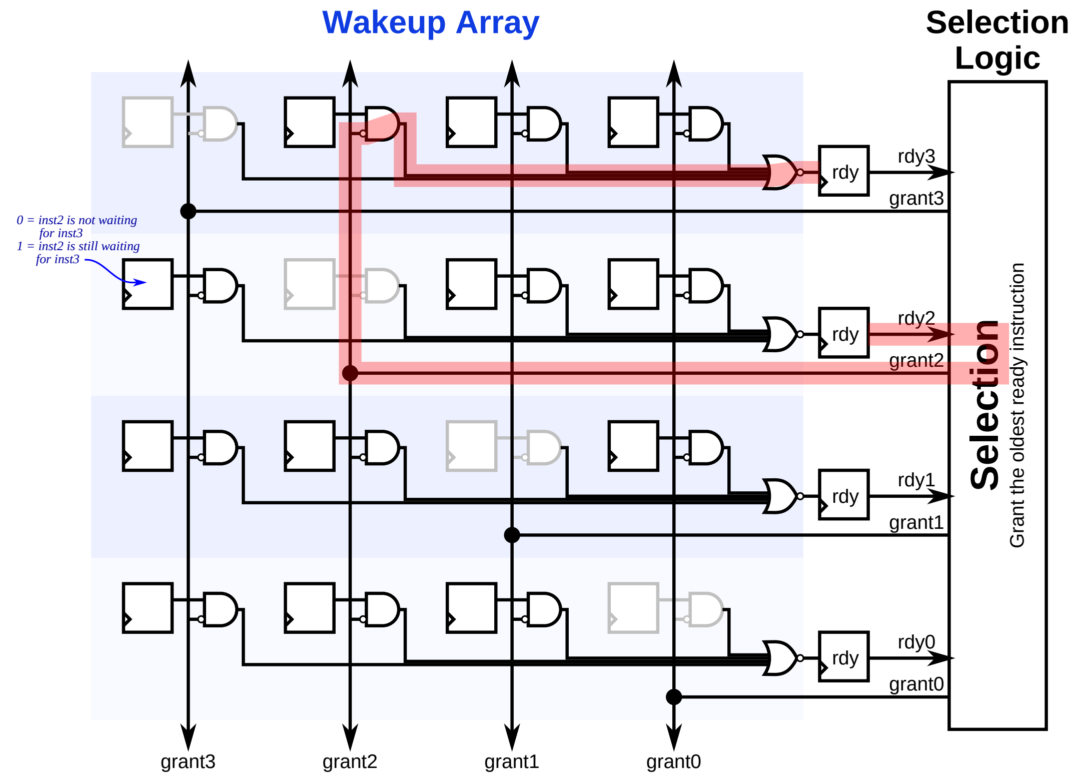


Figure 9.5: Matrix wakeup circuit. Diagonal is omitted as an instruction does not depend on itself. An example critical path is shown.

Wakeup

The wakeup array consists of a matrix of dependency bits. Each row corresponds to an instruction in the scheduler, and the bits in each row indicate which other instructions must execute before this one may do so. These bits are eventually cleared by the grant signals of the parent instructions when they execute. When all of the bit positions were ready or just granted, the ready bit for the row is set, resulting in a N-wide NOR of required-and-not-granted functions. An N-wide NOR of two-input functions ($2N$ inputs) can be computed with a tree of 6-input LUTs with depth $log_6(2N)$.

Position-Based Selection

The position-based select logic grants a ready instruction if there are no other ready instructions before it. As scheduler position does not correlate with instruction age, position priority is essentially random priority. It is implemented as a prefix OR function using the same radix-6 Han-Carlson prefix tree as found in the CAM compaction control (Section 9.4.1) and CAM selection grant logic (Section 9.4.1). It is simpler and has higher frequency than the age-based selection logic, but with lower IPC.

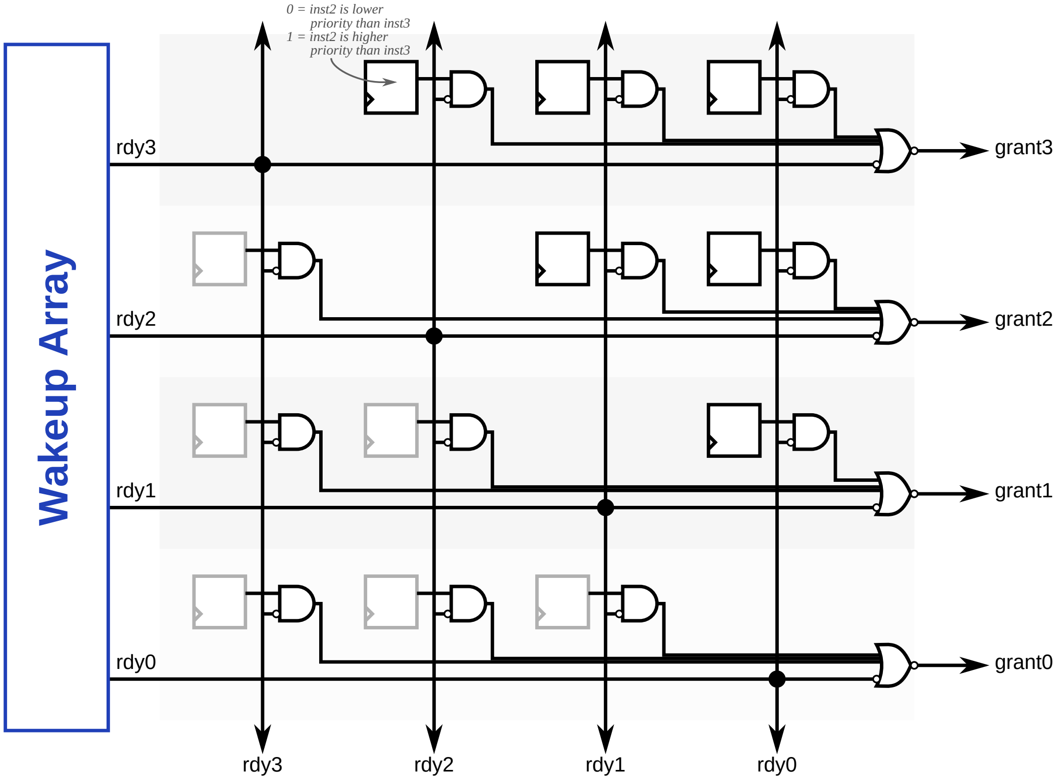

0 = inst2 is lower priority than inst3
1 = inst2 is higher priority than inst3
Wakeup Array
rdy3
rdy2
rdy1
rdy0
grant3
grant2
grant1
grant0
rdy3
rdy2
rdy1
rdy0

Figure 9.6: Age matrix selection circuit. An entry is granted if it is ready and the grant is not "killed" by a higher-priority grant. Lower triangle registers are omitted as it is the complement of the upper triangle.

Age-Based Selection

The age-based selection logic uses an age matrix to dynamically specify age priority, as the scheduler entries are not ordered by age. An age matrix specifies for each row which other instructions are older than itself. When a new (youngest) instruction is inserted into the scheduler, its corresponding row in the age matrix is set to 1 to indicate that every other instruction is older, and its corresponding column is cleared to 0 to indicate to every other instruction that the newly-inserted instruction is younger than it. A ready instruction is granted execution if there are no older ready instructions, which is a N-wide NOR of ready-and-older functions. This is computed with a radix-6 tree with logic depth $log_6 2N$, shown in Figure 9.6. We note that the age matrix has symmetry (if instruction A is older than B, then B must be younger than A), so we omit half of the registers to reduce area.

Compared to the compacting CAM scheduler, the 2-to-1 compaction multiplexer and radix-4 priority multiplexer are removed from the critical loop. Dynamic-priority grant logic is slower than fixed-priority grant logic, with depth $log_6(2N)$ rather than $log_6(N)$. The CAM and matrix wakeup delays scale differently with scheduler size, favouring matrix wakeup for small sizes, but CAM wakeup for large sizes.

9.4.3 Fused-Logic Matrix

As noted in Section 9.3, matrix schedulers were originally formulated for dynamic wired-OR logic, which, in the FPGA context, have to be replaced with trees of LUTs. With separate wakeup and selection circuits, both the CAM and matrix schemes contained two such reduction trees of LUTs in their critical path. In the CAM scheduler, the operand register number comparators reduce the many bits of both operand comparisons down to a single ready bit (wakeup), and the selection logic reduces a vector of ready bits down to a single destination register number for the granted instruction (selection). In the matrix scheduler, a LUT tree reduces one row of the wakeup matrix down to one ready bit (wakeup), and the selection logic reduces a vector of ready bits and one row of age matrix down to a single grant signal (selection). To further improve speed, we endeavoured to create a scheduler with a critical loop containing only *one* reduction tree that would perform *both* wakeup and select functions.

The resulting design is a compacting matrix scheduler with fused wakeup and select logic. Dependency information is expressed as a matrix of dependency bits like the matrix scheduler, but select *and* wakeup are computed using a single radix-4 tree of LUTs. Conceptually, instead of having one instance of selection logic broadcasting its result to per-entry wakeup logic, the selection logic is also replicated per entry and merged with the wakeup logic. Figure 9.7 shows this arrangement.

Wakeup and Select

Scheduler entries are ordered by age using compaction, so the selection uses fast fixed-priority selection. Each row has a combined select-and-wakeup circuit. The two inputs to each instance of the select-wakeup logic are a ready vector indicating which instructions are ready for execution, and a dependence vector indicating whether the instruction in the current row is dependent each instruction. The select-wakeup logic computes whether the current instruction depends on the oldest ready (i.e., selected) instruction. If so, this means one dependency has been satisfied, and a two-bit counter storing the number of outstanding dependencies is decremented. An instruction is ready when the counter reaches zero. Grant logic is still used to generate grant signals to clear dependency bits in the matrix, but is now moved off the critical path.

The select-wakeup logic is equivalent to a priority multiplexer, implemented using the circuit in Figure 9.4, which is a radix-4 tree of 7-input ALMs with a logic depth of $log_4 N$. The priority multiplexer finds the first ready instruction and selects the one bit of data indicating whether the instruction depends on the selected (oldest ready) instruction.

There is more preprocessing that needs to be done than for the matrix scheduler. In addition to encoding dependencies as positions in the scheduler, we also need to count *how many* dependencies are outstanding, which is a population count of the dependence vector. In this implementation, the single-cycle preprocessing is a critical timing path. However, because it is outside the wakeup-select loop, it should be possible for future implementations to further pipeline preprocessing without giving up the ability to schedule dependent instructions in back-to-back cycles.

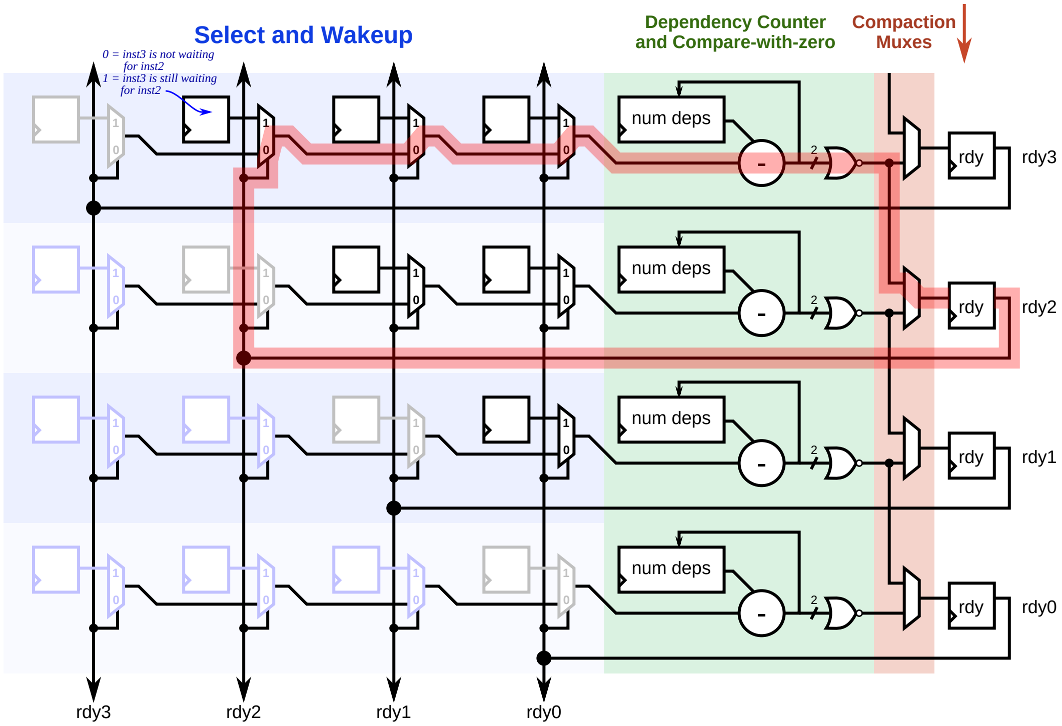

Select and Wakeup
Dependency Counter and Compare-with-zero
Compaction Muxes
0 = inst3 is not waiting for inst2
1 = inst3 is still waiting for inst2
num deps
rdy
rdy3
rdy2
rdy1
rdy0
2
1
0
-

Figure 9.7: Fused-logic matrix circuit. Selection and wakeup are merged and implemented as a one-bit wide priority multiplexer and a two-bit counter. Lower triangle is omitted as instructions do not depend on itself or future instructions.

Compaction

Compaction of a matrix is more complex than for a CAM. In a matrix scheduler, dependencies are represented as a bit vector indexed by the scheduler position of the parent instruction, whose position can change due to compaction. As scheduler entries are compacted downwards in a matrix scheduler, the dependency bit vectors are also compacted horizontally to track the changing instruction positions as they shift down the scheduler. Fortunately, the extra compaction logic is off the critical wakeup and select loop.

9.5 Evaluation Methodology

The main objective of this chapter is to evaluate area and f_{max} of different circuit-level implementations of broadcast-based instruction schedulers. We build optimized circuits for the circuits described in the previous section (CAM, non-compacting matrix, and fused-logic matrix) targeting a Stratix IV FPGA (smallest, fastest speed grade, EP4SGX70-C2) using Quartus 15.0.

We sweep scheduler capacity (entries) and observe area and f_{max} scaling as the scheduler size varies. These results are the mean of 100 random seeds. We focus on area and delay here because all of the scheduler circuits have nearly the same cycle-by-cycle behaviour: they wake up all ready instructions every cycle and select either a random instruction or the oldest ready instruction for execution.

9.6 Circuit Design Results

This section presents area and f_{max} results of implementing CAM, matrix, and fused-logic matrix scheduler circuits on a Stratix IV FPGA.

9.6.1 Area

Figure 9.8 compares the area of the three scheduler circuit types as scheduler size changes. The two matrix schedulers scale similarly, as the size of the matrix grows quadratically with the number of scheduler entries, but the matrix with position-based selection is smaller as it does not have an age matrix. CAM schedulers have better area at large sizes, as the size of comparators increases logarithmically (register number width) but the size of each matrix row's OR gate increases linearly. This can be seen more clearly when plotting area per entry, in Figure 9.8b. Because we scale register number width with the scheduler size (in-flight instructions consume destination physical registers), there are small discontinuities at powers-of-two sizes when the register number width is incremented.

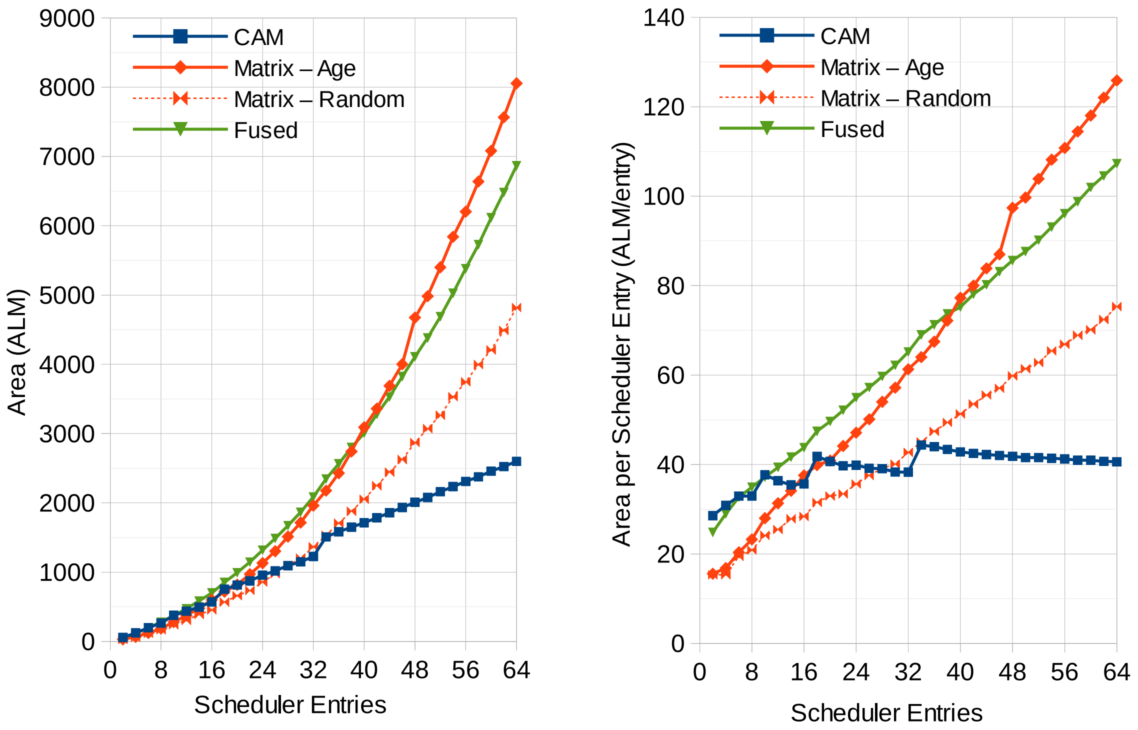
For out-of-order FPGA soft processors, we are mostly interested in small schedulers, generally below 20 entries. All circuit types have similar area below 20 entries, so delay targets will usually determine which scheduler circuit style to choose. The poor area scaling of matrix wakeup logic was also true in custom CMOS, where the original matrix scheduler proposal saw more than four times greater wakeup array area when replacing the CAM wakeup logic with matrices for a 48-entry scheduler, in exchange for halving the wakeup delay [141].

9.6.2 Delay

Figure 9.9 shows the achieved f_{max} for the three scheduler circuit types as scheduler capacity is varied. The general trend, unsurprisingly, is that larger schedulers are slower. The delay for the matrix schedulers increase faster than CAM schedulers at large sizes. On an FPGA where there are no fast wired-OR circuits, we see smaller improvements than those reported for custom CMOS implementations [141].

Among the three age-based schedulers, our new fused-logic matrix scheduler is the fastest option beyond 6–10 entries, though at very large sizes, excessive area causes poor routing delays. CAM schedulers are slow at small sizes, only being faster than the age-based matrix scheduler beyond 24 entries. At small scheduler sizes where giving up age-based selection is acceptable, the position-based matrix scheduler ("Matrix — Random") is the fastest.

For out-of-order FPGA soft processors, we are interested in small schedulers. Below 20 entries, both types of matrix scheduler are faster than CAM schedulers, with little difference in area. To match the clock speed of a Nios II/f on the same FPGA (240 MHz or 4.2 ns), the largest age-based scheduler that will fit a 4.2 ns cycle time is around 20 entries for CAM, 22 entries for age-based matrix, and 42 entries for the compacting fused-logic matrix. A 44-entry position-based matrix scheduler also fits in a 4.2 ns period, but has limited usefulness at this size given the large IPC degradation of the selection policy. For comparison, current high-end x86 processors have 40–60 scheduler entries [112], while earlier out-of-order processors have far less (20 for Alpha 21264 [136], 16 for Pentium 4 [137]). This suggests



(a) Area vs. Capacity

(b) Area per Scheduler Entry

Figure 9.8: Area of four scheduler types. Area per entry gives insight into scaling trends with scheduler size.

that moderately aggressive out-of-order designs are feasible on FPGAs even when targeting the same frequency as simple single-issue in-order soft processors.

9.7 Multiple-Issue Schedulers

In the previous sections, we presented circuit designs for single-issue schedulers. Using one of these circuits alone could form the scheduler of a single-issue out-of-order processor. Superscalar processors that execute more than one instruction per clock cycle can provide large improvements in IPC, but require larger schedulers to find instruction-level parallelism to keep execution units utilized. For example, in our simulated processor design, a dual-issue pipeline has about 70% higher IPC than a pipeline with a single-issue scheduler, but needs about 4 times as many scheduler entries to reach that performance (Figure 9.12b).

To build schedulers for multi-issue soft processors, the circuits presented earlier serve as fundamental building blocks. Building multi-issue schedulers using combinations of the single-issue circuits is reasonably straightforward, but the design space is much larger.

First, one must decide whether to build a unified scheduler where all execution units share the same pool of waiting instructions, or a distributed scheduler where each execution unit has its own private

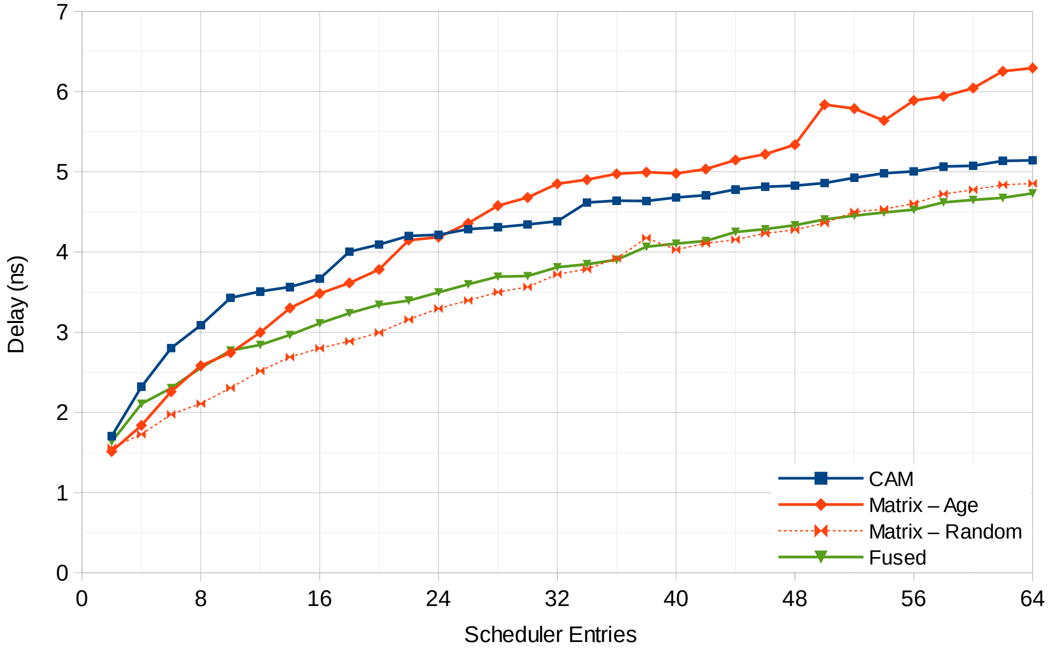


Figure 9.9: Delay of four scheduler types at varying capacities on a Stratix IV FPGA

scheduler. A distributed scheduler then requires choosing a scheduler size for the scheduler attached to each execution unit. In some microarchitectures, there can even be schedulers where certain execution units (e.g., two integer ALUs) are unified while the rest are distributed.

The rest of this section describes our design of a distributed scheduler for use in a two-issue superscalar x86 soft processor that has four different execution units. We compare the IPC between unified and distributed designs, choose capacities for each scheduler in a distributed design, then present area and cycle time results of the four-way distributed scheduler circuit.

9.7.1 Unified vs. Distributed Scheduler

A unified scheduler has a single pool of instructions from which ready operations are chosen, while distributed schedulers use one private scheduler per execution unit. A distributed scheduling scheme stalls the pipeline whenever *any* of its schedulers are full. Thus, unified schedulers make more efficient use of scheduler entries. Distributed schedulers require more entries to achieve the same IPC because the relative demand for each execution unit varies dynamically, and one full scheduler will stall the processor front-end even when other schedulers have free space.

However, circuits for unified schedulers are slower than distributed, as each execution unit's selection logic must search the entire pool of waiting instructions rather than just the scheduler attached to its own execution unit. Some microarchitectures with unified schedulers may even choose to use pick-N selection logic instead of multiple instances of pick-1 selection, which allows even more flexibility in matching instructions to execution units, at the cost of more complex circuits. The wakeup logic is similar for both designs: every executed instruction must broadcast to every scheduler entry.

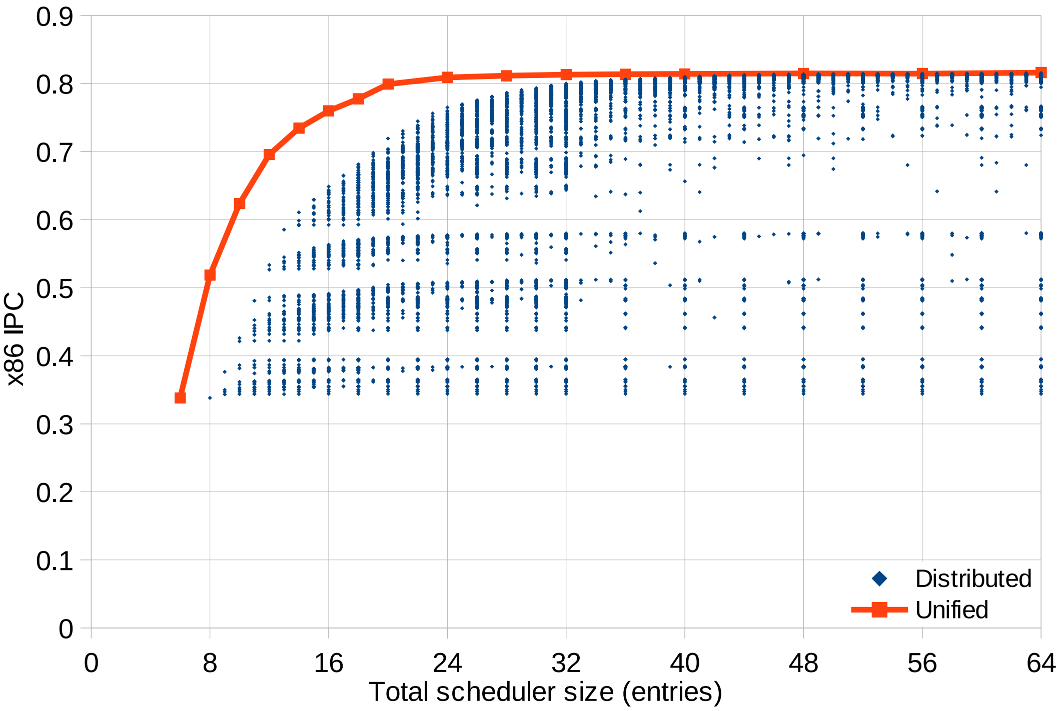


Figure 9.10: Searching the four-component distributed scheduler design space

For our x86 design using complex micro-ops, another complexity of using unified schedulers is the required peak *enqueue* throughput. Renaming and issuing two micro-ops per cycle into the schedulers can produce up to 6 independently scheduled operations (2 micro-ops, each with a load, arithmetic, and store). With a distributed scheduler, these 6 operations are guaranteed to be distributed among at least three different schedulers, so each scheduler only needs to enqueue up to 2 operations per cycle. Using a unified scheme would require the unified scheduler to enqueue up to 6 operations per cycle.

For circuit complexity reasons, we prefer a distributed scheduler design. However, we still wish to know the IPC cost of choosing a distributed design. A proper comparison requires exploring the distributed scheduler design space, which will be discussed in the next section. Figures 9.10 and 9.12b show the final results on two different benchmark sets. The charts show that distributed schedulers need more total entries to achieve the same IPC (e.g., 20 unified and 32 distributed entries have similar IPC).

9.7.2 Tuning the distributed scheduler

Our processor has four execution units (branch and complex operations, integer arithmetic, address generation and loads, and store-data), so a distributed scheduler has four independently-sized components, one for each execution unit. Because each execution unit is specialized for a particular operation type, some of which occur more frequently than others, it is not immediately obvious how to determine the size of each component that maximizes performance for a given total size. Thus, we need to search the four-dimensional design space, then use the results to guide how each individual scheduler should be sized.

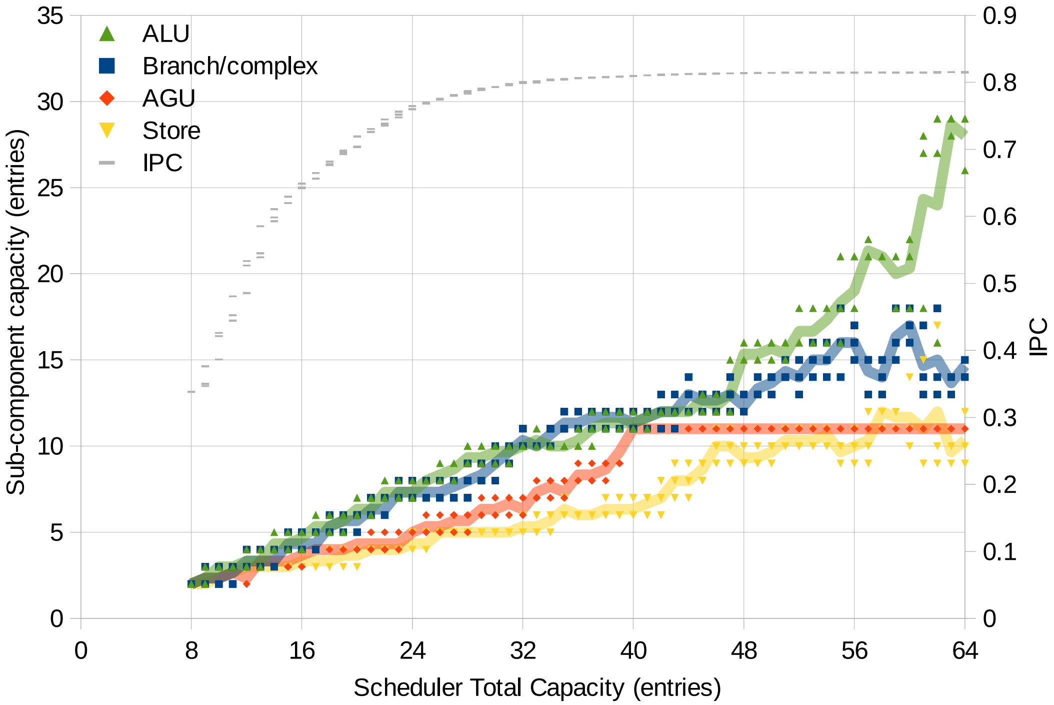


Figure 9.11: Best four-component distributed scheduler parameters vs. total scheduler capacity

It is easy to measure the IPC for unified schedulers as there is only a single parameter (total capacity), but a distributed scheduler has, for us, four independent size parameters. If we limit the maximum total scheduler size to 64 entries, the design space contains just under half a million design points, each requiring simulation. To make this search tractable, we explored the design space using a randomized search, optimizing for maximum IPC at each total scheduler capacity. This is of course a simplified cost function, as it omits details such as cycle time (which is also influenced by the largest scheduler component and not just the total capacity) and that some operation types cost more than others (e.g., stores do not need to wake up dependents). For each design point, we obtained IPC using a cycle-level simulation with a reduced benchmark set, which consists of 279 million instructions taken from five benchmarks: Dhrystone (70M instructions), Mibench (dijkstra 41M and mad 27M), and SPECint2000 (gzip 70M, mcf 70M). We chose these workloads to attempt to keep diversity (ranging from the tiny Dhrystone to the very memory-demanding mcf) despite the small number of workloads. This is a reduction from the workload of 25 billion instructions used in the rest of the chapter.

We first coarsely explored the design space (points where component sizes differ by a multiple of 4, 3060 points). Then, for each scheduler size, we pick several design points at random, strongly skewed toward the best points for that size, and create a new design point for each by perturbing its four parameters randomly. We repeat this process until we were satisfied that the search space was adequately explored. In total, we explored 10 635 points using 150 CPU-days, or about 2% of the total design space of 487 635 points. Figure 9.10 shows the IPC of every design point we simulated. Points closer to the pareto-optimal curve are more densely explored. Also plotted on the chart for comparison is the performance of unified schedulers on the same reduced benchmark set. As expected, unified schedulers achieve the same IPC with fewer total entries.

To use this data to guide how to size each scheduler component, we take only the top-performing three points at each size from Figure 9.10 and examine its four parameters, which are plotted in Figure 9.11. The four lines show the average size of each scheduler for each size of each particular execution unit type. The IPC is also plotted again for reference. Not surprisingly, the best schedulers do not have equally-sized components. However, the relative sizes of the four schedulers do not match their relative utilization (17%, 35%, 37%, 11% for Branch/complex, integer ALU, AGU, and stores, respectively). Perhaps most surprising is that branch/complex operations "prefer" scheduler sizes similar to the integer ALU, despite being utilized only half as often. We speculate this is a side effect of our processor design that executes branches and some complex operations in-order, which causes operations to occupy the scheduler longer than a fully out-of-order ALU would.

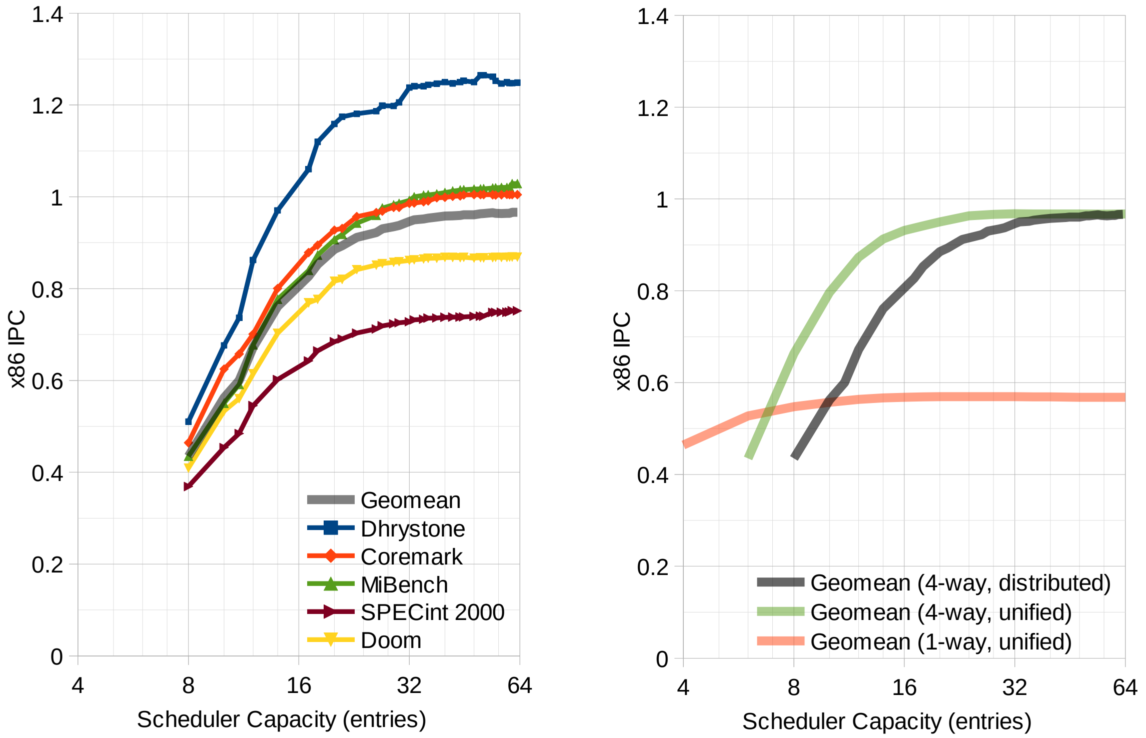
For the remainder of this section, we choose the sizes of each scheduler component based on a linear fit to the portion of the plot with less than 40 total entries. The portion greater than 40 entries behaves somewhat erratically, likely because random noise dominates when the schedulers have more entries than required to achieve maximum IPC.

To verify that our reduced benchmark set was reasonably representative of the full 25-billion instruction workload, we simulated scheduler sizes from 8 through 64 on our full benchmark set, using our linear-fit rule for sizing each execution unit's scheduler. Figure 9.12a shows these results. Figure 9.12b compares this 4-way distributed scheduler with the IPC of a 4-way unified scheduler (from Figure 9.1) and also to a single-issue out-of-order processor with a 1-way unified scheduler. Two-issue (with 4-way scheduling) superscalar execution has a large 70% IPC increase over single-issue, but requires larger schedulers to achieve that performance, while a single-issue out-of-order processor is largely insensitive to scheduler capacity. Single-issue performs better for very small schedulers due to a side effect of our design: The scheduler is considered "full" if there are not enough free entries to accommodate the worst-case output from the renamer, which is 6 operations for dual-issue but 3 operations for single-issue (thus, a dual-issue processor effectively loses 2–3 scheduler entries compared to a single-issue processor).

9.7.3 Proposed Scheduler Microarchitecture

In addition to supporting multiple issue with a distributed scheduler as discussed in the preceding section, we also extended our scheduler to implement details of our target instruction set (x86), such as supporting different instruction types (e.g., arithmetic vs. load), more source operands per operation (up to 3), and different register types (e.g., general-purpose vs. condition codes).

Our processor can decode, rename, and commit two complex micro-ops per cycle. Each complex micro-op is then broken up into up to three operations (load, arithmetic, store) that are executed by four different types of execution unit, with a peak execution throughput of four operations per cycle. (Deciding on the overall processor issue width and arrangement of execution units is outside the scope of instruction scheduler design.) Each scheduler (one per execution unit) can enqueue at most two operations per cycle, and select and dispatch one operation per cycle. In aggregate, the four schedulers



(a) Distributed scheduler IPC

(b) Distributed, unified, and 1-way scheduler comparison

Figure 9.12: IPC comparison between 4-way unified, 4-way distributed, and 1-way unified schedulers.

can enqueue 6 operations (two complex micro-ops with three operations each) and select and dispatch 4 operations per cycle (one of each type). Figure 9.13 shows a high-level schematic of how our four-component distributed scheduler is built out of smaller matrix scheduler circuits.

Unlike for a single-issue processor, the wakeup matrix for each scheduler is much wider than it is tall, as each entry in the scheduler can wait for operands arriving from any scheduler, but the aggregate dimensions of the wakeup matrix is roughly square. It is not precisely square because stores do not wake up any dependent instructions, as they only write data into the store queue.

To help with cycle time, we relaxed the wakeup latency from our earlier assumption of single-cycle latency for some of the less critical paths. Complex ALU operations and memory loads never complete in a single cycle, so the branch/complex and AGU schedulers do not need single-cycle wakeup of its dependents. We also added an extra cycle for operands going to the branch/complex ALU.

According to Figure 9.12a, schedulers with greater than 32–40 entries are wasteful, while those much smaller than 16 have large IPC losses. The final choice of scheduler capacity in the range of 16–32 entries depends on other factors such as the frequency target of the processor and the area budget.

9.7.4 Circuit Design

At this point, we have decided that we should build a distributed scheduler with four sub-schedulers of different sizes (Figure 9.13). We chose to build the scheduler from four *compacting matrix* scheduler cir-

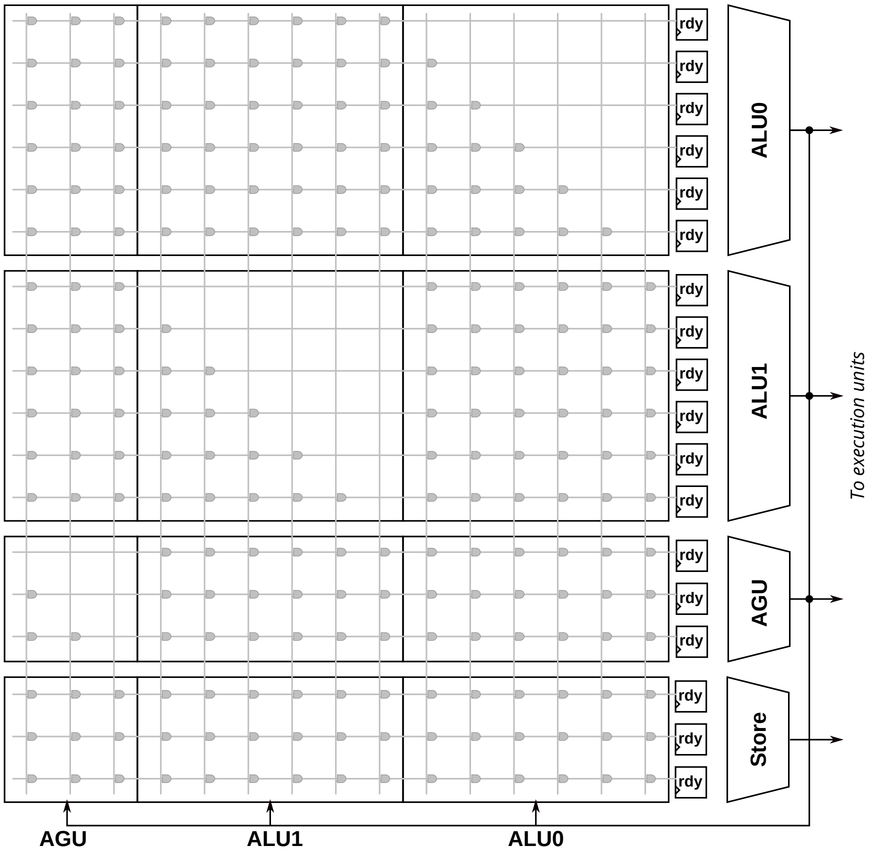


Figure 9.13: Block diagram of a 4-way compacting matrix distributed scheduler. Each cluster can be a different size. Stores do not need to wake up dependent operations.

cuits. This design draws from lessons learned from both the fused-logic and matrix schedulers discussed earlier in Section 9.4.

A compacting matrix uses wakeup and selection logic found in a matrix scheduler, but tracks instruction age by compaction as used by fused-logic scheduler instead of an age matrix. This design attempts to work around shortcomings in each design.

We saw in Figure 9.9 (Section 9.6.2) that the matrix scheduler using an age matrix had higher delay than the fused-logic scheduler. The same figure also suggested that much of the disadvantage for the age-based matrix scheduler may have been the use of an age matrix, as a matrix scheduler without the age matrix (random priority) had lower delay. The choice of using matrix wakeup logic is due to its more straightforward design making it easier to extend to allow more source operands and multiple wakeups per cycle. The fused-logic wakeup logic's use of a counter to count the number of remaining operands becomes problematic because the number of dependencies in our x86-based micro-ops can be up to 3 (instead of two assumed earlier), and up to three dependencies can be satisfied each cycle (instead of one). Extending the counter design will almost certainly add one LUT logic level to the critical path. In contrast, no changes need to be made to the matrix scheduler's wakeup logic to support more source operands or multi-issue. A compacting matrix circuit design attempts to gain some of the fused-logic design's speed without inheriting its extra complexity.

Like all matrix schedulers, we need preprocessing logic to map source register numbers to its producer operation's location in the matrix scheduler. We use an array of one-hot bits to indicate which scheduler operation produces a given *logical* (not physical) register. Because querying this mapping happens in-program-order partially in parallel with register renaming, we can map logical registers directly to scheduler location. This produces a smaller mapping table because there are fewer logical registers than physical registers. This mapping table compacts along with the compaction of scheduler entries. The preprocessing logic is a substantial portion of the scheduler (about 60% of the scheduler's area), and is a significant disadvantage of matrix-based designs. All of our f_{max} and area numbers include this overhead.

9.7.5 Circuit Results

We synthesized distributed schedulers from size 8 through 64 for a Stratix IV FPGA, where the component sizes were determined following the guideline from Section 9.7.1. Figure 9.14 plots the area and delay of these designs. Area and f_{max} results are the median of 50 random seeds.

In Figure 9.14a, the cycle time of the CAM and matrix (random priority) are plotted again (from Figure 9.9) for comparison. One interesting observation is that the distributed multi-issue scheduler has a slightly *faster* cycle time than a single-issue scheduler of the same total capacity, because a distributed scheduler's individual components are smaller. For example, the 32-entry distributed scheduler has components of at most 10 entries each (10, 10, 7, 5) that operate in parallel.

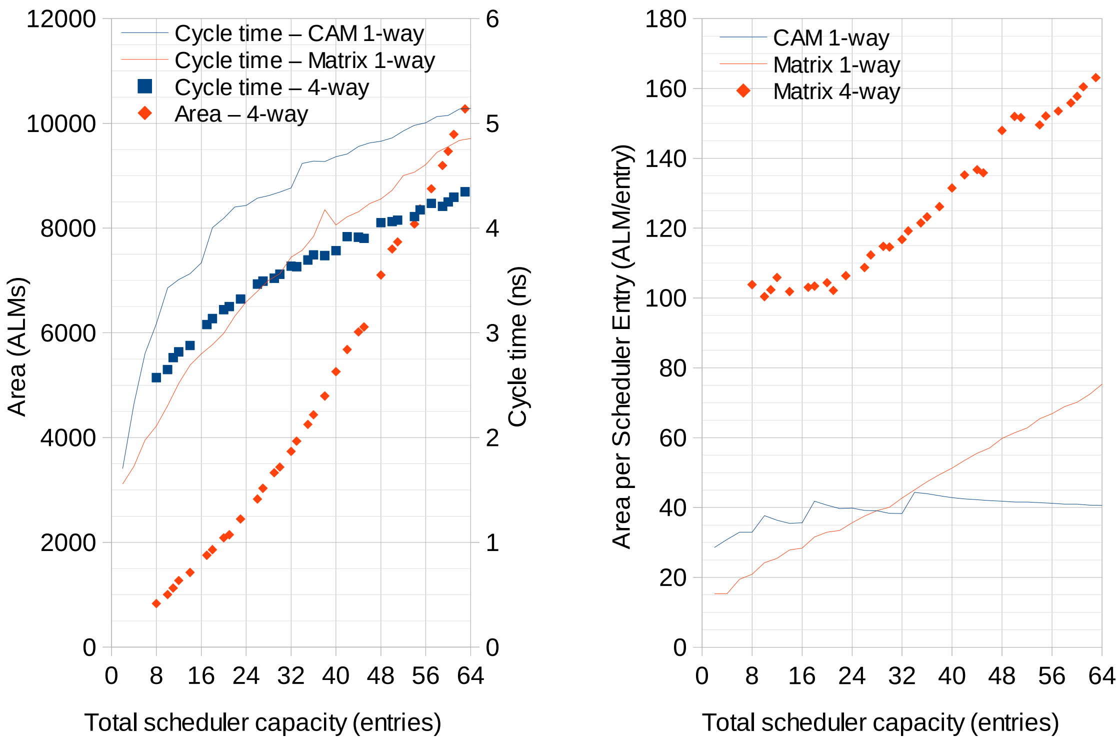
The area of the multi-issue scheduler is substantially greater, as seen in the plot of area per scheduler entry in Figure 9.14b. For comparison, the area per entry of single-issue CAM and matrix (random) circuits (from Figure 9.8b) are also plotted. A large contributor to this increase is the need to map up to 16 (instead of 2) source operands per cycle to the scheduler entry of the instruction that produced the register's value. The wakeup and select logic area alone is similar to the single-issue fused-logic scheduler and slightly higher than the matrix (random priority) due to the addition of compaction.

As mentioned in the previous section, 16–32 scheduler entries is a reasonable range to build from an IPC perspective. Using our compacting matrix circuit results in frequencies of 325–280 MHz (faster than single-issue schedulers of the same size) and area of 1750–3750 ALM (substantially worse than a single-issue scheduler).

9.7.6 Overall Performance

Figure 9.15 combines the IPC (Figure 9.12a) and cycle time (Figure 9.14a) results into a single plot, showing the trade-off between IPC and frequency.

The curved grid lines mark instruction throughput in MIPS, which is the product of IPC and frequency in MHz. Each point on the plot shows the IPC and frequency for the scheduler of a particular capacity. For example, the 17-entry scheduler has 0.83 IPC, 324 MHz, and 268 MIPS, while a 32-entry scheduler achieves almost the same throughput (264 MIPS) but does so with a higher IPC (0.95) and lower frequency (279 MHz). While these two design points have similar overall performance, the latter is easier to build as it imposes a less stringent timing constraint on the rest of the processor.



(a) Area and Delay vs. Capacity

(b) Area per Scheduler Entry

Figure 9.14: Area and delay of distributed 4-way compacting matrix scheduler. One-way CAM and matrix (random selection) circuits (from Figures 9.8 and 9.9) are also plotted in solid lines for comparison.

There is a fairly large region with similar MIPS between 14 and 33 entries (275–347 MHz, 0.95–0.76 IPC, 260–275 MIPS). This may allow a fairly large range of designs to suit the frequency target of the rest of the processor without major performance compromises. Of course, larger schedulers do incur a higher area cost.

We expect soft processor designs would have frequencies toward the lower end on this chart, as the Nios II/f only runs at 240 MHz [146] on a Stratix IV FPGA, and we expect a more complex out-of-order processor would not be able to exceed the Nios II/f's frequency by much. A lower processor frequency target suggests the use of larger, higher IPC schedulers, and motivates more area-efficient designs even if it means some frequency loss for the instruction scheduler.

9.8 Comparisons to Previous Work on FPGAs

Direct comparisons with prior work are difficult to make due to differences in scheduler microarchitecture, but the approximate comparisons can still demonstrate our improvements. In most cases, to match the chip used in prior work, we re-synthesized our scheduler circuits on a different Altera FPGA than the one our circuits were designed for. Our instruction scheduler circuits achieve faster cycle times than schedulers in the literature, in some cases by substantial amounts.

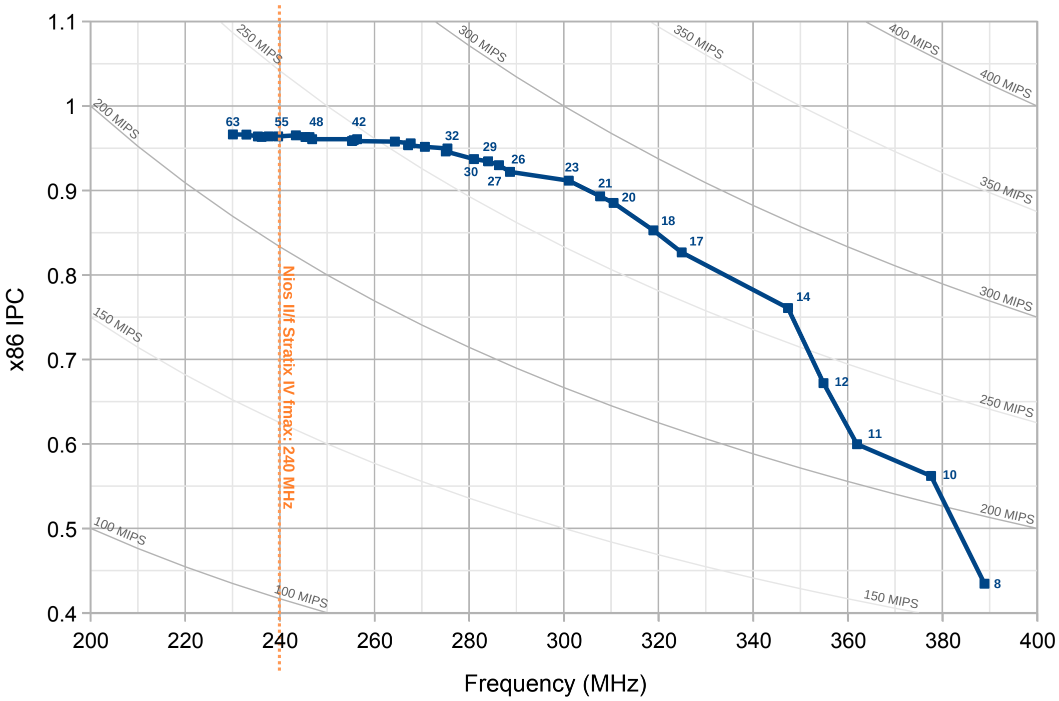


Figure 9.15: IPC vs. frequency of 4-way distributed schedulers from 8 to 64 entries. Peak of 275 MIPS occurs at size 20, but MIPS varies little between 14 and 33 entries.

Single-issue CAM

Aasaraai and Moshovos [36] presented a design space exploration of traditional single-issue CAM schedulers on Stratix III FPGAs. The microarchitecture of their scheduler circuits match well with our CAM (single issue, compacting age-priority, two operands) allowing for a reasonably fair comparison. On the same Stratix III FPGA, we achieve higher frequencies with our CAM scheduler circuit (+40% at 16 entries). Matrix and fused-logic matrix schedulers get additional gains (+47% and +60% at 16 entries, respectively).

It is interesting that Aasaraai and Moshovos recommend using a small 4-entry scheduler because they observe less than 10% change in IPC on their single-issue out-of-order processor between 2 and 32 entries. We also observed insensitivity of IPC to scheduler size for *single-issue* designs, but *dual-issue* demands a much larger scheduler (Figure 9.12b), which makes the ability to build high-capacity schedulers important.

Dual-issue CAM and Matrix

Johri compared two-issue CAM and matrix schedulers on FPGAs [39]. Our single-issue schedulers achieved twice the frequency at 16 entries for both CAM and matrix schedulers, but they use 3 source operands per instruction on a Virtex-6, while we use 2 source operands per instruction on a Stratix IV.

OpenRISC OPA

The OpenRISC OPA out-of-order processor merges the reorder buffer (ROB) and scheduler into a single unit, allowing some circuit simplifications and good f_{max} [34]. On the same Arria V FPGA, our fused-logic matrix scheduler in isolation achieves about 30% higher frequency than their complete processor at both 18 and 27 entries. Its main drawback is that a merged ROB and scheduler is wasteful of scheduler capacity. Schedulers only need to be 30–50% of the ROB size with almost no loss in IPC, which is also seen in our processor design with 64 ROB entries (Figure 9.1a).

Combined ROB and Scheduler

Rosière et al. presented a combined ROB and scheduler [38]. The microarchitecture appears highly unbalanced, with a large (128–512 entry) ROB, but only the oldest few instructions (4–16) are considered for scheduling. On a Virtex-5, they reported slow f_{max} (4.7× slower than our fused-logic matrix at 16 entries), and did not report absolute IPC numbers.

Non-Broadcast Scheduler

SEED is a scheduler designed to avoid broadcast behaviour [37]. On the same Stratix II FPGA, our fused-logic matrix scheduler achieves 1.9–2.4× higher f_{max} over their broadcast-free scheduler, and 6.5–5.5× higher f_{max} over their baseline, an Alpha 21264-like compacting CAM scheduler, for 16 to 64 entries.

9.9 Future Work

There has been much processor microarchitecture research that improves on the fundamental scheduler circuits. Most of these proposals still use the same circuit structures at their core, but trade some amount of IPC to improve area, speed, or power [54, 139, 140, 142, 147–151]. The majority of these techniques can still be used on FPGA designs.

While we aimed for, and achieved, a high speed multi-issue scheduler design, this came with a fairly high area cost. For processors where clock frequency is less critical, slower CAM-based schedulers deserve further exploration for use in multi-issue distributed schedulers. CAM-based schedulers may still be fast enough for a processor with a modest frequency target, potentially at a significant area savings.

9.10 Conclusions

We compared optimized circuit structures used in broadcast-based instruction schedulers. We also presented an improved age-based fused-logic matrix circuit that is faster at age-based scheduling than traditional CAM- or matrix-based schedulers (∼20% faster at 22–36 entries, or twice the capacity at 240 MHz), yet is functionally equivalent. Our careful circuit implementations are substantially faster (∼1.4–6×) than prior work.

Our results show that moderately-aggressive out-of-order soft processors with single-issue schedulers of up to 40 entries are feasible on FPGAs at no frequency loss compared to the small, simple, highly-optimized Nios II/f.

For multi-issue processors, we explored the design space of a 4-way distributed scheduler, and demonstrated that a 6-enqueue/4-dequeue distributed scheduler for complex x86 micro-ops runs at a *higher* frequency than a single-issue scheduler for simpler two-operand instructions of the same total capacity.

The IPC and performance benefit of out-of-order processors is expected to be large, on the order of 2× for a first implementation, and opens the door to even more aggressive designs in the future.

Chapter 10

Register Files and Instruction Execution

After micro-ops are scheduled, they are sent for execution. The register file and execution units are responsible for reading the source operands of an instruction, computing its result, then writing the result back to the register file. Because the scheduler has handled data dependencies between micro-ops, instruction execution is localized and does not need to be aware of the out-of-order nature of the rest of the processor.

The primary objective when designing execution units is to meet clock frequency targets while minimizing the latency (in clock cycles) to compute a result. In a CPU where there is not an abundance of parallelism, the overall performance is sensitive to the latency of an operation. Fortunately, the complex operations that are impractical to implement in a single cycle are also the operations that are less frequently executed. Thus, the frequently-used simple execution units have lower latency than the complex ones.

One challenge in building a processor for a full-featured instruction set (such as x86) is the large variety of operations it needs to support. In addition to arithmetic and memory load/store operations common to essentially all instruction sets, there are also a set of privilege-protection related operations, such as for managing segmentation and paging, changing privilege levels, and exception and interrupt handling. These operations do not occur frequently, so they are not performance sensitive, but still consume area and design effort.

This chapter will first discuss the microarchitectural trade-offs involved in designing the register file and execution hardware before describing the circuit-level designs we used in our design. Microarchitecture design involves designing the structure and cycle-by-cycle behaviour of the execution units. This includes choosing which operations should be executed by which execution unit, the number of execution units, their latency, and how sensitive the overall processor performance is sensitive to each of these parameters. This informs the design of circuits, indicating which circuits should receive more design effort and therefore are designed at the LUT-level and which can be written with less effort using behavioural RTL code.

