ABSTRACT

CHOUDHARY, NIKET KUMAR. A Synthesizable HDL Model for Out-of-Order Superscalar Processors. (Under the direction of Associate Professor Eric Rotenberg).

Many contemporary servers, personal and laptop computers, and even cell phones are powered by high-performance superscalar processors. In the past, conventional microarchitecture and technology scaling has afforded leaps in their performance and functionality. Today, conventional microarchitecture and technology scaling are both yielding lower returns with increasing costs. Therefore, any microarchitecture level decision to increase performance needs to be critically analyzed from a technology standpoint. To address this critical need, we have developed a register transfer level (RTL) model of a superscalar microarchitecture with similar complexity of a current generation processor. The RTL model is written in Verilog and is fully synthesizable. The model can be implemented in different technology nodes using a well established ASIC design flow to provide high fidelity estimation of propagation delay, power consumption, area, and other technology related costs. The RTL model is supplemented with a register file compiler to estimate the costs of multi-ported memory structures which are extensively used in a superscalar microarchitecture. The RTL model is also tightly integrated with a C++ functional simulator to assist and accelerate verification.

A Synthesizable HDL Model for Out-of-Order Superscalar Processors

by
Niket Kumar Choudhary

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2009

APPROVED BY:

Dr. Eric Rotenberg Committee Chair	Dr. Gregory Byrd
Dr. W. Rhett Davis	

DEDICATION

to my parents…

BIOGRAPHY

Niket Kumar Choudhary was born in November 1982 in Patna, India. He received the bachelor degree in Information & Communication Technology from Dhirubhai Ambani Institute of Information and communication Technology (DAIICT), India in May, 2005. During his undergraduate he worked as an intern at Cadence Design System, Bangalore in the logic synthesis group. After graduating from DAIICT, he held an engineering position at ARM Private Ltd, Bangalore (Aug, 2005-Jul, 2007) in the processor division. At ARM, he worked on low power processor design and characterization of ARM processors for performance, power and area on different process technologies.

Currently, Niket is a graduate student in Computer Engineering at North Carolina State University (NCSU), under the guidance of Dr. Eric Rotenberg. His research interests broadly lie in Computer Architecture and VLSI Design. He is also affiliated with Center for Efficient, Scalable and Reliable Computing (CESR) at NCSU.

ACKNOWLEDGMENTS

Foremost, I would like to thank my family for supporting me and encouraging me to obtain a good education throughout my life. I want to express my gratitude to Dr. Amit Bhatt, my undergraduate advisor, for inspiring me to pursue higher studies. Without his continuous encouragement, my journey to NCSU would not have been possible.

It has been a privilege to work with Dr. Eric Rotenberg, my advisor and a wonderful researcher. He has supported me in numerous ways, and has helped me throughout the masters program. It has been a rich learning experience working with him in the last two years.

I would like to thank my CESR and NCSU colleagues for giving their input and feedback to my thesis work: Hashem Hashemi, Abhishek Dhanotia, Shivam Priyadarshi, Muawya Al-Otoom, Elliott Forbes, Sandeep Navada, Salil Wadhavkar, Mark Dechene, Tanmay Shah, Jayneel Gandhi, Siddharth Chhabra, and Devesh Tiwari.

At last, I would like to thank my wife, Tulika, who slipped into my life during the last few months of my MS.

This thesis was supported in part by NSF grant No. CCF-0811707, Intel, and IBM. Any opinions, findings, and conclusions or recommendations expressed herein are those of the author and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Overview

Superscalar processors are at the heart of many high-performance computing platforms, either in the uniprocessor form or as processing cores in recently evolved chip multiprocessors. The superscalar microarchitecture exploits instruction-level parallelism (ILP) available in a program, by executing multiple instructions in parallel. To extract ILP, the superscalar microarchitecture forms a dynamic instruction window, and its instruction scheduler selects independent instructions out of the window for execution in a cycle. The dynamic instruction window is a segment of the dynamic instruction stream, that the processor can work upon concurrently. For issuing more than one instruction in a cycle, the scheduler goes beyond the sequential program order to find independent instructions, bringing out-of-order nature into the execution. In the dynamic instruction stream shown in Figure 1-1, there are data dependencies among the underlined instructions, and they need to be executed serially for a correct program. But the rest of the instructions in the window are independent, and can be issued for execution in parallel. The maximum number of issued instructions depends on the available functional units. Overlapped execution of multiple instructions in parallel leads to higher instructions per cycle (IPC), one key metric for the processor performance.

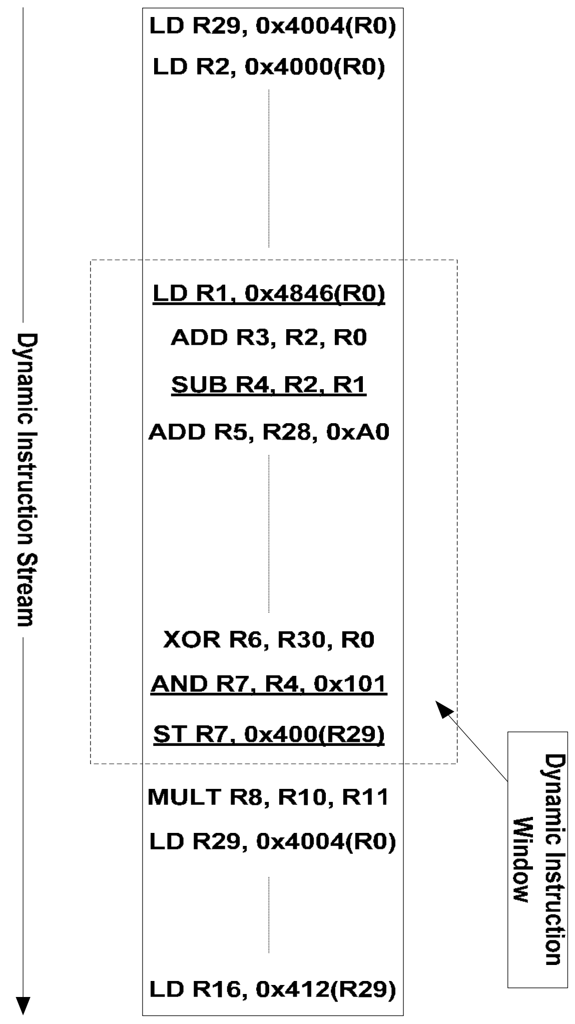
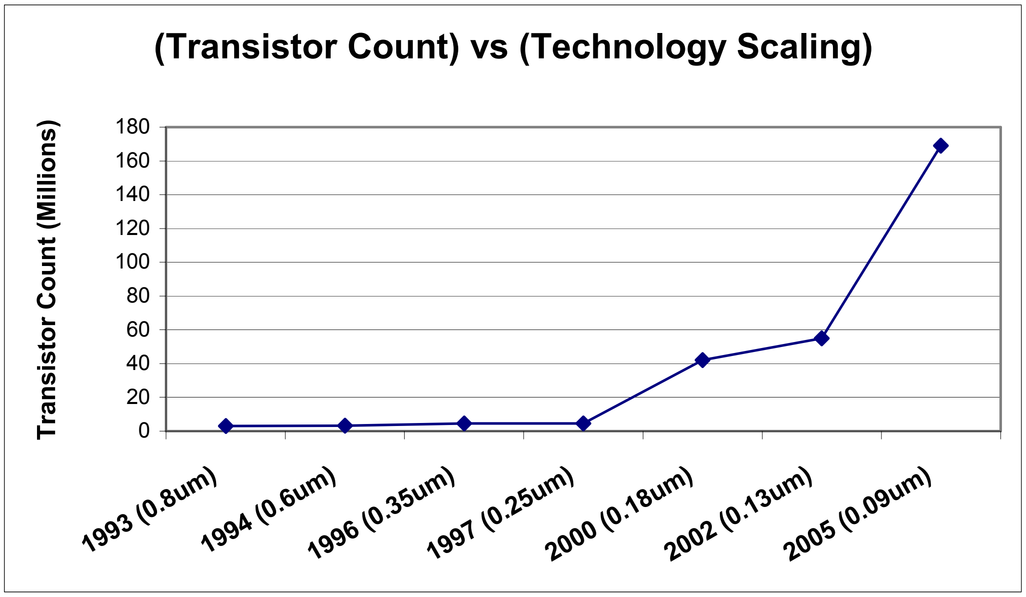


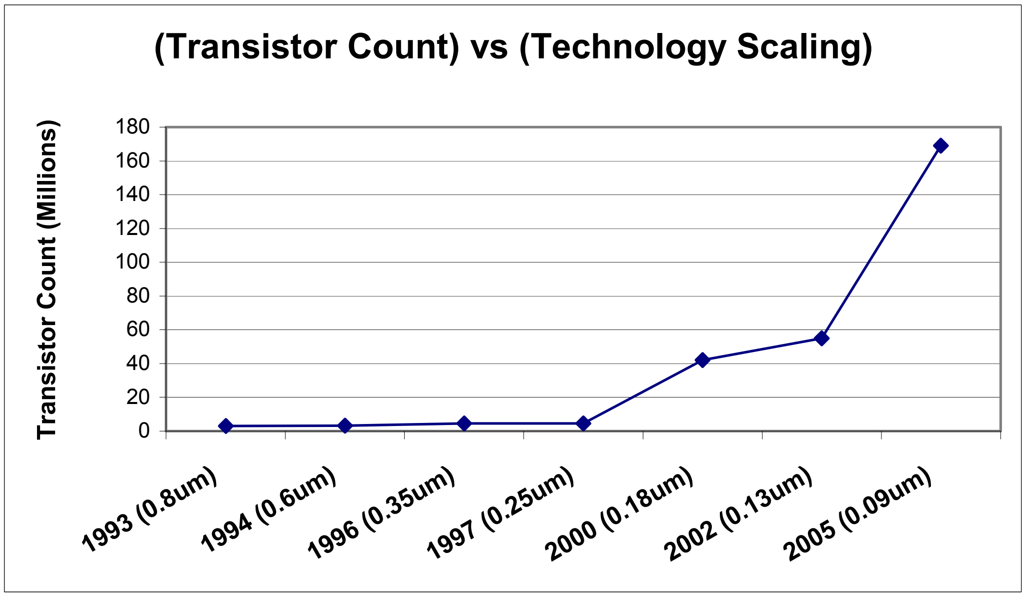
Figure 1-1: Out-of-order superscalar processing.

The amount of available ILP varies depending on the application, input set, and more. It is evident from Figure 1-1, that a larger window and uninterrupted instruction stream are required to achieve higher IPC. Achieving a larger window and an uninterrupted instruction stream requires complex hardware techniques leading to increased logic complexity. The increased logic complexity has a direct impact on the clock frequency, the second key metric for processor performance. Pipelining, coupled with better circuits, is used to manage increased hardware complexity and still achieve a high clock frequency for a given process technology. At the same time, deepening the pipeline may negatively impact IPC because of higher branch misprediction penalties. In practice, both IPC and clock frequency are strongly intertwined, and for a good design, it is important to achieve a good balance between IPC and clock frequency boosting mechanisms.

In the past, processors have experienced exponential growth in performance, owing to innovative microarchitecture and technology scaling. Technology scaling has been the performance growth enabler, providing abundant transistors to add more hardware for complex microarchitectures, and still leaving room to optimize clock frequency because of faster transistors. Figure 1-2 shows clock frequency and transistor integration growth over a decade of technology scaling for the Pentium family of Intel processors [1]. With easy to extract ILP and the bolstering of process technology, designers were able to scale the instruction window and pipeline depth to yield more performance, while still remaining within desirable area and power budgets.



(a)



(b)

Figure 1-2: Impact of technology scaling on the Pentium family of Intel Processors (a) Transistor integration growth (b) and clock frequency scaling. Clock frequency scaling also owes to the deepening of the pipeline over the generations of Pentium processors.

But recently, the performance gained by conventional microarchitecture and technology scaling has hit a wall. Forming a larger instruction window to extract additional ILP is incurring much higher frequency, power, and area costs [2]. Technology scaling trends are compounding the problem: transistor and wire speed and/or power are not scaling as well as in the past. *As a result, any microarchitecture level decision to increase performance needs to be critically analyzed from a technology standpoint.* Technology constraint-aware design necessitates a tool chain that can model all of the design costs of a superscalar microarchitecture or can evaluate the cost of any technique added to the baseline microarchitecture to enhance performance. These costs are clock frequency, power consumption, and die area, but may even include design effort, hard or soft error vulnerability, yield, etc. Cycle based simulators, predominantly used in academic research currently, fail to capture the detailed costs of the design.

To bridge the gap between high level microarchitecture simulation and the associated low level costs to implement the microarchitecture, we have developed a register transfer level (RTL) model of a superscalar microarchitecture with similar complexity of a current generation processor. The RTL model is designed in Verilog, and is fully synthesizable. The model can be implemented in different technology nodes using a well established ASIC design flow [7] to provide high fidelity estimation of clock frequency, power consumption, area, and other technology related metrics. In its current form, the RTL model captures the complexity of canonical stages in a superscalar processor in detail, and it is parameterized by the number of pipeline ways and the sizes of Random Access Memories (RAMs), Content Address Memories (CAMs), and other specialized memories within the stage. As part of future work, the RTL model needs to be more rigorously verified, and needs to be extended for sub-pipelining within canonical stages. The RTL model is supplemented with a register file compiler to estimate the cost of multi-ported RAMs and First-In First-Out (FIFO) structures which are extensively used in a superscalar microarchitecture. The RTL model is also tightly integrated with a high level functional simulator (written in C++) through the Verilog Procedural Interface (VPI), providing a Verilog and C++ co-simulation environment. VPI allows the RTL model to invoke the functional simulator, and then exchange information with it [35] [36]. The RTL model leverages the functional simulator to load a compiled binary, giving the Verilog simulator the flexibility to simulate any standard application benchmark. Moreover, the co-simulation environment can also be used for functional verification by cross-comparing retired instruction results.

The RTL model is the forerunner and basis of a bigger research project, FabScalar [6], which our group is currently working on. FabScalar, a novel toolset, can be used to compose synthesizable Verilog model of arbitrary superscalar processors. FabScalar is a first step toward the practical development of heterogeneous multi-core systems [3] [4] [5] and application-specific superscalar processors [33]. It exploits the fact that different superscalar processors have in common a canonical pipeline, shown in Figure 1-3, and differ primarily in the complexity and sub-pipelining within each canonical pipeline stage.

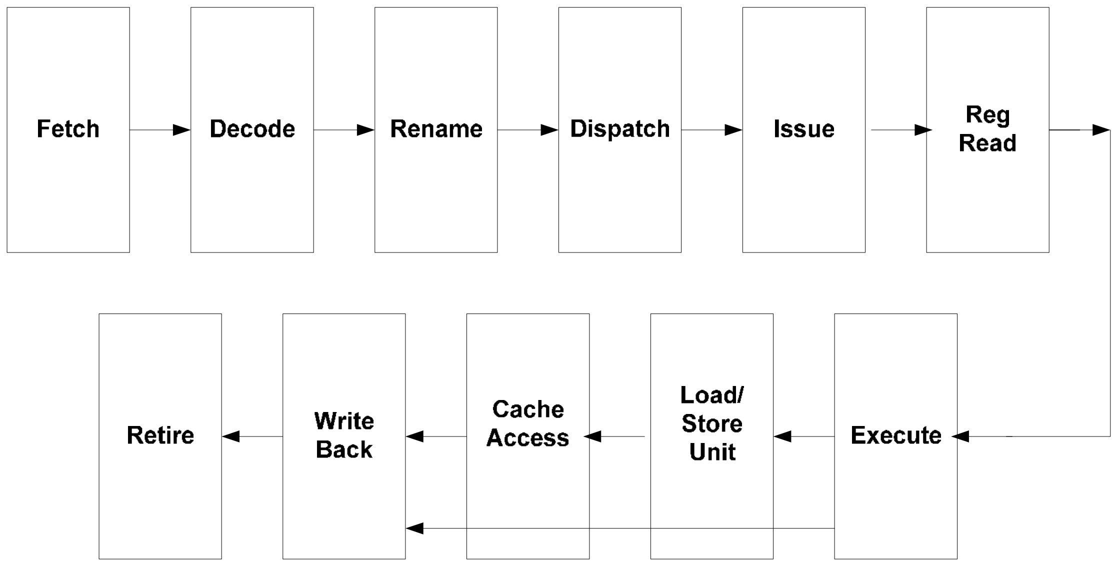


Figure 1-3: Canonical pipeline stages of a superscalar processor.

At the same time, it borrows the notion of a standard cell library for ASIC design where the standard cell library provides many flavors of simple gates, MSI components (e.g., MUXes, decoders), and even LSI/VLSI components (e.g., microprocessor cores). The FabScalar's standard superscalar library (SSL) will contain different flavors of canonical pipeline stages representing different complexity and can be used by an automated tool to assemble a superscalar processor based on microarchitecture constraints and low level cost constraints. More details about the FabScalar project can be found in [6].

1.2 Contributions

This thesis makes the following key contributions and co-contributions:

- Design and implementation of a fully synthesizable and semi-parameterized superscalar processor in Verilog RTL. The model provides the foundation from which FabScalar's SSL [6] will ultimately be derived.
- Design of a register file compiler to estimate the access time, power consumption, and area of multi-ported RAMs and FIFO structures. These structures are very specialized to the superscalar microarchitecture. The author has initially advised and collaborated with fellow student Tanmay Shah on this co-contribution.
- Design and implementation of a Verilog and C++ co-simulation environment to enable running any standard performance evaluation benchmarks on the RTL model and accelerate debugging the RTL model.

1.3 Organization

Chapter 2 describes the design and implementation of a four-wide eleven-stage-deep superscalar processor, forming the initial basis of FabScalar. Designs of individual canonical pipeline stages are discussed in detail. Chapter 3 summarizes the design of the register file compiler specialized to generate multi-ported SRAMs, co-developed with Tanmay Shah. Chapter 4 describes the Verilog and C++ co-simulation environment and how it facilitates a high fidelity verification of the processor design. Chapter 5 discusses logic synthesis and place&route results of different pipeline stages and the processor as a whole.

1.4 Related Work

The Illinois Verilog Model (IVM) [12] is closest to our Verilog model. IVM is a superscalar out-of-order processor designed in Verilog-95 at the University of Illinois by S.J. Patel's group for fault-tolerance research. IVM has twelve pipeline stages and implements a subset of the Alpha ISA. Detailed IVM microarchitecture features and parameters can be found in [12]. The Verilog model is not fully synthesizable because some parts of the design contains un-synthesizable behavioral code, for instance, while/for loops not evaluating to a constant in *missqueue.v*, shown in Figure 1-4.

```
i=0;

while ((i<QUEUE_SIZE) && !(q_valid_f[i] && q_done_f[i] && !q_type_f[i]))
                i=i+1;
```

Figure 1-4: An un-synthesizable construct from the IVM design

Even after fixing un-synthesizable constructs, the IVM synthesizes at a very low frequency attributing to its unoptimized design or coding style. Moreover, if IVM is to be used as a cycle-accurate simulator, it does not provide any support to run SPEC [13] or any standard benchmark suite, which is necessary to evaluate any new microarchitecture technique.

Sun's OpenSparc T1 [14] is an open source Verilog model of UltraSparc T1 and it implements the 64-bit SPARC V9 architecture. OpenSparc T1 is a CMP and has eight homogeneous processors on the same die. Each processor is an in-order, six-stage and scalar pipeline, and has hardware support to run four threads.

OpenRISC 1200 (OR1200) is a freely available Verilog model of processor available from OpenCores [15], and it implements the 32-bit ORBIS32 architecture. OR1200 is a five-stage, in-order, and scalar pipeline.

Although, OpenSparc T1 and OR1200 are freely available Verilog model of processors, they do not represent the complexity of an out-of-order and superscalar microarchitecture.

In the past, several analytical methods to estimate design costs have been proposed. Palacharla et al. [32] analyzed the complexity of key pipeline stages in a superscalar processor, and propose first-order analytical models for estimating their delays. The authors use SPICE simulation for quantifying the delay of timing critical paths in each pipeline stage. Brooks et al proposed Wattch [40], a framework to analyze and optimize power dissipation at the architecture level. Cacti is an analytical tool for modeling the access time, dynamic power, leakage power, and area of caches and other memories [41]. Bazeghi et al. [42] proposed an analytical approach to measure and estimate processor design effort. Although, analytical models or tools based on analytical models might give good early estimation of the associated design costs, the purpose of our RTL model and FabScalar is to provide high fidelity design costs, which become necessary as performance scaling is increasingly costly and technology dependent.

Kumar, Tullsen, and Jouppi [3] [4] and Strozek and Brooks [43] have done groundbreaking research on architectural exploration for heterogeneous CMPs. Strozek and Brooks' work on the high level synthesis of very simple cores for embedded systems [43] is more directly related to FabScalar itself. The Program-In-Chip-Out (PICO) framework out of HP labs [44] is closely related in that it customizes VLIW cores and non-programmable accelerators for embedded applications. Similarly, Tensilica's Xtensa processor generator customizes data path and VLIW cores for embedded applications [45]. FabScalar is distinct in that it targets complex superscalar processors and this is evident in the novel composable SSL.

CHAPTER 2

Design of a Superscalar Out-of-Order Processor

The execution time of an application for a given input set and a given ISA can be defined as:

Execution Time = (Instruction Count x Cycle Time)/(Instructions Per Cycle) [39]

where instruction count is the total number of instructions to be executed for the given application and input set, cycle time is the clock period at which the processor can run, and instructions per cycle is the average number of instructions executed each cycle. Better algorithms at the application level and better compiler optimizations may reduce the instruction count. For a technology node, deeper pipelining and better circuits are used to reduce cycle time, i.e., increase clock frequency. Complex hardware techniques are employed in a processor to increase instruction per cycle (IPC).

