ABSTRACT

SHASTRI, ASHLESHA VIJAY. Microarchitectural Implementation of the MIPS Floating-point ISA in FabScalar-generated Superscalar Cores. (Under the direction of Dr. Eric Rotenberg.)

An implementation of the MIPS floating-point ISA in the FabScalar toolset is presented. The major focus of the thesis is on the microarchitectural design details. The MIPS32 Release 2 ISA is used with its 64-bit floating-point unit specification.

The structured and streamlined model of the FabScalar toolset makes it possible to replicate many of the existing structures in the toolset for the augmentation of the floating-point support. The processor front-end is modified to support floating-point instructions. Most notably, the instruction rename and issue components, present in the Rename and Issue stages, respectively, are replicated for the floating-point side and logic is added to correctly steer instructions to components based on their source and destination operand types. In the FabScalar toolset, the back-end is organized in the form of parallel execution lanes with each instruction having dedicated resources for it when it executes. The floating-point execution lane has a similar structure as that of the integer simple lane, only that it accesses different sets of resources.

The load-store lane is extended to handle floating-point loads and stores and thus establishes cross-interaction between the two sides (loads and stores use the integer side for addresses and the floating-point side for values). Since there are other instructions in the ISA that communicate between the integer and floating-point sides, we use the load-store lane to also execute these instructions. The thesis also considers alternative solutions for the same, to put into perspective the efficiency of our unified load-store lane approach.

As a commercial grade ISA, MIPS has some sophisticated instructions. The thesis describes the implementation of these instructions. Alternative implementation approaches are also discussed. There was a lot of thought put into the implementation of such types of instructions. The foremost considerations throughout were to use as many existing structures as possible, minimize additional resources and work within FabScalar's structure, i.e., avoid fundamental structural changes or cycle-time-degrading changes. For example, FabScalar's existing strategy for splitting intrinsically expensive instructions into multiple, existing simpler instructions, is heavily leveraged.

Microarchitectural Implementation of the MIPS Floating-point ISA in FabScalar-generated Superscalar Cores

by
Ashlesha V. Shastri

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

2012

APPROVED BY:

_____	_____
Dr. Eric Rotenberg Committee Chair	Dr. Rhett W. Davis
_____	
Dr. Huiyang Zhou	

DEDICATION

To my parents and my sister…

BIOGRAPHY

Ashlesha Shastri was born in Maharashtra, India, on 30th March, 1989. She earned her Bachelor of Engineering degree from K.J. Somaiya College of Engineering, Mumbai, India, in 2010. She joined the North Carolina State University in fall, 2010, to pursue her Master of Science degree in the field of Computer Engineering. She will be receiving her Master of Science degree with the defense of this thesis.

ACKNOWLEDGMENTS

I want to thank my advisor Dr. Eric Rotenberg for his guidance throughout the course of my thesis. It was a very enriching experience to have worked with him.

I would like to thank Dr. Rhett Davis and Dr. Huiyang Zhou for agreeing to serve on my thesis committee.

I would like to thank Tomoyuki Nakabayashi and Dr. Takahiro Sasaki, of Mei University, Japan, for suggesting the implementation of conditional move instructions used in this thesis.

I would like to thank Brandon, Niket and Elliott for helping me and answering all my questions very patiently.

I would like to thank my family for encouraging and supporting me to help achieve my goals.

This research was supported by NSF grants CCF-0811707 and CCF-1018517. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the National Science Foundation.

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 5
2.1 FabScalar Template 5
2.1.1 Front-End 5
2.1.2 Back-End 8
2.2 Unified FabScalar Template 10

Chapter 3 Decode 14
3.1 General Decode strategy 14
3.2 Instruction Splitting 15
3.2.1 Floating-point Stores 16
3.2.2 Floating-point Compares 17
3.2.3 Conditional Moves 18
3.2.4 Move to Floating-point 20
3.3 Instruction Buffer 21

Chapter 4 Rename 24
4.1 Organization 24
4.2 Rename circuitry 26
4.3 Interaction with Retire stage 30

Chapter 5 Dispatch and Issue 31
5.1 Organization of Dispatch 31
5.2 Organization of Issue 33

Chapter 6 Execution Lanes 37
6.1 Branch lane 37
6.2 Simple and Complex lane 38
6.2.1 Implementation of integer conditional move 38

6.3 Load-Store lane ……40
6.4 Floating-point lane ……...46
6.4.1 Implementation of Floating-point compares ……47

Chapter 7 Retire ……...49
7.1 Retire circuitry ……49
7.2 Interface with the testbench ……50

Chapter 8 Results ……52
8.1 Simulation setup ……...52
8.2 Results and analysis ……53

Chapter 9 Summary ……59

References ……61

Appendix ……..63
Appendix A MIPS FPU Instruction Formats ……64

LIST OF TABLES

Table 3.1 Basis of instruction splitting …… 15
Table 3.2 Floating-point Store instructions…….. 16
Table 3.3 Floating-point Compare instruction ……17
Table 3.4 Conditional Move instruction ……..18

Table 4.1 Integer and floating-point architectural registers ……27

Table 5.1 Instructions with different type of source and destination ……...34

Table 6.1 Load-store lane bypass usage by instructions ……44

Table 8.1 Inflection points of various resources for 470.lbm benchmark…… 57

LIST OF FIGURES

Figure 2.1 FabScalar template 6
Figure 2.2 Unified FabScalar template 12

Figure 3.1 Decoded instruction packet 14
Figure 3.2 Splitting of floating-point store instruction 17
Figure 3.3 Splitting of floating-point compare instruction 18
Figure 3.4 Operation of conditional move instruction 19
Figure 3.5 Splitting of floating-point move conditioned on integer register 20
Figure 3.6 Splitting of 'move to higher half of floating-point register' instruction 21
Figure 3.7 Instruction buffer input bundle 22

Figure 4.1 Organization of Rename stage 25
Figure 4.2 (a) Instruction bundle and rename state before renaming 28
Figure 4.2 (b) Instruction bundle and rename state after renaming 29

Figure 5.1 (a) Execution lanes 32
Figure 5.1 (b) Instruction bundle input to the scheduler 32
Figure 5.1 (c) Timing diagram showing output of the Execution-pipe scheduler for a stream of instructions 32
Figure 5.2 Instruction writing to the issue queue 35
Figure 5.3 Collapsing of dispatch packet inside the issue stage 36

Figure 6.1 Alternative approach to split conditional moves 39
Figure 6.2 Modified load-store lane 43
Figure 6.3 Direct Programming Interface used in floating-point ALU 47
Figure 6.4 Alternative way to split floating-point compare 48

Figure 8.1 Block diagram of the simulation setup 52
Figure 8.2 Graph showing variation of IPC with Issue Queue size 54
Figure 8.3 Graph showing number of load violations and branch mispredictions with varying Issue Queue size 55
Figure 8.4 Graph showing the variation of IPC with Load/Store Queue size 56
Figure 8.5 Graph showing the variation of IPC with Active List/PRF size 57
Figure 8.6 Pie chart showing the distribution of instructions for the 470.lbm benchmark 58

CHAPTER 1

INTRODUCTION

As the technology scaling slows, heterogeneous multi-core processors provide an exciting new direction to improve processor performance. The downside is that the design and verification effort get multiplied with the increasing number of different core designs in the multi-core processor.