In this chapter we consider the address generation units (AGUs) as execution units, but the remainder of the memory system will be discussed separately in Chapter 12 due to its complexity.

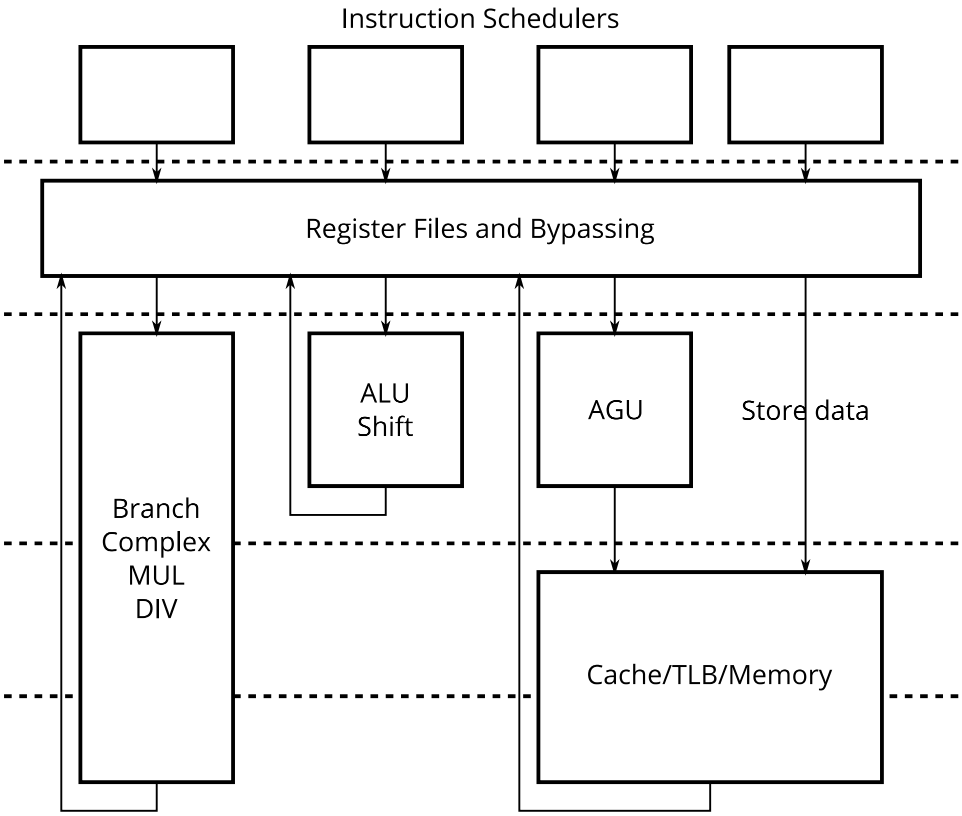


Figure 10.1: Register files and execution units

10.1 Microarchitecture

An overview of our register file and execution stages is shown in Figure 10.1. Because we used a physical register file organization (described in Chapter 7) with a non-data-capturing instruction scheduler, the execution of a micro-op requires one cycle to read its source operands from the physical register file or bypass networks before beginning execution in the following cycle. The operations are then executed by their respective execution units, and may take one or more cycles to do so. Once completed, any results are written back to the register file and optionally bypassed to any operations that may need to use the result immediately without waiting for register writeback to complete.

The instruction schedulers will schedule operations so that data dependencies *can* be satisfied, but it is up to the register file and bypass network logic to find and read data values either from the register file or bypass network. Thus, the bypass network contains comparators to match source operands to operands on the bypass network and select operands from the bypass network with a higher priority than from the register file.

In our design, we have four independent execution units (or execution pipelines). The four execution units are different and are specialized to perform only certain operations with differing latencies. One operation of each type can be sent for execution each cycle.

The branch/complex unit has a minimum latency of 3 cycles and executes all branches, complex operations, and multiplication and division. The simple ALU has a latency of one cycle and executes common arithmetic such as addition and subtraction, bitwise logical operations, and shifts and rotates. The address generation unit handles all memory operations. In x86 memory addressing computations, an *effective address* is computed from register values according to an instruction's addressing mode. The effective address (a segment offset) becomes a *linear address* after the segment base is added, which is then translated by paging to a *physical address*. The AGU is responsible for computing linear addresses from integer registers, displacement, and segment base, and sends the result to the memory system to execute the load and/or store operation. Store data operations do not need execution and are sent directly to the store queue.

Three of these execution units can produce a register or flag result (Branch/complex, simple ALU, and AGU) that is written back to the register file and forwarded. Only one of the execution units (simple ALU) produces a result with a latency of one cycle: the others are slower.

The following subsections will describe in more detail the operation of each sub-unit of the design, and the various trade-offs in their design. These trade-offs include choosing the amount of resources to include, such as the number of register file ports and number of execution units, and more detailed design choices such as the latency of the execution units or subcircuit within execution units.

10.1.1 Experiment Methodology

In this chapter, IPC numbers are measured using cycle-level simulation with a subset of workloads from Table 4.1: SPECint2000, MiBench, Dhrystone, CoreMark, and Doom. This is the same subset used in Chapter 9, and has about 25 billion instructions.

The default microarchitecture we simulated is the one described in Chapter 5 and in Figure 5.1, but with a larger 32-entry unified scheduler. We chose to overprovision the instruction scheduler for the experiments in this chapter because changing execution unit throughput or latency often changes the demand on instruction scheduler capacity. Overprovisioning the scheduler more clearly shows the impact of changing the execution unit parameters without being constrained by having an insufficient number of scheduler entries.

The microarchitecture experiments in this chapter typically start with the default microarchitecture and vary one parameter to measure the sensitivity of performance to the parameter.

10.1.2 Number of Execution Units

As shown in Figure 10.1, our design uses four execution units. These four execution units are specialized, and each execute a different subset of operations. The branch/complex unit executes branches and all "complex" operations such as multiplication and anything related to segmentation, the simple ALU executes common arithmetic in one cycle, the address generation unit (AGU) computes addresses for all loads and stores, and the store data unit executes the data portion of store operations. Table 10.3 lists the operations that are executed by each execution unit.

The two most highly-utilized units are the AGU (loads and stores) and simple ALU, as shown in Table 10.1. As we are building a two-way issue processor, it would be natural to assume that there should be

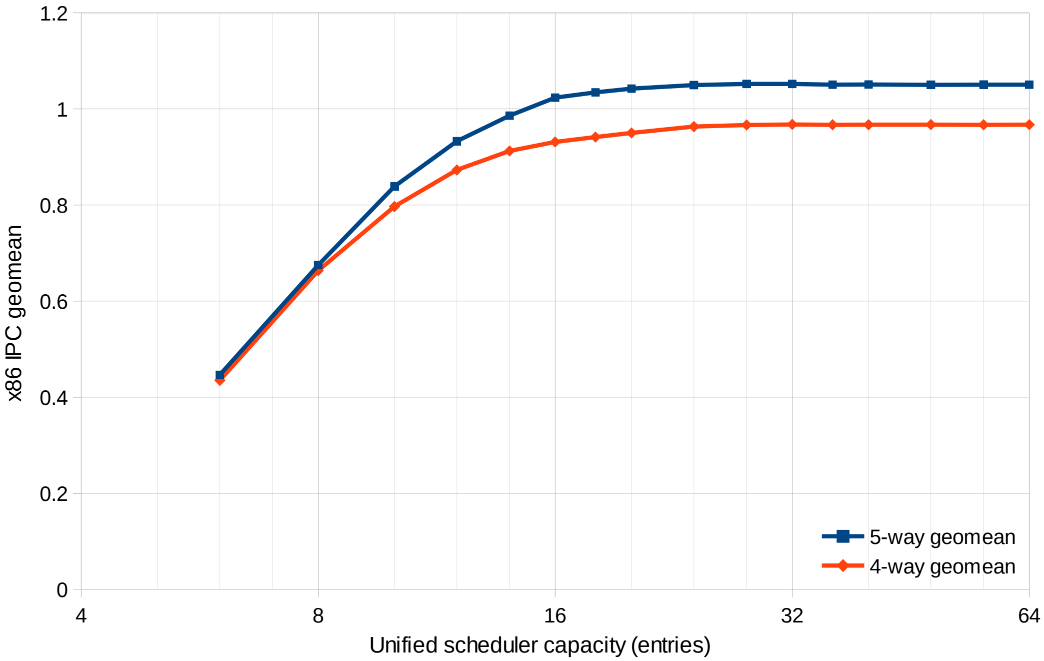


Figure 10.2: IPC of 4-way vs. 5-way execution for varying scheduler size

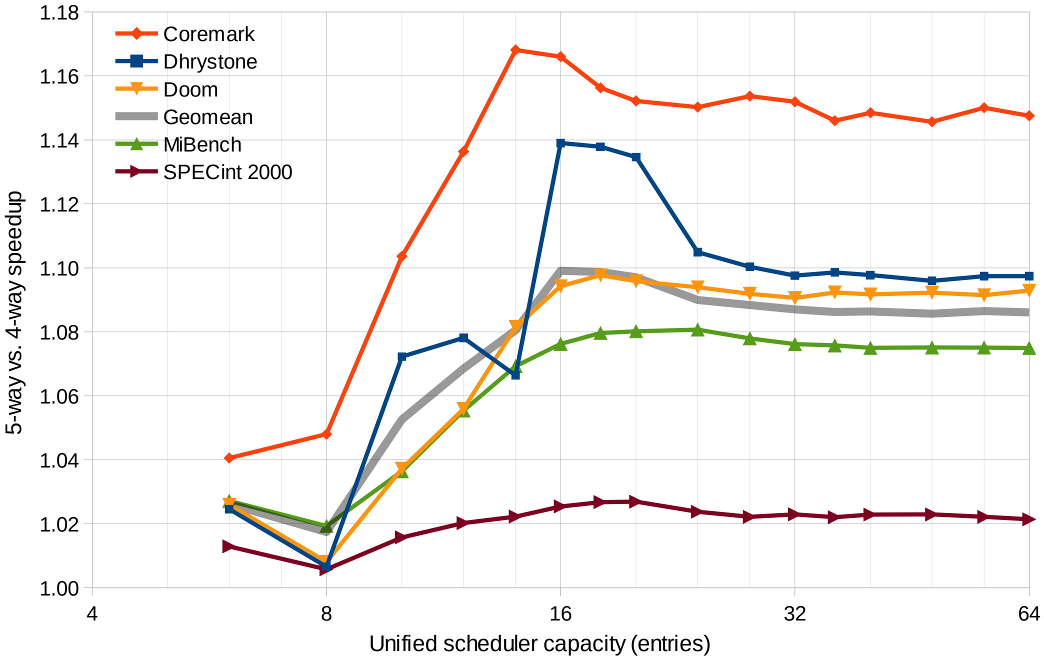


Figure 10.3: IPC speedup of 5-way over 4-way execution, for varying scheduler size

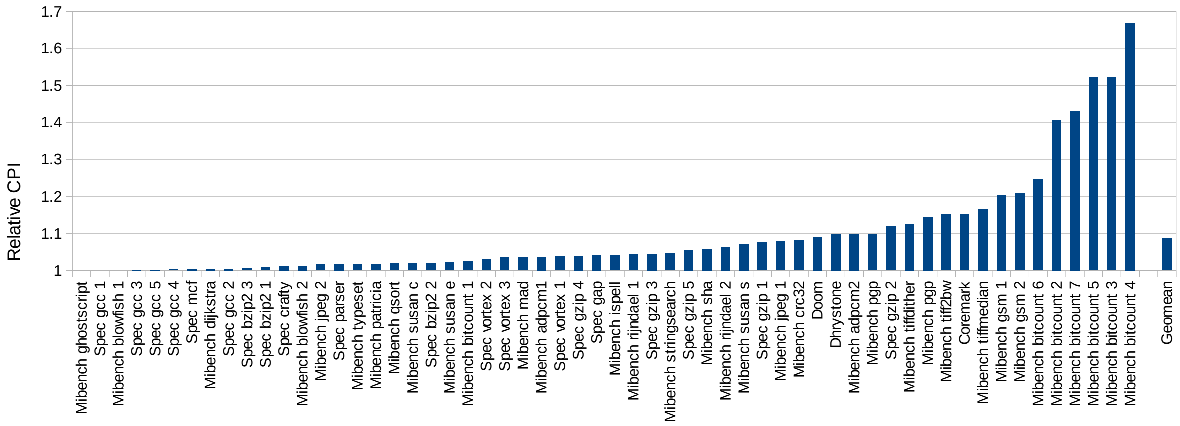


Figure 10.4: IPC speedup of 5-way over 4-way execution at 32 scheduler entries, for each workload

AGU	0.528
Simple ALU	0.419
Branch/Complex ALU	0.132
Store data	0.196

Table 10.1: Execution unit utilization (operations per cycle)

two arithmetic (simple) ALUs, while our design contains only one. Thus, we evaluated the IPC improvement of duplicating the simple ALU for higher arithmetic throughput, though we ultimately decided against this option. Although the memory unit is even more heavily utilized, we did not consider duplicating the AGU because it would require a much more complex dual-ported L1 cache.

Figure 10.2 shows the IPC for our baseline 4-way processor and a 5-way processor with an extra simple ALU, for varying (unified) instruction scheduler sizes. At very small scheduler sizes, IPC is limited by the small schedulers and there is little difference in IPC due to the extra ALU. At large scheduler sizes, an extra ALU provided about 8.6% more IPC. We had anticipated that using more ALUs would require more scheduler entries to find enough IPC to fully utilize the ALUs, but this did not appear to be the case for one extra ALU. IPC for both cases saturates at roughly 24 unified scheduler entries. Figure 10.3 shows the improvement (the ratio between the two curves from Figure 10.2) by benchmark suite. Unsurprisingly, workloads that tend to stress the arithmetic units and not the memory units (CoreMark, Dhrystone) benefit most from an extra ALU, while those that have higher memory system demands benefit less (SPECint 2000). Figure 10.4 further breaks down the benefit by each sub-component of the benchmark suites, at a large scheduler size of 32 entries. Most workloads benefit little from an extra ALU, while a few have a large benefit. The gains are particularly high for the Mibench bitcount routines, as the workload consists of short loops of mainly arithmetic instructions.

The cost of adding an ALU includes widening the instruction scheduler, adding register file read and write ports, and widening the bypass network, in addition to the cost of the extra ALU itself. We decided that in the current design, the 8.6% IPC improvement was not worth this extra cost.

Having decided on the ALUs that will be used in the processor, the following subsections will discuss the microarchitecture of each unit in more detail.

10.1.3 Register files

The first step in executing a micro-op is to gather its source operands from either the register file or bypass network. In our x86 design, we split registers into four types: general-purpose integer, segment registers, condition codes, and immediate values. Other infrequently-accessed registers defined in the x86 instruction set (control registers and model-specific registers (MSRs)) are not renamed and not treated as a "register file". These registers are not performance critical and are accessed by the execution unit as part of its computation after draining or flushing the pipeline.

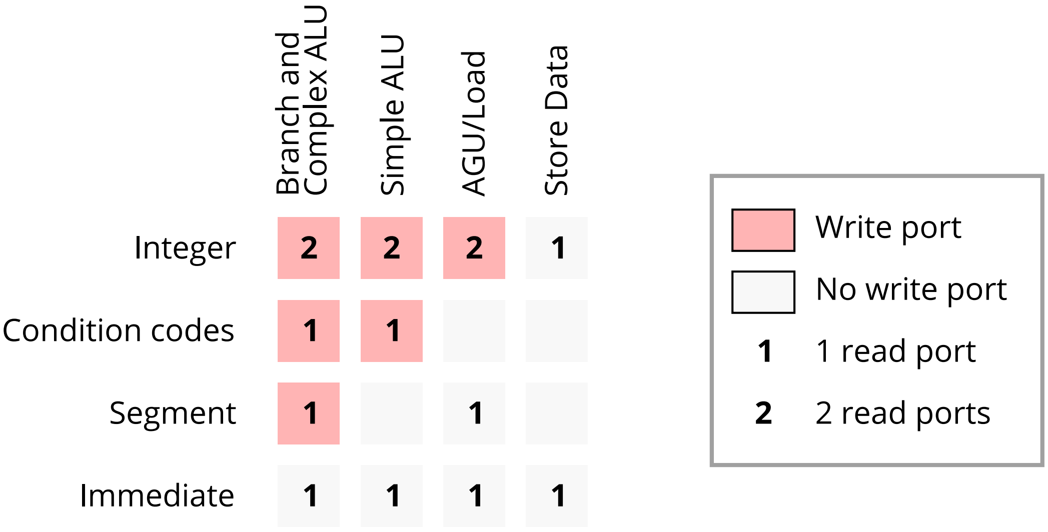
The objective of the register file design is to satisfy register read requests in a single cycle for frequently-accessed registers, while minimizing the number of register file ports required to do this. Microarchitecture design choices that impact the register file include deciding which registers are accessed frequently enough to deserve being treated as a "register file", and assigning operations into specialized execution units so not every execution unit needs to access every type of register.

As we saw in Section 3.4.3, multiported memories are expensive, especially for multiple write ports. Using register renaming (Section 7.2.1) allowed the elimination of multiple write port register file memories, by always allocating destination registers from the physical register file bank that is attached to the execution unit producing the destination value. Different operation types (e.g., arithmetic vs. memory) have different source operand requirements, so the number of register file ports of each type serving each execution unit will differ. We can take advantage of this to further reduce the number of register file ports.

Section 7.2.3 showed that segment registers are rarely modified. Thus, executing all operations that modify segment registers using one execution unit reduces the number of segment register banks (total write ports) to 1. In addition, executing all non-memory operations that read segment registers using that same execution unit reduces the number of segment register read ports to two: One arithmetic unit that reads and writes to segment registers, and the address generation unit (which reads segment registers for use in address computation). A similar situation applies for condition codes: They are never used by address computations and rarely used as a source operand by memory store operations. Stores of condition codes to memory are encoded to first move the condition codes to an integer register before writing the integer register to memory.

Figure 10.5 summarizes the register types read and written by each execution unit in our design. Rows indicate register file type, while columns indicate execution unit type. The number of read ports is indicated by the number in each cell, while the number of write ports is indicated by the colour of each cell (each execution unit writes back at most one result per register type). Using specialized execution units results in fewer register file ports than a hypothetical machine with the same number of execution units where all execution units can execute any operation.

The integer register file is the most complex, with 7 read ports and 3 write ports. Condition codes are only read and written by arithmetic units but not memory units. Micro-ops that manipulate segment registers have been assigned entirely to the complex ALU, which allows the segment register file to have



	Branch and Complex ALU	Simple ALU	AGU/Load	Store Data
Integer	🟥 2	🟥 2	🟥 2	⬜ 1
Condition codes	🟥 1	🟥 1	⬜	⬜
Segment	🟥 1	⬜	⬜ 1	⬜
Immediate	⬜ 1	⬜ 1	⬜ 1	⬜ 1

🟥	Write port
⬜	No write port
1	1 read port
2	2 read ports

Figure 10.5: Register file read and write port usage for each register type and execution unit

just one write port. There are two segment register read ports: One for instructions that manipulate segment registers, and one for use by the segmentation base and limit computations by the AGU. Immediate values are read-only from the perspective of the register file and execution units.

One unusual feature of our microarchitecture is that it allows stores to store an immediate value. This type of operation does not usually exist in RISC-style instruction sets. We chose to allow stores of immediate values because, surprisingly, around 14% of stores are of immediate values. Disallowing this operation would require encoding such operations as a move of an immediate value into a register followed by storing the register, which would consume extra decode, rename, and ALU execution bandwidth.

Future Improvements

There is the potential for further reductions in the number of register file read and write ports, if non-deterministic execution latency is allowed and some amount of arbitration logic is added.

In our design, the number of register file read ports serving each execution unit is sufficient to serve its micro-op in a single cycle. It has been observed that a substantial fraction of operands are actually read from the bypass networks rather than the register files [152, 153], which leads to the possibility of further reducing register file read ports by sharing a smaller number of register file read ports between the micro-ops executing each cycle. We did not consider these approaches due to the design complexity needed to handle the worst-case behaviour correctly.

The number of write ports can also be reduced. In the above discussion on assigning operations to execution units, we considered only whether an operation can or cannot access a particular register type, without considering how often writes will occur. Many instructions do not need to write back results into register files at all (e.g., branches can only flush the pipeline, and store-data operations write into the store queue, not a register file). Because the execution units are specialized for certain types of operations, the

AGU	0.344
Simple ALU	0.419
Branch/Complex ALU	0.008
Store data	0 (never)

Table 10.2: Execution unit writeback port utilization (Integer and/or condition code writes per cycle)

utilization (frequency) of the register file writeback port associated with an execution unit varies greatly between execution unit type. Infrequently-used register write ports can be shared instead of dedicated to further reduce the number of register file write ports (and banks), in exchange for the extra complexity of arbitration.

Table 10.2 lists the register file writeback port utilization for each of the four execution units in our design, defined as the fraction of cycles where a register file write port is in use by the execution unit. These numbers count the fraction of cycles where a general-purpose integer register or condition code register (or both) is written back.

Store-data operations never write back to the register file, and only pass its result to the memory system's store queue.

Almost all (99.97%) of the simple ALU operations write back a result (the writeback utilization (in Table 10.2) is nearly equal to the utilization (in Table 10.1) of the simple ALU itself).

AGU (memory) operations, however, only write back 65% of operations (or 34% of cycles). Only load operations need to write back a register result, but our single AGU processes both load and store operations. This suggests that higher-performance designs may wish to have separate load-only and store-only AGUs, as the store AGU can process 35% of the memory operations without using a register writeback port. Since loads occur roughly twice as often as stores, a well-balanced design might use two load AGUs and one store AGU (which is roughly the design used by Intel's Haswell and Skylake microarchitectures [123]). Load-modify-store instructions are infrequent at around 3% of memory operations, despite being easily encoded in a single instruction on x86.

The branch/complex unit has very few register writebacks (6% of operations, or 0.8% of cycles). This occurs because most of the branch/complex operations are actually branches (89%), which do not produce a register result. The number of writebacks are so few that it would likely be beneficial to share a writeback port with another unit (e.g., load unit) to further reduce the number of register file write ports and bypass networks, at the cost of writeback bus arbitration logic that may impact cycle time or increase execution latency.

10.1.4 Branch/Complex ALU

The branch and complex ALU executes all non-memory operations that access the segment registers, produce or consume a 64-bit value (two registers), or cannot fit into one cycle of latency. Simpler operations that fit into a single cycle are executed by the simple ALU, and are described in Section 10.1.5. We assigned branches to this unit because branches are complex (Branches need to be in-order scheduled in our design because we do not use a branch ordering buffer). Like all of the execution units, the

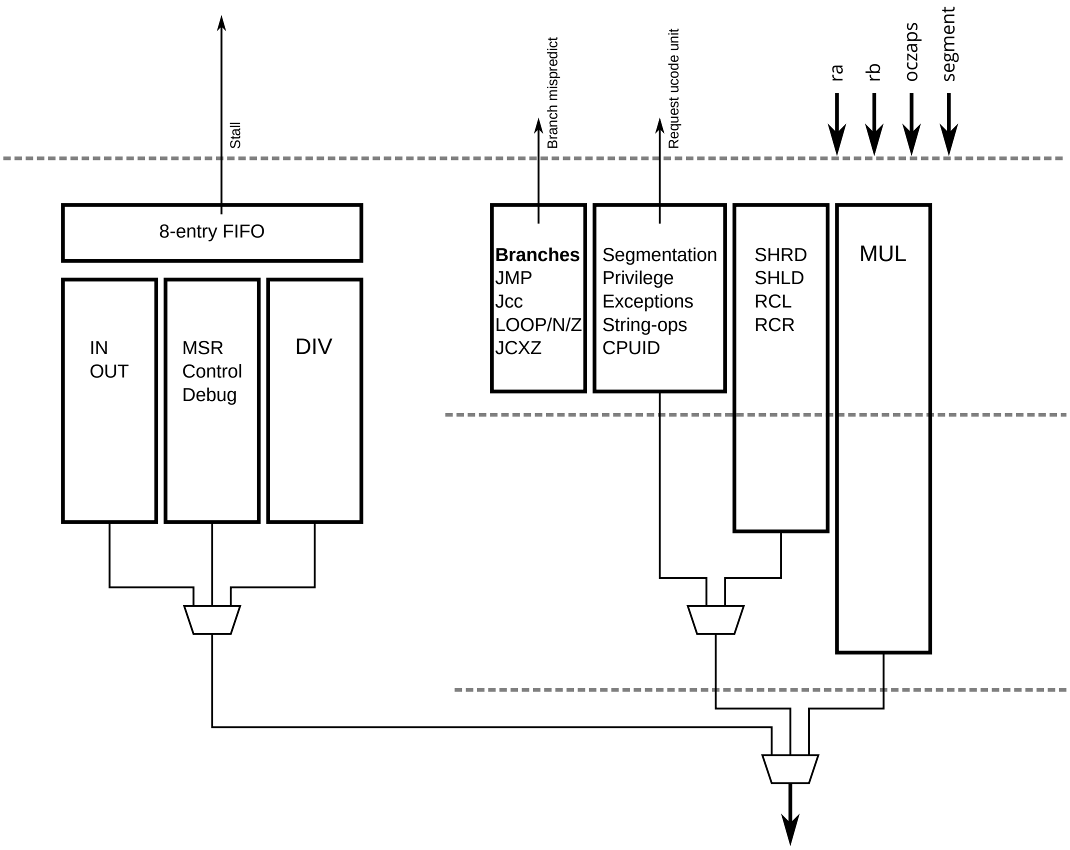


Figure 10.6: Branch and complex ALU

branch/complex unit receives register values and produces a result. It can produce results for all three register types: integer, condition codes, and segment.

The microarchitecture of the branch/complex ALU is shown in Figure 10.6. It is split into a 3-cycle pipelined portion and an unpipelined portion for multi-cycle operations. The ALU can begin execution of one operation per cycle (to either the pipelined or the unpipelined unit, not both). Because the completion times of unpipelined operations are unpredictable, the pipelined and unpipelined operations need to arbitrate for the writeback port, with the pipelined unit having higher priority.

Unlike the other execution units, the integer register result is 64 bits, and one of the integer register inputs (rb) is also 64 bits. The register file bank that is written by the branch/complex unit and read by the branch/complex unit's second source operand is thus widened to 64 bits. This arrangement allows instructions that produce two result registers (such as MUL and DIV) or those that consume more than two registers (such as DIV and SHLD) to be executed out-of-order without the use of a hidden internal register for holding a temporary result or increasing the number of read and writeback ports. The upper 32 bits of the 64-bit physical integer register is architecturally invisible, and are accessed with micro-ops that copy a 32-bit register to or from the upper half of another register.

While this scheme to support 64-bit results may seem more costly than to simply sequence the two writes over two cycles, the latter scheme is actually more complex than it initially appears. Instructions such as MUL with 64-bit results need to write to two different 32-bit destination registers, but a micro-op only has one destination register field. To sequence writes over two cycles would either require doubling the number of destination register fields per micro-op (and either doubling the renamer bandwidth or sequencing renaming over two cycles), or would require decoding these instructions into two micro-ops that must be guaranteed to be scheduled in two consecutive cycles. In contrast, our chosen scheme of widening the register file requires no changes to the renamer or scheduler.

The branch/complex ALU is split into two parts (unpipelined and 3-cycle pipelined). The following sections will discuss the details of each of these parts.

Unpipelined Complex Operations

The unpipelined unit is responsible for executing those branch/complex operations that take either a large number or non-deterministic number of cycles to execute. The unpipelined unit consists of three independent units for executing integer division, I/O (IN and OUT) operations, and operations that access MSRs (model-specific registers), control registers, and debug registers. These operations are implemented as unpipelined operations because they do not necessarily complete in a small and deterministic number of cycles, and do not occur frequently.

Although I/O instructions are functionally related to memory operations in that they make read and write requests outside the processor, we chose to implement them separately from memory operations, as I/O operations are not performance-sensitive, and the memory execution hardware is already very complex. In our design, we treat both I/O and MSR/control/debug accesses as *serializing* operations (requiring pipeline draining before execution), as these operations must be non-speculative (its execution effects are permanent and cannot be discarded).

Operations that execute in the unpipelined unit are enqueued in an 8-entry queue. The main purpose of a queue is to decouple the detection of structural hazards (if a desired unpipelined execution unit is busy) from the assertion of the stall signal. The stall signal is asserted if the FIFO is nearly full. Operations that use the unpipelined units are infrequent enough that an 8-entry queue has not been observed to be full (0.08% of cycles non-empty, 0.0005% of cycles with occupancy more than 1, and maximum observed is 6). Operations are dequeued from the FIFO and sent to its target execution unit (I/O, MSR/Control/Debug, or divider) when the execution unit is not busy. The hardware is designed to allow concurrent execution of different operation types, but this feature is not used in practice because all of the operations except division are always accompanied by a pipeline flush. When the operation is complete, the operation waits until the branch/complex ALU's writeback port is available and uses the empty cycle to write back its result. The execution latency is variable and depends both on the execution time of the operation and any extra delays caused by arbitrating for the writeback port.

Division The latency of the integer divider was explored as it is expected to have some impact on overall processor performance. Divider latency usually ranges from about 1 cycle per bit for radix-2 dividers (32 cycles for 32-bit division) down to a few cycles for dividers with very high radix. It can also increase to hundreds of cycles if division is implemented in a software emulation routine instead. We chose to use

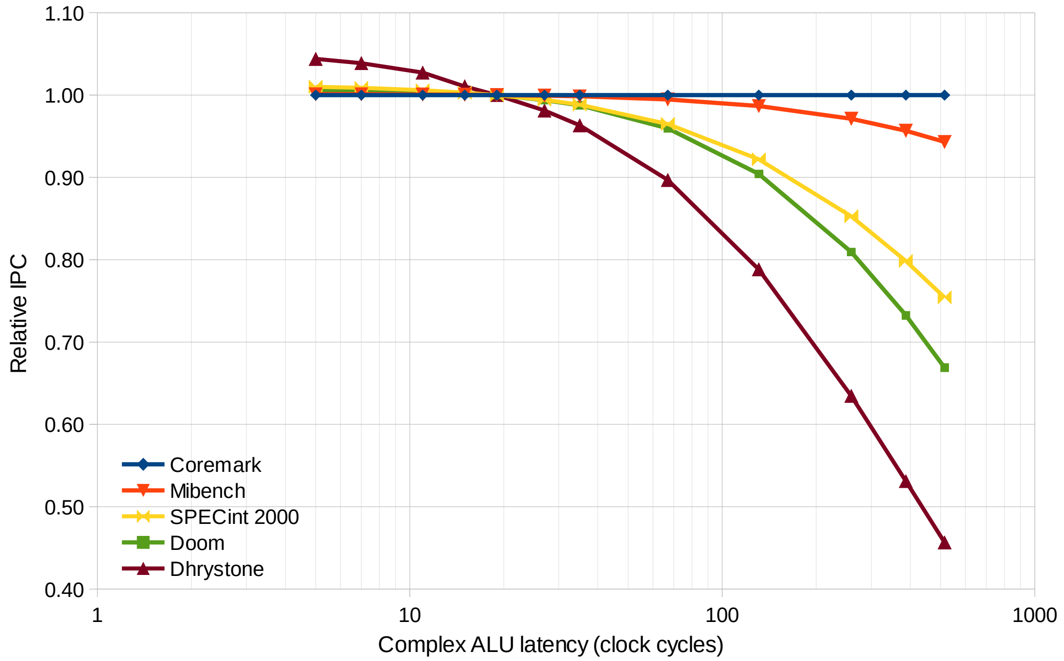


Figure 10.7: Relative IPC for varying integer divider latency

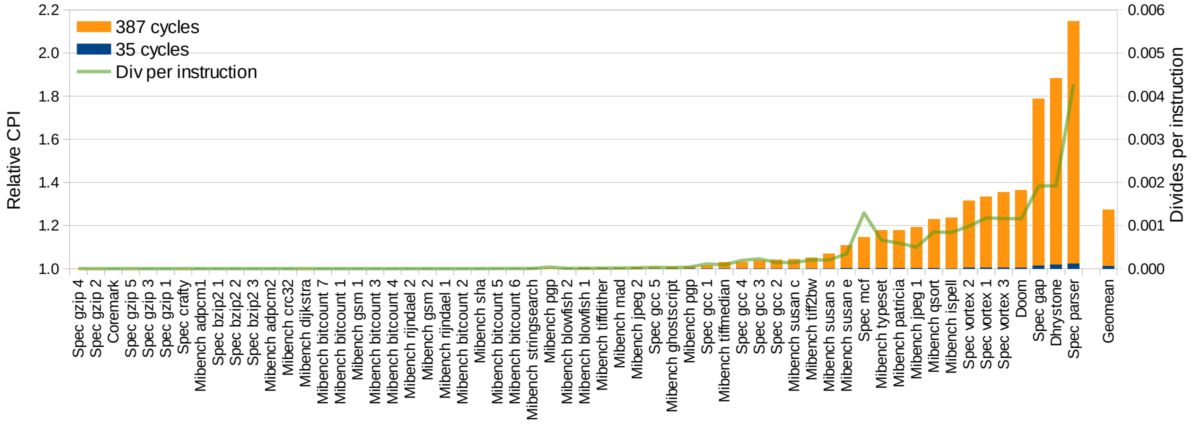


Figure 10.8: Relative CPI for increasing integer divider latency to 35 and 387 clock cycles, compared to 19 clock cycles. Also plotted is the frequency of division operations.

a radix-4 divider with around 19 cycles of latency, and explored the performance impact of varying the divider latency from around 5 to 500 cycles.

The largest x86 division instructions divide a 64-bit dividend by a 32-bit divisor and produce both quotient and remainder in two output registers. Our divider takes 19 cycles (16 + overhead) for most operand values. An x86 division instruction is decoded into a sequence of three micro-ops. A move micro-op first combines two 32-bit registers into a single 64-bit register, then a second micro-op performs division over 19 cycles with two operands: one 32-bit, and one 64-bit. Its result (quotient and remainder) is written as a single 64-bit result. A third micro-op extracts the upper 32 bits of the result. Thus, the remainder is available 3 cycles later than the quotient (because 64-bit moves are executed by the 3-cycle branch/complex ALU).

The choice of the divider's radix (we used radix 4) is a trade-off between circuit complexity, cycle time, and division latency. We chose a radix-4 divider because it was the highest radix for which we were confident that a circuit could be designed to fit in our cycle time target. At the other extreme, some architectures (notably some of the higher-end ARMv7 processors [122]) have optional hardware integer division and implementations without a hardware divider circuit rely on a software emulation routine that takes several hundred cycles. Figure 10.7 shows the sensitivity of IPC to divider latency. Most workloads are not sensitive to small changes in the divider latency (e.g., 35 cycles with a radix-2 divider vs. 19 with radix-4). However, many workloads have a large IPC loss if division latency were increased to a level comparable to software emulation (a few hundred cycles).

Figure 10.8 shows the same data set, further broken down into individual workloads. Many workloads appear to not use division at all, but those that do are severely impacted by 387 cycles of divider latency. However, the impact of an extra 16 cycles of latency (35 vs. 19) is small (worst case 2.4% on SPECint 2000 parser), which indicates there is no great pressure to further increase the radix of the divider circuit to reduce latency below 19 cycles. The sensitivity to divider latency is, unsurprisingly, closely related to how often division operations are used.

One interesting observation (shown in Figure 10.8) is the frequent use of integer division in Dhrystone, while none are used in CoreMark, despite both being small benchmarks aimed at measuring the core arithmetic (not memory system) performance of processors.

Pipelined Branch/Complex

The pipelined portion of the branch/complex ALU executes a variety of complex operations, the most frequent of which are branches (91%). There are also hardware blocks for executing unusual shift and rotates not part of the simple ALU, multiplications, and segmentation-related and other miscellaneous operations. The latency is 3 cycles, and is limited by the multiplier latency. As a result, the other three units are coded in behavioural RTL because they did not need much optimization to fit into cycle time and latency constraints.

The branch unit executes a variety of branches, including unconditional branches, branches conditional on condition codes, and branches conditional on a register generated by the `LOOP` family of x86 instructions. It checks whether the branch target matches the branch prediction and flags a misprediction

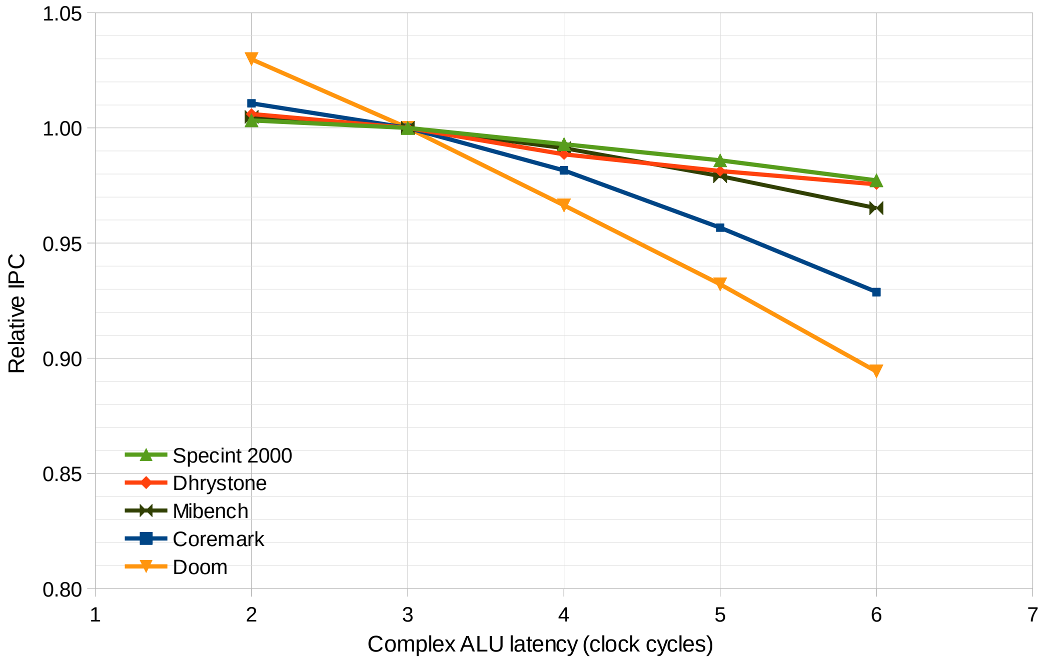


Figure 10.9: Relative IPC for varying branch/complex ALU latency

if necessary. The branch unit also checks the branch target address for staying within segment limits, and flags a fault if the limit check fails. These checks are simple enough to fit in the first cycle of the three-stage branch/complex ALU pipeline. Branch operations do not produce a register result.

The branch unit is designed to execute branches in program order (enforced by the instruction scheduler). This is necessary to avoid the complexity of repeatedly redirecting the front-end if multiple mispredicted branches were discovered in reverse order, where the earlier-in-program-order misprediction must squash the in-flight front-end redirection caused by a later-in-program-order misprediction. Executing in-order avoids this by guaranteeing that if a misprediction is detected, all earlier in-flight branches were correctly predicted and all subsequent in-flight branches are invalid. The mispredicted branch can still be squashed by an interrupt or exception, however.

The segmentation/miscellaneous unit performs a large variety of infrequently-used but fairly simple operations. These execute segmentation-related operations and privilege changes and their permission checks. Operations that need to trigger the microcode unit (Section 5.3.4) such as string operations or interrupt handling are also handled here. Despite the wide variety of operations supported, this unit fits in one pipeline stage.

The shift and rotate unit performs operations that do not fit into the shifter in the simple ALU. These are shifts with 64-bit source operands to support shifting in bits from a register instead of zero, and 33-, 17-, and 9-bit rotate-through-carry where the carry flag is included in the shift. These are pipelined over two cycles.

The multiplication unit performs signed and unsigned multiplication of 8, 16, and 32 bit operands, producing a result of twice the input width.

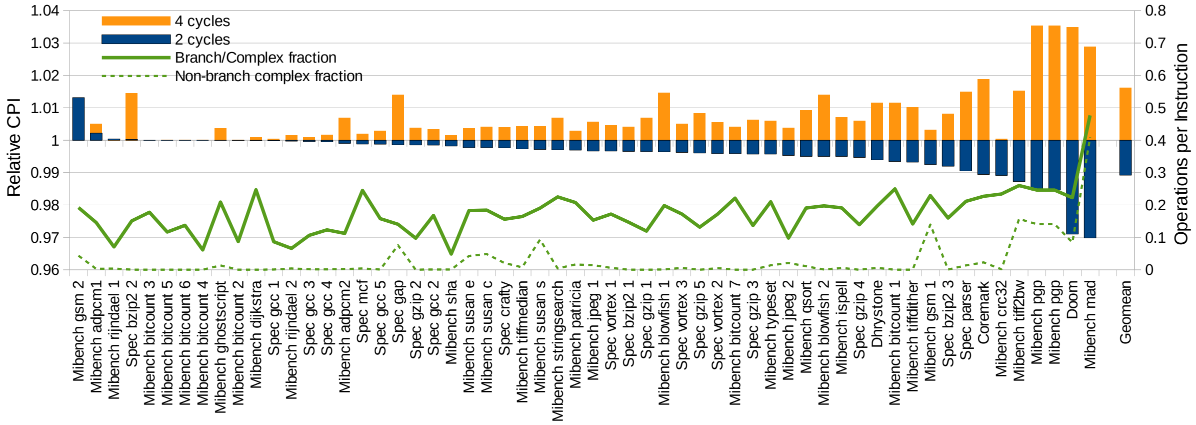


Figure 10.10: Relative CPI for reducing and increasing branch/complex ALU latency by 1 cycle (to 2 and 4 cycles), compared to 3-cycle latency. The utilization (in operations per x86 instruction) of the branch/complex unit (total and non-branch only) are also plotted for each workload, showing utilization is not strongly correlated to latency sensitivity.

To relax timing constraints in the instruction scheduler and bypass networks, it may be beneficial to further increase the latency of this ALU beyond the current latency of 3 cycles. Thus, we evaluated the sensitivity of IPC on the change in the complex ALU latency. Figure 10.9 shows the change in IPC as the latency of the pipelined section of the branch/complex ALU is varied, relative to our current design of 3 cycles. In this plot, branch latency is not varied, as our motivation is to allow more time for complex operations and writeback (branch comparisons are simple and do not write back a result). Most workloads are only slightly sensitive to increased latency, with a loss of about 1.5% IPC for one extra cycle (4 cycles). Figure 10.10 breaks down the IPC impact by workload, for a one-cycle increase and one-cycle decrease in latency to 4 and 2 cycles, respectively. Even the most sensitive workload loses less than 4% IPC for one extra cycle of latency.

In an attempt to find which characteristics of the workloads influence its sensitivity to the complex ALU latency, we also plotted how often each workload uses the branch/complex ALU, and how often each workload uses non-branch complex ALU operations (as only the non-branch operations are affected by the latency change). In Figure 10.10, the utilization (in units of operations per x86 instruction) of the branch/complex ALU is plotted in green lines, for all operations and for the non-branch subset. There seems to be a small amount of correlation between the sensitivity to the branch/complex unit's pipeline latency to total utilization of the unit, but not to the non-branch utilization (which are the operations that are actually affected by the increased latency). Thus, there are likely other factors that determine sensitivity to complex ALU latency beyond how often the unit is used by the workload.

Future Improvements Our branch execution unit currently executes branches in program order. Given the complexity of attempting to do in-order scheduling for certain classes of micro-ops within an otherwise out-of-order system (requiring changes to the instruction scheduler), it may be less complex to allow out-of-order branch execution that is then reordered using a branch ordering buffer. A branch

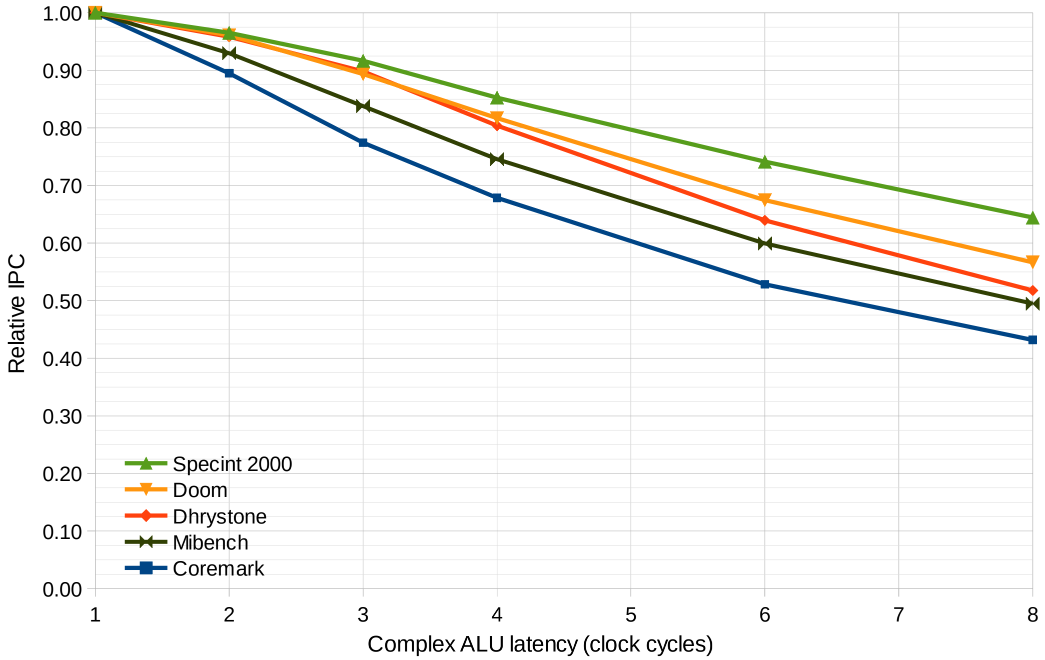


Figure 10.11: Relative IPC for varying simple ALU latency

ordering buffer is similar to a reorder buffer, except it holds only branch operations. It is used to ensure that branch mispredictions and front-end fetch redirections are applied in program order, while allowing branch verification to occur out of order.

10.1.5 Simple ALU

The simple ALU performs single-cycle arithmetic operations. It contains three units: adder/subtractor, bitwise logic, and shifter/rotator. The ALU also generates condition codes in the same cycle. The results are forwarded to dependent operations in the same cycle, which allows back-to-back execution of data-

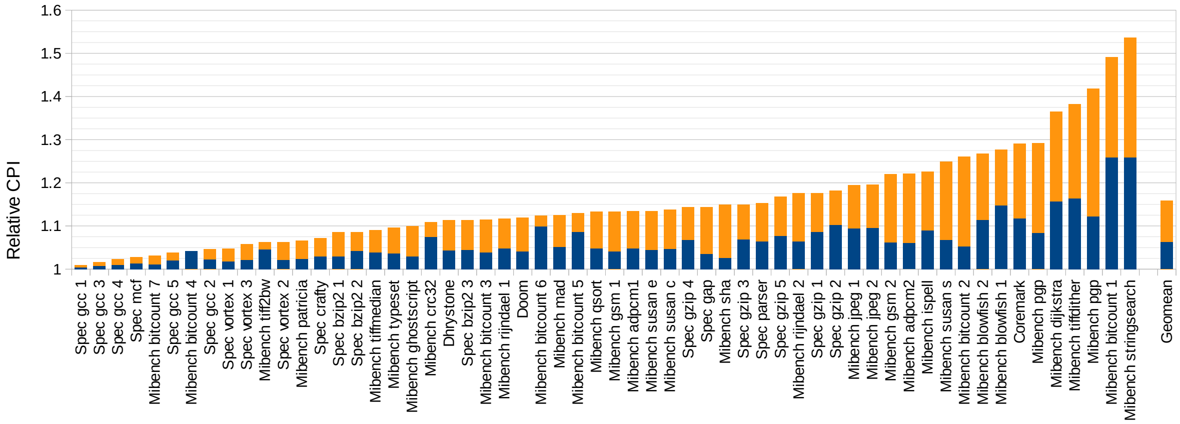


Figure 10.12: Relative CPI for increasing simple ALU latency to 2 and 3 cycles

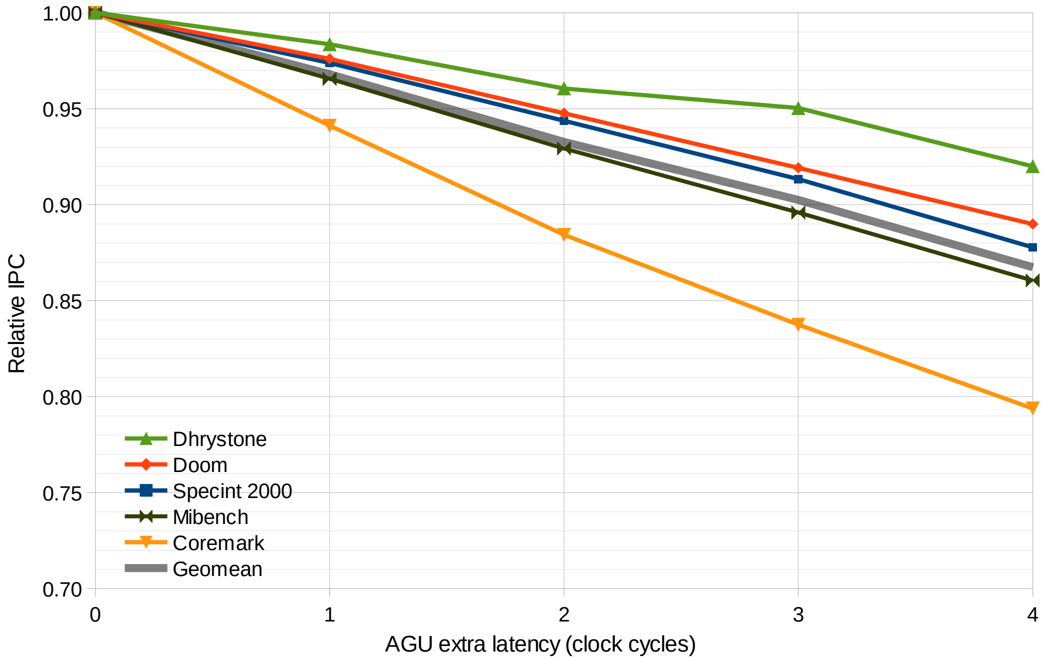


Figure 10.13: Relative IPC for adding latency to the address generation unit

dependent simple-ALU operations.

The critical path in the execution units that limits clock frequency is performing a computation and forwarding the result back to the inputs of the execution units. Thus, we should evaluate the IPC impact of increasing this unit's latency. Figures 10.11 and 10.12 show the IPC impact of increasing the ALU latency beyond 1 cycle. Unsurprisingly, increasing latency for the simple ALU has a larger impact on IPC than for any of the other execution units, at about 6% for 2 cycles and 16% for 3 cycles.

10.1.6 Address Generation Unit

The address generation unit (AGU) computes effective addresses and linear (post-segmentation, virtual) addresses for memory access operations. The operation is then sent to the memory system for load and/or store execution. Loads eventually write back a result to the register file and bypass networks. For x86, the most complex addressing mode is base plus scaled-index plus displacement, which is then added to the segment base address, requiring the AGU to sum four numbers. In the current design, the AGU processes all memory operations regardless of the complexity of the addressing mode used by the instruction. As simple addressing modes tend to be used more often, a potential future optimization is to skip the AGU stage for those operations.

Load, store, and load-modify-store operations are treated as one AGU operation. For load-modify-store operations, the AGU computes the linear address, then uses that address to perform a load *and* writes the address in the store queue. This arrangement has the advantage that load-modify-store operations do not need to be decoded into multiple micro-ops. However, the benefit is limited as load-modify-store operations occur infrequently in typical code (around 3% of memory operations).

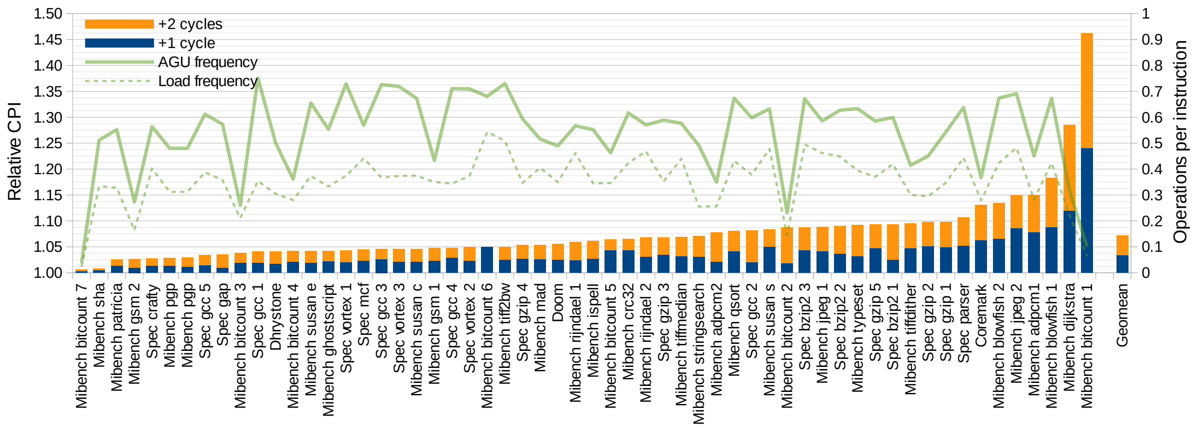


Figure 10.14: Relative CPI for adding one and two cycles of AGU latency. Also plotted is the frequency of AGU operations (total and only loads)

The AGU is currently modelled as a one-cycle operation, but may need to be increased to two stages to satisfy cycle time requirements once the cache and memory execution unit (described in Chapter 12) is added. Increasing the pipeline depth may be required because our single-cycle AGU circuit design (Section 10.2.4) is currently expected to fall far short of our 300 MHz frequency target once the AGU is combined with the cache system. In anticipation of this change, we measured the IPC impact of increasing the AGU latency. Figures 10.13 and 10.14 show the performance loss when the AGU latency is increased (baseline is 0 extra cycles). CPI increases by a moderate 3.3% for one extra cycle of AGU latency. The sensitivity does not appear to be related to the frequency of memory operations or to the frequency of loads (which are more latency-sensitive than stores).

10.2 Circuit Designs

This section discusses the FPGA circuit-level implementation of each block of the execution units. While the microarchitecture design (described above in Section 10.1) considered what operations should be executed on which unit and how many cycles of latency each unit should have, circuit-level design considers circuits that implement the required functionality in the required number of clock cycles and evaluates area and cycle time. In practice, the microarchitecture and circuit design process is somewhat iterative, as the achievable circuit performance influences the choice of execution latency.

The speed of a circuit affects both the achievable clock frequency and the latency of producing a result. While latency (in clock cycles) can be traded for clock frequency by pipelining, pipelining does not improve the latency (in wall-clock time) of a circuit. Reducing the latency (in wall-clock time) of a circuit requires improving the circuit's design.

Single-threaded processors are sensitive to the latency of execution units, as single-threaded code tends to have many data dependencies and not an abundance of parallelism to hide the execution latency. Thus, the execution units need careful circuit design — merely pipelining a slow circuit to achieve a cycle time target is not enough.

Branch/Complex	
IN/OUT	I/O (serialized)
Read/Write Control/Debug registers	(serialized)
Read/Write MSR	(serialized)
Divide	Signed and unsigned, 8, 16, and 32-bit
Multiply	Signed and unsigned, 8, 16, and 32 bit
Insert/Extract upper DWORD	
SHRD/SHLD	16:16 bit or 32:32-bit shift
RCR/RCL	9, 17, or 33 bit rotate-through-carry
Branches	Conditional, unconditional, unconditional with forced pipeline flush
Load segment selector/descriptor	Various forms of privilege checks
Read segment selector/base/limit	
Throw exception	Unconditional, conditional on ZF, conditional on FPU
String operations	REP
Query privileges	ARPL, LAR, LSL, VERR, VERW
Interrupts	Take interrupt and return from interrupt
CPUID	
Simple ALU	
Add/Subtract	With and without carry-in
Bitwise Logical	AND, OR, XOR, ANDNOT
Merge EFLAGS	Merge and extract condition code bits from 32-bit EFLAGS register
Conditional Mux	Implements CMOV instructions
Bit scan	BSF, BSR
Rotate	8, 16, and 32 bit, left and right
Rotate-through-carry by 1	RCL/RCR by 1
Shift	8, 16, and 32-bit, left and right, arithmetic shift right
Sign extension	8-to-16 and 16-to-32 bit
Byte swap	32-bit
Address Generation Unit	
Linear address calculation	Base + scale * index + displacement + segment base
LEA (effective address)	Base + scale * index + displacement

Table 10.3: Operations executed by each unit

Simple ALU	6.2% per cycle
AGU	3.3% per cycle
Complex ALU	1.6% per cycle
Divider	0.08% per cycle

Table 10.4: Sensitivity of IPC to 1 cycle of extra execution unit latency

While speculation can sometimes be used to break dependency loops (which is employed very successfully for predicting branches), doing so for data dependencies (e.g., value prediction [154]) is much more difficult and typically not done in current processors. Since we do not use value prediction, overall processor performance is fairly sensitive to execution latency, as seen in Table 10.4.

In the remainder of this section, we discuss some of the circuits used in the various execution units, with particular attention paid to those circuits that were manually optimized. Optimized circuits include the integer divider, adder condition code generation, and the 32-bit shift/rotate unit. A table summarizing the area and delay of the various circuits can be found in Table 10.6 at the end of this section.

10.2.1 Register File and Forwarding

After being selected for execution, instructions spend one cycle to read its source operands. These operands come from the register files or bypass (forwarding) networks. Figure 10.15 shows the circuit design used to read operands. The design is based on the observation that the forwarding path coming from the execution units (especially from the simple ALU) is far more timing critical than forwarding from the register file or forwarding from the writeback cycle.