A superscalar microarchitecture attempts to achieve higher IPC by processing more than one instruction (also referred to as superscalar width) in each pipeline stage, as opposed to a scalar microarchitecture, which processes only a single instruction in each stage. Moreover, to exploit concurrency in the program, instructions are executed out-of-order. With out-of-order execution, the dynamic instruction stream is no longer executed strictly in the original program order, but based on the availability of source operands. Increasing the width of each stage and executing instructions out-of-order lead to increased logic complexity, which directly impacts clock frequency. To accommodate the increased complexity and/or to increase clock frequency further, the microarchitecture employs pipelining. The number of pipeline stages from fetching an instruction to retiring it, is referred to as the depth of the microarchitecture. All else equal the deeper the pipeline, the less amount of logic there is in a pipeline stage. The pipeline structure in a superscalar processor is logically partitioned into a front-end and back-end, where the front-end processes the instruction stream in the original program order, and the back-end processes instructions in an out-of-order way to extract ILP. Although instructions execute out of program order, they update the processor state in the original program order, to preserve the *sequential execution contract* between the program binary and hardware [28]. A conventional out-of-order superscalar uses a FIFO-like structure to reconstruct the program order, and the completed instructions retire (also referred to as graduate or commit) from the FIFO to update the architectural state of the machine. In some microarchitectures, the size of this FIFO restricts the maximum number of instructions the processor can concurrently work upon, also referred to as the instruction window. To extract peak parallelism of the processor, it is necessary to keep the instruction window full all of the time.

The design of a superscalar microarchitecture tries to find the right balance between IPC and clock frequency to execute the instruction stream as fast as possible. The IPC of a superscalar processor is fundamentally limited by control dependencies and data dependencies within the instruction stream, and the clock frequency is limited by the underlying technology. Control dependencies may force the program to change its course of execution, leading to complete or partial flushing of the instruction window [57]. As shown in Figure 2-1, I4 is a branch instruction and based on its outcome (taken or not-taken), either I40 or I5 would be the next instruction to execute. The outcome of a branch instruction is unknown until it executes, and the deeper the pipeline the longer it takes to resolve a branch instruction. To alleviate the impact of control dependencies on performance, the processor employs a dynamic branch predictor to speculate on the outcome of the branch very early in the pipeline, and speculatively fetch and execute the predicted path accordingly. A dynamic branch predictor predicts the outcome of a specific branch based on its own history and possibly its correlation with the outcome of previous branches [60] [61] [62].

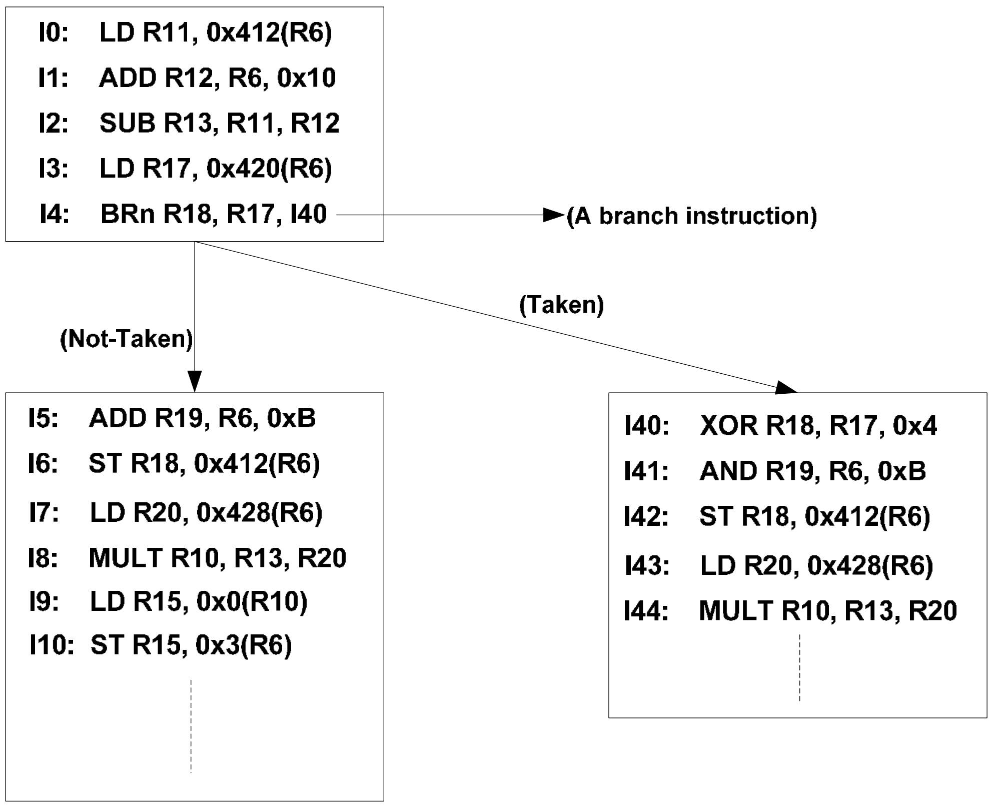


Figure 2-1: Control-flow in a program.

Data dependencies among instructions force their serial execution, leading to decreased parallelism to exploit [58] [59]. In a dynamic instruction stream, data dependencies among instructions occur through either architectural registers or memory, as shown in Figure 2-2. The solid arrows show dependencies propagated through architectural registers and the dashed arrow shows a dependency propagated through a memory location. A data dependency arises from the fact that an instruction's source operand depends on the outcome of another instruction, leading to a producer and consumer relation between the instructions. In Figure 2-2, I2 cannot execute until I0 and I1 have executed; similarly, I8 cannot execute until I0, I1, I2, and I7 have executed. The dependencies propagated through memory are detected late in the pipeline, as the addresses of memory instructions are not known until they execute.

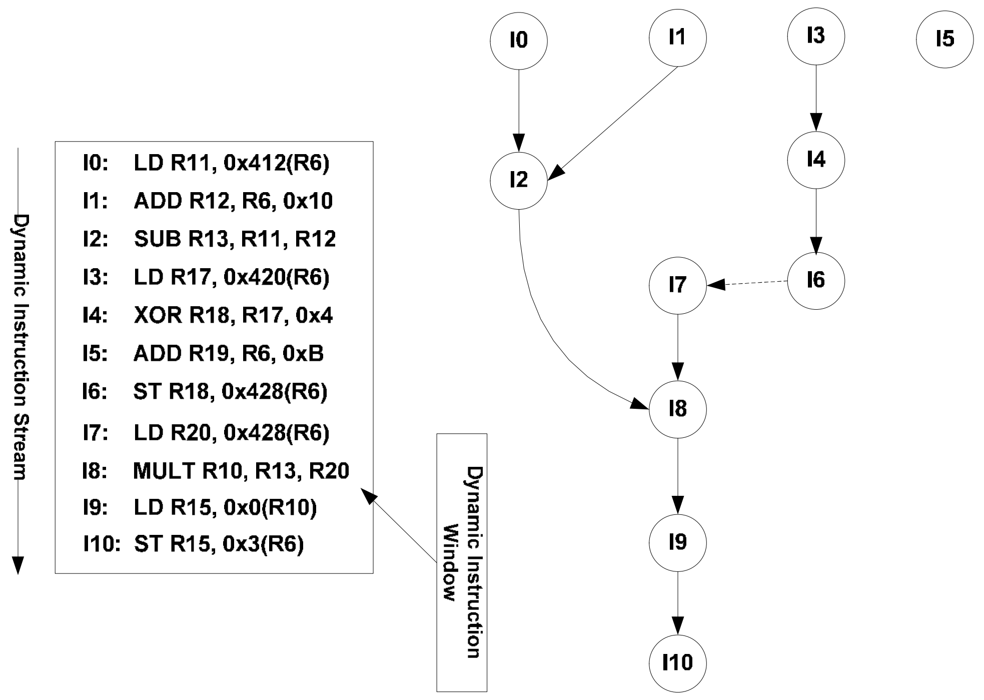


Figure 2-2: Data dependence in the dynamic instruction stream.

To mitigate the impact of data dependencies on the IPC, a superscalar microarchitecture uses complex hardware techniques, for example, speculative wakeup of consumer instructions [63] and memory dependence predictors [64].

A superscalar microarchitecture achieves higher clock frequency by pipelining and better process technology [65]. Over the past decade, technology scaling drastically increased the transistor speed and the number of transistors that can be integrated on a single die, by scaling device dimension, threshold voltage, gate oxide thickness, and supply voltage [37] [38]. Moreover, better circuits and logic families have evolved to supplement the performance growth by technology scaling.

As part of this thesis, we design and implement the canonical pipeline stages of a superscalar microarchitecture, found in most of the commercial superscalar based designs [23] [24]. Although, the RTL model of the individual pipeline stage is parameterized by the width of the stage and the sizes of specialized memory structures within the stage, we choose a specific microarchitecture configuration as a starting point to understand the design complexity involved. Table 2-1 shows the chosen microarchitecture configuration.

Table 2-1: Microarchitectural configuration of the synthesizable-verilog model.

Stage	**Description**
Fetch	4-wide, 512-entry BTB, 128-entry bimodal branch predictor, 8-entry RAS, 16-instruction fetch buffer
Decode	4-wide, ISA = SimpleScalar (MIPS-like)
Rename	4-wide, 32-entry rename map table with 8 read and 4 write ports, 4 shadow map tables (checkpoints)
Dispatch	4-wide
Issue	4-wide issue, 32-entry issue queue
Register Read	4-wide, 128-entry physical register file with 8 read ports and 4 write ports
Execute	1 simple ALU, 1 complex ALU, 1 branch ALU, 1 AGEN + 1 port to load-store unit
Load-Store Unit	16-entry load queue, 16-entry store queue
Writeback	4-wide
Retire	4-wide, 128-entry active list with 4 read and 4 write ports, arch. map table with 4 read and 4 write ports

The rest of the chapter will discuss in detail the design and implementation of a 4-wide out-of-order superscalar microarchitecture.

2.1 Methodology

The designed superscalar processor implements the SimpleScalar ISA [21], called PISA (for pseudo-ISA), a close derivative of the MIPS architecture [22]. PISA is 32-bit RISC architecture, although the instructions are 64 bits wide.

The designs of individual stages have evolved from reading papers in the literature and by using the superscalar knowledge expertise within our research group. Figure 2-3 shows the implementation flow used throughout the design process.

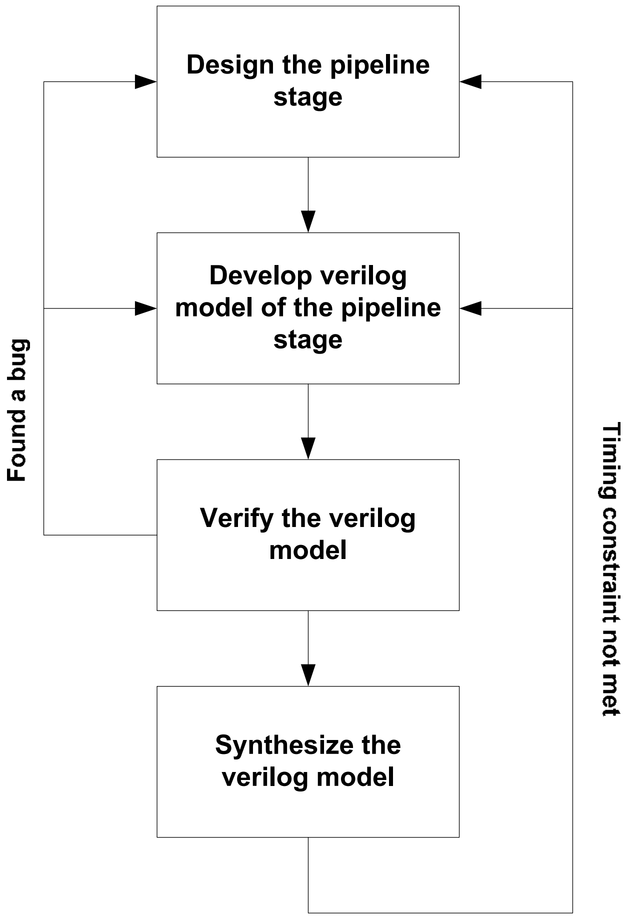

Design the pipeline stage
Develop verilog model of the pipeline stage
Verify the verilog model
Synthesize the verilog model
Found a bug
Timing constraint not met

Figure 2-3: Implementation flow of the superscalar processor.

A mix of the Verilog-95 and Verilog-2000 hardware description language (HDL) is used for designing the hardware of each pipeline stage. The Verilog is synthesizable and we built the Verilog model from scratch, instead of using existing models from IVM or OpenRISC. The microarchitecture extensively uses specialized multi-ported RAMs, CAMs and FIFOs. The Verilog modules containing memory elements are separated from the random logic in a modular way, so that they can be replaced by custom macros during logic synthesis or the rest of the implementation flow. Memories are modeled in RTL for the functional simulation, however. In the case of a memory structure requiring parallel access to multiple contiguous rows in the same cycle, it is implemented using interleaved banks to obviate the need of multiple read or write ports, as shown in Figure 2-4.

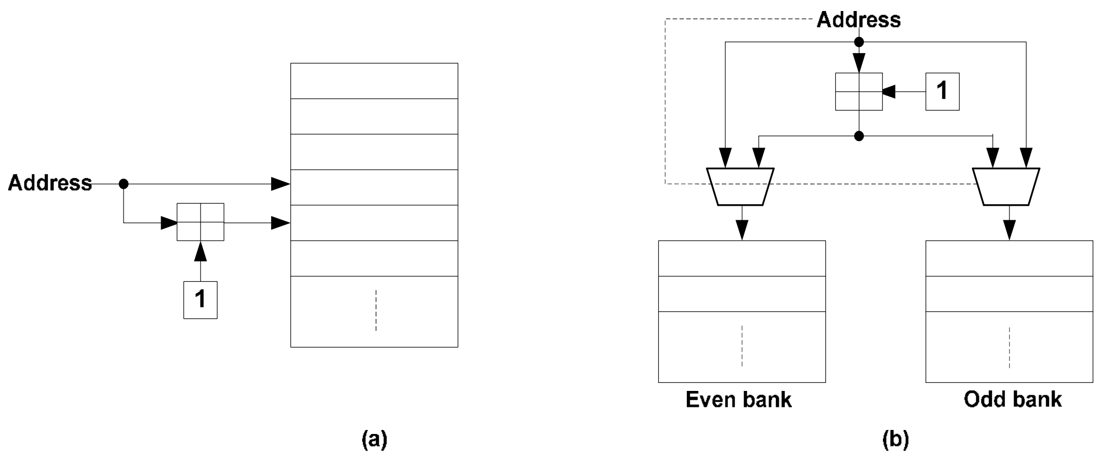


Figure 2-4: (a) Example of accessing two adjacent rows of a memory and (b) its interleaved implementation.

Table 2-2 shows the industry-standard EDA tools used for functional simulation, logic synthesis, and placement and routing. During the full cycle of development, we use a 45nm standard cell library [25] for the logic synthesis and the basic placement and routing. For this standard cell library, we set a target of 1GHz clock frequency for the design with the chosen microarchitecture configuration. Individual pipeline stages went through multiple design iterations to achieve the target frequency. The iterations primarily involved restructuring the logic, or in some cases changing the entire pipeline stage design.

Table 2-2: EDA tools used for the design.

Stage	**EDA Tool/Tools Used**
Functional Verification	Cadence NC-Verilog version: 06.20-s006
Logic Synthesis	Cadence RTL Compiler version: 07.10-s021_1, Synopsys Design Compiler version: X-2005.09-SP3
Place & Route	Cadence SoC Encounter version: 7.1

Describing intermediate design iterations is beyond the scope of this thesis, and we primarily document the final design employed for each pipeline stage.

2.2 Pipeline Stages of a Superscalar Processor

Figures 2-5 shows the high level block diagram of a superscalar processor. The individual pipeline stages are discussed in detail in later sections. The superscalar microarchitecture is logically partitioned into a front-end and a back-end (shown with a dashed line in Figure 2-5). The key features of the microarchitecture are

- Separate level-one (L1) caches for storing recently and frequently used program instructions and data.
- Dynamic branch predictor to speculate on the next instruction cache line to be fetched.
- Physical register file for register renaming and storing both committed and non-committed (speculative) instruction results.
- Out-of-order execution of the instruction stream from the issue queue.

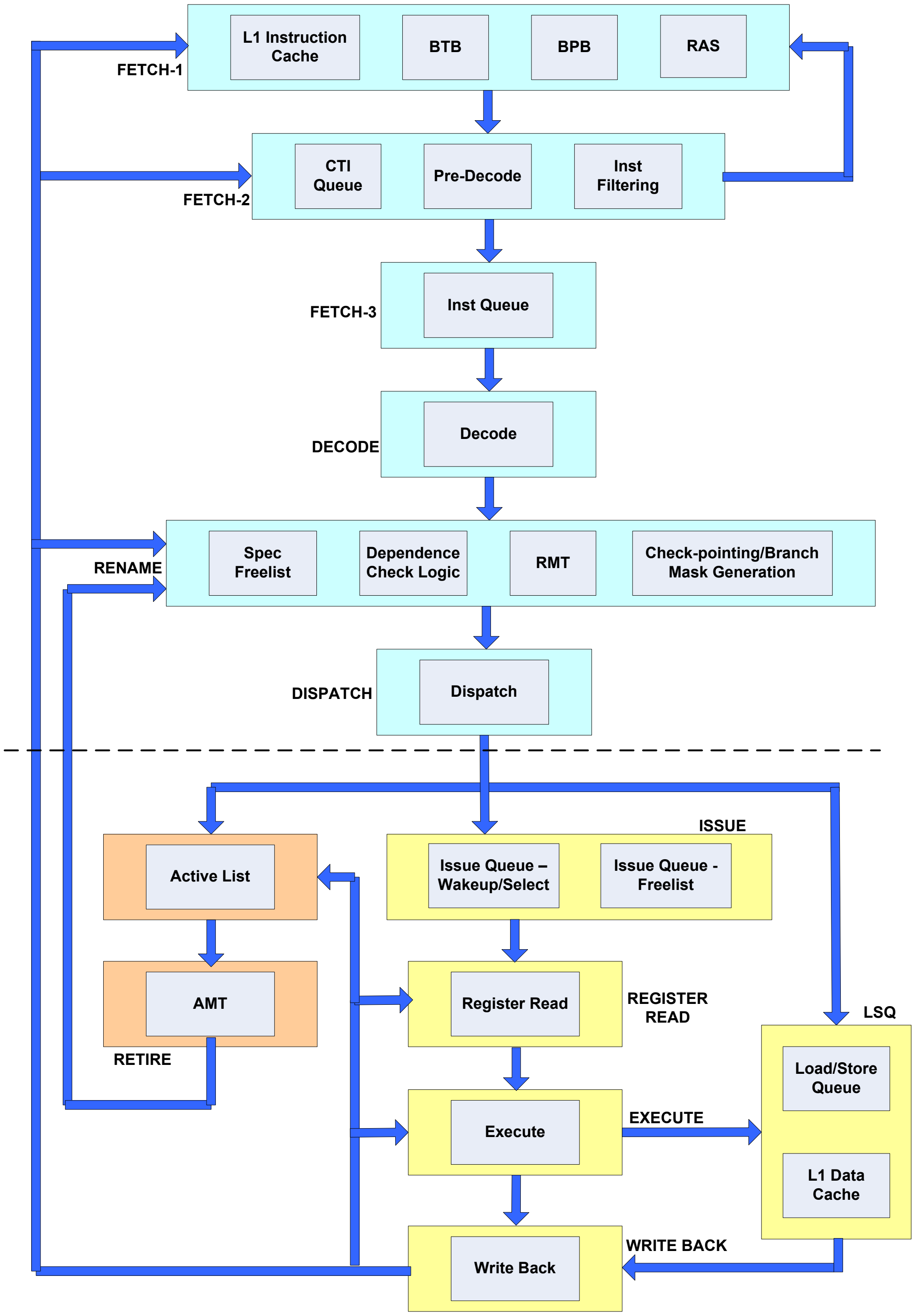


Figure 2-5: High level block diagram of a superscalar processor.

- Store and load queues for resolving within-window memory dependencies and for retiring stores to architectural memory state in program order.
- Active list for the in-order update of the processor's architectural state.
- Checkpoint-based mechanism for fast recovery from branch mispredictions.

2.2.1 Instruction Fetch

Instruction fetch is responsible for providing a continuous instruction stream to the rest of the pipeline. The program counter (PC) (or instruction pointer) in the Fetch-1 stage keeps track of the program address of the current or next instruction to be executed. Every cycle, the PC is incremented sequentially, until there is a control instruction in the instruction stream. Control instructions, for example, a jump direct/indirect, call direct/indirect, return, or conditional branch (if the direction of the branch is taken) change the PC non-sequentially. In a program, conditional branches tend to occur more frequently than other control instructions.

In our design, the fetch stage achieves a fetch bandwidth of four by employing a small but fast L1 instruction cache and a dynamic branch predictor to speculate on the outcome of a branch instruction in one cycle. The branch prediction mechanism is composed of three major hardware structures, along with random logic: branch target buffer (BTB), branch prediction buffer (BPB), and return address stack (RAS). The BTB records the PC of the control instruction, its type, and the associated target address. On an access to the BTB, it identifies if a PC is a control instruction and what its type is. The BTB is implemented as 4-

way interleaved SRAM for a fetch width of four, eliminating the need for a multiported SRAM. The BPB is a simple bi-modal branch predictor and is accessed using the low order bits of the PC. For a branch instruction's PC, the BPB predicts the direction of the branch (taken or not-taken). A call instruction has an associated return address, the sequential address after the call, and the RAS is used to predict this return address. The BTB has poor target address prediction accuracy for return instructions, due to the same function or subroutine being called from multiple call sites in a program.

