ABSTRACT

BOWMAN, JOSHUA WAYNE. Microarchitectural Implementation of a Reduced x86 ISA in FabScalar-generated Superscalar Cores. (Under the direction of Dr. Eric Rotenberg.)

Heterogeneous processors present an opportunity to mix different core designs to achieve better power efficiency and performance. A wider available range of core designs allows for more customization to achieve an optimal design. However, the increase in the number of designs multiplies the required design and verification effort. The FabScalar toolset seeks to provide a core design that can be easily configured to different superscalar sizes and provide several different core configurations with little additional design effort.

Currently, FabScalar only supports two RISC ISAs, PISA and MIPS. A missing element of configurability in FabScalar is the ability to use a configurable CISC core design. The x86 ISA provides a CISC ISA that introduces several design elements not currently covered in FabScalar designs including variable instruction length and complex instructions that convey several simple operations.

In this thesis, an implementation of a reduced x86 ISA on the FabScalar toolset is presented. The primary focus of the thesis is the microarchitectural design details of the changes made to the PISA FabScalar design. Modifications were made to support four large-scale changes introduced by x86. The first was supporting the variable length instructions used by x86. Fetch must now use per-byte logic and bytes must be decoded to extract instructions. Next was expanding the logic used to split complex instructions into multiple simple microinstructions to support more complex x86 instructions. Additionally, support for a flag register was added to allow compatibility with the flag register used in x86. Lastly, modifications were made to allow for execution of instructions with variable data sizes. For
all design decisions an effort was made to do as little modification to the current FabScalar design to maintain a consistent FabScalar structure across all implementations.

The results show that the implemented core design provides robust support for the implemented reduced x86 ISA. Several benchmarks are able to successfully retire millions of instructions. Also, the core design maintains the level of configurability available with other FabScalar implementations. Finally, microbenchmarks aimed at evaluating instructions-per-cycle (IPC) show that IPC scales similarly to prior FabScalar implementations.
Microarchitectural Implementation of a Reduced x86 ISA in FabScalar-generated Superscalar Cores

by

Joshua W. Bowman

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

2013

APPROVED BY:

_______________________________
Dr. Eric Rotenberg
Committee Chair

_______________________________
Dr. Paul Franzon

_______________________________
Dr. Huiyang Zhou
BIOGRAPHY

Joshua Bowman was born in Archdale, NC, on March 12th, 1990. He earned his B.S. in Computer and Electrical Engineering from North Carolina State University in the spring of 2012. During the summer of 2012, he was an intern at Intel Corporation in Hillsboro, OR, working in validation. He returned to North Carolina State University in the fall of 2012 to pursue his Master's in Computer Engineering. He will be receiving his Master's Degree with the defense of this thesis. After completing the Master's program he will start a full-time position as a Hardware Engineer at IBM in Austin, TX.
ACKNOWLEDGEMENTS

This research was supported by grants from Intel.
TABLE OF CONTENTS

LIST OF TABLES ........................................................................................................ vii

LIST OF FIGURES ...................................................................................................... viii

Chapter 1: Introduction ................................................................................................. 1
  1.1 Motivation ........................................................................................................... 1
  1.2 Related Work ..................................................................................................... 2
  1.3 Outline ............................................................................................................... 2

Chapter 2: Background ................................................................................................. 6
  2.1 FabScalar Template ......................................................................................... 6
    2.1.1 Front-End .................................................................................................. 6
    2.1.2 Back-End .................................................................................................. 11
    2.1.3 Recovery .................................................................................................. 14
  2.2 x86 Overview ................................................................................................. 15
    2.2.1 Registers ................................................................................................. 16
    2.2.2 Instruction Parts ..................................................................................... 18
    2.2.3 Memory Operands ................................................................................. 20
    2.2.4 Instruction Operand Sizing .................................................................... 21
    2.2.5 Complex Instructions ......................................................................... 21
  2.3 Modified FabScalar Template ........................................................................... 22

Chapter 3: Fetch ............................................................................................................ 28
  3.1 Per-Byte Fetch Logic ...................................................................................... 28
  3.2 Packet Generation .......................................................................................... 30
    3.2.1 Data Fields ............................................................................................ 31
    3.2.2 Valid Field ............................................................................................ 31
  3.3 Partial Branch Fetch ....................................................................................... 32
    3.3.1 Last Byte PC Lookup ........................................................................... 33
  3.4 Fetch Buffer ..................................................................................................... 35
  3.5 Selecting Fetch Width ..................................................................................... 39

Chapter 4: Length Pre-Decode ..................................................................................... 41
  4.1 Configurability ................................................................................................. 41
  4.2 Length Decoder ............................................................................................... 42
    4.2.1 Parallel Phase .......................................................................................... 44
    4.2.2 Reduction Phase ..................................................................................... 49
  4.3 Fetch Buffer Feedback .................................................................................... 53
  4.4 Repair Last Byte PC Lookup ........................................................................ 54

Chapter 5: Control Pre-Decode .................................................................................... 56
  5.1 Control Pre-Decoder ....................................................................................... 56
  5.2 CTI Queue ........................................................................................................ 57
  5.3 Macro-Instruction Buffer ................................................................................ 58
LIST OF TABLES

Table 4.1 Maximum Length Instruction Example ..........................................................49

Table 5.1 Control Instruction Lengths ...........................................................................57

Table 6.1 Addressing Modes ......................................................................................62
Table 6.2 Updated Logical Register Encodings with Temporary Registers .................64

Table 7.1 Zero-Register Workaround Encoding Change ..............................................78
Table 7.2 8-bit Logical Register Sizing Encoding Change ...........................................80

Table 11.1 Benchmark Categories ...............................................................................108
Table 11.2 Configuration Stage Widths ......................................................................109
Table 11.3 Configuration Structure Sizes ....................................................................109
Table 11.4 Configuration Control Structure Sizes ......................................................109
LIST OF FIGURES

Figure 2.1 FabScalar Template .......................................................... 7
Figure 2.2 General-Purpose Registers [8] ............................................. 16
Figure 2.3 EFLAGS Register [8] .......................................................... 18
Figure 2.4 Instruction Format [8] ......................................................... 19
Figure 2.5 Modified FabScalar Template ............................................. 27

Figure 3.1 Per Instruction vs. Per Byte Logic ....................................... 30
Figure 3.2 First Byte Lookup vs. Last Byte Lookup ............................. 35
Figure 3.3 Bytes Used Signal .............................................................. 38

Figure 4.1 Instruction Byte vs. Pre-Decode Width ................................. 42
Figure 4.2 Instruction Bytes to Length Decoders .................................. 43
Figure 4.3 Parallel and Reduction Phases ............................................. 44
Figure 4.4 Length Decoder General Structure ..................................... 46
Figure 4.5 Length Decoder Interaction ............................................... 50
Figure 4.6 Length Decoder Valid Logic .............................................. 51
Figure 4.7 Length Pre-Decoder Validation Example ............................. 53
Figure 4.8 Control Instruction Last Byte PC Correction Logic .................. 55

Figure 6.1 Determining Operand Direction using Direction Bit ............... 61
Figure 6.2 Instruction Splitting Overview ............................................. 66
Figure 6.3 Split Simple Addition Instruction ......................................... 71
Figure 6.4 Split Complex Call Instruction ........................................... 74

Figure 7.1 Changes to Rename Structures for Option 1 ......................... 82
Figure 7.2 Changes to Rename Structures for Option 2 ......................... 84

Figure 9.1 Updated Execution Lane Data Interfaces ............................. 88
Figure 9.2 Partial Register Read Logic .............................................. 90
Figure 9.3 Partial Register Write Example with 16-bit Sizing .................. 91
Figure 9.4 Selecting Carry Bit Based on Size ..................................... 93
Figure 9.5 Operation with Carry Logic ............................................. 95

Figure 11.1 Simulation Environment ................................................. 106
Figure 11.2 Micro-IPC ................................................................. 110
Figure 11.3 Macro-IPC ............................................................... 111
Figure 11.4 Branch Mispredictions per 1K Instructions ....................... 111
Figure 11.5 Load Violations per 1K Instructions .................................. 112
Figure 11.6 Benchmark Loop Code ................................................. 113
Figure 11.7 Micro-IPC of Cores with Varying Issue Queue Size ............. 114
Figure 11.8 Micro-IPC of Cores with Varying Load/Store Queue Size ..................................115
Figure 11.9 Micro-IPC of Cores with Varying Active List Size.........................................116
Figure 11.10 Average Instruction Length .............................................................................118
Figure 11.11 Split Ratio ........................................................................................................118
Figure 11.12 Instruction Density ............................................................................................120

Figure 12.1 Modified FabScalar Template.............................................................................123
Chapter 1: Introduction

Heterogeneous multi-core processors allow different core designs to be integrated to provide the best balance of performance and power efficiency. The increased complexity of using multiple cores increases the design and verification effort required to create a well designed and validated multi-core processor.

The FabScalar project [1,2,3] seeks to solve this problem by providing a toolset that should be capable of reducing overall development time. The toolset allows the automated generation of synthesizable RTL with configurable design settings while following a consistent canonical superscalar template. This allows the independent generation of cores with various configurations for use in heterogeneous designs.

1.1 Motivation

The FabScalar toolset allows users to generate superscalar processors with arbitrary configurations. The toolset uses a structured design to make it easier for new users to understand and modify. Currently the tool is able to produce processors that support two different Instruction Set Architectures (ISAs), PISA and MIPS. PISA and MIPS are both fixed instruction length and are considered Reduced Instruction Set Computing (RISC) ISAs. The toolset lacks an implementation of an ISA with variable-length instructions or a Complex Instruction Set Computing (CISC) architecture. This leaves a major type of architecture unimplemented. The x86 ISA provides an option that fills both gaps and provides an insightful comparison to the other ISA implementations.
The x86 ISA is also one of the most popular ISAs currently in use. Many programs are written with the intent of being executed on the x86 architecture. Consequently, most tools are designed to provide extensive x86 support and achieve high performance. The motivation of this thesis is to extend the FabScalar toolset to support a variable length and CISC ISA, selected to be a reduced subset of x86.

1.2 Related Work

Several software simulators have been developed for the x86 ISA. A few examples are Bochs [4], PTLSim [5], and Zesto [6]. These simulators provide a comprehensive code source that can be used to run x86 experiments. They provide support for a wide range of x86 features by providing device emulation and operating system support. However, none of these provide RTL designs.

Modifications have been done to an existing FabScalar core in an effort to expand its configurability. For example, floating-point support was added to MIPS FabScalar [7]. This modification was a smaller scale change as the MIPS ISA was only extended rather than a new ISA being implemented.

1.3 Outline

The FabScalar toolset implementing the PISA ISA is extended to support the x86 ISA. The design choice was to reuse the PISA processor’s back-end with few large-scale changes. The front-end of the core underwent several modifications to support the change from fixed-length to variable-length instructions. In addition, the transition from CISC to RISC would be handled by splitting the x86 instructions into PISA instructions that would
execute on the reused PISA back-end. Reuse of the PISA core provided a simpler implementation that closely follows the current FabScalar template compared to creating an entirely new design.

The primary focus was on the micro-architectural modifications needed to convert the PISA implementation to support x86. Many structures were reused with as few changes as possible. For example, the physical register file and rename structures were replicated to support the flag register in x86 and to operate in parallel with the existing general-purpose physical register file. Some stages required significant modifications. Nonetheless, an attempt was made to follow the same structure used by other FabScalar implementations for consistency. The major areas of modification are:

• *Instruction Fetch and Length Decode*

Changes to the fetch stages introduced the most challenge and largest differentiation compared to the PISA implementation. The general fetch unit structure was still maintained from PISA. However, the design was heavily modified to fetch based on a number of bytes rather than a set instruction length. Length decoders also had to be introduced to break up a fetched stream of bytes into x86 instructions. Additionally, for locating branches among fetched bytes and branching accordingly, branch predictor structures must be indexed for each byte and the next-PC logic must account for partially fetched branches and for branches of different lengths.

• *Instruction Splitting*

FabScalar already implemented a mechanism for splitting certain complex instructions such as multiply and divide to two simpler instructions. This
implementation was leveraged and largely expanded to support the complex splitting needed for CISC instructions. This included expanding the maximum number of microinstructions per macroinstruction from two to five. Also, the logic needed to determine and acquire the instruction’s operands was greatly expanded because of the need to support memory locations as operands.

• **Flags Register**

A major back-end modification was the introduction of the flags register. The flags register acts as an additional implied source and destination register for most instructions. Structures needed to rename, read and write an additional register were replicated to support the use of the flags register. Support also had to be added for the instructions that read and/or modify the flags register during execution. For example, conditional branch instructions in x86 rely on the flag register value, set by previous instructions, to determine their direction.

• **Variable Operand Sizing**

Another modification had to be made to the back-end to support the execution of instructions that operate on data sizes of 8, 16 or 32 bits as supported in x86. Most changes were to the execution lanes to allow reading from and writing to parts of registers and support for executing operations and setting execution flags with different data sizes.

The thesis covers the implementation details for each pipeline stage and the modifications made to convert the PISA processor core to support x86 while maintaining configurability. Chapter 2 gives an overview of the FabScalar framework and the modified
framework with x86 support added. Chapters 3 through 10 discuss the implementation details for each pipeline stage. Chapter 11 presents results and Chapter 12 provides a summary and proposes future work.
Chapter 2: Background

FabScalar follows a canonical template that allows users to configure the superscalar width and pipeline depth for various stages of the pipeline. FabScalar supports this by first implementing the maximum supported width, depth and size for each configurable structure. This design is then modified using preprocessor macros to adjust to different widths and pipeline depths. Unused parts of the design are removed and only the necessary structures to support the desired configuration remain. Widths, depths and sizes are set using a separate configuration file external to the core design files. Performance tests can then be run to compare the performance impact of different configurations and aid in the selection of an appropriate configuration.

2.1 FabScalar Template

The FabScalar template contains nine pipeline stages divided between the front-end and back-end. The front-end is comprised of fetch, decode, rename, and dispatch. The back-end is comprised of issue, register-read, execute, writeback and retire. The template is shown in Figure 2.1.

2.1.1 Front-End

The front-end is the in-order part of the architecture. It is responsible for retrieving instructions and packaging them so they can be correctly executed on the out-of-order back-end.
Figure 2.1 FabScalar Template
The first pipeline stage of the front-end is Fetch. Fetch is divided into two sub-stages, Fetch-1 and Fetch-2. The Fetch-1 stage’s responsibility is fetching new instructions from the instruction cache and calculating the next fetch PC. Contained within the Fetch-1 stage are the Branch Predictor (BP), Branch Target Buffer (BTB), Return Address Stack (RAS) and the L1 Instruction Cache. Each instruction that is fetched has its PC looked up in the BP and BTB. The RAS is accessed if the control instruction is determined to be a call or return instruction. Fetch-1 also contains the next-PC logic that selects the next fetch PC based on location and predicted directions of control instructions.

Fetch-2 handles the detection of control instructions that were missed in Fetch-1 because of a BTB miss during lookup. A Control Pre-Decoder detects control instructions and signals Fetch-1 to revert the PC to a new address if necessary. Fetch-2 also contains the Control Queue (CTI Queue) where all control instructions are added and removed in-order. Its purpose is to update the control prediction structures in Fetch-1 in program order.

Fetch-1 and Fetch-2 both share a configurable width of N instructions known as the fetch width. Fetch-2 outputs a decode bundle to the Decode stage containing N or less instructions. If a control instruction is predicted as taken in one of the fetch stages, the instructions following it will be invalidated since they are down the wrong predicted control path. This allows the decode bundles to be as small as a single instruction depending on the presence and location of control instructions.

The Decode stage accepts up to N, fetch width, instructions directly from the Fetch-2 stage. The stage is able to decode up to N instructions in parallel by generating N instruction decoders. Each decoder accepts a single instruction and extracts the relevant data from the
instruction encoding. This extracted data is combined to form a packet containing required fields for future stages. The decoders split certain complex instructions, such as multiply and divide, into a maximum of two microinstructions. The splitting gives the Decode stage an output of up to \(2^N\) valid instructions depending on the number of valid incoming instructions from the Fetch stage and the number of instructions split. For example, a fetch width of two instructions would force the Decode stage to have an output width of four instructions to support the possible worst case where both instructions are split.

Decoded instruction packets from the Decode stage are inserted into the Instruction Buffer. The Instruction Buffer is a circular FIFO with a write width of \(2^N\) instruction packets and a read width of instruction packets equal to the dispatch width. The number of packets that can be stored in the buffer is configurable but must be at least \(2^N\) to accept the maximum number of valid incoming packets from the Decode stage. The Instruction Buffer only writes valid packets from the incoming decode bundle to the buffer. The Instruction Buffer maintains its current instruction count and signals when it cannot accept an incoming decoded instruction bundle or does not have enough instructions to create an outgoing bundle. Outgoing packets are sent to the Rename stage.