The FabScalar project [1] aims to provide a solution to this problem by creating a toolset to improve designer's productivity. The toolset achieves this by automating the generation of synthesizable RTL for a processor core, within a canonical superscalar template.

1.1 Motivation

The FabScalar toolset aids designers and researchers by providing RTL designs for arbitrary superscalar processors. The very structured and streamlined design of the superscalar template also makes it easier to understand the design. The tool, however, lacks support for handling floating-point instructions. This handicaps the tool when it comes to executing modern-day complex workloads with floating-point instructions.

Also it is well known that having floating-point support yields better performance. Thus the key motivation of this thesis is to extend the FabScalar toolset with floating-point instruction support.

1.2 Related Work

The Illinois Verilog model (IVM) [6] is designed to test the fault tolerance on high performance microprocessors. The microarchitecture is a superscalar, dynamically scheduled pipeline. The model implements a subset of the Alpha ISA. However it lacks floating-point instruction support. When it comes to floating-point computations, where no computation is exact, the error detection and correction becomes all the more difficult [3]. Hence to have a better picture of the fault tolerance in modern day microprocessors, it is important to test floating-point instructions.

OpenSPARC T1 [4] is an open-source processor model which implements the 64-bit SPARC v9 architecture. The processor has a floating-point unit which implements the SPARC v9 floating-point instruction set. However, the floating-point unit does not implement some floating-point arithmetic instructions. In addition it also does not execute conditional moves, loads and stores.

Floating-point unit in the OpenSPARC T2 [5], on the other hand, supports the unsupported instructions in OpenSPARC T1. Both the models have a fixed floating-point issue width of one.

1.3 Outline

The FabScalar toolset with integer instruction support is extended and modified to include the support for floating-point instructions. We use the MIPS32 Release 2 ISA with its 64-bit floating-point unit specification. The implementation of the 64-bit floating-point unit, however, is not backward compatible with its 32-bit specification as the ISA demands [7].

The choice of designing a 64-bit floating-point unit is based on the following two factors:

- The floating-point double-precision format is 64-bits wide, and we wanted to allow double-precision arithmetic because of the higher degree of precision and accuracy it has to offer. This is a subset of the fact that the format can represent larger range of numbers in comparison to the 32-bit wide single-precision format.
- As defined by the ISA [7], double-precision arithmetic, if supported with 32-bit wide floating-point registers, requires that the double-precision value be stored in even/odd register pairs. This would imply adding a lot of complexity to the hardware. Hence, we have chosen the 64-bit wide registers for the floating-point datapath.

There were many alternative solutions to implement the design of the pipeline, out of which we chose the best based on certain aspects which we think suited the framework of the FabScalar toolset the best.

The major focus of the thesis is on the micro-architectural design of the floating-point unit. We have tried to use as many existing structures as possible. For example, the major rename components were replicated for the floating-point side and logic was added to steer the instruction's source and destination operands to the correct rename module based on their 'type'. We also use the issue queue components and replicate it for the floating-point side. We then direct the instructions to the correct issue queue based on their source operands' 'type'.

FabScalar's existing strategy for splitting intrinsically expensive instructions into multiple, existing simpler instructions, is heavily leveraged. We split some of the sophisticated instructions introduced by the MIPS ISA in the decode stage.

The implementation of interaction between the integer and the floating-point side was a challenge. There was much thought put into taking the design decision. We have one load-store lane which handles all the cross-communication between the two sides. We also discuss a few alternatives to our approach. The implementation strategy for instructions like conditional moves was carefully selected amongst various other possible solutions which are also discussed in the thesis.

The thesis delves into the details of every pipeline stage and the modification made in the stage to add floating-point support. Chapter 2 discusses the existing FabScalar framework and Unified framework with floating-point support added. Chapters 3 to 7 discuss the individual pipeline stages in detail. Chapter 8 presents the results and Chapter 9 is the summary and future work.

CHAPTER 2

BACKGROUND

Even though, the FabScalar toolset assumes a canonical template, it allows the user to define the complexity within each pipeline stage by varying the superscalar width and pipelining depth. The way this is achieved is by having a single core which is the widest dimension and maximum possible pipelining depth called - the superset core. Preprocessor macros are inserted in the code wherever needed to be able to achieve the desired dimension. The preprocessor strips away parts of the code to achieve the desired width and depth of the core.

2.1 FabScalar Template

The template as shown in Figure 2.1 has nine canonical pipeline stages. The Front-end comprises of the stages through and including Fetch and Dispatch and Back-End comprises of stages through and including Issue and Retire.

2.1.1 Front-End

Fetch is the first canonical pipeline stage. The instruction Fetch is further divided into two sub-stages Fetch-1 and Fetch-2.

Fetch-1 stage contains the Branch Predictor (BP), Branch Target Buffer (BTB), Return Address Stack (RAS) and L1 I-Cache. The stage also contains the next-PC mux which