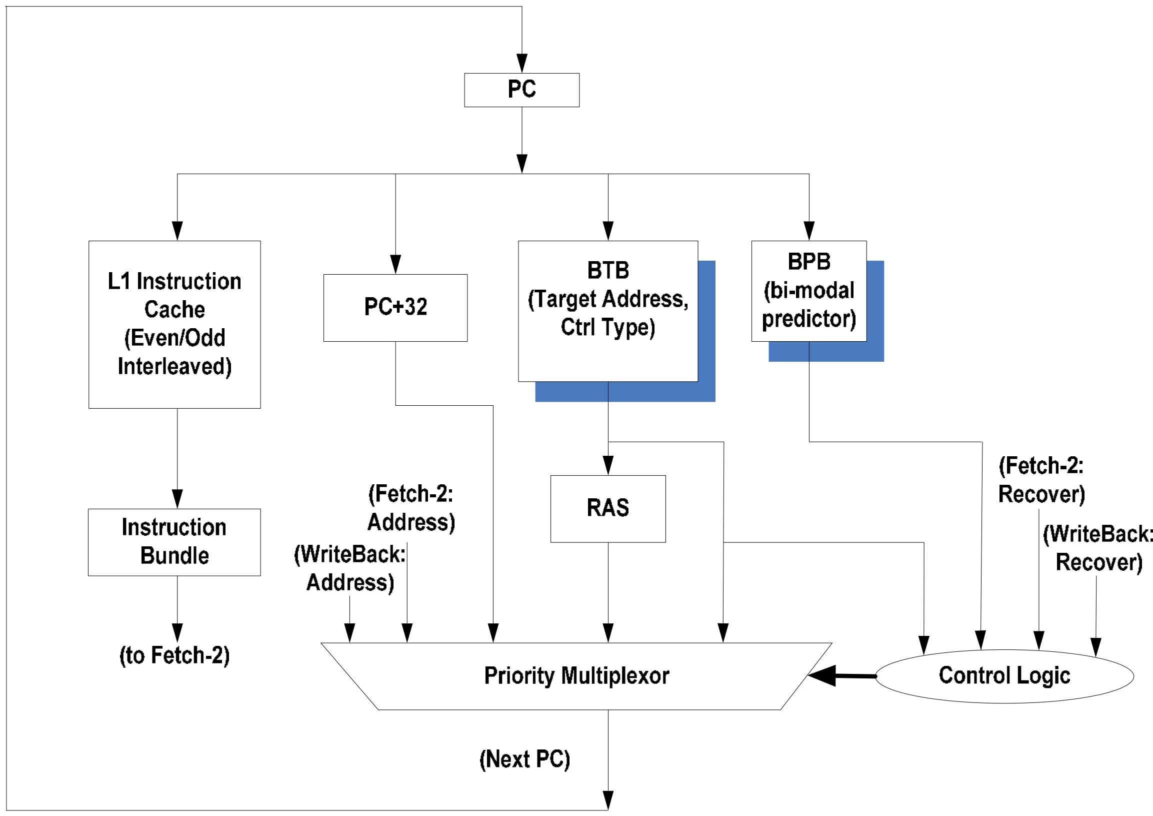


Figure 2-6: Fetch-1 Stage, highlighting the next-PC logic.

The instruction cache module in the RTL model is 2-way interleaved to obviate the need for a dual-port SRAM, guaranteeing two contiguous 4-instruction cache lines in a cycle, from which 4 sequential instructions can be extracted from any unaligned starting PC. One bank contains cache lines with even addresses and the other bank contains cache lines with odd addresses. The instruction cache can provide two adjacent aligned instruction blocks every cycle, if there is no cache miss. The BTB and BPB are accessed using four consecutive PCs, as four instructions are fetched every cycle, to feed the random logic to generate the PC for the next cycle (next PC logic in Figure 2-6). If the BTB indentifies a PC as the address of a control instruction, the target address for the next PC is obtained from either the BTB itself (not return) or RAS (return). Moreover, on every call instruction, the next sequential PC is pushed onto the RAS for predicting the target address of the corresponding return instruction in the future. In the case of multiple control instructions in the fetch block (composed of four instructions in this implementation), the target address of the first taken branch is given priority for the next PC. On a misprediction, the next PC logic recovers the correct target address by the later pipeline stages. We will discuss control mispredictions in detail, later in the chapter.

From a cycle time standpoint, the Fetch-1 stage has two important timing paths:

- Accessing the interleaved L1 instruction cache for reading two aligned cache blocks. The complexity of accessing the cache would increase with increasing the size and the set-associativity of the cache.
- Generating the next PC using information from the BTB, BPB, and RAS for a group of instructions being fetched. The complexity of the next PC logic would increase

with a larger BTB, a more complicated or larger branch predictor [27], or wider fetch bandwidth. Moreover, it is important to generate the next PC in one cycle to avoid losing cycles on every predicted-taken branch, unless a sophisticated and complicated fetch mechanism is employed, for example, Seznec's Multiple-Block Ahead Branch Predictors [26]. Figure 2-7 shows the timing critical logic to generate the next PC.

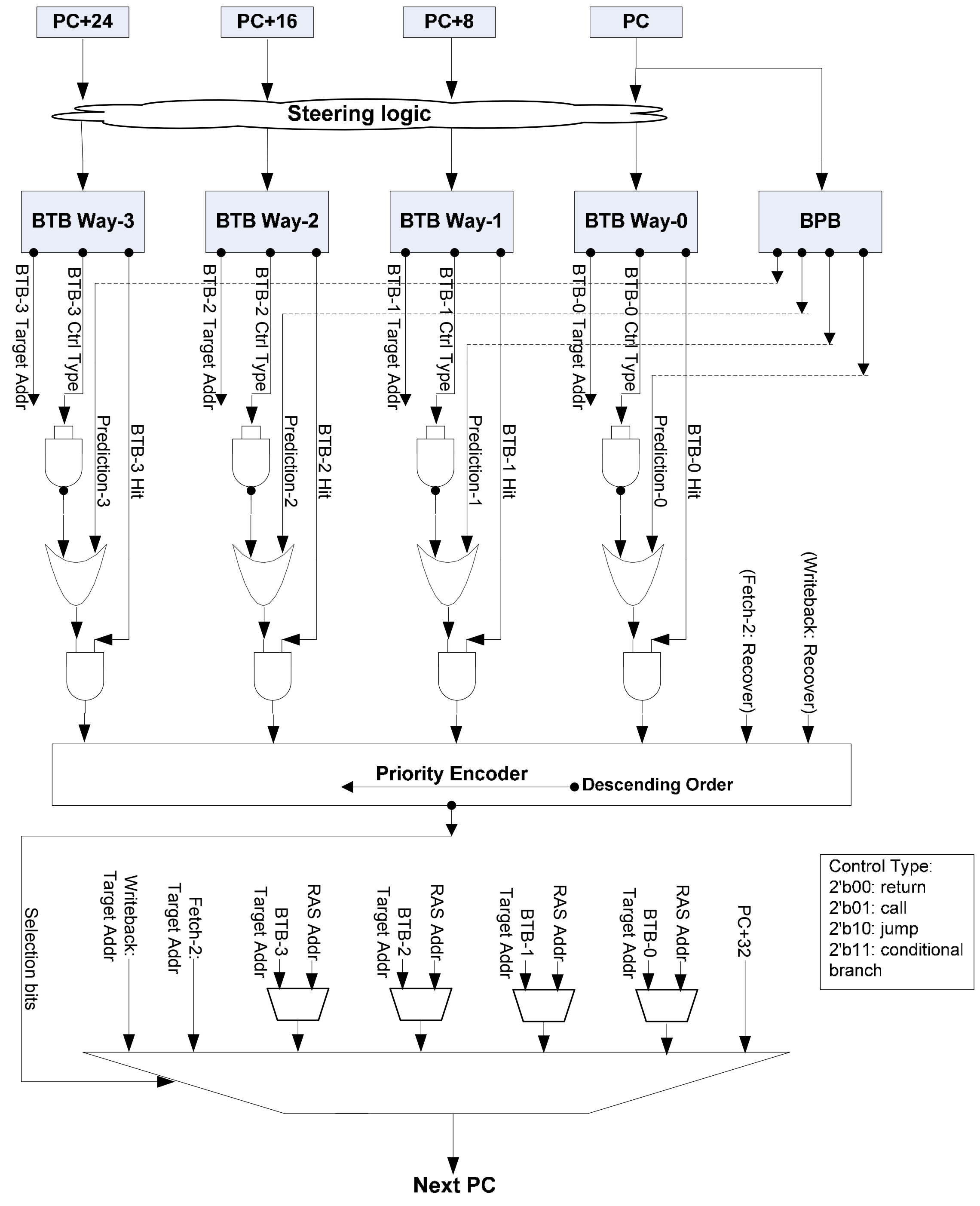


Figure 2-7: Timing critical logic to generate the next PC. In case of return type control instruction, the target address (RAS Addr) comes from RAS.

The Fetch-2 stage contains the instruction alignment logic and extracts up to four consecutive instructions (from among the two consecutive blocks coming from Fetch-1) based on the starting PC, or until the first taken branch, whichever comes first. Fetch-2 pre-decodes the four instructions to explicitly identify control instruction within the fetch block and calculate their target addresses. The PISA ISA has target offsets embedded in the control instruction except for returns and jump or call indirects. If the BTB misses for the control instruction in the previous cycle, Fetch-2 generates a recovery signal and recovery target address for the Fetch-1 stage. If an instruction happens to be a predicted-taken branch in the fetch block, subsequent instructions are discarded. Fetch-2 also contains a FIFO buffer, called the control-transfer instruction queue (CTI queue), to hold all control instructions in their program order. After a control instruction at the head of the FIFO retires, the CTI queue updates the BPB with the computed direction. This leads to in-order update of the branch prediction structure. Instruction alignment, extracting the fetch block, pre-decoding, and generating the recovery signal are serialized logic, and fall onto the timing-critical path of Fetch-2.

The Fetch-3 stage is an instruction queue, and decouples instruction fetching and the rest of the front-end pipeline stages. It receives up to four instructions from the Fetch-2 stage and writes them into a circular buffer at the tail pointer, and always four instructions are read from the head pointer to feed further pipeline stages. The instruction queue serves two purposes:

1. it allows instruction fetching, even though the rest of the front-end is stalled because of a hardware resource limitation, and
2. it simplifies the decode, rename and dispatch logic by always providing a fixed number of instructions (four per cycle).

Reading the circular FIFO for four instructions is the most timing-critical path in Fetch-3.

2.2.2 Instruction Decode

The instruction decode logic is straightforward to implement, due to implementing a RISC ISA. PISA has three instruction formats, as shown in Figure 2-8 (reproduced from [21]). Currently, our design only implements integer instructions and we intend to extend the design for floating-point instructions, as part of the FabScalar project. More details about PISA can be found in [21], some of the important features are:

- There are 32 architectural integer registers, explicitly addressed by integer instructions.
- An instruction has a maximum of 2 source operands and 1 destination register.

Each cycle, the decode stage may receive four instructions from the Fetch-3 stage. The decode logic extracts the opcode from each instruction, based on the instruction's format, and generates appropriate control signals that flow with the instruction downstream.

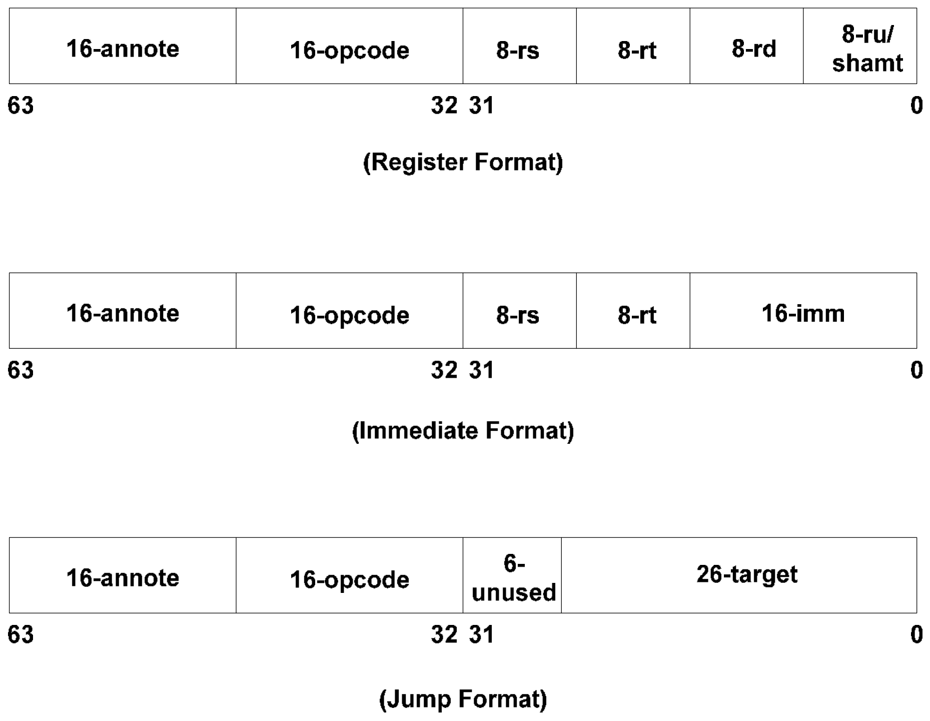


Figure 2-8: SimpleScalar ISA format [21].

2.2.3 Instruction Renaming

The rename stage of the superscalar processor renames the architectural source and destination registers to physical source and destination registers. The compiler is limited by the number of architectural registers for allocating intermediate computation or holding local variables, and the compiler reuses registers to overcome this. Register renaming removes the false dependencies among instructions which are artifacts of limited architectural registers. Fundamentally, a dynamic instruction stream has three types of data dependencies:

- True dependency, where the source register of a younger instruction depends on the outcome of another, older instruction in the dynamic instruction stream.

- Output dependency, where the destination register of a younger instruction is the same as the destination register of another, older instruction in the dynamic instruction stream.
- Anti-dependency, where the destination register of a younger instruction is the same as the source register of another, older instruction in the dynamic instruction stream.

Output and anti-dependencies arise because of limited architectural registers, and are sometimes also referred to as false dependencies. Register renaming eliminates false dependencies by mapping the architectural destination register of each in-flight instruction to a unique physical register. Our design uses a physical register file to hold committed and non-committed (speculative) register values. The physical register file is bigger than the architectural register file, and determines the maximum number of un-committed instructions that are in-flight in the pipeline.

Figure 2-9 shows the register renaming logic for 4-way renaming. A circular FIFO, referred to as the Free List, contains the unused physical registers, and a physical destination register is obtained for an instruction with an architectural destination register by popping a free physical register from the Free List. The Rename Map Table (RMT) maintains the physical registers to which architectural registers are currently mapped. Accordingly, each architectural source register of the instruction is renamed to a physical source register by looking up its mapping in the RMT. After renaming an instruction's source registers, its new architectural-to-physical destination register mapping is updated in the RMT for future instructions to observe. At the same time, true dependencies between source registers and preceding destination registers must be checked for the group of instructions being renamed concurrently.

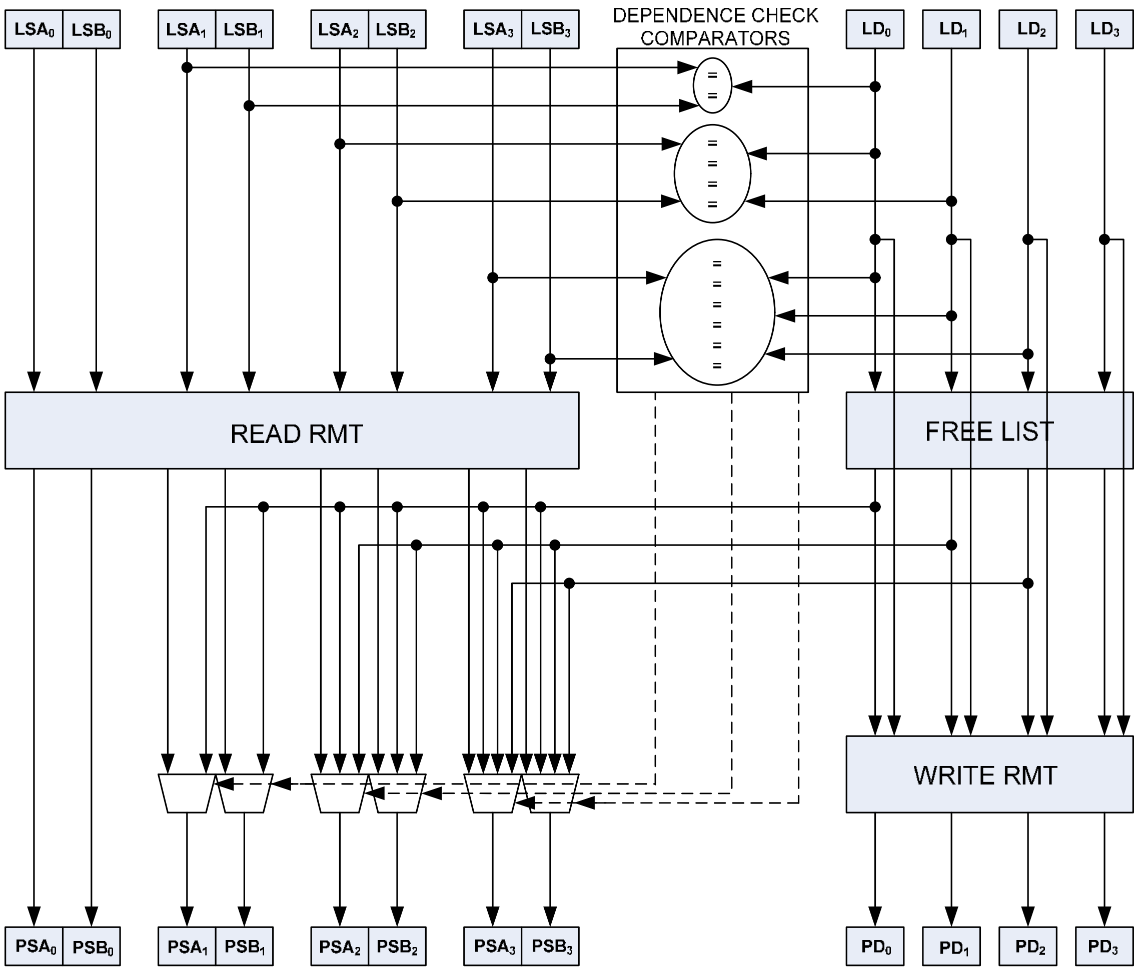


Figure 2-9: Register renaming logic. LSAn and LSBn are the architectural source registers and LDn is the architectural destination register of instruction N. PSAn and PSBn correspond to physical source registers, and PDn corresponds to the physical destination register.

For a 4-way rename stage, renaming is performed for eight source registers and four destination registers in parallel. Physical names for four destination registers are obtained by popping registers from the Free List and are updated in the RMT in the same cycle. Note that, if there are multiple producers of the same architectural registers in the rename group, then only the youngest producer updates the RMT (logic not shown in Figure 2.9). The RMT is implemented as an SRAM, with 8-read ports and 4-write ports, and the Free List is implemented as an interleaved FIFO. True dependencies among source registers and preceding destination registers in the same rename group are also resolved using comparator and multiplexer logic. The access latency of reading the multi-ported RMT plus the bypass MUXes following the RMT makes it the most timing critical logic in the rename stage.

2.2.4 Dispatch

The Dispatch stage is the boundary between in-order instruction processing and out-of-order instruction processing. It is the responsibility of the Dispatch logic to check for available space in the back-end pipeline stages, in particular, the Active List, Issue Queue, and the Load and Store Queues, for newly renamed instructions. If the space is available, the Dispatch stage writes the new instructions in the respective resources (shown in Figure 2-10). In case of the unavailability of enough space in these resources, the dispatch stage generates a stall signal for the decode and rename stages.

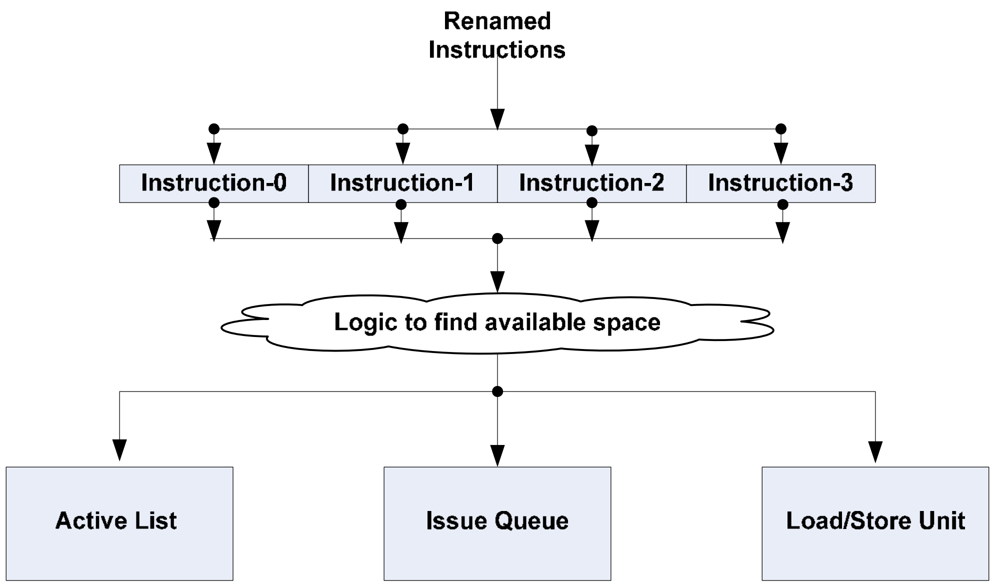


Figure 2-10: Dispatched instructions inserted in the back-end resources.

2.2.5 Issue

The Issue stage is the heart of out-of-order superscalar execution, and is very critical to the performance of a superscalar microarchitecture. The Issue stage buffers the renamed instructions and selects instructions for execution based on the availability of their source operands. The maximum buffer size is referred to as the issue window, and the maximum number of instructions selected for parallel execution in a cycle is referred to as the issue width. The issue window and issue width are the fundamental characteristics of the issue stage, and determine its logic complexity. An instruction in the issue window is ready to execute if all of its source operands are ready, and can participate in the selection process. Since there is a limited number of functional units, and multiple instructions may be ready in the issue window, selection logic is required. The selected instruction is issued for execution, removing the instruction from the issue window. Since an instruction is selected, not in the program order, but based on the availability of its source operands, it leads to out-of-order execution. The issued instruction broadcasts its physical destination register name or *tag* to the issue window to wakeup its dependent instructions. In case of a match, the dependent instruction sets the ready bit associated with its source operand.