The Rename stage is responsible for completing register renaming for each of the incoming instructions. The number of access ports required for the renaming structures is based on the Rename stage superscalar width, known as the dispatch width. Renaming involves taking an instruction’s logical destination, if valid, and mapping it to a physical register popped from the Free List. Source registers are renamed using the Rename Map Table (RMT) to look up the most recent physical registers corresponding to the source
registers' logical addresses. Dependencies that exist between instructions in the same rename bundle are handled within the RMT by forwarding the most recent mapping for a destination to its dependents. The Free List keeps track of available physical registers, with physical registers removed as they are assigned to destinations and physical registers added as instructions retire. The RMT interacts with the Free List to update its mappings as destinations are renamed. The outgoing dispatch packets contain the physical register mappings for each register and are passed to the Dispatch stage.

The Dispatch stage accepts dispatch width renamed instructions from the Rename stage. The instructions are added to the Active List and the Issue Queue. Memory instructions are added to the Load/Store Queue (LSQ). The Dispatch stage also keeps track of the number of occupied entries in the AL, IQ and LSQ. It is responsible for stalling the front-end if there is not enough space for all instructions in the current instruction bundle to be added to their required structures. The Dispatch stage contains the Execution Pipe Scheduler that assigns each instruction to an execution lane based on its type. Having the Execution Pipe Scheduler in the Dispatch stage simplifies the select logic in the Issue Queue. The Dispatch stage also contains an optional Load Dependence Predictor (LDP), which can be activated in the configuration file. By default, loads execute speculatively when they encounter an unknown prior store address. The LDP may override this policy for loads that have previously mispredicted. It is trained as loads retire, using their misprediction history. If a load was previously mispredicted, it will be predicted to mispredict again, hence, it will stall until all prior store addresses are calculated.
The front-end and back-end interact directly through the Issue Queue when instructions leave the Dispatch stage. They also interact through the Active List, which maintains the program order generated in the front-end throughout back-end execution. Stages up to and including the Dispatch stage are all in-order whereas the subsequent back-end stages introduce out-of-order execution.

2.1.2 Back-End

The Issue stage buffers each instruction and sends it to the correct execution lane when its source operands are ready and a lane is available. The Issue Stage output width is set based on the number of execution lanes, known as the issue width. The Issue stage contains two key components, the Issue Queue (IQ) and Select logic.

The Issue Queue stores instructions that are waiting to be executed. The Issue Queue is a CAM+RAM structure managed using a free list. When instructions enter they acquire an entry from the free list and entries are returned to the free list as instructions leave the issue queue. The number of entries in the Issue Queue is configurable. Another part of the Issue Queue is the ready bit array. The ready bit array keeps track of which instructions are ready to be issued. Instructions are ready when both of their source operands are available. This feature allows instructions to execute out-of-order, as their dependencies are resolved.

The Select logic determines which of the ready instructions should be issued for execution. A maximum of one instruction per lane can be selected per cycle. When an instruction is selected it wakes up all dependent instructions by setting the ready bit for its destination register. This allows future dependent instructions to potentially be issued the very next cycle with the use of bypasses for value transfer. Once an instruction issues, it has
dedicated resources within its execution lane to complete register-read, execution and writeback without any further arbitration for resources.

In the Register-Read stage, the values of source operands are read from the Physical Register File (PRF). The PRF has two read ports and one write port for each execution lane. The number of physical registers in the PRF is configurable although it must be at least equal to the number of architectural registers.

After reading values from the PRF, the actual values, which the instruction uses during execution, can come from either the PRF or from another completed instruction via bypasses. The value on the bypass represents the most recent value of the source register and has priority over the value read from the PRF. The bypasses allow instructions to execute back-to-back (i.e., in consecutive cycles) without having to wait for values to be written and then read from the PRF allowing execution to complete sooner.

The Execute stage uses the source values to generate a result. There are four different types of execution lanes. The first is the Simple ALU, which is used to complete simple operations such as simple arithmetic and bitwise operations. The Complex ALU has the same structure as the Simple ALU except that it executes complex operations such as multiplies and divides. It is also possible to configure a lane that is capable of executing both Simple and Complex types of instructions. The next is the Control ALU that handles all control instructions. It computes the next-PC address and detects whether the instructions were mispredicted. The last type is the Load-Store lane that contains an Address Generation (AGEN) ALU that is responsible for calculating the addresses of memory operations. The generated addresses are passed to the LSU to complete memory operations by interfacing
with the L1 Data Cache and Load/Store Queues. The number of each type of lane present is configurable, although there must be at least one of each for a minimum of four lanes.

In the Writeback stage, completed instructions with valid destinations write their values back to the PRF. A value is also broadcast on the bypasses for dependent instructions to use the value as their source operand data. The Writeback stage also sends the execution status to the Active List signaling that the instruction has completed and providing relevant execution information such as if the instruction generated an exception or a control instruction was mispredicted.

The final stage is the Retire Stage. The Retire stage is responsible for committing completed instructions in program order using the Active List. The Active List maintains a list of all instructions in program order as well as status about their execution. The Active List uses a FIFO structure with the head of the Active List signifying the oldest instruction and next to retire. When a store instruction is retired, it signals the LSU to commit the store to memory.

If the committed instruction has a valid destination register, then first the physical destination is sent to the Architectural Map Table (AMT). The Architectural Map Table (AMT) is updated using the logical destination address with the physical register address. The AMT maintains architectural register state. When a mapping is updated in the AMT, the previous mapping is removed and sent to the Free List since the register is no longer needed and can be reallocated. Logic exists in the Active List to force instructions that were split to have their microinstructions retire together. The Active List also contains logic for
recognizing instructions that were marked by previous stages as mispredictions, load violations or exceptions, and initiating recovery, accordingly.

The described FabScalar template was modified to add support for the x86 ISA. Section 2.3 details the changes needed to FabScalar enable x86 instructions to be executed correctly.

2.1.3 Recovery

FabScalar allows speculative execution with branch prediction and speculative loads. FabScalar also is able to detect some exceptions generated by instructions. The design handles three types of recovery situations that may occur: control mispredictions, load violations and exceptions.

A control misprediction occurs when the outcome of a branch does not match the outcome predicted by the BP, BTB and/or RAS. A misprediction is detected in the Control ALU when the calculated next-PC does not match the predicted next-PC. When a mispredicted instruction reaches the head of the Active List and is ready to be retired, it is recognized by the Retire stage logic and recovery begins. Recovery involves flushing the entire pipeline, copying the physical register mappings from the AMT to the RMT, and restarting execution at the correct next-PC for the mispredicted control instruction.

A load is executed speculatively assuming previous stores do not store to the same location allowing some memory operations to execute out of program order. A load violation occurs when a load is allowed to speculatively execute but, later, the address for an older store is calculated and there is a partial or complete address overlap. When a load reaches the head of the Active List and it has been marked as a load violation, recovery begins. Recovery
is very similar to that of a control misprediction, with the pipeline being flushed and the AMT being copied to the RMT. The difference is that execution is restarted at the PC of the load that violated. This re-executes the load instruction, forcing a serialization of memory accesses and ensuring the load obtains the correct data.

Certain conditions trigger exceptions, such as a divide-by-zero condition. Other instructions trigger exceptions when they need to make a call to the operating system such as with the SYSCALL instruction in PISA. FabScalar does not directly handle exceptions but is able to detect them so that the simulation environment can handle them. Instructions are marked as generating exceptions in the Simple and Complex ALUs. When an instruction marked as causing an exception reaches the head of the Active List, exception recovery begins. Exception recovery is different from load violation and control misprediction recovery, with exceptions being handled externally. The generation of the exception signal is used to signal the simulation environment to handle the exception. The pipeline is flushed and the simulation environment is responsible for handling the exception, updating the processor state and restarting execution at the PC of the instruction following the exception-generating instruction.

2.2 x86 Overview

x86 is a CISC ISA that allows several simple operations to be conveyed with a single instruction. The x86 ISA also uses variable-length instructions. The implemented ISA is a reduced version of x86. Included are most arithmetic operations with other supporting operations such as data movement and stack management. Only integer operations are
supported. Not present are more complex operations such as SIMD, string operators and advanced features for operating system support such as permission levels and machine registers.

2.2.1 Registers

Eight general-purpose architectural registers are used in x86. The 32-bit registers have the names EAX, EBX, ECX, EDX, ESP, EBP, ESI, and EDI. As Figure 2.2 shows, 16-bit and 8-bit sections of the 32-bit registers can be addressed using different names. AX, BX, CX, DX, SP, BP, SI and DI access the lower 16 bits, bits 0 to 15, of EAX, EBX, ECX, EDX, ESP, EBP, ESI and EDI, respectively. EAX, EBX, ECX, and EDX are also able to have their lower 16-bits addressed as two 8-bit registers. AL, BL, CL and DL access the lower 8 bits, bits 0 to 7, of EAX, EBX, ECX and EDX, respectively. AH, BH, CH and DH access the upper 8 bits of the last 16 bits, bits 8 to 15, of EAX, EBX, ECX and EDX, respectively.

<table>
<thead>
<tr>
<th>General-Purpose Registers</th>
<th>16-bit</th>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>AH</td>
<td>AL</td>
<td>AX</td>
</tr>
<tr>
<td>BH</td>
<td>BL</td>
<td>BX</td>
</tr>
<tr>
<td>CH</td>
<td>CL</td>
<td>CX</td>
</tr>
<tr>
<td>DH</td>
<td>DL</td>
<td>DX</td>
</tr>
<tr>
<td>BP</td>
<td></td>
<td>EBP</td>
</tr>
<tr>
<td>SI</td>
<td></td>
<td>ESI</td>
</tr>
<tr>
<td>DI</td>
<td></td>
<td>EDI</td>
</tr>
<tr>
<td>SP</td>
<td></td>
<td>ESP</td>
</tr>
</tbody>
</table>

Figure 2.2 General-Purpose Registers [8]
The registers ESP and EBP have special uses. ESP is expected to contain the stack pointer and EBP is expected to hold the base pointer. Certain stack operations will automatically use the stack and/or base pointer registers. Although they have expected uses, there are no restrictions on reading and/or writing ESP or EBP and they can be treated like other general-purpose registers. However, this may cause the values for the stack pointer and base pointer to be clobbered and potentially lead to unexpected behavior.

EFLAGS is the 32-bit flags register used in x86. EFLAGS cannot be addressed the same way as a general-purpose register. Special instructions are used to directly read and/or modify the EFLAGS register to account for its lack of addressability. Certain instructions will modify the EFLAGS value based on their result and instruction type. There is no indication given by an instruction, besides knowing the opcode, that it will modify EFLAGS. Figure 2.3 shows the flags and their indices within EFLAGS. The common arithmetic flags that many instructions read and/or write are the carry flag (CF), zero flag (ZF), sign flag (SF) and overflow flag (OF), and these are the only currently supported flags in this work. Other flags, including all system, control and other status flags, are not supported.

The different flags are set based on the result of an operation. Instructions may modify some or all of the flags. Some of the flags have clear interpretations. For example, the zero flag is set when the result is zero for most operations. However, depending on the instruction type, a flag’s expected value may not be clear. For instance, the overflow flag has no clear meaning for a bitwise operation such as AND or XOR. For these cases, the method of setting the flag is defined by the ISA. Some instructions will leave a flag as undefined if it does not set the flag to a consistent value.
Conditional jump instructions base their direction on values within the flag register. Each conditional jump tests a different flag or combination of flags based on its condition. For instance, the JS instruction, jump if sign, will select its direction based on the sign flag.

![EFLAGS Register](image)

**Figure 2.3 EFLAGS Register [8]**

### 2.2.2 Instruction Parts

The parts of an x86 instruction include prefixes, opcodes, ModR/M, Scale-Index-Base (SIB), displacements and immediates. The opcode is the only required part of the
instruction. The prefixes, opcode and ModR/M are used to determine the existence and sizes of the other fields. The possible sizes of the fields are seen in Figure 2.4 although not all shown combinations are supported. The existence and sizes of these fields are used to calculate the total length of instructions. Based on the sizes of fields and currently supported instruction the largest possible instruction is 12 bytes and the shortest is 1 byte.

![Instruction Format Diagram](image)

Figure 2.4 Instruction Format [8]

The first possible part of an instruction is a prefix. Prefixes occur before the instruction opcode. Each prefix can only occur once and must appear in a certain defined order if multiple prefixes are used. Some prefixes, such as the 16-bit operand size override prefix, provide data about the total length of an instruction.

The next and only required part of the instruction is the opcode. The opcode can either be 1 or 2 bytes in the current implementation and is unique for every instruction. The opcode signifies the operation type of the instruction. Certain opcodes can also specify registers, operand type and other properties of the instruction.
The ModR/M and SIB byte are used for determining the type of operands, memory or register, used for an instruction and for calculating memory addresses for memory operands. The presence of the ModR/M byte depends on the type of instruction, determined by the opcode, although most instructions use it. The presence of the SIB is directly dependent on the ModR/M value. The SIB is only used for memory operands, in order to express additional addressing modes that cannot be expressed only the ModR/M value.

The last parts of the instruction are the displacement and immediate values that can both be 1 to 4 bytes each if they are present. The displacement value is used in address calculations and its existence and size are based on the ModR/M and SIB byte values. The immediate value’s existence and size is based on the opcode, which signals if the instruction uses an immediate operand.

2.2.3 Memory Operands

CISC architectures introduce the ability to use memory locations as operands and x86 supports this feature. When an instruction has a memory source operand, it is inferred that a load operation will be used to acquire it. If the memory operand is a destination, then a store operation is expected to store the result. The x86 ISA allows only one of the defined operands to be a memory location. However, most x86 instructions have an operand that acts as both a source and destination. Therefore, a single instruction may require a load to acquire the original source data and a store to save the updated result.

Several addressing modes are supported for a memory operand. The addressing mode is specified using the ModR/M and SIB parts of an instruction. The supported addressing modes are absolute, register, register-register and register-register-displacement. The x86
ISA also supports the ability to left-shift one of the register values used in the address calculation up to 3 bits.

2.2.4 InstructionOperand Sizing

Certain instructions support different operand sizes. The supported operand sizes are 8, 16 and 32 bits. The operand size for an instruction is determined based on the instruction’s opcode as well as the presence of prefixes. For register operands, an operand size of 8 or 16 bits involves reading and writing parts of registers. For memory operands, it involves loading and storing smaller data sizes.

The execution of an instruction is also affected by sizing. Results may vary based on operand size. For instance, the PUSH instruction must update the stack pointer based on how many bytes it stores to the stack which is based on operand sizing. An additional execution change is the setting of the flags register based on operand sizing. The operand size of an instruction will determine how the result will be interpreted and whether or not a flag will be set. For instance, detecting an overflow condition for an 8-bit operation is different than detecting an overflow condition for a 32-bit operation.

2.2.5 Complex Instructions

Previously discussed was the ability for a CISC instruction to use memory operands. This introduces a single CISC instruction requiring multiple operations to be completed including load and/or store operations. However, x86 introduces other, more complex examples of CISC instructions that require a unique set of simple operations to complete.
One example of this is the PUSH instruction. This instruction is used to push a data value onto the stack. This instruction requires an operation to store the data to the stack as well as an operation to update the stack pointer. This set of operations is unique to this specific PUSH instruction. This contrasts with memory operands, which use a common format utilized by several different instructions.

One other example of a complex instruction is the RET instruction. This instruction is used to return from function calls. This instruction requires an operation to read a return PC from the stack, update the stack pointer, and jump to the PC read from the stack. As with the PUSH instruction, this set of operations is unique to the RET instruction.

There are several other examples of these complex x86 instructions that require several simple operations to complete. They all share the characteristic of being unique from one another and not following any generalized format among them.

2.3 Modified FabScalar Template

Figure 2.5 shows the modified FabScalar template with support for the reduced x86 ISA added. This overview of the modified FabScalar template details the changes from the standard FabScalar template of Section 2.1. Design details for the modifications as well as reasoning for design choices are presented for each stage in the following chapters. The main changes to the FabScalar Template were:

1. Fetch was changed to fetch on a per-byte basis rather than per-instruction. This impacts the next-PC logic in two ways. First, the PC of the next sequential fetch bundle is based on the number of bytes fetched instead of the number of instructions.
Second, to detect and predict branch instructions within the current fetch bundle, each fetched byte must be treated as a potential branch instruction. This significantly increases the complexity of the branch prediction structures and logic. In particular, branch prediction structures had to be changed to support a higher number of parallel accesses so that each byte-level PC could be searched rather than a single PC per eight bytes.

2. A Fetch Buffer was added between the Fetch and Pre-Decode stages. The Fetch Buffer contains the fetched bytes along with control prediction information combined into a packet. The inclusion of the Fetch Buffer allows Fetch and the following stages to have different superscalar widths. Decoupling Fetch and Pre-Decode reduces a burden that Pre-Decode would otherwise place on Fetch width: our Pre-Decode stage implementation must be supplied the maximum number of bytes in an x86 instruction, which is 12 for the reduced x86 ISA. Per-byte fetch logic for a worst-case instruction length of 12 bytes is unduly complex, and unnecessary as the average instruction length is much lower than that. The Fetch Buffer accumulates and maintains the required width for Pre-Decode without burdening the Fetch stage with this width.