A source operand can come from one of three places: from register files, or from one of two sets of forwarding buses. Forwarding occurs in the same cycle as execution (near the end of the cycle), to allow for single-cycle latency. Forwarding also occurs from one cycle after execution (during writeback) as well, because execution results take another cycle to write into the register file. A full cycle is needed to write into the register file (implemented in LUT RAMs) because the Stratix IV MLAB RAMs are not fast enough to write to and read new data from the same location in the same cycle.

Of the three sources of register values, forwarding from the execution unit is the most timing critical. Thus, the forwarding is structured to merge any reads from the writeback forwarding bus and register files first, to keep the final forwarding multiplexer small.

Because the instruction schedulers track source operand *availability* but not where the register value is actually located, this stage is responsible for finding the source operand. It does this using tag comparisons, to see whether a register value being broadcast on a forwarding bus matches the source operand's physical register number. If there are no matches, the source operand value in the register file RAM must therefore be the valid value. When forwarding from the execution unit, it is only the data value that is timing critical, not the tag match. An execution unit already knows its destination register number at the beginning of the cycle and can broadcast that number early, but will not know the destination register's *value* until late in the cycle once execution has completed.

As listed in Table 10.6, the register files and bypassing logic each consume just over a quarter of the area of the entire execution unit. This may seem disproportionately large, but expensive bypass networks are characteristic of wide-issue processors, even for hard processors. For example, the 6-way 12-operand Itanium-2 spends half of its clock cycle bypassing (and only half a cycle for ALU execution), uses 288 register tag comparators, and spends more than half of its integer execution unit area on register files and bypassing [155]. For comparison, our design has 4-way execution with approximately 9 bypassed operands (7 integer and 2 condition codes — segment register operands do not have forwarding), spends about 20% of the critical path on bypassing, and uses 43 register tag comparators.

16-bit and 8-bit registers

The 32-bit x86 architecture supports multiple operand sizes of 32, 16, and 8 bits, with some non-intuitive behaviours. Handling multiple operand sizes requires extra hardware support that complicates operand forwarding.

For 16-bit operands, the operands are simply the lower 16 bits of the eight full-sized registers. However, eight 8-bit registers are packed into the lower *two* bytes of first *four* registers, rather than being the low byte of each register.

There are two issues that need special handling. First, writing to a narrow-sized register leaves the upper bits of the 32-bit register unchanged, and this behaviour requires hardware support. Our design requires the ALUs to read the old destination register value and merge the upper bits with the narrow 8- or 16-bit result being computed. Second, 8-bit high-byte accesses (AH, CD, DH, BH) require accessing the second byte (bits [15:8]) of a 32-bit register rather than the low byte (bits [7:0]), which requires yet more multiplexers, both in the register read path and the register write path. These extra multiplexers cost on the order of 10% increase in cycle time.

A future implementation may instead choose to decode reads and writes of the high 8-bit operands to use an extra micro-op, to allow removing these multiplexers from the critical path. This incurs a delay for instructions that use the four high-byte registers and increases the complexity of instruction decoding, but decreases the cycle time for instruction execution. This is promising because accesses of the four high registers are infrequent, at around 0.0011 reads and 0.0008 writes per instruction.

10.2.2 Branch/Complex ALU

As shown in Figure 10.6, the branch/complex unit is divided into several independent units. The latency of this unit is limited by the DSP block used for multiplication. Since it is simpler for the entire ALU to have a single fixed latency rather than variable depending on operation type, many of the circuits have a generous latency budget of over two cycles. As a result, many of the circuits were not timing critical, and were written in behavioural RTL code and not highly optimized. The remainder of this section will mainly focus on the optimized radix-4 integer divider circuit.

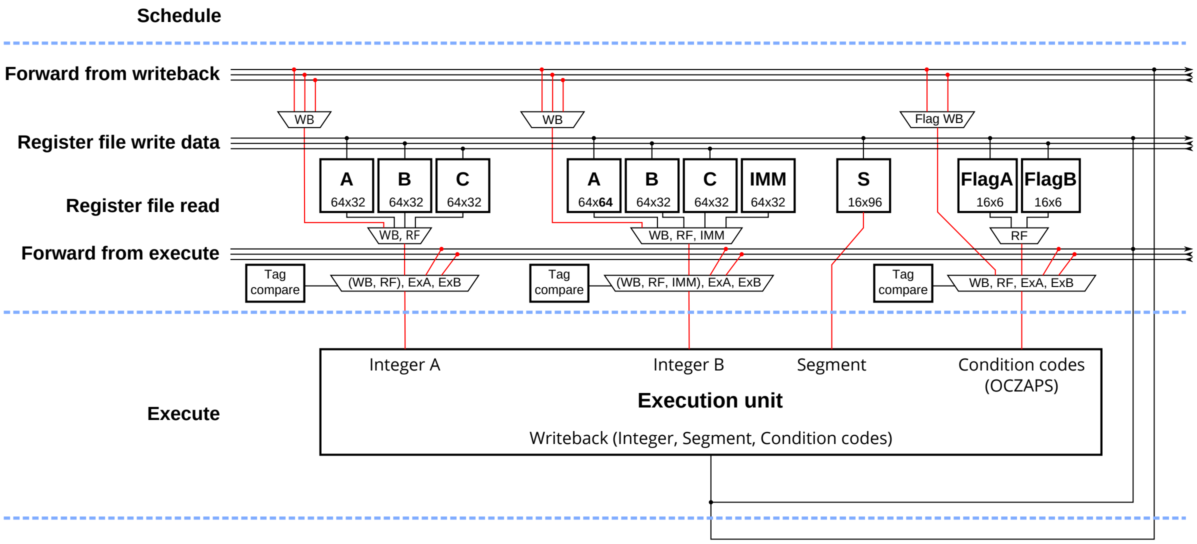


Figure 10.15: Register bypass network and register file circuit. The circuit for one execution unit is shown. This circuit is replicated for each execution unit, with modifications for the required forwarding and register file ports.

Divider

The integer divider circuit performs integer division on both signed and unsigned operands of three different sizes. Our design is a radix-4 non-restoring divider that computes two bits of quotient per cycle, with roughly three extra cycles of overhead (one cycle within the divider, two outside). It computes 32-bit quotients in 19 cycles, 16-bit quotients in 11 cycles, and 8-bit quotients in 7 cycles.

Figure 10.16 shows our divider circuit. The design is based on a standard radix-4 non-restoring divider. A variety of minor extensions were added to allow the divider to handle both signed and unsigned operands, handle three operand sizes, and to detect overflow and division by zero. The first pipeline stage aligns the dividend and divisor to suit the operand size (32, 16, or 8 bit). The next section performs the division itself, and is unpipelined, taking multiple cycles to compute the result. The divider repeatedly subtracts (a multiple of) the divisor from the partial remainder each cycle, generates two quotient bits, and shifts the shift register by two. This process runs a fixed number of cycles based on the operand size (2 bits per cycle). This is followed by one pipeline stage that adjusts the final remainder and quotient, which is required by non-restoring dividers. Overflow is also computed in this cycle.

More details on how our algorithm and circuit implements all the necessary variations of division can be found in Appendix B.

The first row of Table 10.5 shows the achieved frequency and area use on a Stratix IV FPGA for the divider circuit alone. Our divider achieves 346 MHz using 364 ALMs. Both the divider's core compare-and-subtract logic and the final remainder and quotient adjustment logic are nearly equally-critical paths.

This table also compares our divider to several other dividers of various designs. Two of the dividers came from OpenCores [156, 157], while the rest are parameterizations of Altera's LPM_DIVIDE. The comparison is not entirely fair, as none of the other dividers provide the same set of features required for

Circuit	Size	Signed?	Pipelined	Overflow?	Area	$f_{\max}$	Latency	
					(ALM)	(MHz)	clk	ns
Ours (Radix-4)	64/32	both	No	Yes	364	346	18	52
Radix-2 restoring [156]	32/32	unsigned	No	No	103	306	33	108
Radix-2 non-restoring [157]	64/32	unsigned	Yes	Yes	2733	368	33	90
LPM_DIVIDE 32-stage [158]	64/32	unsigned	Yes	No	2174	203	32	158
LPM_DIVIDE 32-stage	64/32	signed	Yes	No	2249	202	32	158
LPM_DIVIDE 16-stage	64/32	signed	Yes	No	2075	113	16	142
LPM_DIVIDE 1-stage	64/32	signed	–	No	1877	8.55	1	117

Table 10.5: Comparison of divider circuits of varying designs

our processor, in particular, supporting both signed and unsigned division at runtime, producing both quotient *and* remainder, supporting multiple operand sizes, and detecting overflow. Some of the designs are fully pipelined, which provides more throughput than required for a typical CPU, at the expense of much higher area compared to multi-cycle implementations.

Our divider uses much less area than the fully-pipelined dividers. Its frequency is higher than most of the other dividers compared, even though our divider is the only radix-4 divider (the others are simpler radix-2). The combination of radix-4 (fewer cycles) and high frequency allows our divider to have low latency (though not high throughput), while providing more functionality. It has 58% of the latency of the next fastest divider, and one third of the pipelined LPM_DIVIDE dividers.

Further improvements in latency can come from division algorithms that improve on the non-restoring division algorithm we used (e.g., SRT), but they are complex and difficult to understand. These algorithms seem more suitable for floating-point division (where the dividend and divisor are normalized) than for integer division. However, there is little incentive to pursue these in the near future because in the current design, overall IPC is not very sensitive to further improvements in the divider latency.

Pipelined Complex Unit

The pipelined portion of the branch/complex execution unit (shown on the right side of Figure 10.6) contains four separate blocks: Branch, complex (segmentation, privilege checks, etc.), shift and rotates, and multiplier. To avoid the complexity of arbitrating or scheduling for access to the writeback port, the four units have the same latency. The latency is three cycles, limited by the slowest unit, the DSP-block-based multiplier. Since most of the circuits are given more cycles than necessary for the computation, the circuits in this unit were fast enough with only a small amount of detailed circuit design.

In the current design, the branch and complex units both complete in one cycle. The shift unit completes in two cycles: one cycle to perform the shift or rotate, with the second cycle used to generate parity, sign, and carry flags. Although when compared to the single-cycle shifter in the simple ALU, it may seem like the complex (unoptimized) shifter is doing more complex shifts than the simple ALU's shifter in one cycle, the complex shifter actually has a larger timing budget. The simple ALU must do shifts, compute condition codes, merge its results with the other units onto the writeback bus, *and* broadcast the result over the bypass network in one cycle, while the shifter here only performs the shift. Computing flags,

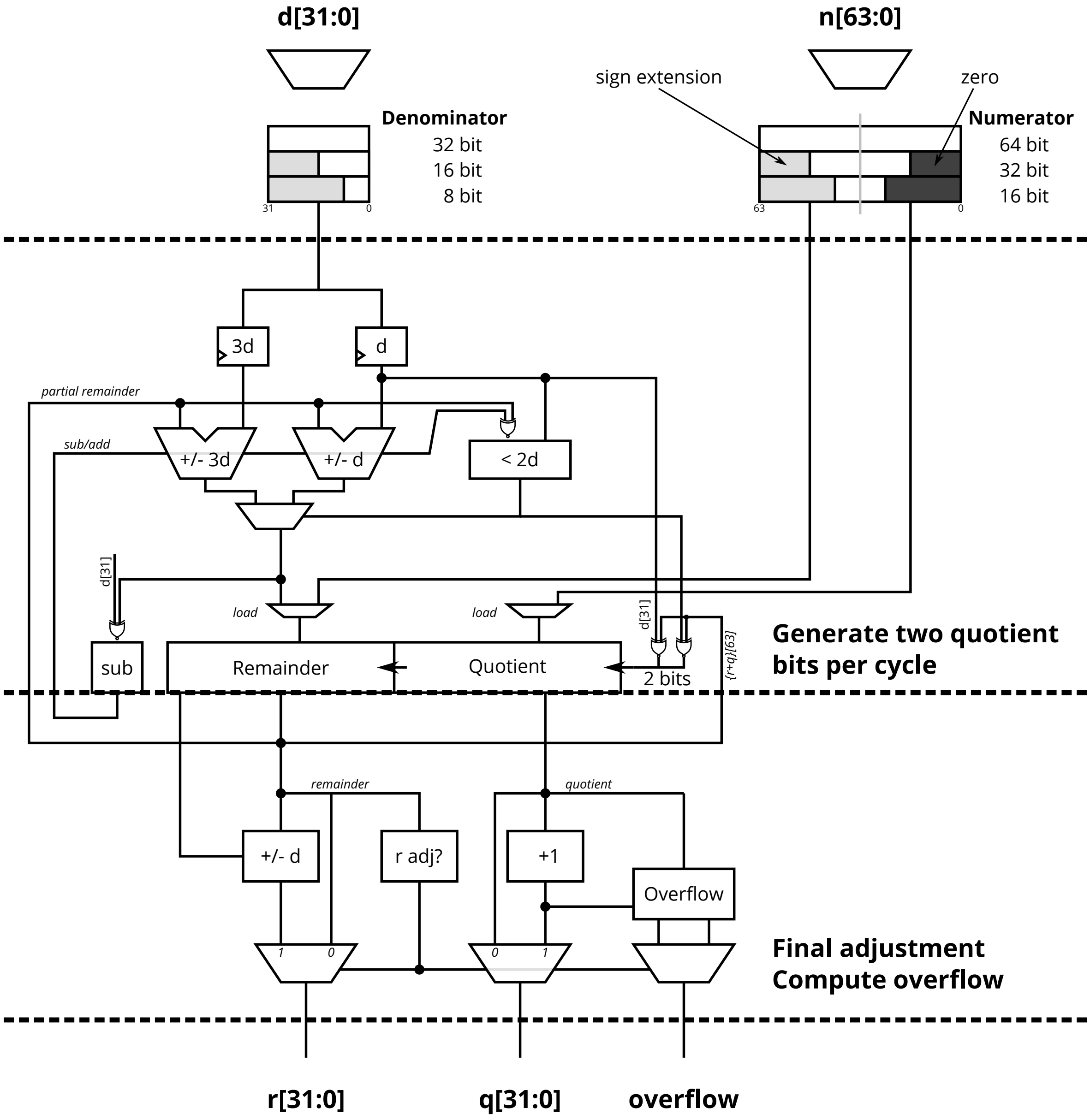


Figure 10.16: Radix-4 non-restoring divider circuit. Handles 64/32, 32/16, and 16/8 signed and unsigned division.

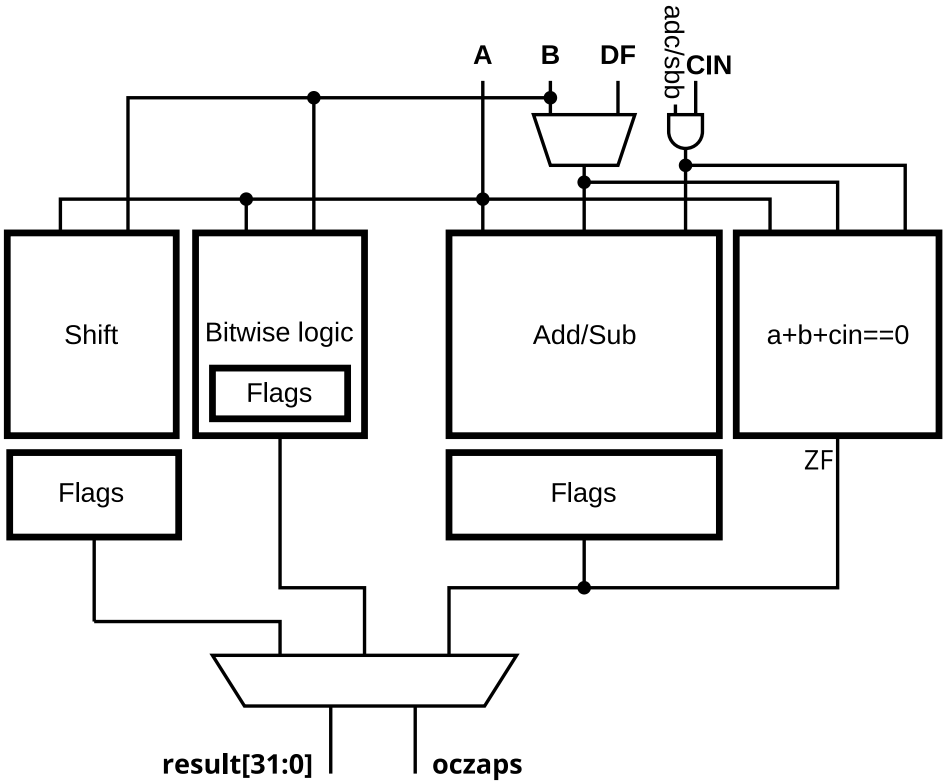


Figure 10.17: Simple ALU circuit

merging the output, and bypassing occurs in another cycle.

The critical path of the pipelined complex unit is through the multiplier, limiting maximum frequency to 279 MHz. We use the DSP block with its internal register enabled to create a two-cycle pipelined multiplier (the third stage is for merging results and bypassing). The pipelined complex portion of the branch/complex ALU uses around 820 ALMs.

10.2.3 Simple ALU

The simple ALU (Figure 10.17) executes arithmetic operations with a single-cycle latency, allowing data-dependent operations to execute in consecutive cycles. Due to the single-cycle latency for performing the computation *and* forwarding the result to any dependent operations, the simple ALU has stringent timing constraints and significant effort was spent to optimize its circuits. It contains three major sub-blocks: Shift/rotate, (mostly-) bitwise logic, and an adder/subtractor.

The simple ALU has a latency of 3.4 ns, not including the delay of the bypass multiplexers that are located in the same cycle. The final cycle time, limited by a critical path that includes both the simple ALU and the bypass multiplexers, will be somewhat below our target of 300 MHz. (We will see in Section 10.2.5 that this complete path is 239 MHz.) The simple ALU uses 509 ALMs, with more than half going to the shifter.

The following subsections describe the detailed circuit design of each of the three sub-components

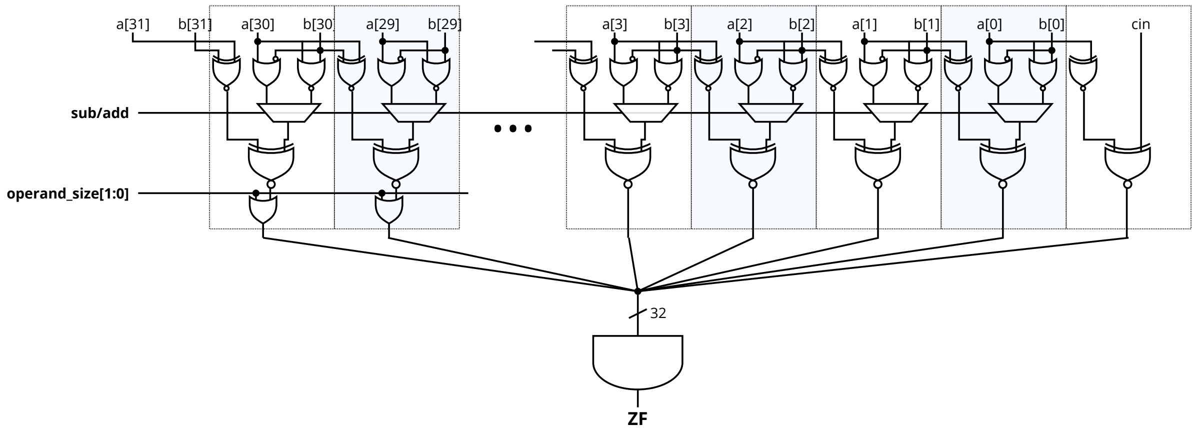


Figure 10.18: Adder zero-flag computation circuit. ZF is computed in parallel with the adder.

of the simple ALU. Due to timing constraints, all three units have significant circuit-level optimizations. Where practical, we also attempted to compare the area and frequency difference between our optimized circuits and circuits with the same functionality synthesized from behavioural RTL code.

Adder/Subtractor

The adder/subtractor itself is built from a standard FPGA carry chain, but computing condition codes was not as straightforward. Due to timing constraints, we try to compute condition code values in parallel with the addition/subtraction instead of after.

Using the FPGA's carry chain to implement an adder/subtractor is the lowest-area implementation method, and was sufficiently fast at 32 bits. For an adder only 32 bits wide, it is difficult to make large improvements over the standard implementation that uses hard carry-chains on this FPGA. This may be different on other FPGAs depending on the relative delay of the carry chain compared to a LUT.

Implementing condition codes is more difficult. Condition codes are conceptually computed based on the result of the (addition) operation. As some of the condition codes are wide functions (e.g., the zero flag) and thus slow to compute, we wanted to find ways to compute the condition codes in parallel with the adder rather than in series.

There are 6 condition codes: Overflow, Carry, Zero, Adjust, Parity, and Sign. Of these six, four are mostly-straightforward outputs from an adder. Carry and Sign flags are signals directly from the adder. The overflow flag is computed using the sign bits of the operands and result rather than tapping the second-last carry-out bit of the adder. Due to having three possible operand sizes, the carry, sign, and overflow flags select from three possible positions in the adder (bits 31, 15, and 7).

The Adjust flag is the carry-out of bit 3 of the adder (it is used for 8-bit, two-digit, binary-coded decimal arithmetic). We implemented this by tapping the carry chain, by extending the carry chain from 32 to 33 bits and performing a "1+0" operation at bit position 4 to extract and propagate the carry-out from bit 3. However, many other implementation methods could work equally well.

The parity and zero flags are substantially more difficult: The x86 parity flag is defined to be the

XNOR of the lower 8 bits of the adder result, while the zero flag indicates whether the result is zero, which is a NOR function of all 32 bits. We implemented the parity flag using the straightforward method of computing a function of the lowest 8 output bits (requiring two logic levels when using 6-input LUTs).

The zero-flag (ZF) computation is slower than the parity computation, as it is a function of all 32 bits of output *and* the current operand size (we need to compute the correct ZF value for 8, 16 and 32-bit additions). While computing a 32-bit ZF (using a 32-bit NOR gate) can be packed into 2 logic levels (using 7 LUTs), a ZF computation that needs to handle three operand sizes requires 3 logic levels and 9 LUTs. To improve speed, we computed the ZF result (a function of 68 inputs) in *parallel* with the adder instead of in series, in three logic levels. This is possible because computing the comparison $A + B = K$ for constant K ($K = 0$ in this case) can be done with only one reduction operation, without carry propagation (It is unnecessary to do *both* a carry propagation and a wide-OR reduction) [159]. This technique has also been previously applied to sum-addressed caches, where an addition is fused with a memory row decoder [143]. Figure 10.18 shows our circuit for computing ZF. There is one layer of 32 6-LUTs (shown as rectangles) derived using the equations from [159], with some modifications to support two operations (addition *and* subtraction) and for omitting portions of the comparison when narrow (8- or 16-bit) operand sizes are used. This is followed by a 32-input AND gate (2 logic levels) for a total of 3 logic levels to compute ZF. Thus, this ZF circuit moved 3 logic levels of delay (which is similar to the adder delay) in *series* with the adder to 3 logic levels of delay in *parallel* with the adder. Computing ZF in parallel consumes more area, however, requiring 39 ALMs, compared to around 9 ALMs that would be required for computing ZF based on the adder output.

The parallel zero-flag computation improves the speed of the adder by 24% compared to the same circuit with the zero flag generated by a 32-input NOR gate that compares the result to zero (2.3 ns vs. 2.9 ns), with a roughly 30-ALM increase in area (79 ALM vs. 51 ALM).

The final adder/subtractor circuit thus resembles the block diagram in Figure 10.17: There is a carry-chain based adder/subtractor, four condition code outputs that are simple functions of a few output bits, a somewhat larger 8-input block for computing the parity flag in series with the adder output, and a larger block for computing the zero flag without using the adder's output.

Future Adder Improvements The parity flag is currently computed as an XNOR of the lower 8 bits of the result of the adder. This could be improved by computing the parity flag in parallel with the adder, as a function of around 18 inputs (two 8-bit sources, carry-in, and whether to use the carry in) in two logic levels in parallel the adder instead of in series. However, we did not implement this improvement because the parity flag computation is currently not critical (the parity flag computation through the shifter is slower).

Bitwise Logic

The bitwise logic unit performs bitwise operations (AND, OR, XOR, ANDNOT), several forms of bit copying (insert and extract condition code bits from the 32-bit EFLAGS, conditional mux), and bit scan. The design of this unit was fairly straightforward as logic synthesis performs well when each output bit

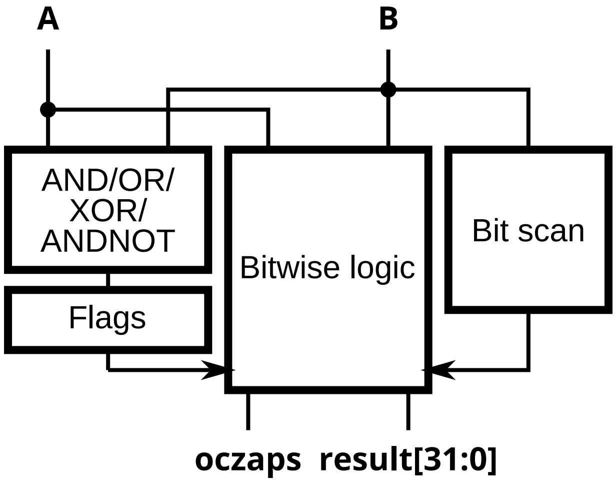


Figure 10.19: Bitwise logic unit. Flags are computed in parallel with the result.

is a function of only a few input bits (typically the two corresponding operand bits and control bits to indicate which operation to perform). Condition codes, like for the adder/subtractor, are designed to be computed (somewhat) in parallel with the data result, where practical.

Figure 10.19 shows the structure of the bitwise logic unit. The bitwise logic unit uses 120 ALMs and is the fastest of the three units in the simple ALU, with a delay of around 2.3 ns.

The standard condition codes (i.e., a function of the computation's result) are generated by only four of the simplest operations: AND, OR, XOR, and ANDNOT. We use this observation to speed up computation of the condition codes. Instead of computing condition codes based on the final result of the bitwise logic unit, we redundantly compute the result of one of those four operations (i.e., a simplified version of the result that is correct only if AND, OR, XOR, or ANDNOT is executed), then compute the condition codes based on this simplified result. This arrangement allows condition code computations to avoid being delayed by the slower parts of the unit that do not produce condition code results, such as the conditional mux operations or the 32-bit bit scan.

Computing the condition codes separately and partially in parallel moves condition code generation entirely off the critical path. This increases the speed of the bitwise logic ALU circuit by 25% (2.3 ns vs. 2.9 ns), at the expense of some extra logic (120 ALM vs. 104 ALM).

The bit scan operation is composed of 8 4-bit selectable-direction priority encoders (5-input LUTs) followed by a two-layer tree of selectable-direction 3-input priority multiplexers (7-input LUTs). This design is slightly (3%) slower than a simple behavioural implementation written using Verilog for-loops (1.85 ns vs. 1.79 ns) despite having fewer logic levels (3 vs. 4). However, the custom design uses fewer resources (30 ALM vs. 40 ALM).

Barrel Shifter

The barrel shifter is responsible for executing 8-, 16-, and 32-bit shifts and rotates. It is also responsible for several other operations that can be implemented using the same set of multiplexers, such as byte swap and sign-extension operations. x86 also has 9-, 17-, and 33-bit rotates, which we implemented in the slower complex shifter execution unit (Section 10.1.4).

This is the slowest of the three hardware blocks in the simple ALU, so it has been carefully optimized for speed. Figure 10.20 shows our circuit design. The datapath consists of three levels of LUTs implementing multiplexers. Control logic is matched to the data path so that the path from any control signal to the output is usually at most three levels of LUTs. As a result, the control logic for the later stages is more complex than for earlier stages.

The structure of the shifter is based on a 32-bit right-rotator that is then masked to produce the other types of operations. A 32-bit rotate function requires at least three levels of logic to implement, as a 6-LUT can at most implement 4-to-1 multiplexing. The first two layers are 7-input functions implementing 4-to-1 multiplexers that implement rotation by 0–3 and by 0, 4, 8, or 12. The third layer uses a 6-input LUT to implement a final 2-to-1 multiplexer to rotate by 0 or 16 along with combining the mask bits from the control logic. It is straightforward to see that this datapath of three layers of multiplexers can implement 32-bit rotates. One of the challenges in designing the shifter is to also implement all of the other operations using the same datapath structure without increasing the logic depth.

Our rotator rotates right by default (this decision was arbitrary, and rotating left is equivalent). To implement left shifts and rotates, we rotate right by the negation $(32 - n)$ of the amount (labelled `amount_adj`). Performing this negation is distributed. In the first layer, the negation of the lowest two bits is packed into a 7-input function by adding an extra "direction" input. Changing the direction causes a rotation of 0, 3, 2, or 1 bits instead of 0, 1, 2, or 3. The negation for later stages is not as easily computed, but we have a higher timing budget of 1 LUT for the second stage, and 2 LUTs for the third stage.

Byte swap is implemented by slightly changing the rotate amount control signals (`amount_adj`). Only 32-bit byte swaps are handled here. A 16-bit byte swap is implemented as a 16-bit rotate by 8, requiring no special support in the shifter. Byte swaps do not rotate in the first layer. They always rotate right by 8 bits in the second stage (transforming `ABCD` to `DABC`). This is followed by a rotate-by-16 in the third stage, but only for even numbered bytes (transforming `DABC` to `DCBA`). This is implemented by having two separate sets of `amount_adj2` signals, one for even bytes and one for odd bytes.

Shifts are implemented as rotations in the datapath, but are masked to zero out the bits that were supposed to be shifted in. This mask can be generated in less time than a true shift operation because the data value is a constant rather than variable. Left shifts and logical right shifts always set the masked bits to zero. Arithmetic right shifts are implemented by allowing the mask to have two values: A 32-bit mask indicates which bit positions need to be masked, and a single mask_value signal indicates whether the masked bits should be set to zero or one.

This masking mechanism enables implementing sign extension and rotate-through-carry-by-1 (RCL/RCR by 1) operations. Sign extension (8 to 16 or 16 to 32 bits) is a rotation by 0, but with the upper result bits masked, with the mask value set to the sign bit of the smaller (source) value. Rotate-through-carry-by-1 is implemented as a rotation (left or right) by 1, with the shifted-in bit masked out, with a mask_value set to the carry flag. Rotate-through-carry is defined as a 33-, 17-, or 9-bit rotation, so a variable rotation amount or a constant rotation amount greater than one is handled by the complex shifter (Section 10.1.4).

Supporting narrow (16- and 8-bit) rotates require further changes, since shifts and rotates of narrow operands are not equivalent to performing the same operation on a full-sized 32-bit shifter and truncating the result. We assume that the upper bits of a narrow operand are always zero (rather than don't-care), but that the shifter output bits are don't-care for the upper bit positions. Narrow rotates are treated as 32-bit rotates, but with the source operand replicated (2 or 4 times). This replication is performed in the second and third LUT layers of the datapath. Enabling `replicate_bits` causes the LUT to perform a *bitwise OR* of two inputs instead of the usual *selection* of one of its inputs. If we assume the unused bits of the source operand are zero, then this operation can be viewed as a rotation (by 8 and/or 16) *and* leaving behind an un-rotated copy of the data.

Like other arithmetic operations, the shifter also produces condition codes values based on the result of the operation. Given the complexity of the shift unit, it is difficult to compute the condition codes in parallel with the data result. However, our design does improve on the straightforward serial implementation.

For the carry, overflow, and sign flags, the flag is set according to one or two bits of the result of a shift or rotate. These bits are not affected by the mask, as it is either a bit that has just been shifted out or about to be shifted out, both of which belong to the portion of the shift value that is not masked. We observed that the third layer of the datapath only spends 3 inputs of 6-input LUT performing the 2-to-1 multiplexing, with the other inputs used to manage the mask. Thus, we implemented carry, overflow, and sign flag logic by replicating the multiplexing functionality of the third datapath layer (using 3 LUT inputs) and used the other inputs (no longer used for the mask) for flag generation logic. This was enough to make these three flags non-critical.

We computed the zero flag by generating a shifted mask suitable for the operation and operand size, then applying the mask to the non-shifted source operand. Generating a shift of a constant is slightly less complex than shifting a variable value, so the zero-flag generation circuit ends up faster than if the final shifter result were fed to a NOR gate. This method results in the zero flag being computed in 5 logic levels with the final logic level being having just 2 inputs (which can be combined with a later multiplexer), while using a NOR gate would require 6 logic levels (3 for the shift and 3 for a 32-bit NOR gate that supports multiple operand sizes). Also, the zero-flag generation logic is now separated from the shifter datapath, which would likely improve routing. Despite this, the zero flag remains the slowest of the condition codes, tied with the parity flag.

The parity flag is the XNOR of the lowest 8 bits of the final output. Since the low 8 bits of the output is affected by all of the inputs *and* the mask, we did not find a promising method of improving delay. We implemented parity flag generation by attaching an 8-input XNOR gate (2 logic levels) to the final output of the shifter. The parity flag is tied with the zero flag for being the slowest.

The shifter uses 258 ALMs with a latency of around 2.9 ns (or 345 MHz if the shifter were the only logic in the clock cycle). To evaluate the benefit of the custom circuit design, we compared the final design to a design with the same functionality coded entirely using moderately-optimized behavioural Verilog. This code was based on an earlier version of the design before we custom-designed the datapath, so they

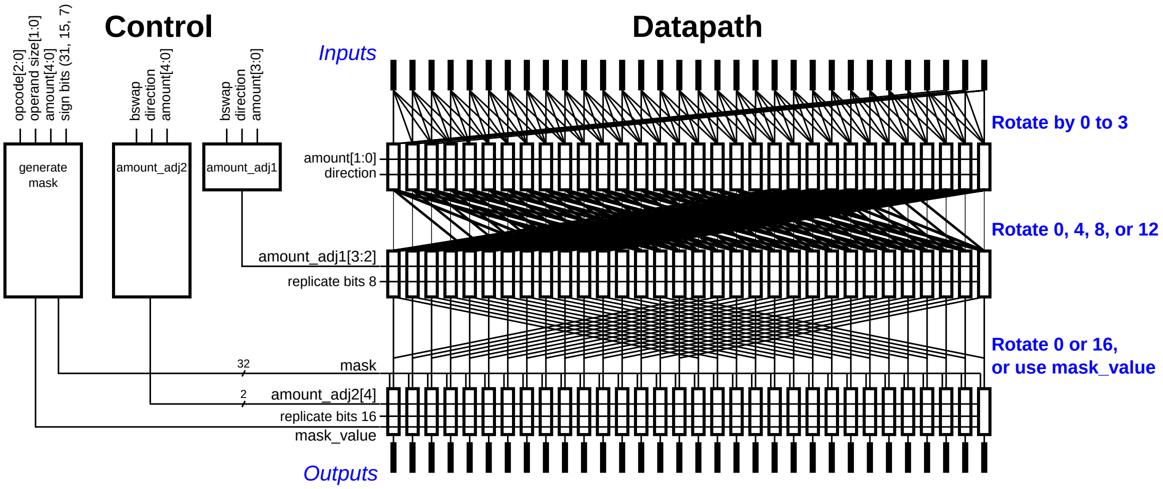


Figure 10.20: Barrel shifter, rotator, sign-extender, and byte-swapper circuit

are functionally equivalent. The custom-designed version is 54% faster (345 MHz vs. 224 MHz) and 46% of the size (258 vs. 561 ALMs) of the behavioural version.

10.2.4 Address Generation Unit

The address generation unit (AGU) computes addresses for all load and store operations. In x86, there are three address computations due to the use of both segmentation and paging. First, an *effective address* is computed based on integer registers and an instruction's addressing mode. The effective address added to the instruction's selected segment's base address (i.e., after segmentation) is a *linear address*. The linear address is then translated using paging into a *physical address*. The AGU is responsible for computing a linear address. The cache and memory system (described in Chapter 12) performs paging and memory accesses.

In addition to computing the linear address, the AGU must also perform segmentation permissions checks and flag a fault if necessary (Paging permissions checks are performed by the paging system). These checks involve checking the segment types and privilege levels, as well as whether the memory access will exceed the selected segment's limit. Our implementation omits some of these checks, as some of the checks are not used by most operating systems, which tend to rely on paging permission checks more than segmentation checks.

Figure 10.21 shows the current design of the AGU. The logic on the left side of the diagram computes the linear address. The more complicated logic on the right performs (simplified) segmentation permission checks.

An effective address is computed using a three-input adder to sum the base, scaled-index (index multiplied by 1, 2, 4, or 8), and displacement. The effective address is then truncated to the instruction's address size and added to the segmentation base address. The need for truncation of the effective address prevents further optimization of the adder structure (e.g., by adding the segmentation base earlier). When using 16-bit address sizes, only the lowest 16 bits of the effective address are used (upper 16 are zeroed),

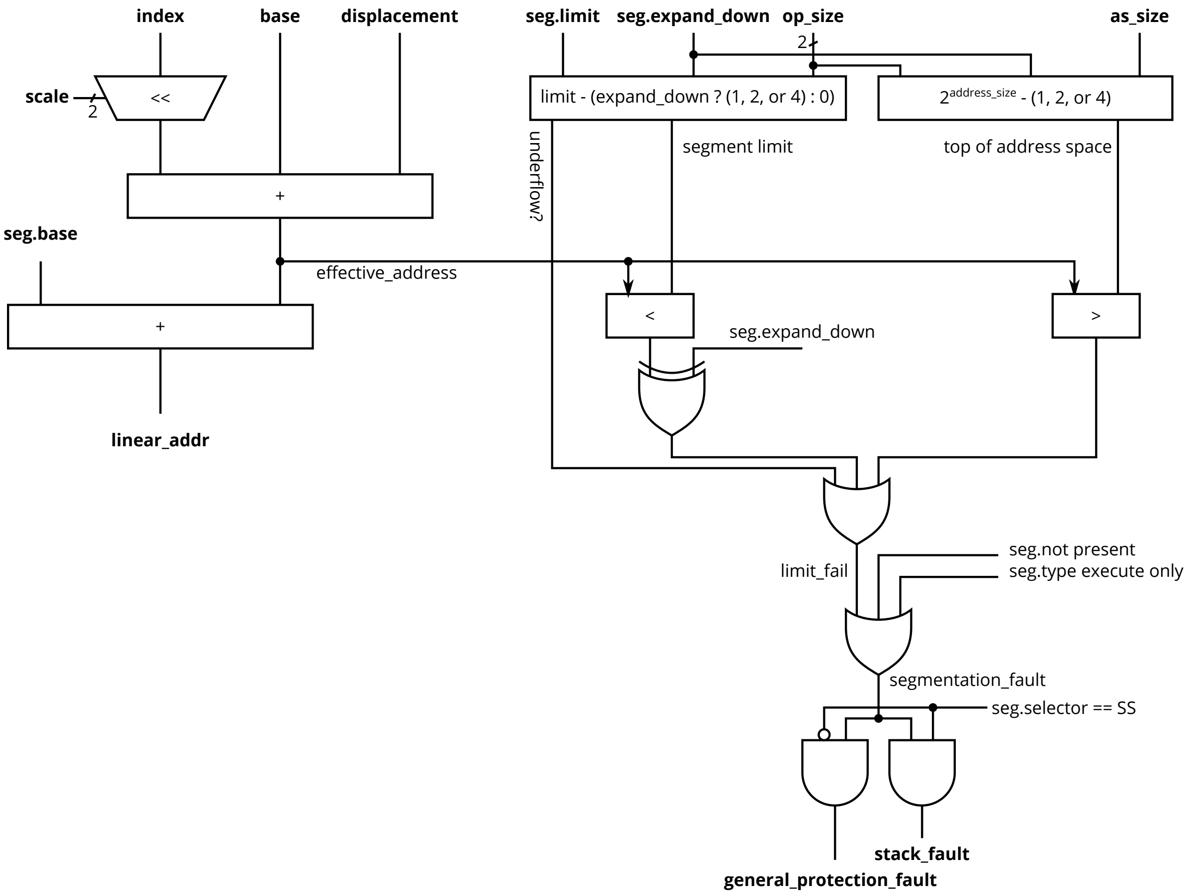


Figure 10.21: Address generation unit circuit. The AGU computes linear addresses and performs (simplified) segmentation permission checks.

but this is always added to a 32-bit segment base[1].

Our AGU implementation omits some permission checks to reduce complexity, as some checks are not used by modern operating systems, which tend to use paging-based checks instead. We perform checks for segment limit, segment-not-present, and data load/store to execute-only (non-read/writeable) segments. We omitted checks based on current and segment privilege levels (similar paging checks are typically used instead), and omitted support for 16-bit segment types (used by 16-bit protected-mode operating systems that targeted the Intel 286). We expect that including the remaining checks would have negligible impact on cycle time, as none of the omitted checks depends on the slow effective address computation, so the delay of the extra logic can be hidden behind the effective address computation delay.

The current AGU on its own achieves 258 MHz. However, in the current design, the cache system has retimed a significant amount of logic into the AGU's cycle, so the actual clock frequency would be somewhat lower than this unless the AGU or cache design were improved or the AGU pipeline length

[1] One reason this behaviour is required is to support so-called "unreal mode", a bug in the 386 that allowed loading 32-bit segments in protected mode, switching back to 16-bit real mode, and still being allowed to use the 32-bit segments. This behaviour was frequently used by real-mode (mainly DOS) software requiring more than 1 MB memory, and has been preserved in later x86 processors.

increased. This is a viable (though undesirable) option, as an extra cycle of AGU latency costs around 3.3% in IPC (Section 10.1.6).

The critical path of the AGU includes computing the effective address, and then comparing it with segment limits to determine whether some form of segmentation fault occurred. The linear address computation is actually much faster: 319 MHz if the fault checks were ignored. This points to one method of increasing circuit frequency without increasing latency, by allowing fault checks to take one more cycle. As faults are rare (and have a penalty of hundreds of cycles to handle anyway), delaying the detection of a fault by one cycle would have no performance impact. However, the memory system must be modified to expect a deferred "segmentation_fault" signal, as the memory system must still be able to abort the operation and its side effects if a segmentation fault occurs.

10.2.5 Complete Execution Unit

Now that we have discussed each execution unit in isolation, they need to be assembled into a complete circuit, following the structure from Figure 10.1 on page 123. Measuring the combined unit is important because the bypass network contains many combinational paths that cross sub-unit boundaries, especially the single-cycle latency path starting from the source operands of the simple ALU, through the ALU, bypass network, and back to the source operands of the ALU in time for the following (data-dependent) operation. Because of these cross-unit combinational paths, we expect the maximum frequency of the entire unit to be lower than any of the units in isolation.

The complete execution unit uses 4667 ALMs, which is approximately the sum of the ALMs used by the individual units. The maximum frequency is 239 MHz.

Unsurprisingly, the critical path runs through the simple ALU due to the need to compute the result and write back to the bypass network and register file in one clock cycle. Also unsurprisingly, the critical path runs through the slowest component of that ALU, the shifter. Near-critical paths consist of the path through the simple ALU followed by the either forwarding multiplexer or the register file RAM's input data register. As discussed in Section 10.2.1, the bypass network is carefully designed so that the forwarding path coming directly from the execution unit needs to go through just one logic level. Oddly, we observed that the delay to reach the register file memory's MLAB input register (zero logic levels) usually exceeds the delay of the bypass multiplexer (one-logic-level).

10.2.6 Summary

Table 10.6 summarizes the frequency (or delay) and area usage of the various blocks in our execution units. We show the area and frequency of some of the subcircuits when synthesized in isolation (e.g., divider), as well as the performance of larger circuits containing them higher in the hierarchy (e.g., branch/complex ALU contains the divider, and "Combined" contains the branch/complex ALU). A block that contains another would be slower than the contained block (e.g., the complex ALU is slower than the divider it contains). We chose to list the simple ALU's sub-component delay in nanoseconds instead of MHz because we know the delay of the ALU itself is not a good approximation of one complete pipeline stage, as the register-to-register path through the simple ALU must also pass through the delay of the register forwarding network.

The most difficult circuits to design were the simple operations that had stringent latency constraints. These circuits were the simple ALU, the forwarding network, and the logic used to handle 16- and 8-bit operand sizes (Section 10.2.1). Most of the "complex" x86 behaviours were not difficult to implement because they are less performance-sensitive: We allocated 3 pipeline stages for the complex execution unit (limited by DSP block latency for multiplication), resulting in more than two usable cycles of timing budget (some needs to be spent multiplexing the results onto the writeback port), but most of those operations could be done in one to two cycles, even without high-effort optimizations. These complex operations do cost area though: The complex ALU is more than three times the size of the simple ALU.

The register file RAMs and forwarding multiplexers each use about a quarter of the execution unit area. We speculate that due to multiplexers being inefficient on FPGAs, the forwarding network would be relatively larger than it would be on a hard processor of a similar design, though we have no reliable information on whether this is the case. However, bypass networks in superscalar processors certainly do consume a significant portion of the execution unit area (and delay), and our design is no different.

While the final frequency is lower than we hoped, 239 MHz is close to the clock frequency achieved by a Nios II/f on the same FPGA, despite our execution units being far more complex.

These execution unit circuit designs suggest that the performance cost of using the complex x86 instruction set compared to simpler instruction sets is actually fairly small. Only the simple and commonly-used operations are on the critical path and limit clock frequency, and most instruction sets share the same set of basic simple operations (adder, bitwise logic, and barrel shifter). Compared to RISC-like instruction sets, the most significant circuit-level change caused by the x86 instruction set is the support for multiple operand sizes. While condition codes were also difficult to implement, condition codes are not unique to x86 and are also used by many RISC-like architectures such as ARM [122], Power [127], and SPARC [160]. Most of the cost of x86 lies in design effort and the area of execution units supporting the many non-performance-critical "complex" operations.

Unit	**Frequency (MHz)**	**Area (ALM)**	**DSP blocks**
Register Files and Bypassing	**∼260**	**2413**	
Register File RAM	–	1168	
Forwarding and Multiplexing	–	1245	
Branch/Complex ALU	**277**	**1651**	**0.5**
Divider	346	443	
Pipelined	279	820	0.5
Simple ALU	**>3.4 ns**	**509**	
Add/Sub	2.3 ns	79	
Bitwise	2.3 ns	120	
Shifter	2.9 ns	258	
Address Generation Unit	**258**	**117**	
Combined	**239**	**4667**	**0.5**

Table 10.6: Circuit area and delay results

Chapter 11

Out-of-order Memory Execution Schemes

In out-of-order processors, the processor extracts instruction-level parallelism from a program by attempting to determine the true (read-after-write) data dependencies between instructions, then executing independent instructions in parallel, even if instructions are executed in a different order than program order. Determining data dependencies through registers is done using register renaming, but determining data dependencies for memory operations, or memory disambiguation, is much more difficult. While register operands are known when an instruction is decoded, memory addresses are not available until memory addresses are computed during instruction execution.

Traditionally, memory disambiguation is performed using CAMs (content addressable memory), by searching for stores whose location (address) overlaps with a load, or vice versa. Some recent work has proposed to replace the address-based memory disambiguation with value-based load re-execution, which speculates on whether a load is dependent on a store, then verifies this by re-executing the load when it commits and comparing whether the loaded data value is still correct [162]. Due to the hardware cost, early out-of-order processors executed arithmetic instructions out-of-order, while still executing memory operations in-order. As processors become more aggressive with larger instruction windows, it becomes more important to be able to execute memory operations out-of-order. Figure 11.1 shows a 40% improvement in micro-ops per cycle (μPC) between using CAM-based memory dependence speculation and disambiguation and in-order memory execution on a two-issue processor. (See Section 11.2 for simulation and workload details.) Other work has shown even more impressive gains for more aggressive designs [163].

While CAMs are undesirable even for hard processors, CAMs are particularly expensive on FPGAs (Section 3.4.4), potentially making re-execution based memory disambiguation schemes even more attractive for FPGA soft processors than for hard processors, especially as the number of speculative load and store operations (i.e., load queue and store queue size) increases.

This chapter examines the suitability for FPGA implementation of four different memory speculation and disambiguation schemes: In-order, Out-of-order using CAMs, Store Queue Index Prediction (SQIP) [164], and NoSQ [165]. Both SQIP and NoSQ are re-execution based and do not use CAMs. We

Portions of this chapter have appeared at FPT 2013 [161].

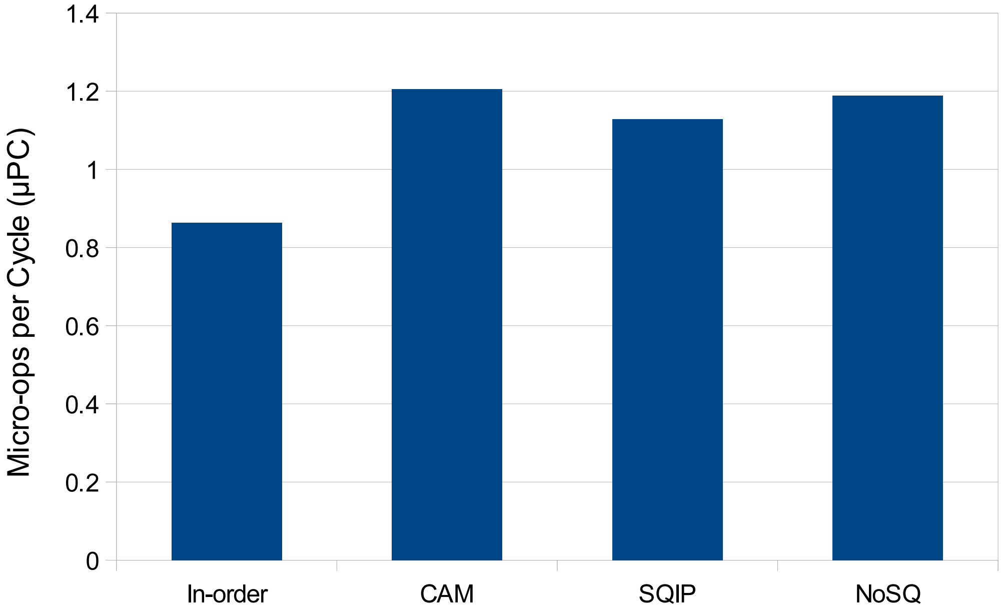


Figure 11.1: Micro-ops per cycle (μPC) of out-of-order processor for various memory scheduling schemes. 64-entry ROB, LQ, and SQ, 256-entry predictors for SQIP and NoSQ.

focus on the FPGA resource usage and maximum frequency of these four schemes when implemented on an FPGA, as earlier work has shown that the two re-execution based schemes have similar IPC (instructions per cycle) to using CAMs [164, 165].

This chapter presents an early exploration of different memory disambiguation schemes, and uses an earlier version of our processor simulation model that models the processor in less detail than the model found in most of the other chapters. The most relevant feature not modelled in this chapter is the detailed behaviour of the memory system (caches, store-to-load forwarding, paging). A detailed design of the memory system will be presented in the next chapter (Chapter 12). Due to the significant differences in the modelled processor and workloads in this chapter, IPC results in this chapter are not directly comparable with the results from other chapters.

This chapter begins with an overview of the function of the memory execution system, followed by a description of each of the memory execution schemes we examine (Section 11.1). We then build an approximation of the hardware for each scheme on a Stratix IV FPGA and evaluate area and frequency (Section 11.3).

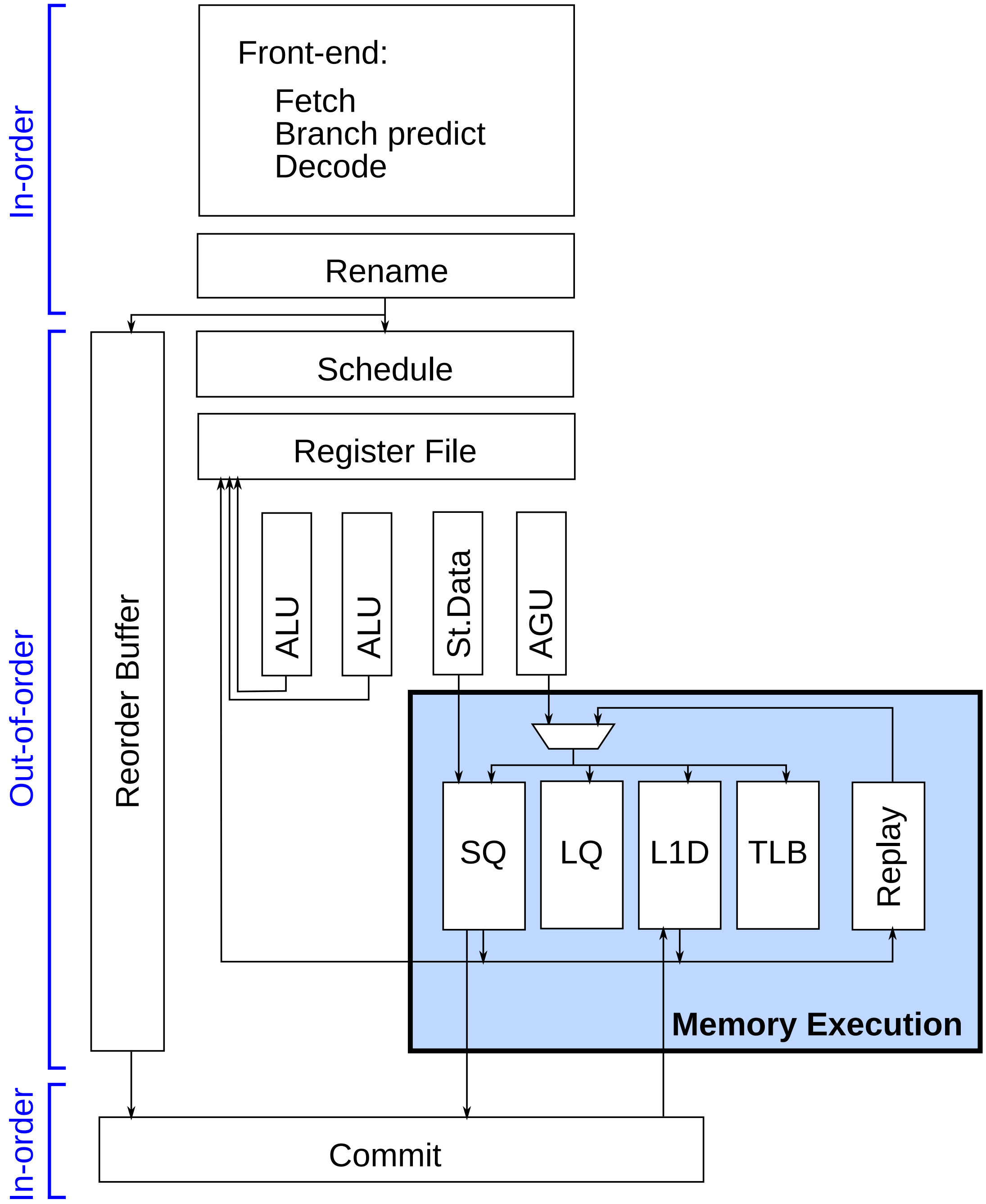


Figure 11.2: Processor microarchitecture block diagram. This chapter focuses on memory execution.

11.1 Background

11.1.1 Memory Execution

In an out-of-order processor, memory execution is concerned with performing loads and store operations after the virtual address has already been computed. In our x86 processor, virtual (linear) addresses are calculated by the address generation unit, and was discussed in Section 10.2.4. Figure 11.2 shows a typical processor pipeline with the memory execution hardware highlighted. For load instructions, the virtual memory address is translated to a physical address, the L1 cache is accessed, and some memory dependence speculation or checking is performed, depending on the memory disambiguation scheme. For store instructions, address translation also occurs, and the store physical address is recorded in the store queue. Store-data is usually executed separately from store-address operations because only the address is needed for memory dependence checks. A store is complete when both its address and data operations have completed.

The memory execution system also has its own scheduler (labelled "Replay"), as a memory operation can fail for a number of reasons (L1 cache miss, TLB miss, load delayed because it depends on a store, etc.). A separate scheduler is used because memory scheduling waits for events such as cache and TLB line fills rather than the availability of register operands.

This chapter focuses on measuring the area and delay of the memory execution hardware, for several memory execution schemes.

11.1.2 Out-of-Order Memory Execution

An aggressive memory execution scheme performs several functions not found in the basic in-order execution scheme: Store-to-load forwarding, out-of-order load execution, memory dependence speculation, and memory disambiguation.

To support precise exceptions in out-of-order processors, store instructions are not allowed to modify memory (or cache) state until the store instruction can be committed (i.e., the store and all earlier instructions have executed without exceptions or misspeculation). Thus, stores are generally written into a store queue to hold speculative stored values until they are committed in-order. If a load reads a value written by a recent (uncommitted) store instruction, the correct value to read is located in the store queue rather than memory. The load must either wait until the store queue is written to memory (cache) before reading the value from memory (cache), or the store queue may support **store-to-load forwarding** and provide the load with the data value. Store queues are allocated and deallocated in program order to facilitate searching the contents by age, for example, searching for all matching stores *earlier in program order than* a given load.