In summary, an Issue stage consists of two major operations: wakeup and select. The wakeup operation is dependence resolution performed in the issue window, and the select operation is arbitrating among ready-to-execute instructions in the issue window. In our design, the issue window is centralized, and the Issue stage is pipelined between wakeup and select logic. A maximum of four instructions can be selected for execution on four parallel but distinct functional units. Each functional unit executes a different type of integer instruction, and instructions are associated with their functional unit type during the decode stage. The wakeup logic tracks the availability of source operands based on the tags allocated to the operands. In our design, a tag is a physical register allocated to an architectural destination register during the rename stage. The Issue stage contains two key memory structures, a CAM for holding source operand tags and a RAM holding payload information for an instruction, for example, destination tag, program counter, Active List id, etc. Currently, we model the CAM as composed of synthesizable registers, although we intend to replace it with a custom-designed component as part of future work.

As shown in Figure 2-11, the result tags of issued instructions are broadcast through as many wakeup ports to all the instructions in the issue window, and each entry in the issue window compares its source tags with the broadcasted destination tags. On a match with any of the broadcasted tags, an instruction sets the corresponding ready bit for its source operand. The complexity of the wakeup operation grows with issue window size and the number of wakeup ports. In our design, the number of wakeup ports is the same as the issue width.

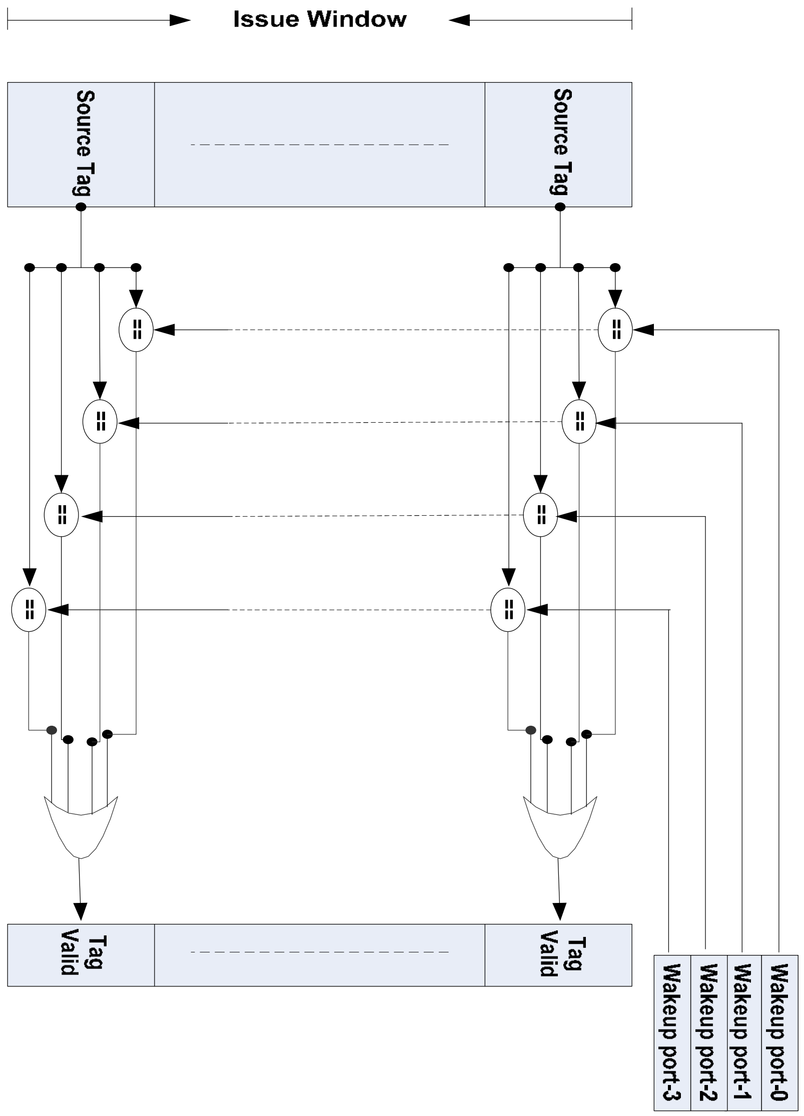


Figure 2-11: Wakeup logic in the Issue stage.

To ensure back-to-back execution of a producer instruction and its consumer instructions in consecutive cycles, an issued instruction broadcasts its result tag even before its completion, and the dependent instructions can read the result data from the bypass network (discussed in section 2.2.9). Currently, loads wakeup their consumers late, only when their data is actually produced. A load instruction may take a variable number of cycles to execute, depending on a data cache miss or memory disambiguation stall (discussed in section 2.2.8). Although the pessimistic approach to wake up load-dependent instructions is easier to implement, it will lead to significant IPC degradation due to data cache hits being the common case. We plan to fix this limitation in the future by speculatively waking up load-dependent instructions and replaying them in the case of load stalls.

As shown in Figure 2-12, we implement three-level tree-based selection logic to select ready-to-execute instructions in a cycle. There is separate selection logic for each function unit type. The implementation is very similar to the one discussed in [31]. An L1 request vector is formed for each function unit type using the source operand ready bits and other control bits. A bit set in the request vector indicates that the corresponding issue window entry is ready to execute. The L1 request vector is divided into multiple blocks, and the size of each block is four entries. Each block selects one ready entry based on the round-robin policy and passes the selection on to the L2 selection logic. Similarly, the L2 request vector is divided into blocks of size four, and each block selects one ready entry based on the round-robin policy.

Eventually, one ready instruction gets selected in the final level, and the instruction is issued after reading the payload information from the payload RAM.

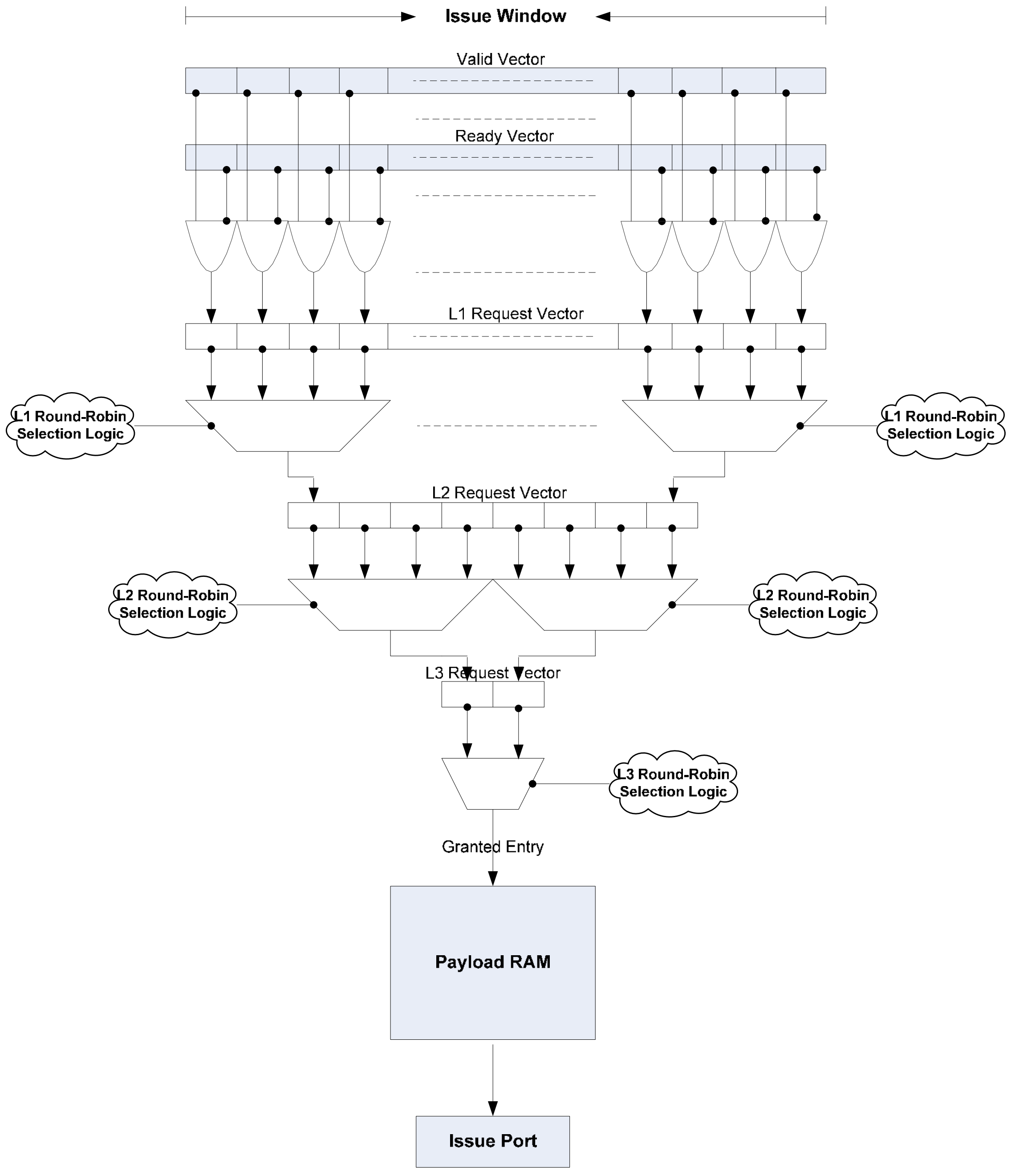

Issue Window
Valid Vector
Ready Vector
L1 Request Vector
L1 Round-Robin Selection Logic
L1 Round-Robin Selection Logic
L2 Request Vector
L2 Round-Robin Selection Logic
L2 Round-Robin Selection Logic
L3 Request Vector
L3 Round-Robin Selection Logic
Granted Entry
Payload RAM
Issue Port

Figure 2-12: Select Logic in the Issue stage.

2.2.6 Register Read

The register read stage contains the physical register file (PRF), which holds all the committed and non-committed instruction results. The source register specifiers of an issued instruction index into the PRF to read the corresponding values, as shown in Figure 2-13. At the same time, source register specifiers are also compared with the Writeback destination register specifiers to detect the scenario whereby a producer instruction's result needs to be directly bypassed to a consumer instruction. In case of a match, the instruction ignores the data read from the PRF and uses the data from the bypass network. The bypass network updates the PRF with the produced value.

The PRF is implemented as an SRAM. For an issue width of four, 8-read and 4-write ports are required. Reading the PRF coupled with MUXes after the PRF falls on the timing critical path for the register read stage. The bypass network is comprised of parallel result buses, originating from the Writeback stage.

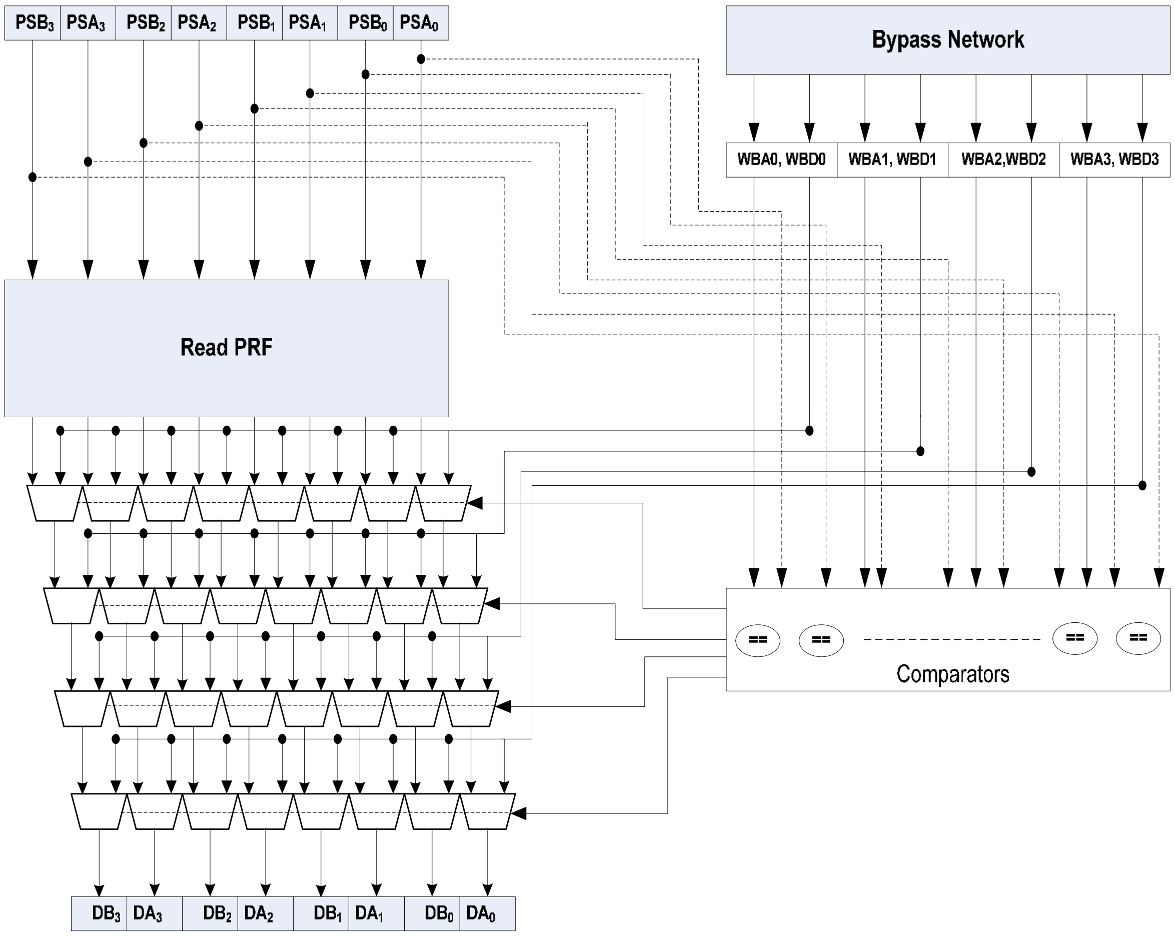


Figure 2-13: Register read stage. PSAn and PSBn correspond to physical source registers of an instruction N. WBAn and WBDn correspond to write back address and data respectively. DAn and DBn correspond to final source operands' data.

2.2.7 Execute

The functional unit in the execute stage performs an arithmetic or logic operation on the source operands of an instruction, and the result of the operation is written into the Writeback latches. As shown in Figure 2-14, we implement four functional units, where each unit executes a different class of integer instructions. The *Simple ALU* performs simpler arithmetic and logic operations, for example, addition, subtraction, xor, etc. and these operations take a single cycle to execute. The *Complex ALU* performs complicated arithmetic operations, for example, multiply, divide, etc. and these operations take multiple cycles to execute. In this implementation, the *Complex ALU* takes three cycles to execute an instruction and is fully pipelined. The *Control ALU* executes control instructions, for example, conditional branches, jumps, calls, etc. and these instructions take one cycle. A dedicated functional unit for control instructions facilitates early resolving of conditional branches. The *AGEN* unit performs address computations for memory operations, i.e, loads and stores. The output of *AGEN* goes to the Load/Store Unit, discussed in Section 2.2.8.

The source operands for the functional units come either from the Register Read stage or from the bypass network.

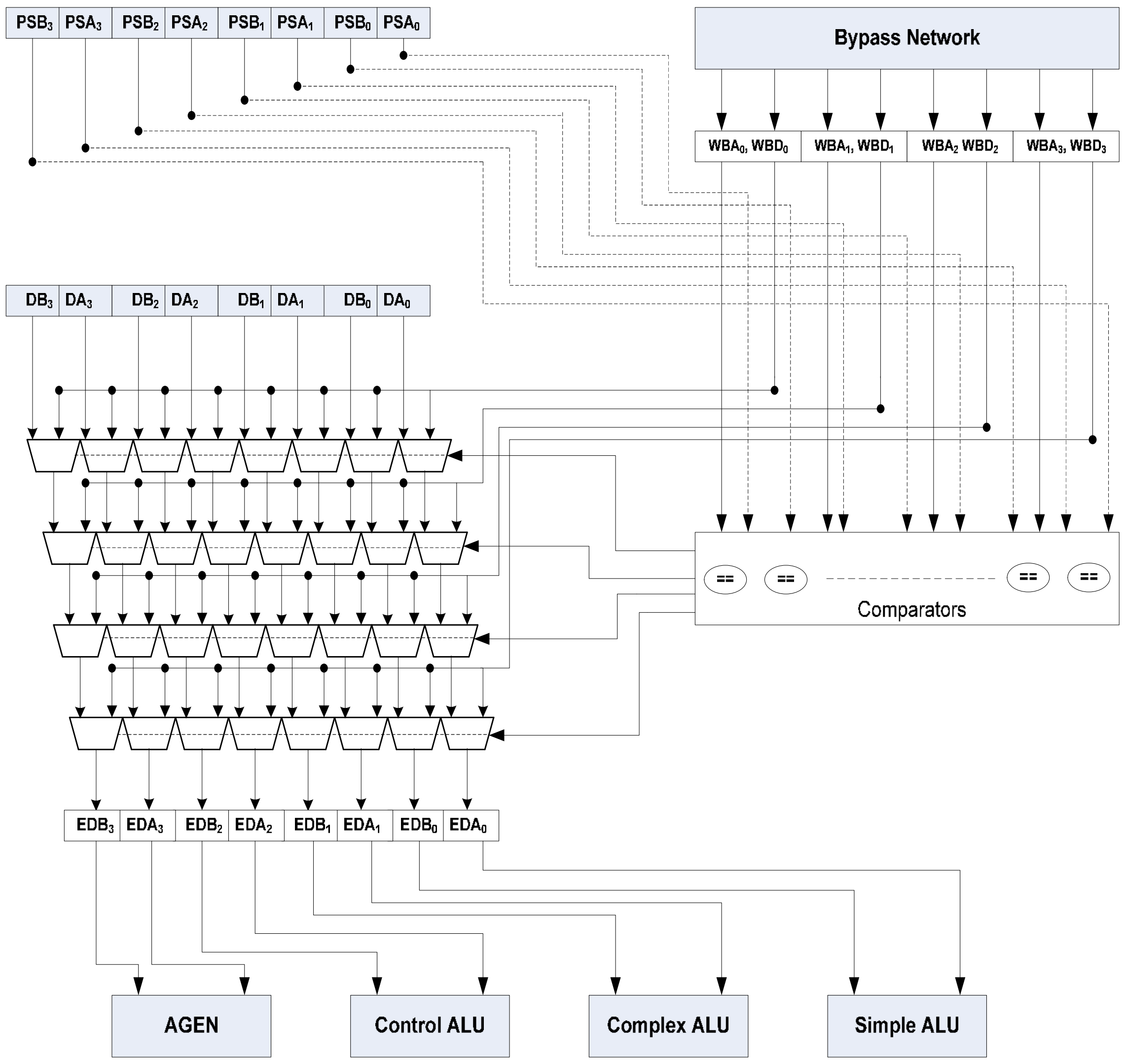

PSB3 PSA3 PSB2 PSA2 PSB1 PSA1 PSB0 PSA0
Bypass Network
WBA0, WBD0
WBA1, WBD1
WBA2 WBD2
WBA3, WBD3
DB3 DA3 DB2 DA2 DB1 DA1 DB0 DA0
==
Comparators
EDB3 EDA3 EDB2 EDA2 EDB1 EDA1 EDB0 EDA0
AGEN
Control ALU
Complex ALU
Simple ALU

Figure 2-14: Execute stage with four functional units. PSAn and PSBn correspond to physical source registers of an instruction N. WBAn and WBDn correspond to write back address and data, respectively. DAn and DBn correspond to the source operands obtained in the previous cycle from the Register Read stage. EDAn and EDBn correspond to final source data feeding functional units.

2.2.8 Load/Store Unit (LSU)

Data dependencies propagated through architectural registers are static in nature, i.e., the addresses (register specifiers) are embedded within the instruction itself. Dependencies propagated through memory are unknown until they execute, as the loads and stores typically use register operands to calculate their addresses. A modern microprocessor employs a special address dependence check mechanism to support correct out-of-order execution of loads and stores. A load compares its address with all the uncommitted stores older in the program order, and in case a store's address matches the load address, the store forwards its data to the load. Moreover, all the stores should update the architectural memory state in program order.

We implement a separate load queue (LQ) and store queue (SQ) to maintain the uncommitted memory operations in their program order. The LQ and SQ insert the loads and stores, respectively, when they are dispatched. An issued load takes at least two cycles to execute; in the first cycle the load's address is computed by the AGEN unit and in the following cycle the load goes through an address dependency check mechanism in the Load/Store Unit (LSU). The LSU logic performs associative searches to resolve address dependencies (also referred to as load disambiguation) and employs store-to-load data forwarding logic. A load might find its data from the data cache or the store queue depending on the outcome of the load disambiguation logic. The access to the data cache happens in parallel with the load disambiguation logic.