3. A Length Pre-Decode stage was added after Fetch. This change was made because each of the following stages operates on an instruction level, and instruction boundaries within the fetch bundle are not fixed for the variable-length ISA. The Length Pre-Decode stage uses multiple length decoders to determine the lengths of all instructions in the fetch bundle, in parallel. Our approach to parallel length decoding
assumes that each byte in the fetch bundle could be the start of an instruction, and sorts out actual instruction boundaries in an overall reduction phase among length decoders. The length decoders also extract fields needed for future stages such as opcodes and immediate values.

4. Fetch2 was renamed the Control Pre-Decode stage, as its primary purpose is to pre-decode control instructions to determine their control type and start recovery for missed control instructions. Some of the fetch-related logic was relocated from this stage to the Fetch stage, further removing its relation to Fetch.

5. The Control Pre-Decoders had to be changed to detect control instructions and calculate their target PCs based on x86 encodings. The Control Pre-Decoders use the information extracted from instructions in the Length Pre-Decode stage, such as the opcodes and immediate values, to make its calculations.

6. An instruction buffer was added after the Control Pre-Decode stage. The buffer is known as the Macro-Instruction Buffer and contains macroinstruction (x86) packets. The inclusion of the Macro-Instruction Buffer allows the Pre-Decode stages (Length Pre-Decode and Control Pre-Decode) and the following stages to have different superscalar widths. Moreover, the Macro-Instruction Buffer reflects the conversion from a variable-length instruction encoding to a fixed-length instruction encoding of 12 bytes (the maximum length): each macroinstruction entry in the buffer is 12 bytes regardless of the actual number of bytes used by the macroinstruction.

7. The Decode stage’s instruction splitting feature is significantly expanded for the reduced x86 ISA as compared to PISA. First, instead of a maximum split into two
microinstructions, the Decode stage can now split complex instructions into as many as five microinstructions. Second, the splitting is also more complex and occurs for a higher percentage of instructions. The increase in the maximum number of microinstructions emitted for a single macroinstruction (from two to five) also increases the input width of the existing instruction buffer following the Decode stage. With the addition of the Macro-Instruction Buffer, this existing buffer is now referred to as the Micro-Instruction Buffer since it contains microinstructions (PISA-like instructions) generated by splitting macroinstructions (x86).

8. Support was added for renaming the flag register. Essentially, this involved replicating the renaming structures used for integer registers – the PRF, RMT, and Free List – yielding the Flag PRF (FPRF), the Flag RMT, and the Flag Free List. The renaming structures for the flag register only differ from their integer register counterparts in their configurations, as they are able to have an independent number of logical registers and physical registers.

9. The addition of the flag register adds an additional source register and/or destination register to some instructions. The Issue Queue was changed to set instruction readiness based on the conventional source registers and the additional flag register. Also, the logic needed to issue dependent instructions in consecutive cycles, intending to use bypasses, was updated to account for the flag register.

10. The execution lanes had their Register Read and Writeback stages add support for accessing the Flag Physical Register File. Execution units were modified to accept
and return a flag value when valid. Bypasses were also added to support the bypassing of the flag register result to dependent instructions.

11. New microinstructions were added to the Control ALU to calculate the next-PC and direction for x86 control instructions. x86 follows a format different from PISA, by virtue of testing the flag register, and therefore not allowing the existing microinstructions to be used.

12. New microinstructions were added to the Simple ALU, to set/clear flags in the flag register, implement new logical operations (such as the use of a carry bit), and perform several other novel operations above and beyond PISA.

13. Support was added to the Complex and Simple ALUs for setting the flag register based on the outcome of an operation. Most PISA operations retained their previous format with extra logic added for analyzing the result to determine the flags.

14. The Simple and Complex ALUs are now able to operate on data of various sizes, namely, 8 bits, 16 bits, or 32 bits, depending on the size specified by the instruction. All the supporting logic is contained within the execution units, with no changes to reading or writing of the physical register file.

15. Recovery methods were updated to additionally restore the flag register renaming structures when state needs to be restored.

The modified template attempted to closely follow the original FabScalar template. Resources were reused whenever possible such as the register file and rename structures replicated for the flag register. In addition, an attempt was made to make minor changes to the back-end so that it would still be capable of executing PISA instructions.
Figure 2.5 Modified FabScalar Template
Chapter 3: Fetch

Fetch is the first stage of the pipeline. Its primary purpose is to fetch instruction bytes from the instruction cache and manage access to the control prediction structures. The fetched bytes and prediction data are combined into packets that are inserted into the Fetch Buffer. The previous FabScalar Fetch stage relied on PISA instructions being a fixed length. Most of the changes in the Fetch stage involve modifications to support variable-length instructions.

3.1 Per-Byte Fetch Logic

The previous FabScalar implementation based the configurability of the Fetch stage on a number of instructions. With variable-length instructions, the number of instructions contained in a group of bytes is unknown. Therefore, the Fetch stage needed to be changed to fetch on a per-byte basis.

The first step was to modify the Instruction Cache to return a group of bytes rather than a group of instructions. The Instruction Cache had been designed assuming that the bytes fetched were always 8-byte aligned, based on the 8-byte size of PISA instructions. This allowed the lower three-bits of the PC to no longer be needed. The lower bits of the PC were restored to allow individual bytes to be addressed. Other logic in the Instruction Cache based on a fixed length assumption was modified to be variable based on a configurable number of bytes. The Instruction Cache can no longer fetch based on a fixed 8-byte boundary and may need to perform multiple accesses to fetch all needed bytes.
The next-PC logic also had to be changed. The structure already existed to increment the PC based on a constant equal to the fetch width. However, this constant was changed to represent the number of bytes fetched rather than a number of instructions. With the PC now a 32-bit value, all updates to the PC must be on a byte level. Also, other values used to set the next-PC, such as addresses from BTB banks and the RAS, are now 32-bit values to match the increased width of the PC.

Another area of modification was to support the logic needed for checking every byte fetched rather than once per 8-byte instruction. For instance, the PISA implementation could fetch two instructions, or 16 bytes, and have logic that operates on each of the two instructions. However, the updated implementation needs to be modified to now support operating on each of the 16 bytes separately. Figure 3.1 provides an overview of the change with each byte now requiring dedicated logic. Most changes were simple since configurability was already supported. The changes involved expanding the configurability to support the handling of more pieces of data. The same process was followed that would have been used to increase the maximum Fetch stage width in the original FabScalar implementation.

An additional change is to the prediction structures. Since no instruction information is available, every byte must be treated as a potential control instruction. Every byte fetched must now use its PC to access the BP and BTB in parallel. To allow these parallel accesses, the BTB and BP are now configured to be as highly banked as the number of bytes fetched. This involved modification of the configurability of the BP and BTB structures and surrounding logic to now be configured to support a fetch width up to 16 bytes. The BTB and
BP lookup are also affected by the increase in the PC to 32 bits to support a byte-addressable PC. This PC change had other consequential changes, such as the tag used to identify instructions in the structure being wider and requiring more storage.

![Figure 3.1 Per Instruction vs. Per Byte Logic](image)

3.2 Packet Generation

Output packets must now be generated for every byte rather than per-instruction. In order to handle this correctly, each byte must be treated as a potentially complete instruction. Extra information must be saved for each packet. Additionally, the updated structure complicates the setting of the valid field for packets.
3.2.1 Data Fields

Many of the data fields remained the same as the previous implementation. However, some changes were needed since each byte must be treated as an instruction. The PC must now be saved for each byte in its packet rather than per instruction. Control information, such as whether or not the byte was in the BTB and its predicted direction, must also be determined and saved for each byte packet.

“addrRAS” is the address supplied by the RAS. It is selected as the next PC when a return is detected in the fetch bundle by the BTB. Only one “addrRAS” value is generated per fetch bundle. Previously, this value was transferred directly from the Fetch1 stage to the Fetch2 stage. However, the introduction of the Fetch Buffer now requires this address to be saved per packet. The next stage is able to read packets that may be from different fetch bundles that may require different “addrRAS” values. Therefore, “addrRAS” must be saved per packet to guarantee the correct address is used.

3.2.2 Valid Field

Validation logic marks each packet as a valid byte that can be used by future stages. This is set based on whether or not a byte is down the predicted control path. Control instructions are detected based on hits in the BTB. The logic works by detecting the first predicted taken control instruction in a group of bytes and setting all packets in the bundle after the taken instruction as invalid. The next-PC is set based on this first found taken control instruction so all following packets are no longer valid. The PISA FabScalar implementation completed this validation in the Fetch2 stage. This implementation moved this logic to the Fetch stage.
Another issue with the validation logic is determining which bytes to mark valid for the control instruction that is predicted as taken. Whenever a control instruction is predicted as taken, all the bytes of the instruction need to be marked as valid but since the length of the instruction is not known it is not possible to determine how many bytes to mark as valid. Changes to the control prediction logic detailed in the next section explain the method used to handle this.

3.3 Partial Branch Fetch

The length of control instructions in x86 varies between instructions. With x86 the PC of an instruction is based on its first byte. Therefore, a BTB hit based on a lookup using this PC does not provide information on the length of an instruction and only guarantees that the first byte of the instruction has been fetched. A method to determine the length of the detected control instruction is needed so that all the bytes of the instruction can be fetched before the PC is updated to follow the predicted target.

The instruction length is required since once a control instruction is predicted taken it will immediately update the next-PC causing the next fetch PC to be down the predicted path based on the target provided by the BTB. The complication that occurs is fetching the first byte of a control instruction and receiving a BTB hit does not guarantee that the entire instruction has already been fetched. For instance, if the fetch width is sixteen bytes and there is a BTB hit on the last fetched byte, then the fetch unit would signal to update the next-PC to follow the predicted target assuming the control instruction is predicted as taken. However, if the control instruction is larger than a single byte, then the PC will be updated and the
remaining bytes of the control instruction will never be fetched. This creates a need for a method to access the BTB and BP for each fetched byte while being able to guarantee that the entire control instruction has been fetched if there is a hit.

Two methods were explored. The first is to acquire the length of the control instruction in the Fetch stage. This length is then used to check that all bytes of the control instruction have been fetched and marked as valid. The problem with this method would be acquiring the instruction length. Adding length decoders to the fetch stage to determine the length of instructions could do this. This would greatly increase the logic complexity of the timing-critical Fetch stage. Another, less logic-intense option would be to add the instruction length to the BTB entry and have it returned as part of the data for a BTB hit. However, even if the instruction length can be easily obtained, complex logic is still needed to fetch the missing bytes for a detected control instruction. For instance, if by using the instruction length it was determined that 10 bytes of the instruction were missing and the fetch width was only 4 bytes, then there would need to be complex state-machine logic to finish fetching the remaining 10 bytes over 3 cycles.

The other method that was implemented was to train the control prediction structures (BTB and BP) based on the PC of the last byte of control instructions. This would guarantee the entire instruction was fetched before it would cause a BTB hit. This solution did not require complex logic to be added to the Fetch stage and was simple to implement.

3.3.1 Last Byte PC Lookup

The implemented solution to the partial branch fetch problem bases BTB and BP entries on the last byte PC of control instructions. The principle behind this method is that if
the last byte of an instruction has been fetched then it can be guaranteed that the entire instruction has been fetched. This method works by the BTB and BP having their entries added based on the last byte of an instruction rather than the first.

The selected design is implemented by adding the length of an instruction to its first byte PC when updating the BTB and BP entries. This is supported by adding the length of an instruction as one of the pieces of data that is provided for a prediction update. This method does not require the length to be saved in the BTB since updating the instruction PC before modifying a BTB and BP entry is implicitly saving the length. Implementing this method requires very little modification to the Fetch stage except to adjust the BTB and BP update logic to complete the addition to compute the last byte PC. There is some complication added to later stages which are designed based on the assumption that a BTB hit is marked based on the beginning of an instruction. However, the later stage modifications that are described in later sections are generally simple modifications compared to the changes needed to implement the other methods.

The RAS also needed a minor adjustment owing to the fact that different call instructions have different lengths. The return target that gets pushed onto the RAS is calculated as the call instruction’s PC plus its length in bytes. Conceptually, this is the same as before, except the increment value is variable instead of fixed.

Figure 3.2 demonstrates the partial branch fetch problem and how the last byte PC lookup solves the issue. A five-byte control instruction, Jz 0xDEADBEEF, whose instruction encoding is shown, is detected by a BTB hit in a core with a fetch width of four bytes. By using the first byte lookup, only the first few bytes of the instruction are retrieved and the
others are never fetched since the PC is set the following cycle based on the target of the branch. However, with the last byte lookup, all bytes of the control instruction are fetched since the second cycle fetched the next sequential bytes after fetching the first few bytes of the control instruction.

<table>
<thead>
<tr>
<th>Control Instruction Encoding</th>
<th>First Byte Lookup</th>
<th>Last Byte Lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x75 0xEF 0xBE 0xAD 0xDE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle N</th>
<th>PC = PC + 4</th>
<th>PC = PC + 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 0</td>
<td>0x01</td>
<td>Byte 0</td>
</tr>
<tr>
<td>Byte 1</td>
<td>0x23</td>
<td>Byte 1</td>
</tr>
<tr>
<td>Byte 2</td>
<td>0x75</td>
<td>Byte 2</td>
</tr>
<tr>
<td>Byte 3</td>
<td>0xEF</td>
<td>Byte 3</td>
</tr>
</tbody>
</table>

**BTB Hit**

<table>
<thead>
<tr>
<th>Cycle N + 1</th>
<th>PC = PC + 0xDEADBEEF</th>
<th>PC = PC + 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 0</td>
<td>0x01</td>
<td>Byte 0</td>
</tr>
<tr>
<td>Byte 1</td>
<td>0x23</td>
<td>Byte 1</td>
</tr>
<tr>
<td>Byte 2</td>
<td>0x45</td>
<td><strong>Next Instruction Bytes Down Taken Path</strong></td>
</tr>
<tr>
<td>Byte 3</td>
<td>0x67</td>
<td></td>
</tr>
</tbody>
</table>

**BTB Hit**

Figure 3.2 First Byte Lookup vs. Last Byte Lookup

3.4 Fetch Buffer

In the PISA FabScalar release, the widths of the Fetch1 and the Fetch2 stages were required to be the same since packets were passed directly between them. This limits the
configurability and the introduction of variable-length instructions makes the implementation more difficult to support.

One issue introduced is that the Length Pre-Decode stage may not be able to use all the bytes fetched. Previously, when instruction length was fixed, the Fetch2 stage would be guaranteed to use all bytes it was given. However, with variable-length instructions, out of all the bytes fetched, it is possible that only a fraction of the bytes may be used. This creates a dependence between the Fetch and the Length Pre-Decode stages, where the Fetch stage cannot update the PC without knowledge of how many of the bytes were actually used.

Additionally, the width of the stage following Fetch, Length Pre-Decode, has a minimum width of twelve bytes in order to handle the maximum-length instruction. This forces the Fetch stage to be at least 12-bytes wide. Also, the Length Pre-Decode stage may benefit from a width even larger than the minimum of twelve bytes to extract more instructions. In order to accommodate these large widths, the Fetch stage would have to support a large fetch width. This would force the control prediction structures to be highly banked in order to look-up \textit{fetch width} bytes each cycle. Other logic needed on a per-byte basis would also need to be enlarged in order to support the high number of bytes being fetched per cycle.

The solution to these issues is the introduction of the Fetch Buffer that is located between the Fetch and Length Pre-Decode stages. The Fetch Buffer is a circular FIFO structure with a configurable size. The write width is equal to the Fetch width and the read width is equal to the Length Pre-decode width. Fetch pushes packets to the tail of the buffer while Length Pre-Decode pops the packets from the head of the buffer. The Fetch Buffer
follows a similar structure to the Instruction Buffer used in the previous FabScalar implementation.

The immediate benefit of adding the Fetch Buffer is the ability for the Fetch and Pre-Decode stages to have different widths. This allows the width of the Fetch stage to be less than twelve, the minimum width of the Length Pre-Decode stage. The Fetch stage can now be designed so that a reasonable amount of logic complexity is present based on the desired core complexity and performance. This also expands configurability by allowing Fetch width to be sized independently from all other widths in the processor.

The logic needed to allow the Fetch Buffer to support partial reads by the Pre-Decode Length stage is more complex. The Fetch Buffer must rely upon a delayed update from the Length Pre-Decode stage to update its state after each read. The Length Pre-Decode stage accepts a certain number of bytes read from the Fetch Buffer but may only use some of them. For instance, assume the Length Pre-Decode stage reads twelve bytes from the Fetch Buffer but is only able to use ten bytes. The leftover two bytes must not be removed from the Fetch Buffer since they have not been used and must be saved to be used in the future. Instead, the Length Pre-Decode stage returns the number of used bytes, 10, as the “bytesUsed” value. The method used to generate this signal is described in the next chapter discussing the Length Pre-Decode stage.