Executing load instructions out-of-order is desirable as it reduces load latency. Out-of-order execution requires knowing the data dependencies between instructions, in particular, knowing whether a load instruction depends on an earlier store. The most basic form (no speculation) delays load instructions until the load is known to be independent from all earlier store instructions, allowing loads to be reordered with other loads but not allowing loads to pass any stores. More aggressive reordering

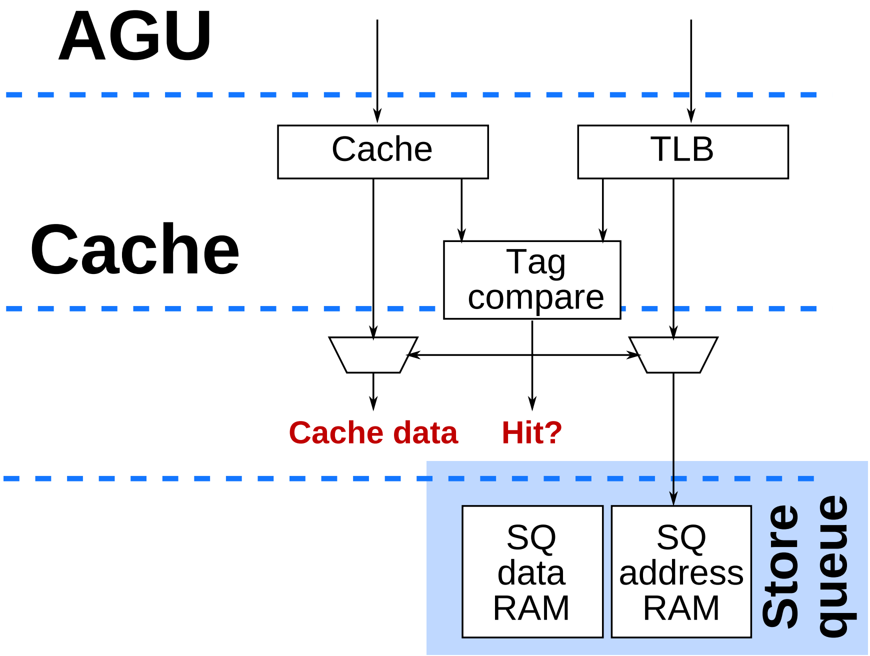


Figure 11.3: In-order memory execution

is enabled by **memory dependence speculation**, which speculates whether a load is dependent on some earlier store and only delays those that are predicted to have a dependence. Earlier work has shown that memory dependencies are highly predictable [163–166].

Using speculation creates a new requirement of needing to verify the prediction using a **memory disambiguation** scheme. There are two main classes of memory disambiguation schemes: Address-based, and value-based. Address-based schemes compare the addresses of loads and stores to determine whether the memory locations overlap. Value-based schemes re-execute the load at instruction commit time and compare whether the non-speculative (correct) load value was the same as the value that was speculatively loaded earlier. CAM-based disambiguation is address-based, while SQIP and NoSQ are value-based.

In Figure 11.1, all three out-of-order schemes perform store-to-load forwarding, out-of-order load execution, memory dependence speculation, and memory disambiguation. The in-order scheme performs none of these.

11.1.3 In-order Memory Execution

The basic in-order memory execution scheme executes loads in-order, and delays loads until all earlier stores have committed into cache. Thus, there is no memory dependence speculation nor store-to-load forwarding. This scheme does not require any CAMs to check whether any earlier stores overlap with a load, nor to search earlier stores for a data value to forward, thus it is low-cost and simple to implement in hardware. However, as seen earlier (Figure 11.1), there is a significant μPC loss.

Figure 11.3 shows our implementation of a hardware pipeline needed to implement an in-order mem-

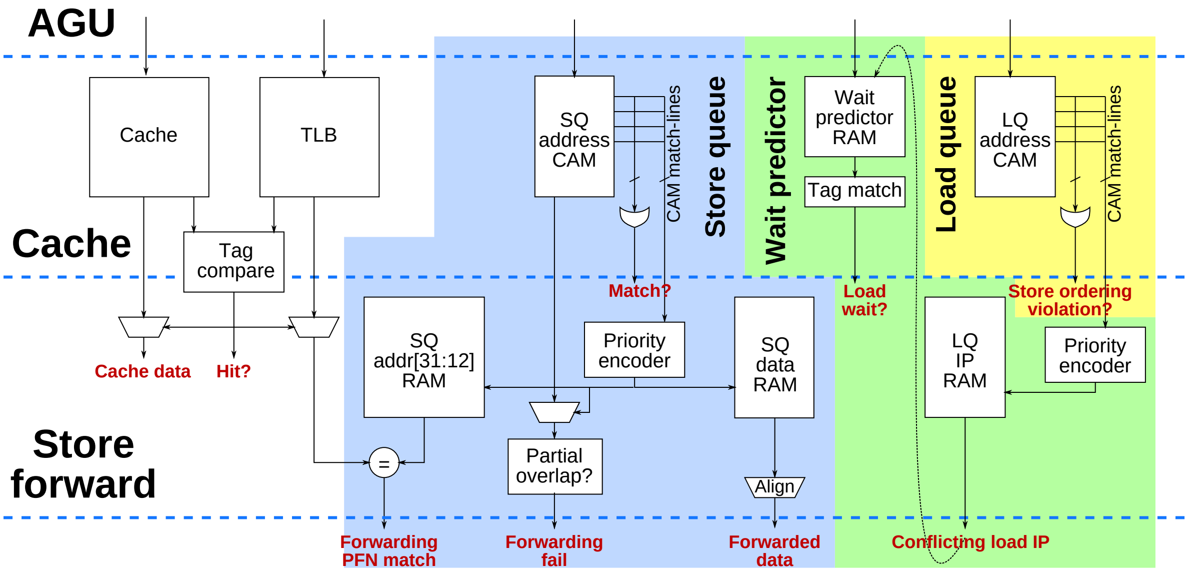


Figure 11.4: CAM out-of-order memory execution

ory execution scheme. Virtual addresses are computed by the address generation unit (AGU). The virtual address is used to look up a TLB for address translation and a virtually-indexed, physically-tagged data cache. TLB and cache tag comparisons occur in the latter half of the cache-access pipeline stage and the beginning of the next, and the store address is recorded in the store queue RAM in the following cycle. The store queue data RAM is populated by store-data operations, which do not go through the address generation and cache access pipeline (not shown, see Figure 11.2).

Cache data is available the cycle after cache access, for a total load latency of three cycles (AGU, cache/TLB, writeback/bypass).

11.1.4 CAM-based Out-of-Order Memory Execution

Figure 11.4 shows our implementation of CAM-based memory execution. In addition to the standard cache, TLB, and tag checking hardware, the store queue becomes a CAM to allow load instructions to search for stores that overlap in memory location. There is also a simple "wait" predictor for memory dependence speculation, and a load queue CAM for memory disambiguation.

When a load executes, the store queue is searched for all earlier stores that overlap with the load. There are four possible outcomes of this search:

- No matches: This load is guaranteed to be independent of all earlier stores, and it is safe to execute the load.
- Match: This load is known to be dependent on an earlier store. The CAM match-lines will also indicate which store queue entries match, allowing store-to-load forwarding.
- Ambiguous: If there are earlier stores whose addresses are not yet known, it is impossible to know which store this load matches with. If there is also a match with another store queue entry, then we

know the load matches with some store, although precisely which store is not yet known. If there are no other matches, we are forced to predict whether there will be a match or not.

- Conflict: There is a match, but it is impossible to perform store-to-load forwarding. In our implementation, this occurs when a load only partially overlaps with a store, requiring a load to gather bytes from multiple sources. This is rare and increases the hardware complexity, so we chose to disallow this type of forwarding. A larger store is allowed to forward to a smaller load that is completely contained within the store. This scheme is equivalent to those used in recent high-performance x86 microarchitectures [167].

If the store queue search is ambiguous, the wait predictor is consulted. The predictor we used is a 64-entry direct-mapped table containing the instruction pointers (IP) of the load instructions. If a load is found in the wait predictor, the load is predicted to be dependent on a store and delayed until all earlier stores commit, otherwise the load is allowed to execute speculatively.

Because loads are permitted to execute before earlier store addresses are known, there is the possibility that a true dependence is missed. These are detected with the load queue CAM. As each store address is computed, it searches the load queue for all *later* loads that have already executed early (i.e., reordered). A match indicates that the load that matched in the load queue actually depends on the current store, yet had executed too early. We handle this by flushing the pipeline after the store has completed (by flagging a store ordering violation). The load queue also has a priority encoder to select the load IP of a matching load in order to train the wait predictor.

We split the store queue's address storage into a RAM for the physical frame number (PFN), bits [31:12], and a CAM for the page offset, bits [11:0]. The store queue is allowed to be conservative, so we only use the page offset bits for address searches because they are invariant across address translation, allowing store queue searches in parallel with address translation at the cost of some aliasing. Using a RAM for the PFN improves FPGA area efficiency.

In our pipeline design, the cache, TLB, store queue, wait predictor, and load queue accesses occur in parallel. Cache hits that do not match any earlier stores and are predicted to not need waiting are available in the following cycle. Selecting the latest matching store using a priority encoder and reading data out of the store queue for store-to-load forwarding takes longer, and is available one cycle later. Thus, the total load latency is 3 cycles for cache hits (AGU, cache/TLB, bypass/writeback), and 4 cycles for loads that require store-to-load forwarding.

11.1.5 Store Queue Index Prediction

Store Queue Index Prediction (SQIP) replaces the CAMs used for memory disambiguation with a predictor that *predicts* whether a load depends on a store, and also from which entry in the store queue the load should forward, allowing the store queue to be implemented in a RAM. A load that is predicted to not need forwarding reads its data from the cache. Correctness is ensured through in-order re-execution. Loads are first filtered using a store vulnerability window (SVW) filter [168] that allows most loads to skip re-execution. A load that needs re-execution waits until the most recent store on which it is poten-

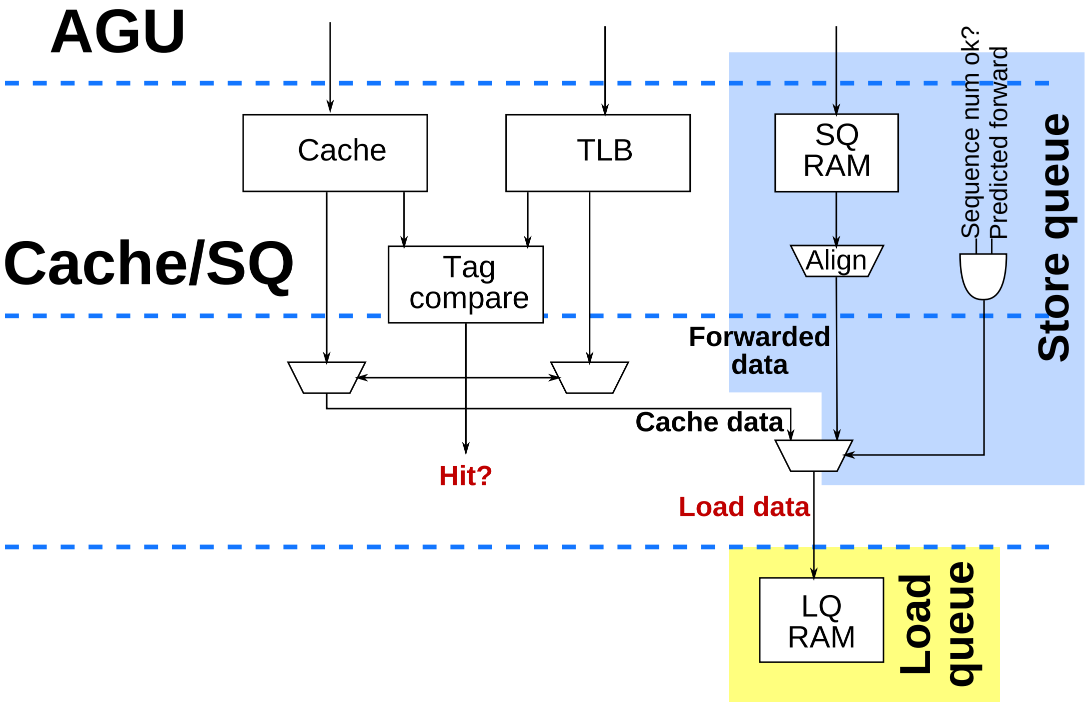


Figure 11.5: SQIP memory execution

tially dependent has committed, reads data from the cache, and compares it to the value that was loaded speculatively.

Figure 11.5 shows our implementation of the load execution portion of SQIP, shown here without the predictors. We implemented the predictor and predictor training hardware as well, but those are outside the critical memory execution hardware and not shown in this figure. Querying the predictor occurs over two pipeline stages at instruction rename. SVW filtering (deciding whether a load must re-execute) and re-execution takes 3 stages each, occurring after instructions complete but before commit. Predictor training takes another 4 cycles.

The complicated store queue and load queue from the CAM-based design are now replaced with simpler RAMs. The load queue now also stores the data values that were loaded so that they can be used for comparison during re-execution. The load queue is now used only for re-execution and not read during execution of loads nor stores.

The SQIP hardware we implemented has a 3-cycle load latency for both cache hits and loads that are forwarded from a store.

11.1.6 NoSQ

NoSQ goes one step further than SQIP by removing the store queue out of the load critical path. This is done by using the memory dependence predictor to *rename* store to load dependencies through the register file (speculative memory bypassing [169]). Instead of taking a data value from a register, storing it to memory, then loading it into another destination register, speculative memory bypassing uses the

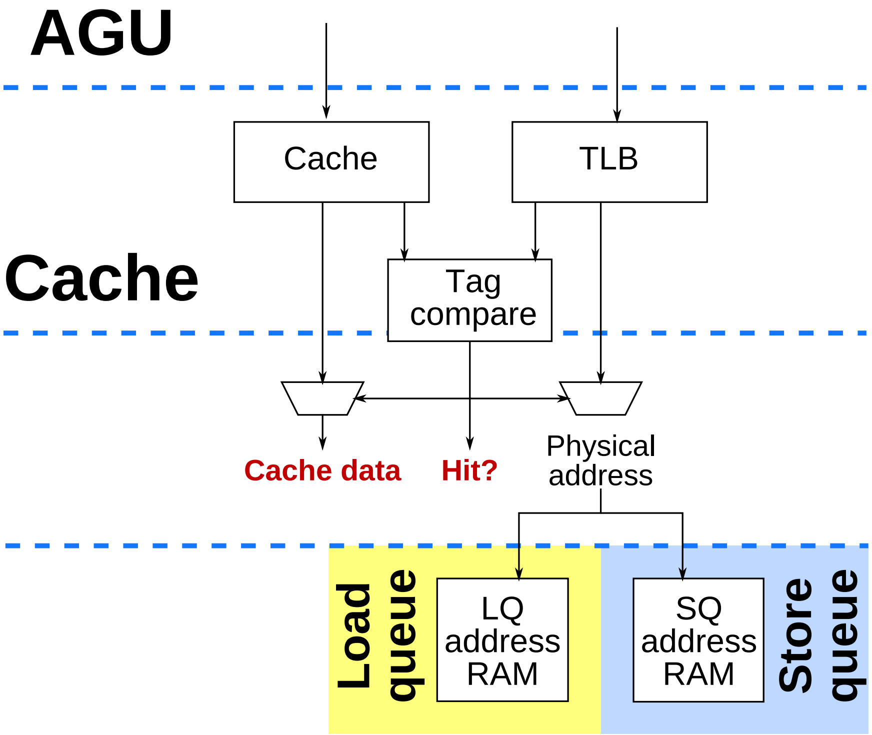


Figure 11.6: NoSQ memory execution

register renamer to point the final destination register's renamer table entry at the source register of the store, thus bypassing the memory system entirely.

Figure 11.6 shows the load hardware for NoSQ. Because all of the loads that require forwarding have been bypassed, the load pipeline is nearly identical to the in-order scheme: Only the cache and TLB are accessed. We chose to include a load queue and store queue address RAM to store addresses for use by re-execution. NoSQ originally proposed removing both of these queues and replacing them with an address generation unit and TLB port to recompute the load and store addresses during re-execution. However, that would require duplicating the AGU and TLB port, and a queue is less costly than the three register file read ports needed for an AGU. From a performance perspective, these two design choices are equivalent. We also used a less aggressive dependence predictor than originally proposed in the NoSQ paper, as a single two-way associative predictor performed sufficiently well, at a significant hardware savings. (See Section 11.3.2.)

NoSQ has some impact on processor hardware beyond memory execution which we did not include in our area and delay measurements. Speculative memory bypassing (and thus, NoSQ) requires the physical register file to be reference counted, but the hardware required to do this is not onerous. NoSQ also allows store operations to be scheduled in-order, reducing the cost of instruction scheduling.

Like SQIP, the total load latency is 3 cycles for loads that hit in the cache, but are even faster for loads that are bypassed. The prediction and SVW pipelines are similar to NoSQ, while the predictor training is simpler in NoSQ due to simpler training rules.

11.2 Methodology

We performed both cycle-level simulation of the various memory execution schemes and implemented the memory execution hardware and associated predictors on an FPGA.

11.2.1 Cycle-Level Simulation

We simulated the memory execution schemes on a cycle-level simulator derived from Bochs [115], a functional full-system x86 emulator. We replaced the CPU simulation with an execute-in-execute cycle-level model of an out-of-order x86 CPU, leaving the rest of the system functionally simulated.

Like other out-of-order x86 designs, the simulator fetches and decodes x86 instructions into micro-ops. The micro-ops are renamed, scheduled, then executed out of order. Micro-ops are then committed in-order, according to x86 atomicity semantics. Interrupts, exceptions, system-mode behaviours, and x86 quirks are handled sufficiently accurately to boot many unmodified operating systems, but our simulator has no support for user-mode simulations. Thus, our workload set is currently limited to booting various operating systems to the desktop. (See Section 11.2.2.)

Table 11.1 shows our simulated processor configuration. We note that the branch/complex ALU unit is in-order scheduled and *only* performs complex operations.[1] Around 23% of micro-ops are "complex", so the effective ALU execution throughput averages only 1.23 per cycle. We report performance in micro-ops per non-halted cycle (μPC) because the number of micro-ops per instruction varies greatly, particularly with string operations. We count only non-halted cycles because of HLT instructions that halt the processor for many cycles until the next interrupt.

For SQIP and NoSQ, we default to predictor sizes roughly $1/16^{th}$ the size as originally proposed, as we found that this gave significant area and frequency improvements with only a small loss in μPC. (See Section 11.3.2.) For SQIP, we use a two-way 256-entry Forwarding Store Predictor (FSP), 32-entry Store Alias Table (SAT), and 256-entry Store PC Table (SPCT) and Store Sequence Bloom Filter (SSBF) [164]. For NoSQ, we use a 256-entry store-load bypassing predictor, an untagged 256-entry SSBF, and a 64-entry Store Register Queue (SRQ) that matches the 64-entry reorder buffer size.

Our focus in this chapter is on the area and delay trade-offs of the various predictors. The main value of the cycle-level simulation is to verify implementation details, as many IPC results have already been published in previous work [163–165]. Although parts of our simulated processor are not realistic, we believe our μPC results are reflective of the relative performance of the memory execution schemes even though the absolute numbers may change. In particular, a more realistic memory hierarchy would only increase the sensitivity of overall μPC to memory system performance, increasing the gap between in-order and the three out-of-order memory execution schemes.

[1] "Complex" operations include branches, string instructions, multiplication, division, 9-, 17-, and 33-bit shifts and rotates, and instructions that use segment, control, or model-specific (MSR) registers.

Parameter	Value
Fetch	16 bytes per cycle
Decode	2 x86 instructions per cycle, 2 micro-ops per cycle
Rename	2 micro-ops per cycle
Execute	1 branch/complex, 1 ALU, 1 AGU, 1 store-data
Execution latency	1 cycle per micro-op, 3 cycles for loads
Commit	4 micro-ops per cycle
Caches and TLB	Perfect
Branch prediction	Overriding predictor. Fetch: 4-way 16k-entry BTB, 16-entry RAS; Decode: 8K-entry gshare, 512-entry 1-way indirect branch, 16-entry RAS
Speculation	64 micro-op reorder buffer, 64-entry store queue, 64-entry load queue
SVW	9-bit SSNs, 256-entry 4-bank untagged SSBF, For SQIP: 256-entry FSP, 32-entry SAT; For NoSQ: 256-entry 1-way predictor, 64-entry SRQ.

Table 11.1: Simulated processor configuration

11.2.2 Simulation Workloads

In this chapter, our workloads consist of booting operating systems. Table 11.2 lists the workloads. The aggregate μPC results are aggregated by summing total runtime and total instruction counts, so are weighted more heavily to slower OSes. A breakdown of μPC by workload for the default configuration described in the previous section is shown in Figure 11.7.

11.2.3 Hardware

To make area and operating frequency measurements, we implemented the memory execution portion of the processor on a Stratix IV FPGA, using Quartus II 13.0 SP1. Because the cache and TLB are closely coupled with the load and store queues, we also implemented a mock-up of the cache and TLB that includes the caches, read ports, and tag comparison logic, but without the more complicated but less timing critical cache miss handling and page table walking logic. (See highlighted region of Figure 11.2.) In all cases, we use a 2-way 8 KB cache (in M9K block RAM) with 64-byte cache lines and a 2-way 64-entry TLB (in MLAB LUT RAM). Except for a small change to the register renamer and instruction scheduler (see Section 11.1.6), the omitted portions are independent of the memory execution scheme used, and thus, do not affect our comparison between memory execution schemes.

We report FPGA resource utilization using the “Logic utilization” metric reported by Quartus, which takes into account how often logic functions of various sizes can be packed into a dual-output fracturable LUT (Stratix IV ALM). We also include the area of used memory blocks in equivalent ALUTs (1/20 of a LAB), based on Stratix III relative areas reported in [50]. Frequency and area results are the average over 20 random seeds.

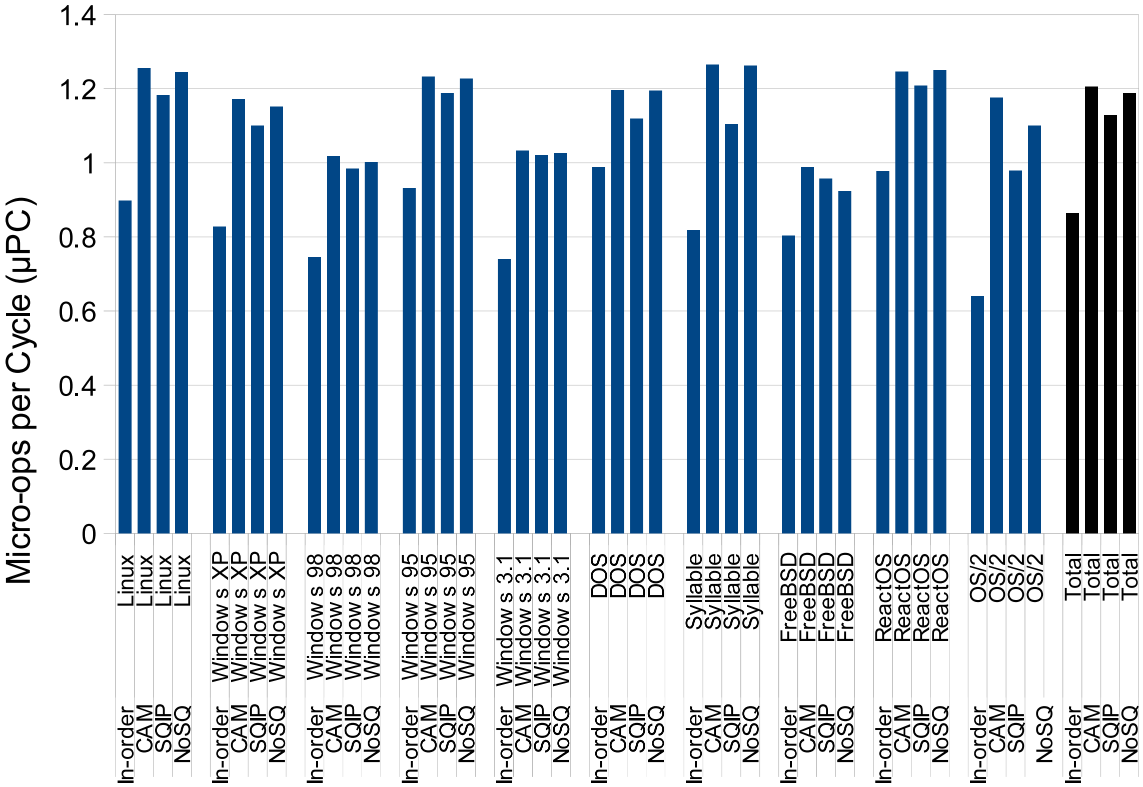


Figure 11.7: μPC by workload at 64-entry ROB, LQ, and SQ, and 256-entry SQIP and NoSQ predictors.

Operating System	Inst. (10^9)	μops (10^9)	**Boot to**
Mandriva Linux 2010.2	15.5	18.5	Desktop
Windows XP	5.2	6.6	Desktop + web browser
Windows 98	0.6	1.0	Desktop + web browser
Windows 95	0.3	0.6	Desktop
Windows 3.1	0.3	1.0	Desktop
MS-DOS 6.22	0.5	0.7	Command prompt
Syllable Desktop 0.6.7	4.0	5.3	Desktop
FreeBSD 9.0	2.1	2.5	Installer
ReactOS 0.3.14	1.6	2.2	Desktop
OS/2 2.1	0.033	0.045	Fails to boot
Total	30.16	38.40	

Table 11.2: Simulation workloads

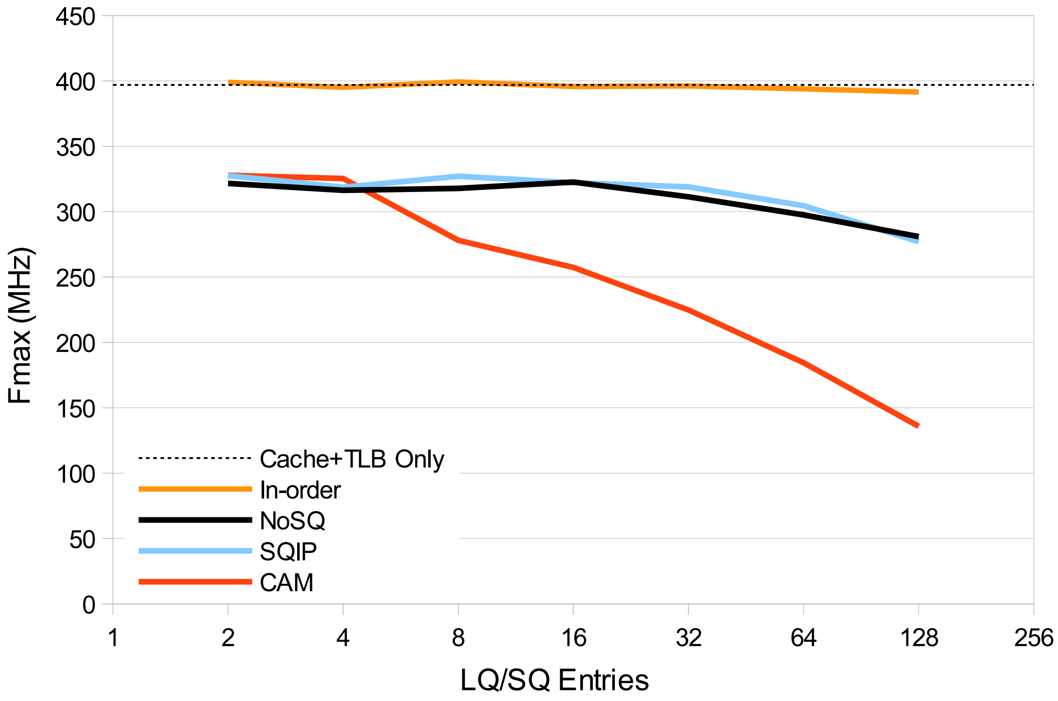


Figure 11.8: Maximum frequency at varying LQ and SQ sizes

11.3 Results

11.3.1 Performance over Varying Queue Sizes

Figures 11.8 and 11.9 show the maximum frequency and area of the in-order, CAM-based out-of-order, SQIP, and NoSQ memory execution schemes. We evaluated load queue and store queue sizes from 2 through 128 entries. In the plot of area (Figure 11.9), the area of the cache and TLB alone is also marked with a dashed horizontal line.

In-order The simple in-order scheme has a low hardware cost. Its operating frequency of around 400 MHz is limited by the delay through the cache for the range of queue sizes we evaluated. Because the majority of the area is consumed by the cache and TLB, the area usage does not noticeably grow until 64 entries, when additional MLABs need to be used for the store queue, since MLAB RAMs can have a depth of at most 64.

CAM Not surprisingly, the CAM-based load/store queues are sensitive to the number of queue (and CAM) entries. The delay increases slightly quicker than logarithmic in the number of entries, while area use grows linearly with the number of CAM entries. With very few entries, CAMs perform similarly to in-order, since there is little extra hardware beyond the CAMs.

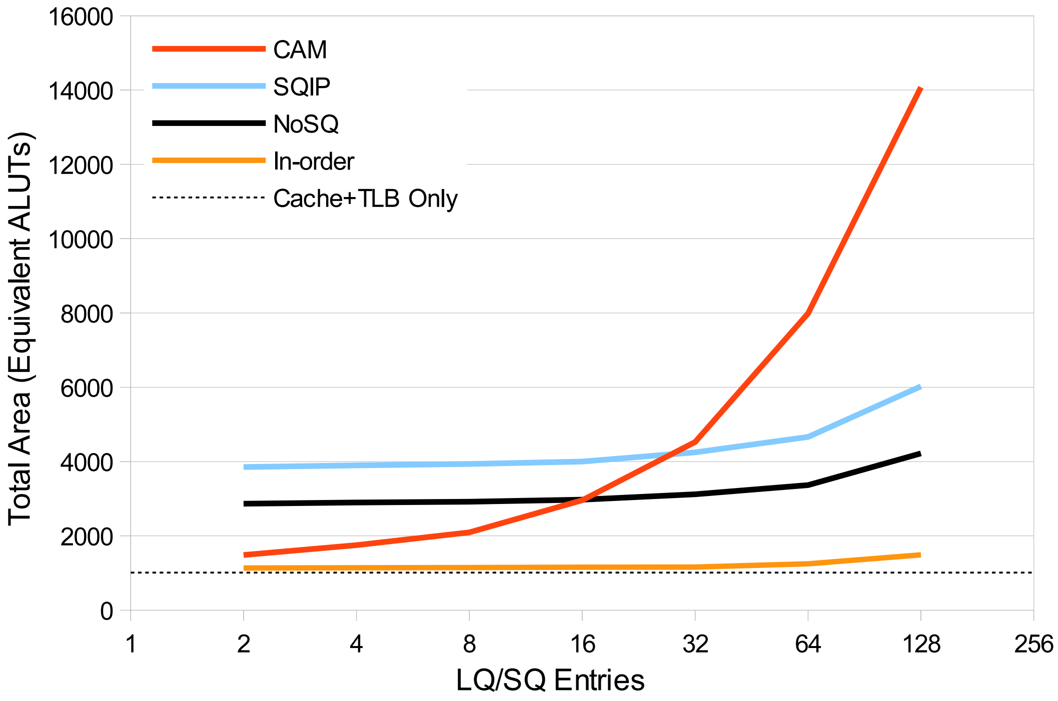


Figure 11.9: Total area at varying LQ and SQ sizes

SQIP and NoSQ SQIP and NoSQ behave similarly with varying queue size. The critical path is mainly limited by various sequence number and tag comparisons, which grow slowly with increasing queue size. Similarly, the area consumption also grows slowly with increasing queue size. However, the predictor tables and associated logic to do prediction, re-execution, and predictor training adds a significant amount of area overhead. NoSQ and SQIP have greater area usage than the CAM-based scheme below 16 and 32 entries, respectively.

Table 11.3 gives a breakdown of FPGA resource usage for the four schemes at 64 load queue and store queue entries.

Scheme	Logic Utilization (ALUTs)	M9K	M9K Area (Equiv. ALUTs)	Total Area (Equiv. ALUTs)	f_{max} (MHz)	μPC
Cache and TLB only	570	8	459	1029	395	—
In-order	771	8	459	1230	397	0.86
CAM	7256	8	459	7715	186	1.21
SQIP	3649	16	918	4568	324	1.13
NoSQ	2667	12	689	3356	328	1.19

Table 11.3: Area and μPC for varying memory execution schemes for our default configuration: 256 predictor entries, 64 LQ and SQ entries.

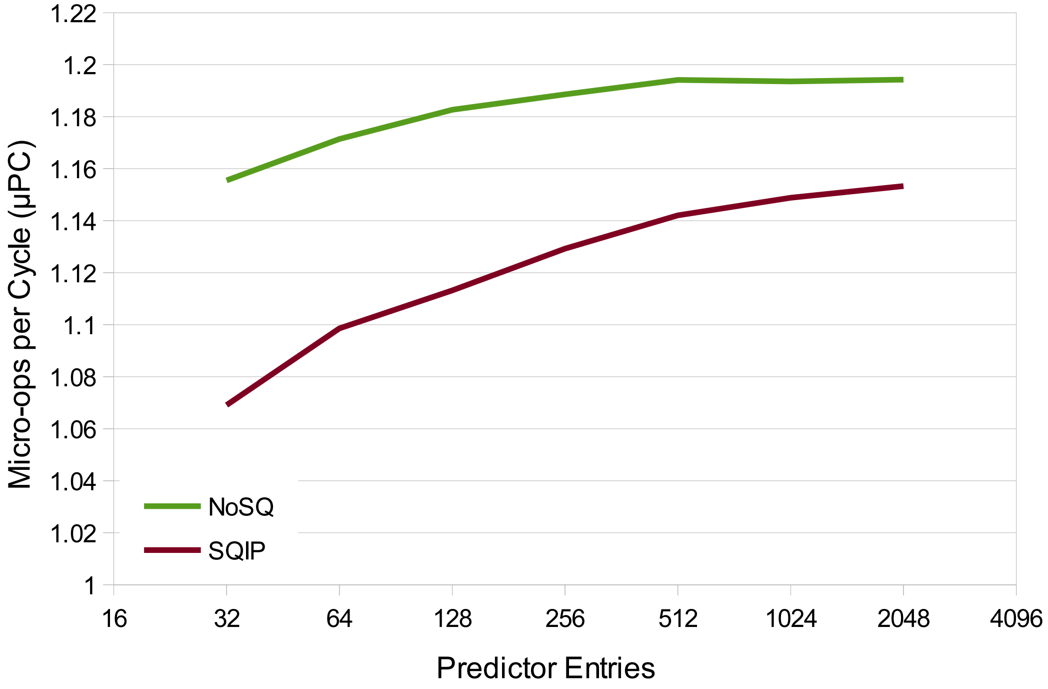


Figure 11.10: μPC impact of varying SQIP and NoSQ predictor size. Note the non-zero vertical axis origin.

11.3.2 SQIP and NoSQ Predictor Size

The previous section showed that for a fixed predictor size, both SQIP and NoSQ are only slightly sensitive to the size of the load and store queues. In this section, we evaluate the sensitivity to varying predictor sizes at a fixed 64 load and store queue entries.

SQIP has three predictor tables. Our implementation uses a 256-entry Forwarding Store Predictor (FSP), 256-entry Store PC Table (SPCT) and untagged Store Sequence Bloom Filter (SSBF), and 32-entry Store Alias Table (SAT). NoSQ also uses three tables. We use a 256-entry store-load bypassing predictor, a 256-entry untagged SSBF, and a 64-entry Store Register Queue (SRQ). To evaluate scaling predictor table sizes, we scale all of the tables together, except the SRQ, which must match the reorder buffer size.

Figure 11.10 shows the impact of varying the predictor table sizes on μPC. For our workload of booting OSes, the μPC changes little with decreasing predictor capacity. This generally agrees with previous work where most workloads also show little sensitivity to reduced predictor capacity [164, 165].

Figures 11.11 and 11.12 show the hardware cost for varying predictor sizes. Both maximum frequency and area show significant degradation above 512 entries. Thus, we chose 256 predictor entries for our default configuration, which sacrifices some μPC loss for a higher frequency and lower area.

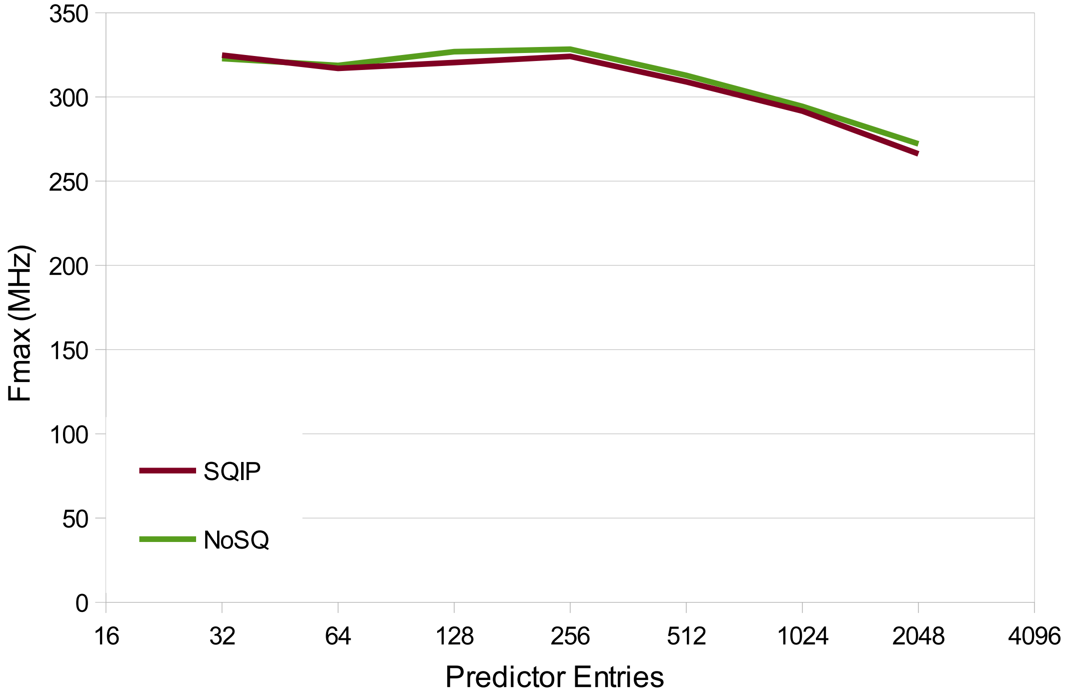


Figure 11.11: Maximum frequency impact of varying SQIP and NoSQ predictor size

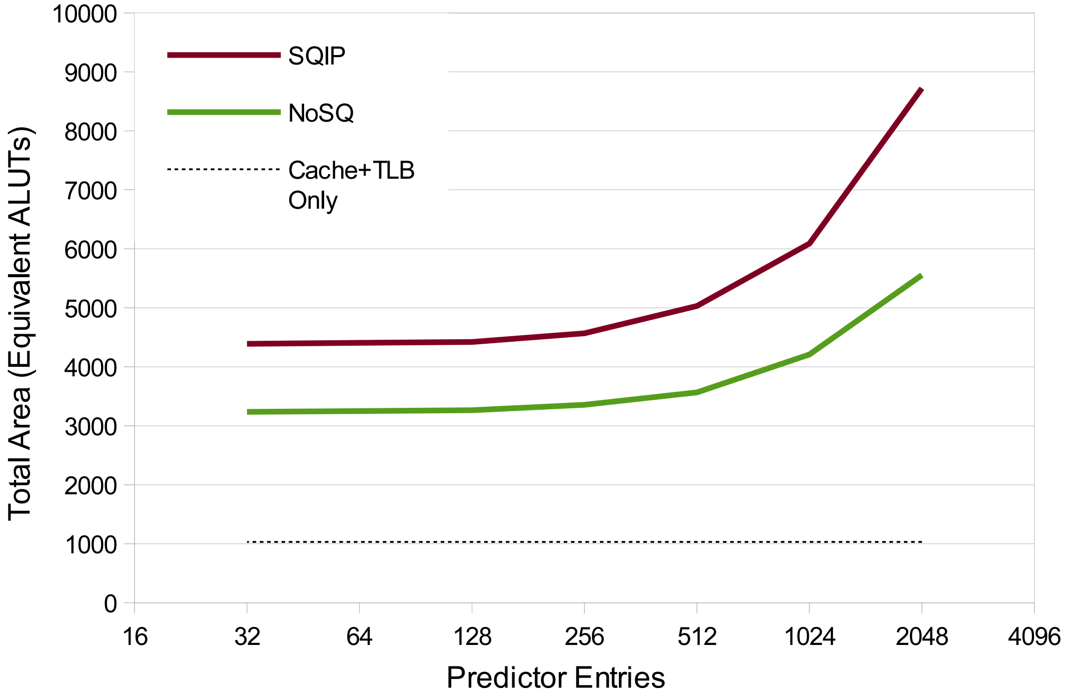


Figure 11.12: Total area impact of varying SQIP and NoSQ predictor size

11.4 Conclusions

As single-threaded soft processors increase in performance and complexity, there will be a need for out-of-order execution of memory operations and memory dependence speculation. However, the traditional method of using load queue and store queue CAMs is particularly inefficient on FPGAs. In this chapter, we evaluated four memory execution schemes: In-order, CAM-based out-of-order, and two schemes that do not use CAMs: SQIP and NoSQ.

We implemented each of the four schemes on a Stratix IV FPGA and measured the maximum frequency and area usage. We find that the area of the CAM-based scheme grows quickly with the number of load queue and store queue entries, while SQIP and NoSQ have a greater fixed area overhead independent of queue size, with NoSQ being more efficient. In our implementations, SQIP and NoSQ are more area efficient than using CAMs beyond 32 and 16 entries, respectively. We observe similar trends in maximum frequency, where the CAM based scheme loses performance quicker than SQIP or NoSQ with increasing queue size, with a break-even point of around 4 queue entries. For high-performance soft processors, in-order memory execution is unattractive because even if the rest of the processor could be designed to run at 400 MHz, the higher clock frequency does not make up for the loss in IPC (e.g., at 64-entry LQ/SQ, in-order has 21% higher frequency but 40% lower IPC).

Chapter 12

Memory and Cache System

The previous chapter compared several memory disambiguation schemes, but without a detailed memory system design. In this chapter, we choose one of those schemes and design a complete memory execution and cache system around it.

The previous chapter compared the traditional CAM-based (address-based) memory disambiguation scheme with two re-execution based disambiguation schemes (without CAM), and concluded that the latter schemes were faster and more area efficient, particularly for designs supporting more in-flight loads and stores. However, we chose to design the memory system based on the CAM-based scheme to reduce risk. The memory system with multiple caches and its coherence and ordering requirements is already very complicated. Choosing a traditional scheme at least guarantees that there is a method to implement the x86 memory model sufficiently correctly. To our knowledge, no x86 processor has yet used a re-execution based memory disambiguation scheme. In addition, we selected load queue and store queue sizes smaller than envisioned in the previous chapter (16 vs. 64), which reduces the relative inefficiency of the CAM-based scheme. At 16 entries, the CAM-based scheme was expected to be similar in area to the re-execution based schemes, although around 30% slower.

This chapter begins with the CAM-based memory disambiguation structure, and designs a complete memory system. We study trade-offs in the cache design, out-of-order memory execution, and memory disambiguation [166]. Many aspects of the design are applicable to non-x86 systems as well. Also, because our scope includes creating the ability to boot modern general-purpose operating systems, the memory system must support virtual memory, including paging and translation lookaside buffers (TLBs). This influences many of the trade-offs, and increases the complexity of the system. We also discuss FPGA circuit-level considerations and the detailed circuit design of our two-way associative TLB and cache lookup, showing that careful design resulted in a circuit faster than the simpler TLB and direct-mapped cache access in the Nios II/f.

In this study, we use detailed cycle-level processor simulation to evaluate the impact in IPC of the various processor microarchitecture features. Compared to the simple in-order memory systems with one level of cache that are currently used in soft processors, we evaluate the impacts of a second level of

This chapter has appeared in ACM TRETS [170].

caching, speculative out-of-order memory execution, and non-blocking cache miss handling. We then design highly-tuned circuits that implement all of the above features while correctly handling unaligned accesses and other requirements of a uniprocessor x86 system.

This chapter demonstrates that large increases in IPC (up to 2.1× on SPECint2000) can be achieved through improvements in the memory and caching system alone — even with a conservative 30-cycle memory latency — on top of the IPC provided by out-of-order instruction execution. It also demonstrates that the hardware that delivers this IPC and the features to support general-purpose OSes can be built at a reasonable resource usage (14 000 equivalent ALMs) at high frequency (200 MHz), which is faster than most in-order soft processors and within 17% of the 240 MHz Nios II/f.

This chapter is organized as follows: we discuss related work in Section 12.1, the requirements of the memory system in Section 12.2, the methodology of design, simulation and benchmarking in Section 12.3, a summary of the microarchitecture of the processor in Section 12.4 and the structure of the memory system in Section 12.5. Section 12.6 explores the trade-offs in the memory system microarchitecture, while Sections 12.7 through 12.9 describe circuit level design trade-offs, optimization, and synthesized results.

12.1 Related Work

The microarchitectural dimensions explored in this chapter are not new, as they traverse a long and distinguished history of processor architecture [12]. Our focus is in part on how to bring that knowledge into the soft processor space, and to deal with the realities of processors that boot real operating systems. Most existing soft processors are relatively small and employ simple, single-issue pipelines, including commercial soft processors (Nios II [17], MicroBlaze [16]) and non-vendor specific synthesizable processors that target both FPGA and ASIC technologies (Leon 3 [20], Leon 4 [20], OpenRISC OR1200 [34], BERI [171]). The vendor-specific commercial processors are tuned for high frequencies (e.g., 240 MHz for Nios II/f vs. 150 MHz for Leon 3 and 130 MHz for OR1200, all on the same Stratix IV FPGA).

The memory systems of all of these in-order processors stall the processor pipeline whenever a cache miss occurs. A key issue explored in this chapter is the effect of allowing the processor to proceed when one or more cache misses occur. The RISC-V Rocket [172] is notable for using a non-blocking cache with an in-order pipeline. The RISC-V project also has an out-of-order synthesizable core (BOOM) with an out-of-order non-blocking cache system, but does not appear to be designed for FPGAs [22]. BERI [171] is particularly interesting in that it implements the MIPS instruction set well enough to boot the FreeBSD OS. It uses a two-level cache hierarchy, but to our knowledge there is no published analysis of the trade-offs involved in their cache hierarchy, such as the one we present in this chapter.

A non-blocking cache for soft processors was proposed in [173]. They use an *in-cache* Miss Status Holding Register (MSHR) scheme that tracks outstanding memory requests in the cache tag RAM to avoid associative searches. We avoid this scheme because the port limitations and high latency of FPGA block RAMs (particularly for writes) make an in-cache implementation difficult and slow. Instead, we use a small number (4) of MSHRs, which results in a small and fast associative search, while giving up almost no IPC (Section 12.6.1).

Another performance-enhancing aspect of our memory system that is not found in current soft processor memory systems is out-of-order execution, including memory disambiguation (determining data dependencies between stores and loads) and memory dependence speculation [166]. Conventional designs use associatively-searched load queues and store queues to perform store-to-load forwarding and memory disambiguation. Previous work has explored both conventional and newer non-associative schemes for use by FPGA soft processors [161]. Despite being slower and less area-efficient, we chose to use the conventional disambiguation scheme to reduce risk, as to our knowledge, the more efficient schemes have not yet been fully proven in an x86 design.

There have been previous projects that synthesized modern x86 processors into FPGAs [21, 42, 43]. However, these processors were not designed for FPGA implementation, so they tend to be much larger and slower (ranging from 0.5 to 50 MHz operating frequency) than processors designed with an FPGA target in mind. We intend to achieve a much higher operating frequency (>200 MHz) in the processor described here.

12.2 Memory System Requirements

A memory system performs memory accesses for both instruction fetches and data loads and stores. A bootable system with virtual memory requires both paging support and coherence between various memory and I/O transactions. This, together with a goal of high performance, makes the memory system design complex. In this section we describe the requirements of the memory system imposed by the x86 instruction set architecture, and those requirements arising from the goal of high performance.

The instruction set architecture specifies many properties of the memory system. Due to the x86 instruction set's long legacy, it tends to keep complexity in hardware to improve software compatibility. The key features of a uniprocessor x86 memory system include the following:

1. Paging is supported, with hardware page table walks.

2. 1, 2, and 4-byte accesses have no alignment restrictions. Accesses spanning page boundaries are particularly challenging as they require two TLB and cache tag lookups.

3. Cacheability is controllable per-page. In particular, the UC (uncacheable) type disallows speculative loads (typically used for memory-mapped I/O).

4. Data and instruction caches (but not TLBs) are coherent, including with reads and writes from I/O devices (e.g., DMA). Self-modifying code is supported.

A multiprocessor system also needs to obey the memory consistency model, which we leave for future work.

In addition to the functionality requirements, our goal of building a high-performance soft processor adds complexity to the memory system. Cache sizes and latency play an important role in determining memory system performance. Cache miss handling and out-of-order memory execution also greatly impact overall performance by reducing pipeline stalls and increasing opportunities for finding overlapping operations.

A simple blocking cache stalls memory operations when a cache miss occurs, and in most in-order soft processors, this also stalls the entire processor. A non-blocking cache continues to service independent requests rather than stalling. This is particularly important for out-of-order processors that can find independent operations to execute. The simplest non-blocking cache allows “hit under miss”, where cache hits are serviced while waiting for a single cache miss to return; these caches *will* stall on a second miss. This notion can be extended to support multiple outstanding misses using multiple Miss Status Holding Registers (MSHR) to track the in-flight cache misses [174]; in this chapter we will explore the impact of a non-blocking cache and the number of MSHRs to provide.

Memory dependence speculation also enhances performance. Simple memory systems execute load and store operations in program order. To execute them out of order, the processor must know whether a load reads from a location written to by an earlier in-flight store in order to know whether the load is dependent on the store. However, this is difficult because load and store addresses are not known until after address generation (unlike register dependencies that are known after instruction decoding). Non-speculative out-of-order memory systems allow limited reordering, as long as loads only execute after all earlier store addresses are known. Memory dependence speculation allows further reordering, but must detect misspeculations and roll back if necessary [166]. We will explore the impact of memory dependence speculation in our memory system.

12.3 Methodology and Benchmarks

This chapter uses the same methodology as described Chapter 4. The microarchitecture design and instruction-per-cycle performance measurements are done using a detailed cycle-level simulation. Once satisfied with the correctness and performance of the microarchitectural design, we implemented the memory hierarchy in SystemVerilog, then the measured cycle time and FPGA resource usage (area).

12.3.1 Simulation Benchmarks

A key advantage of using the x86 instruction set is the wide availability of benchmark programs, in both source and unmodified binary forms. We have simulated a wide variety of workloads that we expect would have varying sensitivity to memory system performance, and are listed in Table 12.1. We discuss several aspects of these here.

The SPECint2000 suite stresses both the processor core and memory system, and was simulated with the reference input, skipping two billion instructions then simulating one billion. We only used the subset that did not have an excessive amount of floating-point content because our processor only emulates floating point in firmware. The *mcf* benchmark of the SPECint2000 suite is particularly challenging for the memory system, as it performs pointer chasing on a graph and has high cache miss rates [175].

The MiBench benchmark suite has a fairly small memory footprint. It comes with “small” and “large”

Workload	**x86 Inst.** (10^6)	**Description**
SPECint2000	20 000	gzip, gcc, mcf, crafty, parser, gap, vortex, bzip2 **Excluded:** eon, vpr, twolf, perlbmk
MiBench [118]	2 568	**Automotive:** bitcount, qsort, susan (edges, corners, smoothing) **Consumer:** jpeg, mad, tiff2bw, tiffdither, diffmedian, typeset **Office:** ghostscript, ispell, stringsearch **Network:** patricia, dijkstra **Security:** blowfish, pgp, rijndael, sha **Telecom:** adpcm, crc32, gsm **Excluded:** basicmath, lame, tiff2rgba, fft
Windows 98	600	DOS + 9x kernel
Windows XP	5 200	32-bit NT kernel
Windows 7	16 000	32-bit NT kernel
Mandriva Linux 2010.2	15 500	Desktop Linux OS (kernel 2.6.33)
FreeBSD 10.1	4 000	UNIX-like OS (no GUI)
ReactOS 0.3.14	1 600	Windows NT clone
Syllable Desktop 0.6.7	4 000	Desktop operating system
Dhrystone	260	200 000 iterations
CoreMark	247	600 iterations
Doom 1.9s	1 938	-timedemo demo3
Total	71 913	

Table 12.1: Benchmarks and x86 instruction counts

input data sets. We used "small" inputs for all of the benchmarks except *stringsearch*, and ran the benchmarks to completion. We omitted benchmarks that contained an excessive amount of floating-point.

The Dhrystone and CoreMark benchmarks have minimal demands on the memory system, with L1 data cache miss rates of around 0.02%.

Doom is a good example of a legacy x86 software program — a 3D first person shooter game released in 1993. It uses 32-bit protected-mode but not paging, and has a significant amount of self-modifying code. Surprisingly, this workload does not use any floating-point instructions.

Finally, and most importantly, we also boot several x86 operating systems which have a mix of user and system code. OS workloads measure the time from power-on until the boot process is complete (desktop loaded or login screen).

12.4 Processor Architecture and Microarchitecture

Our processor's x86 instruction set architecture (ISA) is based on Bochs' Pentium model [115]. We decided to eliminate the x87 floating point unit (FPU) to reduce hardware design effort; since modern operating systems expect x87 support, it was replaced with a new OS-invisible trap mechanism and em-

Property	Value
Fetch	8 bytes per cycle
Branch prediction	16K entry BTB, 16-entry return address stack
Decode	2 x86 instructions/cycle, 2 µops/cycle
Rename	Integer, segment, and flags renamed, 2 µops/cycle
OoO	64-entry ROB, 8-entry scheduler per execution unit
Execution Units	1 Branch/complex, 1 ALU, 1 AGU, 1 Store data.
Store queue	16 entries
Load queue	16 entries
L1I	2-way, 8 KB, blocking
ITLB	2-way, 128-entry, 4 KB pages, blocking
L1D	3-cycle latency, 2-way, 8 KB, writeback, 4 MSHR
DTLB	2-way, 128-entry, 4 KB pages, 2 outstanding page walks
L2	4-way tree pseudo-LRU, 256 KB, 9-cycle load latency, writeback, 8 outstanding misses
Memory	30 cycles (44 cycles observed by software, including L1 miss, L2 miss, and queuing delays)

Table 12.2: Baseline system configuration

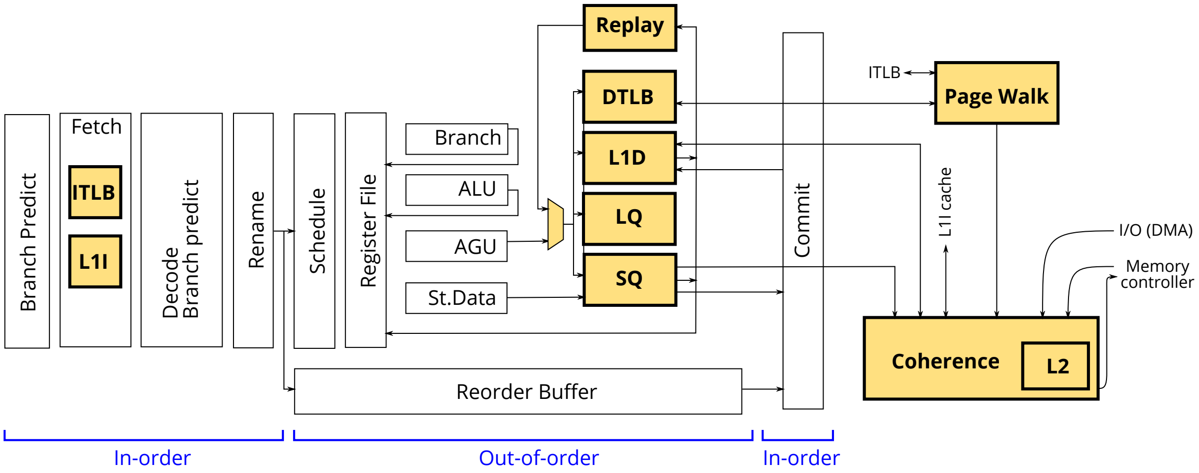


Figure 12.1: CPU microarchitecture diagram. This chapter focuses on shaded portions.

ulation code in firmware.

The CPU microarchitecture is a typical single-threaded out-of-order design with in-order front-end (fetch, decode, rename), out-of-order execution, and in-order commit. The baseline configuration used in this chapter is detailed in Table 12.2 and Figure 12.1. The microarchitecture was chosen to be feasible to implement on an FPGA. The explorations in this chapter will consist of removing certain aspects or features of this baseline architecture to see the net effect on performance.

We model main memory as a 30-cycle latency and throughput of one 32-byte access per cycle, which translates to 150 ns and 6.25 GB/s with a CPU clock frequency of 200 MHz. While modern memory systems on high-performance systems can have access times on the order of 50 ns, we use a 150 ns latency to account for additional delays through the memory controller and access contention that is not captured with this simple memory model.

12.5 Memory System Microarchitecture

The memory system performs memory accesses and paging using separate instruction and data L1 caches and TLBs, hardware page table walking, and a unified L2 cache. The store and load queues enable out-of-order memory execution. The shaded portions in Figure 12.1 show the memory system hardware components. The remainder of this section describes these units in more detail.

12.5.1 Instruction Fetch

Because instruction fetches happen in program order, the instruction memory hierarchy is fairly straightforward. An 8 KB two-way virtually-indexed physically-tagged (VIPT) instruction cache is looked up in parallel with a large 128-entry instruction TLB.

12.5.2 Data loads

The L1 load pipeline (illustrated as the stack of shaded boxes with “Replay” at the top in Figure 12.1) is very complex due to the need to support ISA features such as cacheability control and unaligned operations, and performance-enhancing features such as multiple outstanding loads [174] and out-of-order speculative execution [166].

The load pipeline is shown in more detail in the right half of Figure 12.2. After the address generation unit (AGU) computes the linear (virtual) address for an operation, address translation, cache lookup, and store queue lookup are done in parallel. Cache hits are complete at this point, but the many possible failure conditions are handled by a generic replay scheduler (stage 1), avoiding stalls until the replay scheduler is full.