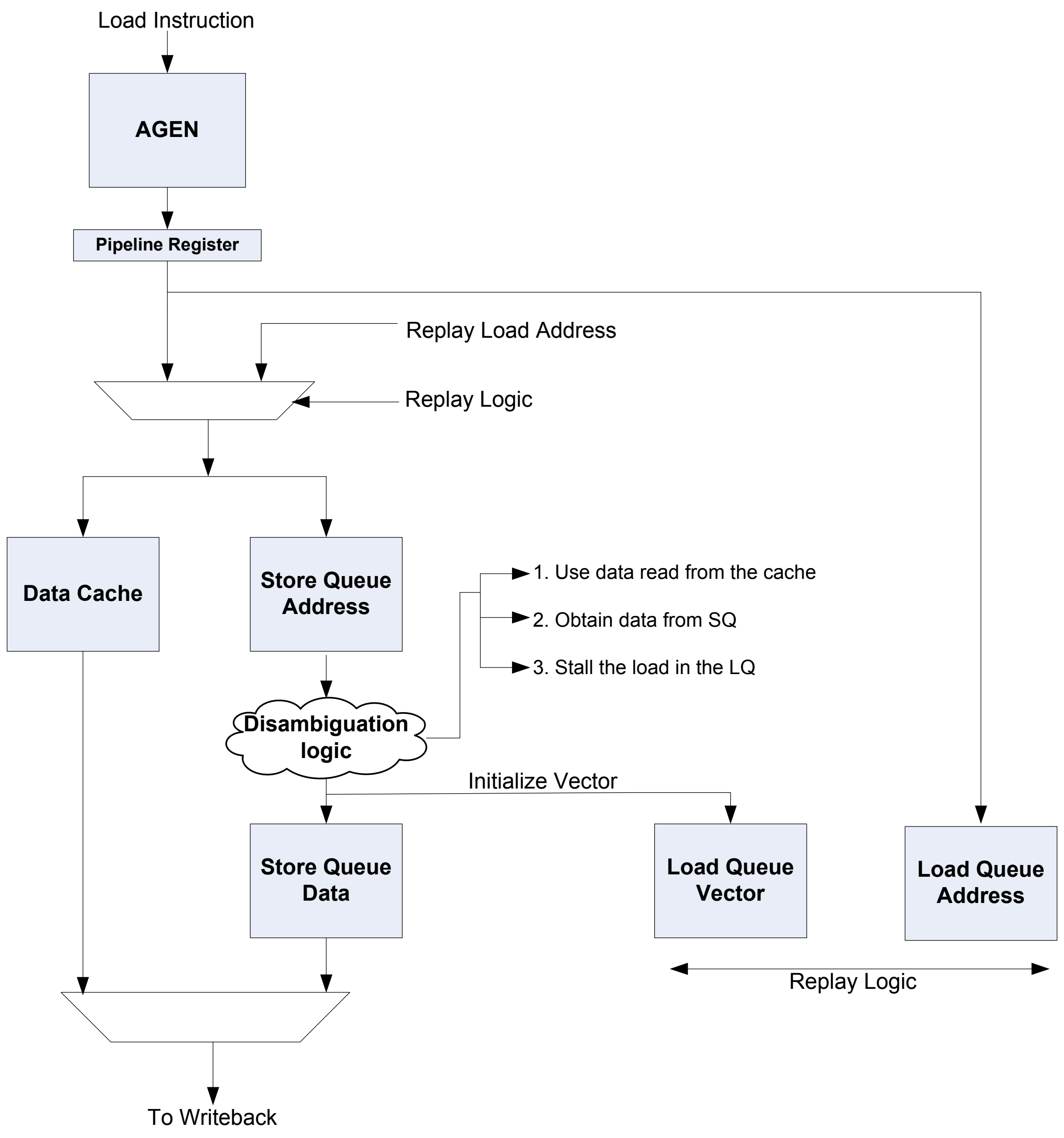


Figure 2-15: Load hit path in the Load/Store Unit.

Figure 2-15 shows the load hit path and different outcomes of the load disambiguation logic:

1. There are no *unknown stores* (stores who have not computed address yet) or *conflicting stores* (stores whose addresses match that of the load) prior to the load. In this case the load should use the data read from the data cache.
2. There are conflicting or unknown stores prior to the load.
    a. If the nearest store among these is conflicting, store forward its data to the load (load obtains data from SQ data).
    b. If the nearest store among these is unknown, stall the load.

When a store's address is computed, it broadcasts its address to any disambiguation-stalled loads. A load waiting on this store is re-injected into the load hit path. A store commits its value to the memory when it is at the head of the active list. The path shown in Figure 2-15 is the most timing critical in the LSU.

2.2.9 Writeback

The Writeback stage contains the latches holding the results from the execute stage, which serve as the source for feeding the bypass network. The bypass network forwards the result values from the executed instructions to the dependent instructions, to support optimal execution of the producer and its dependent instructions in consecutive cycles. The instructions in the Register Read stage and Execute stage compare their source register specifiers with all the destination register specifiers on the bypass network for matches, and in the case of a match, the instruction uses the result value from the bypass network. The Writeback stage also acts as the source for branch misprediction signals.

From an implementation standpoint, the bypass network is essentially parallel buses running from the Writeback stage to Register Read and Execute stages (shown in Figure 2-16) with each wire observing a load of comparator and MUX logic, and is very critical to the timing because of increased wire delays with technology scaling.

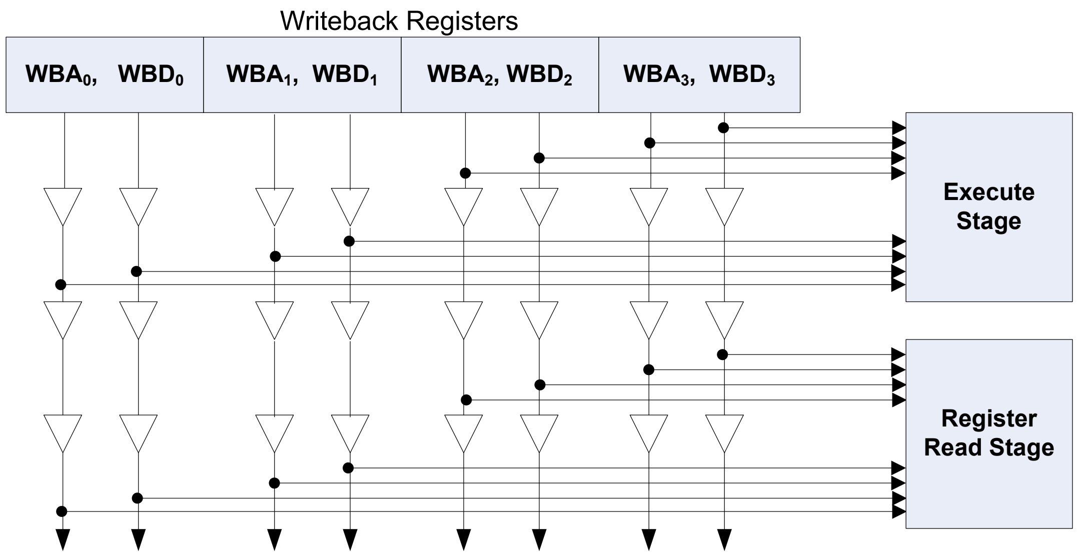


Figure 2-16: Writeback registers and bypass network. WBAn and WBDn correspond to write back address and data, respectively.

2.2.10 Retire

Although instructions execute out-of-order, they update the architectural processor state in the correct program order to maintain the sequential execution model. The in-order commit of instructions naturally leads to the implementation of precise interrupts [29].

The Retire stage maintains the program order among instructions using a circular FIFO with head and tail pointers, referred to as the Active List or Reorder Buffer. The dispatched instructions are inserted into the Active List at the tail pointer, giving each instruction a unique entry into the Active List. Upon execution of an instruction, the Writeback stage updates the completed bit in the Active List entry for this instruction. The Retire stage also maintains an Architectural Map Table (AMT), containing mappings between architectural registers and physical registers for committed versions of architectural registers. The Active List keeps probing the completed bits for the entries starting from the head pointer, and any completed instructions at the head are committed and removed from the Active List. When an instruction commits, the Active List updates the AMT with the instruction's physical destination register mapping and releases the previously mapped physical register. The released physical register gets added to the Free List. In the case of a store instruction, the Active List signals the Store Queue to commit the store data to memory. Figure 2-17 shows the retirement operation of an instruction. It takes two cycles for the complete retirement operation: in the first cycle the head of the Active List is read and in the following cycle, the AMT, Free List, and Store Queue are updated with appropriate information.

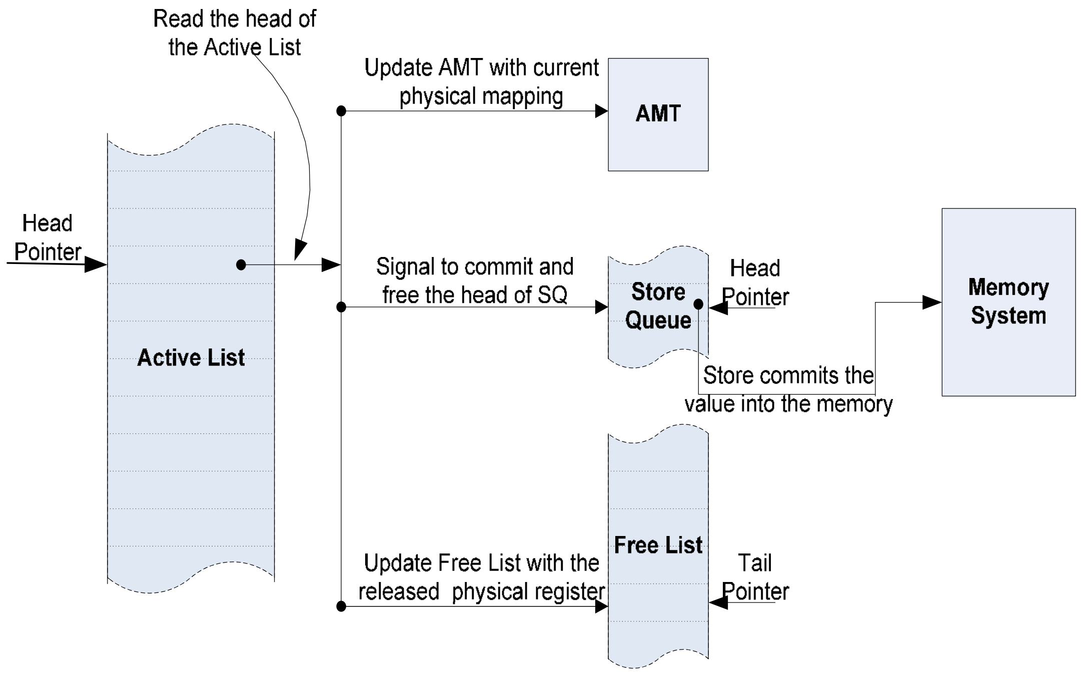

Read the head of the Active List
Update AMT with current physical mapping
AMT
Head Pointer
Active List
Signal to commit and free the head of SQ
Store Queue
Head Pointer
Memory System
Store commits the value into the memory
Update Free List with the released physical register
Free List
Tail Pointer

Figure 2-17: Retirement operation of an instruction.

2.2.11 Branch Misprediction Recovery

Branch mispredictions are a major source of performance degradation in a superscalar processor. On every misprediction, clock cycles are wasted to 1) while waiting for the branch to execute, 2) while flushing the pipeline, and 3) while refilling the pipeline, deteriorating IPC as the processor is doing no useful work during these three phases of recovery. Moreover, the deeper the pipeline, the bigger the penalty a misprediction will incur. In fact, Sprangle et al. [19] identified branch mispredictions as the single largest contributor to performance degradation as pipelines are deepened.

For a fast mechanism to restore the processor to a known state after a misprediction, we implement checkpointing [30] of the RMT as well as the branch mask logic [31]. A checkpoint mechanism makes a copy of the RMT and the Free List head pointer when a branch instruction is encountered. A copy of the RMT is also referred to as a Shadow Map Table (SMT), and each branch instruction carries the associated SMT id. The branch mask logic is similar to that of the MIPS R10000 [31], and it is used to indicate the pending branches an instruction depends on. On detecting a branch misprediction, the front-end pipelines stages are completely flushed, and the branch mask is used for selective removal of instructions in the back-end pipeline stages, i.e., removing only those instructions that are after the branch in program order. The PC is set to the correct target address and the RMT is quickly restored from the associated SMT of the mispredicted branch. If a branch resolves correctly, the associated SMT is released and the branch mask associated with each instruction is cleared of that branch.

Although fast to recover, the checkpoint mechanism is expensive in terms of die area and power consumption. Currently, we allow four unresolved branches in the out-of-order pipeline stages, requiring four checkpoints and a 4-bit branch mask. Figure 2-18 shows the checkpointing logic in the Rename stage. The Branch Vector maintains a list of the occupied SMTs, i.e., the branches associated with the occupied SMTs have not executed yet. If all SMTs are being used, the processor will keep renaming instructions until it encounters a branch instruction, which must stall dispatch until an SMT becomes available.

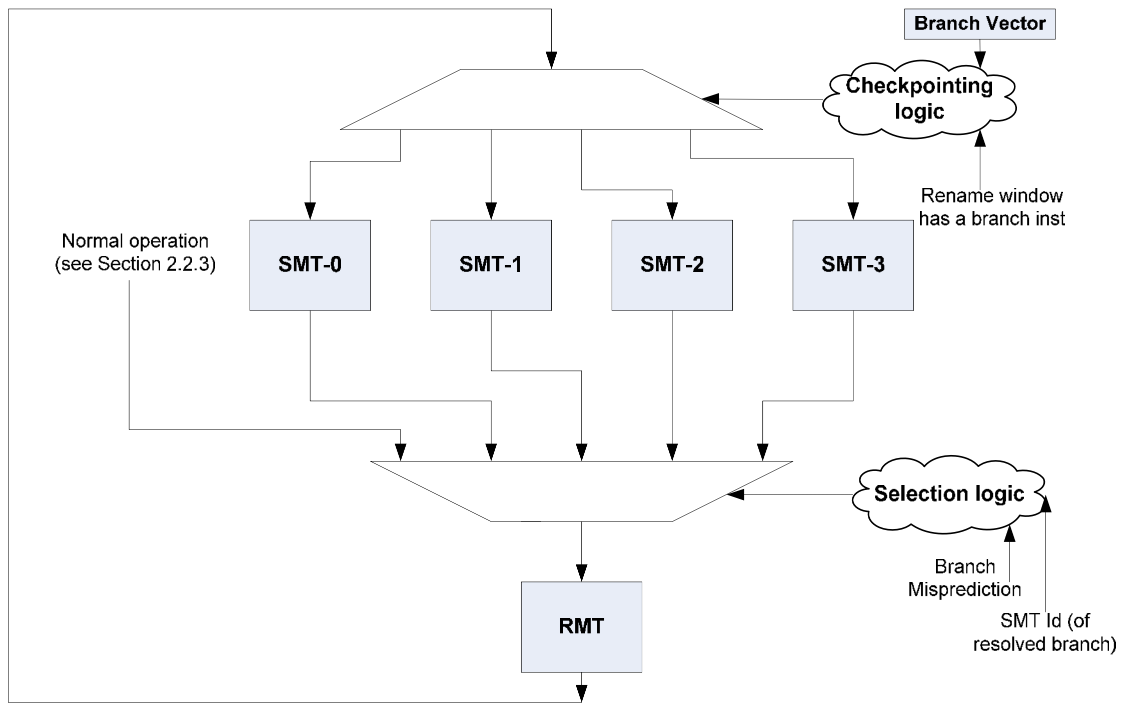

Branch Vector
Checkpointing logic
Rename window has a branch inst
Normal operation (see Section 2.2.3)
SMT-0
SMT-1
SMT-2
SMT-3
Selection logic
Branch Misprediction
SMT Id (of resolved branch)
RMT

Figure 2-18: Checkpointing logic in the Rename stage.

CHAPTER 3

Register File Compiler

In a modern processor design, small, yet complex, memory structures are implemented as custom memories. These specialized memory structures play an important role in determining the performance and the power budget of a microprocessor design, as they most often contribute to the timing-critical and frequently-exercised paths in a pipeline stage. A register file for storing intermediate computation in a processor is a classic example of a specialized memory [39]. Furthermore, a superscalar microarchitecture, which processes more than one instruction in a pipeline stage per cycle, gives rise to multi-ported memories to support multiple parallel reads and writes. For instance, the physical register file (PRF) forms an important part of the superscalar processor, and for high IPC, the source operands of all the issued instructions must be read simultaneously and all the result values from the bypass network should be written simultaneously [46] [47]. Similarly, for a centralized issue queue design, the read (Rd) and write (Wr) ports of the payload memory depend upon the issue width and the dispatch width, respectively. In general, most of the pipeline stages in a superscalar processor require some form of memory structure to buffer instructions or instruction related information. The dependence of the number of Rd and Wr ports upon the pipeline width makes the memory structures very specialized to the superscalar design. To estimate the design cost of a pipeline stage, it is important to estimate the design cost of the associated memory structures.

Designing custom memories requires significant design effort and time, and analytically modeling the cost associated with memories is vulnerable to lower fidelity. We take a hybrid approach to develop a multi-ported register file compiler for the 45nm process node [25]. The compiler is very specialized to generate the memories required in a superscalar microarchitecture, and is capable of estimating timing, area, and energy consumption in memories for pipeline widths of one to eight and of different sizes. The memory organization considered in our compiler is similar to the well established SRAM (Static Random Access Memory) based Cache design [55], and its organization and operation details can be found in [53] [54]. Our approach is to

1. pre-design (including the layout) key circuit components, for instance, bitcells, sense amplifiers, decoders, column MUXes, etc., with varying Rd and Wr ports, address bits, and transistor sizes,
2. define composable interfaces of individual components, so that they can be stitched together to compose any arbitrary memory structure,
3. use existing analytical models [56] to estimate intrinsic and coupling capacitances of wires connecting different components, and
4. develop a tool (which we refer to as a register-file compiler) in C++ that can automatically compose a memory based on the size and the number of Rd and Wr ports, and output the SPICE netlist with annotated intrinsic and coupling capacitances, the simulation file with appropriate test vectors, and the estimation of area numbers.

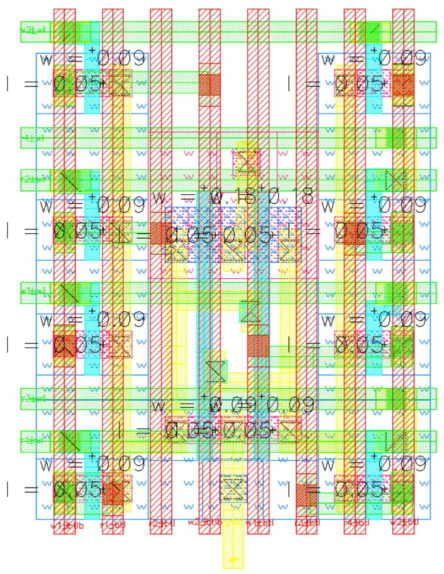
The netlist can further be simulated using any standard SPICE simulator to estimate timing and power consumption. Table 3-1 lists all the pre-designed components used in the register-file compiler and their different flavors. Table 3-2 shows the industry-standard EDA tools used for the layout and the circuit simulation. Figure 3-1 shows layout of four different bitcell configurations used in our register-file compiler.

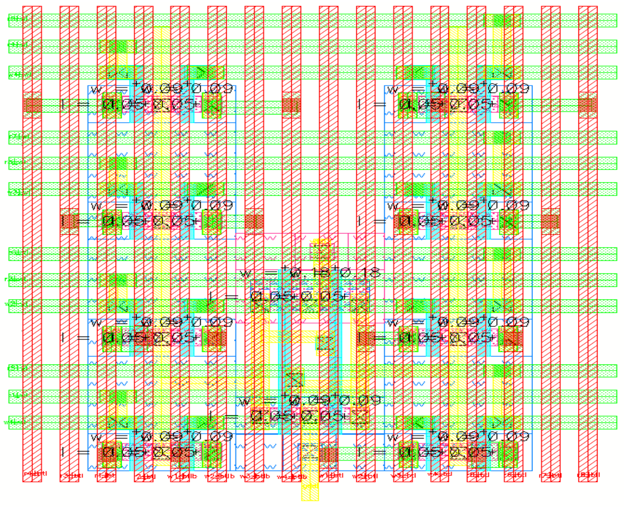
Table 3-1: List of the pre-designed components.

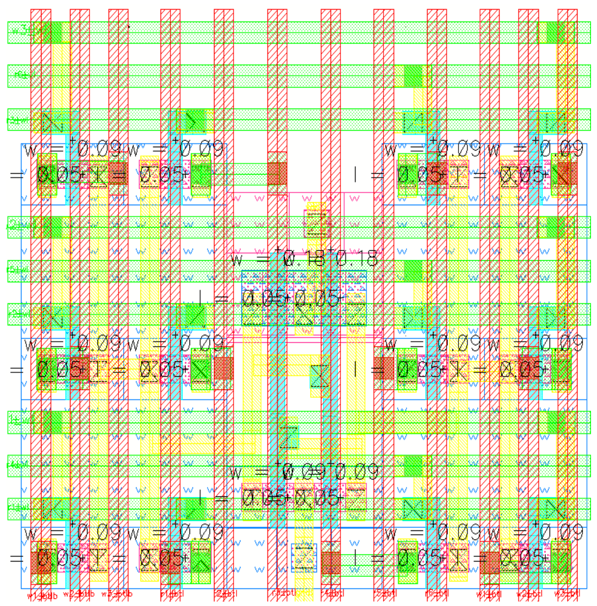
Pre-designed Components	**Different Flavors**
Bitcells	2Rd-1Wr, 4Rd-2Wr, 6Rd-3Wr, 8Rd-4Wr, 10Rd-5Wr, 12Rd-6Wr, 14Rd-7Wr, 16Rd-8Wr, 1Rd-1Wr, 2Rd-2Wr, 3Rd-3Wr, 4Rd-4Wr, 5Rd-5Wr, 6Rd-6Wr, 7Rd-7Wr, 8Rd-8Wr
pre-charge	W_{PMOS}=360nm, W_{PMOS}=720nm
sense amplifier	W_{PMOS}=360nm/ W_{NMOS}=360nm
word-line driver	W_{PMOS}=180nm/ W_{NMOS}=90nm, W_{PMOS}=360nm/ W_{NMOS}=180nm, W_{PMOS}=720nm/ W_{NMOS}=360nm, W_{PMOS}=1440nm/ W_{NMOS}=720nm
row decoder	Decoder width=1, 2, 3, 4, 5, 6, 7, 8
Column multiplexor	Multiplexor width=1, 2

Table 3-2: EDA tools used for the design.