When the Length Pre-Decode stage reads its bytes, the Fetch Buffer does not remove these packets. Instead the Fetch Buffer logic does nothing based on the initial read. The Fetch Buffer is updated by the “bytesUsed” value that comes from the Length Pre-Decode stage. The Fetch Buffer removes the number of bytes specified by the “bytesUsed” value since
these bytes were successfully used and are no longer needed. Figure 3.3 shows the interaction between the Fetch and Length Pre-Decode stages.

Since the Fetch Buffer is separated from the next stage by a pipeline register, there is a single-cycle delay between when the Fetch Buffer outputs data to when the Length Pre-Decode stage receives the data. This creates a delayed response for the Fetch Buffer adjusting its output based on the “bytesUsed” value. Based on this design, there will always be a single-cycle delay while waiting for the Fetch Buffer to react to the “bytesUsed” value and return the correct bytes for the Length Pre-Decode stage to use. However, the higher density of instructions allowed by variable-length instructions, as well as the increased number of operations conveyed per instruction, should help compensate for this delay.

![Figure 3.3 Bytes Used Signal](image-url)
3.5 Selecting Fetch Width

Previously, selecting a fetch width guaranteed a throughput of a certain number of instructions fetched per cycle. This width also set the width of the following stages since they are directly connected without the use of buffers. The introduction of the Fetch Buffer and the variable lengths of instructions complicate the decision of setting the fetch width.

In general, the fetch width can limit the performance of the rest of the processor as it controls the number of instructions that will be fetched per cycle. For instance, suppose the processor width is universally four instructions wide and the fetch width is only eight bytes; in this case, if the majority of the instructions being fetched are longer than two bytes, then fewer than four instructions would be fetched per cycle. This would cause a bottleneck in the Fetch stage by starving the later stages of instructions they could handle. This makes setting the fetch width important to the overall performance.

The setting of the number of fetch bytes can be based on many factors. For instance, if performance were the primary concern, then making the fetch width as large as reasonably possible would guarantee the most number of potential instructions being fetched per cycle. The cost would be more resources needed to read more bytes as well as more complex logic to handle the bytes, such as prediction structure banking and Fetch Buffer write width. If the desire were to minimize power, then the fetch width could be set low, so that only a few bytes are read, creating a simple structure surrounding the bytes and quick cache accesses.

A reasonable starting point for setting the fetch width is to base it on the average instruction length. The average instruction length can be determined from representative benchmarks. It can be accurately measured by running the benchmark and observing the
average length of the dynamic instructions executed, or it can be estimated by analyzing the static instructions in the binary. The average instruction length can then be used with the sizing of other processor structures to determine a good fetch width. For instance, if all other stages of the processor are 4-wide and the average instruction length is 3 bytes, then selecting a fetch width of 12 bytes should, on average, provide sufficient fetch bandwidth.
Chapter 4: Length Pre-Decode

The Length Pre-Decode stage is the first of the two pre-decode stages. This stage extracts x86 instructions from a group of fetched bytes. The lengths of the instructions are calculated and used to locate the boundaries between instructions within the unorganized bytes. After locating the instructions, relevant data needed for future stages is extracted and placed into an instruction packet.

4.1 Configurability

The Length Pre-Decode stage is a new stage introduced to support variable-length instructions. The width of this stage is set to the pre-decode width which is the width for both pre-decode stages, Length and Control Pre-Decode. The width of the stage represents the maximum number of instructions that can be detected in a group of bytes. The number of Length Decoders generated is equal to the width of the stage to handle the maximum case.

Another parameter that is unique for the Length Pre-Decode stage is the instruction byte width. This value represents the number of bytes read from the Fetch Buffer. This width determines the total number of bytes that are examined to check for instructions. This value must be equal to or greater than the superscalar pre-decode width of the stage so that every Length Decoder will receive at least one unique byte. In order for an instruction to be extracted, every byte of the instruction must be present in the stage. Therefore, there are cases where the beginning of an instruction is detected, but the entire instruction has not been read from the Fetch Buffer. The instruction byte width allows extra bytes that may not start new instructions but can provide needed bytes to complete other instructions. Varying the
instruction byte width allows more complete instructions to potentially be detected without increasing superscalar width of the stage. Figure 4.1 illustrates the difference between instruction byte width and pre-decode width. In the figure, the pre-decode width is four, however, the larger instruction byte width of eight allows additional bytes to be analyzed.

Figure 4.1 Instruction Byte vs. Pre-Decode Width

4.2 Length Decoder

The Length Decoder’s primary function is to accept a group of bytes and, based on the assumption that the first byte is the start of an instruction, determine the length of the instruction. Each Length Decoder can accept up to the maximum number of bytes needed for the longest instruction, twelve bytes, although not all bytes it receives may be valid. The
Length Decoders receive overlapping byte sequences that are staggered by one byte, as illustrated in Figure 4.2 for a pre-decode width of four and an instruction byte width of fourteen. For this case, each Length Decoder receives twelve bytes and can detect the maximum length instruction. However, it is possible that a Length Decoder could receive less than the maximum twelve bytes. In this case, the Length Decoder would only be able to detect an instruction no longer than the number of valid bytes it receives. The assumption that the first byte is the start of an instruction will often be incorrect causing the instruction the Length Decoder finds to be invalidated. This aspect is discussed next.

![Figure 4.2 Instruction Bytes to Length Decoders](image)

The decoding of instruction lengths is broken into two phases, parallel and reduction. In the parallel phase, all Length Decoders determine their lengths in parallel. The reduction phase determines which Length Decoders started decoding at the beginning of a valid
instruction and validates only these instructions. This reduction phase proceeds serially from one Length Decoder to the next, since the validity of a Length Decoder depends on prior decoded lengths. Figure 4.3 shows an overview of the two phases. The reduction phase is dependent upon the completion of the parallel phase.

![Figure 4.3 Parallel and Reduction Phases](image)

**Figure 4.3 Parallel and Reduction Phases**

### 4.2.1 Parallel Phase

The primary outcome of the parallel phase is determining lengths based on the bytes supplied to the Length Decoders. Each Length Decoder accepts twelve consecutive bytes, the largest possible size of an instruction. The Length Decoder assumes that the first byte is the
beginning of an instruction. However, whether or not the instruction length is valid is determined based on feedback from prior Length Decoders during the reduction phase. The types of instructions that the Length Decoders must determine the length for vary widely in structure. However, a general approach is used in the decoders that allow them to handle all the possible cases in an efficient manner. The decoding method is based on the set structure of x86 instructions.

Figure 4.4 shows the overall decoding flow. First, prefixes are checked. Next, the opcode is found and based on its value the existence of an immediate value and/or a ModR/M byte is determined. The ModR/M byte is then used to check for the existence of a displacement or a SIB byte. If the SIB byte exists, then it will determine the existence of a displacement. Each element will contribute to the length and generate a final total length for the instruction.

The first step is checking for the existence of one-byte prefixes. The method for detecting prefixes is to check the next byte, in sequence, for a match for each possible prefix. Certain ordering rules govern the order that prefixes will appear if they are present, simplifying the logic. Since no opcodes share the same values as the prefixes, it is safe to assume that any match of the next byte represents a prefix and the first non-match signals the end of prefixes, if any. Once a prefix is located, the matching byte is removed and the following byte is now treated as the next byte to be examined. A single-bit flag is set for each prefix noting its existence for later pipeline stages. The total length of the instruction increases by one byte for each prefix found, for a maximum length of three and a minimum length of zero after checking for prefixes.
The first byte following prefixes is always the opcode. Since all prefixes have been removed and all instructions must have an opcode, the next byte after removing any prefixes is the opcode. The opcode is saved in its entirety since its exact value will be needed for further length decoding and in future stages. The next step is to use this opcode value to determine lengths of other parts of the instruction. The addition of the opcode creates a minimum length of one and a maximum length of four at this point in the decoding process.

Following the opcode bytes are all optional bytes. The Length Decoder is designed to handle the common-case instructions. The common case assumes an instruction with an opcode followed by a ModR/M byte and related addressing data without any immediate. The
The common-case method of determining operand size is to check the last bit of the opcode. If the last bit is a one, then the operand size is 32 bits and, if it is zero, then the operand size is 8 bits. This operand size is overridden by the existence of a 16-bit operand prefix, which has priority and sets the size to 16 bits regardless of opcode. Special handling must be done for the other cases that do not follow the common format. For these special instructions, the operand size and the existence of the other optional fields such as the ModR/M byte and an immediate must be determined on a case-by-case basis.

After the case logic has handled the common and special cases, a flag will have been set telling whether or not the instruction contains a ModR/M byte. The ModR/M byte will always follow the opcode if it exists. This byte contains information for the source and destination operands to be used for the operation. The instructions that do not use a ModR/M byte, and therefore did not set the ModR/M flag, will skip this decoding of the ModR/M byte.

For the normal RISC-like instruction, where both operands are registers, then the ModR/M byte does not signal any extra length for the instruction besides itself. However, with CISC-like instructions, there is an ability to support memory operands, which adds complexity to calculating the length. First of all, the ModR/M byte can signal the existence and size of a displacement used as an offset to a memory address. For instance, if the ModR/M byte indicates a 32-bit displacement, then the length of the instruction is increased by four.

Besides a displacement, the other effect the ModR/M byte can have on length is signaling the existence of the SIB byte. The SIB byte is very similar to the ModR/M byte, as
it encodes certain complex addressing modes used to load and/or store operands. However, similarly to ModR/M, the only effect it has on the instruction length is signaling the existence of a displacement. Between the ModR/M and SIB bytes, only a single displacement can be used and the ISA has been defined so that there is a consistent format followed so that a single, consistently sized displacement is determined for each ModR/M and SIB byte combination. Even if there is no displacement, the existence of the SIB byte still signals an increase in length to account for itself.

After the ModR/M and SIB bytes have been used, they are saved to the instruction packet for future use. The displacement value will follow the ModR/M byte or the SIB byte. The displacement value is extracted and saved to the instruction packet as well. At this point, after handling the ModR/M bytes, SIB byte and displacement, there is a minimum length of one byte (opcode) and a maximum length of ten bytes (three prefixes, opcode, ModR/M, SIB, four-byte displacement).

The last step is extracting the immediate value between zero and four bytes. Based on the immediate size, which was set in previous logic, a certain number of bytes are removed and then saved to the packet. This final step arrives at a minimum total length of one byte and a maximum total length of twelve bytes based on the possible combinations of instruction parts.

After all previous calculations have been completed, the last step is to compute the total length. Adding up subtotals for each of the previous steps to compute a total does this. For example, the maximum-length 12-byte instruction would be composed of one prefix (1 byte), an opcode (1 byte), a ModR/M and SIB byte (2 bytes), a 32-bit displacement (4 bytes)
and a 32-bit immediate (4 bytes). Table 4.1 shows an example of a possible maximum-length instruction and its encoding.

<table>
<thead>
<tr>
<th>Assembly</th>
<th>lock or DWORD PTR [edx+ebx*4+0x1234567], 0x89abcdef</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete Encoding</td>
<td>F0 81 8C 9A 67 45 23 01 EF CD AB 89</td>
</tr>
<tr>
<td>Prefix</td>
<td>F0</td>
</tr>
<tr>
<td>Opcode</td>
<td>81</td>
</tr>
<tr>
<td>ModR/M</td>
<td>8C</td>
</tr>
<tr>
<td>SIB</td>
<td>9A</td>
</tr>
<tr>
<td>Displacement</td>
<td>67 45 23 01</td>
</tr>
<tr>
<td>Immediate</td>
<td>EF CD AB 89</td>
</tr>
</tbody>
</table>

After the length of an instruction has been determined, the last step of the parallel phase is to generate the output packet for the possible instruction. Generating the output packet is simple and involves copying the extracted fields from the incoming bytes to the output packet, such as the opcode, ModR/M byte, SIB, displacement and immediate. Single bit flags are set to denote the existence of prefixes. Also, relevant data is copied from the input packet of the first byte into the Length Decoder, such as the PC and control information, to the output packet.

4.2.2 Reduction Phase

Determining whether to mark output packets as valid is the primary outcome of the reduction phase. Interaction is required between a given Length Decoder and the one
preceding it, to determine whether or not it was correct in treating the first byte it received as the beginning of a valid instruction. The general strategy used requires each Length Decoder to communicate with its neighboring Length Decoders. Figure 4.5 illustrates the serialization of Length Decoders to accumulate the lengths of valid instructions and use the cumulative length at each step to validate or invalidate a Length Decoder.

The index of a Length Decoder identifies it uniquely and relates to which byte in the pool of bytes was given as the Length Decoder’s first byte. To begin, the first Length Decoder will be given an input total length of 0. It will then compare its index, which is 0 for the first Length Decoder, to this input total length. Since its index matches the input total length, it marks its packet as valid. This Length Decoder will then pass its input total length plus the length of its correctly-decoded instruction to the next Length Decoder. For instance, if the first Length Decoder analyzed an instruction that was 3-bytes long, then it will pass 3 (0+3) to the next Length Decoder. The first Length Decoder will always mark its instruction as valid. This occurs since the first byte read from the Fetch Buffer will always begin a new instruction since only bytes of complete instructions are removed from the Fetch Buffer.

The second Length Decoder will follow the same process as the first Length Decoder, except that its input total length comes from the first Length Decoder. Unlike the first Length Decoder, the index of the second Length Decoder will identify it uniquely and relate to which byte in the pool of bytes was given as its first byte. The second Length Decoder will then compare its index to the input total length passed from the first Length Decoder. If the index matches the input total length, it marks its packet as valid. If not, it marks its packet as invalid. This process continues through all Length Decoders, with each subsequent Length Decoder passing its input total length plus the length of its correctly-decoded instruction to the next Length Decoder. The final Length Decoder will compare its index to the input total length passed from the previous Length Decoder and determine whether or not it is the last Length Decoder in the sequence. If it is, it will mark its packet as valid. If not, it will pass its input total length plus the length of its correctly-decoded instruction to the next Length Decoder.

Figure 4.5 Length Decoder Interaction
Decoder, which is guaranteed to have a valid instruction, the second Length Decoder may not. If the first Length Decoder had an instruction longer than a single byte, then when the second Length Decoder compares its index (which is 1) to its input total length (which is greater than 1 in this case), there will be a mismatch. Thus, the second Length Decoder will mark its packet as invalid because of this mismatch. It will then pass through the total length given on its input to its output without modification. Figure 4.6 shows the internal structure of the Length Decoder where the decision is made whether to alter the total length or pass it through.

The total length will continue to be accumulated until it reaches the last Length Decoder. The total length outputted by the last Length Decoder will represent the total number of valid bytes used by all the Length Decoders. This value will be used in forming the “bytesUsed” value that is discussed in a following section.

Figure 4.6 Length Decoder Valid Logic
For instance, assume a Length Pre-Decode stage with a pre-decode width of four. Assume that, within the incoming bytes, there are two instructions in the first four bytes, a three-byte instruction and a single-byte instruction. Figure 4.7 shows an example of the process of selecting the Length Decoders with valid instructions. The first Length Decoder compares the incoming value of 0 to its index of 0 and has a match. Therefore, it sets its instruction as valid (green) and sends a value of 3, the length of its valid instruction plus the input value of 0, to the next Length Decoder. The next two Length Decoders with indices of 1 and 2 will not match with their given value of 3 from the first Length Decoder. Therefore, they will mark their packets as invalid (red). Finally, the last Length Decoder with an index of 3 will match and will set its instruction as valid. It will add its valid instruction length of 1 to generate a final count of 4 for the 4 bytes used for instructions.

Another factor that affects validity of an output packet is whether or not all the bytes of the instruction are available. This is handled by having each Length Decoder check if the new total length it creates is larger than the total instruction byte width of the Length Pre-Decode stage. This comparison simply involves each Length Decoder comparing its index plus its calculated instruction length to the instruction byte width. For instance, assume the third Length Decoder with an index of two calculates its instruction to have a length of eight. If the instruction byte width is less than ten, the Length Decoder's index plus its calculated length, it will fail this check since not all bytes are available. If the instruction byte width is greater than or equal to ten, the check will pass.
Once both checks have been completed, one checking the assumption of the first byte starting an instruction and the other that all bytes are present, the two outcomes are ANDed together. This result is then used to set the valid field of the outgoing instruction packet.

4.3 Fetch Buffer Feedback

The previous chapter detailed the design of the Fetch Buffer. One issue discussed was the need for feedback based on the variable number of bytes that were used by the Length Pre-Decode stage. The Length Pre-Decode stage needs to communicate to the Fetch Buffer how many bytes it has used so that the appropriate number can be removed from the buffer. Also, the number of bytes used is needed so that the Fetch Buffer will return the next group of bytes starting directly after the last byte used.