To allow for multiple outstanding requests, we use four associatively-searched MSHRs (miss status holding register) to track up to four outstanding cache lines. Each MSHR entry holds information for one cache line and one outstanding load (the primary miss) to that cache line. A miss to a cache line already in-flight (secondary miss) is replayed and not sent to the coherence unit. This design allows multiple outstanding misses to the same cache line, without the increased MSHR entry complexity that would be needed if the MSHR were responsible for tracking multiple outstanding loads, but at the expense of a higher latency for secondary misses due to replay. In Section 12.6 we explore the effect of the quantity of MSHRs.

Memory dependence speculation [166] allows loads to be executed without waiting for all previous store addresses to be computed. Since most loads are not dependent on an earlier store, a load is speculated to be independent of earlier stores unless a dependence predictor indicates otherwise. We use a simple dependence predictor. All loads query a 32-entry direct-mapped tagged table indexed by instruction pointer. A hit causes the load to be replayed and wait for earlier stores to execute. Entries are created in the table to remember loads that caused a dependence misspeculation in the past, and are never explicitly removed.

The store queue and load queue (SQ and LQ in Figure 12.1) are used to find dependencies and dependence misspeculations, respectively. Loads search the store queue for earlier dependent stores. If a load is found to be fully contained within an earlier store, the store data is forwarded to the load. Stores search the load queue to find later dependent loads that may have been incorrectly executed too early.

An unaligned memory accesses has no penalty unless it crosses cache line boundaries ("split")[1]. Cacheable split loads (including split-page loads) are executed in two consecutive cycles, doing one TLB lookup and tag check each cycle and reading the data array for both lines on the second cycle.

From a design effort perspective, correctly implementing the combination of cacheability control and split loads has been extremely painful, because all of these features interact and produce a huge number of corner cases. For example, split-page loads require two TLB lookups, each of which may miss or cause a page fault. In addition, if the load is uncacheable (UC), and if one TLB lookup misses or faults, the other half-load must not be sent to main memory; discarding a speculative load result is not sufficiently correct, since UC loads from memory-mapped I/O can have side effects.

12.5.3 L1 Data Cache

We chose a VIPT cache, which allows TLB and cache lookups to proceed in parallel. This is possible if the cache index is unmodified by address translation[2] — in this case the lower 12 bits for 4 KB pages. Thus, paging increases the latency penalty for building large L1 caches (making two-level caches favourable), as increasing L1 cache size either increases associativity (and logic delay) or increases latency by serializing the TLB and cache lookups (PIPT).

12.5.4 Coherence Unit and L2 Cache

The coherence unit merges requests from 8 sources, most of which are shown in the left hand side of Figure 12.2: L1D loads, L1D stores, L1D evictions, L1I reads, page table walker, memory fill, I/O device-initiated requests (DMA), and I/O-space requests. The coherence unit services requests, maintains coherence of all caches, and provides a global ordering to satisfy memory consistency requirements.

The L2 cache uses a non-inclusive/non-exclusive ("accidental inclusion") policy, so all L1 cache tags are checked at every request. A second copy of all L1 cache tag arrays is used for this purpose. A non-inclusive policy reduces complexity compared to an inclusive policy by avoiding reverse invalidations when the L2 wishes to evict a line that is also stored in at least one L1 cache.

12.5.5 TLB and Page Table Walker

The instruction and data TLBs are both 128-entry, two-way set-associative. Although processors have often used higher-associativity or fully-associative TLBs, we believe low-associativity TLBs are more suitable for FPGA processors. First, storage capacity is reasonably cheap, so it is easier to reduce miss rates by increasing capacity rather than associativity. Increasing cache capacity also reduces sensitivity to associativity. Our large 128-entry L1 data TLB loses less than 1% IPC for two-way associativity for SPECint2000.

[1] While x86 is often maligned as unnecessarily complex, this complexity is often beneficial. For example, unaligned accesses are enough of a win that the ARM ISA began supporting them with ARMv6 [122].

[2] Alternatively, requiring the OS to compensate for the cache organization is so onerous that ARMv7 no longer requires this for data caches [122].

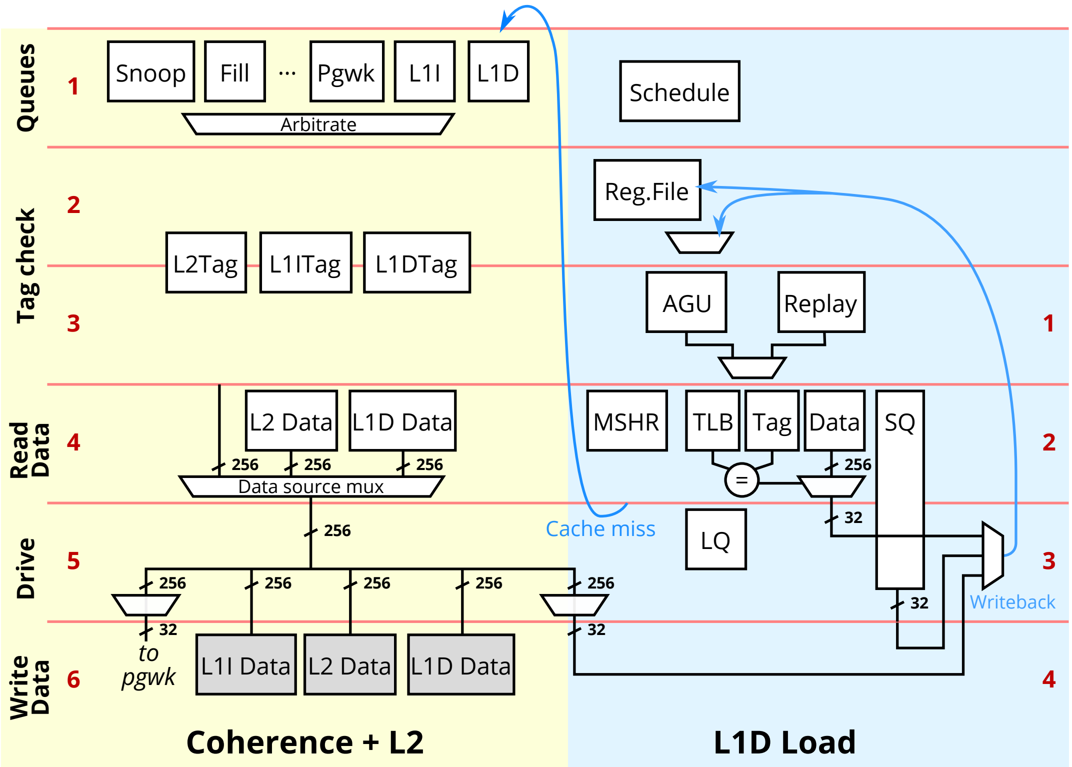


Figure 12.2: Load and coherence pipeline diagram. 3-cycle L1D hit latency, 9-cycle L2 hit latency.

Second, using a low-associativity TLB and cache allows cache tag comparisons to be overlapped with TLB tag comparisons, which has a large circuit speed advantage, which we discuss in Section 12.8. Serializing the tag comparisons to accommodate a higher-associativity (or fully-associative) TLB in the same latency and cycle time would require the TLB lookup to complete in roughly one LUT delay (∼0.8 ns on a Stratix IV), which is impossible to achieve, especially for higher-latency BRAM-based CAMs (e.g., [75, 176]). Small and fast (L1) caches cannot tolerate the slower CAM circuits, while larger, slower caches are less sensitive to associativity.

A TLB miss results in a hardware page table walk, as required in the x86 architecture. The page walk accesses the two-level x86 page tables, making memory access requests to the coherence unit, which can be cached by the L2 cache. The page table walker is non-blocking, allowing up to three concurrent page walks and one outstanding memory operation per page walk currently in progress. A table holds the current state of each active page table walk, and services one ready page walk each cycle. This is similar to a proposal for page table walking on GPUs [177].

12.6 Microarchitecture Simulation Results

This section evaluates the impact of some of the microarchitecture design choices described in the previous section. We vary the microarchitecture against the baseline architecture described in Table 12.2, and compare its instructions-per-cycle (IPC) performance using the cycle-level simulator and benchmarks

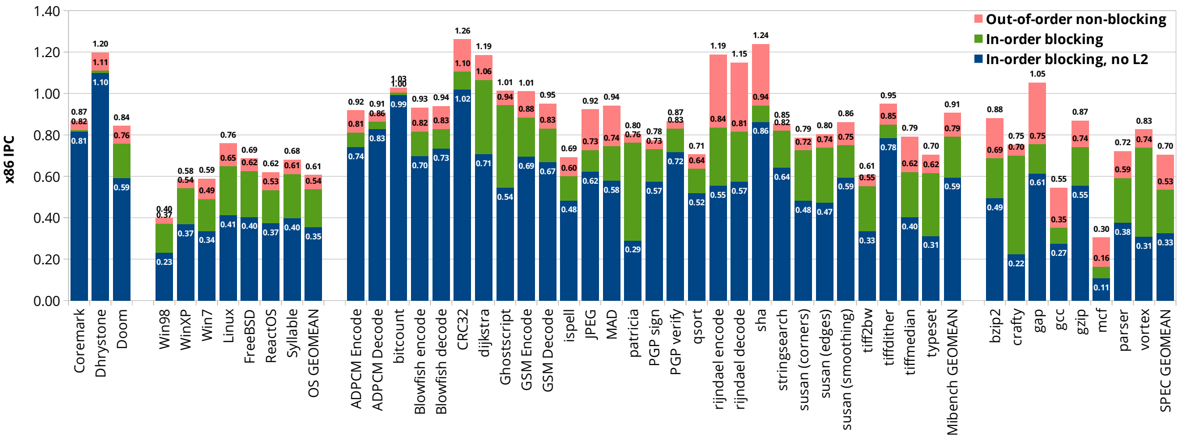


Figure 12.3: Benchmarks: x86 IPC for out-of-order non-blocking baseline, in-order blocking cache, and in-order blocking without L2 cache

described in Section 12.3. Note that we do not measure the microarchitectural impact on frequency and area yet, but leave those results to Section 12.9. Recall that the baseline processor and memory system configuration has out-of-order execution, memory dependence speculation, multiple outstanding data cache misses, and L1 and L2 caches.

First, we will evaluate our choice of 16 for the number of store queue and load queue entries. Figure 12.4 shows the IPC change as the store queue and load queue sizes are decreased from 63 entries, where both queue sizes are decreased at the same time. The IPC loss for 16 entries ranges from none for benchmarks with a small memory footprint (Dhrystone, CoreMark) and up to 3.6% IPC for the memory-intensive SPECint2000. Given the high hardware costs (in both timing and area) of CAMs (used by both the store queue and load queue), we opted to keep them small at the expense of a few percent IPC.

The two major performance-related aspects of our memory system are non-blocking speculative out-of-order execution of memory operations and the use of a two-level cache hierarchy. Figure 12.3 shows a per-benchmark breakdown of IPC to evaluate the impact of successively disabling these two performance features. The "blocking" configuration disallows out-of-order memory execution and speculation and stalls all memory operations during cache misses. The bottom bar shows the result of also disabling the L2 cache. As noted in Section 12.1, most soft processors stall *all* instructions during a cache miss. Our "blocking" memory configuration blocks only memory operations and continues to allow *non-memory* operations to execute out-of-order (It is an out-of-order processor with an in-order memory system), which is more aggressive than in-order soft processors, even with comparable memory systems.

There is a lot of data in Figure 12.3; we will make some general observations followed by more detailed ones. The chart shows that most workloads benefit from having a non-blocking cache, particularly those with large working sets (e.g., SPECint2000's mcf [175]). Workloads with very small working sets (CoreMark and Dhrystone) show no sensitivity to removing the L2 cache, but still benefit from out-of-order execution of cache hits. Overall, the benefit of a two-level non-blocking memory system are large (2.1× IPC on SPECint2000). These large gains are seen even with our modest two-issue processor and the smaller memory latencies seen on lower-clock speed FPGA processors.

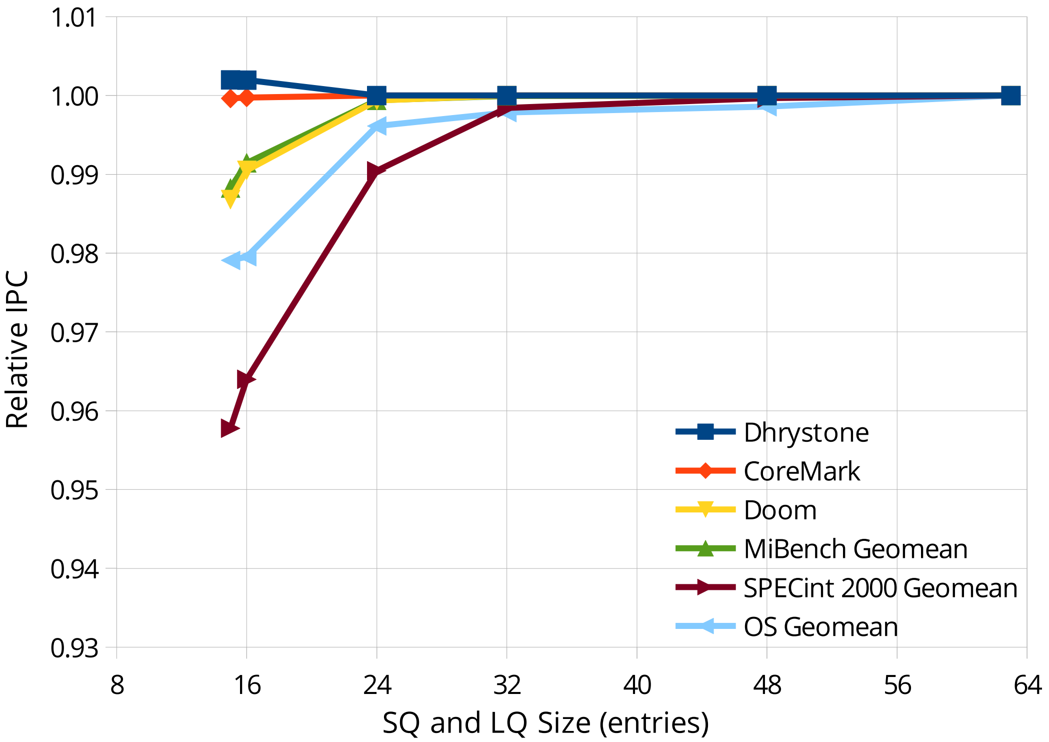


Figure 12.4: IPC sensitivity to load queue and store queue size (both queues of equal size)

The rest of this section looks at the benefits of non-blocking caches and a two-level cache hierarchy in more detail.

12.6.1 Non-blocking, Out-of-Order Cache

Figure 12.5 breaks down the contribution to performance of the non-blocking and out-of-order speculative properties of the cache, for a total of four design points. The chart compares the IPC of blocking and non-blocking caches with and without speculative out-of-order memory execution to a blocking in-order cache.

Of these four design points, three have been used by processors in practice: blocking in-order, non-blocking in-order, and non-blocking out-of-order. The blocking out-of-order design point is impractical because store-to-load forwarding must still be non-blocking to avoid deadlock, because a load can depend on an earlier store which depends on an even earlier load (a dependency carried through memory). This configuration would have nearly all of the complexity of a non-blocking *memory system* but none of the benefits of a non-blocking *cache and TLB*.

We modelled the four design points using our non-blocking out-of-order design by disabling features. In practice, each design point would use a different memory execution pipeline structure, but we expect the IPC difference between the two approaches to be minor. Starting with a non-blocking out-of-order design, we model in-order memory systems by admitting memory operations into the memory system in program order, while continuing to allow memory operations to execute and complete in any order.

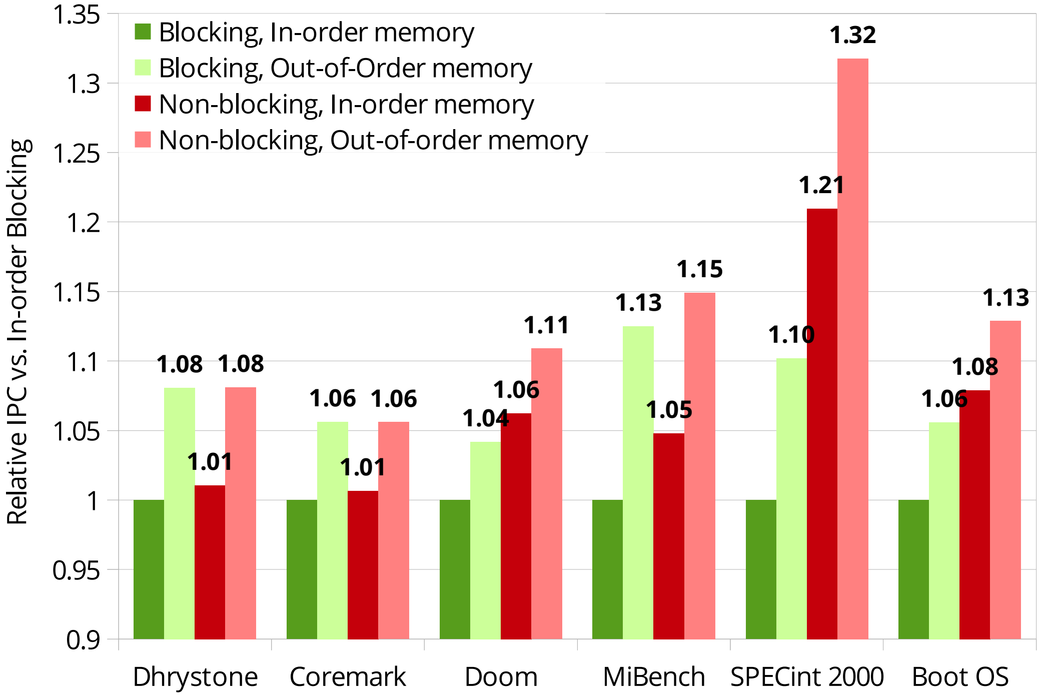


Figure 12.5: IPC impact of non-blocking and out-of-order memory execution (256 KB L2, 4 MSHR entries). A blocking out-of-order memory system is an impractical design. Pink and dark green bars correspond to bars of the same colour in Figure 12.3.

In-order blocking behaviour is modelled by blocking the execution of all memory operations whenever a memory operation does not complete for any reason (e.g., TLB miss, cache miss, store-to-load forward not ready), while the out-of-order blocking configuration blocks on TLB and cache misses only but allows store-to-load forwarding to be non-blocking to avoid deadlock.

Out-of-order speculative execution benefits all workloads, while non-blocking caches disproportionately benefit workloads with more cache misses. We expect in-order processors (like existing soft processors) to benefit less from a non-blocking cache because in-order processors must stall as soon as an unavailable memory value is used, limiting the amount of useful independent operations that can be found.

Figure 12.6 examines the benefit of multiple outstanding loads in more detail, by varying the number of outstanding loads permitted before stalling (which is the number of MSHRs, described in Section 12.5). Allowing just one outstanding load (hit-under-miss) is enough to get most of the performance benefit for most workloads. Workloads that are sensitive to memory performance (SPECint2000, booting OSes) continue to see improvement with more outstanding loads, with diminishing returns. Only one workload (SPECint2000's mcf) benefits from more than our baseline of four outstanding loads.

The large gains for memory dependence speculation and multiple outstanding loads make both techniques important for a high-performance soft processor. The low area cost of MSHRs (Table 12.3) makes it practical to build the four MSHRs needed to capture most of the available performance.

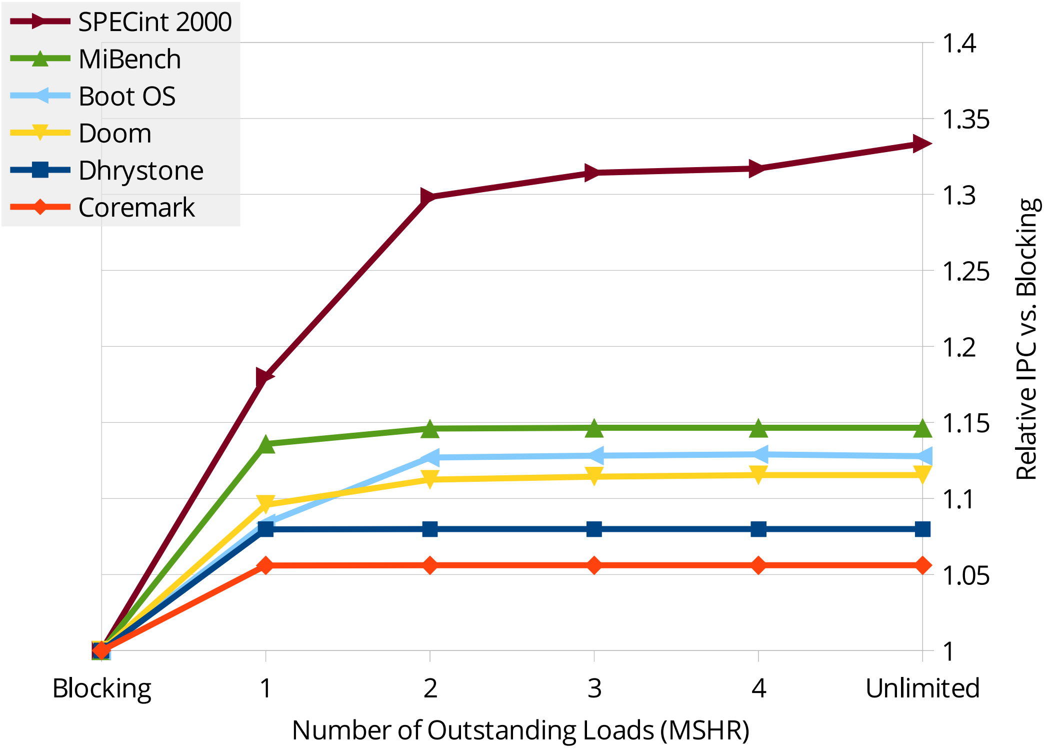


Figure 12.6: Relative IPC vs. number of outstanding loads

12.6.2 Two-Level Cache Hierarchy

We saw in Figure 12.3 that simply disabling the L2 cache can have a large performance impact. This section examines cache sizes in more detail, and whether using larger L1 cache sizes has similar performance to two-level hierarchies.

Figure 12.7 shows the impact of varying the L1 cache size with and without a 256 KB L2 cache. These results are optimistic for L1 cache sizes larger than 8 KB, as we increased L1 cache size without accounting for any extra latency that would be needed to access the larger caches. The instruction and data L1 cache sizes are increased together, from 4 KB 1-way to 256 KB 32-way. In this chart, left-to-right slope shows sensitivity of IPC to L1 cache size, while the size of the blue portion of the bars indicate sensitivity to the presence of a 256 KB L2 cache.

Unsurprisingly, workloads that fit in a small L1 cache (CoreMark and Dhrystone) do not benefit from a larger L1 cache nor L2 cache. For the other workloads, adding a second-level cache makes performance less sensitive to L1 cache size.

Since the modelled L1 cache latency (3 cycles) is lower than L2 (9 cycles), one might expect a large L1 cache to perform better than a two-level hierarchy with a small 8 KB L1 (In Figure 12.7, a yellow bar greater than 1.0). However, a two-level hierarchy with 8 KB L1 and 256 KB L2 caches performs similarly to using large and unrealistic single-level 256 KB L1 caches (512 KB total). The reason is because page tables are cached in the L2 cache and page walks become far more expensive (by about 65 cycles for two page table memory accesses) when the L2 cache is removed. The slower TLB miss handling time has a

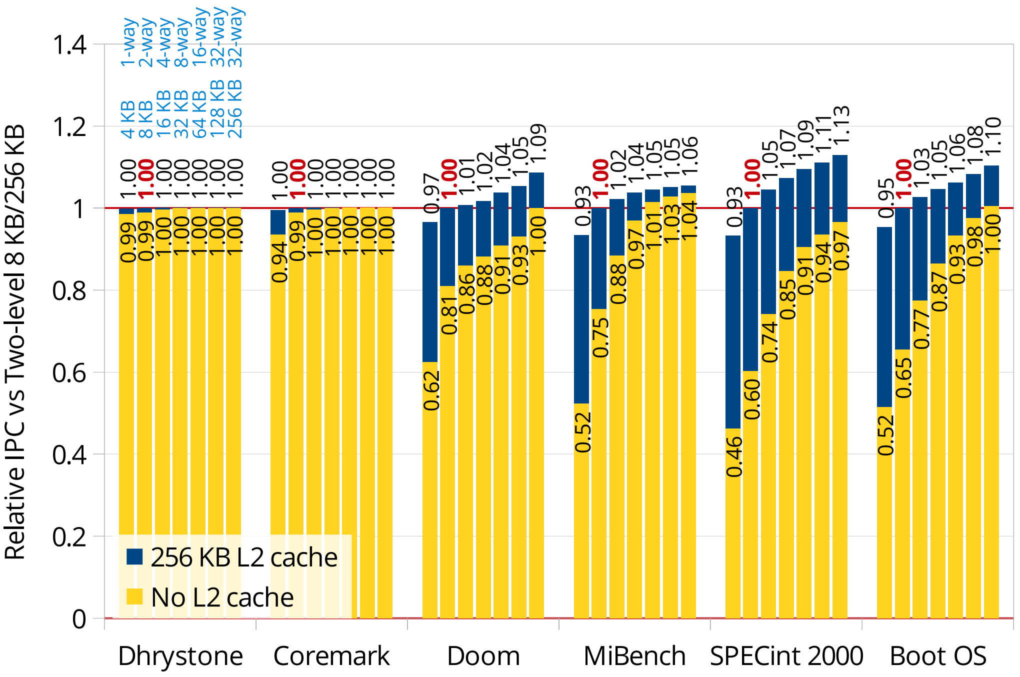


Figure 12.7: L2 cache: Relative IPC for varying L1 cache sizes with and without a 256 KB L2 cache

significant impact on performance even with low TLB miss rates (e.g., SPECint2000 has 1.8 page walks per thousand instructions). We already use large 128-entry TLBs, so improving paging performance to compensate for removing the L2 cache would be difficult, likely requiring the complexity of a two-level TLB hierarchy and page walk caches.

For the workload that does not use paging (Doom), the effect of slower page table walks does not apply. However, Doom uses self-modifying code, which requires cache lines to be transferred between the instruction and data caches. We do not allow direct L1I to L1D cache line transfers nor most cases of cache line sharing, so self-modifying code causes L1 cache misses that become more expensive when there is no unified L2 cache.

The presence of paging makes a two-level cache hierarchy preferable, due to the increased latency (for translation or high associativity) of a larger L1 cache, and the benefit of a unified L2 cache being able to cache page table walks.

12.7 FPGA Memory System Design

The IPC benefits from an out-of-order memory system described in the previous section can be translated into performance benefits only if the required circuits can be built to run at a high frequency with low latency. This requires both designing the microarchitecture with the FPGA substrate in mind and careful circuit design.

Due to the high effort required to design memory system hardware, we built hardware only for the highest-performing non-blocking out-of-order memory system. As the various configurations typically require complete hardware redesigns, simply disabling functionality as done in Section 12.6 gives reasonable IPC estimates but does not result in reasonable hardware designs.

This section briefly discusses FPGA-specific design considerations for each hardware unit. We then present detailed circuit-level design and optimization techniques used by the TLB and cache access stage (DTLB, L1D, and SQ blocks in Figure 12.1) in Section 12.8.

Our overriding goal of correctly supporting x86 features, flaws, corner cases, and legacy code complicates the design trade-off space. In addition to the traditional metrics of performance (IPC and frequency) and cost (FPGA resources), this *correctness* requirement constrains the design space and makes design effort an important design goal that can be traded for performance and/or cost.

12.7.1 L1 Data Cache

While microarchitecture-level cache design concerned sizes and latency, FPGA circuit-level design also needs to consider the cost of RAM block read and write ports, to avoid needing more ports than provided by a block RAM. The design of the L1 cache data array needs to service four types of requests:

1. Loads (32-bit read, unaligned, read both ways)
2. Stores (32-bit write, unaligned)
3. Evictions to L2 (256-bit read, aligned)
4. Fills from L2 (256-bit write, aligned)

As cache data arrays are large, they need to be designed to fit the characteristics of the FPGA block memories. Each FPGA block memory has two read-write ports (2rw), but the available port width is doubled when used in single port (1rw) or simple dual port mode (1r1w). Our high-frequency (and low-latency) requirements preclude time-multiplexing (double-pumping) the memory block, while the large size makes it desirable to use the memory block in its most area efficient 1r1w mode.

A straightforward implementation of two-way associativity duplicates the data array and selects the correct way for each access. However, it is wasteful to duplicate the wide fill and eviction ports as only one line is filled or evicted at a time. Our design, shown in Figure 12.8a, shares a read port between loads and evictions, and shares a write port between stores and fills. This arrangement uses the FPGA dual-ported block RAMs in simple dual-port mode and supports two-way associativity without replication, and separates the high-traffic load and store interfaces on separate physical ports to reduce the IPC impact of contention (Loads and stores occur more frequently than evictions and fills). To avoid duplicating the wide fill and eviction ports, the lower and upper 16 bytes of alternating cache lines are swapped. This arrangement satisfies all four types of requests. Fills and evictions can access any one (possibly swapped) full cache line by accessing both halves with the same address. Loads can access the same offset from both ways of the cache by reading both halves but with one address incremented or decremented. Independent control of the block memory read addresses even allows reads that cross into the next cache line to be accessed the same way. This scheme can be extended to caches of higher associativity.

We chose a cache geometry of 8 KB, two-way associativity, with 32 byte cache lines. This is the result of many competing design goals, including IPC, area, delay, ISA constraints, and design effort. The following are some of the less obvious trade-offs we needed to consider:

- Cache hit latency: A virtually-indexed physically-tagged (VIPT) organization reduces cache hit latency over physically-indexed physically-tagged (PIPT) by doing address translation and cache lookup in parallel instead of sequentially.
- ISA constraint: x86 caches must behave like PIPT, so VIPT caches must have at most 4 KB/way (page size) or extra alias-detection logic used during cache misses.
- Design effort: We chose the lower-effort option: 4 KB/way, but this makes large caches more difficult due to high associativity.
- Design effort: Coherence unit should transfer whole cache lines in one cycle, to avoid the complexity of multiple-cycle transfers.
- Area: Transferring full cache lines makes cache port widths and coherence unit multiplexer sizes equal to cache line size. Smaller cache lines reduce area, but increase the the cache tag RAM array depth (number of words). The depth of the cache tag RAM is equal to the number of cache sets.
- Delay: A tag array no deeper than 64 words is ideal. MLAB LUT RAMs are faster than the larger M9K block RAMs, and MLABs with depth greater than 64 words require slow soft-logic multiplexers to stitch them together.

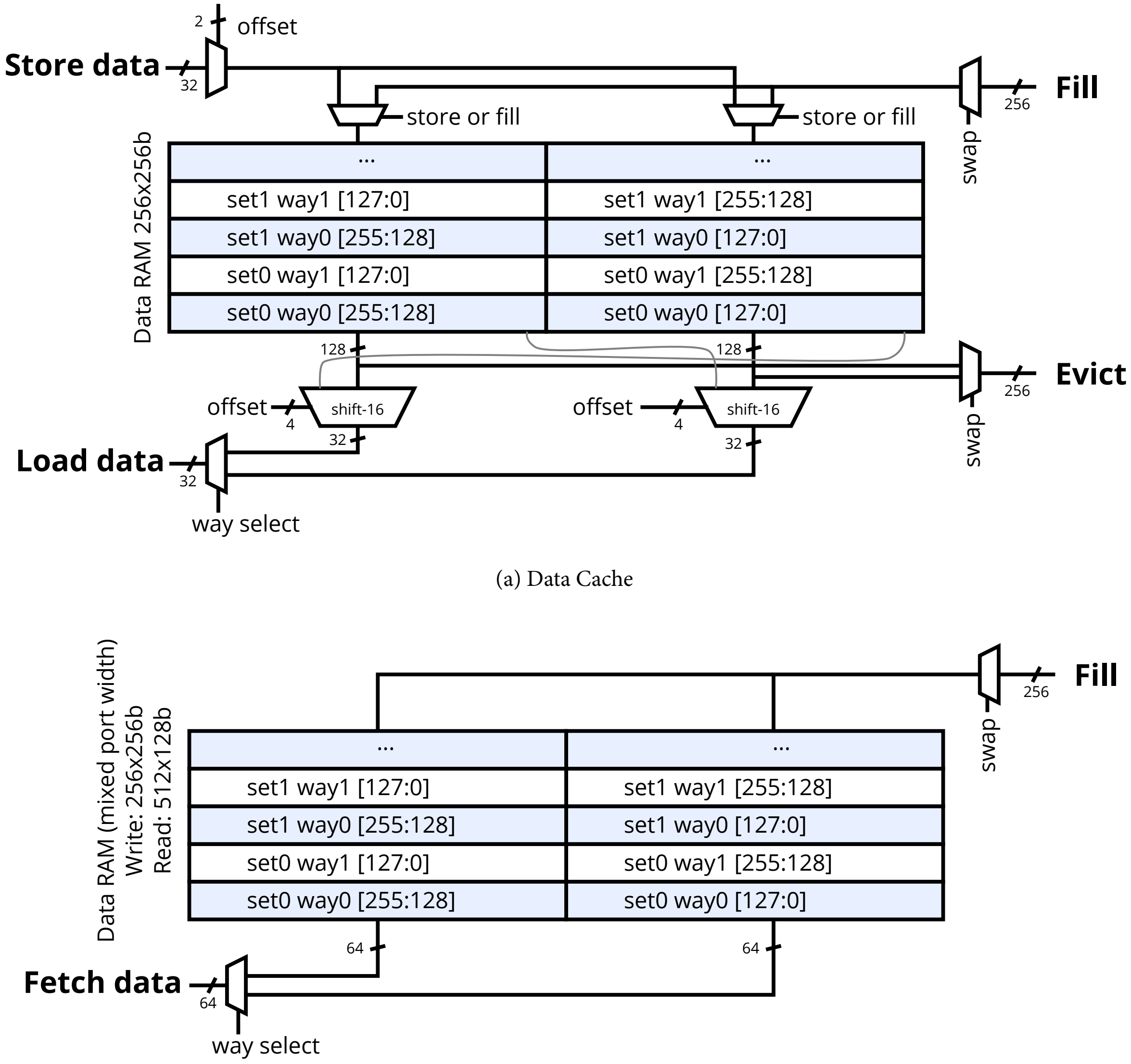
Our chosen geometry reduces hit latency with a two-way, 4 KB/way VIPT design, without alias detection logic. Its small capacity is partially compensated by a large 256 KB L2 cache. The 32-byte cache line size is a compromise between using 256-bit (32 byte) multiplexers in the coherence unit and a depth-128 (4 KB/32 B) cache tag array. The extra multiplexer needed to stitch two depth-64 MLAB blocks together was implemented reasonably efficiently by merging it into the cache tag comparator using 7-input LUTs (See Section 12.8).

12.7.2 L1 Instruction Cache

The L1 instruction cache, shown in Figure 12.8b, is derived from the data cache, with the same VIPT structure. Since the instruction cache is read-only, it omits the store port and eviction port. It also does not need to support unaligned reads, as all instruction fetches are aligned 8-byte reads. Thus, the instruction cache only has load and fill interfaces, and can use mixed port widths to reduce multiplexing.

12.7.3 Coherence Unit and L2 cache

As shown in Figure 12.2, the coherence unit has many data paths that connect the output of block RAMs to the input of other block RAMs. This occurs both for tag checks feeding the data array's read address,



(a) Data Cache

(b) Instruction Cache

Figure 12.8: L1 cache data RAM arrays and interleaving

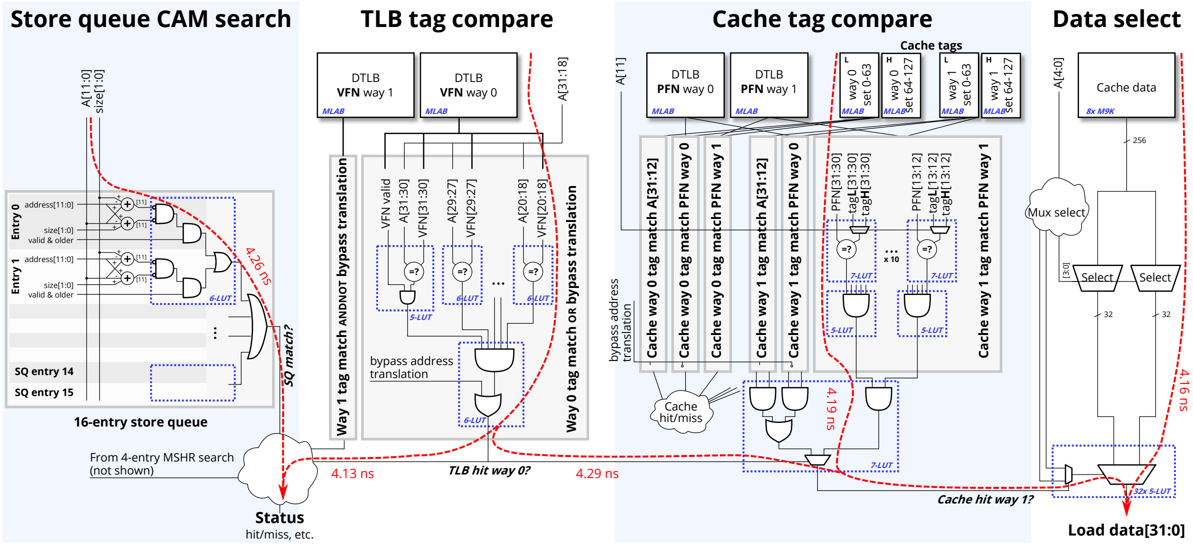


Figure 12.9: Circuit-level design of the L1 data TLB and cache access stage

and to allow copying a cache line from one cache to another. Due to the block memories' large setup times, long read latency, and routing delay, these paths have been problematic for delay. To mitigate this, there are two cycles dedicated to tag checks, and a cycle dedicated mostly to routing delays (stage 5 in Figure 12.2), but the coherence unit remains the most timing-critical block.

As mentioned in Section 12.7.1, we used cache-line sized buses throughout the coherence unit to reduce design effort at the expense of area. The bandwidth provided is more than necessary, so future designs can use multi-cycle transfers to save area.

12.8 Detailed design of the L1 TLB, Cache, and Store Queue

The TLB, store queue, and cache access stage is one of the more complex parts of the memory system. It is nearly timing critical[3], and has been carefully optimized. This stage must determine whether a given virtual address (`A[31:0]` in Figure 12.9) hits in the L1 data cache and returns the data if there is a hit. This involves address translation using the TLB, cache access and data selection, and a store queue search to find dependencies on earlier stores to overlapping addresses. It must also correctly handle a variety of corner cases. Section 12.5.2 and stage 2 of Figure 12.2 described the functionality in more detail.

12.8.1 High-Level Circuit Structure

The general circuit design principle is well-known: try to do as many operations in parallel as possible. Using a VIPT organization allows the TLB and cache lookups to be performed in parallel. As FPGAs do not have wired-OR logic, comparators are built using trees of LUTs and are relatively slow. Therefore, we also parallelize the tag comparison logic by performing all pairwise cache tag comparisons with physical

[3] The coherence unit is currently more critical by 6%.

frame numbers (PFNs), then selecting the result after the TLB tag comparison is known. The store queue lookup also occurs in parallel as it uses only the lower 12 bits of the address that is not affected by address translation[4]. The four-entry MSHR search also occurs in this cycle, but is small and non-critical, so we do not discuss it further. To support unaligned accesses, the cache data has a 32-to-1 multiplexer to allow each output byte to select any byte from the 32-byte cache line. The first 16-to-1 selection is independent of address translation and cache hit/miss status, and is done in parallel. Only the final 2-to-1 way selection depends on the result of the TLB and cache tag comparisons.

The result of this parallelization is our design in Figure 12.9. The delays of several important paths are labelled on the diagram. This is a two-way associative TLB lookup, a two-way associative cache lookup, and unaligned data selection, in 5 LUT logic levels and 4.29 ns on a Stratix IV FPGA. For comparison, the Nios II/f performs TLB and cache lookup sequentially over two cycles (E and M stages, respectively), taking a total of 6.0 ns on the same FPGA (2.1 ns for TLB tag comparisons, 3.9 ns for cache tag comparisons and generating a hit/miss signal), despite having simpler functionality. The Nios II does not have a store queue or data selection multiplexer, and does not support TLB dirty and accessed bits, and unaligned, split-cache line or split-page accesses.

12.8.2 Circuits

The store queue is a 16-entry associatively-searched structure, where each entry must decide whether a given load (address and size) partially or completely overlaps an earlier store (address and size). This is determined using an interval-intersection comparator using two 12-bit three-input adders per store queue entry, which take advantage of the three-input adders in Stratix IV FPGAs.

The 14-bit TLB tag comparison logic consists of a two-level tree for each TLB way, comparing three bits of tag in each 6-LUT leaf, followed by a 5-input AND gate.

The 20-bit cache tag comparison logic is more complicated. It is replicated six times to compare the two possible cache tags with the three possible physical frame numbers (two TLB ways, and one if translation is bypassed). Two bits of cache tag comparison is merged with the 2-to-1 multiplexer used to stitch together the two depth-64 MLABs used to create the depth-128 cache tag array, using a 7-input LUT. Altera Adaptive Logic Module (ALM)-based FPGAs can implement 7-input logic functions in a single ALM if the function can be expressed as a 2-to-1 multiplexer selecting between two 5-LUTs that share 4 inputs. A second layer of LUTs reduces the 10 comparison results down to two signals, and a final 7-LUT combines the result from the three comparators for one way of the cache, selected by the *TLB hit way 0?* signal. This structure is well balanced, with two levels of logic in both the cache tag comparators and the TLB tag comparators before the final 7-LUT.

The cache data selection requires 32-to-1 multiplexers due to the need to support unaligned load operations, selecting any byte from the 32-byte cache line. The multiplexers are decomposed into two 16-to-1 blocks that select a four-byte value from each way of the cache, which is selected once the cache hit way is known. As a result, unaligned accesses are supported with little extra delay.

[4] This optimization trades a small IPC loss for a large circuit-level improvement. It causes "4 KB aliasing" stalls also seen in other processors.

12.8.3 Manual Circuit Optimization

The circuit diagram in Figure 12.9 shows many critical portions of the circuit technology-mapped for Stratix IV ALMs. Careful manual technology mapping can produce better results than the Quartus synthesis tool. We have found two areas where a human can improve on the synthesis tool: A human can be better aware of the arrival times of signals, and can sometimes do better mapping to LUTs.

Knowledge of arrival times allows replicating non-critical logic, then selecting the result using the late-arriving signal nearer to the output of the cone of logic, which is essentially Shannon decomposition. This was used effectively in replicating cache tag comparators and in restructuring the data selection multiplexers, where it is not obvious to a synthesis tool or where the logic function needs to be designed to make this possible. Manually decomposing the logic can also be used on a smaller scale to force more aggressive replication than the synthesis would normally do, trading area for performance. The `keep` synthesis attribute prevents the synthesis tool from optimizing away the manual duplication.

More control over mapping to LUTs can be accomplished by using the `keep` directive to delimit LUT boundaries, or using `LCELL` buffers, depending on which syntax is more convenient. However, in many cases involving 7-LUTs, the synthesizer refuses to create them even with clear boundaries, and we had to resort to using low-level device primitives, which Altera calls WYSIWYG primitives.

The `keep` attribute can also be used to push non-critical logic further up a logic cone to reduce the size of the final critical LUT. This was used to keep the final stage of the data selection multiplexers no larger than a 5-LUT, which is slightly faster than if the synthesizer were allowed to combine some non-critical signals into a larger LUT.

The impact of the above manual optimizations can be approximately measured by removing the `keep` attributes and instructing Quartus synthesis to allow resynthesis of LCELL buffers and WYSIWYG primitives. This is an inexact comparison, as the desired circuit structure is still obvious in the code, rather than the typical scenario where the synthesis tool must infer technology-dependent circuit structures from technology-independent behavioural RTL code.

With manual optimizations resynthesized away, the delay of the cache access stage increased from 4.29 ns to 4.87 ns (13.5%), due to an increase from 5 logic levels to 7 (best of 30 placement random seeds). Both the TLB and cache tag comparison logic increased by one logic level. A second increase occurred in the data selection multiplexers, as Quartus synthesis did not place the most critical signal nearest the output of the cone of logic. Manual optimizations increased ALM usage by a negligible 68 ALMs (0.5%).

Overall, the synthesis tools work well for non-critical paths, and manual mapping to LUTs is only necessary in critical regions where the synthesized circuit structure is sub-optimal. However, being aware of technology mapping during microarchitecture design can enable new circuit-level optimizations and allow verifying the synthesized output's circuit structure.

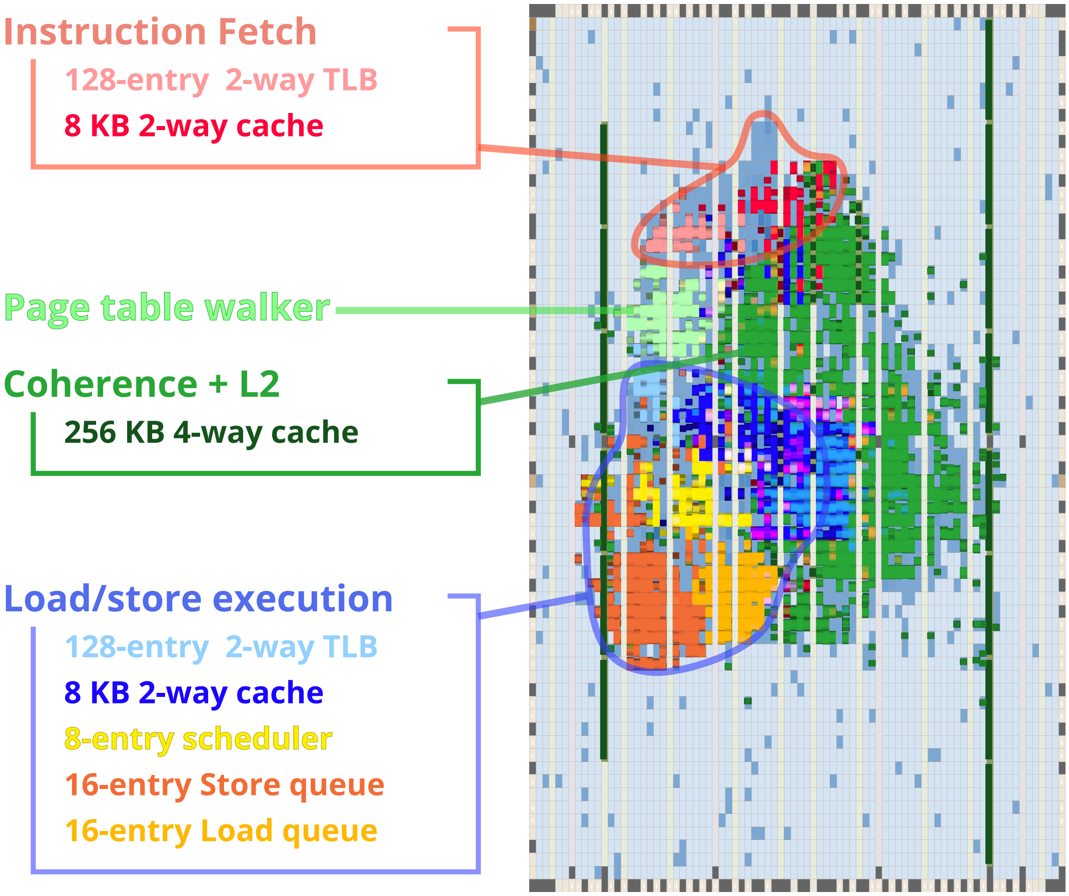


Figure 12.10: Memory system layout on the smallest Stratix IV FPGA (4SGX70)

12.9 FPGA Memory System Implementation Results

We completed the full design of the baseline configuration of the memory system (described in Table 12.2) and synthesized it on an Altera Stratix IV FPGA (smallest chip, fastest speed grade, EP4SGX**70**HF35**C2**), using Quartus 15.0.

12.9.1 Area

The total resource usage is just under 14 000 Stratix IV equivalent ALMs. Equivalent ALMs combine the area of soft logic and RAM blocks using a common unit, using scaling factors from [50]. A layout of the memory system synthesized for the smallest Stratix IV FPGA (Figure 12.10) gives a qualitative area comparison between hardware units. Table 12.3 shows a quantitative breakdown.

The coherence unit and L2 is the biggest unit due to the large L2 cache RAMs, although the L2 cache (excluding the RAM) contributes little to logic complexity (under 500 ALMs) despite the wide 256-bit buses. The coherence unit is not solely optimized for area: our choice of wide 256-bit buses and multiplexing (Figure 12.2) increases area to reduce design complexity by using single-cycle cache line transfers.

The L1 memory execution is the next biggest component. The associatively-accessed store queue,

Unit	ALM	M9K	M144K	Equiv. ALM[a]
Fetch	**995**	**8**	**0**	**1225**
ICache	488	8	0	718
ITLB	191	0	0	191
Coherence and L2	**3766**	**23**	**14**	**8164**
L2 Cache	484	23	14	4882
8-entry MSHR	326	0	0	326
Page table walk	**392**	**0**	**0**	**392**
L1 memory execution	**4462**	**8**	**0**	**4692**
16-entry Store queue	1113	0	0	1113
16-entry Load queue	528	0	0	528
8-entry Scheduler	375	0	0	375
4-entry MSHR	65	0	0	65
DCache	1513	8	0	1743
DTLB	156	0	0	156
Total[b]	**9080**	**39**	**14**	**13937**

[a] Equiv. ALMs: M9K = 28.7 ALM, M144K = 267 ALM [50]

[b] The sum of ALMs for the four modules exceeds the total ALM count, as Quartus double-counts ALMs that are shared between modules.

Table 12.3: Memory system resource usage on Stratix IV

load queue, and replay scheduler contribute about half of the memory execution unit. Although the L1 data cache has the same organization as the L1 instruction cache (2-way 8 KB), the data cache uses more than twice the area (Figures 12.8a and 12.8b) because it needs to support stores, evictions of dirty lines, and unaligned accesses, requiring shifters and more multiplexers. The instruction cache uses mixed RAM port widths to reduce this multiplexing.

Another source of cache area comes from cache tag replication. The data cache tags are replicated three times (for loads, stores, and L2 snoops), whereas instruction cache tags are replicated twice (for fetches and L2 snoops), while L2 cache tags are not replicated. This is another reason first-level caches are more expensive to scale to larger sizes than second-level caches. Indeed, the L2 cache has 32× the capacity but is less than 3× the size of the L1 data cache.

Typical out-of-order processors spend 15-35% of their core area (excluding L2 cache) on the L1 memory system. We believe our area target of ∼40 000 equivalent ALMs is achievable. Our memory system equivalent area (excluding L2 cache) of roughly 9000 ALMs is 23% of our budget, which fits comfortably in the expected range.

12.9.2 Frequency

The achieved frequency is 200 MHz on the fastest speed grade, smallest Stratix IV FPGA. Due to extensive work to maximize the frequency, the design currently has near-critical timing paths in many places (e.g., Figure 12.9), with the coherence and L2 unit being most critical. The L2 data array (using M144K blocks) with long routing paths are particularly problematic. However, removing the L2 cache only results in a

7% increase of the final frequency, as the L1 cache access stage is nearly critical (L1D Load stage 2 in Figure 12.2).

12.10 Conclusions

A major component of a high-performance processor is its memory system. In this chapter we have presented an out-of-order non-blocking memory system for a soft processor that implements many features of the x86 ISA, including those required to boot a full operating system. We have explored a number of microarchitectural options for the memory system, and showed that a two-level cache hierarchy is favoured particularly when paging is enabled. Scaling the L1 cache size is difficult due to higher latency and tag replication area, and the L2 cache is important for caching page table entries. We also showed that non-blocking caches provide a large IPC improvement even for a relatively narrow two-issue processor with the relatively low memory latency seen on low-clock speed FPGA processors. The IPC increases (1.32× SPECint2000 vs. a blocking cache, and an additional 1.60× vs. no L2 cache) reflect only changes in the memory system. We expect IPC gains to be far greater when comparing a full out-of-order processor to current single-issue in-order soft processors.

We have demonstrated high-speed circuits for our non-blocking, speculative out-of-order memory system with two cache levels — features rarely found in current soft processors. The large IPC increases come at a large but affordable area increase, but only a small frequency loss over high-frequency in-order processors. Careful microarchitecture and circuit design resulted in a faster TLB and cache lookup (4.29 ns) than the simpler Nios II/f (6.0 ns). We also saw that manual technology mapping of critical paths in our memory system improved delay over automated synthesis.

Chapter 13

Processor Performance

The preceding chapters presented the design process for our out-of-order soft processor, the trade-offs and design of each of the components of the processor, and the area and cycle time for each unit. In this chapter, we put together all of these results to provide a final estimate of the entire processor's FPGA resource usage and performance (both microarchitectural per-clock performance and the cycle time on an FPGA), and compare this to various other processors.

This chapter has two main objectives. First, we wish to compare our soft processor design to the commercially-available Nios II/f soft processor, to evaluate the FPGA resource usage and performance increases for using a higher-performance soft processor. Second, we compare per-clock processor performance to various other (hard) processors to give some context of where our particular design sits in the larger space of processor microarchitecture, and to see how our microarchitecture performs compared to existing hard processor designs of similar complexity.

Processor performance can be broken into two components: per-clock (microarchitectural) performance when running a particular workload, and the cycle time of the processor. Per-clock performance includes both instructions-per-clock (IPC) and dynamic instruction count, but we will mostly not consider them separately in this chapter because we are making comparisons across multiple instruction sets (x86, Nios II, and ARMv7) where instruction count can also vary between processors. The per-clock performance can be evaluated using cycle-level simulation of the processor with various workloads, while cycle time is evaluated using synthesized circuit designs. Most of this chapter is dedicated to measuring the per-clock performance of various processors, since per-clock performance varies with workload, while cycle time depends only on the circuit design.

In this chapter, per-clock performance is measured using a set of user-mode workloads running under Linux. Because we wish to compare to processors using different instruction set architectures, we cannot use x86-specific workloads (such as operating systems). In addition, booting operating systems (even on the same instruction set) has inconsistent behaviour due to I/O (disk) and hardware initialization (device driver) delays, which differ depending on the system configuration. Since we also compare against twenty years of x86 systems, it is impossible to keep I/O and hardware devices unchanged between machines.

Most of our comparisons are with other x86-based systems, mainly due to the lower effort required. All of our x86-based systems were able to boot off the same disk image: The highest effort task was finding suitable disks to boot from, as disk interfaces changed from parallel ATA to SATA and USB over time. In contrast, the ARM system required using a custom OS distribution built for our particular system, while the Nios-based system required cross-compiling a Linux distribution from source code along with hardware and kernel changes to create a usable system. This illustrates one of the advantages of using a standardized instruction set and system (x86 PC).

The rest of the chapter discusses our performance measurement methodology (Section 13.1), then describes the workloads used (Section 13.1.1) and the processors we tested (Section 13.1.2). To give some context to the processor performance, we first measure the memory system latency for the various systems (Section 13.2) before presenting per-clock performance results for running our workloads (Section 13.3). Finally, we combine area and cycle time estimates for our processor and compare the final wall-clock performance with other FPGA processors in Section 13.4.

13.1 Methodology

To evaluate the performance of our processor design, we need to measure the runtime of a set of workloads on both our processor and a collection of other processors.

The workloads we use in this chapter are user-mode Linux programs. The need to compare across multiple instruction sets requires that the workloads be portable across instruction sets. These workloads are similar to, but different from those presented in Table 4.1 on page 48. In this chapter, we recompiled all of the workloads and ran all workloads to completion instead of sampling part of the workload.

All of the workloads were compiled using gcc 6.2.0, targeting four instruction sets: 32-bit x86 (P5), 32-bit x86 (P6), Nios II, and ARMv7. The P6 (Pentium Pro) instruction set is a superset of the P5 (Pentium), mainly by adding conditional move instructions. We did not test any systems older than the P5. We used the P6 executables as the default because our design targets this instruction set, but used P5 executables for older systems that do not support the P6 instructions. All of the P6-capable systems, *including our processor*, ran the same executable binary. Using the same compiler for all instruction set architectures reduces the impact of compiler quality on the measured performance. This is desirable because we mainly wish to evaluate the processor microarchitecture. For the same reason, we did not enable any new x86 extensions beyond the P6 instruction set: While new vector instructions and x86-64 do provide performance improvements, these features are more architectural than microarchitectural features.

Due to the lack of a hardware floating-point unit in our processor design, we excluded workloads that have a non-negligible amount of floating-point arithmetic.

In this chapter, we ran all workloads to completion. The primary motivation for avoiding sampling is because there is no easy method to run a specified section of a workload on real hardware without manually instrumenting the source code for every workload. Because this chapter does little design space exploration, it was feasible to run all of our workloads to completion. The benchmark set has 1.7 trillion instructions (nearly all from SPECint2000), using around 72 days of CPU time to simulate. As a side effect of running to completion, we avoided the common problem of deciding how to choose a representative sample(s) of a workload [178].

Where possible, we ran the workloads from a RAM disk to minimize I/O delays. This was particularly important for the MiBench benchmarks because they are short-running, yet contain large data files, so disk speed and the time spent in OS kernel filesystem routines can significantly distort the runtime.

13.1.1 Workloads

We chose a variety of workloads, ranging from very old (and small) benchmarks (Stanford, Dhrystone), embedded benchmarks (MiBench, CoreMark), and much larger and memory-intensive benchmarks (SPECint2000). We also used two x86 simulators as workloads: Bochs 2.6.8 and our cycle-level detailed simulator of our processor[1] (s86sim), a heavily-modified simulator based on Bochs 2.4.6. While based on the same code base, the two simulators actually have little (dynamic) code in common: Our simulator spends almost all of its time doing detailed microarchitecture-level simulation of our processor, which is our code that is not part of Bochs.