Stage	**EDA Tool/Tools Used**
Schematic and layout	Cadence Virtuoso version: IC6.1.2.500.13
SPICE simulation	Synopsys HSPICE version: C-2009.03-SP1


(a) bitcell: 4Rd-2Wr


(b) bitcell: 6Rd-3Wr


(c) bitcell: 8Rd-4Wr

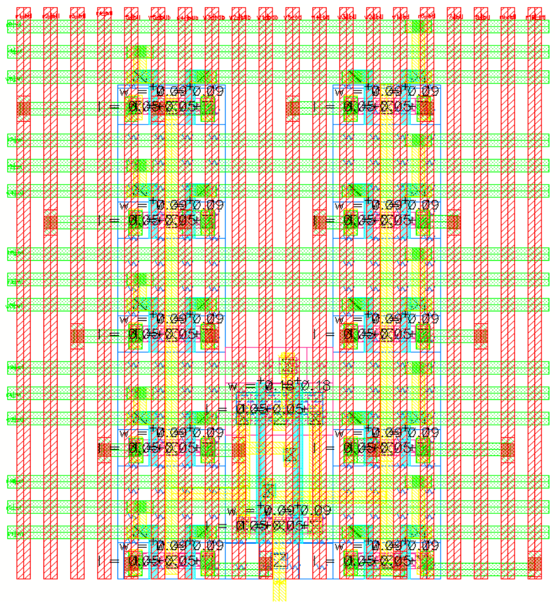

(d) bitcell: 10Rd-5Wr

Figure 3-1: Layouts of four different bitcell configurations.

Estimation of propagation delay and energy consumption requires simulation of multiple test vectors (testing different cases for worst-case scenario) on the SPICE netlist with annotated capacitances. Unfortunately, the SPICE simulation of the complete netlist takes a long time, and to alleviate this problem, we generate the netlist of only the critical path (reading or writing the farthest bitcell from the row decoder) and perform simulation on it. Although we perform only critical-path simulation, the error incurred compared to the full netlist simulation is typically within 5%.

The timing, area, and energy consumption numbers obtained for different memory structures are used during the logic synthesis and place & route of canonical pipeline stages. The results are discussed in chapter 5.

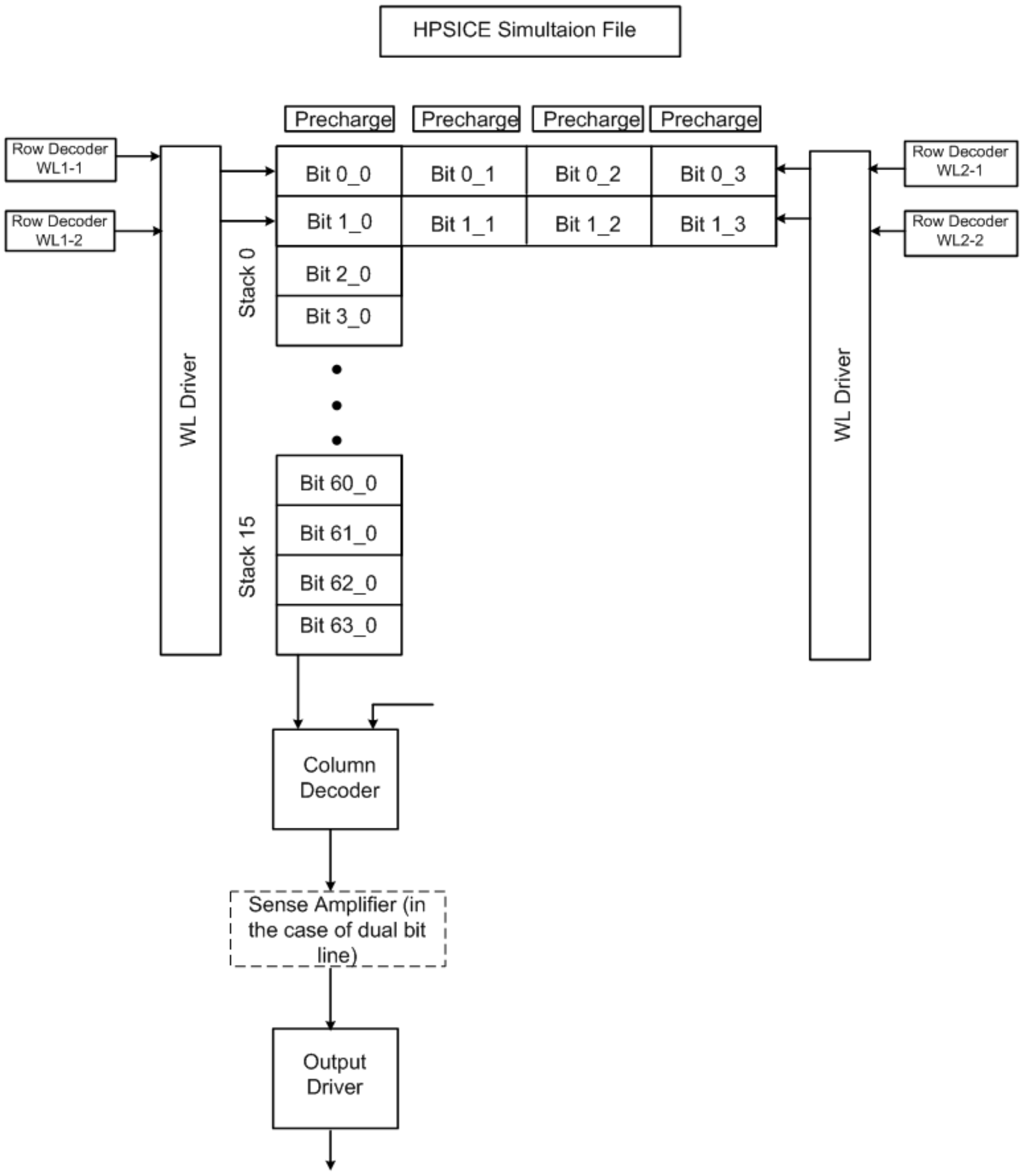


Figure 3-2: Critical path simulation for measuring the read access time and energy consumption.

CHAPTER 4

Simulation Methodology

It is necessary to simulate widely accepted benchmarks for architectural evaluation (IPC) of a new microarchitecture or a technique added to the baseline microarchitecture to enhance performance. SPEC suites [13] and MiBench [34] are some of the standard benchmarks used in academia and industry. To support such simulations, we tightly integrate a high-level functional simulator (written in the C++ programming language) through the Verilog Procedural Interface (VPI) [35] [36], providing a Verilog and C++ co-simulation environment (shown in Figure 4-1). The VPI is a software interface for Verilog, and it consists of a set of access and utility routines to call C++ functions. These routines can be used to exchange information between the instantiated simulation objects contained in the Verilog design. We currently use Cadence NC-Verilog for simulation, and the Cadence environment allows compiled C++ modules to be called from Verilog modules.

The RTL model leverages the functional simulator to load a compiled binary and initialize the processor state, giving the Verilog simulator the flexibility to simulate any standard application benchmark. Moreover, the co-simulation environment can also be exploited for assisting and accelerating functional verification of the Verilog design by asserting correctness of retired results via comparisons with the functional simulator. The functional simulator fetches and executes one instruction at a time, and its execution result is considered golden reference for the verification purpose. The functional simulator always executes instructions in the program order.

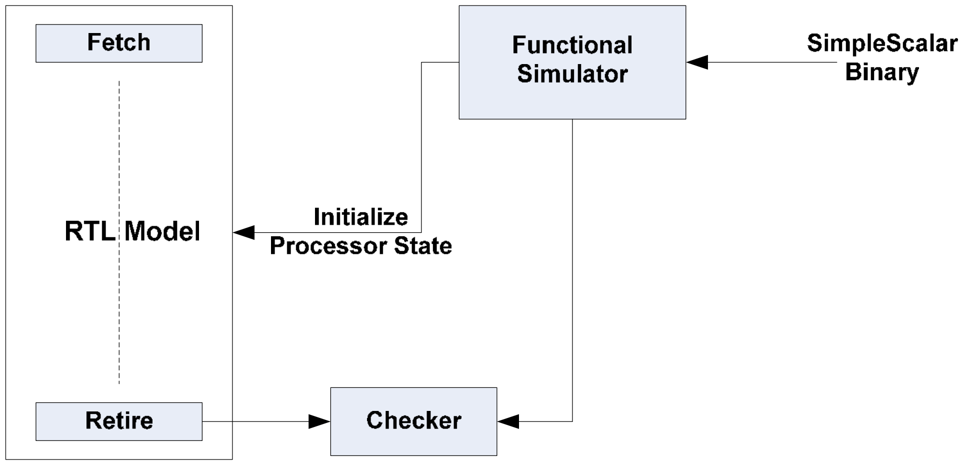


Figure 4-1: Verilog and C++ co-simulation. The left-hand side is the RTL of the processor's pipeline. The checker compares the outputs of the Retire stage with the instructions' results from the functional simulator, shown on the right-hand side.

A microprocessor has an immensely large state space, making the validation of logical correctness of the design a daunting task [48]. Lungu and Sorin [51] argue that design verification, consuming 60-70% of non-recurring engineering (NRE) in the creation of a new microprocessor [49] [50], should be considered as a first-class design constraint, like power consumption and die area. In this thesis, we try to uncover as many design bugs in the limited amount of time and human resources available by running micro-benchmarks on the RTL model and verifying the execution result with the functional simulator output. We use the microarchitecture configuration mentioned in Table 2-1 as a baseline design for all verification purposes. As shown in Figure 4-1, we extensively use the functional simulator for simulation and functional verification. At the beginning of the simulation, the functional simulator loads the compiled SimpleScalar binary into the co-simulation environment and initializes the processor state. During the Retire stage, the RTL model verifies the PC and result of the retiring instruction with the corresponding result of the functional simulator. On a mismatch, the source of the design bug is traced manually using the signal viewer in the Cadence NC-Verilog. As part of the FabScalar project, we are developing a cycle-accurate C++ representation of the RTL model, allowing the C++ and the Verilog implementations of each pipeline stage to cross-check their outputs every cycle; this is left for future work. Table 4-1 shows the major VPIs used in the RTL model.

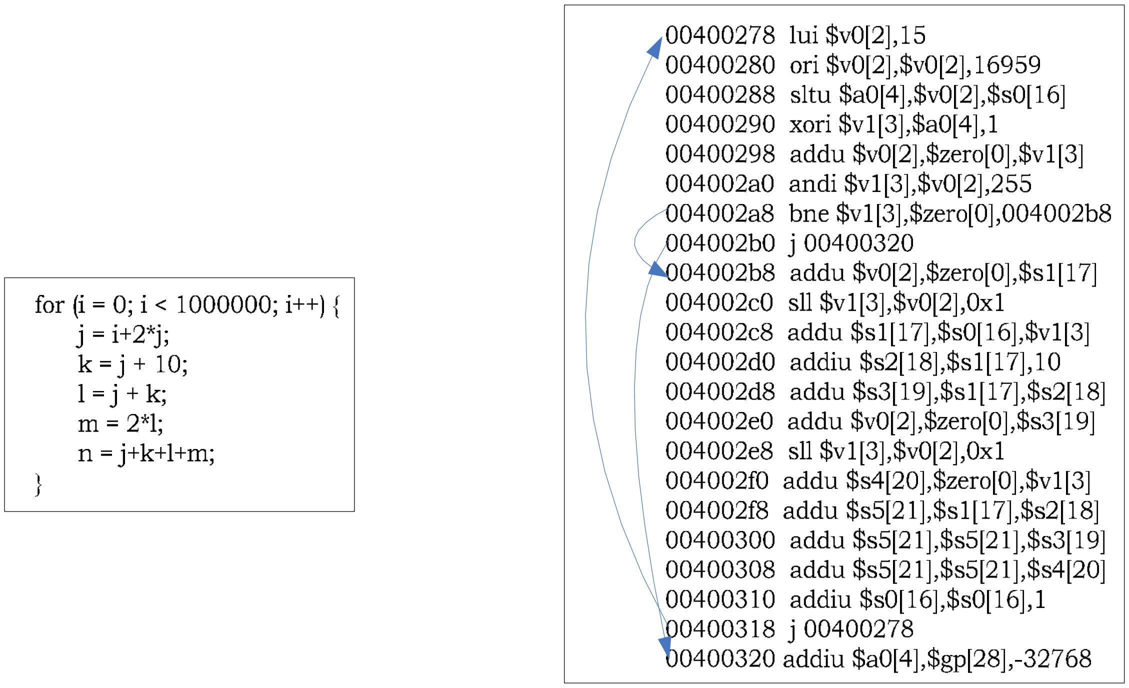
Table 4-1: VPIs used in the RTL model.

VPI Name	Functionality
$initialize_sim()	Invokes the functional simulator and loads the program binary.
$getArchRegValue()	Copies the architectural register values from functional simulator to the corresponding physical registers in the RTL model.
$getArchPC()	Initializes the PC in the RTL model.
$getRetireInstPC()	Retire stage checks the PC of the retiring instruction with the functional simulator output.
$getRetireInstValue()	Retire stage checks the result of the retiring instruction with the functional simulator output.

To simplify the verification effort, we identify three major high level functional aspects of the superscalar microarchitecture (the design) and the logic associated with them:

1. Register data-flow: the design includes register renaming, allocation of physical registers (popped from the Free List) to a new instruction with a valid destination operand, allocation of dispatched instructions in the issue queue and active list hardware structures, wake-up and select logic, bypass logic, and functional units.
2. Control-flow: the design includes next PC logic, checkpointing and branch mask logic, and logic for flushing the front-end and selectively removing instructions on the wrong control flow path from the hardware structures in the out-of-order stages on a branch misprediction.
3. Memory data-flow: the design includes allocation of dispatched load/store instructions in the Load/Store Unit, memory disambiguation logic, and store-to-load forwarding logic.

This breakdown helped us in developing micro-benchmarks stressing one functional aspect at a time and allowed us to concentrate our efforts in finding design bugs in a limited state space. Figure 4-2 shows an example of the kernel inside a micro-benchmark which stresses the register data-flow with no memory operations and very simple control-flow. The kernel runs for 10 million cycles on the RTL model with no control misprediction (after initial training of the BTB and BPB within 1000 cycles), retiring approximately 12 million instructions. Similarly, we developed micro-benchmarks stressing control-flow but keeping the data-flow simple, for instance, a toggling branch instruction inside a loop.



```
for (i = 0; i < 1000000; i++) {
      j = i+2*j;
      k = j + 10;
      l = j + k;
      m = 2*l;
      n = j+k+l+m;
}
```

```
00400278  lui $v0[2],15
00400280  ori $v0[2],$v0[2],16959
00400288  sltu $a0[4],$v0[2],$s0[16]
00400290  xori $v1[3],$a0[4],1
00400298  addu $v0[2],$zero[0],$v1[3]
004002a0  andi $v1[3],$v0[2],255
004002a8  bne $v1[3],$zero[0],004002b8
004002b0  j 00400320
004002b8  addu $v0[2],$zero[0],$s1[17]
004002c0  sll $v1[3],$v0[2],0x1
004002c8  addu $s1[17],$s0[16],$v1[3]
004002d0  addiu $s2[18],$s1[17],10
004002d8  addu $s3[19],$s1[17],$s2[18]
004002e0  addu $v0[2],$zero[0],$s3[19]
004002e8  sll $v1[3],$v0[2],0x1
004002f0  addu $s4[20],$zero[0],$v1[3]
004002f8  addu $s5[21],$s1[17],$s2[18]
00400300  addu $s5[21],$s5[21],$s3[19]
00400308  addu $s5[21],$s5[21],$s4[20]
00400310  addiu $s0[16],$s0[16],1
00400318  j 00400278
00400320 addiu $a0[4],$gp[28],-32768
```

Figure 4-2: A kernel inside a micro-benchmark, stressing the register data-flow logic. The left-hand side is C++ code and the right-hand side is the corresponding SimpleScalar machine code. The arrows in the right-hand box indicate the control flow.

Although we did unit-level testing of individual Verilog modules during their development phase, we could start full processor-level verification at the beginning of February, 2009. We have documented all the design bugs uncovered during full processor-level verification and did a classification study to understand how bugs were introduced. We characterize the bugs into four categories as shown in Figure 4-3, 1) *coding mistake:* these bugs were introduced by typing mistakes, copy and paste, and careless coding, 2) *microarchitecture:* these bugs were due to wrong or incomplete microarchitectural definition of a particular pipeline stage, 3) *logic changes:* these bugs were introduced after performing logic optimization in the design, 4) *corner cases:* these are special scenarios we failed to foresee during the design process. We realize most of the design bugs were introduced due to coding mistakes and wrong or incomplete microarchitectural definition.

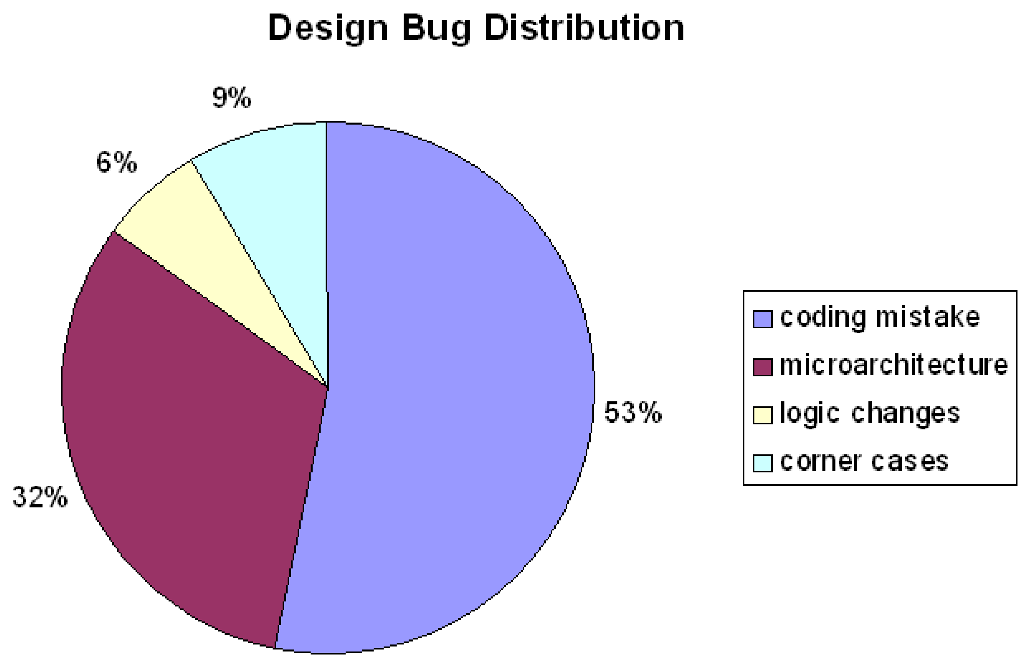


Figure 4-3: Design bug distribution over 4 categories.

We are yet to exhaustively verify the entire processor, or to a level the FabScalar project requires. As part of the future effort in the functional verification we would like to fix known microarchitectural bugs, for instance, bugs in the logic to wake-up a disambiguation-stalled load in the load queue. Eventually, we would like to run the SimPoint [52] associated with each SPEC benchmark, consisting of approximately 100 million instructions each, on the RTL model.

CHAPTER 5

Results and Discussions

A processor design has an associated cost, where the cost can be quantified in terms of propagation delay, power consumption, die area, design effort, manufacturability, or fault vulnerability. Until recently, microarchitectural innovations and technology scaling have led to exponential growth in performance, with associated design cost within an acceptable budget. However, achieving further performance enhancement requires excessively complex microarchitecture solutions, and the logic complexity to implement such a design has significant impact on costs: propagation delay, power consumption, and area.

A complex microarchitecture might enhance IPC, but at the same time could increase the propagation delay. For instance, increasing the size of the issue window can boost IPC for applications with abundant ILP, but at the same time, clock rate may decrease to accommodate the larger content addressable memory. Figure 5-1 shows the impact of increasing issue window on the delay of the wakeup-select logic for an issue width of two. In general, any attempt to increase microarchitectural complexity to get better IPC has a direct impact on the propagation delay.

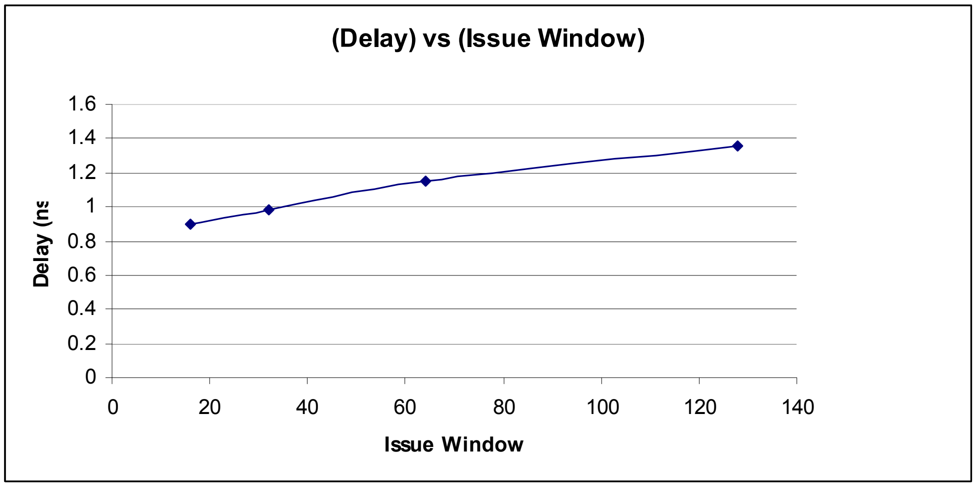


Figure 5-1: Delay of wakeup-select logic for different issue window sizes. (Issue width is fixed at 2 instr/cycle.)