The value given to the Fetch Buffer is the “bytesUsed”. This value represents the total number of bytes used by the Length Pre-Decode stage. A byte is considered used when it is
determined to be part of an instruction that is marked as valid. Once the “bytesUsed” value is available, it is sent to the Fetch Buffer. The Fetch Buffer then uses this value to remove the correct number of bytes. This will cause the next group of bytes supplied by the Fetch Buffer to begin after the last byte used. In turn, this guarantees that the first byte of the next group is the first byte of an instruction. Previous chapters detail how the Fetch Buffer uses this value.

The “bytesUsed” value must also be used by the Length Pre-Decode stage for the next cycle. A pipeline register separates the Fetch Buffer and Length Pre-Decode stage. Therefore, there is a cycle delay while the Length Pre-Decode stage waits for new bytes. However, while waiting for the next bytes, the Length Pre-Decode stage must not reuse any of the bytes from the previous cycle. The Length Pre-Decode stage can use the “bytesUsed” value from the previous cycle to invalidate instructions that attempt to reuse bytes from the previous cycle while waiting for the new bytes to arrive from the Fetch Buffer.

4.4 Repair Last Byte PC Lookup

During the discussion of the Fetch Stage, a design was introduced wherein control prediction structures are trained based on the PC of the last byte of a control instruction, to simplify logic. Accordingly, a Length Decoder must select information affiliated with the last byte of its instruction to pass on, so that future stages can handle the packets correctly.

A Length Decoder achieves this by copying control information from the packet of the last byte of the instruction to that of the first byte. This includes the predicted next-PC, the branch direction and whether or not the instruction was a BTB hit. The copying of this
information relies on knowing the length of the instruction, making the Length Pre-Decode stage the earliest stage in which it can be completed.

Figure 4.8 shows the logic used to steer control information from the packet of the last byte of the instruction to the output packet. The last byte’s packet is selected simply using the length of the instruction.

Figure 4.8 Control Instruction Last Byte PC Correction Logic
Chapter 5: Control Pre-Decode

Control Pre-Decode is the second of the Pre-Decode stages. Its responsibility is to detect control instructions using Control Pre-Decoders. The detection of control instructions is needed so Fetch can recover if a control instruction was missed by the control prediction structures. In addition, the Control Pre-Decode stage contains the CTI Queue, which manages an in-order list of control instructions. Lastly, this stage is responsible for marking packets as invalid if they follow the wrong control path after analyzing the missed control instructions. This stage writes its packets to the Macro-Instruction Buffer.

5.1 Control Pre-Decoder

The Control Pre-Decoder was a structure present in the past implementation of FabScalar. It needed to be updated to detect x86 instructions and provide a next-PC for each new control instruction type. Although different than PISA, the x86 Control Pre-Decoder could be implemented in a structure closely following the previous implementation. The Control Pre-Decode stage and Length Pre-Decode stage have the same width and pass packets directly between themselves. The number of Control Pre-Decoders generated is equal to the pre-decode width. This allows every instruction to be checked in parallel to see if it is a valid control instruction.

The first change was to support the different opcodes used by x86 instructions. Many control instructions have common opcode bits as well as a common method for computing the taken target. For instance, all conditional jumps use the same method of calculating their taken targets. Another change was made to check the ModR/M byte. Some instructions must
have a field in the ModR/M byte checked to distinguish the type of instruction they are. To handle these control instructions, the ModR/M byte and opcode are passed to the Control Pre-Decoder to determine the instruction type and set fields appropriately.

The biggest change involves the use of the instruction length. The previous implementation added a fixed value to the PC of an instruction to determine the PC of the next instruction (next-PC). With x86, the design uses the instruction length calculated in the previous stage, Length Pre-Decode. Table 5.1 shows each type of control instruction and its length. Control instructions with a range of possible lengths are those that use a ModR/M byte for register or memory addressing.

Table 5.1 Control Instruction Lengths

<table>
<thead>
<tr>
<th>Control Instruction Type</th>
<th>Length (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call Immediate</td>
<td>5</td>
</tr>
<tr>
<td>Call Register/Memory</td>
<td>2 - 7</td>
</tr>
<tr>
<td>Return</td>
<td>1</td>
</tr>
<tr>
<td>Conditional Branch 8-bit</td>
<td>2</td>
</tr>
<tr>
<td>Conditional Branch 32-bit</td>
<td>6</td>
</tr>
<tr>
<td>Jump 8-bit</td>
<td>2</td>
</tr>
<tr>
<td>Jump 32-bit</td>
<td>5</td>
</tr>
<tr>
<td>Jump Register/Memory</td>
<td>2 - 7</td>
</tr>
</tbody>
</table>

5.2 CTI Queue

The CTI Queue maintains an in-order list of control instructions. It is responsible for generating the signals used to update the control prediction structures in the Fetch stage. A
control instruction is added during the Control Pre-Decide stage when a valid control instruction is detected.

The CTI Queue remained largely unchanged. The only change is that its entries now store the length of an instruction. The length is required so that when control prediction structures are updated by a retiring control instruction, the not-taken PC can be calculated. In addition, with the change to the BP and BTB where they are now updated with the last byte PC of an instruction, the instruction length is needed for the calculation.

5.3 Macro-Instruction Buffer

The Macro-Instruction Buffer stores macroinstruction packets and introduces a new buffer not present in the previous FabScalar implementation. The buffer has a write width equal to the pre-decode width and a read width equal to the decode width. The Macro-Instruction Buffer follows the same structure as the Instruction Buffer in the previous FabScalar implementation. The only change was a modification of the size of the packets stored. This buffer allows the pre-decode width and decode width to be set independently and therefore increases the configurability options for the core. It also enables a variable number of instructions to be written for the case where some of the instructions become invalidated and the number of packets to be written is less than the pre-decode width.
Chapter 6: Decode

The primary purpose of the Decode stage is to interpret x86 macroinstructions and generate one or more PISA-like microinstructions. Decoders are generated based on a decode width to support the parallel decoding of a decode-width number of macroinstructions. The Decode stage reads the macroinstruction packets from the Macro-Instruction Buffer and writes the microinstruction packets to the Micro-Instruction Buffer.

6.1 Operands

With fixed-length instructions in a RISC architecture that only supports register operands, determining the operands only involved extracting bits from consistent locations for each instruction to find the register encoding. With variable-length instructions and a CISC architecture, determining the operands becomes more complex. There is now more than one operand type as well as several locations within an instruction that can specify the operands.

6.1.1 Operand Types

The first information to be decoded is the operand types. The possible source operand types are immediate, register and memory location. Destination operands can only be registers or memory locations. Most instructions only use two operands, wherein one is only a source operand and the other acts as both a source and destination operand, referred to as the source/destination operand. Also, x86 only allows one of the two operands to be a memory location. The operand that may be a memory location is referred to as the complex operand since it may use complex addressing modes whereas the other operand, which may
only be a register or immediate, is referred to as the simple operand. Based on these rules and by examining relevant instruction parts, the operand types can be determined.

The simple operand is declared in the Reg field of the ModR/M byte. The three bits typically provide the logical encoding for a register. For immediate operations, the simple operand is replaced by the immediate value. Other operations that do not need to specify a second operand do not use the simple operand.

The complex operand can be a memory location or register. The Mod field of the ModR/M byte specifies if it is a register or memory location. Other values within the ModR/M field are used to specify which addressing mode is used if the operand is a memory location. The SIB byte can also be used to select other addressing modes for the complex operand. The complexities of calculating the addresses for memory operands are explained in a following section. If the complex operand is a register, then the ModR/M byte will simply specify a logical register encoding similarly to the simple operand.

6.1.2 Direction

After decoding an instruction to locate its operands, there will be two operands available, a simple and a complex. The direction for the operands determines which operand will be used as the source operand and which will be used as the source/destination operand.

For most instructions, the second-to-last bit or the second LSB of the opcode provides this information. This bit can be used as a direction bit to determine which operand acts as both the source and destination. If the bit is high, then the complex operand will act as the source/destination operand with the simple operand acting as the source operand. If the bit is low, then the opposite is true, with the simple operand being used as the source/destination
operand and the complex operand as the source operand. Figure 6.1 shows the logic used to
select the direction.

Figure 6.1 Determining Operand Direction using Direction Bit

6.1.3 Memory Addressing Modes
Before memory operands can be loaded or stored, the method to calculate the
memory address must be determined. x86 supports many complex addressing modes used to
calculate this address. Fields in the ModR/M byte and the SIB byte determine the addressing
mode if it is present. The encoding of these addressing modes with the ModR/M byte and
SIB byte are complex. This section does not go into the details of decoding each of these
cases but gives a high-level coverage of the different addressing modes and how they are handled. Table 6.1 shows each addressing mode and an example calculation for that mode.

Table 6.1 Addressing Modes

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>[0xDEADBEF]</td>
</tr>
<tr>
<td>Register</td>
<td>[EAX]</td>
</tr>
<tr>
<td>Register and Displacement</td>
<td>[EAX + 0xDEADBEF]</td>
</tr>
<tr>
<td>Register with Shift</td>
<td>[EAX*4]</td>
</tr>
<tr>
<td>Register with Shift and Displacement</td>
<td>[EAX*4 + 0xDEADBEF]</td>
</tr>
<tr>
<td>Register and Register</td>
<td>[EAX + ECX]</td>
</tr>
<tr>
<td>Register with Shift and Register</td>
<td>[EAX*8 + ECX]</td>
</tr>
<tr>
<td>Register with Shift and Register and Displacement</td>
<td>[EAX*2 + ECX + 0xDEADBEF]</td>
</tr>
</tbody>
</table>

Most of the addressing modes supported in x86 were supported in the previous PISA architecture. There are two exceptions. The first is the use of the ‘scale’ that shifts a register value before it is used. This required an additional field to be added to the microinstruction packet so that the shift can be applied in future stages during address generation. The second is the ability to use two register sources in address generation. This required changes to the load microinstructions to have a second source available. For stores, which already use two sources, one for address and one for data, supporting a second source for address calculation causes the stores to be split. This is explained in more detail in the following section on instruction splitting.
6.2 Temporary Registers

Instruction splitting causes single macroinstructions to generate one to several microinstructions. In order to execute these microinstructions, temporary values may need to be generated and passed between them. ISA-specified logical registers cannot be used to hold these temporary values because logical registers may be live across the macroinstruction. Temporary registers are introduced to allow communication between microinstructions from the same macroinstruction without disrupting the state of logical registers.

The temporary registers can be viewed collectively as an extension to the logical registers. The key difference is that they are not visible to software and they are only live within a macroinstruction, not across it. On the other hand, like logical registers, they get renamed to physical registers. Thus, the same temporary register can be used by multiple in-flight macroinstructions, since each macroinstruction will have its version of the temporary register renamed.

Three temporary registers are used. Three was determined to be the fewest number of temporary registers needed to support all splitting cases. The temporary registers are treated the same as other 32-bit logical registers. They have entries in the RMT and AMT and are renamed.

The temporary registers are assigned encodings that sequentially follow the logical registers. Table 6.2 shows the updated register encodings. This increases the total number of logical registers from eight to eleven, increasing the number of bits required to store the logical register encoding from three to four. However, as is discussed in the Rename stage chapter, renaming issues force an increase to four bits anyway.
Table 6.2 Updated Logical Register Encodings with Temporary Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>4'b0000</td>
</tr>
<tr>
<td>ECX</td>
<td>4'b0001</td>
</tr>
<tr>
<td>EDX</td>
<td>4'b0010</td>
</tr>
<tr>
<td>EBX</td>
<td>4'b0011</td>
</tr>
<tr>
<td>ESP</td>
<td>4'b0100</td>
</tr>
<tr>
<td>EBP</td>
<td>4'b0101</td>
</tr>
<tr>
<td>ESI</td>
<td>4'b0110</td>
</tr>
<tr>
<td>EDI</td>
<td>4'b0111</td>
</tr>
<tr>
<td>TMP1</td>
<td>4'b1000</td>
</tr>
<tr>
<td>TMP2</td>
<td>4'b1001</td>
</tr>
<tr>
<td>TMP3</td>
<td>4'b1010</td>
</tr>
</tbody>
</table>

Temporary Register 1, TMP1, has a special purpose. It is used to hold a value that has been loaded or will be stored. When operands are memory locations and need to be loaded or stored, TMP1 is the register where the value is expected to be located. However, macroinstructions that do not have memory operands may use TMP1 any way they need. The other temporary registers, TMP2 and TMP3, have no defined purpose and are used for many different tasks across different splitting cases for different macroinstruction types. For instance, TMP2 is used for holding the calculated address for split stores.

Temporary registers are used to communicate values between microinstructions of the same macroinstruction. An example is the call instruction. The call instruction needs to calculate an address and then push it to the stack. A temporary register is used to hold the address after it is calculated so that it can be used as store data.
A special case of using temporary registers is saving a register value before it is modified, so that the value prior to the update can be used again. One example is the PUSH instruction, which pushes data onto the top of the stack. In order to calculate the top-of-stack address, the stack pointer, ESP, must be incremented. However, if the value being pushed is ESP itself, then calculating the top-of-stack address will cause the value that is supposed to be pushed to be lost. A temporary is used to save the original value of ESP so that it can be used as the data pushed to the stack.

Finally, one other special use for temporary registers is to store immediate values. If a macroinstruction does not have a microinstruction version that supports using an immediate, then a temporary can be used to hold the immediate value. The register version of the microinstruction is then completed using the temporary register as an operand. An example is the multiply macroinstruction which has a variation that supports an immediate. There is no microinstruction that supports multiplication with an immediate, so a temporary register is used instead.

6.3 Instruction Splitting

Instruction splitting involves the translation of a single x86 macroinstruction into one or more PISA-like microinstructions. The highest number of microinstructions that a macroinstruction can be split into is five. The first aspect of splitting an instruction is acquiring and saving the operands. Instructions having only register operands do not require any extra microinstructions for this aspect. On the other hand, instructions that have a memory operand as a source and/or destination must have the operand loaded and/or stored.
The second aspect of splitting is facilitating the operation to be performed on the operands. Depending on the complexity of the operation and how well-suited the previous PISA architecture is to supporting it, it may take one or several instructions to complete. Figure 6.2 shows an overview of instruction splitting. The ModR/M, SIB, and opcode fields are used to generate microinstructions to handle operands. The opcode is used to generate microinstructions to complete the operations conveyed by the macroinstruction.

![Figure 6.2 Instruction Splitting Overview](image-url)
6.3.1 Load/Store Memory Operands

The first part of instruction splitting is retrieving and saving the operands. This may require the generation of a load and/or a store. The maximum number of microinstructions that may be generated for handling the operands is three. This occurs when the source/destination operand is a memory operand, hence, the memory operand must be both loaded (source) and stored (destination). One microinstruction is used for the load and two are used for the store. The reason the store requires two microinstructions is that the register+register addressing mode yields a store with three source registers, two for address and one for data, yet FabScalar’s design supports only two register read ports per execution lane and only two source registers in the Issue Queue’s Wakeup CAM. Whether or not an operand is loaded and/or stored does not affect the generation of the other microinstructions used in execution. The only side-effects are changes in the source and destination register fields that the other microinstructions will use.

During the determination of the operand types and direction, two flags are set. A “preLoad” flag is set when an instruction needs to have an operand loaded from memory. A “postStore” flag is set when an instruction needs an operand stored to memory after it has completed.

The opcodes used for the generated load and store microinstructions vary based on operand size, since each of the load/store opcodes is operand-size specific. The size of an operand, 8, 16 or 32 bits, is set in the Length Pre-Decode stage. The implementation selects a different opcode, for loads and stores, based on the operand size: load/store byte (8),
load/store halfword (16), and load/store word (32), respectively. No modifications are needed in address generation or other parts of the microinstruction for different operand sizes.

If a load is needed for a source operand, it will always be the first microinstruction generated for the macroinstruction. This is required since all future microinstructions will rely on the source value to be loaded. The load microinstruction is created if the “preLoad” flag is set. The generated load simply calculates its address using relevant registers, displacement and scale value, and loads from that location into register TMP1.

If a store is needed for a destination operand, it will always be the last two microinstructions generated for the macroinstruction. Alternatively, the store could have been implemented with only one microinstruction, if the microarchitecture were modified to support three source registers per microinstruction. Unfortunately, this would be very complex to implement and require large-scale changes: three register read ports in the load/store execution lane (instead of two) and three source registers in the Issue Queue’s Wakeup CAM (instead of two). Moreover, stores would be the only instruction type that would currently be using the three source registers, leaving these large-scale changes mostly unused.

Consequently, the implemented solution for supporting two registers for address calculation for stores is to split the store into two microinstructions. In this solution, an extra microinstruction is generated immediately before the store. It has the two source registers along with any displacement and/or scale value. It computes the memory address and saves it in TMP2. TMP1 is already used to hold the data to be stored. After the address calculation,
the store microinstruction is generated wherein it uses TMP1 for the data and TMP2 for the address.