Table 13.1 lists the workloads used in this chapter. The table also includes the dynamic instruction count of the workload on five different systems. The second column reports the instruction count measured using the cycle-level simulation of our processor design running the P6 binaries. The following four columns lists the instruction counts when running the P6 and P5 binaries on a Haswell system, the Nios II binaries on a Nios II/f system, and the ARMv7 binaries on an ARM Cortex-A9. In all cases, the instruction counts include all instructions, both user and supervisor mode, including OS activity unrelated to the workload (e.g., a timer interrupt).

On the x86 (Haswell) and ARMv7 systems, instruction counts were measured using the Linux perf_event interface to access the hardware performance counters. The counters were configured to measure all instructions (not just user mode) on a particular processor, and the workloads were pinned to that processor on multiprocessor systems. The perf_event interface has a fairly high overhead (e.g., on Haswell, querying a perf_event counter costs about 4000 instructions and 9000 cycles), but this is negligible because our workloads run for much longer.

Measuring Nios II instruction count is much less straightforward because the Nios II/f does not have performance counters. We measured Nios II instruction count by reverse-engineering the Nios II/f processor to tap into some of its internal control signals, using those signals to control a set of counters, then creating a Qsys memory-mapped device to allow software running on the Nios II/f to read those performance counters. Our performance counters measured clock cycles and instruction count, separately for user and supervisor mode. We verified the correct operation of these counters by creating assembly microbenchmarks with known instruction counts.

Knowing both the runtime and instruction counts for a workload enables a comparison of the architectural-level differences between instruction sets (i.e., instruction count), as well as the microarchitecture-level performance (instructions per clock, or IPC). IPC is not used for performance comparisons across different instruction sets due to the difference in instruction count, but IPC is useful for measuring how close a microarchitecture performs relative to its peak issue width.

[1] Simulators recursively simulating themselves are fun! [179, p.198]

The instruction count data can be used to indirectly verify the correctness of our processor, by observing whether the correct number of instructions are executed. Our simulations are run until completion of the workload, which is signalled to the simulator by the benchmark when the benchmark finishes[2]. If our processor did not execute the workload correctly, it would most likely either terminate after running an incorrect number of instructions, or never terminate. Therefore, executing the correct number of instructions is an indicator that execution was correct. In Table 13.1, the instruction counts for our processor and Haswell (P6) are extremely close, except for those workloads that use a significant amount of floating-point emulation (See the FPU emulation column for gcc, susan.smoothing, ghostscript, typeset, and bochs). The instruction count on Haswell tends to be around 0.3% lower than our simulator. We believe that this is mostly due to the periodic kernel timer interrupt running relatively more frequently on our processor (simulated at around "100 MHz"[3]) than on a 4.3 GHz Haswell processor.

Comparing instruction counts in Table 13.1 across instruction sets, we see slight increases (around 6% for SPECint2000) in instruction count for P5 vs P6. The RISC-like instruction sets unsurprisingly uses more instructions, by about 25-30% compared to P6. However, the relative instruction count varies significantly between workloads. Notably, both Dhrystone and CoreMark use *more* instructions on x86 than both Nios II and ARMv7.

One particularly interesting data point is SPECint2000 `mcf`, a workload known for its memory access patterns with poor locality [175]. The instruction count for the Nios II is nearly twice that for x86, and the difference was essentially all *supervisor* mode instructions. The P6 binary executed 48 billion user-mode (CPL 3) and 0.16 billion supervisor-mode (CPL 0) instructions, while the Nios II executed 42 billion user-mode and *39 billion* supervisor-mode instructions. We believe the large number of supervisor-mode Nios II instructions is due to the Nios II architecture's use of software TLB miss handling and the large number of TLB misses in `mcf`. This data point suggests that software TLB miss handling can be expensive: the Nios II/f spends almost half of the instructions (although only 21% of cycles) in the TLB miss handler for `mcf`.

To give some sense of the memory system demands of the workloads, we also listed the maximum resident set size for each workload in Table 13.1, as reported by Linux running the P6 binaries. This provides an approximation of how much memory a workload uses, but does not provide much information about its *working set* size or memory access patterns. The SPECint2000 suite clearly has a much larger memory footprint than the MiBench, Stanford, Dhrystone, and CoreMark.

Compared to earlier chapters in the thesis, we omitted several more components of MiBench due to issues noticed when trying to compare results across different machines. We omitted `qsort` because it relies on the C runtime library's qsort function, which requires using the same version of the C runtime library for a fair comparison (We saw up to 33% difference in instruction count). We excluded `pgp` for doing too much I/O while being a very short benchmark: We could not get consistent instruction count or runtime even for multiple runs on the same machine (We saw around 50% variation in instruction count

[2] We used the `CPUID` instruction with a parameter (`eax`) to signal the simulator about events such as workload start and end. `CPUID` with an invalid parameter does not throw an exception, so the same binary runs on real hardware as well.

[3] The simulated clock actually runs at a rate of 100M committed instructions per second. See Section 4.4.2 for how we keep track of simulated time.

Workload	Instruction Count					% FPU	Max. mem
	Ours (P6)	x86 P6	x86 P5	Nios II	ARMv7	emulation	(MB)
SPECint2000			$\times 10^9$				
bzip	273.7	273.3	306.4	342.5	319.6		186
crafty	192.5	192.1	209.3	245.3	245.0		3
gap	197.6	197.1	217.0	220.8	205.0		194
gcc	118.7	113.5	114.9	195.4	159.3	4	144
gzip	284.4	283.5	308.6	361.3	366.7		182
mcf	48.2	48.1	47.9	81.1	64.2		80
parser	281.4	280.1	294.5	320.0	447.6		28
vortex	307.8	306.2	293.8	380.2	346.3		67
MiBench 1.0			$\times 10^6$				
bitcount	573	572	591	512	741		2.7
susan.smoothing	404	379	518	323	257	6	1.4
dijkstra	171	171	171	155	208		2.6
blowfish.enc	770	754	794	1036	865		3.1
blowfish.dec	770	757	796	1035	860		3.1
rijndael.enc	301	298	300	493	390		1.9
rijndael.dec	298	297	294	478	396		2.0
sha	139	137	130	126	106		2.6
ghostscript	1358	936	940	1382	934	31	4.0
ispell	985	969	972	1327	1219		2.6
stringsearch	4.00	3.97	3.99	4.98	4.20		2.7
jpeg.enc	86	85	94	105	97		2.4
jpeg.dec	22.8	22.2	25.0	25.7	20.2		1.2
mad	260	258	260	241	217		2.7
tiff2bw	184	155	214	518	166		2.6
tiffdither	964	956	979	1455	997		2.6
tiffmedian	588	561	620	1411	636		2.6
typeset	426	400	416	548	483	4	7.5
adpcm.enc	841	839	791	618	803		0.9
adpcm.dec	674	684	669	518	544		0.9
crc32	1724	1688	1733	2300	1996		2.6
gsm.enc	1085	1080	1170	1321	986		1.9
gsm.dec	498	492	506	493	579		1.1
Stanford			$\times 10^6$				0.6
perm	0.87	0.87	0.91	1.36	1.03		–
towers	0.67	0.67	0.71	1.08	0.79		–
queens	0.54	0.54	0.58	0.60	1.01		–
intmm	0.53	0.53	0.53	0.58	0.39		–
puzzle	5.35	5.35	5.33	5.31	5.75		–
quicksort	0.86	0.86	0.93	0.74	1.14		–
bubblesort	1.01	1.01	1.01	1.03	1.27		–
treesort	1.90	1.90	1.94	1.98	2.33		–
dhrystone	231	224	213	196	202		0.5
coremark	221	221	258	190	199		0.6
bochs	3292	2697	2817	3236	2842	18	21.6
Our simulator	5515	5504	5577	6863	6504		46.3

Table 13.1: Benchmarks and instruction counts

between runs). The `tiff` benchmarks also need special handling as the OS's I/O code contributes up to a third of the entire benchmark, resulting in non-negligible variations between systems: For example, the instruction count is 23% higher when running `tiff2bw` from a disk compared to a RAM disk (tmpfs), with the entire difference due to OS (supervisor mode) code.

13.1.2 Descriptions of Systems Compared

After choosing the set of workloads and a measurement methodology, we need to choose a set of machines to compare. Since we designed an FPGA soft processor, we compare against the two most obvious options for FPGA processors: A pipelined RISC soft processor offered by an FPGA vendor (Altera Nios II/f) and an embedded ARM hard processor system (ARM Cortex-A9) available on some recent FPGAs. We also compared against a variety of x86 microarchitectures to provide context within the large design space of existing x86 processors, both historical and recent, both in-order and out-of-order, and both high-performance (for its time) and low-power designs.

Table 13.3 lists the systems we compare. The rest of this section will briefly describe each of these systems.

Our processor design

By this point in the thesis, the microarchitectural parameters of our processor design have already been decided. Table 13.2 lists the microarchitectural parameters we used. A block diagram of our microarchitecture (originally from Figure 5.1) has been reproduced here (Figure 13.1).

To make a reasonably fair comparison with other processors, we need to model a memory system with reasonable performance. For our simple performance model of main memory, this means choosing the memory latency. Choosing a representative latency is harder than it seems because memory system latency can vary by a factor of ten between systems of the same memory type, and by a factor of four even for the same memory on the same FPGA and board. Section 13.2.1 will present some measurements of the raw memory latency on various systems.

We chose to model a (raw) memory system latency (excluding cache and pipeline delays) of 30 cycles. This was chosen because we thought 150 ns (30 cycles at 200 MHz) of main memory latency could be achieved on a low-cost memory system found in embedded systems. For comparison, high-performance desktop processors using commodity DDR3 SDRAM achieve around 40 ns of (raw) latency, while on a Cyclone V SoC FPGA, the hard processor system with DDR3 SDRAM achieves approximately 125 ns while the Nios II using the same memory controller achieves around 390 ns. As suggested by our calculations from Section 3.5.5 that suggested FPGA memory controllers might achieve around 85 ns of latency, we expect that the memory latency of a soft processor *could* be significantly better than the 390 ns we measured on this particular Nios II/f system. In our memory model, we did not impose a memory bandwidth constraint. Due to the low clock speed of soft processors (and unlike most hard processors), our sustained L2 cache bandwidth (one 32-byte cache line every two clock cycles at 200 MHz, or 3.2 GB/s) is well below the bandwidth an off-chip DRAM memory system can provide (e.g., one 64-bit DDR3 DIMM at 400 MHz or 800 MT/s already provides 6.4 GB/s).

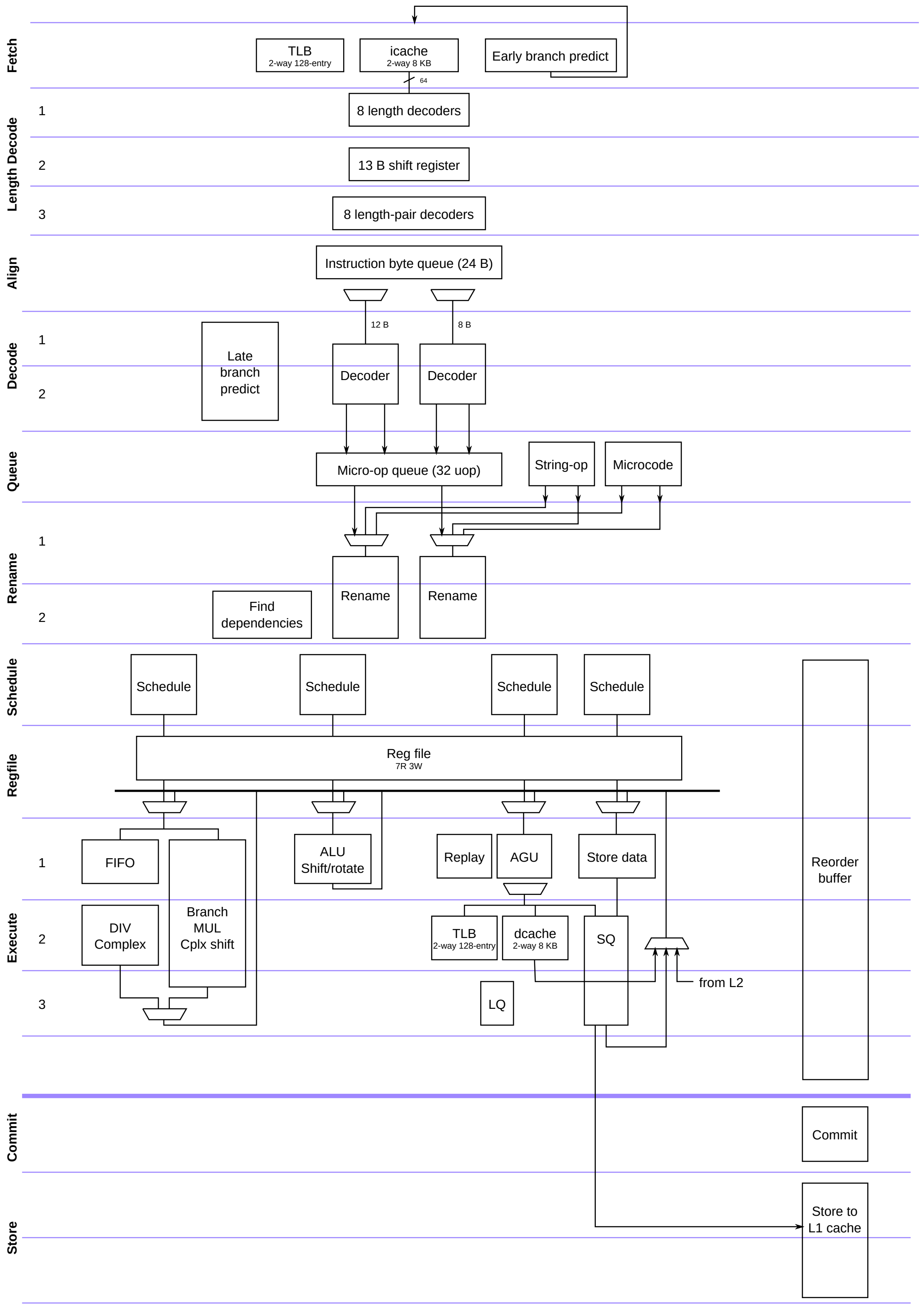


Figure 13.1: Processor core microarchitecture (Reproduced from Figure 5.1)

Parameter	Value
Fetch	8 bytes per cycle
Instruction Cache	8 KB, 2-way, 32-byte lines
Instruction TLB	128-entry, 2-way
Branch prediction	Overriding predictor. Fetch: 4-way 16k-entry BTB, 16-entry RAS; Decode: 8K-entry gshare, 512-entry 1-way indirect branch, 16-entry RAS
Decode	2 x86 instructions per cycle, 2 micro-ops per cycle
Rename	2 micro-ops per cycle
Reorder buffer	64 micro-op reorder buffer
Commit	2 micro-ops per cycle
Scheduler	10/10/7/5 entries, distributed
Execute	1 branch/complex, 1 ALU, 1 AGU, 1 store-data
Execution latency	3 cycles complex, 1 cycle simple ALU, 3 cycle load
Memory execution	16-entry store queue, 16-entry load queue
Main memory	30-cycle latency (150 ns at 200 MHz)
L1 Data Cache	8 KB, 2-way, 32-byte lines
L2 Cache	256 KB, 4-way, 32-byte lines
Data TLB	128-entry 2-way

Table 13.2: Simulated microarchitectural parameters

For the purpose of a fair per-clock performance comparison with the Nios II/f we tested, choosing 30 cycles of latency is conveniently a good approximation. The cache miss (with TLB hit) latency of our Nios II/f system at 100 MHz is 39 cycles, while 30 cycles of raw latency on our design results in 44 cycles of L2 cache miss (with TLB hit) latency, which is fairly close to the Nios II/f. (All other things equal, a system with two levels of cache should be expected to have slightly higher effective main memory latency due to the extra delay for querying the second-level cache before sending the memory request to the memory controller.)

ARM Cortex-A9

The Cortex-A9 is an out-of-order processor capable of decoding two instructions per clock cycle, with four execution units (two ALUs, memory, and floating-point) [180].

The ARM Cortex-A9 processor we tested is part of the Altera Cyclone V SoC FPGA, on a DE1-SoC FPGA development board. The Cyclone V SoC includes a hard processor system with two Cortex-A9 processors, which we ran at 800 MHz. The processor is attached to a 32-bit DDR3 SDRAM interface at 400 MHz (800 MT/s).

Nios II/f

The Nios II/f is a 6-stage single-issue in-order pipelined processor [17]. There are currently two versions of the Nios II/f: Gen2 and Classic [17, 181]. The two versions have essentially the same area and performance characteristics. We tested Gen2, but also briefly tested Classic and observed no performance difference between the two.

Being a soft processor, several microarchitectural parameters are configurable. We chose values that tuned the processor toward higher performance. We configured our Nios II/f with 32 KB instruction and data caches, an MMU with 6-entry DTLB, 4-entry ITLB, and 128-entry L2 TLB, a radix-2 divider, 32-bit multiplier, and 256-entry dynamic branch predictor.

The Nios II/f system was implemented as part of the same Cyclone V SoC FPGA used for the ARM Cortex-A9, running at 100 MHz. Its memory system is the same as the one used for the A9: 32-bit DDR3 at 400 MHz. However, memory requests traverse through a FPGA-to-HPS SDRAM bridge in order to access the hard processor system's memory controller from the FPGA, which appears to add a large amount of latency (around 250 ns extra). Unfortunately, the FPGA-to-HPS SDRAM bridge seems to be the fastest method to use the hard memory controller on this FPGA SoC.

We were able to use 472 MB memory out of the 1 GB on the board. Attempting to use more memory would cause Linux to hang early in the boot process.

Intel Pentium

The Pentium (also known as P5) was Intel's first superscalar x86 processor, capable of two-issue in-order execution. It was the microarchitecture that preceded the out-of-order Pentium Pro.

Our test system used 256 MB of SDRAM. From our test systems, the Pentium and K6 processors shared the same motherboard, and therefore have the same L2 cache and memory system.

Intel Pentium Pro

The Pentium Pro (also known as P6) was Intel's first out-of-order x86 processor. The Pentium Pro can decode and commit three instructions per clock cycle, and has 5 execution units (two ALUs, load address, store address, and store data) [103]. The microarchitecture later evolved into the Pentium II and Pentium III (and many more) microarchitectures. Our test system used a Pentium Pro processor with 256 KB of in-package L2 cache (The Pentium Pro is a dual-die package) and 320 MB of registered EDO DRAM memory.

Looking solely at issue width and execution unit count would suggest that the Pentium Pro would perform slightly better than our two-way design with 4 execution units.

Intel Pentium 4

After years of evolving the P6 microarchitecture, the Pentium 4 was a new design that traded per-clock performance for higher clock speeds. After several evolutions of this microarchitecture, high clock speeds eventually led to excessive power consumption, and this microarchitecture was eventually abandoned in favour of P6-derived designs. We tested the second iteration of the microarchitecture (130 nm Northwood). Our test system used 2 GB of DDR SDRAM memory.

Intel Nehalem

Nehalem is a high-performance four-issue out-of-order microarchitecture from 2008. We included this processor because there is an FPGA emulation of it based on the RTL code for this processor [21]. Our test system used 8 GB of DDR3 SDRAM memory.

Intel Haswell

Haswell is a high-performance microarchitecture from 2013, and the newest microarchitecture we tested. Our test system used 16 GB of DDR3 SDRAM memory.

Intel Atom Bonnell

The Intel Atom (Bonnell) is a two-issue in-order design, targeting lower-performance low-power applications. Recent low-power processors provide interesting comparisons, as their microarchitectures are a recent design, yet with high-level parameters (such as issue width) similar to the much older designs (e.g., both Bonnell (2008) and Pentium (1993) are two-issue in-order processors). Due to improved process technology (45 nm vs. 350 nm), the Atom has much higher clock speeds (and thus, higher memory latency in clock cycles), but also much larger caches (24 KB L1 + 512 KB L2 vs. 8 KB L1).

Our test system used 1 GB of DDR3 SDRAM memory.

Intel Atom Silvermont

The out-of-order Silvermont is a relatively new x86 microarchitecture that was designed for low power rather than high performance. Despite the shared name, it is a different design than the Bonnell. It is an interesting comparison because it is a modern design, yet targets the same issue width (2-way) as our design.

Our test system used 2 GB of DDR3 SDRAM memory.

AMD K6

The K6 was AMD's second out-of-order x86 microarchitecture (following the K5), providing a second example of an out-of-order microarchitecture from around the same time as the Pentium Pro. Our test system used the same motherboard as our Pentium test system (256 MB of SDRAM).

AMD Piledriver

The Piledriver is AMD's high-performance microarchitecture from 2012. Our test system used 16 GB of DDR3 SDRAM memory.

Processor	ISA	Issue	Data caches	Mem[1]		Frequency	Process	Year[2]
		width	L1/L2/L3 (KB)	clk	ns	MHz		
Out-of-Order Processors								
This work	x86 (P6)	2	8/ 256/	73	365	∼200	Stratix IV	—
ARM Cortex-A9	ARMv7	2	32/ 512/	190	240	800	28 nm	2007
Intel Pentium Pro	x86 (P6)	3	8/ 256/	87	370	233	350 nm	1995
Intel Pentium 4	x86	3	8/ 512/	400	140	2800	130 nm	2002
Intel Atom Silvermont	x86-64	2	32/1024/	335	140	2420	22 nm	2013
Intel Nehalem	x86-64	4	32/ 256/ 8192	190	59	3200	45 nm	2008
Intel Haswell	x86-64	4	32/ 256/ 8192	225	53	4300	22 nm	2013
AMD K6	x86 (P5)	2	32/1024/	78	470	166	350 nm	1997
AMD Piledriver	x86-64	4	16/2048/ 8192	217	62	3500	32 nm	2012
VIA Nano	x86-64	3	64/1024/	177	177	1000	65 nm	2009
In-Order Processors								
Nios II/f	Nios II	1	32/ /	150	1500	100	Cyclone V	2008[3]
Intel Pentium	x86 (P5)	2	8/1024/	85	425	200	350 nm	1993
Intel Atom Bonnell	x86-64	2	24/ 512/	231	144	1600	45 nm	2008
VIA C3	x86 (P5)	1	64/ 64/	235	430	550	150 nm	2001

[1] We report the read latency for randomly accessing a 128 MB array, including any cache and TLB effects. See Section 13.2 on the difficulty of measuring memory latency.

[2] We list the year of introduction of the first implementation of the microarchitecture, rather than the release date of the specific implementation we tested.

[3] The Nios II was released in 2004 [181], with the MMU first appearing in 2008 [181]. We tested the Nios II Gen2 released in 2015 [17], which is essentially unchanged in performance and area compared to the Nios II "Classic"

Table 13.3: Test system microarchitectures

VIA C3

The VIA C3 is a single-issue in-order microarchitecture. It was mainly used in low-power/low-performance systems. Our test system used 512 MB of SDRAM.

VIA Nano

The VIA Nano is a three-issue out-of-order microarchitecture designed by Centaur Technologies. Like the Atom, it is found mainly in low-power, lower-performance systems than high-performance processors from Intel and AMD. Like the Atom Silvermont (but earlier by four years), it is an example of a recently-designed out-of-order microarchitecture that did not target the high-performance market.

Our test system used 1 GB of DDR3 SDRAM memory.

13.2 Memory System Latency Measurements

In this section, we seek to characterize the memory system latency of the various systems we tested. There are two motivations for this. First, we wish to know what is a "typical" memory latency for an FPGA-based memory system so we can make a fair comparison between our simulated memory system and current FPGA processors. Second, we can use these measurements to evaluate the effect of cache and TLB designs on the overall latency of the memory system, as seen by software.

In this section, we measure memory latency using a microbenchmark that uses a random cyclic access pattern over a fixed-size array. The microbenchmark cycles through a random permutation of the cache lines in the array, visiting each cache line in the array in a random order, exactly once, before repeating the cycle again. The inner loop of the microbenchmark consists only of the loop index computation (decrement and conditional branch) and data-dependent load instructions (unrolled), so there is minimal arithmetic overhead. On most architectures (especially out-of-order), the loop overhead is completely hidden by the higher-latency data-dependent loads. We then increase the size of the array being randomly accessed and observe how the access latency increases as various caches and TLBs are overflowed.

13.2.1 Measuring raw main memory latency

We wish to know the typical latency of a soft processor's memory system, excluding any delays caused by the processor design itself, so that we can simulate an appropriate memory latency that gives reasonably fair comparisons with our processor.

Measuring the portion of effective memory latency of a system that is caused solely by the main memory is surprisingly difficult. It is inevitable that any measurement of main memory (i.e., a cache miss through all levels of cache) must include the delay of detecting the cache misses and restarting the processor pipeline. We can approximately compensate for this by subtracting out the last-level cache *hit* latency, however it is actually not possible to measure the cache *miss* latency: It is only possible to measure the cache *hit* latency, which can be greater than the delay to detect a cache *miss*. However, to make things even more complicated, many microarchitectures have TLBs that are too small to cover much beyond its caches (e.g., the Pentium Pro has a 64-entry TLB that covers 256 KB but an L2 cache of 256 KB), and it becomes impossible to choose an array size or access pattern that will reliably cause last-level cache misses but no TLB misses. Newer microarchitectures that support large page sizes work around this problem, because the same number of TLB entries covers much larger arrays.

We were able to measure with reasonable accuracy the memory latency of recent x86 systems (using large 2 MB page size), the Nios II/f (by creating a configuration with a very small data cache), and our simulated processor design (we have large TLBs). Figure 13.2 shows these measurements.

Figure 13.2a shows measurements (using small 4 KB pages) of the Nios II/f at 100 MHz and our processor (cycle-level simulation). We measured two configurations of the Nios II/f, one with 32 KB L1 cache and one with 512 bytes of L1 cache. Using a very small L1 cache allowed measuring the memory latency without any TLB misses (The L1 data TLB is 6 entries, covering 24 KB). The main memory access latency, including the delays of restarting the processor pipeline, is around 39 cycles (or 390 ns at 100 MHz). The delay of main memory alone is likely a few cycles lower, but we have no method of measuring it.

On the Nios II/f, memory latency starts increasing at 24 KB due to misses in the 6-entry L1 DTLB. Between 64 KB and 512 KB, both configurations of the Nios II/f have the same memory latency: At those array sizes, memory accesses on both configurations experience a data cache miss, a L1 DTLB miss, and a L2 TLB hit. Beyond 512 KB, the Nios II/f with a smaller L1 cache again sees higher memory latency.

because the L1 cache size again affects how much of the page tables (in memory) are cached, increasing the TLB miss latency of the small-cache configuration.

For our design (shown in the green curve in Figure 13.2a), we use a large 128-entry L1 TLB, so TLB miss effects only affect array sizes greater than 512 KB. The L2 cache hit time is 9 cycles, while the L2 miss (and TLB hit) time is 44 cycles. Their difference (35 cycles) is the memory access latency including some overhead (Recall from Section 12.5 that memory requests pass through the L2 pipeline a second time when returning to the CPU). This measurement illustrates the difficulty of isolating the portion of latency that is entirely outside of the processor: In our case, we know the memory latency is 30 cycles, while the measurement shows 35 including overhead. Using a simulated memory latency of 30 cycles matches the Nios II/f's measured latency reasonably well, which allows a reasonably fair per-clock performance comparison between our design and the Nios II/f.

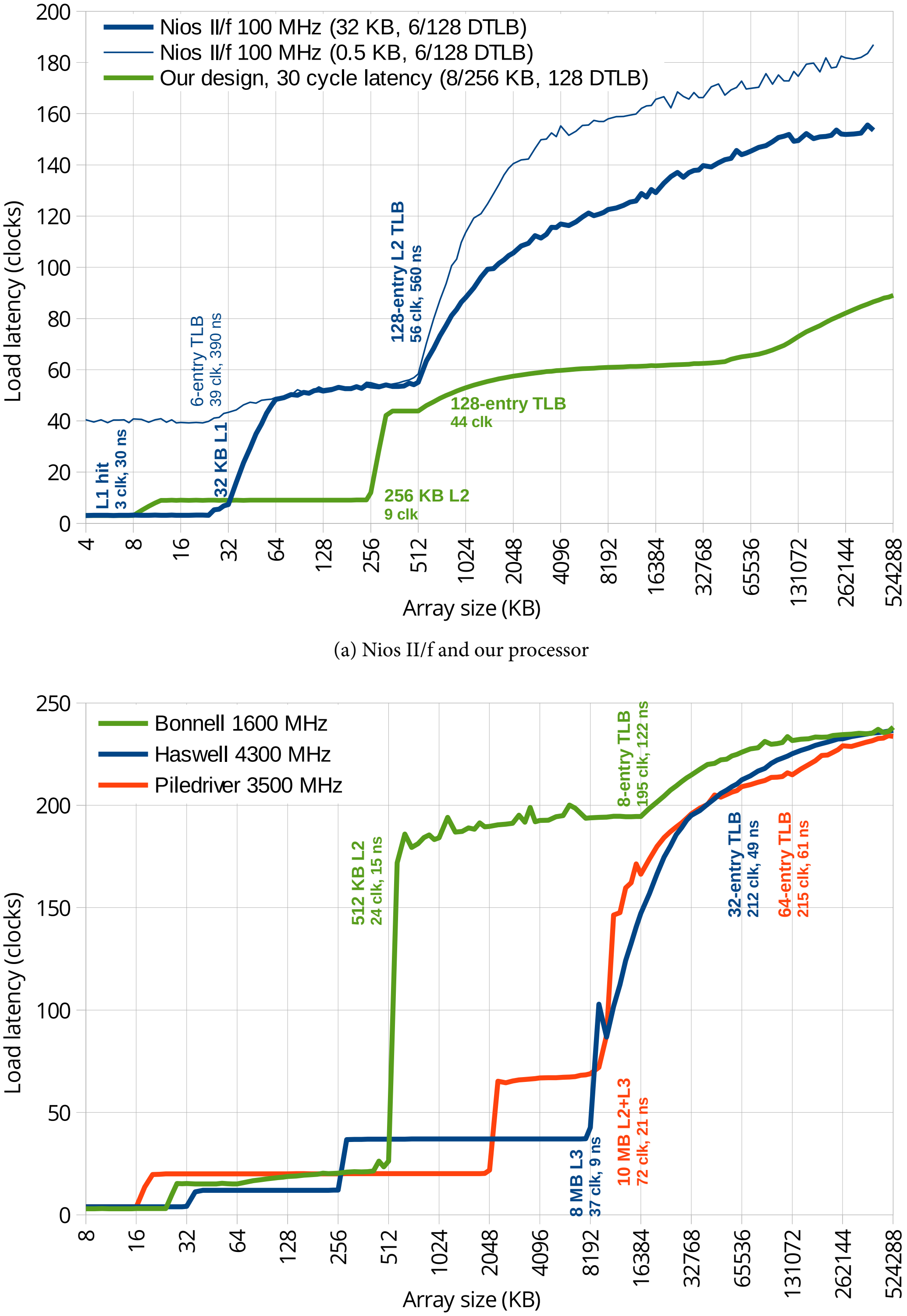
Figure 13.2b shows similar measurements on several x86 systems, using large 2 MB pages to reduce the impact of TLB misses. The memory latency including overhead is around 171 cycles ($195 - 24$) or 107 ns (at 1.6 GHz) for the Intel Atom (Bonnell, in-order)[4], 175 cycles or 41 ns (at 4.3 GHz) for Intel Haswell, and 143 cycles or 41 ns (at 3.5 GHz) for AMD Piledriver. The difference in memory latency between the "high-performance" and "low-power" systems is surprisingly large (2.6 times) despite all three systems using DDR3 SDRAM. But the 390 ns Nios II/f is much slower still.

Our Nios II/f is implemented in a Cyclone V SoC FPGA, containing a hard ARM processor system including DDR3 SDRAM controller. The memory system connects the CPU through a FPGA-to-HPS SDRAM bridge to access DDR3 SDRAM through the hard memory controller. The unusually slow Nios II/f memory latency leads us to ask whether the memory latency is this high for the entire memory system, or whether is it due to a slow FPGA-to-HPS SDRAM bridge. To do this, we measure the memory latency from the ARM Cortex-A9 hard processor on the same chip, using the same memory controller and DRAM, but without passing through the FPGA. Figure 13.2c shows measurements using 4 KB pages[5]. Due to the use of 4 KB pages, it is difficult to separate out the latency caused by TLB misses from the latency of main memory. We estimate the memory latency to be around 104 cycles ($140 - 36$), or 130 ns (at 800 MHz). While there is considerable uncertainly about this number, it is clear that it is much better than the memory latency from the Nios II/f (390 ns), and comparable to the low-power/low-cost Intel Bonnell (107 ns). We conclude that soft processor memory systems could be made to match the latency of low-cost PCs, and are not necessarily three times worse.

For the remainder of this chapter, we will model a main memory latency of 30 cycles for our processor. This provides a reasonably fair per-clock performance comparison with the Nios II/f when both processors are running at a similar clock speed, and is also a reasonable (slightly pessimistic) estimate of the achievable memory latency for a low-cost embedded system if our processor ran at 200 MHz (30 cycles at 200 MHz is 150 ns).

[4] It can be seen in the plot that Bonnell's 16-entry L1 DTLB actually fractures 2 MB pages into 4 KB TLB entries, and costs about 10 cycles to access the 8-entry L2 TLB when the array size exceeds 16 4 KB pages (64 KB).

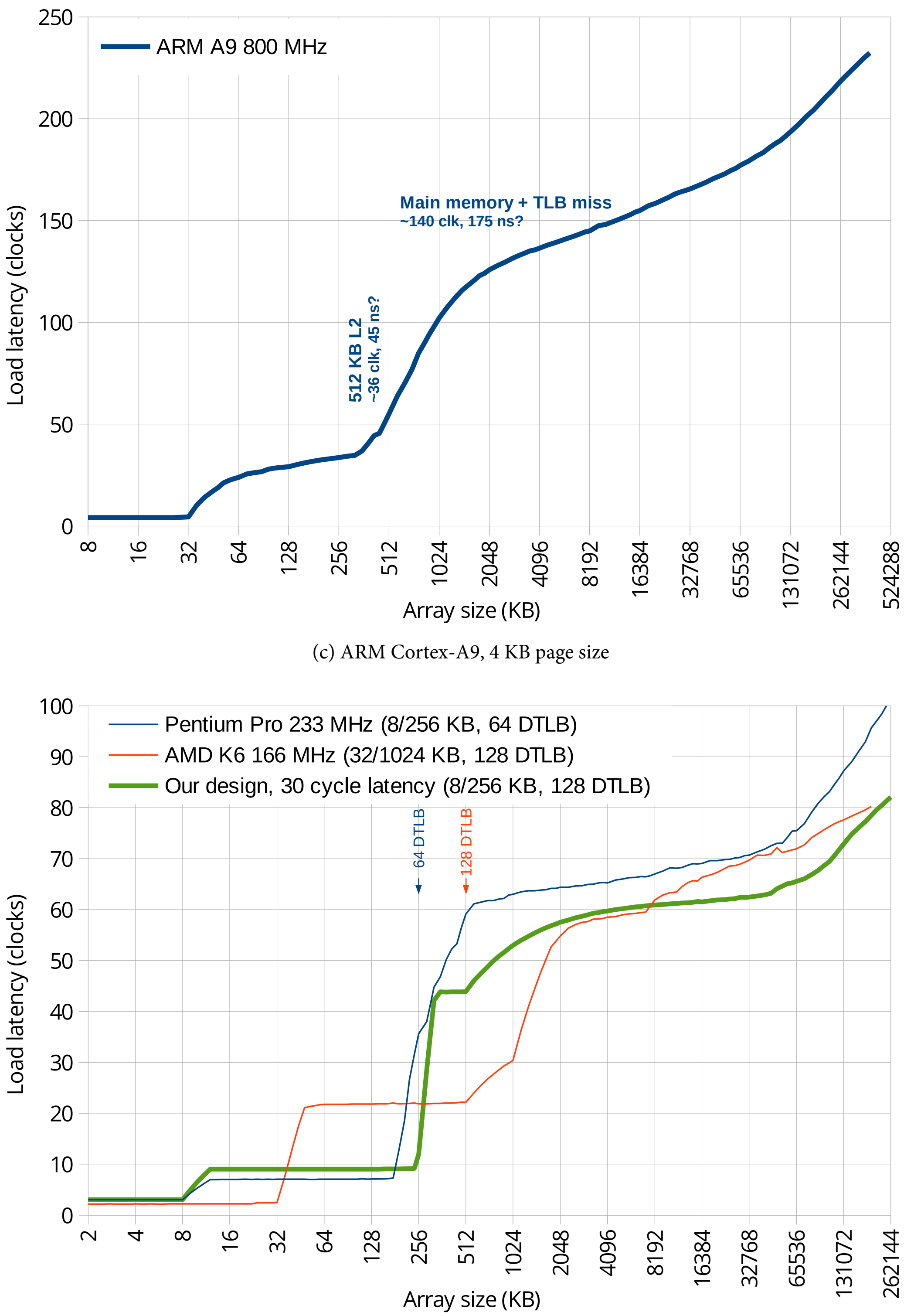
[5] The Linux mainline kernel does not support huge pages without LPAE (Large Physical Address Extensions), which the Cortex-A9 does not have.



(a) Nios II/f and our processor

(b) x86 processors using 2 MB page size

Figure 13.2: Measuring memory system latency



(c) ARM Cortex-A9, 4 KB page size

(d) Pentium Pro and AMD K6

Figure 13.2: Measuring memory system latency (continued)

13.2.2 Effect of caches and TLB

The previous section attempted to find the memory latency of a reasonable FPGA-based memory system, so that we can make a fair comparison to the existing Nios II/f soft processor. In this section, we use the same experiments, but look at the performance impact of the entire memory system design, including caches and TLBs.

The impact to the performance of the processor is not solely determined by the raw latency of the memory system, but by the effective latency actually perceived by load/store instructions running on the processor, including the effects of caches and TLBs. In this section, we compare the Nios II/f with our processor design, as well as two out-of-order x86 designs running at a similar clock rate. Thus, we are comparing the performance of the cache and TLB systems between systems with similar raw memory latency, both in clock cycles and in wall-clock time.

We will first revisit Figure 13.2a, a comparison between Nios II/f and our processor where the raw memory latency is similar. Because the Nios II/f has larger 32 KB data caches, its memory latency is lower than our design that has 8 KB L1 cache, for arrays below 32 KB. However, our larger L1 TLB (128 entries vs 6) and the 256 KB L2 cache keeps latency lower for larger array sizes. For both systems, page table walks occur for arrays beyond 512 KB because both systems have 128-entry (L1 for our design, L2 for Nios II/f) TLBs which cover a 512 KB virtual address range. At 512 KB, the Nios II/f memory latency increases much faster than our design because the Nios II/f has slower TLB miss handling. The Nios II architecture uses software TLB miss handling (by throwing an exception), while x86 designs use hardware page table walks. In addition, page table walks are cached by the L2 cache, keeping TLB miss (page table walk) latency low. It appears Nios II/f page table walks are also cached by its L1 cache (the TLB miss handling time is higher if the L1 cache is smaller), but does not appear to be as effective, possibly because a software TLB miss handler makes multiple cached memory accesses beyond just reading the page tables.

Figure 13.2d shows the same comparison, but with two x86 systems with similar clock speeds: the Intel Pentium Pro at 233 MHz, and AMD K6 at 166 MHz. We do not compare to more recent processors here because it is difficult to make meaningful comparisons when the memory latency (in clock cycles) differs drastically between systems. The AMD K6 has larger caches, but the 1 MB of L2 cache is located on the motherboard, and thus much slower than an on-chip or on-package L2 cache. Unlike for the Nios II/f, all three x86 designs behave similarly for large array sizes, as all designs use hardware page table walks that are cached. As we will see in the next section, the relatively poor Nios II/f memory system performance for large array sizes tends to disproportionately hurt workloads with large working sets such as SPECint2000 compared to small benchmarks such as Dhrystone and CoreMark.

13.3 Performance-per-Clock Results

This section compares the per-clock performance for running the workloads described above (in Section 13.1.1) on a cycle-level simulation of our processor and on other systems (described in Section 13.1.2). This section focuses on per-clock performance, which attempts to measure the processor microarchitec-

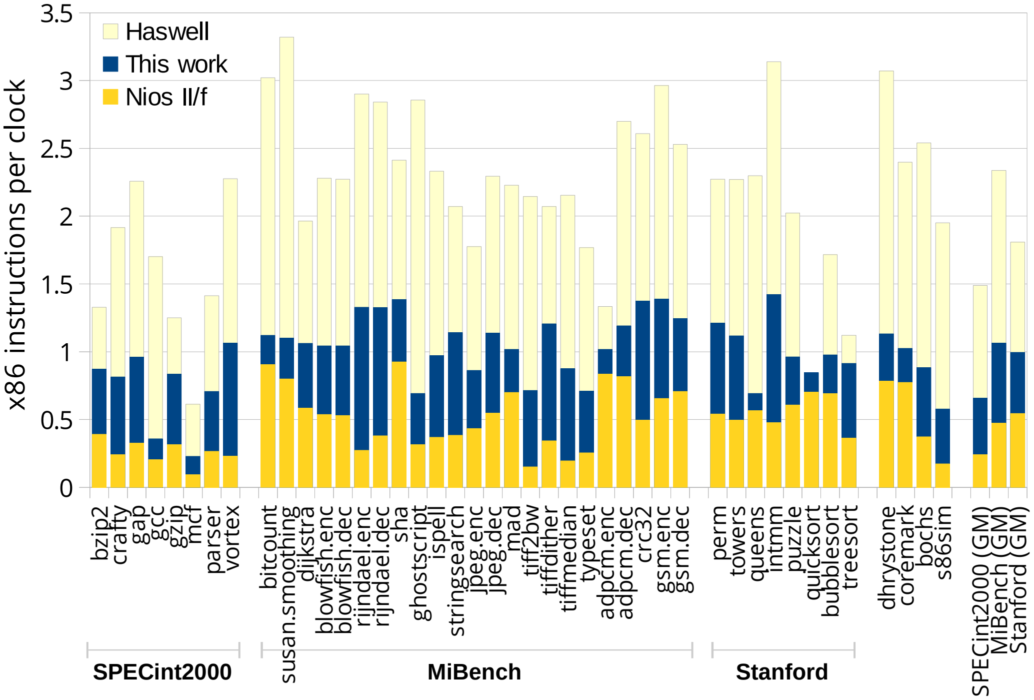


Figure 13.3: Per-clock performance (in x86 instructions per clock) for Nios II/f, our design, and Intel Haswell. Includes geometric means for SPECint2000, MiBench, and Stanford suites.

ture's contribution to processor performance, separating out the impact of the implementation technology. While microarchitecture can impact wall-clock performance via clock speeds by pipelining, most of the variation in clock speeds is due to the implementation technologies: The systems we tested span clock frequencies from 100 MHz to 4300 MHz and implementation technologies including FPGAs and 20 years of silicon fabrication technology (from 350 nm to 22 nm). We will incorporate area and clock speed comparisons in the next section, where we compare only to the same implementation technology.

We begin this section with a comparison between our processor, the Nios II/f, and Intel Haswell (the fastest processor in our tests). We then present the same comparison with more processors, but present only the geometric means of each benchmark suite due to the large amount of data. We then evaluate the sensitivity of our processor design to the memory latency we assumed in our cycle-level simulation.

Figure 13.3 and Table 13.4 compare our processor design to the Nios II/f and Intel Haswell, for each benchmark. The metric used in this comparison is the number of non-floating-point-emulation x86 instructions (as measured on our processor's simulation) divided by the number of processor clock cycles to finish the benchmark on the measured processor. We use a unit of non-floating-point-emulation x86 instructions because it reflects the amount of useful work performed by the benchmark, while excluding microarchitectural effects such as choosing to emulate the floating-point unit. This metric is proportional to per-clock performance (amount of useful work performed per clock cycle), but is not necessarily the same as instructions-per-clock (IPC). The metric is equivalent to IPC on x86 systems that do not need floating-point emulation or on benchmarks that have negligible use of floating-point instructions. On

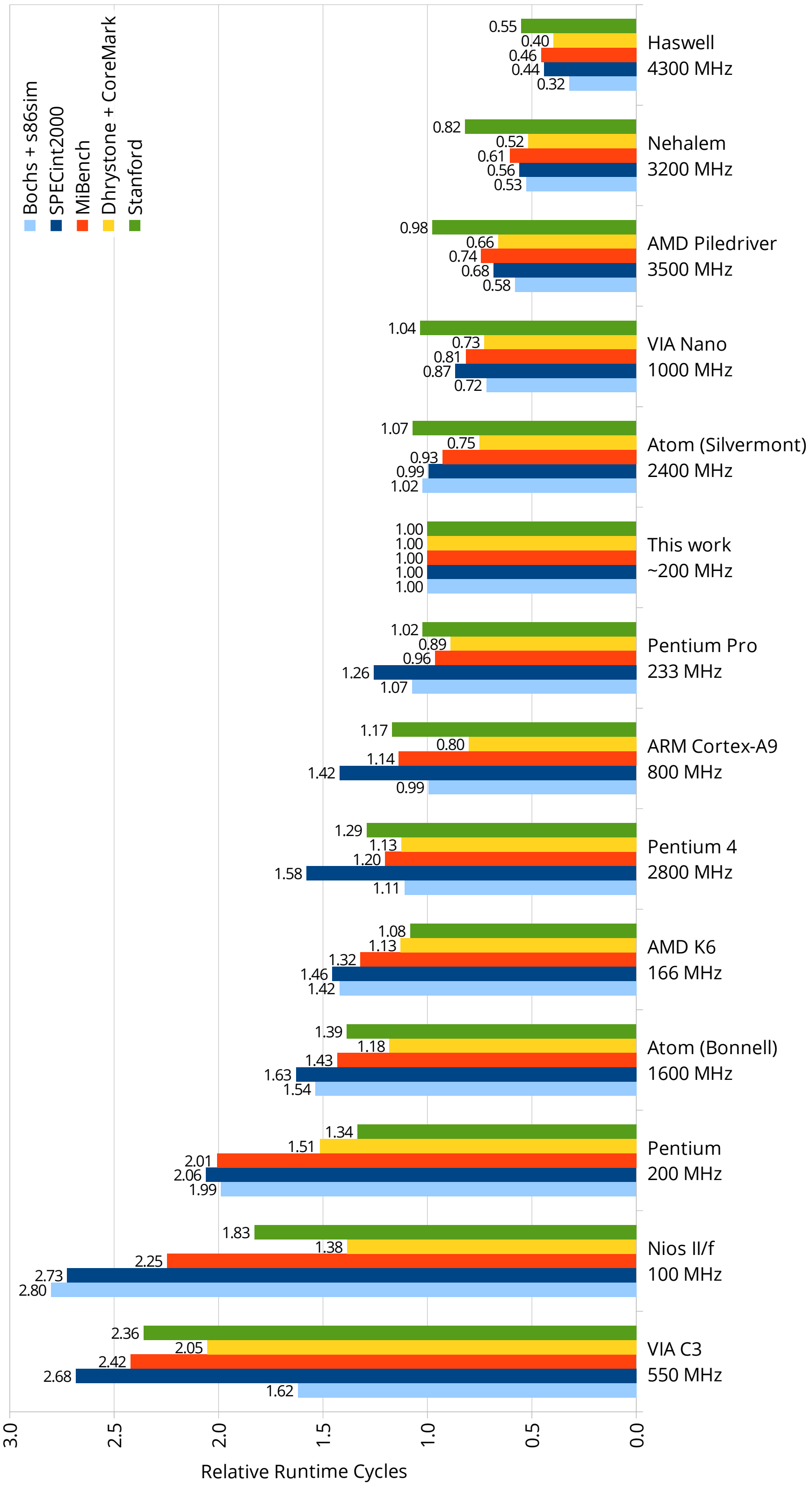


Figure 13.4: Relative per-clock performance, compared to our processor

Benchmark	Nios II/f	This work	Haswell	This work vs. Nios II/f
Dhrystone	0.78	1.13	3.07	1.45
CoreMark	0.77	1.02	2.40	1.32
Bochs	0.37	0.88	2.54	2.37
Our simulator (s86sim)	0.17	0.58	1.95	3.32
SPECint2000 (geomean)	0.24	0.66	1.49	2.73
MiBench (geomean)	0.47	1.06	2.34	2.25
Stanford (geomean)	0.54	1.00	1.81	1.83

Table 13.4: Per-clock performance (in x86 instructions per clock) for Nios II/f, our design, and Intel Haswell, and speedup of this work compared to the Nios II/f. This is the same data as in Figure 13.3.

Nios II, this metric accurately shows the relative per-clock performance, but in units of (x86 instructions per Nios II clock cycle), and is not the Nios II IPC. The raw Nios II IPC tends to be higher because the same workload usually requires more instructions on Nios II.

These measurements show that we have achieved large gains in per-clock performance over the Nios II/f of around two times, somewhat higher for the memory-intensive SPECint2000 benchmarks and less for non-memory-intensive benchmarks such as Dhrystone and CoreMark. However, it also shows that there is more potential performance improvements for even more aggressive designs: Haswell achieves 2 to 3 times higher per-clock performance with a design that is at least two times greater than our design in essentially every high-level design parameter (issue width, execution width, number of ALUs, etc.) This figure also shows that the memory-intensive SPECint2000 benchmarks tend to achieve lower per-clock performance than benchmarks with smaller memory working sets, especially on the Nios II/f.

We believe that the overall per-clock performance achieved on our processor is reasonable. Although our processor was designed for a peak IPC of 2, single-threaded processors rarely sustain an IPC close to its peak. Our use of just one general-purpose (simple) ALU means that sustained IPC would unlikely reach the peak IPC even for "nice" workloads with no cache misses. Despite the ALU limit, our processor exceeds 1 IPC on many benchmarks, and averages above 1 IPC on benchmarks with small memory working sets: Dhrystone, CoreMark, and the MiBench and Stanford benchmark suites. For comparison, the Intel Haswell achieves a sustained IPC of about half of its peak throughput of 4 IPC, and the Nios II/f achieves about half of its peak throughput of 1 IPC.

We ran the same set of workloads on more processors to see how our processor compares to various x86 processors over the last 20 years. Figure 13.4 shows this data, normalized to our processor, and summarized into five groups of benchmarks (Each group is the geometric mean of its components). Higher numbers are slower. The chart is sorted by each processor's per-clock performance. Note that the memory latency (in clock cycles) differs greatly between systems, so truly fair microarchitecture comparisons can only be made between systems with similar memory latency in clock cycles (which are usually those with similar clock speeds). This figure shows that on our workloads, our processor design is slightly faster per clock than the Pentium Pro (3-way x86, 1995) and slightly slower than the Atom Silvermont (2-way x86, 2013).

Interestingly (but not too surprisingly), the only in-order processors are the four slowest processors in this list, with the two-issue processors (Bonnell and Pentium) being faster than single-issue processors (Nios II/f and VIA C3).

Included in this comparison are three processors that are intended to be used on an FPGA: Our soft processor, the Nios II/f soft processor, and the ARM Cortex-A9 found as a hard processor on some current-generation FPGAs. Our processor achieves higher per-clock performance than the two-issue out-of-order Cortex-A9, but the A9 has 4 to 5 times higher clock speed due to being a hard processor.

Compared to the Nios II/f (and many other processors), our design performs relatively poorly on Dhrystone and CoreMark, possibly due to the lack of (simple arithmetic) ALUs. Conversely, the RISC-like architectures (Nios II/f and ARM Cortex-A9) seem to do unusually well on these two benchmarks.

13.3.1 Sensitivity to Memory Latency

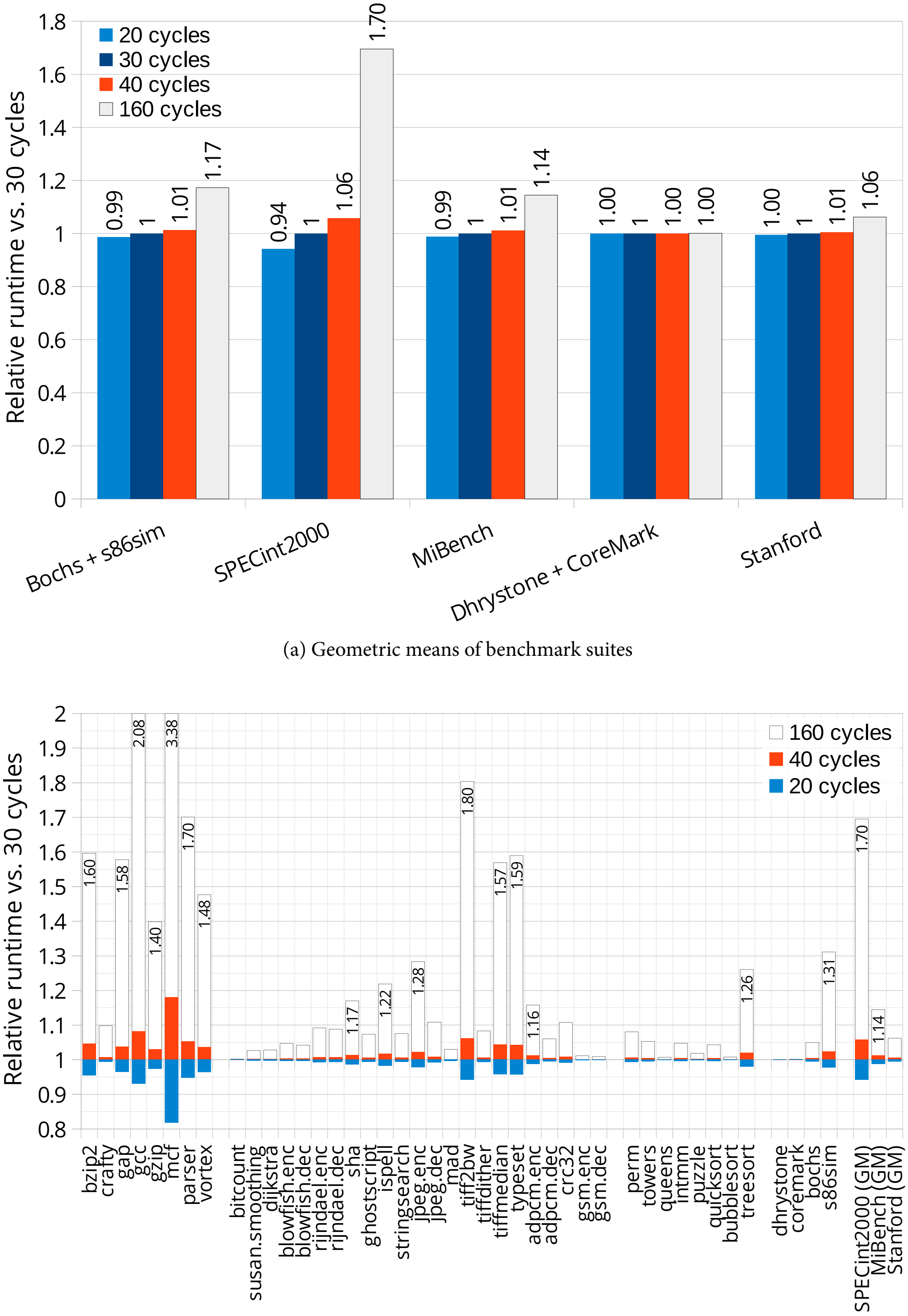
The results above for our processor assumes a memory latency of 30 cycles (or 150 ns at 200 MHz). This provided a reasonably fair comparison with the Nios II/f we tested. In this section, we measure how performance changes if the assumed memory latency changes. Figure 13.5 shows the sensitivity of performance to memory latency, with Figure 13.5a showing per-benchmark suite geometric means, and Figure 13.5b showing per-benchmark results. We simulated memory latency of 20, 30, and 40, and 160 cycles. While 20–40 cycles may be a reasonable range of memory latencies for an FPGA soft processor, we included 160 cycles to model what might happen if we were able to use our microarchitecture unmodified as a high-performance processor, running at 4 GHz with 40 ns memory latency (or 160 cycles).

For moderate changes in memory latency (20–40 cycles), caches work well and performance changes are mostly small. Unsurprisingly, benchmarks with larger memory footprints (such as SPECint2000) are more sensitive to memory latency, while those that fit completely in the cache are not. Compared to 30 cycles, for 20 and 40 cycles of memory latency (±33%), SPECint2000 performance increases and decreases by 6%, respectively, with `mcf` being particularly sensitive (±18%).

At 160 cycles of latency, SPECint2000 runtime increases by 70% (1.7×), with `mcf` (with its high cache and TLB miss rates) taking 3.4 times as long to run. The MiBench geomean of 1.14 hides the fact that several of its component benchmarks have large (>57%) slowdowns due to the high 160-cycle memory latency, hidden by many of the other components that have small enough working sets to be minimally affected by the high memory latency. For small benchmarks, both Dhrystone and CoreMark fit entirely in the cache and are 0.1% slower at 160 cycles of memory latency. If our processor were used unmodified in a system with 160-cycle memory latency, the decrease in performance would move the processor down two places in Figure 13.4, ranked slightly faster than the Pentium 4.

13.4 Area, Frequency, and Wall-clock Performance

This section combines the per-clock performance from the previous section with area and clock frequency measurements. We compare our processor with other processors to evaluate the final wall-clock performance. Because both area and frequency are heavily dependent on the implementation technology, comparisons are meaningful only when the implementation technology is similar. In our case, this



(a) Geometric means of benchmark suites

(b) Per-benchmark

Figure 13.5: Performance sensitivity to memory latency, relative to 30 cycles

Component	Area (Equiv. ALM)	Estimate[1] (Equiv. ALM)	Frequency (MHz)	Chapter	Status
Decode	4300	5000	247	6	Partial
Branch prediction	—	1000	—	6	No circuit
Register Renaming	1900	1900	317	7	Complete
ROB + Commit	—	2000	—	8	No circuit
Scheduler	3800	4000	275	9	Complete
Register file	2400	2400	260	10	Complete
Execution	2300	2300	240	10	Complete
Memory system			200	12	Complete
Logic	9000	9000			
L1/L2 Cache memory	5000	5000			
Microcode	—	500	—	5.3.4	No circuit
Total		28 700	200		

[1] We estimated the area required for components that are not yet implemented, and the final area for components that are only partially implemented.