In this chapter, we present the impact of microarchitecture complexity on the propagation delay of the superscalar processor's canonical pipeline stages. By increasing complexity, we mean wider pipeline stages (more ways) and larger specialized memories within a stage. We derive the Verilog RTL of the pipeline stage for varying complexity, from the baseline four-wide processor design. To estimate the propagation delay, we synthesized and did basic place-and-route of each pipeline stage using 45nm technology standard cell library [25]. For memory structures, we use timing numbers from the register file compiler. As part of the future work, we would like to estimate the impact of microarchitecture complexity on the area and energy consumption for each pipeline stages.

5.1 Microarchitectural Complexity Study

To continuously feed the rest of the pipeline stages, it is necessary to fetch more instructions in a cycle. Wider fetch width leads to increased logic complexity and, hence, increased propagation delay. In the Fetch-1 stage, logic to generate the next PC is the most critical from a cycle time standpoint. The timing critical path of the next PC logic consists of reading the BTB and the BPB for each PC, and feeding the read information to the priority logic to select the next PC. Figure 5-2 shows the impact of fetch width on the propagation delay of the next PC logic.

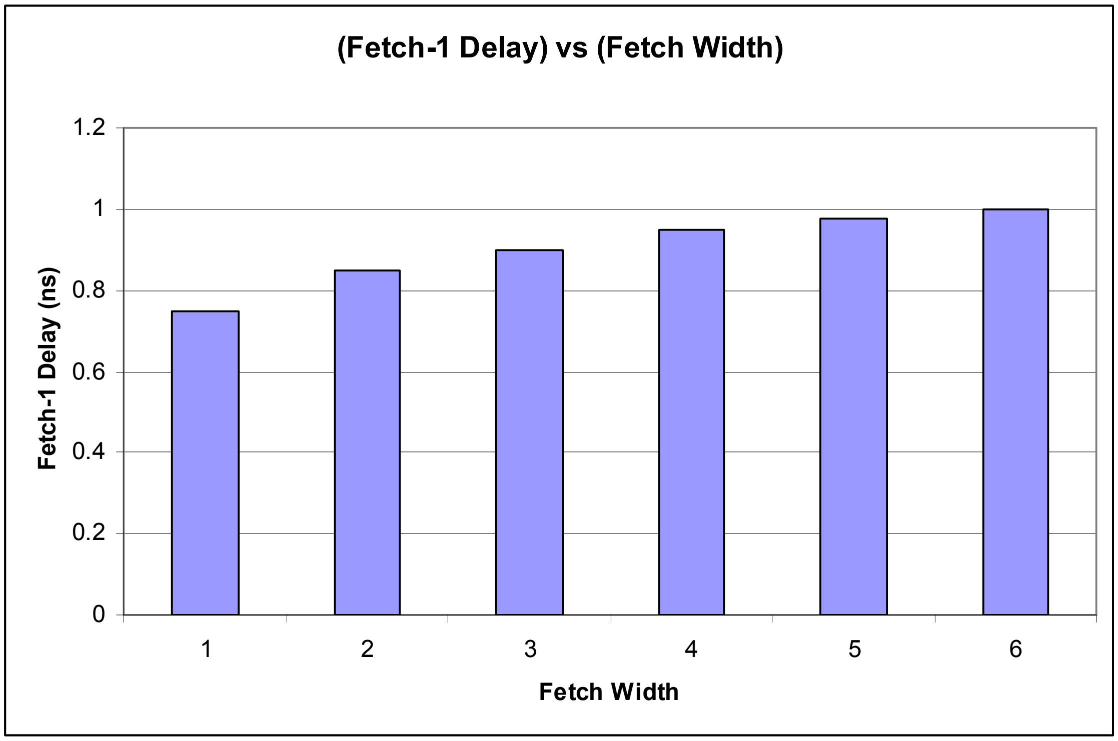


Figure 5-2: The next PC logic delay for varying fetch widths. Sizes of the BTB and the BPB are fixed at 64KB and 4KB, respectively.

The timing critical path of the Fetch-2 stage consists of instruction alignment, extracting the fetch block, pre-decoding the instructions for control instructions, and generating the recovery signal for the Fetch-1 stage. All of these steps are serialized, with the fetch block extraction logic (which depends on the fetch width) consuming most of the propagation delay in this stage. Figure 5-3 shows the impact of fetch width on the propagation delay of the Fetch-2 stage.

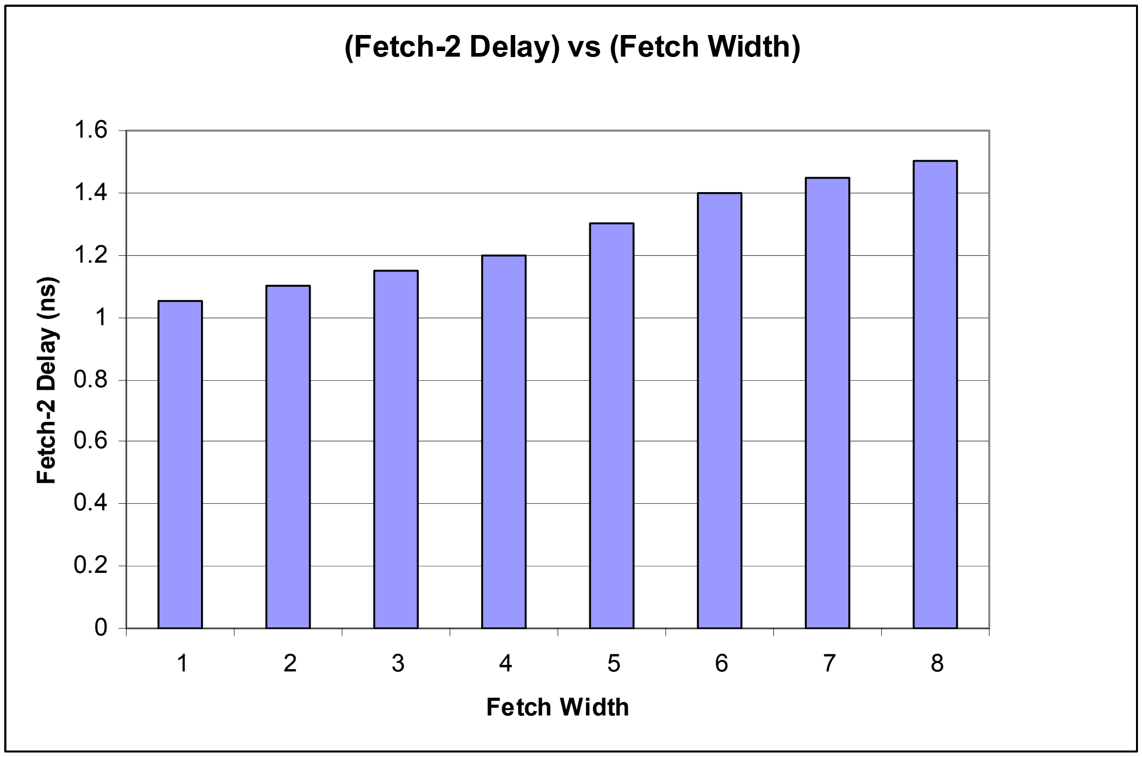


Figure 5-3: The Fetch-2 stage delay with varying fetch width.

The instruction decode logic, like other RISC ISAs, is straightforward to implement. The delay is noticeably less than other stages. Figure 5-4 shows the impact of decode width on the propagation delay of the instruction decode logic.

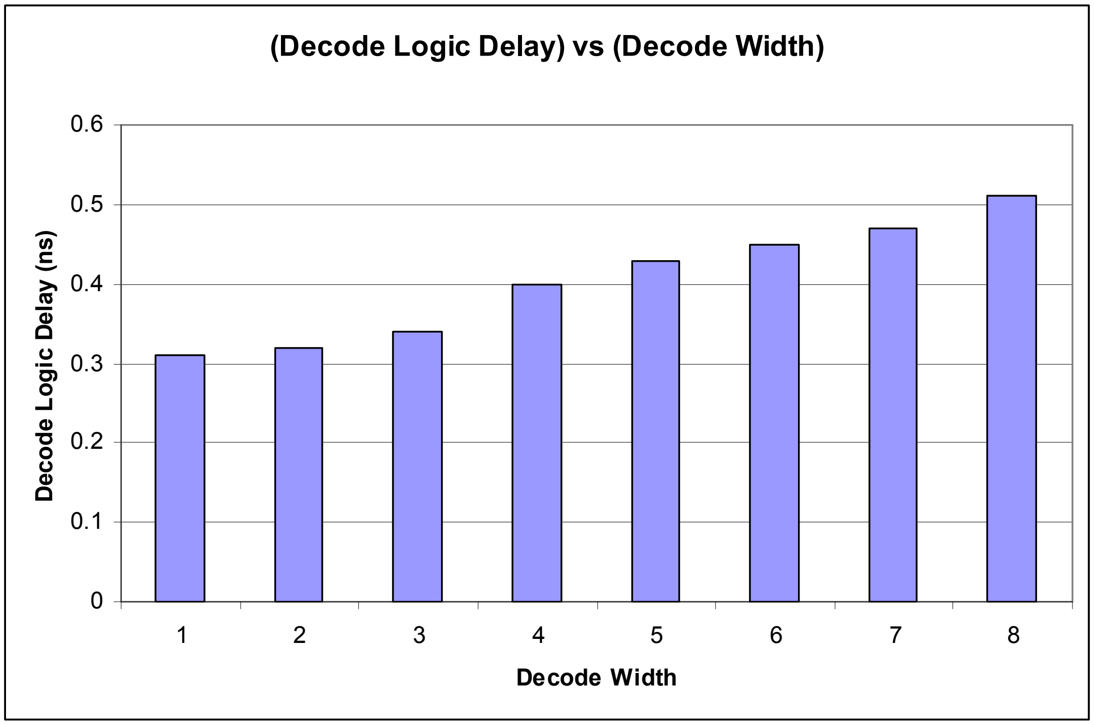


Figure 5-4: The Decode stage delay with varying decoder width.

The propagation delay of the Rename Stage is dictated by the latency of reading the multi-ported RMT and the bypass multiplexors following the RMT. The RMT is implemented as SRAM, with the number of read and writes ports being multiples of the rename width, shown in Table 5-1.

Table 5-1: RMT read and write ports for varying rename width.

Rename Width	RMT read ports	RMT write ports
1	2	1
2	4	2
3	6	3
4	8	4
5	10	5
6	12	6
7	14	7
8	16	8

With increasing rename width, the complexity of the SRAM and the subsequent bypass multiplexors increases, leading to more propagation delay. Figure 5-4 shows the impact of the rename width on the register rename logic. Although the bit width of the RMT changes with varying physical register file size, its impact on the rename delay is marginal.

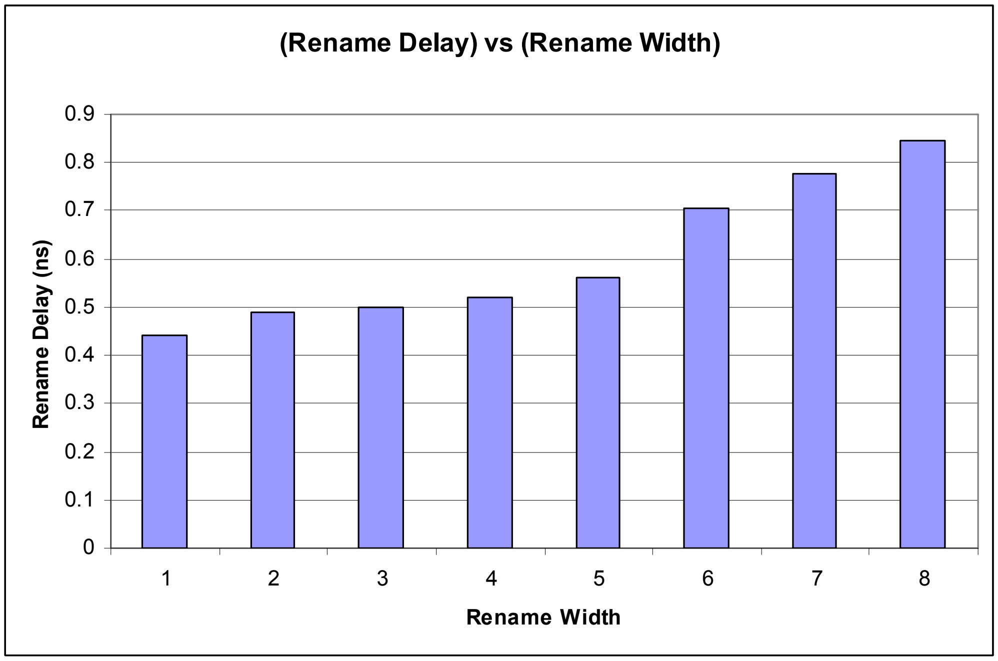


Figure 5-5: The Rename stage delay with varying rename width.

The dispatch logic involves writing the renamed instructions into the Active List, the LQ (for loads), the SQ (for stores), and the Issue Queue. The Active List and the LQ/SQ are FIFOs, unlike the SRAM in the issue queue. Hence, writing the renamed instructions to the issue queue forms the timing critical path in the dispatch stage. Figure 5-6 shows the delay of the Dispatch stage for varying issue queue size and for different dispatch width. As evident from the figure, increasing the dispatch width has more impact on the delay than increasing the issue queue size, attributing to the increasing number of SRAM write ports with the increasing dispatch width.

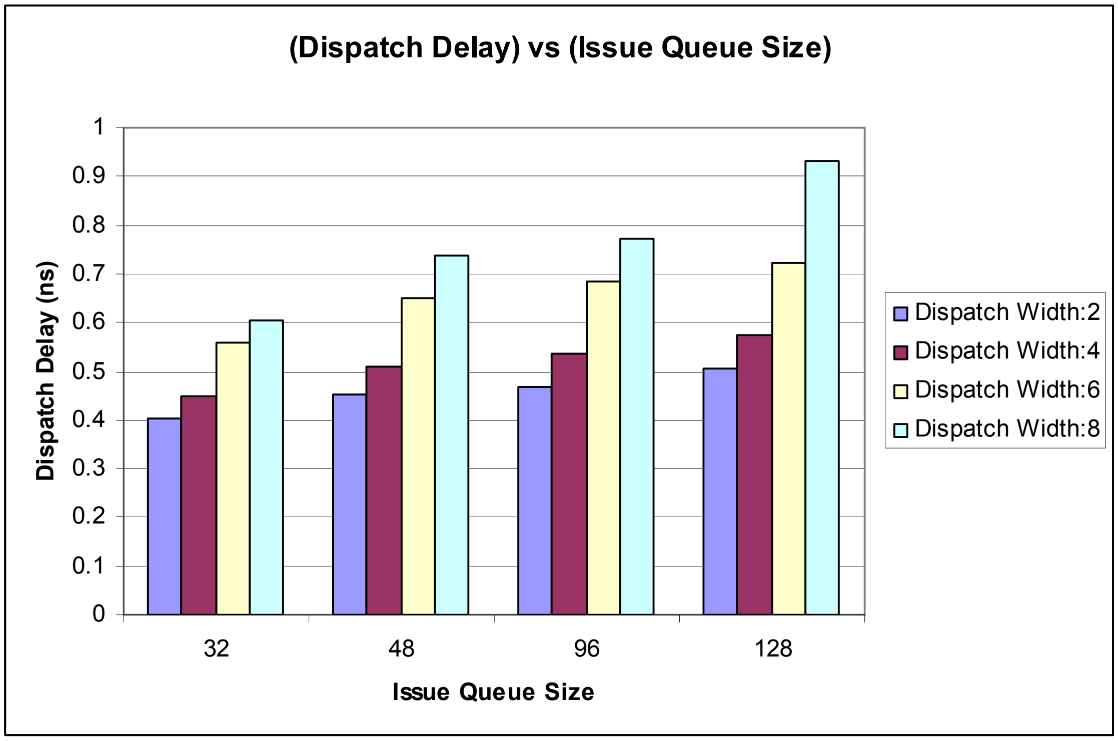


Figure 5-6: The Dispatch delay with varying Issue Queue size, and for different dispatch widths.

Figure 5-7 shows the delay of the wake-up logic for varying the issue queue size and the issue width. The delay of the wakeup logic increases more significantly with increasing issue width, attributing to two factors. Increasing issue width increases the number of parallel comparators and the fan in of the following OR gate. Moreover, increasing issue width increases the number of issue queue read ports (implemented as CAM).

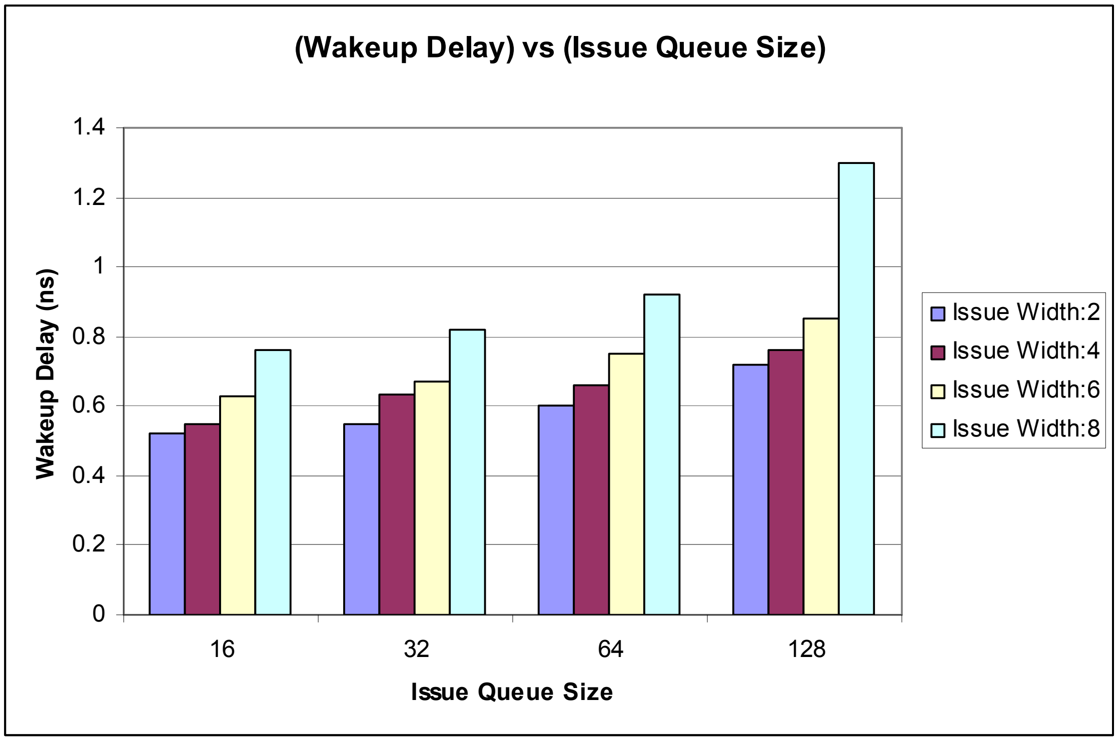


Figure 5-7: The Wakeup logic delay for varying issue queue size and issue width.

The logic complexity of the select logic increases with increasing issue queue size and issue width. Increasing the issue queue size increases either the number of levels in the tree-based selection logic or the sizes of the multiplexors at each level. Increasing the issue width leads to delayed generation of the request vector at each level, as the selection of an instruction depends upon all the previous selections. Figure 5-8 shows the select logic delay as the issue queue size and the issue width are varied.

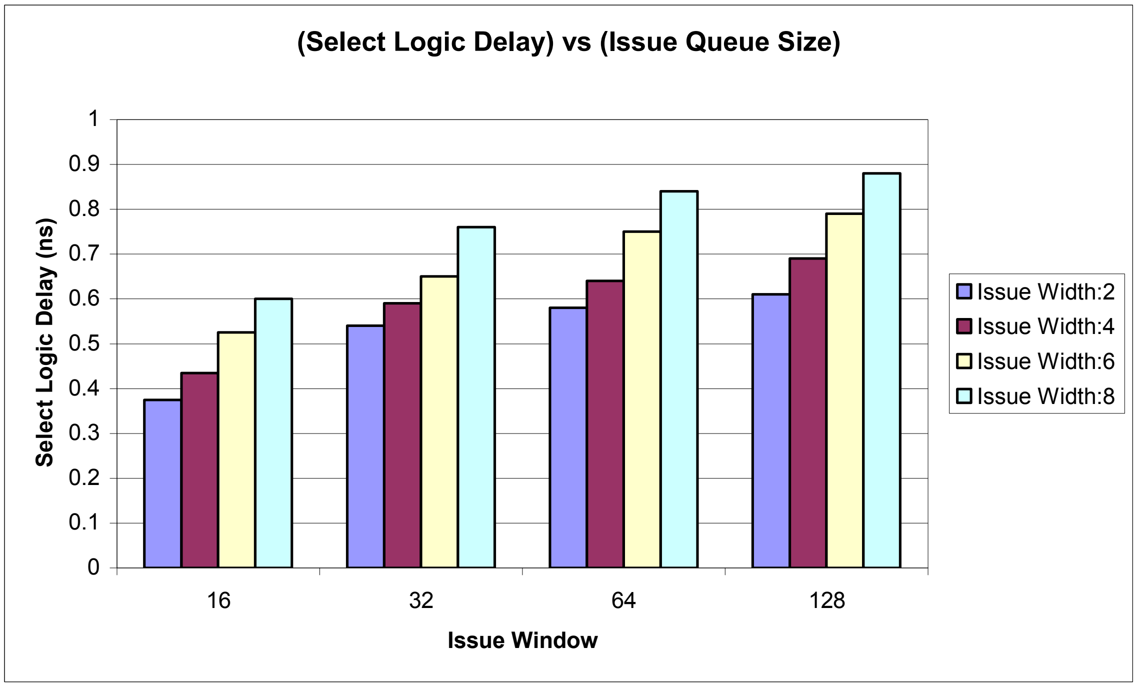


Figure 5-8: The Select logic delay for varying issue queue size and issue width.