The negative consequence of the split store is that it requires an additional microinstruction for every instance where an operand needs to be stored. This means that an instruction could have up to three microinstructions generated, one for loading an operand and two for storing, without including the microinstructions that execute the operation. However, instruction splitting is already fairly complex and able to support the splitting into a large number of microinstructions. Therefore, the increase in complexity was not drastic.

6.3.2 Execute

The more complex part of instruction splitting is selecting the microinstructions that will perform the operation on the operands. Instructions that do not follow the common format will have their operands acquired as part of their execution. There are two primary classes of macroinstructions that are split, simple and complex. The simple vs. complex distinction is based on the complexity of the logic needed for generating the microinstructions. The distinction is in the complexity required to generate the microinstructions rather than the number of microinstructions generated.

The maximum number of microinstructions generated for execution is five. Macroinstructions that generated loads and stores for operands, which may be up to three required microinstructions, are limited to two microinstructions to execute their operations because of this limit.
6.3.2.1 Simple

Two types of instructions are considered as simple. The first is the common format instructions. Their simplicity comes from their use of the common format for declaring operands. Also, their operations are simple operations that do not require many microinstructions to complete. The next simple type of instruction is branch instructions. The simplicity of branch instructions stems from their lack of required operands and their high uniformity among the different types of branch instructions. The last type of simple instruction is flag-setting operations. These operations are used to set and clear flags.

The first type of simple instruction is the type that follows the common format. These instructions have their sources and destination specified in a ModR/M byte and/or SIB byte. They also rely on generated loads and stores to handle operands that follow a common format among many different instruction operations. Since their operands are predetermined, the logic needed to generate their microinstructions can be simpler.

Most of the instructions that follow the general format are simpler arithmetic operations. This includes operations such as addition and subtraction along with several bitwise operations like XOR and NOT. These operations only require a single microinstruction to execute and were already directly supported by PISA. Therefore, generating the microinstruction only required reusing the PISA operation with accommodations for the updated operands and new fields, such as operand size. The splitting of these instructions is very similar to how they would have appeared in the PISA implementation. Figure 6.3 shows an example of splitting a simple instruction, an addition
that uses a memory operand. The AGEN microinstruction is used to calculate the store address as part of the split store.

Figure 6.3 Split Simple Addition Instruction

The next type of simple instruction is conditional branch instructions using an offset. Most branch instructions follow the exact same format when generating a microinstruction. The only difference among them is which opcode is used. The opcode of the branch is used in the Execute stage to identify the type of branch and the condition that needs to be tested for it. Since new microinstructions were added to support the new branches in x86, the
opcodes of the microinstructions were mapped to those of the x86 macroinstruction. Therefore, the logic for generating the microinstruction opcode for a branch instruction involves extracting bits from the x86 opcode and using those to set the microinstruction opcode. This makes the generation of all the different branches simple and requires very little additional logic.

The last type of simple instruction is flag-setting instructions. These instructions are used to set, clear or complement individual flags within the flag register. Although each of the instructions has a unique opcode and requires a different microinstruction to complete, they have a uniform format. Also, since the flag-setting instructions do not have any general-purpose logical register sources or destinations, there is no logic needed to determine their operands.

6.3.2.2 Complex

Complex instructions are instructions for which splitting is complex and usually unique to each instruction. A complex instruction does not follow the common format: it does not use ModR/M to specify operands and/or does not have a size bit in the opcode to allow for easy operand size determination. These instructions require a more complex combination of microinstructions to be generated to obtain their operands and complete the operation. The complexity of the instruction is usually caused by operand sizing issues, incompatibility with the previous PISA implementation, and the macroinstruction itself conveying a complex multi-step operation.

Move instructions in x86 offer one example of the complexity caused by operand sizing. These instructions allow the movement of data to and from memory locations and
registers. The movement of data between registers, memory and immediates of the same size is simple. The complexity is introduced when the sizes of the source and destination differ. For instance, the move with sign-extend supports sign-extending an 8-bit value to create a 32-bit value to be stored in a register or memory location. For this case, microinstructions need to be generated that are capable of handling an 8-bit source and a 32-bit destination.

Multiply and divide instructions offer an example of complexity introduced by incompatibility with the PISA implementation. The PISA implementation splits a multiply (or divide) instruction into two microinstructions, one to write the high-half of the product (or quotient) and the other to write the low-half of the product (or remainder). Although this splitting approach was repeated for x86, there were some compatibility issues. The first issue was operand sizing. x86 supports multiply and divide operations with 8-bit, 16-bit or 32-bit operands whereas PISA only supports 32-bit operands. In addition, x86 requires the use of the flags register and setting the flags register when two operations are used to complete the operation creates complications. Although, the x86 multiply and divide instructions could be supported using the existing PISA microinstructions, some complexity was created because of incompatibility that would not have been necessary if the back-end had been designed solely with x86 in mind.

A complex x86 instruction usually requires several microinstructions. Also, this set of microinstructions is unique so that it cannot be used by other instructions. One example is the call instruction. The call instruction pushes a return address onto the stack and jumps to the address of the called subroutine. Several operations need to be performed to complete this instruction. The first step is to calculate the target address of the call. If the target address is a
memory operand, then it must be loaded into a temporary register. The second step is to update the stack pointer to prepare for the stack-push operation. This requires an operation to subtract four from the stack pointer, ESP, corresponding to the four bytes of the return address. Lastly, the return address is pushed onto the stack. This requires a store microinstruction to store the return address to the calculated top-of-stack address in ESP. Figure 6.4 shows an example of the microinstructions that a call could be split into.

Figure 6.4 Split Complex Call Instruction

The call instruction is one of the most complex instructions. A few other instructions exist with similar complexity such as the push, pop and return instructions. Each of these has
a unique set of microinstructions that must be generated to complete each operation required by the macroinstruction.

6.4 Micro-Instruction Buffer

The Micro-Instruction Buffer accepts the microinstruction packets after splitting. The Micro-Instruction Buffer supports the need to write a variable number of valid packets depending on how the macroinstruction is split. The Micro-Instruction Buffer follows the same structure as the Macro-Instruction Buffer. The write width must be equal to the maximum possible number of split microinstructions. With the maximum split size per macroinstruction set at 5, this sets the write width equal to decode width multiplied by 5. This is much larger than the original Instruction Buffer from PISA, which had a maximum split size of 2 that this buffer is replacing. The read width is equal to the rename width. The presence of the Micro-Instruction Buffer allows the Decode and Rename stages to have independent widths.
Chapter 7: Rename

The Rename stage is responsible for register renaming to support out-of-order execution. Register renaming is able to overcome false dependencies between instructions caused by a lack of logical registers. A large number of physical registers are used and assigned to remove the false dependencies. Register renaming relies on a map table, free list, and active list to manage physical register addresses from a physical register file. The Rename stage reads rename-width instructions from the Micro-Instruction Buffer.

7.1 Number of Logical Registers

The x86 ISA has only 8 logical registers (for integer registers), compared to 34 used by the PISA ISA. The configurability of FabScalar allows the number of logical registers to be defined in the configuration file. The fewer number of logical registers reduces the number of entries in the RMT. The Free List changes by making it larger since there are now more free physical registers, for the same physical register file size. The lower number of logical registers provides a higher number of physical registers that are available for renaming.

7.2 Zero Register Workaround

The zero register in PISA is a register that acts as a sink and a source that will always have a value of zero. This register is often used for instructions that do not have valid source or destination registers. The PISA implementation leveraged the properties of this register for logic optimizations such as marking the register as always ready in the Issue Queue. The zero register has an encoding of zero in PISA. Unlike PISA, x86 uses the zero encoding to address
a general-purpose register, EAX. Therefore, changes needed to be made to support references to the EAX register and prevent the zero register optimizations from pertaining to it. The zero register is still used by many generated microinstructions in the x86 implementation. The changes to the Rename stage are needed to avoid the zero register being used when a general-purpose register is conveyed.

The first option was to eliminate the special treatment of the zero register in other stages and allow it to be treated as all other logical registers. This would have caused a lot of significant changes to other stages. For instance, the Issue stage has special checking that assumes that the zero register is a source that is always ready. The simpler solution was to modify the register encodings so that the zero register encoding is never used as a general-purpose register.

There were several possible options for modifying the encodings. For instance, all register encodings could have been left-shifted by one bit and a one placed in the LSB. This would guarantee that the zero register was never used as a valid source or destination and would be simple logic to implement. However, this would lead to several logical register encodings, all with zero in the LSB, being unused causing space to be wasted in the RMT.

The implemented solution was to add one to each register encoding. If it is valid, the logical register encoding is set to the current logical register encoding plus one. The zero register is guaranteed to not be used as a valid register since it will always be incremented to at least one. This improves upon other solutions in that it only creates one additional register encoding and does not waste any resources. The compromise is that adders must be used to increment every single logical register encoding for all instructions to be renamed. This
workaround increases the number of logical registers by one to account for the zero register which causes the number of bits used for the logical register encoding to increase to four. Table 7.1 shows each logical register encoding after it has this method applied to it.

<table>
<thead>
<tr>
<th>Logical Register</th>
<th>Original Encoding</th>
<th>Updated Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>3'b000</td>
<td>4'b0001</td>
</tr>
<tr>
<td>ECX</td>
<td>3'b001</td>
<td>4'b0010</td>
</tr>
<tr>
<td>EDX</td>
<td>3'b010</td>
<td>4'b0011</td>
</tr>
<tr>
<td>EBX</td>
<td>3'b011</td>
<td>4'b0100</td>
</tr>
<tr>
<td>ESP</td>
<td>3'b100</td>
<td>4'b0101</td>
</tr>
<tr>
<td>EBP</td>
<td>3'b101</td>
<td>4'b0110</td>
</tr>
<tr>
<td>ESI</td>
<td>3'b110</td>
<td>4'b0111</td>
</tr>
<tr>
<td>EDI</td>
<td>3'b111</td>
<td>4'b1000</td>
</tr>
</tbody>
</table>

7.3 Register Sizing

In PISA, all operations operate on 32-bit register values. x86 introduces variable sizing where registers can be addressed as 32-bit, 16-bit or 8-bit values. The reading and writing of partial values is handled in future pipeline stages, however, some changes are needed in the Rename stage to assure the correct logical registers are used.

32-bit and 16-bit register encodings are not affected because they address the same logical register. For instance, the 32-bit register EAX is addressed with 3’b000 and the 16-bit register AX, the lower 16 bits of EAX, is addressed with 3’b000 as well. These addresses will read and write the correct register without modification to the logical register address.
Registers with 8-bit encodings require special handling. 8-bit values have encodings that conflict with the 32-bit encodings. For instance, in 32-bit sizing, an encoding of 3’b100 would correspond to ESP. In 8-bit sizing, this encoding would correspond to the upper 8-bits of AX or bits 15-8 of EAX, referred to as AH. This requires special handling of encodings representing 8-bit values so that they can be steered to the correct logical register. The incoming packets have a size field that signals the size, 8, 16 or 32 bits, that the instruction uses.

The implemented solution operates on a property of 8-bit encodings. 8-bit encodings are only able to address parts of four logical 32-bit registers, EAX, EBX, ECX, and EDX. The lower bit of the three address bits is used to identify the register as a “high” or “low” register, leaving only the upper two bits to specify the logical register. The corresponding 32-bit logical register can be inferred by taking the upper two bits of the register address and setting the MSB, the third bit, to zero. For instance, AH has an encoding of 3’b001. By performing the transformation, it would result in an encoding of 3’b000, which corresponds to EAX, the 32-bit register AH is a part of.

The Rename stage implements this method by first checking for the size of an instruction. If the size is 8 bits, then the transformation is applied to the register address. The more complex job of using the correct 8 bits is handled by the Execute stage that is simply passed the entire 32-bit register value. The lower address bit, used to specify “high” or “low” register, is passed to future pipeline stages to identify which bits of the register to use. Table 7.2 shows the updated encodings after this transformation has been applied. After this
transformation, the updated encoding has the zero register workaround applied to it as well by adding one to each encoding (not shown).

<table>
<thead>
<tr>
<th>Logical 8-bit Register</th>
<th>Original Encoding</th>
<th>Updated Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>3'b000</td>
<td>3'b000</td>
</tr>
<tr>
<td>AH</td>
<td>3'b001</td>
<td>3'b000</td>
</tr>
<tr>
<td>CL</td>
<td>3'b010</td>
<td>3'b011</td>
</tr>
<tr>
<td>CH</td>
<td>3'b011</td>
<td>3'b001</td>
</tr>
<tr>
<td>DL</td>
<td>3'b100</td>
<td>3'b010</td>
</tr>
<tr>
<td>DH</td>
<td>3'b101</td>
<td>3'b010</td>
</tr>
<tr>
<td>BL</td>
<td>3'b110</td>
<td>3'b011</td>
</tr>
<tr>
<td>BH</td>
<td>3'b111</td>
<td>3'b011</td>
</tr>
</tbody>
</table>

7.4 Flag Register Renaming

With the introduction of the flag register there is an additional register that can act as a source and/or destination register for instructions. This additional register needs to be renamed similarly to the other registers. Two possible options were explored.

The first option is to absorb the flag register into the existing renaming structures, as shown in Figure 7.1. The flag register would be assigned a logical register encoding not yet assigned to a logical register. The flag and general-purpose registers would share a Physical Register File, RMT, AMT and Free List. This would require the sizes of the RMT, AMT and Free List to be adjusted accordingly: the RMT and AMT would both have one more entry for the flag register, and the Free List would have one fewer entry if the Physical Register File size is not adjusted.
The drawback of Option 1 is that it significantly increases the complexity of the renaming structures. The RMT must now be able to rename three source and two destination registers per instruction. The Free List must be able to allocate and free up to two destination registers per instruction. This would create a large increase in the complexity of these structures and increase timing pressure. The other issue is that the general-purpose and flag registers would share a Physical Register File. Consequently, the Physical Register File size must be increased to accommodate the same number of in-flight instructions. Also, taking into account future pipeline stages, the Physical Register File needs one additional read port and one additional write port per execution lane to support the renamed flag source and destination registers. This would involve another large increase in complexity for the Physical Register File structure. Figure 7.1 shows an overview of the major changes to the rename structures for Option 1 with changes highlighted in red.

Option 2, which was implemented, is to duplicate the renaming structures as shown in Figure 7.2. The general-purpose and flag registers are managed independently, similarly to integer and floating-point registers in other architectures. The key advantage of this approach is not increasing the complexity of the existing renaming structures used for the general-purpose registers: the number of read and write ports to all of the structures, and the size of the Physical Register File, are unchanged.

A simplified Flag RMT was created for the flag register. It has just two entries, one for the logical flag register and one for the zero register workaround. Moreover, the Flag RMT has only one read port (flag register source) and one write port (flag register
destination) per instruction, as opposed to two and one for the RMT. Also, since there is only one logical flag register, there is no need to save it for every instruction.

The Free List was duplicated as well. The resulting Flag Free List is sized based on the Flag RMT and Flag Physical Register File. Each free list signals when it is empty and cannot allocate a new physical register for incoming instructions. The two empty signals are simply ORed together to create a unified signal. However, the free lists must also be aware if the other is empty and feed each other these signals so each free list knows whether it must wait for the other free list to have entries before it completes its allocation.
The Physical Register File was duplicated to create the Flag Physical Register File, which can be sized independently. The Flag Physical Register File was simplified to support reading only one source register per instruction instead of two. Other pipeline stages that interact with the physical register files were modified as well, including the Register Read and Writeback stages; their changes are discussed in future chapters.

Lastly, to support recovery of the flag register and freeing of flag physical registers, a Flag AMT was created. Its structure mirrors that of the Flag RMT. All of the interfaces between the general-purpose AMT and other pipeline stages, for facilitating recovery, freeing of physical registers, and so forth, were duplicated for the Flag AMT and signal widths adjusted as needed.

This solution allows the flag and general-purpose registers to be renamed, read and written in parallel, in a manner that is complexity-effective. It also provides the added configurability of sizing the Flag Physical Register File independently, so that its size can be set based on usage patterns. Figure 7.2 shows an overview of the major changes to the rename structures for Option 2, with changes highlighted in red.
Figure 7.2 Changes to Rename Structures for Option 2
Chapter 8: Dispatch and Issue

The Dispatch stage is the last stage of in-order processing of instructions. The Dispatch stage is responsible for sending instructions to the Load/Store Queues, Active List and Issue Queue. It is also responsible for managing the status of these queues. The Dispatch stage will stall the front-end when any of its managed queues cannot accept the full dispatch bundle.

The Issue stage is responsible for issuing instructions when they are ready to be executed. This stage contains the Issue Queue, which holds instruction packets until they are issued. Wakeup logic within the Issue Queue keeps track of the dependencies for an instruction and marks it as ready when all its values are available. Select logic determines which ready instructions to issue and to which Execution Lane. The Issue stage also supports issuing dependent instructions consecutively so that the bypasses can be used to allow faster execution.