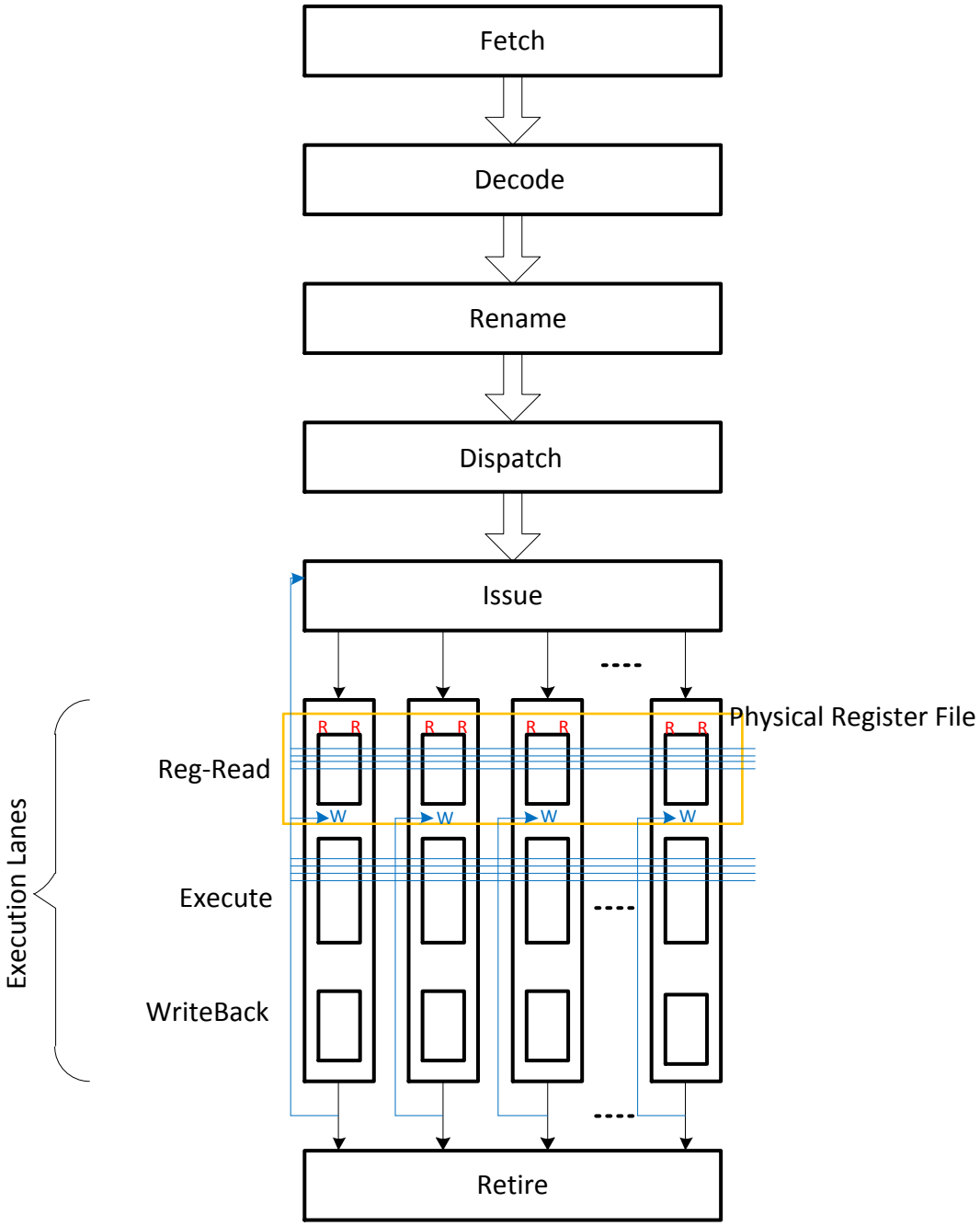


Figure 2.1: FabScalar template

generates the next fetch program counter. Fetch-2 stage contains the control queue - a structure to ensure in-order update of the Branch Predictor and BTB.

The output of Fetch-2 is the Decode bundle comprising of N or less instructions, where N is the fetch width of the processor. The bundle may comprise of less than N instructions because we terminate the bundle at the first taken branch (based on prediction of BP). Fetch-2 also instantiates the Pre-Decode block, which decodes only the branch instructions in the fetch bundle, to check for missed branches on account of a BTB miss.

Every cycle the Decode stage receives up to N instructions. The Decode stage has N number of decoders instantiated to decode N instructions in parallel. The instruction formats are interpreted by the decoder to generate a packet per instruction, which contains information required to process the instruction by further stages. Each instruction can be split into 2 micro-instructions, out of which the packet containing the second split part will be valid only if the instruction is split. Thus, for fetch width of 4 the output of decode will be 8 decode packets.

The decoded instruction packets get written into the Instruction Buffer, a circular FIFO, which has N*2 write ports and N read ports. The buffer allows us to continue fetching and decoding newer instructions, even if the back-end is stalled. Even though the instruction buffer has N*2 write ports for the worst case, only the valid packets get written into the buffer. N packets are popped from the head of the buffer, when back-end is ready, and go to the Rename stage.

In the Rename stage, an instruction’s logical destination (if valid) gets mapped to a physical register popped from the free list. The source operands are renamed by reading the

physical mappings for corresponding logical registers, from the Rename Map Table (RMT). The bypass logic in the Rename stage checks for register dependencies between the instructions in a bundle. The RMT gets restored from the Architectural Map Table (AMT) in case an exception or branch misprediction is detected in the retire stage. The dispatch bundle formed contains the physical registers, and is sent to the Dispatch stage.

The Dispatch stage gets N renamed instructions. It extracts the relevant information from these packets and sends them to the Issue Queue, Load-Store Queue and the Active List. The loads, in the current implementation, execute speculatively in the case of unknown prior store addresses. The load violation predictor in the dispatch stage marks the load in the dispatch bundle, if it has violated in the past. The predictor is a direct-mapped structure which gets updated with the load PC, broadcasted from the Active List, at retire time. The Execution pipe scheduler in the dispatch stage assigns an execution lane number to an instruction based on its opcode. The instruction arbitrates for that lane in the issue queue. This simplifies the select logic in the Issue stage.

The front-end design is so that the instructions, after they issue from the issue queue, can flow freely in the execution lanes.

2.1.2 Back-End

The Issue stage is the first of the back-end stages. The issue queue free list outputs free issue queue entries for the new instructions dispatched. The instructions that get dispatched into the issue queue write their information into the issue queue payload RAM at the entries indicated by the free list. An instruction becomes ready to issue when both its source

operands are available. The instruction selection based on the readiness of an instruction's source operands is the main concept of out-of-order execution.

The Select logic in the issue queue selects one instruction per execution lane every cycle from all the instructions which are ready to issue. These granted instructions, wake up their dependent instructions in the next cycle and also set the destination physical register's ready bit in the ready bit array for future instructions. After an instruction issues, it can now flow freely through the execution lane because it has dedicated resources. Each execution lane has the Reg-read, Execute and Writeback stages.

In the Reg-read stage, the instruction reads the source operand values from the Physical Register File. The value which flows with the instruction is either the value read from the register file or the value from the bypass. The value on the bypass has higher priority, if the instruction's source register tag matches the destination tag broadcasted on any of the bypasses. The bypasses are present to ensure back-to-back execution of dependent instructions.

The Physical Register File has 2 read ports and 1 write port per execution lane as shown in Figure 2.1. The destination value gets written into the Physical Register File in the Writeback stage. The Reg-read stage can be sub-pipelined by reading the register file value in multiple cycles.

The instructions can now execute since they have the values of their sources. But if the producer instruction for this instruction's source was in the writeback stage, the value it has would be stale. Hence the value is bypassed to the start of Execute stage also. The load-store lane has the address generation logic, for the load and store instructions.

In the Writeback stage, the instruction writes the destination value into the Physical Register File and broadcasts it to the bypass logic in the execution lanes. The Active List is updated with the execution/status flags for the particular instruction.

The Active List commits instructions in program order. If the head instruction of the Active List is a store, it signals the store queue to commit the store to the data cache. If the head instruction has a valid destination register, the AMT is updated with the current mapping. This update of the AMT frees the old physical register and it is pushed onto the Free List. The logic in the Active List makes sure to commit split instructions as a pair.

The FabScalar template described above is extended and modified to add floating-point support. Modifications to the template have been made wherever required. Section 2.2 explains in brief extensions to the FabScalar template to enable floating-point support.

2.2 Unified FabScalar Template

Figure 2.2 shows the unified template for FabScalar. The brief overview of the template is explained in terms of its differences from the template described in Section 2.1. Details of the implementation and the design choices involved are elaborated in the following chapters.

1) The decoder now decodes some extra information about the instructions. It adds the ‘type’ of the instruction’s source and destination registers to the decode packet. This is added to the decode packet to ensure correct steering of the instructions to either the integer or floating-point issue queue. Source type ‘0’ implies integer operand and source type ‘1’ implies floating-point operand. The instruction is marked if it is a move from or to the floating-point side. Also floating-point store instructions are specially marked.

2) Each instruction can be split into up to three micro-instructions, unlike two before. This led to the increase in the write ports of the Instruction Buffer.

3) The rename circuitry was replicated for the floating-point so as to have separate renamers, one for integer and one for floating-point.