Table 13.5: Area and frequency estimates for completed processor

Component	Logic Area (ALM)	RAM (M9K)	 (M144K)	Total Area (Equiv. ALM)	Frequency (MHz)
Nios II/f[1]	2100	76		4400	
DDR3 SDRAM controller	5100	19	2	6200	
Two timers	180	—	—	180	
1 KB SRAM[2]	2	2	—	60	
UART	90	2	—	150	
Qsys interconnect	2200	8	—	2400	
Total	9600	107	2	13000	245

[1] Nios II/f configured with 32 KB instruction and data caches, MMU with 6-entry DTLB, 4-entry ITLB, 128-entry L2 TLB, radix-2 divider, 32-bit multiplier, 256-entry dynamic branch predictor, and no JTAG debug.
[2] 1 KB "tightly-coupled" on-chip RAM holds the interrupt vector and TLB miss handler code.

Table 13.6: Area and frequency of a minimal Nios II/f system with 32 KB caches and MMU

means comparisons between processors implemented on similar FPGAs.

13.4.1 Area

Table 13.5 summarizes the area consumption of the various hardware blocks in our processor. The second column (Area) lists the ALM usage of each component as presented in earlier chapters. We converted block RAM area into equivalent ALM area using scaling ratios from Table 3.2. The only components that use block RAM are the L1 and L2 cache memories. For hardware blocks that are not yet implemented or incompletely implemented, we estimated the resource usage for the component so that we can compute an approximate resource usage for the complete processor. We estimate that our entire processor, including caches, will use about 28 700 equivalent Stratix IV ALMs, with about 17% of that total being used by block RAMs for the L1 and L2 caches.

For comparison with a Nios II/f, we synthesized a minimal Nios II/f Gen2 system for the same Stratix

IV FPGA. We configured the Nios II/f with the same parameters as those used for the per-clock measurements in the previous section (32 KB instruction and data caches). We omitted optional features that do not affect performance such as the JTAG debugging interface. The area and cycle time will vary depending on the configuration of the processor and system.

The FPGA resource usage of our (much more complex) processor is, as expected, much greater than the resource usage of a performance-tuned Nios II/f processor, which is around 4400 equivalent ALMs. When comparing the processor alone, our processor design provides around 1.4–2.8 times higher per-clock performance at a cost of about 6.5 times resource usage. With FPGA capacities still growing exponentially, this is often a reasonable trade-off if performance is important.

In practice, processors are always used as a component of a larger system, consisting at minimum of a memory controller and a few I/O devices. Because the Nios II/f is so small, other hardware in the system consume more resources than the processor even in a simple system consisting only of a memory controller and enough devices to boot Linux without doing anything else particularly useful. Table 13.6 shows the resource usage of such a small computer system. The Nios II/f only comprises about one third of the system, with the DDR3 controller and Qsys interconnect making up most of the other two-thirds. Replacing the Nios II/f processor with one that is 6.5 times larger (from 4400 to 28 700 ALMs) would only cause the resource usage of this minimal computer system to grow by 2.9 times.

We also attempt to make some approximate area comparisons to three x86 processors that have been synthesized for FPGA and presented in prior work [21, 42, 43]. These comparisons are shown in Table 13.7. It is difficult to make fair comparisons with the three Intel processors. The Intel processors are synthesized from RTL code that was not originally intended for implementation on an FPGA. There are also differences in which processor components are included in the FPGA design: the FPGA synthesized versions of the Atom and Nehalem both omitted the L2 cache that exists in the original (hard-processor) design, while our processor does include a 256 KB L2 cache. They also used different FPGA architectures. Nonetheless, it is clear that our design (28.5K Stratix IV ALMs) uses no more FPGA resources than the FPGA-synthesized (2-way in-order) Pentium (66K Virtex-4 4-LUTs), while having a higher-performing out-of-order microarchitecture that also achieves much higher frequency. We believe that much of the lower area and higher frequency of our design is due to the difference between designing a processor's microarchitecture and circuits for an FPGA compared to taking a non-FPGA design and modifying it to be synthesizable.

13.4.2 Frequency

In Table 13.5, we also list the achieved clock frequency of each component we implemented. Our estimate of the clock speed for the complete processor is the minimum of the per-component clock speed estimates. Although combining multiple hardware blocks into a larger circuit usually results in a circuit slower than the slowest component, we also anticipate that some clock frequency optimizations would be done on the final design, which may compensate for the frequency loss. We estimate that our processor will achieve around 200 MHz, limited by the slowest component, the memory system. As most of the other components are significantly faster (the next slowest unit is 240 MHz), this suggests that extra pipelining of the memory system may be worthwhile.

For comparison, the Nios II/f system we synthesized runs at 245 MHz (See Table 13.6). To achieve this frequency, we had to increase the pipeline depth of the Qsys interconnect (which would add memory latency) so that the interconnect would not be on the critical path. Although our processor design has a lower frequency than the Nios II/f, the increase in per-clock performance of about two times is far greater than the frequency loss, resulting in higher wall-clock performance.

In Table 13.7, we also listed the frequency of the three FPGA-synthesized Intel processors. However, the frequency comparisons are not meaningful. Although their processors were modified to better suit the FPGA, the processors were intended to demonstrate an accurate emulation of a hard processor rather than maximizing the performance of an FPGA implementation. This is especially true for the Nehalem, where time-domain multiplexing of signals crossing between FPGAs in their multi-FPGA implementation causes very low 520 kHz operation.

13.4.3 Wall-Clock Performance

The wall-clock performance of a processor is the product of its per-clock performance (or IPC if the instruction set is fixed) and clock frequency. The rightmost two columns of Table 13.7 incorporate the relative performance-per-clock for SPECint2000 from Figure 13.4 for each processor then multiplies it by the processor's clock speed, and normalizes that to our processor at 200 MHz.

Despite the clock speed disadvantage compared to the Nios II/f, our design has higher wall-clock performance on SPECint2000 of about 2.2 times. Our processor, somewhat unexpectedly, has higher per-clock performance than the two-issue out-of-order ARM Cortex-A9 found in the hard processor system in the Cyclone V SoC FPGA. However, hard processors have a much higher clock speed: At four times the clock frequency, the Cortex-A9 is 2.8 times faster than our processor.

For the three Intel processors, the table lists the per-clock performance as measured on a hard processor system running our benchmarks. These results would be different than if those processors were implemented on an FPGA, as their FPGA implementations omit the L2 and L3 caches and the memory latency (in clock cycles) would be very different due to the lower clock speed on an FPGA. We combined these per-clock measurements of the hard-processor implementation with area and frequency measurements of the FPGA implementations. As with the above, the results for the three FPGA-synthesized Intel processors are not as accurate nor meaningful.

In Table 13.7 we also listed the performance of the hard processors we tested, as an informal comparison between FPGA soft processors and hard processors including the differences in the substrate.

13.5 Conclusions

In this chapter, we evaluated the performance and area of our processor design, and compared this to an FPGA soft processor (Nios II/f), a hard processor found on an FPGA (ARM Cortex-A9), and various x86 hard processors over the last 20 years.

Processor	Substrate	Area	Freq.	Relative	Relative
			(MHz)	perf/clk	perf
FPGA Soft Processors					
This work	Stratix IV (40 nm)	28 500 ALM	200	1	1
Nios II/f	Stratix IV (40 nm)	4400 ALM	245	0.37	0.45
Intel Nehalem [21][1]	Virtex-5 (65 nm)	670K 6-LUT	0.52	1.79	0.005
Intel Atom Bonnell [42][2]	Virtex-5 (65 nm)	176K 6-LUT	50	0.61	0.15
Intel Pentium [43]	Virtex-4 (90 nm)	66K 4-LUT	25	0.49	0.06
Hard Processors					
Pentium	350 nm		200	0.49	0.5
AMD K6	350 nm		166	0.69	0.6
Pentium Pro	350 nm		200	0.80	0.8
VIA C3	150 nm		550	0.37	1.0
ARM A9	28 nm		800	0.70	2.8
Atom Bonnell	45 nm		1600	0.61	4.9
VIA Nano	65 nm		1000	1.15	5.8
Pentium 4	130 nm		2800	0.63	8.9
Atom Silvermont	22 nm		2417	1.01	12.2
AMD Piledriver	32 nm		3500	1.46	25.6
Nehalem	45 nm		3200	1.79	28.6
Haswell	22 nm		4300	2.26	48.7

[1] Excludes 256 KB L2 cache

[2] Excludes 512 KB L2 cache

Table 13.7: Area, frequency, and SPECint2000 performance comparison

We evaluated each processor's per-clock performance using a collection of workloads. Our processor's expected performance per clock is about twice that of the Nios II/f soft processor, with higher gains for memory-intensive workloads such as SPECint2000 (2.7 times). Compared to other x86 processors, our design performs reasonably well, with per-clock performance (IPC) between the out-of-order Pentium Pro and out-of-order Atom Silvermont. Per-clock performance is not highly sensitive to memory latency: Varying the memory latency by 33% resulted in 6% change in the memory-intensive SPECint2000 (at most 18% for mcf), but increasing latency by 5 times would lower our processor's IPC to be similar to the Pentium 4.

Our processor is expected to run at 200 MHz, limited by the memory system, which could potentially be further pipelined for higher frequency (The next slowest unit is 240 MHz). This is not much slower than the Nios II/f's 245 MHz. Combining clock frequency with the per-clock performance gains results in a net wall-clock performance increase of 2.2 times on SPECint2000 vs. the Nios II/f.

Our processor is expected to consume 28 700 equivalent ALMs (including block RAM and DSP blocks), which is 6.5 times the size of a performance-tuned Nios II/f (4400 ALMs). It is smaller than Intel x86 processors that were synthesized for (but not designed for) FPGAs. As FPGA capacities grow to millions of LUTs (e.g., 1.8M in the largest Stratix 10) and as current soft processors only consume a small part of even a minimal soft computer system, this area increase is likely acceptable in many applications where higher soft processor performance is useful.

Chapter 14

Conclusions

As FPGAs continue to get larger and less expensive, it becomes increasingly affordable and desirable to build higher-performance soft processors, including out-of-order processors. As we mentioned in the introduction (Chapter 1), there is a need for increased single-threaded soft processor performance. We have shown in this thesis the design of an out-of-order soft processor that achieves double the single-threaded (wall-clock) performance of a performance-tuned Nios II/f (2.2× on SPECint2000) at a cost of 6.5 times the area of the same processor. This area is about 1.5% of the largest Altera (Stratix 10) FPGA.

We presented a methodology for simulating and verifying the microarchitecture of our processor, which we used to design a microarchitecture that is sufficiently complete and correct to boot most unmodified 32-bit x86 operating systems. We showed that the FPGA substrate differences from custom CMOS do affect processor microarchitecture design choices, such as our use of a physical register file organization, low-associativity caches and TLBs, and a relatively large TLB. The resulting microarchitecture did not require major microarchitectural compromises to fit an FPGA substrate, and remains a fairly conventional design. As a result, the per-clock performance of our microarchitecture compares favourably to commercial x86 processors of similar design. Our two-issue design has slightly higher per-clock performance than the three-issue out-of-order Pentium Pro (1995) and slightly less than the newer two-issue out-of-order Atom Silvermont (2013).

While we designed the microarchitecture and circuits for just one specific processor microarchitecture, the processor building blocks can be used to build other soft-processor designs. This includes simpler designs by omitting features, more aggressive designs by scaling up our circuit designs, or even other (usually simpler) instruction sets. In this processor design, we have tried to be aggressive in our design choices to show that even aggressive microarchitectures are feasible on an FPGA. These choices include the use of the complex x86 instruction set, a speculative out-of-order memory system, and single-cycle ALU execution latency. Creating new microarchitectures by simplifying features from an aggressive design is easier than designing new features on top of a simpler design.

We have shown an out-of-order x86 soft processor microarchitecture that improves performance compared to current in-order pipelined soft processors, and developed processor circuits that can be used in future microarchitectures. For soft-processor systems where performance is important, the availability of a faster single-threaded soft processor will allow an easy path to increasing performance, without rewriting existing software or designing new hardware accelerators, at a moderate area cost that continues to become increasingly affordable.

14.1 Summary of Lessons Learned

This section summarizes some of the lessons learned regarding the design methodology, microarchitecture design, and circuit design of our soft processor.

14.1.1 Verification Methodology

When aiming to build a processor, correctness verification requirements are higher than for processor performance studies. For processor performance studies, a cycle-level simulator where the functional and timing components are decoupled is often used, as it allows sufficiently-accurate performance measurements without needing to worry about correctly modelling every corner case. When correctness is required, such a simulator would hide subtle design errors. Instead, we use an execute-in-execute approach where the behaviours of instructions are computed by the detailed pipeline model. This allows an error in the detailed design (or timing) to be visible as a functional error that could be detected and fixed, was essential in verifying the correctness of the hardware design (see Section 4.1.2).

Verification of the behaviour of the processor was done by comparing the cycle-level simulator to an instruction-set level (functional-only) simulator. When using a cycle-level simulator where timing can affect the functionality of the simulated instructions (due mainly to asynchronous events such as interrupts), it was important to track simulation time in a way that is equivalent between both the instruction-set level simulator and the cycle-level simulator so behaviour between the two simulators can be compared. We tracked simulation time for the x86 system in units of committed instruction count, which progresses identically for both simulators, and track real clock cycles separately for performance measurements (see Section 4.4.2).

14.1.2 Microarchitecture

Chapter 3 compared various "building-block" circuits when implemented on FPGA vs. custom CMOS, and found that circuits such as adders and block RAM that have dedicated hardware support on FPGAs are particularly area-efficient, but that different circuit types had similar delay ratios (FPGA delay vs. custom CMOS delay) when comparing the same circuit implemented on FPGA vs. custom CMOS. Thus, we concluded that soft processors do not need highly unconventional microarchitectures in order to be feasible when implemented on an FPGA.

However, some design decisions are affected by the use of an FPGA, such as strongly preferring the use of low-associativity caches and a physical register file (PRF) organization (as discussed in Section 3.5). Using a PRF organization in combination with register renaming (see Chapter 7) and separating execution units into clusters (see Section 5.1) further benefits an FPGA implementation by allowing all register files to need only one write port.

The processor front-end is a fairly conventional x86 fetch and decode-into-micro-ops design. We chose to use fairly complex (load-operate-store) micro-ops, as we believe it would reduce multiplexing (at equivalent performance) compared to supporting a higher throughput of simpler micro-ops (see Section 5.2).

After decoding, each micro-op's register references are renamed. We found that using multiple register renamer checkpoints was unnecessary: Recovering from a pipeline flush by copying the committed register renamer state to the speculative register renamer was sufficient (see Chapter 7).

After renaming, micro-ops are inserted into instruction schedulers to be scheduled for execution. For our multiple-issue (4-way) scheduler, we chose to use a distributed design, where each execution cluster uses its own scheduler. Distributed schedulers have lower circuit delay, but require more total scheduler entries to achieve the same instructions-per-cycle as unified schedulers (see Section 9.7.1). We compared several scheduler circuit implementation styles, and found that while matrix schedulers are faster than CAM-based schedulers at a small number of entries, their area and delay increase more quickly than CAM-based schedulers at large sizes (see Section 9.6). We also found that the ability to instantaneously clear the scheduler at a pipeline flush was important for keeping the pipeline flush latency low (see Section 8.3.2).

Most of the difficulty of the design of the execution units was in the circuit design, to fit the required functionality in the fewest number of pipeline stages. The extra delay caused by computing condition codes has been a particular challenge, with new circuit designs to reduce this delay (see Section 10.2.3).

The requirement to support a modern operating system creates new requirements, including support for serializing operations, precise exceptions and atomic commit of instructions, and, for x86, floating-point support (see Section 5.3). However, the most complex changes to support an operating system were in the memory system to support paging.

For memory system performance, our processor has caches (and TLBs), and supports load-to-store forwarding, multiple outstanding cache misses, and memory dependence speculation. All of these features interact, creating a very complex design (Chapter 12). For circuit speed, we used a virtually-indexed, physically tagged cache design, to allow the TLB lookup to proceed in parallel with the cache access (Section 12.8). We used a large (128-entry) but low-associativity (2-way) TLB to better suit the FPGA substrate: Comparators and multiplexers required for associativity are expensive, while RAM capacity is relatively cheap, so increasing performance by increasing TLB capacity is a better option than increasing TLB associativity (Section 12.5.5). Due to the complexity of the L1 data access hardware, we chose to attach the page table walk unit to the L2 cache rather than to the L1 cache. Thus, an L2 cache is important for caching page tables, to speed up page table walks (Section 13.2.2).

14.1.3 Circuits

Throughout the entire processor design, we made frequent use of circuits manually designed at the LUT level, as we found large performance improvements compared to synthesis from RTL (e.g., 54% faster barrel shifter in Section 10.2.3 and 14% faster L1 cache access in Section 12.8.3). We found that merely structuring the HDL code to make the desired LUT structure obvious was not sufficient, as the synthesis algorithm tends to aggressively extract out common shared sub-circuits (presumably to save area) even at the expense of logic depth. Thus, we made frequent use of `keep` directives and `lcell` (logic cell) primitives [182] to force the synthesis tool create the desired circuit.

The result of the microarchitecture and circuit designs is a processor that achieves IPC comparable to similar Intel processors (Pentium Pro and Atom Silvermont), with an expected frequency of about 18% less than a much simpler Nios II/f soft processor on the same FPGA.

14.2 Contributions

This thesis presents the design of a superscalar out-of-order x86 FPGA soft processor. In this thesis, we make the following contributions:

- Study how the FPGA substrate affects circuits and out-of-order processor microarchitecture (Chapter 3)
- Design a microarchitecture that correctly implements the x86 instruction set sufficiently well to boot modern operating systems (Chapters 4 and 5)
- Design FPGA circuits for key processor components that implement the out-of-order microarchitecture (Chapters 6 to 12)
- Evaluate the performance and costs of an out-of-order soft processor compared to a commercial in-order soft processor and x86 hard processors (Chapter 13)

14.3 Future Work

In any processor design this complex, there are design decisions that turn out problematic in hindsight, design choices that were made for simplicity rather than optimality, and potential improvements not explored. We list some of them here, hoping they will be revisited in the future:

Microcode The current microcode strategy relies on the existing out-of-order mechanisms to handle exceptions within microcode sequences. While conceptually simple, this design makes long microcode sequences behave as atomic instructions, which imposes a minimum size to various queues in the processor (load queue, store queue, and reorder buffer). This minimum-size requirement prevents scaling down the processor to a less aggressive design. The microcode should be rewritten to not require atomic commit of the entire sequence, by checking for all potential exception conditions before changing any system state.

Memory system The memory system's frequency is 17% slower than the next slowest unit. Thus, its pipeline length should be increased. However, this is likely non-trivial, as nearly every pipeline stage is currently critical, so adding one pipeline stage will likely require a large amount of circuit-level redesign.

Future work should also implement the NoSQ memory disambiguation scheme, which we abandoned due to risk and effort.

Partial register and flag writes We chose the least-complex method of handling partial register and flag writes by serializing all operation that perform a partial write. Although the overall performance impact is small, we noticed some workloads are significantly affected, especially when the code was compiled targeting newer processors. Recent Intel and AMD processors can handle partial register and flag writes with little performance penalty, so compilers seem more willing to generate code with partial writes.

CAM-based instruction scheduling Although we determined that CAM-based schedulers are slower than matrix schedulers at smaller scheduler capacities, matrix schedulers are difficult to use (requiring preprocessing to map register dependencies into instruction dependencies) and limit flexibility (e.g., execution latency that depends on operand type). Matrix schedulers also scale poorly to larger sizes (especially in area). Thus, CAM-based instruction scheduling deserves further exploration.

Floating-point unit While our current design uses FPU emulation to provide the functionality of a x87 floating-point unit, the performance overhead is so high that it is impractical for any workload that uses on the order of 1% or more of floating-point instructions. The emulation mechanism (described in Appendix A) also complicates the design of the integer portion of the processor. A hardware floating-point unit is essential.

Applicability to other FPGA architectures Many of our circuits make use of the 7-input mode of the Stratix IV Adaptive Logic Module (ALM). Not all FPGAs allow 7-input functions. Notably, Xilinx FPGAs currently use 6-input LUTs. We expect that the overall circuit performance will only be moderately slower if 7-input functions are not available, but some of the circuit structures may need significant redesign.

General performance optimizations Due to design effort considerations, there are many places where our microarchitecture handles certain operations sub-optimally. This list of potential performance improvements never ends, however.

Create new microarchitectures In addition to optimizing the existing design, the circuits developed in this thesis can be extended or simplified to create new soft processor microarchitectures, of both more and less aggressive designs. New microarchitectures will enable future processors to span more of the performance-cost trade-off space.

Appendix A

Floating-Point Emulation Mechanism

To reduce design effort, we decided to omit the x87 floating-point unit (FPU) from our processor. However, since all x86 processors since the Pentium (1993) have contained a floating-point unit, almost no modern operating system (OS) will run without the x87 FPU being present. Thus, we must add a mechanism to emulate (in software) the x87 FPU, and cannot simply remove the hardware FPU without replacing its functionality.

Traditionally, emulating non-existent instructions used by user programs would be done by the operating system by intercepting the undefined instruction exception and emulating the behaviour in the exception handler. In our case, we need an emulation mechanism that is invisible to the OS. Hiding a trap-and-emulate mechanism from the OS is surprisingly complicated: The emulation mechanism must work in *any* processor mode (16-bit, 32-bit, real, protected, paged or non-paged, etc.) and not disturb any memory (e.g., stack) used by the OS. This appendix describes the mechanism we added to our processor to perform trap-and-emulate of the x87 floating-point instruction set.

There is an existing x86 operating mode called System Management Mode (SMM) that allows interrupting the processor to run firmware code invisibly from the operating system. However, it is unsuitable for instruction emulation as it lacks the ability to generate exceptions and the ability to access memory in the context of the OS or the user-mode program. Therefore, we created a new mechanism based loosely on the existing interrupt mechanism. The new mechanism puts the processor into a new execution mode ("emulation mode"), executes the floating-point emulation code in this mode, then returns back to normal execution. We did not consider the interaction between SMM and our emulation mode as we did not implement SMM.

An interrupt mechanism needs to save the current state of the processor, then transfer control to the exception handler. The x86 interrupt mechanism first switches stacks (so each privilege level has its own stack), then pushes some processor state onto the new stack and branches to the exception handler. We use the same mechanism for floating-point emulation, but it differs in the location of the stack, the state that is pushed, and the handler address. x86 interrupts use the `iret` instruction to return from an interrupt, and we use the same instruction (with slightly modified behaviour) to return from floating-point emulation.

In addition to entering and leaving emulation mode, the floating-point emulation mechanism also needs to emulate memory accesses (for floating-point load and store instructions) including any memory faults this may produce, and emulate floating-point exceptions.

After all of this complexity, we finally reach the floating-point emulation routine itself. Our implementation is derived from the floating-point emulation code used inside Bochs [115], which is itself based on the SoftFloat library from Berkeley [183]. The emulation code is written in C, with a small x86 assembly wrapper to interface with the emulation mode entry and exit mechanism. Due to design effort, we focused only on correctness and spent no effort to optimize the execution speed of our floating-point emulation routine.

Hardware support for emulation mode is fairly localized, requiring modified decoding for the floating-point instructions and a few others involved in manipulating the FPU control word and status word, and the ability for the (instruction and data) memory system to bypass paging and segmentation. However, there are still some outstanding issues with correctness (which we did not encounter in practice) when interacting with corner-case behaviours of the memory system, especially on multiprocessor systems.

The rest of this appendix will describe in more detail the mechanism used to enter and exit emulation mode, throw exceptions, and access memory.

A.1 Emulation Mode

Emulation mode is a new execution mode used only for executing floating-point emulation routines. It is nearly equivalent to 32-bit real mode, where all instruction fetches and memory accesses bypass both segmentation and paging and have no permission restrictions. We chose to use a new bit in the `EFLAGS` register to indicate emulation mode, but in retrospect, this was not necessary or useful, although not particularly harmful either, other than defining a previously undefined flag bit.

Emulation mode requires both code and data memory space. We placed the emulation code as part of the BIOS firmware code. We allocated 3 KB of data space (`0x9f000` to `0x9fc00`) immediately below the EBDA (extended BIOS data area, at `0x9fc00` to `0xa0000`) to minimize disruption to existing OSes. We modified the BIOS to report a larger memory hole at the EBDA location to reserve the region so OSes will avoid using this memory range. This involves modifying the word at `0x413` in the BIOS data area that changes the result of BIOS service INT 12h (get conventional memory size), and also ensuring that the INT 15h function `e820` handler code returns the correct system memory address map to the OS.

This data area is used both for static data (the emulated FPU registers and other state) and for the stack used by the emulation code.

Within emulation mode, the emulation code is responsible for interpreting the opcode, emulating the behaviour of the instruction (by manipulating the emulated FPU state), then setting the (hardware) FPU control word and status word registers to reflect the result of the floating-point instruction. (FPUCW and FPUSW are used for exception handling. See Section A.5.1.) The emulation routine then restores the previous processor state, and returns from emulation mode (using `IRET`).

A.2 Entering emulation mode

Emulation mode is entered when a floating-point instruction is decoded. The instruction decoder treats the floating-point block of opcodes similarly to a software interrupt: It emits a sequence of micro-ops that perform the CPU mode change and state saving operations.

On entry into emulation mode, the stack pointer is changed to a hard-coded `0x9fa00`, and then the following 8 values are pushed onto the stack:

SS stack segment selector (before the stack switch)
ESP stack pointer (before the stack switch)
EFLAGS
CS code segment selector (before branching)
EIP instruction pointer (before branching)
EIP_next instruction pointer of next instruction (before branching)
Opcode and Memory operand segment
Memory operand linear address ← *Top of stack*

Of the first five elements, only EFLAGS, ESP, and EIP are useful. We used this stack frame because it is the same stack frame used by a privilege-changing protected-mode exception or interrupt. Future implementations may optimize this further, as it turned out there were no gains in simplicity for keeping the stack frame the same.

The last three elements reduce the complexity of the emulation routine by supplying partially-decoded instruction information. EIP_next is equal to EIP plus the length of the floating-point instruction. The memory operand segment and linear address provides the segment (DS, unless overridden by a segment-override prefix) and linear address of the memory operand for the instruction, if any.

The processor then flushes the pipeline and branches to the FPU emulation code entry point at `0xfffc0a00`.

A.3 Leaving Emulation Mode

The emulation mechanism uses the `IRET` (return from interrupt) instruction to leave emulation mode. `IRET` is modified to behave differently in emulation mode. Unlike the normal `IRET`, returning from emulation mode restores EFLAGS, EIP, and ESP, but *not* the code segment (CS) or stack segment (SS). It is incorrect to set any of the segment registers (e.g., CS or SS) in emulation mode because this causes the associated segment descriptor to be reloaded from memory into the segment descriptor cache. In x86, the segment descriptor cache is architecturally-visible, and it is legal to deliberately have the segment descriptor in the descriptor cache be different from the segment descriptor in memory, so an unexpected reload from memory can be incorrect. As a result our emulation routines do not change any of the segment registers or descriptor caches.

The emulation code is expected to have copied EIP_next to the EIP field in the stack frame so that execution is resumed after the emulated floating-point instruction, rather than restarting the instruction again.

A.4 Memory Accesses

There are three different kinds of memory accesses that need to be considered: Code fetches to run the emulation code, data accesses into the emulation routine's internal state (e.g., FPU state), and data accesses in the context of OS or user memory to perform floating-point load and store operations.

Normally, code fetches always occur through CS (code segment), applying any permission checks, segmentation, and paging as required. In emulation mode, we simply fetch from physical addresses instead, bypassing any permission checks, segmentation, and paging. However, for correctness, we cannot modify the CS selector nor the CS descriptor cache.

In x86, all data accesses must go through one of the segments (DS by default), applying any permission checks, segmentation, and paging. We change this behaviour in emulation mode. In emulation mode, all data accesses by default will bypass all permissions checks, segmentation, and paging. Again, none of the segment registers are modified. Bypassing segmentation and paging allows access to the emulation routine's data (at physical addresses `0x9f000 to 0x9fc00`).

Memory accesses that need to access OS or user memory is more complicated, as these must be subject to the usual permission checks, segmentation, and paging. We allow access to segmented/paged accesses by using an explicit segment override prefix, even for DS. In emulation mode, memory accesses bypass segmentation and paging by default unless overridden by a segment override prefix, which allows data accesses using any of the segments.

Memory accesses that are subject to segmentation and paging can cause exceptions, which will be discussed in the next section.

A.5 Exceptions

Floating-point instructions can cause exceptions. There are two types of exceptions that can occur: Those caused by floating-point arithmetic, and those caused by the memory access for those FPU instructions that access memory.

It is impossible to throw exceptions within emulation code. Exceptions are intended to be handled by the OS currently running. Because the emulation mechanism must be invisible to the OS, the system state when the exception occurs (current EIP) must point to the FPU instruction that faulted, rather than any code within the emulation routine. In addition, faulting within emulation code implies aborting the emulation routine before completion, and the possibility of re-running the emulation routine for the same instruction from the beginning when the OS attempts to retry the same instruction at some time in the future. Our solution to this problem is to ensure that any exceptions are caught before entering emulation code, ensuring emulation code can never fault, and disabling interrupts while in emulation mode.

A.5.1 FPU exceptions

In x86, FPU exceptions are triggered based on the values of the FPU control word and FPU status word. The x87 FPU is designed as a co-processor, so FPU exceptions are not triggered directly by the FPU. Instead, the FPU sets status codes in the FPU status word, and most FPU instructions will check the status word and throw an exception *before* its own execution if the *previous* FPU instruction needed to trigger an exception.

Because the FPU control word and status word interact with the exception handling mechanism, they must be implemented in hardware, and cannot be handled purely by software emulation.

A.5.2 Memory exceptions

Memory accesses that are subject to segmentation and paging can cause exceptions. As it is impossible to throw exceptions after entering emulation mode, these exceptions must be detected and handled before entering emulation mode. Thus, for FPU instructions with memory operands, the micro-op sequence when entering emulation mode generates load micro-ops immediately before entering emulation mode to test for and trigger any exceptions that would occur when accessing memory. If a memory exception were to occur, it would be handled prior to entering emulation mode, using the existing exception-handling mechanism.

This mechanism is technically incorrect, as duplicating a load is not permitted if the load address is in an uncacheable region of memory (i.e., memory-mapped I/O with load side effects). It is also incorrect in multiprocessor systems where page tables can be modified by another processor (or in less-sane scenarios, even DMA from I/O), as there is a race condition. A successful load when entering emulation mode does not guarantee that a load to the same address a short time later will necessarily succeed, if page tables can be modified in between. We did not encounter either issue in practice, in a uniprocessor system.

A.6 Conclusions

While omitting a hardware FPU and replacing it with a software emulation mechanism saved FPU design effort, we believe it made the CPU more complex than if the FPU actually existed. Thus, even if two orders of magnitude lower performance were not a concern, replacing hardware with emulation is likely not a good choice. There is considerable extra complexity in creating an OS-invisible emulation mechanism that will handle instruction emulation, memory accesses, and exceptions. Also, while our mechanism is currently sufficient for uniprocessor systems that do not attempt to perform floating-point operations on memory-mapped I/O ranges, future systems will most likely need to fix the corner cases mentioned in the previous section. We anticipate that modifying the processor design to fix (and verify) all of these interactions with the (already complex) memory system would be more effort than designing a FPU in hardware.

We note that although an *OS-invisible* emulation mechanism suffers from complexity and correctness issues, *OS-visible* emulation done by the OS itself is not affected because exceptions can occur normally while executing an OS's emulation code.

Appendix B

Integer Division Algorithm and Circuit

This appendix describes the basic operation of our radix-4 non-restoring division algorithm and the extensions that allow implementing signed and unsigned division of three operand sizes using the same circuit. Figure B.1 (originally from Figure 10.16) shows our divider circuit.

Our divider uses a radix-4 non-restoring algorithm. We will describe the algorithm by first describing the most basic radix-2 restoring division algorithm then extending it to radix-4 and non-restoring. In radix-2 restoring division, one quotient bit is generated each cycle. If the partial remainder is no less than the divisor, a `1` quotient bit is generated and the divisor is subtracted from the partial remainder. Otherwise, a `0` quotient bit is generated and the partial remainder is unchanged. The partial remainder is then left shifted left by 1. This process is repeated until the entire quotient is computed. This algorithm is long division done by repeated shift-and-subtract.

In radix-2 restoring division, the partial remainder is always between 0 (inclusive) and $2d$ (twice the divisor, exclusive): subtracting the divisor always leaves a non-negative partial remainder less than the divisor, so it is always smaller than twice the divisor after left-shifting by 1. Radix-2 **non-restoring** division allows the partial remainder to be negative (between $-2d$ inclusive and $2d$ exclusive), produces quotient bits `-1` or `+1` (instead of 0 or 1), and always adds or subtracts d at every step (instead of subtracting 0 or d). This trades an extra cycle to "correct" the final quotient and remainder (if the final remainder is of the wrong sign) for a faster circuit. A **radix-4** non-restoring divider produces *two* quotient bits per cycle (with value -3, -1, +1, or +3), keeps the partial remainder between $-4d$ (inclusive) and $+4d$ (exclusive), and always adds $-3d$, $-d$, $+d$, or $+3d$ each cycle. Using a higher radix reduces the number of cycles needed to compute the quotient, at the expense of a more complex circuit.

The core operation of a radix-4 divider at each cycle is to examine the value of the current partial remainder and determine inside which of the four possible intervals ($[-4d, -2d)$, $[-2d, 0)$, $[0, 2d)$, or $[2d, 4d)$) it is, then subtract a multiple of the divisor (out of four possibilities). Normally this would require three comparisons (with $-2d$, 0, and $+2d$) and four additions ($-3d$, $-d$, $+d$, or $+3d$). Comparison with 0 is trivial, by observing the sign bit of the partial remainder, leaving two non-trivial comparisons. Knowing the sign of the partial remainder can eliminate half of the additions and comparisons because it is immediately known whether the partial remainder lies in the negative two intervals or positive two intervals. Thus, our divider uses two adder/subtractors, and a single comparison of the (optionally in-

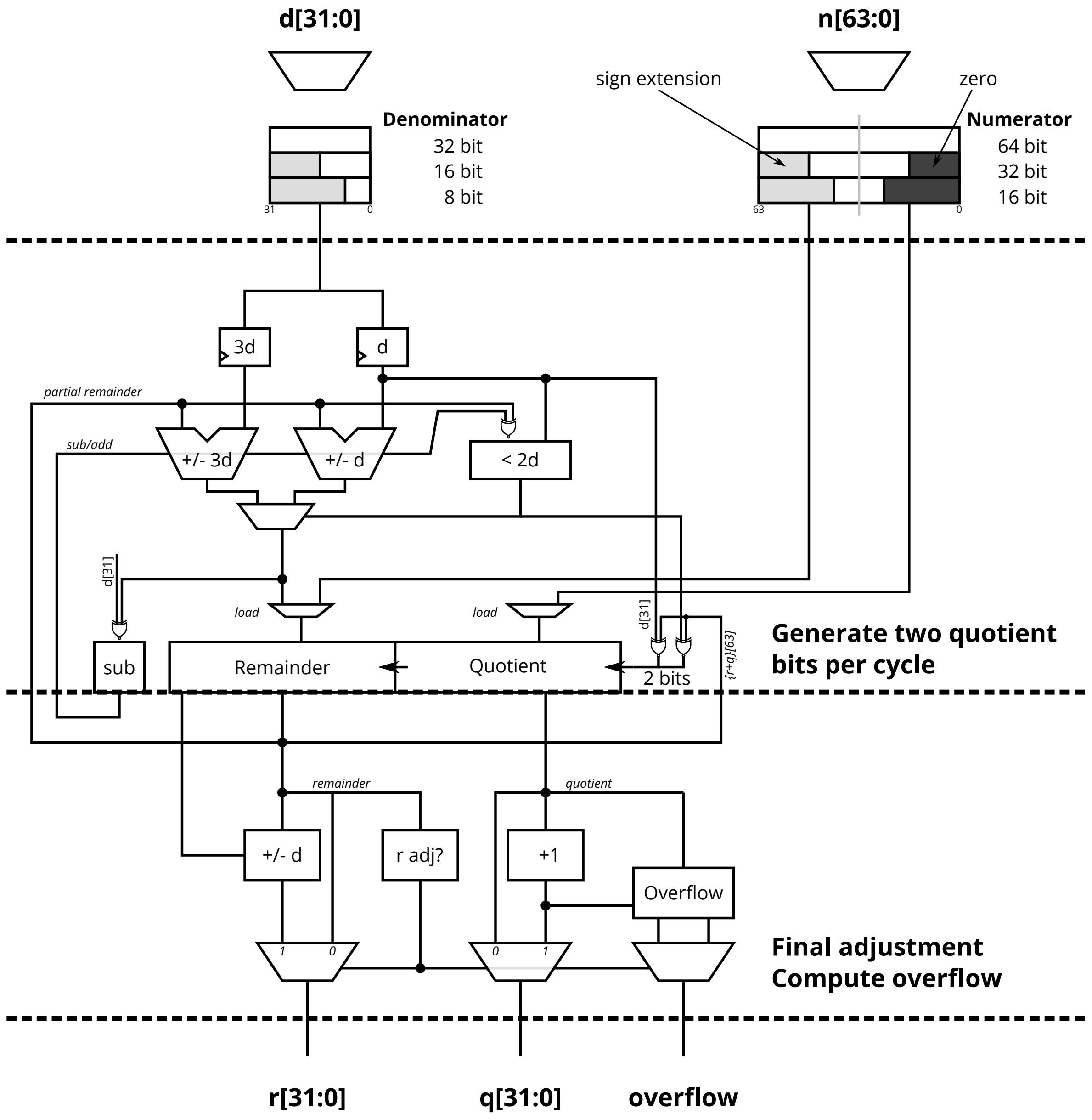


Figure B.1: Radix-4 non-restoring divider circuit. Handles 64/32, 32/16, and 16/8 signed and unsigned division. (Reproduced from Figure 10.16)

verted) partial remainder to $2d$ (We use inversion rather than negation because it gives the correct behaviour when the partial remainder is exactly $-2d$). The result of the comparison then selects which add/subtract output to use, which becomes the next partial remainder.

Our circuit is extended to allow the divider to process both signed and unsigned operands. For example, whether to subtract the divisor depends not only on the sign of the partial remainder, but also the sign of the divisor. The partial remainder is also extended by one bit to allow the full range of an unsigned remainder yet still be able to distinguish sign (It essentially treats both signed and unsigned 64-bit partial remainders as a signed partial remainder of 65 bits). Support for multiple operand sizes can be achieved by properly aligning the dividend and divisor values at the beginning of the division operation, then running the divider for a fewer number of cycles.

The final stage of the divider implements the adjustment that is required for non-restoring division. This compares whether the final remainder is out of range (remainder is non-zero and has a different sign than the dividend n, or if the remainder is exactly $-d$), and corrects the remainder and quotient. Much of the overflow detection computation also occurs in this stage, overflow detection of *signed* division is complex and depends on the final *corrected* quotient (Overflow detection of unsigned division is easier). To improve delay, the decision of whether to adjust the remainder and quotient occurs in parallel with overflow computation, which computes the overflow for both cases.

Appendix C

Test System Configuration Details

This appendix contains more details about the test systems used in Chapter 13, as well as benchmark results from more systems.

Table C.1 lists all of the systems we tested with, including CPUID family, model, and stepping for x86 processors.

Table C.2 lists the benchmark runtime for each workload on each processor we tested. The scale factor for each row is listed in the second column. For example, the number of clock cycles for running the bzip2 workload is in units of billions (10^9) of instructions. The last few rows of the table computes geometric means of each benchmark suite or category, and expresses the runtime relative to our design with 30 cycles of memory latency. The final row is the geometric mean of the five suites/categories. This metric implicitly assigns a weighting to each workload (e.g., a fairly high weight to Dhrystone, but low for each workload in the MiBench suite), which may not necessarily be appropriate in all cases. The rightmost four columns lists the same results for our design with varying main memory latency.

Figure C.1 shows the same results as Figure 13.4, but with additional systems. This is the same data as the five geometric means in Table C.2.

Figure C.2 shows a plot of random access memory latency vs. array size for each processor. We usually tested newer machines with huge pages (either 2 MB or 4 MB).

Processor	ISA	Issue	Data caches			Chipset	Memory	Frequency	Process	Year[1]
		width	L1	L2	L3			(MHz)		
Out-of-Order Processors										
This work	x86 (P6)	2	8	256		—	30 cycles	∼200	Stratix IV (40 nm)	—
ARM Cortex-A9	ARMv7	2	32	512		Cyclone V SoC	DDR3	800	28 nm	2007
Intel Pentium Pro (6-1-9)	x86 (P6)	3	8	256		440FX	EDO	233	350 nm	1995
Intel Pentium 4 (15-2-9)	x86	3	8	512		875P	DDR	2800	130 nm	2002
Intel Atom Silvermont J1900 (6-55-3)	x86-64	2	32	1024		—	DDR3	2420	22 nm	2013
Intel Core 2 Q9550 (6-23-10)	x86-64	4	32	6144		P965	DDR2	3400	45 nm	2007
Intel Nehalem i7-860 (6-30-5)	x86-64	4	32	256	8192	P55	DDR3	3200	45 nm	2008
Intel Sandy Bridge i5-2500K (6-42-7)	x86-64	4	32	256	6144	P965	DDR3	4200	32 nm	2011
Intel Ivy Bridge i5-3570K (6-58-9)	x86-64	4	32	256	6144	P965	DDR3	4200	22 nm	2012
Intel Haswell i7-4790K (6-60-3)	x86-64	4	32	256	8192	H81	DDR3	4300	22 nm	2013
AMD K6 (5-6-1)	x86 (P5)	2	32	1024		VIA MVP3	SDRAM	166	350 nm	1997
AMD K6-2+ (5-13-4)	x86 (P5)	2	32	128	1024	VIA MVP3	SDRAM	600	180 nm	1999
AMD K8 Opteron 2220 SE (15-65-2)	x86-64	3	64	1024		BCM5785	DDR2	2800	90 nm	2013
AMD Phenom 9550 (16-2-3)	x86-64	3	64	512	2048	780G	DDR2	2200	65 nm	2007
AMD Phenom II X6 1090T (16-10-0)	x86-64	3	64	512	6144	760G	DDR3	3400	45 nm	2009
AMD Bulldozer FX-8120 (21-1-2)	x86-64	4	16	2048	8192	760G	DDR3	3200	32 nm	2011
AMD Piledriver FX-8320 (21-2-0)	x86-64	4	16	2048	8192	760G	DDR3	3500	32 nm	2012
VIA Nano U3500 (6-15-10)	x86-64	3	64	1024		VIA VX900	DDR3	1000	65 nm	2009
In-Order Processors										
Nios II/f Gen2	Nios II	1	32			Cyclone V SoC	DDR3	100	Cyclone V (28 nm)	2008
Intel Pentium (5-2-12)	x86 (P5)	2	8	1024		VIA MVP3	SDRAM	200	350 nm	1993
Intel Pentium MMX (5-4-3)	x86 (P5)	2	16	1024		VIA MVP3	SDRAM	200	350 nm	1997
Intel Atom 330 Bonnell (6-28-2)	x86-64	2	24	512		945GC	DDR2	1600	45 nm	2008
VIA C3 (6-7-3)	x86 (P5)	1	64	64		VIA PLE133	SDRAM	550	150 nm	2001

[1] We list the year of introduction of the first implementation of the microarchitecture, rather than the release date of the specific implementation we tested.

Table C.1: Test systems

Microarchitecture		VIA C3	Nios II/f	Pentium	Pentium MMX	Atom (Bon-nell)	AMD K6	Pentium 4	AMD K6-2+	ARM Cortex-A9	Pentium Pro	**This work**	Atom (Silver-mont)	VIA Nano	AMD K8	Bulldozer	Piledriver	Phenom	Phenom II	Core 2	Nehalem	Sandy Bridge	Ivy Bridge	Haswell	This work 20–160 cycles memory			
Frequency		550	100	200	200	1600	166	2800	600	800	200	**~200**	2400	1000	2800	3200	3500	2200	3400	3400	3200	4200	4200	4300	20	**30**	40	160
SPECint2000 (Execution time in clock cycles)																												
bzip2	× 10^9	892	700	481	452	576	419	619	568	416	378	**314**	350	327	327	286	280	265	245	218	249	214	207	206	300	**314**	329	502
crafty	× 10^9	428	796	689	384	313	311	321	252	290	287	**236**	226	152	141	173	161	127	124	123	121	113	108	100	235	**236**	238	260
gap	× 10^9	573	605	535	384	334	368	320	387	264	263	**206**	198	159	200	146	139	154	148	124	123	110	104	88	199	**206**	213	325
gcc	× 10^9	833	557	646	558	340	368	273	531	232	387	**317**	156	146	184	120	115	115	113	101	92	79	76	67	295	**317**	343	659
gzip	× 10^9	830	902	638	582	485	540	443	508	564	317	**341**	321	294	277	358	336	243	241	259	257	243	234	227	332	**341**	351	476
mcf	× 10^9	960	509	382	359	608	357	685	1011	539	436	**211**	337	372	433	132	131	192	177	87	93	86	80	79	173	**211**	249	713
parser	× 10^9	1006	1060	690	594	632	555	629	626	655	471	**398**	416	358	370	332	293	344	310	242	255	214	206	199	377	**398**	419	676
vortex	× 10^9	738	1334	673	468	523	429	467	438	448	347	**289**	347	282	250	208	204	210	188	205	186	164	145	135	279	**289**	299	427
MiBench (Execution time in clock cycles)																												
bitcount	× 10^6	1035	634	604	591	521	563	453	461	477	453	**511**	434	423	391	376	338	379	375	272	269	233	205	190	511	**511**	511	512
susan.smoothing	× 10^6	1085	475	970	893	644	631	367	609	275	259	**345**	290	238	281	232	185	229	225	173	175	129	126	114	344	**345**	346	354
dijkstra	× 10^6	310	293	279	257	161	141	149	126	190	142	**161**	125	135	121	131	131	113	112	98	95	104	105	87	160	**161**	161	165
blowfish.enc	× 10^6	1506	1435	1062	931	992	922	732	893	1281	613	**737**	594	552	438	544	641	508	414	408	479	394	363	338	735	**737**	740	772
blowfish.dec	× 10^6	1513	1452	1057	932	999	928	727	899	1274	620	**738**	595	556	445	527	644	521	415	414	478	397	365	339	736	**738**	740	769
rijndael.enc	× 10^6	600	1107	556	389	423	316	219	280	313	197	**228**	236	198	192	165	153	174	155	140	143	118	112	104	226	**228**	229	249
rijndael.dec	× 10^6	602	792	598	374	407	324	214	275	311	197	**227**	238	194	187	164	152	165	147	138	137	117	111	106	226	**227**	229	247
sha	× 10^5	2058	1502	1208	1057	1107	1106	1380	1075	767	811	**1003**	908	719	673	888	801	698	656	577	567	608	585	576	990	**1003**	1016	1173
ghostscript	× 10^6	1810	2983	2113	1075	1118	1139	1058	1026	878	876	**1359**	800	712	659	666	552	558	537	516	493	403	381	329	1352	**1359**	1367	1459
ispell	× 10^6	2173	2634	1515	1283	1514	1265	1241	1327	1446	924	**998**	926	720	825	728	642	665	657	585	549	455	431	416	982	**998**	1015	1217
stringsearch	× 10^4	786	1042	1054	777	513	422	485	377	405	381	**350**	363	381	297	310	303	266	257	264	276	210	203	193	348	**350**	352	377
jpeg.enc	× 10^5	2121	1972	1563	1409	1137	1060	986	1058	1033	818	**991**	862	727	758	744	691	779	717	553	598	527	504	482	970	**991**	1012	1272
jpeg.dec	× 10^5	560	406	457	385	312	299	227	272	226	171	**195**	240	190	203	169	152	173	167	133	134	110	106	97	194	**195**	197	217
mad	× 10^6	1576	372	642	607	514	350	505	329	207	253	**256**	425	264	230	263	247	222	217	161	168	138	126	117	255	**256**	256	264
tiff2bw	× 10^6	812	999	940	870	381	492	317	487	255	244	**210**	142	145	159	156	146	161	127	119	93	76	74	70	198	**210**	223	379
tiffdither	× 10^6	1633	2792	1462	1381	1147	1037	994	982	842	1142	**794**	774	713	632	706	675	596	564	532	562	498	484	463	789	**794**	799	860
tiffmedian	× 10^6	1590	2832	1718	1626	852	991	750	947	767	688	**633**	570	518	556	497	444	543	453	368	350	303	290	257	607	**633**	661	994
typeset	× 10^6	1163	1584	1329	942	757	758	940	796	688	613	**568**	508	417	419	440	369	368	347	377	337	266	249	228	544	**568**	592	903
adpcm.enc	× 10^6	1455	1009	1072	918	1017	813	1131	816	617	997	**827**	877	801	771	851	818	876	730	685	679	638	627	630	817	**827**	837	957
adpcm.dec	× 10^6	1108	826	773	737	653	692	698	681	440	773	**566**	460	377	302	391	333	441	285	332	314	285	268	250	563	**566**	569	600
crc32	× 10^6	3203	3479	2449	2116	2271	2152	1557	2079	3011	1466	**1254**	1304	1238	906	1072	1312	1144	896	807	954	810	729	661	1244	**1254**	1264	1389
gsm.enc	× 10^6	3130	1655	2858	2823	1696	1372	1328	1274	984	737	**781**	794	713	565	666	712	561	546	526	534	421	414	366	780	**781**	781	790
gsm.dec	× 10^6	1025	699	905	821	682	588	644	554	582	401	**396**	411	372	292	289	302	312	309	251	258	239	248	195	396	**396**	396	400
Stanford (Execution time in clock cycles)																												
perm	× 10^3	1512	1611	959	854	897	737	837	734	982	764	**719**	764	627	663	666	557	639	643	481	570	428	397	384	715	**719**	724	777
towers	× 10^3	1297	1358	609	557	675	664	786	660	729	567	**601**	589	524	412	492	427	380	380	409	431	303	294	296	598	**601**	603	633
queens	× 10^3	1455	952	830	769	926	651	871	650	878	847	**776**	823	517	762	795	688	736	693	735	717	309	285	234	775	**776**	776	781
intmm	× 10^3	1365	1108	1282	1193	838	651	554	637	389	336	**372**	351	350	327	316	301	320	320	202	173	204	214	169	371	**372**	373	390
puzzle	× 10^4	1330	880	568	487	618	505	523	483	636	523	**556**	535	687	498	552	536	443	441	376	368	290	293	265	555	**556**	557	566
quicksort	× 10^3	2249	1226	1136	1083	1544	1000	1646	1014	1247	1050	**1017**	1330	1414	1020	1333	1350	1115	1130	1086	1162	1122	1125	1089	1013	**1017**	1021	1061
bubblesort	× 10^3	2656	1467	1003	1008	1493	989	1427	1009	860	1133	**1038**	1234	1477	1171	1444	1384	1037	1036	1231	1220	664	649	590	1037	**1038**	1038	1046
treesort	× 10^3	4718	5211	3827	3190	3086	2779	3015	2618	3176	2365	**2076**	2295	2359	1962	2541	2549	2006	1839	1882	1928	1785	1735	1689	2035	**2076**	2117	2618
dhrystone	× 10^6	442	286	342	318	237	238	261	208	164	182	**198**	130	149	156	137	118	138	164	92	90	85	82	73	198	**198**	198	198
coremark	× 10^6	408	286	286	270	251	228	207	215	168	185	**216**	185	152	153	160	157	136	136	119	127	106	99	92	216	**216**	216	216
bochs	× 10^7	520	725	431	315	322	332	272	295	265	252	**306**	226	216	192	176	166	173	161	154	151	133	128	106	305	**306**	307	321
s86sim	× 10^7	1474	3163	2679	2227	2143	1774	1319	1413	1087	1336	**954**	1353	695	732	639	590	573	548	648	536	459	449	283	933	**954**	976	1251
Geomean Runtime (relative)																												
Bochs + s86sim		1.62	2.80	1.99	1.55	1.54	1.42	1.11	1.20	0.99	1.07	**1.00**	1.02	0.72	0.69	0.62	0.58	0.58	0.55	0.58	0.53	0.46	0.44	0.32	0.99	**1.00**	1.01	1.17
SPECint2000		2.68	2.73	2.06	1.65	1.63	1.46	1.58	1.79	1.42	1.26	**1.00**	0.99	0.87	0.91	0.72	0.68	0.69	0.65	0.56	0.56	0.50	0.47	0.44	0.94	**1.00**	1.06	1.70
MiBench		2.42	2.25	2.01	1.70	1.43	1.32	1.20	1.25	1.14	0.96	**1.00**	0.93	0.81	0.75	0.78	0.74	0.74	0.67	0.60	0.61	0.52	0.49	0.46	0.99	**1.00**	1.01	1.14
Dhrystone + CoreMark		2.05	1.38	1.51	1.42	1.18	1.13	1.13	1.02	0.80	0.89	**1.00**	0.75	0.73	0.75	0.72	0.66	0.66	0.72	0.51	0.52	0.46	0.44	0.40	1.00	**1.00**	1.00	1.00
Stanford		2.36	1.83	1.34	1.22	1.39	1.08	1.29	1.07	1.17	1.02	**1.00**	1.07	1.04	0.92	1.05	0.98	0.89	0.87	0.81	0.82	0.62	0.60	0.55	1.00	**1.00**	1.01	1.06
Geomean (of 5 groups)		2.20	2.13	1.75	1.50	1.43	1.27	1.25	1.24	1.09	1.03	**1.00**	0.95	0.82	0.80	0.76	0.72	0.71	0.69	0.60	0.60	0.51	0.49	0.43	0.98	**1.00**	1.02	1.19

Table C.2: Benchmark runtimes on 23 CPUs and with varying memory latency on our design.