8.1 Dispatch Packet Generation

The Dispatch stage was left almost completely unchanged from the previous implementation. The only change involved updating the generation of packets. The new implementation added and modified fields of the instruction packet. Therefore, the logic had to be updated to copy the correct fields from the input packet to the output packet. The same change was made to the packets sent to update the Active List. The x86 implementation used more fields in the Active List packet and the Dispatch stage was updated to correctly set these packets.
8.2 Flag Ready Vector

The flag register introduces a new source operand that must be considered when checking the readiness of an instruction. Since the flag register uses a dedicated physical register file, the preexisting logic for maintaining a ready vector was duplicated. It needed to be modified to account for the independent configurability of the Flag Physical Register File.

The wakeup logic was expanded to take into account the readiness of the flag register source before considering an instruction as ready. The Issue Queue wakeup CAM was augmented with an additional source physical register tag per instruction, corresponding to the flag source. The Issue Queue payload RAM was augmented with an additional destination physical register tag per instruction, corresponding to the flag destination. The Issue Queue’s wakeup ports, including the shift registers which properly time the wakeup of dependent source operands, were augmented to broadcast destination physical register tags of the flag register in addition to those of the general-purpose registers.
Chapter 9: Execution Lanes

The Execution Lanes handle the execution of instructions using provided data sources to generate a result. Each lane contains a Register-Read stage, Execute stage and Writeback stage. The four types of lanes are Simple ALU, Complex ALU, Control and Memory. Instructions are assigned to a specific lane based on their type. At least one of each type of lane must be present so all instruction types can be executed. Bypasses are used within and across lanes to allow producer and consumer instructions to execute in consecutive cycles, instead of waiting for the values to be written into and read from the Physical Register Files.

9.1 Register-Read, Writeback and Bypasses

The use of the flags register in x86 created a need for a second physical register file to support its use. The Register-Read and Writeback stages needed to be modified to support this new register file. Figure 9.1 shows the modified Execution Lane with the added bypasses and register file read/write accesses for the flag register highlighted in red.

The Register-Read stage was augmented to read from the Flag Physical Register. The instruction packets it receives contain the physical addresses for the general-purpose sources as well as the physical address for the flag source. The Register-Read stage passes the physical address for the flag source to the Flag Physical Register File. The returned flag value is added to the instruction packet and used in execution.
The Writeback stage was modified to write to the Flag Physical Register File. If the instruction packet has a valid flag destination, then it will provide a flag destination physical register address as well as the flag result data. This address and data will be sent to the Flag Physical Register File.

Bypasses were also added to support the direct transfer of flag results to dependents. The bypasses created for flag registers follow the same structure as the bypasses for general-purpose registers. The only change is in the size of the tags used to identify physical registers. Since the Flag Physical Register File can have a different number of registers compared to the Physical Register File, they may require a different number of bits to identify registers.
9.2 Simple and Complex

The Simple Execute stage features a Simple ALU and the Complex Execute stage features a Complex ALU. Together, these ALUs are responsible for most arithmetic and bitwise logical operations. The Simple and Complex ALU share a very similar structure. This allows modification strategies to be shared between them.

9.2.1 Partial Register Read and Write

x86 supports operations on operand sizes of 8, 16 and 32 bits. Yet, the inputs and output of the ALU are 32 bits. Therefore, the internals within the ALU are responsible for extracting the correct operand, completing the operation, and then returning the correct 32-bit value.

Partial register reads involve extracting the correct bytes from the provided 32-bit value. This extracted operand will then be used in the specified operation. The operand size along with the “isUpper” flag is used to determine which bytes to extract. Figure 9.2 shows an example of extracting an operand from the provided 32-bit source data. The operand size selects the number of bits that should be extracted from the register data. The “isUpper” flag, which was set in the Rename stage, is used for 8-bit sizing to signal whether it is the upper 8 bits, bits 15-8 or byte 1, or the lower 8 bits, bits 7-0 or byte 0. The extracted 8 or 16 bit value is sign-extended to a 32-bit value. This value is then used in the specified operation. This logic is applied to both provided sources.
A partial register write involves only modifying a subset of the logical destination register. The remaining “background” bytes must be left as-is. With renaming, however, the logical destination register is mapped to a “blank” physical destination register that does not have the background bytes. The background bytes are only in the physical register to which the logical destination register was previously mapped. Theoretically, obtaining the background bytes would require that the microinstruction have an additional logical source register that is the same as the logical destination register. Fortunately, a property of x86 achieves this for free: the logical destination register is also one of the logical source registers already. Therefore, since the logical destination register is the same as a logical source
register, the background bytes needed for the partial write to the logical destination register are available from that logical source register.

Figure 9.3 shows an example of completing a partial write with an operand size of 16 bits. Similarly to the partial register read logic, the operand size is used to extract the correct bytes from the 32-bit result generated by the operation. These bytes serve as the modified portion of the destination register. Also based on operand size, the unmodified portion of the destination register is obtained from the source register of the same logical name. This allows a correct partial write to be completed without the need for an additional logical source register or large-scale microarchitectural changes.

Figure 9.3 Partial Register Write Example with 16-bit Sizing

9.2.2 Handling Flags

Many x86 instructions read and/or write the flag register. With the Register-Read and Writeback stages augmented to support the flag register, the Simple and Complex ALUs were modified to accept a flag value and generate a flag value.
Most instructions executed in the Simple and Complex ALUs set flags based on their calculated result. Logic was added for each flag-setting instruction to set the flag register based on its result. Different operations have different criteria for setting flags. For instance, an addition instruction sets the overflow flag based on checking for the standard overflow condition using the addend and sum values. However, an OR operation defaults the overflow flag to zero since an overflow condition is not defined for a bitwise OR operation. The implementation for most flag conditions simply involves implementing the logic defined by the x86 ISA for setting flags. Most are simple and do not require much complex logic.

When setting flags, the operand size must also be used to determine how to set the flag. For example, Figure 9.4 shows how setting the carry flag for an addition operation will vary based on operand size. Variable operand size complicates the logic needed to set flags. However, the logic simply involves selecting which bits to use in checking the flag conditions. For instance, for setting the zero flag for an addition operation, the lower 8 bits will be checked for 8-bit sizing, the lower 16 bits for 16-bit sizing, and the entire 32-bit result for 32-bit sizing.
Another way to manipulate the flags register is through explicit read-flags and write-flags instructions. A read-flags instruction copies the flags register to a general-purpose register. A write-flags instruction copies a general-purpose register to the flags register. Since the flags register is already available as a source and/or destination register for arbitrary instructions, implementing these read and write instructions was straightforward.

The x86 ISA also provides instructions that set, clear, or complement specific flags in the flags register. These instructions were added to the Simple ALU. A key property of these instructions is that they do not modify the other flags. Therefore, these instructions must read the current flags, update just the specified flag, and write the adjusted flags back. Thus, as with partial register writes of general-purpose registers, these instructions have the flags register as both a logical source register and a logical destination register. These instructions do not have any other source or destination registers besides the flags register.
9.2.3 Carry Operations

Two supported instructions in x86, add-with-carry and subtract-with-borrow, use the carry flag to complete the operation. These instructions add the carry flag to one of the source operands before performing the overall addition or subtraction.

A new field, “carryValid”, was added to the instruction packet. During decode, the instructions that use the carry flag, add-with-carry and subtract-with-borrow, set the “carryValid” flag. All other instructions clear the “carryValid” flag. The ALU that executes the instruction will recognize the “carryValid” flag and add the carry flag to the appropriate operand if necessary. This allows existing microinstructions to be used without the need for new microinstructions specifically for carry operations.

Figure 9.5 shows the logic for selecting the operands using the “carryValid” flag. A high “carryValid” flag signals that the instruction will use the carry flag. Therefore, the carry flag, regardless of its value, is added to the source operand. This source operand is then used with the other source operand to complete the required operation. If the “carryValid” flag is low, then operation continues on like a normal operation with no modification to the source operand.

9.3 Control

The Control Execute stage handles instructions capable of modifying the PC. The main part of executing a control instruction is calculating its next-PC which is used to set the PC of the instruction after the control instruction. The primary types of supported control instructions are branches, jumps and interrupts.
9.3.1 Next-PC Calculation

The PC of the instruction following the branch, termed “next-PC” in this section, is either the not-taken target or the taken target of the branch, depending on whether the branch is conditional or unconditional and, if the former, whether the conditional branch is not-taken or taken. This section describes how both the not-taken and taken targets are calculated.

The not-taken target is the PC of the branch plus the branch’s length. The length was calculated in the Length Pre-Decode stage. Both the PC and length are part of the instruction.
packet that reaches the Control Execute stage. This method of calculating the not-taken target is valid for all control instruction types.

The taken target for relative branches is calculated by adding together the PC of the branch, the branch’s length (as before), and the branch’s offset value. Absolute branches simply copy the target address to the PC.

9.3.2 Conditional Branches

Conditional branches in x86 are based on testing values in the flag register. There were no existing instructions that supported this since PISA did not use a flag register. First, the execution unit needed to be modified to accept the flag source register. Second, in order to use this flag source register, a new conditional branch instruction had to be created for each conditional branch instruction supported in the reduced x86 ISA.

The next-PC logic was consistent for all conditional branches. However, unique logic was added to calculate the direction for each type of conditional branch. A separate microinstruction was created for each conditional case. The determination of a direction is based on checking specified flags within the flag source register. Some of the conditions are more complex than others to compute. However, they still only operate on single-bit values simplifying the underlying logic. An example condition is the JE, “jump if equal”. Here, the zero flag is checked to be high. The logic for this condition is simply to set the direction based on the value of the zero flag. A more complex example is the JLE, “jump if less or equal”. In this case, the zero flag is checked to be high and the sign flag and overflow flag are checked to be not equal (ZF || (SF != OF)).
9.3.3 Large Offset Jumps

Both PISA and x86 support conditional and unconditional jumps. However, x86 supports unconditional jumps with 8-bit and 32-bit immediate values used for offsets from the PC. The PISA implementation only supports unconditional jumps with an absolute address of 26 bits without a method for implementing jumps with offsets. To handle the unconditional jumps in x86 with larger offsets, a new instruction was added to the control unit. This instruction uses an immediate value of up to 32 bits. This immediate value is used as the offset in the calculation of the taken target, as discussed in a previous section.

Similarly, PISA conditional branches only support a 16-bit offset whereas x86 conditional jumps support 8-bit and 32-bit offsets. Accordingly, the new conditional jump instructions, which were added to support each condition, were implemented with the larger offset.

9.3.4 Interrupts

x86 uses the ‘INT’ instruction to trigger an interrupt. This instruction is accompanied by an immediate value that signals the interrupt type it is referring to. The ‘INT’ instruction, however, is still fundamentally a control instruction. A new instruction was added in the control unit to handle the ‘INT’ instruction and calculate the next-PC.

Currently, the functional simulator handles the interrupts in the C++/Verilog co-simulation environment. Therefore, in the Verilog description, the interrupt merely sets its next-PC to the PC of the instruction following the interrupt. Adding the PC of the control instruction to its length sets the next-PC. The exception flag is also set for the instruction in
the Active List so that it can be recognized as an exception at retirement and initiate the interrupt handling.

9.4 Memory

The Memory Execute stage contains an address generation unit. This unit must compute an address and interact with the LSQ to complete memory operations. For the updated implementation, the execution of the memory instructions within the LSQ was unchanged. However, changes were made to the address generation unit to support the more complex addressing modes of x86.

9.4.1 Address Generation

x86 introduces new complex addressing modes. It allows memory addresses to be calculated using two source registers, one of which may be scaled, and a displacement of up to 32 bits. In contrast, the PISA implementation supports only a single source register and 16-bit displacement. The address generation unit was modified to accept two source registers, a shift amount (for scaling), and a 32-bit immediate value.

The address calculation was modified as follows. A shift is applied to the scaled source register value, and then this value is added to the other source register value and the immediate. Each load instruction, of the various sizes (byte, halfword, word), was modified for the new address calculation.

The same complex calculation is always performed even if the instruction’s addressing mode is simpler. This relies on the Decode stage setting unused fields to zero so that they do not affect the address calculation. For example, if the addressing mode did not
specify a shift, then setting the shift amount to zero will prevent a shift occurring. Also, if only one of the source registers is to be used, then the unused source register would be set to the zero register so that the register value is zero and have no effect on the calculation. Using this uniform method simplified the address calculation logic and did not require several different calculations based on different addressing modes.

Stores do not implement the same complex calculation. The complex address calculation, wherein two source registers may be used, would have forced the need for three source registers since stores need a source register for the data to be stored. As discussed previously, this is handled by splitting the store.

The store that enters the address generation unit has two sources, one for the address and one for the data to be stored, as well as an immediate value. Only the address and the immediate value are used in the address calculation, when they are added together. The expectation is that if a complex addressing mode is required, the previous microinstruction would have completed the calculation and the result is now available in the address of the store. This format for stores makes them follow the same implementation as PISA, with the only change being that the immediate value used in the calculation is increased to 32 bits from 16 bits.
Chapter 10: Retire and Recovery

The Retire stage is responsible for committing instructions in program order. The Active List stores all in-flight instructions in program order. The AMT is used to maintain architectural state. When an instruction retires, it updates the AMT with its logical-to-physical register mapping (if it has a destination register), thereby committing its physical register and freeing the previously committed physical register to be used by other instructions. Instruction information is also sent to the LSU for the committing of memory operations. The Retire stage is also responsible for initiating recovery. The three types of recovery situations are branch mispredictions, load violations and exceptions. Recovery involves rolling-back the pipeline to the precise architectural state, including (1) flushing the pipeline of all in-flight instructions, (2) rolling-back the RMT to the AMT and restoring the free list, and (3) redirecting the fetch unit’s PC to the correct path (for branch misprediction case), the load instruction to be replayed (for load violation case), or the handler (for exceptions).

10.1 Flag Destination Register

The structure of the Retire stage remained largely unchanged. However, some simple changes were needed to support the flag register. This involved some changes to the Active List as well as to the logic of the Retire stage.

The first change was to the Active List entry. Since the flag register may be another destination for an instruction, the Active List entry must hold the physical register mapping for the flag destination register. One advantage of the flag destination register over the
general-purpose destination register is that, since there is only a single logical flag register, there is no need to save the logical address for the flag destination register.

The Active List records the logical-to-physical register mapping for each instruction in order to commit these mappings to the AMT in program order. Thus, the AMT keeps track of architectural state in case recovery is needed at some point. However, since the flag register is renamed to a dedicated physical register file, it also needs a dedicated AMT. Therefore, the Flag AMT was created. The Flag AMT has the same structure as the general-purpose AMT, except it has fewer entries. In fact, like the Flag RMT, the Flag AMT has just two entries, for the logical flag register itself and a zero register.

Just prior to committing a mapping to the AMT, the physical register of the previous mapping is returned to the free list. This will allow the freed register to be used again in renaming another logical destination register. Support for freeing flag physical registers was added by sending flag physical registers, when replaced in the Flag AMT, to the Flag Free List. The design for freeing flag physical registers mirrors that of freeing general-purpose physical registers.

10.2 Recovery

A common recovery step for branch mispredictions and load violations is copying the mappings from the AMT to the RMT. When a recovery situation is encountered, execution is stalled for multiple cycles while the AMT is copied to the RMT in parts. Changes were made to this method to support the new number of general-purpose logical registers as well as to support flag register recovery.
As discussed in previous chapters, x86 has fewer logical registers than PISA, which reduced the RMT size. As a consequence, the size of the AMT was also reduced to have the same number of entries. An advantage of the fewer number of logical registers is quicker recovery. Since there are fewer mappings to be copied, it will take fewer cycles to copy all of them. This helps reduce the overall recovery time.

In addition to the RMT, the Flag RMT also needs to be recovered. Accordingly, signals were created between the Flag AMT and the Rename stage to allow the transfer of mappings. Only a single cycle is needed for this copying since only two mappings are present in the Flag AMT. The Flag AMT-to-RMT copying proceeds in parallel with the general-purpose AMT-to-RMT copying. Since the latter takes more cycles, it dictates when execution may resume.

10.3 Ordering of Instructions

At present, the x86 implementation does not explicitly retire macroinstructions atomically. The Active List does commits at the microinstruction level. This may lead to problems if a recovery scenario – a branch misprediction or load violation – arises in the middle of a macroinstruction. Fortunately, the order of microinstructions within their macroinstructions always works out such that the offending microinstruction is either the first (for loads) or last (for branches) microinstruction in the sequence.

Loads tend to be the first microinstruction within their macroinstruction, since they load data that will be needed in the following microinstructions. This works out nicely in terms of recovering from a load violation. In the case of a load violation, execution needs to
restart at the offending load, to replay it. Since the offending load is the first microinstruction, neither it nor any other microinstructions in the macroinstruction will retire, which is precisely the desired behavior since the macroinstruction needs to re-execute as a whole.

On the other hand, branches tend to be the last microinstruction within their macroinstruction, since they modify control-flow and there can be no microinstructions following them. This also works out nicely in terms of recovering from a branch misprediction. In the case of a branch misprediction, execution needs to restart at the correct target of the branch. Since the branch is the last microinstruction, it and all prior microinstructions in the macroinstruction have retired, which is precisely the desired behavior.