4) The Execution pipe scheduler in the dispatch stage was extended to schedule the floating-point instructions also.

5) Each of the integer and floating-point sides has a dedicated issue queue which issues instructions to the execution lanes connected to it. The issue queue is responsible for selecting instructions to issue, based on their source operand availability, to the lanes which have respective function units. In order to simplify this process, we chose to have separate issue queues.

6) The instructions in the dispatched bundle, which contains integer as well as floating-point instructions, get steered to the integer or the floating-point issue queue depending on the source operand type. Instructions with integer source(s) are written into the integer issue queue and those with floating-point source(s) get written into the floating-point issue queue.

7) The Floating-point Physical Register File (PRF) is similar to the integer register file structure except that it has 64-bit values, unlike 32-bit values in the integer register file. Both the PRF's have an extra MSB bit to support the implementation of conditional moves.

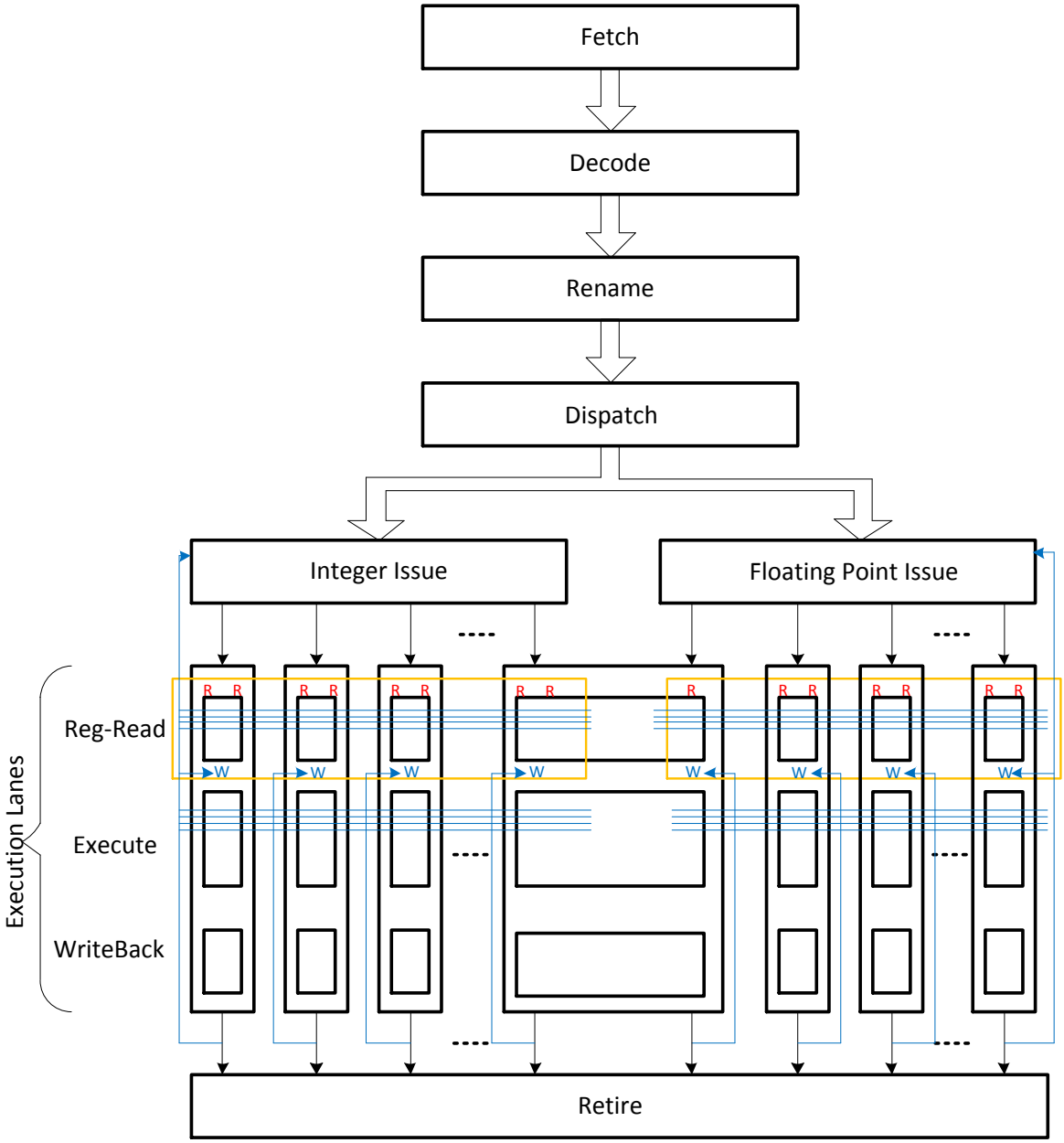


Figure 2.2: Unified FabScalar template

8) A floating-point lane was added. The load-store lane was modified to handle floating-point loads and stores and also support the cross-communication between the integer and the floating-point sides.

9) The structures like the load queue, store queue and active list are common for instructions of both types. The Architectural Map Table (AMT) was replicated for the floating-point side.

The unified template was designed in line with the FabScalar template and effort was to leverage the existing resources. Components of the rename stage and issue stage were replicated for the floating-point side. The fetch stage was not modified and hence is excluded from the discussion.

CHAPTER 3

DECODE

Decode is the pipeline stage after Fetch. The main function of the Decode stage is to interpret the instruction and create a packet with all information regarding a particular instruction, which the instruction needs for correct execution in the future pipeline stages. Refer to Appendix A for the floating-point instruction formats defined by the Mips-ISA.

3.1 General Decode strategy

The instruction format is appropriately decoded by the decoder to form the decode packet as shown in Figure 3.1.

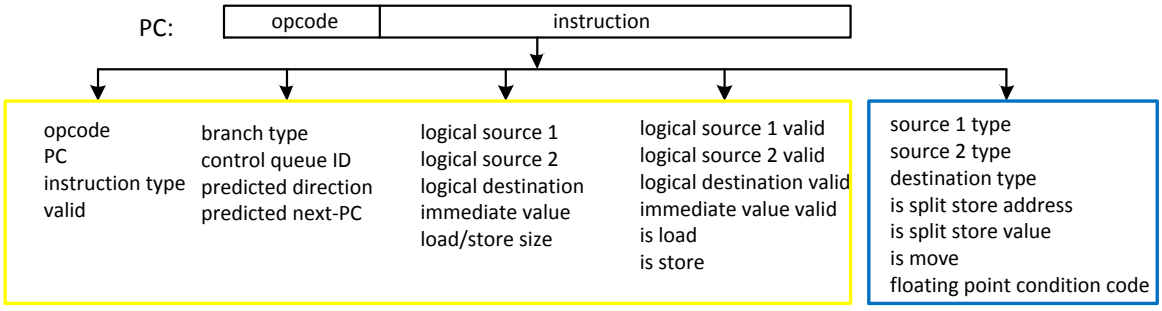


Figure 3.1: Decoded instruction packet

The yellow box has the general decode information. Information in the blue box is added specifically to handle the floating-point instructions in the future stages. The ‘type’ field is added with each of the instruction’s source and destination registers. Type ‘0’ implies an integer type and type ‘1’ implies a floating-point type. The ‘is move’ field is set if the

instruction is a move to/from the floating-point unit. ‘is split store address’ and ‘is split store value’ fields mark the upper and lower parts of the floating-point store instruction, respectively. The ‘floating-point condition code’ field is that bit of the floating-point condition codes register (FCCR) which is updated by the floating-point compare instruction. Floating-point branches and floating-point conditional moves also require this information as they depend on the compare instruction’s result.