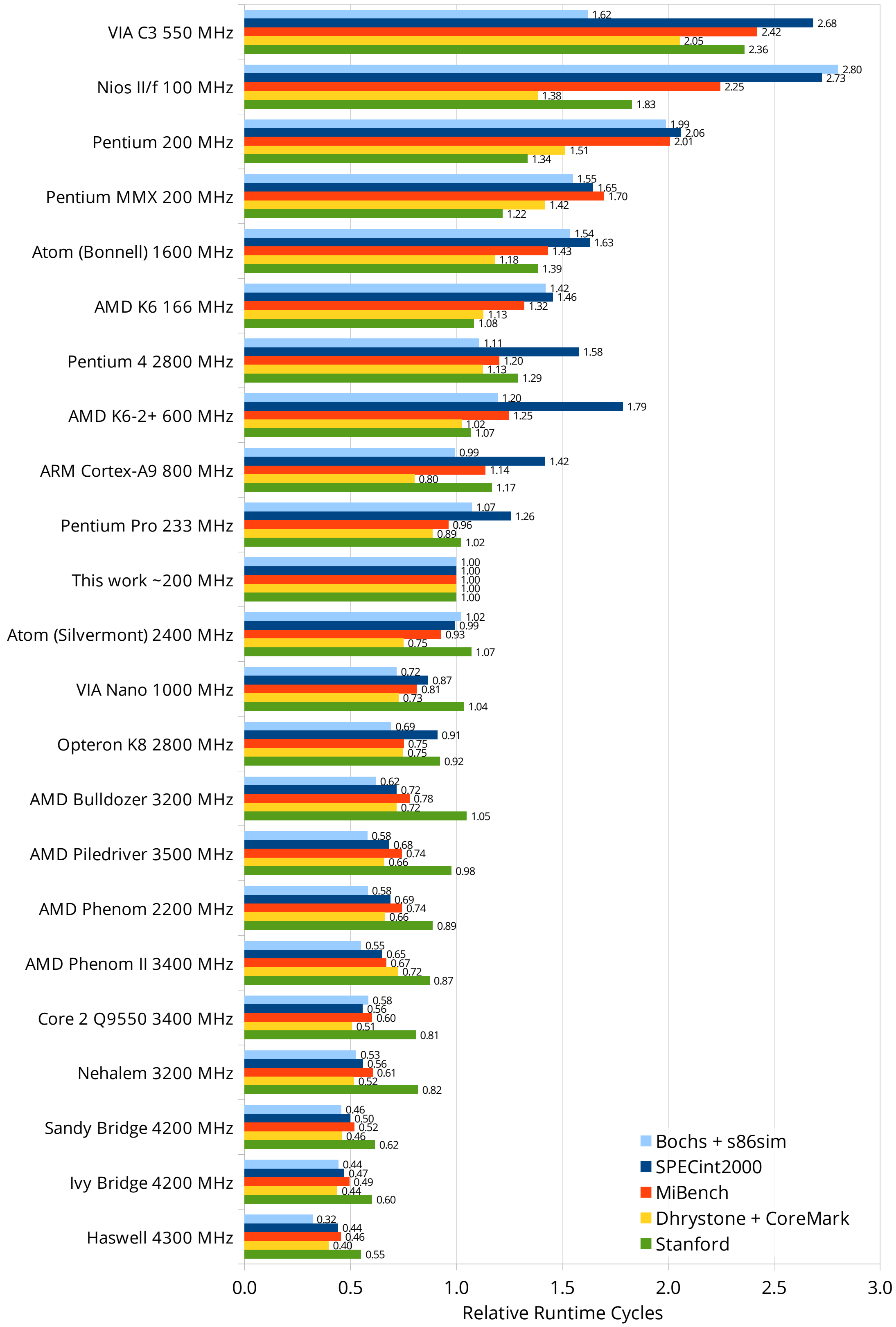
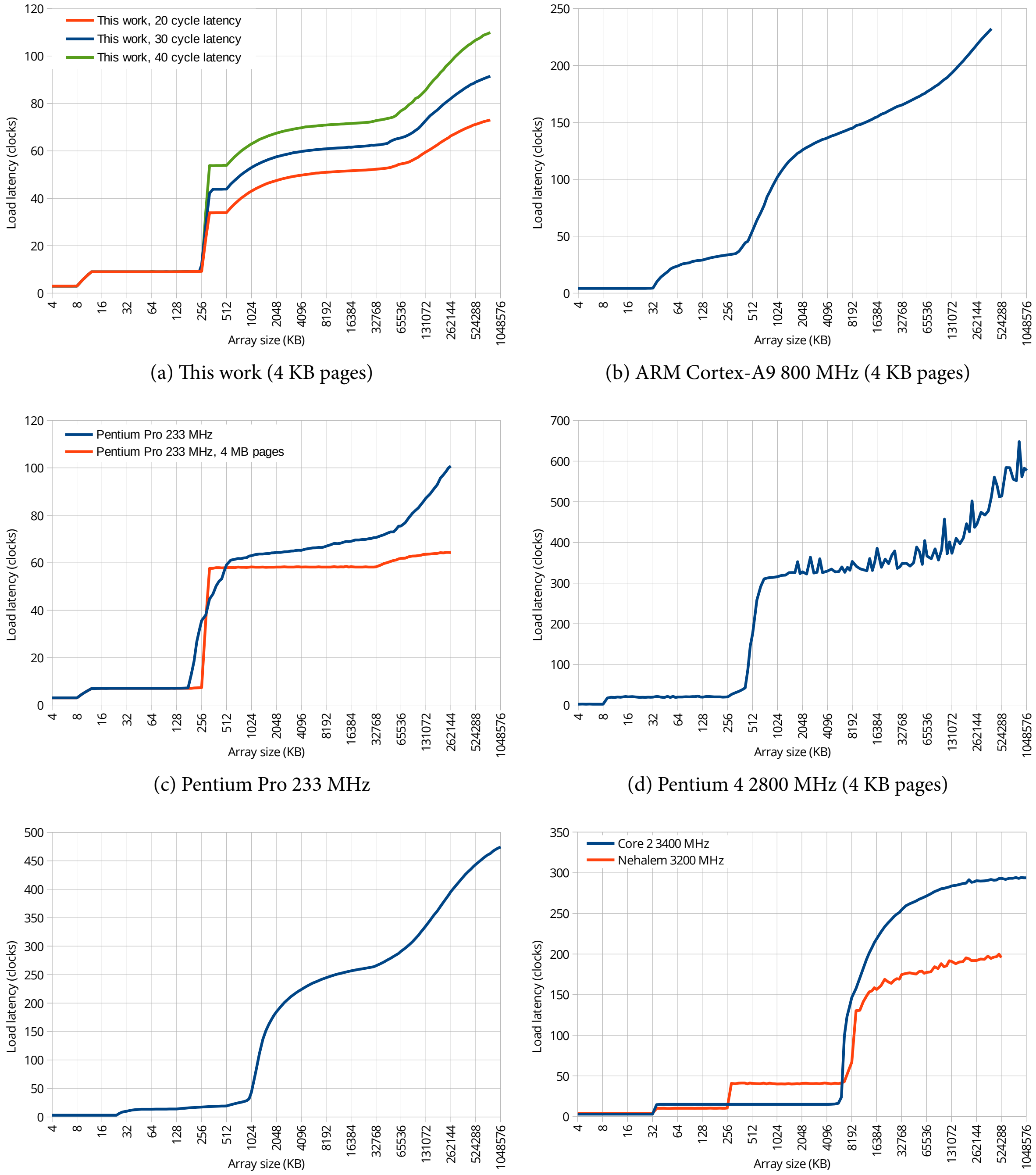


Figure C.1: Relative runtime



(a) This work (4 KB pages)

(b) ARM Cortex-A9 800 MHz (4 KB pages)

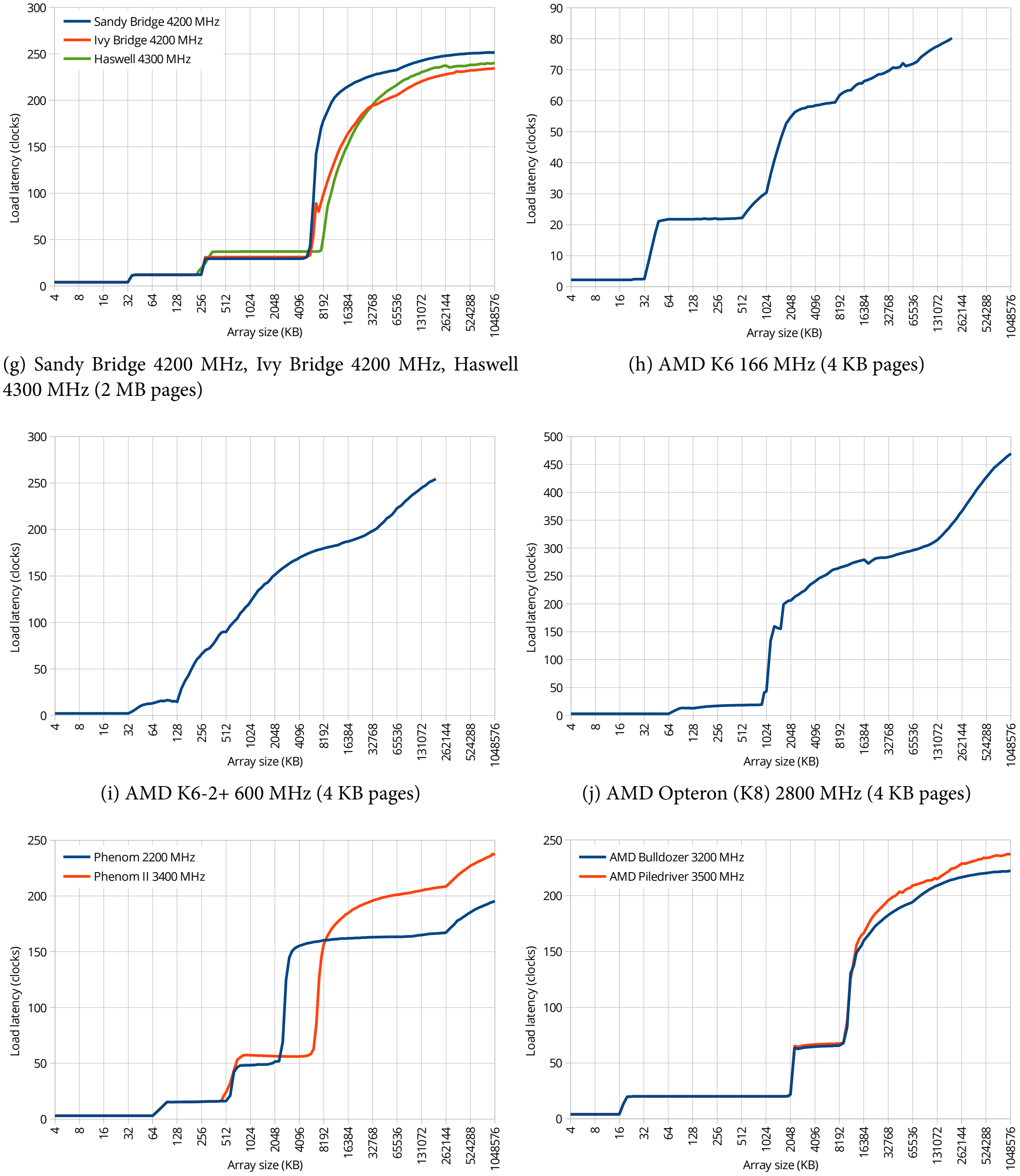
(c) Pentium Pro 233 MHz

(d) Pentium 4 2800 MHz (4 KB pages)

(e) Atom Silvermont 2417 MHz (4 KB pages)

(f) Core 2 3400 MHz, Nehalem 3200 MHz (2 MB pages)

Figure C.2: Memory latency



(g) Sandy Bridge 4200 MHz, Ivy Bridge 4200 MHz, Haswell 4300 MHz (2 MB pages)

(h) AMD K6 166 MHz (4 KB pages)

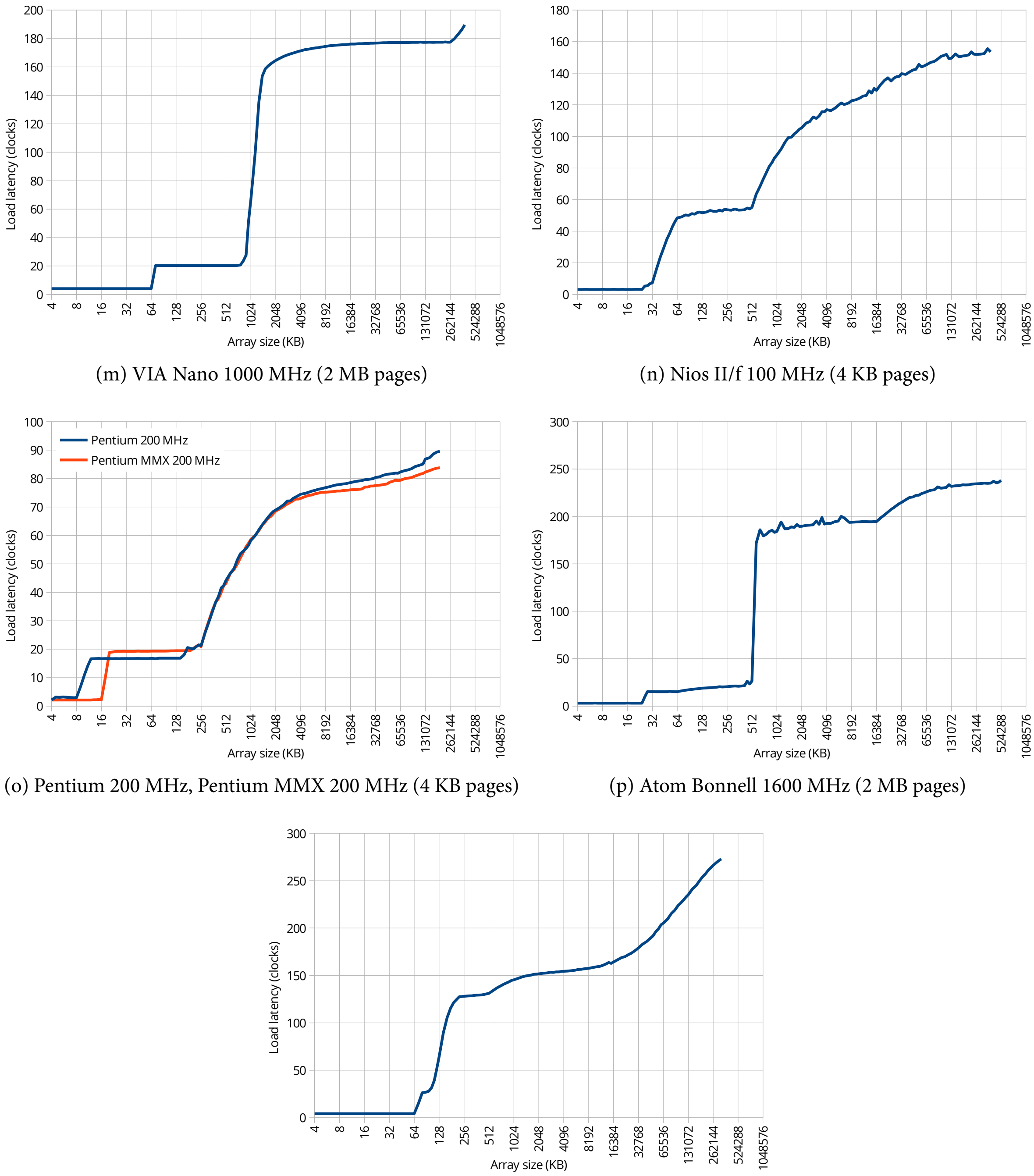
(i) AMD K6-2+ 600 MHz (4 KB pages)

(j) AMD Opteron (K8) 2800 MHz (4 KB pages)

(k) AMD Phenom 2200 MHz, Phenom II 3400 MHz (2 MB pages)

(l) AMD Bulldozer 3200 MHz, Piledriver 3500 MHz (2 MB pages)

Figure C.2: Memory latency (continued)



(m) VIA Nano 1000 MHz (2 MB pages)

(n) Nios II/f 100 MHz (4 KB pages)

(o) Pentium 200 MHz, Pentium MMX 200 MHz (4 KB pages)

(p) Atom Bonnell 1600 MHz (2 MB pages)

(q) VIA C3 550 MHz (4 KB pages)

Figure C.2: Memory latency (continued)

Appendix D

List of Micro-Ops and Logical Registers

D.1 List of Micro-ops

The following two tables list the micro-ops that the execution units of our processor supports. x86 instructions and other behaviours (exception handling) are decoded into sequences of these micro-ops. Each of our micro-ops contain exactly one ALU operation *and* one memory operation, either (or both) of which may be a NOP.

Table D.1: List of ALU micro-ops

ALU Micro-op(s)	Function
Arithmetic	
`NOP`	Often used internally to pad pure memory micro-ops with no arithmetic operation.
`ADD`	Add
`ADC`	Add with carry.
`SUB`	Subtract
`SBB`	Subtract with borrow
`ADDSUBDF`	Add or subtract based on DF (for incrementing pointers in string operations)
`AND`	Bitwise AND
`ANDNOT`	Bitwise AND-NOT (Not directly used by an x86 instruction)
`OR`	Bitwise OR
`OR_ZF`	Bitwise OR, but modify only ZF flag (for CMPXCHG8B)
`XOR`	Bitwise XOR
`MUX*`	16 conditional mux micro-ops to implement CMOVcc
`BSF/BSR`	Bit scan forward/reverse
`CBW`	Sign-extend 8/16 to 16/32 bits
`CWD`	Sign-extend 32 to 64: Compute upper 32 bits
`BSWAP`	32-bit byte swap

Table D.1: List of ALU micro-ops (continued)

ALU Micro-op(s)	Function
`ROL/ROR`	8/16/32-bit rotates
`RCL/RCR`	9/17/33-bit rotate-through-carry
`SHL/SHR/SAR`	8/16/32-bit shifts
`SHLD/SHRD`	32/64-bit double-length shifts
`MUL/IMUL`	Unsigned and signed multiplication
`DIV/IDIV`	Unsigned and signed division
`MOVREG_MD`	Extract upper 32-bits of a 64-bit GPR to another 32-bit GPR
`MOVMD_REG`	Combine two 32-bit values into a 64-bit GPR
`OUT`	I/O
`IN`	I/O
Branches	
`Jcc`	16 conditional branch micro-ops to implement Jcc
`LOOPcc/JCXZ`	Four conditional branch micro-ops that depend on ZF and a register value
`JMP`	A family of three micro-ops for unconditional branches. First variation always signals a misprediction. Second variation changes CS permission checks far jumps.
String Operations	
`REP_*`	Corresponds to the x86 REP instructions
`REP_LOADREGS`	Set up string unit parameters
`CMPSE/CMPSNE`	SUB, but causes string-op abort depending on ZF
Segmentation	
`MOVSEG_RM`	A family of micro-ops with minor variations to move an integer register into a segment selector.
`MOVRM_SEG`	Read segment selector into integer register
`MOVRM_SEGBASE`	Read IDTR or GDTR base into integer register (for SGDT and SIDT)
`MOVRM_SEGLIMIT`	Read segment base into integer register (for SGDT and SIDT)
`MOV_SEGD_LO`	Move 32-bit value into lower half of segment descriptor
`MOV_SEGD_HI`	Move 32-bit value into upper half of segment descriptor
`SET8/SET9`	Set bit 8 or bit 9 of operand if segment selector is non-zero
`ARPL`	Adjust RPL field of selector
`LAR/LAR2_HI`	Load access rights byte
`LSL/LSL2_LO/LSL2_HI`	Load segment limit
`VERR/VERR2_HI/VERW/VERW2_HI`	Verify read or write access permissions
Exception Handling	
`PE_INT*`	Micro-ops to represent various phases of protected-mode exception handling and protected-mode IRET
`THROW/THROWNZ`	Signals a trap unconditionally or if OF is non-zero (for INTO)

Table D.1: List of ALU micro-ops (continued)

ALU Micro-op(s)	Function
FPU_CHECKEXCEPTIONS	NOP, but checks for FPU exceptions.
Other	
MERGE_FLAGS	Reads condition codes and merges with 32-bit EFLAGS. A family of 3 micro-ops that vary on whether some bits are masked out.
UPDATE_FLAGS	Writes condition codes based on 32-bit EFLAGS. A family of 3 micro-ops that vary on which bits may change (POPF and IRET are special).
MOVCRREG_RM	Move into control register
MOVDREG_RM	Move into debug register
RDMSR/WRMSR	Read and write machine-specific register
CPUID	CPUID
HLT	Halt until next interrupt

Table D.2: List of memory micro-ops

Memory Micro-op(s)	Function
None	NOP
Load	Load
Load + RMW	Load, but checks for write permissions
Store	Store
Load and Store	Read-modify-write (x86 memory destination)
LEA	Compute effective address but no memory operation
INVLPG	Invalidate TLB entry

D.2 List of Logical Registers

There are three types of registers that are renamed: Integer general-purpose registers, condition codes, and segment registers. The following tables lists the logical registers of each type (before renaming).

Table D.3: List of logical general-purpose integer registers

Integer Register	Function
eax	Architectural register
edx	Architectural register
ecx	Architectural register
ebx	Architectural register
esp	Architectural register
ebp	Architectural register
esi	Architectural register
edi	Architectural register
eflags	The EFLAGS register except the 6 bits of condition codes.
fpucw	16-bit FPU control word
fpusw	16-bit FPU status word
tmp0	Temporary register for micro-op sequences
tmp1	Temporary register for micro-op sequences

Table D.4: List of logical condition code registers

Flags Register	Function
oczaps	6-bit condition codes

Table D.5: List of logical segment registers

Segment Register	Function
es	Architectural ES
cs	Architectural CS
ss	Architectural SS
ds	Architectural DS
fs	Architectural FS
gs	Architectural GS
zs	Zero base address, no limit, and no paging. Used for FPU emulation.
idtr	Architectural IDTR
gdtr	Architectural GDTR
ldtr	Architectural LDTR
tr	Task Register (pointer to task state segment)
id	Interrupt descriptor temporary

Appendix E

Comparison with Intel Haswell

Using performance counters on an Intel Haswell processor, we can get a more detailed microarchitecture-level comparison between a commercial out-of-order x86 processor and our design. The processors are not designed to be comparable in performance (Haswell is a much larger and more complex processor), so we expect Haswell to substantially outperform our design. However, the detailed comparisons using performance counters can serve to sanity-check various components in our processor design.

In this appendix, we run CoreMark on Haswell (Core i7-4770K) and compare it to our microarchitecture simulation using the configuration found in Table 13.2. We run the same (32-bit Linux) executable on both systems. On Haswell, data is collected using Intel VTune Amplifier XE 2013, using Haswell's performance counters and sampling-based profiling. On our processor design, we use statistics computed by our microarchitecture-level simulator.

The following table shows the comparisons for CoreMark run with 6000 iterations. We included metrics where the Haswell performance counters roughly correspond to metrics reported by our cycle-level simulator. For the metrics that count the number of occurrences of a particular event, we report the count in units of millions.

Metric	Haswell ($\times 10^6$)	Our Processor ($\times 10^6$)	Section
Instructions and micro-ops Instructions retired [inst_retired.any]	1994	1998	
Clock cycles [cpu_clk_unhalted.thread]	1004	1870	
Micro-ops issued (fused) [uops_issued.any]	1974	2359	5.2
Micro-ops retired (fused) [uops_retired.retire_slots]	1888	2039	8.3
Branches retired			6.7
Branches retired [br_inst_retired.all_branches]	302	303	
Branch mispredictions retired [br_misp_retired.all_branches]	1.1 (0.36%)	18.1 (6.0%)	
Branches executed (% Mispredicted)			
Total branches executed [br_inst_exec.all_branches]	263	303	
Conditional [br_inst_exec.all_conditional]	236 (0.5%)	276 (5.9%)	
Direct unconditional branch [br_inst_exec.all_direct_jmp]	12 (0%)	11 (0%)	
Direct call [br_inst_exec.all_direct_near_call]	7.7 (0%)	7.7 (0%)	
Return [br_inst_exec.all_indirect_near_return]	7.8 (0.01%)	7.7 (0.05%)	
Indirect branch [br_inst_exec.all_indirect_jump_non_call_ret]	0.0005 (35%)	0.004 (97%)	
Indirect call [br_inst_exec.taken_indirect_near_call]	0.009 (5%)	0.02 (2%)	
Memory	**Retired**	**Executed**	
Loads (including RMW) [mem_uops_retired.all_loads]	452	472	
Stores (including RMW) [mem_uops_retired.all_stores]	158	173	
Loads			
L1D hit [mem_load_uops_retired.l1_hit]	452	472	12.6.2
L1D misses [mem_load_uops_retired.l1_miss]	0.17	0.20	12.6.2
L1D secondary miss [mem_load_uops_retired.hit_lfb]	0.03	0.07	12.5.2
Store-to-load forwarding conflict [ld_blocks.store_forward]	1.38	1.36	11.1.4
Pipeline flushes			
Branch mispredictions retired [br_misp_retired.all_branches]	1.1	18.1	
Non-branch pipeline flushes [machine_clears.count]	0.085	0.002	8.3.2
Cycles stalled for RAT checkpoint [int_misc.recovery_cycles]	12	27	7.2.2
RAT checkpoint stall cycles per pipeline flush	9.7	1.5	7.2.2

As the two processors are running the same user-mode workload, we expect the number of retired instructions to be similar. This is indeed the case: The number of retired instructions is about 0.3% lower on Haswell than our processor. This agrees with the observation made in Section 13.1.1 that faster processors tend to execute fewer instructions due to less overhead from the periodic timer interrupt. Runtime (clock cycles) is, of course, much lower on Haswell (46% less). The number of micro-ops on Haswell is also lower and is even lower than the instruction count (0.95 micro-ops per x86 instruction). We speculate that this is due to macro-op fusion, which allows many compare-and-branch pairs of x86 instructions to be decoded into one micro-op.

One significant difference between the two processors is the branch misprediction rate. Although our design has a reasonable 6.0% branch misprediction rate (or 9.1 misses per 1000 instructions (MPKI)), Haswell has far fewer mispredictions, at 0.36% misprediction rate (or 0.57 MPKI). This difference affects other metrics as well, as more branch mispredictions cause more micro-ops to be executed and then discarded (Issued but not retired micro-ops are 4.6% on Haswell vs. 16% on our design).

The distribution of branch types is fairly similar between the two processors, although we question the accuracy of the count of indirect branches because there are so few. Another unusual observation on the Haswell processor is that the number of branches executed is 13% lower than the number of branches retired. This may be a bug in the performance counters, or a hint that there is a mechanism that allows some conditional branches to retire without being executed.

The memory micro-op counts are measured at different parts of the pipeline between the two processors. On Haswell, performance counters exist only to count cache behaviour for committed micro-ops, while our simulator currently counts cache events only during execution. Thus, it is expected that the number of loads and stores will be higher on our processor when counting executed (including squashed) micro-ops than on the Haswell which only counts committed (excluding squashed) micro-ops. It is difficult to collect reliable cache statistics using CoreMark because the number of cache misses is extremely small.

The number of stall cycles waiting for a register renamer (RAT) checkpoint to be recovered may not be directly comparable between the two processors. For our processor, the metric measures the number of cycles new corrected-path instructions must wait at the register renamer while waiting for the checkpoint to be restored after a pipeline flush, and does not include any cycles where the new instructions have not yet reached the renamer stage. In other words, it measures only the impact of RAT recovery on new instructions, rather than measuring the number of cycles taken by the recovery mechanism. On Haswell, it is unclear which case is being measured, but given the relatively larger number of stall cycles per pipeline flush (9.7), we speculate that it may actually be measuring all cycles spent doing RAT checkpoint recovery regardless of whether any new instructions are being delayed.

Bibliography

[1] Intel, *Intel 64 and IA-32 Architectures Software Developer Manuals*, 2016.

[2] Advanced Micro Devices, *AMD64 Architecture Programmer's Manual*, 2013.

[3] C. E. LaForest and J. G. Steffan, "Octavo: an FPGA-centric processor family," in *Proc. FPGA*, 2012, pp. 219–228.

[4] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, Jul. 2008.

[5] A. Severance and G. G. F. Lemieux, "Embedded supercomputing in FPGAs with the VectorBlox MXP Matrix Processor," in *Proc. CODES+ISSS*, Sept 2013, pp. 1–10.

[6] SPEC, "SPEC CPU95 results," https://www.spec.org/cpu95/results/, 2000.

[7] B. Kuttanna, "Technology insight: Intel Silvermont microarchitecture," IDF 2013, https://software.intel.com/sites/default/files/managed/bb/2c/02_Intel_Silvermont_Microarchitecture.pdf, 2013.

[8] E. Blem, J. Menon, T. Vijayaraghavan, and K. Sankaralingam, "ISA wars: Understanding the relevance of ISA being RISC or CISC to performance, power, and energy on modern architectures," *ACM Trans. Comput. Syst.*, vol. 33, no. 1, pp. 3:1–3:34, mar 2015.

[9] A. Waterman, "Design of the RISC-V instruction set architecture," University of California at Berkeley, Tech. Rep., 2016. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.pdf

[10] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović, "The RISC-V instruction set manual volume II: Privileged architecture version 1.9.1," University of California at Berkeley, Tech. Rep., 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-161.html

[11] C. Trippel, Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, "TriCheck: Memory model verification at the trisection of software, hardware, and ISA," in *Proc. ASPLOS*. ACM, 2017, pp. 119–133.

[12] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann, 2003.

[13] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Res. Dev.*, vol. 11, no. 1, pp. 25–33, Jan. 1967.

[14] J. Smith and A. Pleszkun, "Implementing precise interrupts in pipelined processors," *IEEE Trans. Computers*, vol. 37, no. 5, pp. 562–573, 1988.

[15] J. P. Shen and M. H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press, 2013.

[16] Xilinx, *MicroBlaze Processor Reference Guide*, 2014.

[17] Altera, *Nios II Gen2 Processor Reference Guide*, 2016.

[18] L. Semiconductor, *LatticeMico32 Processor Reference Manual*, 2009.

[19] ARM. (2016) Cortex-M1 processor. [Online]. Available: http://www.arm.com/products/processors/cortex-m/cortex-m1.php

[20] A. Gaisler, *GRLIP IP Core User's Manual 1.4.1*, 2015.

[21] G. Schelle, J. Collins, E. Schuchman, P. Wang, X. Zou, G. Chinya, R. Plate, T. Mattner, F. Olbrich, P. Hammarlund, R. Singhal, J. Brayton, S. Steibl, and H. Wang, "Intel Nehalem processor core made FPGA synthesizable," in *Proc. FPGA*, 2010, pp. 3–12.

[22] C. Celio, D. A. Patterson, and K. Asanovic, "The Berkeley Out-of-Order Machine (BOOM): An industry-competitive, synthesizable, parameterized RISC-V processor," University of California at Berkeley, Tech. Rep. UCB/EECS–2015-167, Jun 2015. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html

[23] W. Terpstra, "OPA: Out-of-order superscalar soft CPU," in *ORCONF*, 2015.

[24] A. Severance and G. Lemieux, "VENICE: A compact vector processor for FPGA applications," in *Proc. FPT*, 2012, pp. 261–268.

[25] P. Yiannacouras, J. G. Steffan, and J. Rose, "Fine-grain performance scaling of soft vector processors," in *Proc. CASES*, 2009, pp. 97–106.

[26] P. Yiannacouras, "FPGA-based soft vector processors," Ph.D. dissertation, University of Toronto, 2009.

[27] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster, "An FPGA-based VLIW processor with custom hardware execution," in *Proc. FPGA*, 2005, pp. 107–117.

[28] R. Rashid, J. G. Steffan, and V. Betz, "Comparing performance, productivity and scalability of the TILT overlay processor to OpenCL HLS," in *Proc. FPT*, Dec 2014, pp. 20–27.

[29] K. Andryc, M. Merchant, and R. Tessier, "FlexGrip: A soft GPGPU for FPGAs," in *Proc. FPT*, Dec 2013, pp. 230–237.

[30] B. Fort, D. Capalija, Z. G. Vranesic, and S. D. Brown, "A multithreaded soft processor for SoPC area reduction," in *Proc. FCCM*, April 2006, pp. 131–142.

[31] M. Labrecque and J. Steffan, "Improving pipelined soft processors with multithreading," in *Proc. FPL*, 2007, pp. 210–215.

[32] H. Y. Cheah, S. A. Fahmy, and D. L. Maskell, "iDEA: A DSP block based FPGA soft processor," in *Proc. FPT*, Dec 2012, pp. 151–158.

[33] K. Aasaraai and A. Moshovos, "SPREX: A soft processor with runahead execution," in *2012 International Conference on Reconfigurable Computing and FPGAs*, Dec 2012, pp. 1–7.

[34] D. Lampret, *OpenRISC 1200 IP Core Specification*.

[35] C. Celio, K. Asanovic, and D. Patterson, "The Berkeley out-of-order machine (BOOM!): An open-source industry-competitive, synthesizable, parameterized RISC-V processor," https://riscv.org/wp-content/uploads/2016/01/Wed1345-RISCV-Workshop-3-BOOM.pdf, Jan 2016.

[36] K. Aasaraai and A. Moshovos, "Design space exploration of instruction schedulers for out-of-order soft processors," in *Proc. FPT*, Dec 2010.

[37] F. J. Mesa-Martínez, M. C. Huang, and J. Renau, "Seed: Scalable, efficient enforcement of dependences," in *Proc. PACT*, 2006.

[38] M. Rosière, J.-L. Desbarbieux, N. Drach, and F. Wajsburt, "An out-of-order superscalar processor on FPGA: The reorder buffer design," in *Proc. DATE*, 2012.

[39] A. Johri, "Implementation of instruction scheduler on FPGA," Master's thesis, University of Tokyo, 2011.

[40] K. Aasaraai and A. Moshovos, "Towards a viable out-of-order soft core: Copy-free, checkpointed register renaming," in *Proc. FPL*, Aug 2009, pp. 79–85.

[41] ——, "An efficient non-blocking data cache for soft processors," in *Proc. Reconfig*, Dec 2010, pp. 19–24.

[42] P. H. Wang, J. D. Collins, C. T. Weaver, B. Kuttanna, S. Salamian, G. N. Chinya, E. Schuchman, O. Schilling, T. Doil, S. Steibl, and H. Wang, "Intel Atom processor core made FPGA-synthesizable," in *Proc. FPGA*, 2009, pp. 209–218.

[43] S.-L. L. Lu, P. Yiannacouras, R. Kassa, M. Konow, and T. Suh, "An FPGA-based Pentium in a complete desktop system," in *Proc. FPGA*, 2007, pp. 53–59.

[44] K. Huang, J. Lu, J. Pang, Y. Zheng, H. Li, D. Tong, and X. Cheng, "FPGA prototyping of an Amba-based Windows-compatible SoC," in *Proc. FPGA*, ser. FPGA '10, 2010, pp. 13–22.

[45] Z. G. Marmolejo. (2014) Zet processor. [Online]. Available: http://zet.aluzina.org/

[46] HT-Lab. (2016) CPU86. [Online]. Available: http://www.ht-lab.com/cpu86.htm

[47] A. Osman. (2014) ao486. [Online]. Available: https://github.com/alfikpl/ao486

[48] E. Ogawa, Y. Matsuda, T. Misono, R. Kobayashi, and K. Kise, "Reconfigurable IBM PC compatible SoC for computer architecture education and research," in *Proc. IEEE Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, Sept 2015, pp. 65–72.

[49] D. Lewis, E. Ahmed, G. Baeckler, V. Betz, M. Bourgeault, D. Cashman, D. Galloway, M. Hutton, C. Lane, A. Lee, P. Leventis, S. Marquardt, C. McClintock, K. Padalia, B. Pedersen, G. Powell, B. Ratchev, S. Reddy, J. Schleicher, K. Stevens, R. Yuan, R. Cliff, and J. Rose, "The Stratix II logic and routing architecture," in *Proc. FPGA*, 2005, pp. 14–20.

[50] H. Wong, V. Betz, and J. Rose, "Comparing FPGA vs. Custom CMOS and the Impact on Processor Microarchitecture," in *Proc. FPGA*, 2011, pp. 5–14.

[51] ——, "Quantifying the gap between FPGA and custom CMOS to aid microarchitectural design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 10, pp. 2067–2080, Oct 2014.

[52] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, Feb. 2007.

[53] D. Chinnery and K. Keutzer, *Closing the Gap Between ASIC & Custom, Tools and Techniques for High-Performance ASIC Design*. Kluwer Academic Publishers, 2002.

[54] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proc. ISCA*, 1997.

[55] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus IPC: The end of the road for conventional microarchitectures," *SIGARCH Comp. Arch. News*, vol. 28, no. 2, pp. 248–259, May 2000.

[56] P. Metzgen and D. Nancekievill, "Multiplexer restructuring for FPGA implementation cost reduction," in *Proc. DAC*, 2005, pp. 421–426.

[57] P. Metzgen, "A high performance 32-bit ALU for programmable logic," in *Proc. FPGA*, 2004, pp. 61–70.

[58] S. Tyagi, C. Auth, P. Bai, G. Curello, H. Deshpande, S. Gannavaram, O. Golonzka, R. Heussner, R. James, C. Kenyon, S. H. Lee, N. Lindert, M. Liu, R. Nagisetty, S. Natarajan, C. Parker, J. Sebastian, B. Sell, S. Sivakumar, A. S. Amour, and K. Tone, "An advanced low power, high performance, strained channel 65nm technology," in *Proc. IEDM*, 2005, pp. 245–247.

[59] K. Mistry, C. Allen, C. Auth, B. Beattie, D. Bergstrom, M. Bost, M. Brazier, M. Buehler, A. Cappellani, R. Chau, C. H. Choi, G. Ding, K. Fischer, T. Ghani, R. Grover, W. Han, D. Hanken, M. Hattendorf, J. He, J. Hicks, R. Huessner, D. Ingerly, P. Jain, R. James, L. Jong, S. Joshi, C. Kenyon, K. Kuhn, K. Lee, H. Liu, J. Maiz, B. McIntyre, P. Moon, J. Neirynck, S. Pae, C. Parker, D. Parsons, C. Prasad, L. Pipes, M. Prince, P. Ranade, T. Reynolds, J. Sandford, L. Shifren, J. Sebastian, J. Seiple, D. Simon, S. Sivakumar, P. Smith, C. Thomas, T. Troeger, P. Vandervoorn, S. Williams, and K. Zawadzki, "A 45nm logic technology with high-k+metal gate transistors, strained silicon, 9 Cu interconnect layers, 193nm dry patterning, and 100% Pb-free packaging," in *Proc. IEDM*, Dec. 2007, pp. 247–250.

[60] A. S. Leon, K. W. Tam, J. L. Shin, D. Weisner, and F. Schumacher, "A power-efficient high-throughput 32-thread SPARC processor," *IEEE JSSC*, vol. 42, no. 1, pp. 295–304, 2007.

[61] U. G. Nawathe, M. Hassan, K. C. Yen, A. Kumar, A. Ramachandran, and D. Greenhill, "Implementation of an 8-core, 64-thread, power-efficient SPARC server on a chip," *IEEE JSSC*, vol. 43, no. 1, pp. 6–20, 2008.

[62] G. Gerosa, "A sub-2 W low power IA processor for mobile internet devices in 45 nm high-k metal gate CMOS," *IEEE JSSC*, vol. 44, no. 1, pp. 73–82, 2009.

[63] R. Kumar and G. Hinton, "A family of 45nm IA processors," in *Proc. ISSCC*, Feb. 2009, pp. 58–59.

[64] Sun Microsystems, "OpenSPARC," http://www.opensparc.net/, 2010.

[65] J. Davis, D. Plass, P. Bunce, Y. Chan, A. Pelella, R. Joshi, A. Chen, W. Huott, T. Knips, P. Patel, K. Lo, and E. Fluhr, "A 5.6GHz 64kB dual-read data cache for the POWER6 processor," in *Proc. ISSCC*, 2006.

[66] M. Khellah, N. S. Kim, J. Howard, G. Ruhl, M. Sunna, Y. Ye, J. Tschanz, D. Somasekhar, N. Borkar, F. Hamzaoglu, G. Pandya, A. Farhang, K. Zhang, and V. De, "A 4.2GHz 0.3mm^2 256kb dual-V$_{cc}$ SRAM building block in 65nm CMOS," in *Proc. ISSCC*, Feb. 2006, pp. 2572–2581.

[67] P. Bai, C. Auth, S. Balakrishnan, M. Bost, R. Brain, V. Chikarmane, R. Heussner, M. Hussein, J. Hwang, D. Ingerly, R. James, J. Jeong, C. Kenyon, E. Lee, S. H. Lee, N. Lindert, M. Liu, Z. Ma, T. Marieb, A. Murthy, R. Nagisetty, S. Natarajan, J. Neirynck, A. Ott, C. Parker, J. Sebastian, R. Shaheed, S. Sivakumar, J. Steigerwald, S. Tyagi, C. Weber, B. Woolery, A. Yeoh, K. Zhang, and M. Bohr, "A 65nm logic technology featuring 35nm gate lengths, enhanced channel strain, 8 Cu interconnect layers, low-k ILD and 0.57 μm^2 SRAM cell," in *Proc. IEDM*, 2004, pp. 657–660.

[68] P. Bai, “Foils from “a 65nm logic technology featuring 35nm gate lengths, enhanced channel strain, 8 Cu interconnect layers, low-k ILD and 0.57 μm^2 SRAM cell”,” IEEE International Electron Devices Meeting, 2004.

[69] L. Chang, Y. Nakamura, R. K. Montoye, J. Sawada, A. K. Martin, K. Kinoshita, F. H. Gebara, K. B. Agarwal, D. J. Acharyya, W. Haensch, K. Hosokawa, and D. Jamsek, “A 5.3GHz 8T-SRAM with operation down to 0.41V in 65nm CMOS,” in *Proc. VLSI*, Jun. 2007, pp. 252–253.

[70] S. Hsu, A. Agarwal, M. Anders, S. Mathew, R. Krishnamurthy, and S. Borkar, “An 8.8GHz 198mW 16x64b 1R/1W variation-tolerant register file in 65nm CMOS,” in *Proc. ISSCC*, 2006, pp. 1785–1797.

[71] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, “CACTI 5.1,” HP Laboratories, Palo Alto, Tech. Rep., 2008.

[72] C. E. LaForest and J. G. Steffan, “Efficient multi-ported memories for FPGAs,” in *Proc. FPGA*, 2010, pp. 41–50.

[73] K. Pagiamtzis and A. Sheikholeslami, “Content-addressable memory (CAM) circuits and architectures: A tutorial and survey,” *IEEE JSSC*, pp. 712–727, 2006.

[74] K. McLaughlin, N. O’Connor, and S. Sezer, “Exploring CAM design for network processing using FPGA technology,” in *Proc. AICT-ICIW*, 2006, p. 84.

[75] J.-L. Brelet and L. Gopalakrishnan, “Using Virtex-II block RAM for high performance read/write CAMs,” Xilinx Application Note XAPP260, 2002.

[76] I. Arsovski and R. Wistort, “Self-referenced sense amplifier for across-chip-variation immune sensing in high-performance content-addressable memories,” in *Proc. CICC*, 2006, pp. 453–456.

[77] D. W. Plass and Y. H. Chan, “IBM POWER6 SRAM arrays,” *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 747–756, 2007.

[78] W. Hu, J. Wang, X. Gao, Y. Chen, Q. Liu, and G. Li, “Godson-3: A scalable multicore RISC processor with x86 emulation,” *IEEE Micro*, vol. 29, no. 2, pp. 17–29, 2009.

[79] A. Agarwal, S. K. Hsu, H. Kaul, M. A. Anders, and R. K. Krishnamurthy, “A dual-supply 4GHz 13fJ/bit/search 64×128b CAM in 65nm CMOS,” in *Proc. ESSCIRC 32*, 2006, pp. 303–306.

[80] S. K. Hsu, S. K. Mathew, M. A. Anders, B. R. Zeydel, V. G. Oklobdzija, R. K. Krishnamurthy, and S. Y. Borkar, “A 110 GOPS/W 16-bit multiplier and reconfigurable PLA loop in 90-nm CMOS,” *IEEE JSSC*, vol. 41, no. 1, pp. 256–264, 2006.

[81] W. Belluomini, D. Jamsek, A. Martin, C. McDowell, R. Montoye, T. Nguyen, H. Ngo, J. Sawada, I. Vo, and R. Datta, “An 8GHz floating-point multiply,” in *Proc. ISSCC*, 2005.

[82] J. B. Kuang, T. C. Buchholtz, S. M. Dance, J. D. Warnock, S. N. Storino, D. Wendel, and D. H. Bradley, "The design and implementation of double-precision multiplier in a first-generation CELL processor," in *Proc. ICIDT*, 2005, pp. 11–14.

[83] P. Jamieson and J. Rose, "Mapping multiplexers onto hard multipliers in FPGAs," in *IEEE-NEWCAS*, 2005, pp. 323–326.

[84] A. E.-N. A. Agah, S. M. Fakhraie, "Tertiary-tree 12-GHz 32-bit adder in 65nm technology," in *Proc. ISCAS*, 2007, pp. 3006–3009.

[85] S. Kao, R. Zlatanovici, and B. Nikolic, "A 240ps 64b carry-lookahead adder in 90nm CMOS," in *Proc. ISSCC*, 2006, pp. 1735–1744.

[86] S. B. Wijeratne, N. Siddaiah, S. K. Mathew, M. A. Anders, R. K. Krishnamurthy, J. Anderson, M. Ernest, and M. Nardin, "A 9-GHz 65-nm Intel Pentium 4 processor integer execution unit," *IEEE JSSC*, vol. 42, no. 1, pp. 26–37, Jan. 2007.

[87] X. Y. Zhang, Y.-H. Chan, R. Montoye, L. Sigal, E. Schwarz, and M. Kelly, "A 270ps 20mW 108-bit end-around carry adder for multiply-add fused floating point unit," *Signal Processing Systems*, vol. 58, no. 2, pp. 139–144, 2010.

[88] K. Vitoroulis and A. Al-Khalili, "Performance of parallel prefix adders implemented with FPGA technology," in *Proc. NEWCAS Workshop*, 2007, pp. 498–501.

[89] M. Alioto and G. Palumbo, "Interconnect-aware design of fast large fan-in CMOS multiplexers," *IEEE Trans. Circuits and Systems II*, vol. 54, no. 6, pp. 484–488, Jun. 2007.

[90] Altera, *Stratix III Device Handbook Volume 1*, 2009.

[91] E. Sprangle and D. Carmean, "Increasing processor performance by implementing deeper pipelines," *SIGARCH Comp. Arch. News*, vol. 30, no. 2, pp. 25–34, 2002.

[92] A. Hartstein and T. R. Puzak, "The optimum pipeline depth for a microprocessor," *SIGARCH Comp. Arch. News*, vol. 30, no. 2, pp. 7–13, May 2002.

[93] M. S. Hrishikesh, D. Burger, N. P. Jouppi, S. W. Keckler, K. I. Farkas, and P. Shivakumar, "The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays," in *Proc. ISCA 29*, 2002, pp. 14–24.

[94] ITRS. (2007) International technology roadmap for semiconductors. [Online]. Available: http://www.itrs.net/Links/2007ITRS/Home2007.htm

[95] Altera, *External Memory Interface Handbook, Volume 3*, Nov. 2011.

[96] P. Yiannacouras, J. Rose, and J. G. Steffan, "The microarchitecture of FPGA-based soft processors," in *Proc. CASES*, 2005, pp. 202–212.

[97] N. Malik, R. J. Eickemeyer, and S. Vassiliadis, “Interlock collapsing ALU for increased instruction-level parallelism,” *SIGMICRO Newsl.*, vol. 23, no. 1-2, pp. 149–157, Dec. 1992.

[98] J. Phillips and S. Vassiliadis, “High-performance 3-1 interlock collapsing ALU’s,” *IEEE Trans. Computers*, vol. 43, no. 3, pp. 257–268, Mar. 1994.

[99] P. G. Sassone and D. S. Wills, “Dynamic Strands: Collapsing Speculative Dependence Chains for Reducing Pipeline Communication,” in *Proc. MICRO*, Dec. 2004, pp. 7–17.

[100] A. W. Bracy, “Mini-graph processing,” Ph.D. dissertation, University of Pennsylvania, 2008.

[101] M. Zhang and K. Asanovic, “Highly-associative caches for low-power processors,” in *Kool Chips Workshop, Micro-33*, 2000.

[102] G. Sohi, “Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers,” *IEEE Trans. Computers*, vol. 39, pp. 349–359, 1990.

[103] L. Gwennap, “Intel’s P6 uses decoupled superscalar design,” *Microprocessor Report*, vol. 9, no. 2, pp. 9–15, feb 1995.

[104] M. Golden, S. Hesley, A. Scherer, M. Crowley, S. C. Johnson, S. Meier, D. Meyer, J. D. Moench, S. Oberman, H. Partovi, F. Weber, S. White, T. Wood, and J. Yong, “A seventh-generation x86 microprocessor,” *IEEE JSSC*, vol. 34, no. 11, pp. 1466–1477, nov 1999.

[105] K. Yeager, “The MIPS R10000 superscalar microprocessor,” *Micro, IEEE*, vol. 16, no. 2, pp. 28–41, apr 1996.

[106] G. Hinton, M. Upton, D. J. Sager, D. Boggs, D. M. Carmean, P. Roussel, T. I. Chappell, T. D. Fletcher, M. S. Milshtein, M. Sprague, S. Samaan, and R. Murray, “A 0.18-μm CMOS IA-32 processor with a 4-GHz integer execution unit,” *IEEE JSSC*, vol. 36, no. 11, pp. 1617–1627, nov 2001.

[107] T. N. Buti, R. G. McDonald, Z. Khwaja, A. Ambekar, H. Q. Le, W. E. Burky, and B. Williams, “Organization and implementation of the register-renaming mapper for out-of-order IBM POWER4 processors,” *IBM Journal of Research and Development*, vol. 49, no. 1, pp. 167–188, Jan. 2005.

[108] R. Kalla, B. Sinharoy, and J. Tendler, “IBM Power5 chip: a dual-core multithreaded processor,” *Micro, IEEE*, vol. 24, no. 2, pp. 40– 47, mar-apr 2004.

[109] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. V. Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams, “IBM POWER7 multicore server processor,” *IBM Journal of Research and Development*, vol. 55, no. 3, pp. 1:1–1:29, May 2011.

[110] B. Sinharoy, J. A. V. Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. M. Fernsler, “IBM

POWER8 processor core microarchitecture," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 2:1–2:21, Jan 2015.

[111] B. Burgess, B. Cohen, J. Dundas, J. Rupley, D. Kaplan, and M. Denman, "Bobcat: AMD's low-power x86 processor," *Micro, IEEE*, vol. 31, no. 2, pp. 16–25, march-april 2011.

[112] M. Golden, S. Arekapudi, and J. Vinh, "40-entry unified out-of-order scheduler and integer execution unit for the AMD Bulldozer x86-64 core," in *Proc. ISSCC.*, Feb 2011, pp. 80–82.

[113] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "A decoupled KILO-instruction processor," *Proc. HPCA*, pp. 53–64, Feb. 2006.

[114] Altera, *Nios Soft Core Embedded Processor Data Sheet*, June 2000.

[115] K. P. Lawton, "Bochs: A portable PC emulator for Unix/X," *Linux J.*, vol. 1996, no. 29es, sep 1996.

[116] D. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, no. 3, pp. 13–25, Jun. 1997.

[117] M. T. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," in *2007 IEEE International Symposium on Performance Analysis of Systems Software*, April 2007, pp. 23–34.

[118] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Workshop on Workload Characterization(WWC-4)*, Dec 2001, pp. 3–14.

[119] S. H. Yuko Hara, Hiroyuki Tomiyama and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis," *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.

[120] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz, "VTR 7.0: Next generation architecture and CAD system for FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 2, pp. 6:1–6:30, Jul. 2014.

[121] R. P. Weicker, "An overview of common benchmarks," *Computer*, vol. 23, no. 12, pp. 65–75, Dec. 1990.

[122] ARM, *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*, 2012.

[123] Intel, *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2016.

[124] Advanced Micro Devices, *Software Optimization Guide for AMD Family 15h Processors*, 2014.

[125] ARM, *Cortex-A57 Software Optimization Guide*, 2016.

[126] ——, *Cortex-A72 Software Optimization Guide*, 2016.

[127] IBM, *Power ISA Version 2.07 B*, 2015.

[128] Intel, *80286 and 80287 Programmer's Reference Manual*, 1987.

[129] ——, *386 DX Microprocessor Programmer's Reference Manual*, 1990.

[130] D. Wu and A. Moshovos, "Advanced branch predictors for soft processors," in *Proc. ReConFig*, Dec 2014, pp. 1–6.

[131] A. Fog, *The Microarchitecture of Intel, AMD and VIA CPUs*, 2016.

[132] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint processing and recovery: An efficient, scalable alternative to reorder buffers," *IEEE Micro*, vol. 23, no. 6, pp. 11–19, Nov. 2003. [Online]. Available: http://dx.doi.org/10.1109/MM.2003.1261382

[133] A. Moshovos, "Checkpointing alternatives for high-performance, power-aware processors," in *Proc. ISPLED*, Aug 2003, pp. 318–321.

[134] A. Roth and G. S. Sohi, "Squash reuse via a simplified implementation of register integration," *JILP*, vol. 4, p. 2002, 2002.

[135] H. Wong, V. Betz, and J. Rose, "High performance instruction scheduling circuits for out-of-order soft processors," in *Proc. FCCM*, May 2016, pp. 9–16.

[136] J. Farrell and T. C. Fischer, "Issue logic for a 600-MHz out-of-order execution microprocessor," *IEEE JSSC*, vol. 33, no. 5, pp. 707–712, May 1998.

[137] S. Vangal, M. Anders, N. Borkar, E. Seligman, V. Govindarajulu, V. Erraguntla, H. Wilson, A. Pangal, V. Veeramachaneni, J. Tschanz, Y. Ye, D. Somasekhar, B. Bloechel, G. Dermer, R. Krishnamurthy, K. Soumyanath, S. Mathew, S. Narendra, M. Stan, S. Thompson, V. De, and S. Borkar, "5-GHz 32-bit integer execution core in 130-nm dual-VT CMOS," *IEEE JSSC*, vol. 37, no. 11, pp. 1421–1432, Nov 2002.

[138] L. Gwennap, "MIPS R12000 to hit 300 MHz," *Microprocessor Report*, vol. 11, no. 13, Oct 1997.

[139] M. D. Brown, J. Stark, and Y. N. Patt, "Select-free instruction scheduling logic," in *Proc. MICRO*, 2001.

[140] J. Stark, M. D. Brown, and Y. N. Patt, "On pipelining dynamic instruction scheduling logic," in *Proc. MICRO*, 2000.

[141] M. Goshima, K. Nishino, T. Kitamura, Y. Nakashima, S. Tomita, and S.-i. Mori, "A high-speed dynamic instruction scheduling scheme for superscalar processors," in *Proc. MICRO*, 2001.

[142] P. G. Sassone, J. Rupley, II, E. Brekelbaum, G. H. Loh, and B. Black, "Matrix scheduler reloaded," in *Proc. ISCA*, 2007, pp. 335–346.

[143] W. Lynch, G. Lautterbach, and J. Chamdani, "Low load latency through sum-addressed memory (SAM)," in *Proc. ISCA*, 1998.

[144] Altera, *Stratix IV Device Handbook Volume 1*, 2011.

[145] D. Harris, "A taxonomy of parallel prefix networks," in *Proc. Signals, Systems and Computers.*, vol. 2, Nov 2003.

[146] Altera, *Nios II Performance Benchmarks, DS-N28162004*, 2015.

[147] D. Ernst and T. Austin, "Efficient dynamic scheduling through tag elimination," in *Proc. ISCA*, 2002.

[148] I. Kim and M. Lipasti, "Half-price architecture," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, June 2003.

[149] C.-H. Chen and K.-S. Hsiao, "Scalable dynamic instruction scheduler through wake-up spatial locality," *IEEE Trans. Computers*, vol. 56, no. 11, pp. 1534–1548, Nov 2007.

[150] R. Canal and A. González, "A low-complexity issue logic," in *Proc. Supercomputing*, 2000.

[151] P. Michaud and A. Seznec, "Data-flow prescheduling for large instruction windows in out-of-order processors," in *Proc. HPCA*, 2001.

[152] D. Balkan, J. Sharkey, D. Ponomarev, and K. Ghose, "Selective writeback: Reducing register file pressure and energy consumption," *IEEE Trans. VLSI*, vol. 16, no. 6, pp. 650–661, June 2008.

[153] I. Park, M. D. Powell, and T. N. Vijaykumar, "Reducing register ports for higher speed and lower energy," in *Proc. Micro*, 2002, pp. 171–182.

[154] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," in *Proc. Micro*, ser. MICRO 29, 1996, pp. 226–237.

[155] E. S. Fetzer, M. Gibson, A. Klein, N. Calick, C. Zhu, E. Busta, and B. Mohammad, "A fully bypassed six-issue integer datapath and register file on the Itanium-2 microprocessor," *IEEE JSSC*, vol. 37, no. 11, pp. 1433–1440, Nov 2002.

[156] J. Clayton. (2009) Unsigned serial divider. [Online]. Available: https://opencores.org/project,serial_div_uu

[157] R. Herveille. (2009) Hardware division units. [Online]. Available: https://opencores.org/project,divider

[158] Altera, *Integer Arithmetic IP Cores User Guide*, Jun 2016.

[159] J. Cortadella and J. M. Llaberi, "Evaluating 'a+b=k' conditions in constant time," in *Proc. ISCAS*, 1988, pp. 243–246 vol.1.

[160] SPARC International, *The SPARC Architecture Manual*, 1994.

[161] H. Wong, V. Betz, and J. Rose, "Efficient methods for out-of-order load/store execution for high-performance soft processors," in *Proc. FPT*, Dec 2013, pp. 442–445.

[162] H. W. Cain, "Memory ordering: A value-based approach," in *Proc. ISCA*, 2004, pp. 90–101.

[163] A. Moshovos and G. Sohi, "Memory dependence speculation tradeoffs in centralized, continuous-window superscalar processors," in *Proc. HPCA*, 2000, pp. 301–312.

[164] T. Sha, M. M. K. Martin, and A. Roth, "Scalable store-load forwarding via store queue index prediction," in *Proc. Micro*, 2005, pp. 159–170.

[165] ——, "NoSQ: Store-load communication without a store queue," in *Proc. MICRO*, 2006, pp. 285–296.

[166] A. Moshovos, "Dynamic speculation and synchronization of data dependencies," in *Proc. ISCA*, 1997, pp. 181–193.

[167] H. Wong. (2014) Store-to-load forwarding and memory disambiguation in x86 processors. [Online]. Available: http://blog.stuffedcow.net/2014/01/x86-memory-disambiguation/

[168] A. Roth, "Store vulnerability window (SVW): A filter and potential replacement for load re-execution," *JILP*, vol. 8, 2006.

[169] A. Moshovos and G. S. Sohi, "Speculative memory cloaking and bypassing," *Int. J. Parallel Program.*, vol. 27, no. 6, dec 1999.

[170] H. Wong, V. Betz, and J. Rose, "Microarchitecture and circuits for a 200 MHz out-of-order soft processor memory system," *ACM TRETS*, vol. 10, no. 1, pp. 7:1–7:22, Dec. 2016.

[171] J. D. Woodruff, "CHERI: A RISC capability machine for practical memory safety," Ph.D. dissertation, University of Cambridge, 2014.

[172] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanovic, and K. Asanovic, "A 45nm 1.3GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators," in *European Solid State Circuits Conference (ESSCIRC)*, Sept 2014, pp. 199–202.

[173] K. Aasaraai and A. Moshovos, "An efficient non-blocking data cache for soft processors," in *Proc. ReConFig*, Dec 2010, pp. 19–24.

[174] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proc. ISCA*, Minneapolis, MN, USA, 1981, pp. 81–87.

[175] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation," Intel VSSAD, Tech. Rep., 2007.

[176] A. M. S. Abdelhadi and G. G. F. Lemieux, "Modular SRAM-based binary content-addressable memories," in *Proc. FCCM*, 2015, pp. 207–214.

[177] J. Power, M. Hill, and D. Wood, "Supporting x86-64 address translation for 100s of GPU lanes," in *Proc. HPCA*, Feb 2014, pp. 568–578.

[178] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. ASPLOS*, 2002, pp. 45–57.

[179] H. Wong, "A Superscalar Out-of-Order x86 Soft Processor for FPGA," Ph.D. dissertation, University of Toronto, 2017.

[180] ARM, "The ARM Cortex-A9 processors," Sep. 2007.

[181] Altera, *Nios II Classic Processor Reference Guide*, 2016.

[182] ——, *Designing with Low-Level Primitives User Guide*, 2007.

[183] J. R. Hauser, "Berkeley softfloat," http://www.jhauser.us/arithmetic/SoftFloat.html.