In both cases of branch misprediction recovery and load violation recovery, the effect of atomic retirement of macroinstructions is achieved. Even the case where a macroinstruction generates both a load and a branch will be handled correctly with this method, whether the offending instruction is the load, the branch, or both. One issue, left for future work, is the possibility of exceptions anywhere within a macroinstruction. Precise exception recovery may require a more explicit strategy for atomic commits.
Chapter 11: Results

This chapter introduces the simulation environment that was used for simulating and verifying the Verilog implementation. Also presented, are results from running a variety of micro-benchmarks on several microarchitecture configurations, to stress varied types of instructions, diagnose superscalar performance and scalability, and demonstrate preservation of FabScalar configurability.

11.1 Simulation Setup

The Cadence® NC-Sim simulation environment is used. A functional simulator running in parallel with the Verilog simulation is used for checking. The functional simulator is a C++ model that simulates the reduced x86 ISA used in this implementation. The Verilog simulator simulates its own architectural memory state using a C++ model. Figure 11.1 shows a visual of the interaction between each part of the simulation environment.

The memory model used by the Verilog simulator is initialized to be empty. When a value is attempted to be read by the Verilog simulation that is not present, then the value is retrieved from the functional simulator and placed in the memory model. All other memory operations by the Verilog simulation complete by accessing its own memory model without any interaction with the functional simulator. This allows the Verilog simulation to have its own memory model, which allows errors to be detected when memory values are read and written. However, it also allows the Verilog simulation to access the functional simulator to “bootstrap” its memory model. Such gradual bootstrapping occurs after the functional simulator loads the benchmark binary image into its own memory model and after the
functional simulator handles a system call (e.g., for file I/O). These system-level tasks are only handled in the functional simulator.

Checking occurs when an instruction retires from the Active List. The first check is for the PC. The PC from the Verilog simulation is compared to that of the functional simulator. The functional simulator steps through x86 instructions. The x86 instructions are macroinstructions in the implementation, but the Active List retires microinstructions. Therefore, a PC check is only completed once per group of microinstructions that correspond to a single macroinstruction.

Another check that occurs at retirement is for the destination register value. The value is compared to its counterpart in the functional simulator. A single macroinstruction may affect more than one register. For instance, the pop instruction will place a value into a register as well as update the stack pointer register, ESP. Therefore, every microinstruction that retires has its destination register checked. This ensures that all registers that may be modified by a macroinstruction are verified. Microinstructions that do not have a destination register or have a temporary destination register skip this checking.

The functional simulator also handles exceptions, such as system calls. When an instruction that triggers an exception attempts to commit, a status flag is set. The simulation environment recognizes this flag and signals the functional simulator to handle the exception. Once the functional simulator is finished, the updated register state is copied to the Verilog simulation from the functional simulator. The Verilog simulation also has its memory model cleared, so that updated values will be retrieved from the functional simulator. After state has
been updated, the Verilog simulation restarts at the instruction following the exception instruction.

![Diagram of Simulation Environment]

**Figure 11.1 Simulation Environment**

11.2 Benchmarks

The design was tested with micro-benchmarks and one SPEC benchmark. The benchmarks were chosen to represent general-purpose tasks. An effort was made to select a diversity of benchmarks that would stress several different aspects. Also, the diversity would
provide a wider range of instructions to improve verification. The benchmarks were compiled with a compiler designed to use the reduced x86 as a target. Many of the features available with a traditional compiler were supported. Many popular libraries were supported as well, such as the ability to do “soft” floating-point operations and memory management operations such as “malloc”. Table 11.1 shows each benchmark in each category.

The first group of benchmarks is the kernel class of benchmarks. Kernels are code modules that might be seen as part of a larger, more complex application. These benchmarks included quick sort, bubble sort and Fast Fourier Transform (FFT). Each of these benchmarks provided highly unpredictable branch cases. FFT also provided a benchmark with floating-point arithmetic.

The second group of benchmarks is the “toy” class of benchmarks, which contains interesting benchmarks that have no direct usage case. The only benchmark in this group is Fibonacci. This benchmark provides a case of deep function recursion during its execution.

The third group of benchmarks is the SPEC2K class of benchmarks. The only benchmark used from this group is Bzip. This benchmark has the closest real-world application use case among benchmarks in Table 11.1. SPEC2K is also a widely used benchmark suite for academic experiments.

The last group of benchmarks is the diagnostic class of benchmarks. This class includes the 2.4 IPC Loop and 3.0 IPC Loop. These two benchmarks feature loops that were generated with an easily calculable IPC. These benchmarks are useful in two respects. Firstly, they are useful for gauging how well the microarchitecture does in extracting ILP. In particular, one would expect the microarchitecture to extract the full amount of ILP of the
loop, given sufficient window and width resources. Moreover, the efficiency with which this ILP is extracted can be gauged (in terms of resources required). Secondly, while many of the other benchmarks exhibit very low IPCs, either due to high branch misprediction rates or lack of ILP, these loop benchmarks provide relatively high IPC cases that can be used to demonstrate the out-of-order effectiveness of the core and its ability to outperform in-order solutions.

Table 11.1 Benchmark Categories

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Toy</th>
<th>SPEK2K</th>
<th>Diagnostic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick Sort</td>
<td>Fibonacci</td>
<td>Bzip</td>
<td>3.0 IPC Loop</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td></td>
<td></td>
<td>2.4 IPC Loop</td>
</tr>
<tr>
<td>FFT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11.3 Results and Analysis

11.3.1 Performance

In order to test the performance of the FabScalar x86 core, several benchmarks were ran with three different superscalar configurations: small, medium and large. The stage widths, structure sizes, and predictor sizes for these configurations are shown in Table 11.2, Table 11.3, and Table 11.4, respectively.

The varying sizes each correspond to a different level of superscalar width. The small core is approximately 2-wide, the medium core is approximately 4-wide, and the large core is approximately 8-wide. Other structures are sized appropriately to support the required superscalar width for each configuration.
Table 11.2 Configuration Stage Widths

<table>
<thead>
<tr>
<th></th>
<th>Small</th>
<th>Medium</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch Byte Width</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>Pre-Decode Width</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Decode Width</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Dispatch Width</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Issue Width</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td># Simple Lanes</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td># Simple+Complex Lanes</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td># Load/Store Lanes</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td># Control Lanes</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Commit Width</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 11.3 Configuration Structure Sizes

<table>
<thead>
<tr>
<th></th>
<th>Small</th>
<th>Medium</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch Buffer</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>Macro-Inst. Buffer</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>Micro-Inst. Buffer</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>Active List</td>
<td>64</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>Physical Reg. File</td>
<td>96</td>
<td>150</td>
<td>300</td>
</tr>
<tr>
<td>Flag Physical Reg. File</td>
<td>96</td>
<td>150</td>
<td>300</td>
</tr>
<tr>
<td>Issue Queue</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>Load/Store Queue</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>CTI Queue</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
</tbody>
</table>

Table 11.4 Configuration Control Structure Sizes

<table>
<thead>
<tr>
<th></th>
<th>Small</th>
<th>Medium</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTB</td>
<td>4K entries</td>
<td>4K entries</td>
<td>4K entries</td>
</tr>
<tr>
<td>BP</td>
<td>64K entries</td>
<td>64K entries</td>
<td>64K entries</td>
</tr>
<tr>
<td>Return Address Stack</td>
<td>32 entries</td>
<td>64 entries</td>
<td>128 entries</td>
</tr>
</tbody>
</table>
Figure 11.2 shows the micro-IPC for the different benchmarks for each configuration. Figure 11.3 shows the macro-IPC for the different benchmarks for each configuration. Figure 11.4 shows the branch mispredictions per 1,000 microinstructions for each benchmark on each configuration. Lastly, Figure 11.5 shows the load violations per 1,000 microinstructions for each benchmark on each configuration.

![Micro-IPC Graph]

**Figure 11.2 Micro-IPC**
Figure 11.3 Macro-IPC

Figure 11.4 Branch Mispredictions per 1K Instructions
Many of the benchmarks, such as FFT, Quick Sort, Fibonacci, and Bzip, do not show the expected result of higher performance with a larger core design. The branch misprediction results in Figure 11.4 show that several of the benchmarks are dominated by branch mispredictions. Moreover, for some benchmarks, the misprediction rate increases with a larger window (possibly due to branch predictor update latency). Figure 11.5 shows that load violations do not play a major role in the behavior, except possibly for FFT. On the other hand, the diagnostic benchmarks (2.4 IPC and 3 IPC Loops), which have almost no branch mispredictions, show the expected scaling behavior. As seen in Figure 11.2 and Figure 11.3, the IPCs of the diagnostic benchmarks increase with larger and wider cores, and approach their theoretical IPC limit on the Large core.
11.3.2 Diagnostic Performance Testing

In order to test the performance of the core a benchmark was developed with an easily estimated potential peak IPC. This benchmarks was then ran on the x86 FabScalar core as well as the PISA FabScalar core. The PISA and x86 FabScalar core were sized comparatively with an approximate superscalar width of four.

Figure 11.6 shows the main loop body for the benchmark. The loop was manually unrolled to assure consistent treatment across the multiple compilers used to compile for the different targets. The theoretical peak micro-IPC was determined to be 2.4.

```
for(i = 0; i < (N/10); i = i+10) {
    x += a[i];
    x += a[i+1];
    x += a[i+2];
    x += a[i+3];
    x += a[i+4];
    x += a[i+5];
    x += a[i+6];
    x += a[i+7];
    x += a[i+8];
    x += a[i+9];
}
```

Figure 11.6 Benchmark Loop Code

Figure 11.7 shows the increase in micro-IPC as the Issue Queue size is increased. For these experiments, the Active List, Physical Register Files, and LQ/SQ were sized to not be
resource bottlenecks: 256, 280, and 64 entries, respectively. As would be expected, the micro-IPC increases as the Issue Queue size increases since this relates directly to the window size available to locate independent instructions to extract parallelism. Both the x86 FabScalar core and PISA FabScalar core come very close to reaching the peak micro-IPC, which is annotated in the graph with the horizontal line labeled “Bound”. However, it seems that after an Issue Queue of 64 entries, the returns begin to diminish for both cores.

![IPC with Varying Issue Queue Size](image)

Figure 11.7 Micro-IPC of Cores with Varying Issue Queue Size
Figure 11.8 shows how micro-IPC increases as the Load/Store Queue size is increased. In this experiment, the Issue Queue size is set to 64 based on the previous set of experiments, which showed the peak micro-IPC being approached at this Issue Queue size. The Active List and Physical Register Files remain large, as before.

Figure 11.8 Micro-IPC of Cores with Varying Load/Store Queue Size
Based on the results for the Load/Store Queue sizing in Figure 11.8, a size of 64 entries was chosen for PISA FabScalar and 16 entries for x86 FabScalar. The same experiment was completed again, except now the number of Active List entries was varied. Figure 11.9 shows the results.

The Issue Queue results were as expected, with the micro-IPC increasing with Issue Queue size. Also, the PISA FabScalar core’s performance was sensitive to the sizes of the Load/Store Queue and Active List. However, the x86 FabScalar core does not seem to respond similarly to these two resources, in that smaller sizes for these structures do as well as larger sizes. This interesting phenomenon will be investigated in future work.
11.3.3 x86 ISA

Introduced in this implementation were variable-length instructions. This allows x86 instructions to have a length between 1 and 12 bytes compared to the constant 8 bytes for every PISA instruction. Figure 11.10 shows the average instruction length in bytes for different benchmarks. The average instruction length was measured by running the benchmarks, i.e., this is a dynamic analysis and not a static analysis. The average instruction length varies between 2 and 4 bytes across the benchmarks, with most having average instruction lengths close to 3 bytes.

Also introduced in this implementation was advanced instruction splitting. This resulted in several microinstructions being generated per macroinstruction. By running different benchmarks the average number of microinstructions generated per macroinstruction can be calculated. Figure 11.11 shows the average split ratio (number of microinstructions per macroinstruction, on average) for different benchmarks. The average split ratio ranges between 1.5 to 2.2 across the benchmarks, with a majority close to 2.
Figure 11.10 Average Instruction Length

Figure 11.11 Split Ratio
Notice that the two diagnostic benchmarks, which are the simplest of the benchmarks, have the smallest ratio of microinstructions to macroinstructions. The diagnostic benchmarks use simple x86 instructions, hence, not many microinstructions need to be generated for each macroinstruction.

Fixed-length ISAs provide a fixed ratio of bytes to instructions. For instance, with PISA, there is a single instruction per eight bytes creating a ratio of 8 bytes-per-microinstruction. With instruction splitting and variable-length instructions in x86, this ratio varies based on the program being executed. By dividing the average instruction length by the average split ratio (how many PISA-like microinstructions each x86 macroinstruction is split into), a comparable metric can be obtained: bytes-per-microinstruction, which we call “instruction density”. The results are shown in Figure 11.12. The interesting thing about the instruction density metric is that it gives a sense of how many bytes are needed to encode the same amount of work measured in microinstruction-equivalents. As can be seen in Figure 11.12, x86 is much more dense than PISA, having a density that ranges between 1 and 2 bytes per microinstruction.
Figure 11.12 Instruction Density
Chapter 12: Summary

This thesis presented the modifications needed to support a reduced x86 ISA on a previous PISA FabScalar implementation. The flexibility of FabScalar allowed the previous implementation to be modified and added upon while still following the same general architecture.

12.1 Summary

Four main areas of modification were needed to update the existing PISA FabScalar core to support x86. The four main areas of modification were to support variable-length instructions, instruction splitting, the flag register, and multiple operand sizing. Figure 12.1 shows the updated FabScalar template with each of the four main modifications having their respective changes color-coded.

The first large-scale change was to support variable-length instructions. The fetching of instructions had to be modified to now fetch bytes rather than instructions. The branch prediction logic had to be modified to detect branch instructions at any byte position, handle branch instructions of different lengths, and handle partially-fetched branch instructions. In addition, length decoders were added to pre-decode the fetched bytes and separate them into x86 instructions. Throughout the process, any previous logic, which relied on instructions being of a fixed length, had to be changed. This logic now relies on the specific lengths of instructions, which can vary between one and twelve bytes.

To handle the CISC instructions introduced by x86, the splitting of macroinstructions into microinstructions was significantly expanded and modified. The splitting of instructions
can now generate up to five microinstructions per macroinstruction. Memory locations can be used as source and destination operands and are handled using generated loads and stores, respectively. Some of the more complex x86 instructions are split into unique sequences of microinstructions to complete the multiple operations conveyed by the x86 instruction.

The x86 ISA introduced the use of a flag register. Two options for renaming the flag register were considered, either renaming it to the general-purpose physical register file along with all other logical registers or renaming it to a dedicated physical register file. The unified option drastically increases complexity by nearly doubling ports to all renaming-related structures and increasing the physical register file size to accommodate many instances of the flag register. The separate option requires duplicating all renaming-related structures but is overall more complexity-effective and was therefore implemented. With the flag physical register file in place, the Execution Lanes were modified to support reading and writing of the flag source and destination registers. Within the Execute stage, changes were made to test and set the flag data. Many arithmetic and logical instructions set the flag data based on their results while control instructions test the flag data to determine their directions.

The last large-scale change was adding support for the different operand sizes of 8, 16 and 32 bits supported in x86. The Decode stage had to be modified to appropriately set the operand size of instructions. The Execute stages were modified to allow operations on different data sizes. Also, the Execute stage now supports the ability to read and write parts of registers while maintaining unmodified parts of registers for partial writes.
The functional correctness of the Verilog implementation is checked using an independent C++ model. The model compares its assumed perfect state to the state of the Verilog implementation to verify correct operation on a per-microinstruction basis. Results were presented from running several benchmarks. The variety of benchmarks that successfully run demonstrates the correctness of the processor implementation and the support for a wide range of instructions. IPC performance was analyzed along three fronts:
configurability of the x86 FabScalar implementation, scaling of IPC with resources and comparing to theoretical peak IPC for microbenchmarks, and IPC comparisons between x86 FabScalar and PISA FabScalar. Similar microinstructions-per-cycle between x86 and PISA FabScalar demonstrate the similarities between the underlying designs.

12.2 Future Work

There are several possible ways to expand upon this work. The first would be trying to improve the performance of the processor. There are a few areas in fetch and decode where there is potential for new and more complex microarchitectures to be implemented that provide increased performance. Another expansion would be support for more x86 instructions. The current reduced x86 implementation lacks support for many features of x86 such as floating-point, string operations and permission levels. Supporting these features would expand the possible code base that could run correctly on the core. Lastly, it would be interesting to analyze the cycle time, area, and power differences between the x86 and PISA FabScalar implementations, for a wide range of core configurations. This would provide insight into the overall differences between implementing a CISC ISA and RISC ISA and the effects this fundamental distinction might have on cycle time, area, and power consumption.
REFERENCES