3.2 Instruction Splitting

If an instruction falls into one of the categories in Table 3.1, it is a candidate for instruction splitting. Sections 3.2.1 to 3.2.4 discuss the splitting method of each of the following classes of instructions.

Table 3.1: Basis of instruction splitting

Basis of instruction splitting	Instruction opcodes that fall in the category
Instructions with two types of source operands	Floating-point stores, floating-point conditional moves
Instruction modifies only a part of its destination register or conditionally modifies an entire register.	Floating-point compare, Move word to high half of floating-point register, integer conditional moves, floating-point conditional moves
Forced split	Floating-point compare, Move control word to floating-point
Instruction has more than 2 source operands	Multiply-add fusion instructions

We introduce a logical register called the ‘REG_TEMP’ on the floating-point side. It is added to handle instructions which do partial writes to a register. It holds the intermediate result of the operation which is later merged with the destination register value. The following sections explain the way in which the temporary register is used.

3.2.1 Floating-point Stores

The Mips-ISA defines different types of stores as shown in Table 3.2.

Table 3.2: Floating-point Store Instructions

Mnemonic	Instruction
SWC1	Store Word From Floating-point
SDC1	Store Double Word From Floating-point
SWXC1	Store Word Indexed from Floating-point
SDXC1	Store Double Word Indexed from Floating-point
SUXC1	Store Double Word Indexed Unaligned from Floating-point

Floating-point store instructions have two types of source operands. The memory address to store to, is integer. The store value that is stored to the memory address is a floating-point value. As our mechanism to steer the instructions to either the integer or floating-point issue queue is based on the type of the instruction’s source operand, we split the store instruction into two micro-instructions shown in Figure 3.2.

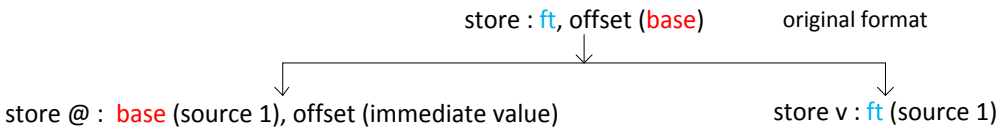


Figure 3.2: Splitting of Floating-point Store Instruction

The ‘Store @’ instruction goes to the integer issue queue and the ‘Store v’ goes to the floating-point issue queue. The split instructions get marked as ‘is split store address’ and ‘is split store value’ respectively. Only the @ part is marked as a store, the reason is explained in Chapter 6.

3.2.2 Floating-point Compares

The floating-point compare instruction as defined by the Mips-ISA is shown in Table 3.3.

Table 3.3: Floating-point Compare Instruction

Mnemonic	Instruction
C.cond.fmt	Floating-point Compare

The floating-point compare instruction compares two floating-point register values and sets a bit (cc) in the floating-point condition codes register (FCCR). Refer to Appendix A for the compare instruction’s format.

As the compare instruction modifies only a part of the FCCR, we split the instruction into two micro-instructions as shown in Figure 3.3 (a). The third micro-instruction, as shown in Figure 3.3 (b), is added because we maintain a redundant copy of the FCCR in both the

integer and floating-point register files. The redundant copy of FCCR facilitates execution of floating-point branch instructions and integer conditional moves based on FCCR on the integer side. The third part of the split instruction is thus a forced split and does not contribute to the actual operation of the compare instruction.

```
compare_1 : REG_TEMP, fs, ft
compare_2 : FCCR_fp, FCCR_fp, REG_TEMP
```

(a)

```
move from : FCCR_int, FCCR_fp
```

(b)

Figure 3.3: Splitting of the floating-point compare

3.2.3 Conditional Moves

Table 3.4 shows all the conditional moves in the Mips-ISA.

Table 3.4: Conditional Move Instructions

Mnemonic	Instruction
MOVT	Integer Move Conditional on Floating-point True
MOVF	Integer Move Conditional on Floating-point False
MOVN	Integer Move Conditional on Not Zero
MOVZ	Integer Move Conditional on Zero
MOVT.fmt	Floating-point Move Conditional on Floating-point True
MOVF.fmt	Floating-point Move Conditional on Floating-point False
MOVN.fmt	Floating-point Move Conditional on Not Zero
MOVZ.fmt	Floating-point Move Conditional on Zero

The conditional moves modify their destination register based on a condition. This means that after executing the move instruction, destination register will have either a new value or

will retain its previous value. Thus we will have to read the destination's previous value before clobbering it. Without splitting, the instruction would thus have three source operands. Hence we split the operation over two instructions. An extra bit is added to the integer and floating-point physical register files to support these instructions [2].

As shown in Figure 3.4, the destination register and the first source register of the first micro-instruction (cmov_1) is the same as that of the conditional move's destination. As 'rd' is the destination it will get renamed, which ensures that the dependence chain of the register 'rd' ends with this instruction. Newer instructions that have 'rd' as their source will be dependent on this new instance of 'rd'. The second source operand is the condition register. The first micro-instruction copies the value of 'rd' from one physical register to another. The micro-instruction also sets the MSB bit of the destination register based on the condition (second source operand).

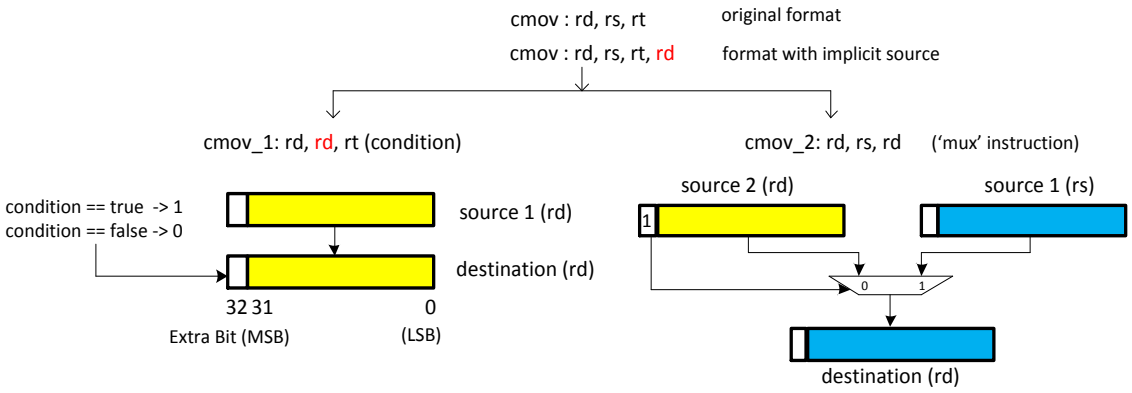


Figure 3.4: Operation of the Conditional Move Instruction

The second micro-instruction (cmov_2), called the 'mux' instruction, writes the destination register with either source 1(original source) or source 2 (implicit source)

depending on the MSB bit of source 2. This mechanism is used for all types of conditional moves shown in Table 3.4, but floating-point moves which depend on an integer register (MOVN.fmt and MOVZ.fmt) get split into 3 micro-instructions as shown in Figure 3.5. As these instructions have two types of sources, the first instruction moves the integer source over to the floating-point side, into the temporary register (REG_TEMP).

```
move to    : rt,  REG_TEMP
cmov_1     : fd, fd, REG_TEMP
cmov_2     : fd, fs, fd
```

Figure 3.5: Splitting of floating-point move conditioned on integer register

Cmov_1 reads the temporary register (which now has the condition register value) as a source and then the operation is the same as that of the other conditional moves.