The propagation delay of the Register Read stage is dictated by the latency of reading the multi-ported PRF and the bypass multiplexors following the PRF. The PRF is implemented as an SRAM, with the number of read and writes ports being a multiple of the issue width, as shown in Table 5-2.

Table 5-2: Register File read and write ports for varying issue width.

Issue Width	PRF read ports	PRF write ports
1	2	1
2	4	2
3	6	3
4	8	4
5	10	5
6	12	6
7	14	7
8	16	8

With increasing issue width, the complexity of the SRAM and the following bypass multiplexors increases, leading to more propagation delay. Figure 5-9 shows the impact of the PRF size and the issue width on the register read logic.

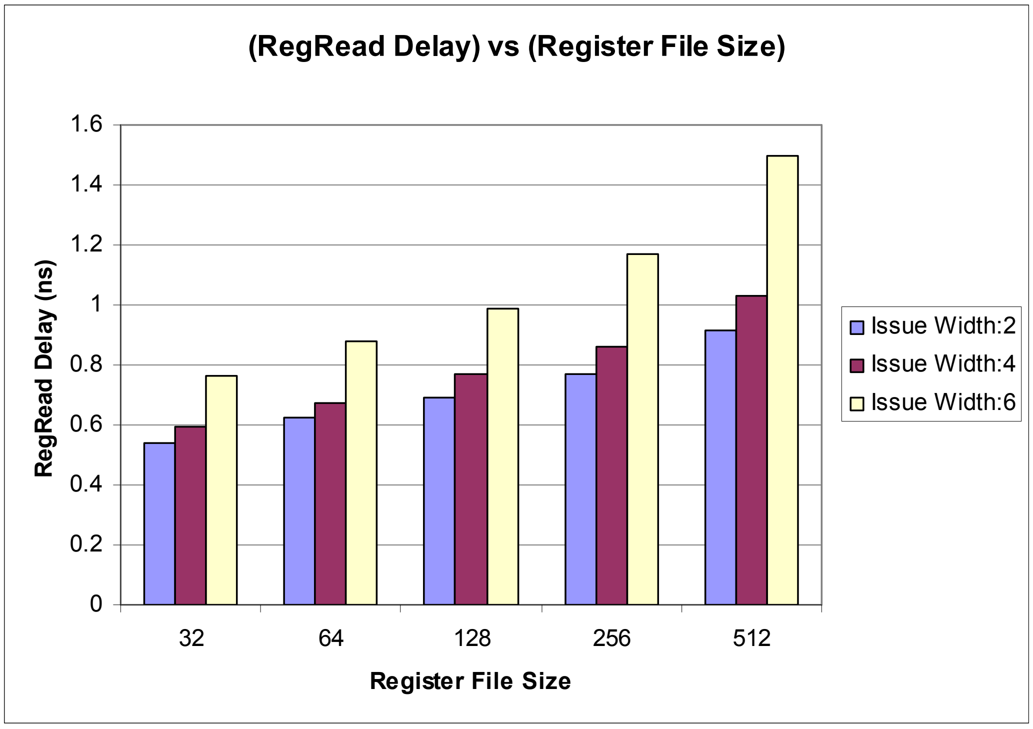


Figure 5-9: The Register Read stage logic delay for varying PRF size and issue width.

Figure 5-10 shows the propagation delay of the LSU with varying LQ and SQ sizes. The LQ is implemented as a pair of CAM and RAM, and so is the SQ. The CAM holds the memory addresses and the RAM holds the data and other control information. With increasing the sizes of the LQ and SQ, the sizes of CAMs and RAMs increase, which fall on the timing critical path of the disambiguation logic.

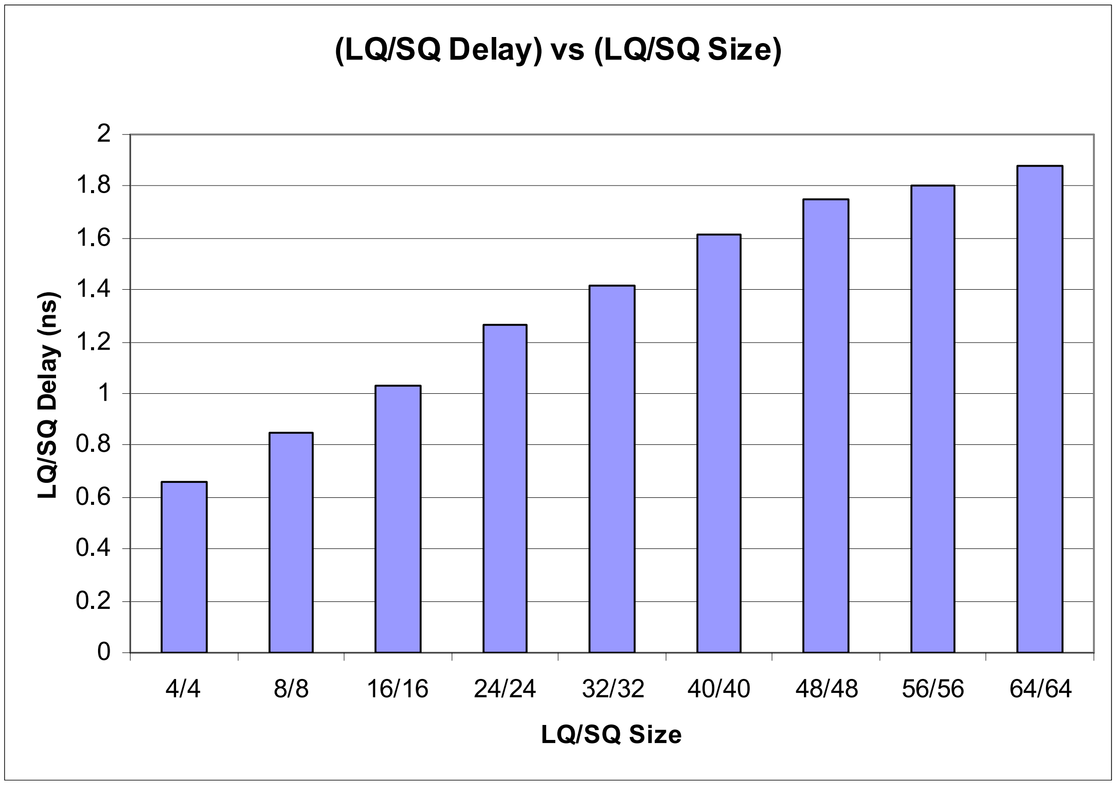


Figure 5-10: The Load-Store Queue logic delay for varying load and store queue sizes.

Table 5-3 shows the delay of all the four types of functional units used in our design. As expected, the complex ALU (implementing multiply and divide operations) has the largest latency.

Table 5-3: Delay of different functional units (un-pipelined).

ALU Type	**Data Width (bits)**	**Total Delay (ns)**
Simple ALU	32	0.45
Complex ALU	32	1.15
Ctrl ALU	32	0.45
AGEN	32	0.44

5.2 Physical Design

Figure 5-11 shows the physical design of a 4-way superscalar processor, with the same configuration mentioned in Table 2-1. The physical design is very preliminary, and can be further improved for better timing. Although cursory, the physical design shows the strength of our RTL model. The RTL model can be implemented using a standard ASIC flow for the detailed study of low level costs associated with the microprocessor design. Table 5-4 shows the physical design data for this specific implementation.

Table 5-4: Design data for 4-way superscalar processor physical implementation.

Technology	45nm
Die area (excluding L1 caches)	2.6 mm^2
Clock frequency	500MHz
Number of ports	324
Number of sequential elements	15,585
Power consumption (only standard cells)	118mW
The worst timing-critical path	Next-PC logic (Fig. 2.7)

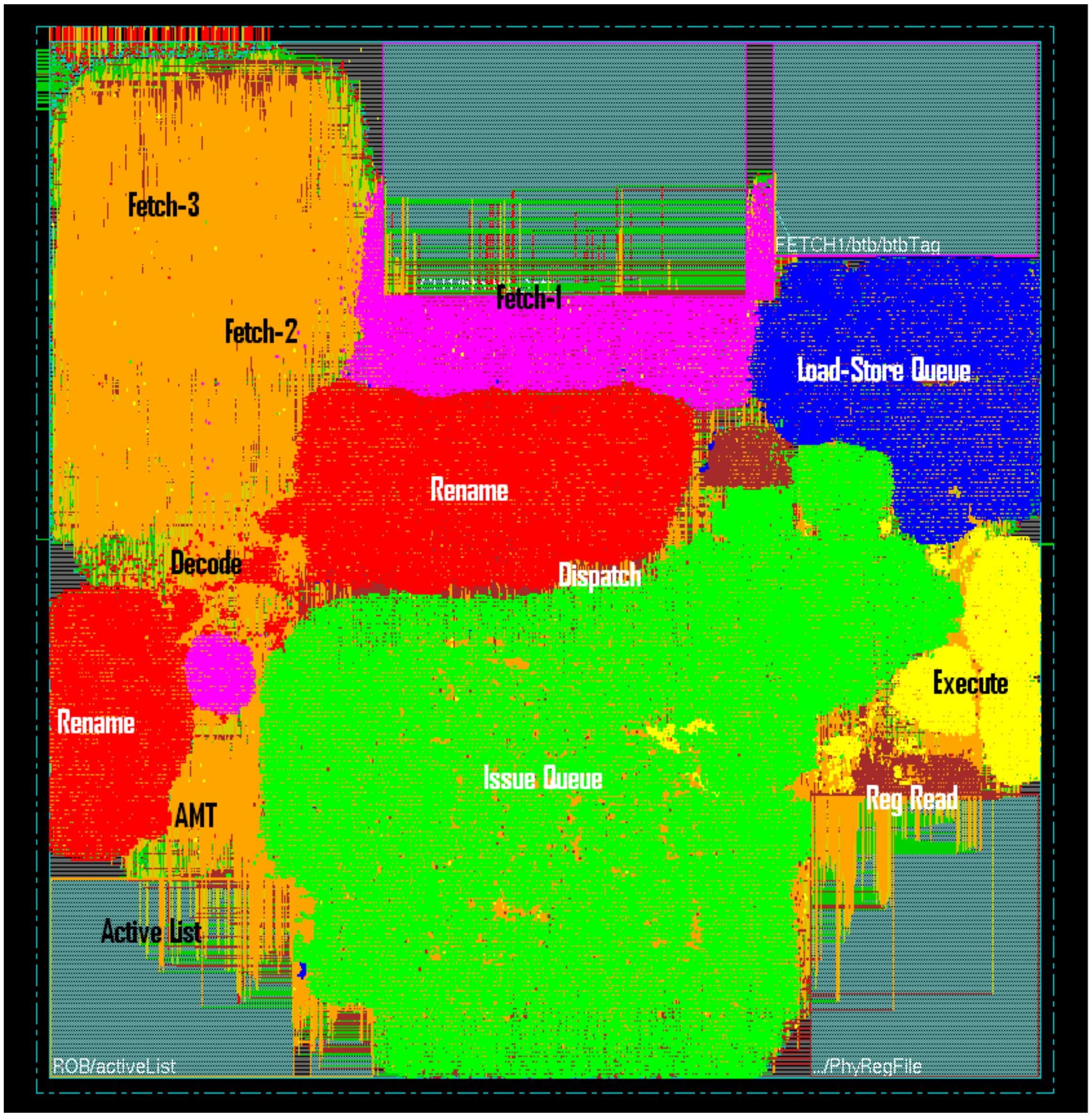


Figure 5-11. Placed-and-routed 4-way superscalar processor, excluding L1 I- and D-caches.

References

[1] http://www.intel.com/pressroom/kits/quickreffam.htm#pentium.

[2] David Lammers. Intel cancels Tejas, moves to dual-core designs. EETimes Article, 2004.

[3] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core Architecture Optimization for Heterogeneous Chip Multiprocessors. PACT, Sep. 2006.

[4] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Par-thasarathy Ranganathan, Dean M. Tullsen. Single-ISA Het-erogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, December 2003.

[5] H. Hashemi Najaf-abadi and E. Rotenberg. Architectural Contesting. Proceedings of the 15th IEEE International Sym-posium on High-Performance Computer Architecture (HPCA-15), pp. 189-200, February 2009.

[6] Niket K. Choudhary, Salil Wadhavkar, Tanmay Shah, Sandeep Navada, Hashem H. Najaf-abadi, and Eric Rontenberg. FabScalar. In the Workshop on Architecture Research Prototyping (WARP), in conjunction with ISCA-36, 2009.

[7] Kurt Keutzer, A. Richard Newton, and Narendra Shenoy. The future of logic synthesis and physical design in deep-submicron process geometries. Proceedings of the 1997 international symposium on Physical design, p.218-224, April 14-16, 1997, Napa Valley, California, United States.

[8] H. P. Hofstee. Power Efficient Processor Architecture and the Cell Processor. HPCA, 2005.

[9] Benjamin C. Lee, David M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. ASPLOS-XII: International Conference on Architectural Support for Programming Languages and Operating Systems. San Jose, CA, October 2006.

[10] Sukhun Kang and Rakesh Kumar. Magellan: A Framework for Fast Muti-core Design Space Exploration and Optimization Using Search and Machine Learning. Design, Automation, and Test in Europe, DATE, Munich, March 2008.

[11] J.L. Cruz, A. González and M. Valero. Multiple-Banked Register File Architecture. ISCA-27, IEEE-ACM International Symposium on Computer Architecture. Vancouver, June 2000.

[12] N.J. Wang, J. Quek, T.M. Rafacz, and S.J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In International Conference on Dependable Systems and Networks. IEEE Computer Society, Jun 2004.

[13] The Standard Performance Evaluation Corporation, http://spec.org.

[14] http://www.opensparc.net/

[15] http://www.opencores.org/

[16] J. E. Smith and G. S. Sohi. The Microarchitecture of Superscalar Processors. In Proceedings of the IEEE, December 1995.

[17] A. R. Lebeck, J. J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A Large, Fast Instruction Window for Tolerating Cache Misses. Proceedings of the 29th IEEE/ACM International Symposium on Computer Architecture, pp. 59-70, May 2002.

[18] A. Hartstein, Thomas R. Puzak. The optimum pipeline depth for a microprocessor. Proceedings of the 29th annual international symposium on Computer architecture, May 2002, Anchorage, Alaska.

[19] Eric Sprangle, Doug Carmean, Increasing processor performance by implementing deeper pipelines, Proceedings of the 29th annual international symposium on Computer architecture, May 2002, Anchorage, Alaska.

[20] B.S. Amrutur and M.A. Horowitz. Speed and power scaling of SRAMs. IEEE Journal of Solid State Circuits, 35(2): 175-185, February 2000.

[21] Todd Austin, Eric Larson, Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. Computer, v.35 n.2, p.59-67, February 2002.

[22] Charles Price. MIPS IV Instruction Set, revision 3.1. MIPS Technologies, Inc., Mountain View, CA, January 1995.

[23] K. C. Yeager. MIPS R10000 Superscalar Microprocessor. In IEEE Micro, April 1996.

[24] Jim Keller. The 21264: A Superscalar Alpha Processor with Out-of-Order Execution, October 1996. 9th Annual Microprocessor Forum, San Jose, California.

[25] James E. Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W. Rhett Davis, Paul D. Franzon, Mi-chael Bucher, Sunil Basavarajaiah, Julie Oh, Ravi Jenkal, "FreePDK: An Open-Source Variation-Aware Design Kit," mse,pp.173-174, 2007 IEEE International Conference on Microelectronic Systems Education (MSE'07), 2007.

[26] André Seznec , Stéphan Jourdan , Pascal Sainrat , Pierre Michaud, Multiple-block ahead branch predictors, Proceedings of the seventh international conference on Architectural support for programming languages and operating systems, p.116-127, October 01-04, 1996, Cambridge, Massachusetts, United States.

[27] Daniel A. JimCnez, Stephen W. Keckler, and Calvin Lin. The impact of delay on the design of branch predictors. In Proceedings of the 33th Annucri Internutionul Symposium on Microorchitecture, December 2000.

[28] B. Ramakrishna Rau , Joseph A. Fisher, Instruction-level parallel processing: history, overview, and perspective, The Journal of Supercomputing, v.7 n.1-2, p.9-50, May 1993.

[29] James E. Smith, Andrew R. Pleszkun. Implementation of precise interrupts in pipelined processors, Proceedings of the 12th annual international symposium on Computer architecture, p.36-44, June 17-19, 1985, Boston, Massachusetts, United States.

[30] W. W. Hwu , Y. N. Patt, Checkpoint repair for out-of-order execution machines, Proceedings of the 14th annual international symposium on Computer architecture, p.18-26, June 02-05, 1987, Pittsburgh, Pennsylvania, United States.

[31] Kenneth C. Yeager, The MIPS R10000 Superscalar Microprocessor, IEEE Micro, v.16 n.2, p.28-40, April 1996.

[32] Subbarao Palacharla , Norman P. Jouppi , J. E. Smith. Complexity-effective superscalar processors. Proceedings of the 24th annual international symposium on Computer architecture, p.206-218, June 01-04, 1997, Denver, Colorado, United States.

[33] T. Karkhanis and J. E. Smith. Automated Design of Application-Specific Superscalar Processors. ISCA, 2007.

[34] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, MiBench: A Free, Commercially Representative Embedded Benchmark Suite, IEEE 4th Annual Workshop on Workload Characterization, Austin, TX (December 2001).

[35] C. Dawson, S.K. Pattanam, D. Roberts, "The Verilog Procedural Interface for the Verilog Hardware Description Language," ivc, pp.17, 1996 IEEE International Verilog HDL Conference (IVC '96), 1996.

[36] Stuart Sutherland. The VERILOG PLI Handbook: A User's Guide and Comprehensive Reference on the VERILOG Programming Language Interface. Kluwer Academic Publishers, Norwell, MA, 1999

[37] M.Bohr el al., “A high-performance 0.25-pm logic technology optimized for 1.8V operation”, IEDM, pp. 847-850, 1996.

[38] Scott Thompson, Paul Packan, and Mark Bohr. “MOS Scaling: Transistor Challenges for the 21st Century”, Intel Technology Journal, Q9, 1998.

[39] David A. Patterson , John L. Hennessy. Computer architecture: a quantitative approach, Morgan Kaufmann Publishers Inc., San Francisco, CA, 1990

[40] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. 27th International Symposium on Computer Architecture (ISCA-27), Vancouver, Canada, June 2000.

[41] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical Report 2001-2, HP, Western Research Laboratory, 2001.

[42] Cyrus Bazeghi , Francisco J. Mesa-Martinez , Jose Renau. uComplexity: Estimating Processor Design Effort. Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture, p.209-218, November 12-16, 2005, Barcelona, Spain.

[43] Strozek, Brooks. Efficient Architectures through Application Clustering and Achitectural Heterogeneity. CASES, 2006.

[44] V. Kathail, et al. PICO: Automatically Designing Custom Computers. IEEE Computer, 35(9):39-47, Sep. 2002.

[45] Tom R. Halfhill. Tensilica's software makes hardware. Microprocessor Report, 23 June 2003.

[46] Il Park, Michael Powell, and T. N. Vijaykumar. Reducing register ports for higher speed and lower power. In Proceedings of the 35th Annual International Symposium on Microarchitecture (MICRO), pages 171-181, November 2002.

[47] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In Proceedings of the 34th International Symposium on Microarchitecture (MICRO 34), pages 237-249, Dec. 2001.

[48] R. M. Bentley. Validating the Pentium 4 Microprocessor. In Proceedings of the International Conference on Dependable Systems and Networks, pages 493-498, July 2001.

[49] P. Bose, D. H. Albonesi, and D. Marculescu. Guest Editors' Introduction: Power and Complexity Aware Design. IEEE Micro, pages 8–11, Sept/Oct 2003.

[50] R. Hum. How to Boost Verification Productivity. EE Times, January 10 2005.

[51] Anita Lungu and Daniel J. Sorin. Verification-Aware Microprocessor Design. Sixteenth International Conference on Parallel Architectures and Compilation Techniques (PACT), September 2007.

[52] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. In the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, October 2002.

[53] Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic´. Digital Integrated Circuits. Prentice Hall, 2nd Edition.

[54] N. Weste and K. Eshraghian. Principles of CMOS VLSI Design. AddisonWesley, 2nd Edition.

[55] S. Wilton and N. Jouppi. Cacti: An enhanced cache access and cycle time model. In IEEE Journal of Solid-State Circuits, May 1996.

[56] N. Delorme, M. Bellevile, and J. Chilo. Inductance and capacitance formulas for VLSI interconnects. Electronic Letters, Vol. 32, No. 1 I, pp. 996-997, May 1996.

[57] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In the 19th International Symposium on Computer Architecture, May 1992.

[58] Y. Sazeides and J. E. Smith. The predictability of data values. In Proc. 30th Annu. Int. Symp. Microarchitecture, Dec. 1997, pp. 248-258.

[59] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In Proceedings of the 29th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture, pp. 226-237, December 1996.

[60] James E. Smith. A study of branch prediction strategies. 25 years of the international symposia on Computer architecture (selected papers), p.202-215, June 27-July 02, 1998, Barcelona, Spain.

[61] Tse-Yu Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. Proceedings of the 20th annual international symposium on Computer architecture, p.257-266, May 16-19, 1993, San Diego, California, United States.

[62] S. McFarling. Combining branch predictors. In DEC WRL Technical Note TN-36. DEC Western Research Laboratory, 1993.

[63] Mary D. Brown, Jared Stark, and Yale N. Patt. Select-free instruction logic. In 34th Int'l Syrup. on Microarchitecture, pp. 204-213, December 2001.

[64] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. Proceedings of the 25th annual international symposium on Computer architecture, p.142-153, June 27-July 02, 1998, Barcelona, Spain.

[65] R. Ronen, A. Mendelson, K. Lai, L. Shih-Lien, F. Pollack, and J. P. Shen. Coming challenges in microarchitecture and architecture. Proc. IEEE, vol. 89, pp. 325, Mar. 2001.