3.2.4 Move to floating-point

The move to higher half of a floating-point register (opcode: mthc1) modifies only a part of the floating-point register. Thus we split the instruction as shown in Figure 3.6. The first instruction moves the integer register value to the temporary register on the floating-point side. The second instruction reads the temporary register and the destination register (implicit source) and writes the temporary register value in the upper 32 bits of the destination register retaining the lower 32 bits.

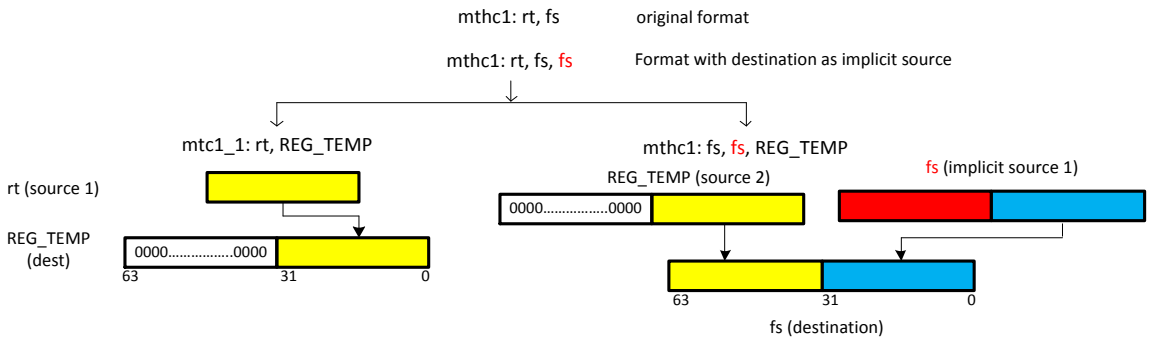


Figure 3.6: Splitting of 'move to higher half of floating-point register' instruction

Control word move to the floating-point side (opcode: ctc1), writes an integer register value to the FCCR. After this instruction, the FCCR value on the integer side would become stale. To maintain the redundancy with the integer FCCR, the instruction is split into two micro-instructions where the second micro-instruction moves the FCCR value from the floating-point side to the integer side. Thus, this instruction is not split to do its own operation but is split forcefully to maintain FCCR redundancy.

3.3 Instruction Buffer

Even if each instruction that gets decoded can be split into three micro-instructions, the valid bits are set only for instructions which are actually split. In Figure 3.7, instruction 2 is split into two micro-instructions and 3 gets split into three micro-instructions. Instructions 1 and 4 are not split at all. Thus, in the decode output bundle formed, 1' and 1" are invalid and the same is true for 4' and 4". Whereas for instruction number three, 3, 3' and 3" all are valid and for instruction number two, only 2 and 2' are valid.

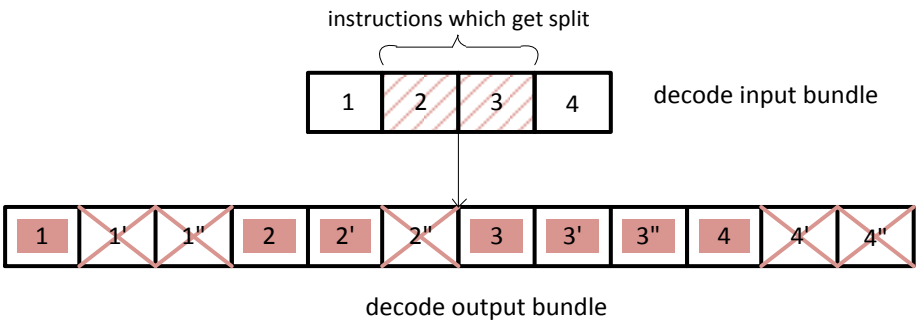


Figure 3.7: Instruction buffer input bundle

The instruction buffer has fetch width times three number of write ports to handle the worst case of each instruction being split into three micro-instructions. The number of read ports is equal to the dispatch width of the processor.

As shown in Figure 3.7, the decode bundle is collapsed and only valid instructions get written into the instruction buffer. In the original FabScalar template, instructions were split into a maximum of two micro-instructions. But as we now split some of the floating-point instructions into three micro-instructions, the instruction buffer's write ports were increased.

The alternative solution that would save us the extra write ports is to decode the instructions which get split into more than two micro-instructions, over multiple cycles. Even though we would save extra write ports, the implementation would complicate the decode logic. Also the front-end will have to be stalled when the Decode stage is decoding for multiple cycles. But it would benefit the cycle time by not having to add the extra write ports. As there was no means to compare the two approaches effectively, we chose to add extra write ports to the buffer.

The instruction buffer sends ‘dispatch width’ instructions to the Rename stage. If the buffer does not have that many instructions, it stalls the Rename stage.

CHAPTER 4

RENAME

The Rename stage does the task of register renaming, and thus is a key stage in the pipeline that enables us to do out-of-order execution. Most processor instruction sets define fixed number of registers for instructions called architectural/logical registers. The compiler would try to detect distinct instructions and assign the registers accordingly, but it is also limited by the number of registers defined by the ISA. This introduces false dependencies between instructions. Hence physical registers are assigned to instructions in hardware to remove the false dependencies. The number of physical registers depends on the hardware design, and is greater than the number of logical registers.

4.1 Organization

The Rename stage in the FabScalar template has the logic to handle instructions with invalid (non-existent) source and destination registers. We leverage this fact and replicate the existing rename circuitry for the floating-point instructions. Thus we now have a Rename stage, within which there is separate rename circuitry for integer and floating-point instructions as shown in Figure 4.1.

The instruction buffer outputs ‘dispatch width’ number of instructions to the Rename stage. The dispatched bundle contains a mix of integer and floating-point instructions. We use the ‘type’ field associated with each of the instruction’s source and destination registers to appropriately steer the operands to the correct rename module. The added logic invalidates

the integer type register for the floating-point renamer and floating-point type register for the integer renamer. The source or destination register is valid for the integer renamer only if it is valid (1) and of integer type (0) and it is valid for the floating-point renamer only if it is valid (1) and of floating-point type (1).

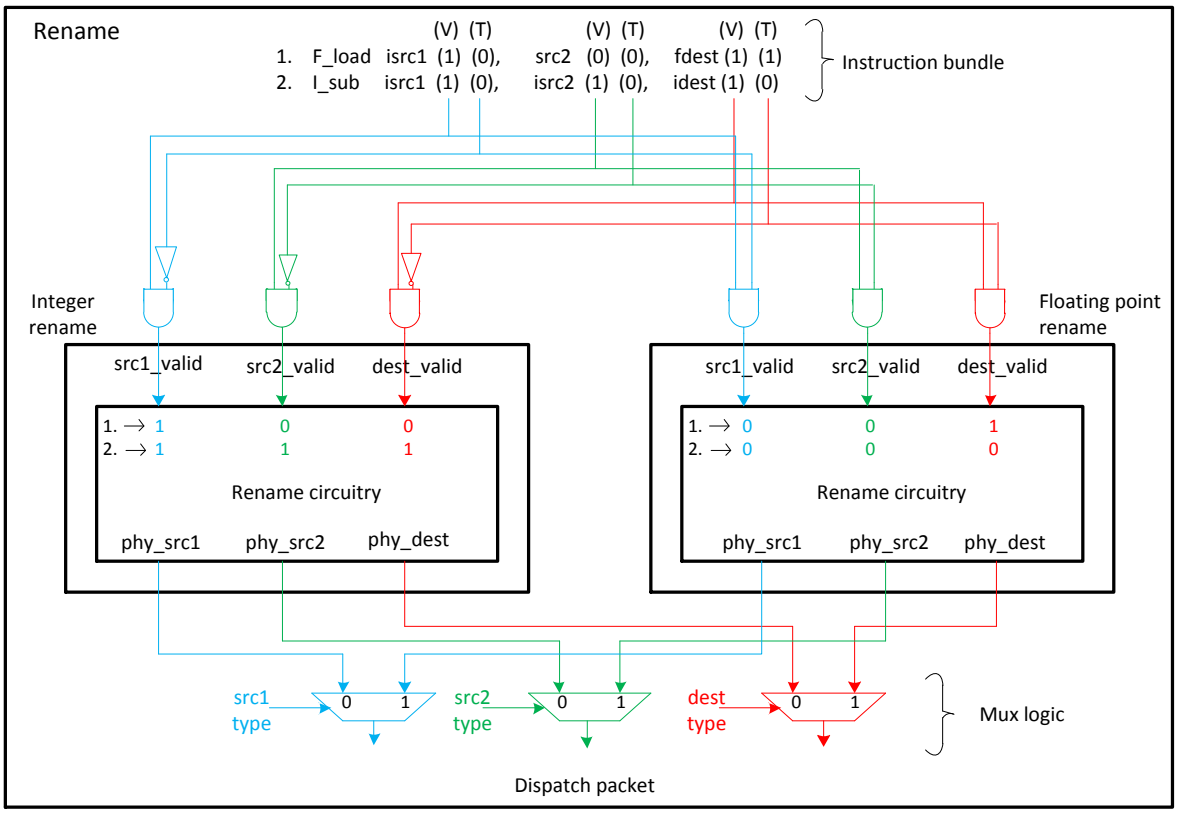


Figure 4.1: Organization of Rename stage

In Figure 4.1, instruction (1) is F_load, a floating-point load with one valid integer source and one valid floating-point destination. The instruction's source and destination register valid bit and type are indicated in brackets by (V) and (T) respectively. The 'Integer rename'

sees only source 1 as valid for the F_load instruction and F_load's destination is valid for 'Floating-point rename'. Thus, the F_load's source gets renamed in the 'Integer rename' and its destination is renamed in the 'Floating-point rename'. Instruction (2) is I_sub, an integer subtract, with two integer sources and one integer destination. As its sources and destination are all of integer type it is entirely renamed in the 'Integer rename'. Similarly, an instruction with all floating-point sources and destination would get renamed entirely in the 'Floating-point rename'.

Individual renamers output the physical mappings of the source and destination register which they see as valid. The 'mux logic' selects between the physical sources and destination coming from the individual renamers based on their types to form the dispatch packet.

The dispatch packet comprises of all the decoded information in addition to the physical register mappings.

4.2 Rename circuitry

The main components of the rename circuitry are the Rename Map Table (RMT) and the Free List (FL). The RMT contains the current physical mappings of the logical registers. The Free List contains the physical registers that are free and hence can be assigned to the destinations of the new instructions in the current rename bundle. The size of the RMT is equal to the number of logical registers and the size of the Free List is equal to the number of physical registers minus the number of the logical registers.

Table 4.1 shows the architectural/logical registers defined. Thus the integer and floating-point RMT sizes are 35 and 34, respectively. The low, high and floating-point condition

codes registers (FCCR) are defined by the Mips-ISA. We define a new logical register called the 'Temp register' to support floating-point conditional moves and compare instructions. In addition, we have the FCCR redundantly in both the RMT's so as to support execution of the floating-point branches in the branch lane and the floating-point compares in the floating-point lane.

Table 4.1: Integer and Floating-point architectural registers

Integer architectural registers		Floating-point architectural registers	
register no.	Name	register no.	Name
0…31	General-purpose registers	0…31	General-purpose registers
32	Low register (LO)	32	Temp register (REG_TEMP)
33	High register (HI)	33	Floating-point condition register (FCCR)
34	Floating-point condition code register (FCCR)	-	-

The Rename stage always gets 'dispatch width' number of instructions which is facilitated by the Instruction Buffer. The RMT has 2 Read ports and 1Write port and the Free List has 1 Read port per instruction. Each of the instructions in the rename bundle obtains the physical mappings of its source operands from the RMT. If the instruction has a valid destination, it pops a free physical register from the Free List.

The forwarding logic checks for register dependencies between the instructions in the bundle and forwards the physical mappings from one instruction to the other in the case of a logical register match.

Figure 4.2(a) and (b) show an instruction bundle, the RMT and the Free List, before and after renaming respectively. An integer register specifier is indicated with ‘r’ and floating-point is indicated with ‘f’. I_add instruction has all integer sources and destination and hence updates Integer rename RMT and Free List. On the other hand, F_load’s destination is renamed in the ‘Floating-point Rename’, but its source is renamed in the ‘Integer Rename’. F_add and F_sub have floating-point sources and destination and hence are renamed by the floating-point renamer.

The output dependence between the F_load and F_add instruction and the anti-dependence between the F_add and F_sub is removed after renaming. Thus register renaming breaks the false dependence chain and allows us to execute the F_add and F_sub independent of I_add and F_load instructions.

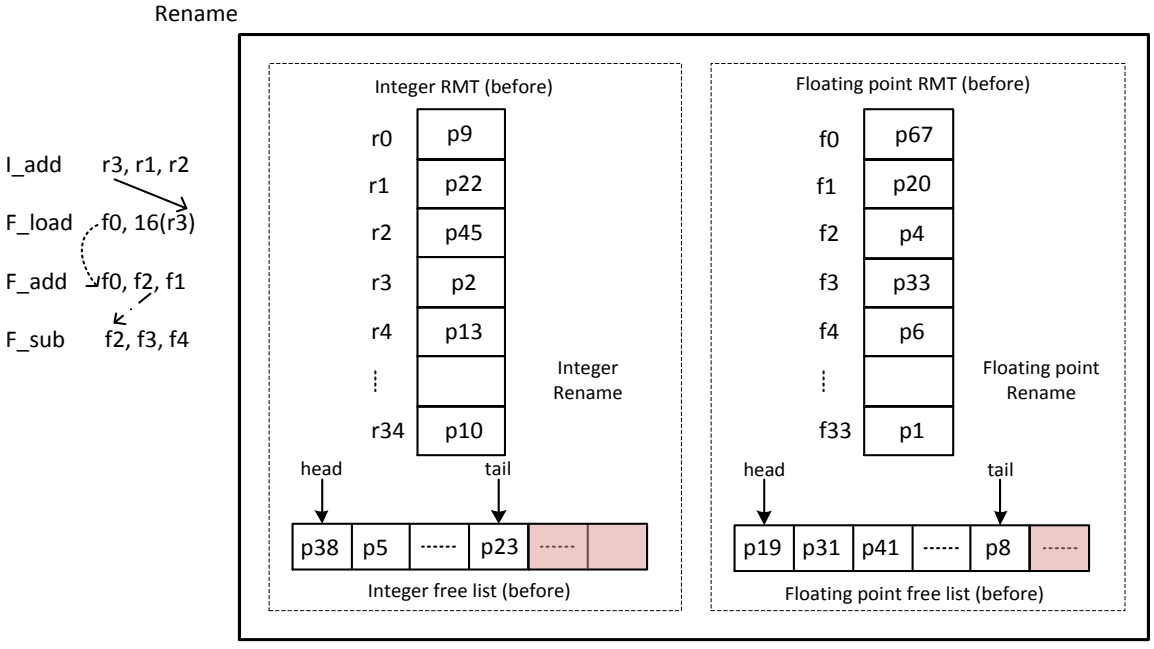


Figure 4.2 (a): Instruction bundle and rename state before renaming

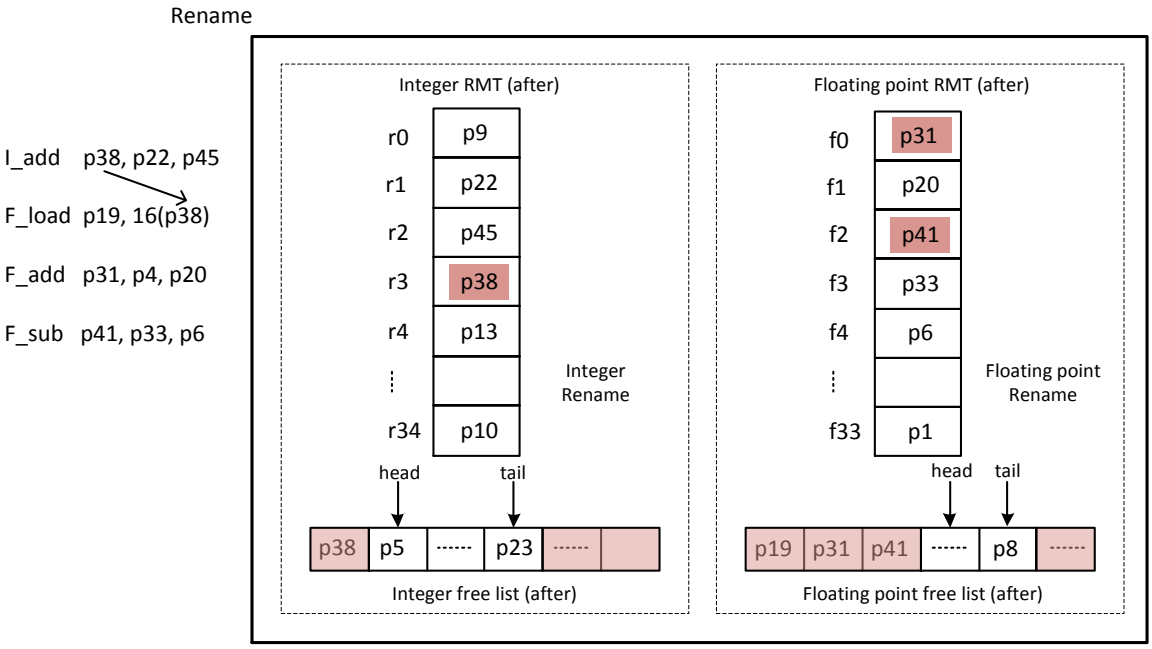


Figure 4.2 (b): Instruction bundle and rename state after renaming

The Rename stage stalls the pipeline register between Instruction Buffer and Rename stage if either the integer or floating-point Free List is empty. Also, the renamers indicate to each other if their Free List is empty i.e no free physical registers. This ensures that we do not rename the instruction bundle this cycle. The more efficient solution, though, is to stall only if the Free List of the type of instructions being renamed this cycle is empty. But since we do not stall fetching newer instructions and writing them into the Instruction Buffer, after we unstall we have buffer full of instructions waiting to be renamed. So having an Instruction Buffer is an advantage in such a case.

4.3 Interaction with retire stage

In the case of a recovery due to an exception or branch misprediction at the head of the Active List, the architectural or committed state is copied over from the respective Architectural Map Tables (AMT) to the RMTs. This is done to maintain precise state. Both the Free Lists are recovered by roll back of the head pointer to the tail pointer.

Before the AMT is updated with a new mapping for a logical register from the Active List (commit), it pushes the previous mapping onto the Free List (free). Chapter 7 talks in detail about the management of the Active List and AMTs.

The dispatch packet formed contains the physical mappings of the source and destination registers. The instructions pass through the Rename/Dispatch pipeline register before they are dispatched into the Active List, Load-Store Queue and respective Issue Queues.

CHAPTER 5

DISPATCH AND ISSUE

The dispatch stage marks the end of in-order execution. The dispatched instructions stay in the Issue stage till they are scheduled out-of-order, based on the availability of their source operands. There are separate issue queues for integer and floating-point instructions, which issue instructions to the execution lanes which are connected to them. The Dispatch stage sends the same instruction bundle to both the issue queues and the logic in the Issue stage writes the integer instructions to the integer issue queue and floating-point instructions to the floating-point issue queue.

Section 5.1 describes the Dispatch stage. The Issue stage is elaborated in Section 5.2.

5.1 Organization of Dispatch

After the Rename stage forms the dispatch bundle, the dispatch logic checks for space for these instructions in both the Issue Queues, the Active List and the Load-Store Queue if there is are loads/stores in the dispatch bundle. If any of the structures is full the Dispatch stage signals stages from Instruction Buffer till Rename to stall. The fetching and decoding of newer instructions can still continue.

If there is space for the renamed instructions, they are dispatched into both the issue queues and the active list. Only load and stores, if any, get dispatched into the load/store queues. An instruction's Active List index and the load-store queue index flow with the instruction through the rest of the pipeline.

Only stores get assigned a store queue entry number. But now, as we split a floating-point store into two parts and only mark the address part as a store, the logic to assign store queue entries was modified. This ensures that even when the value part of the store is marked as a non-store, it gets assigned the same store queue entry as that of its address part.

The decode logic assigns an instruction type to an instruction based on its opcode. This information is translated by the execution pipe scheduler to an execution lane number. The instruction arbitrates for its pre-assigned execution lane in the issue stage and hence simplifies the select logic in the issue queues.

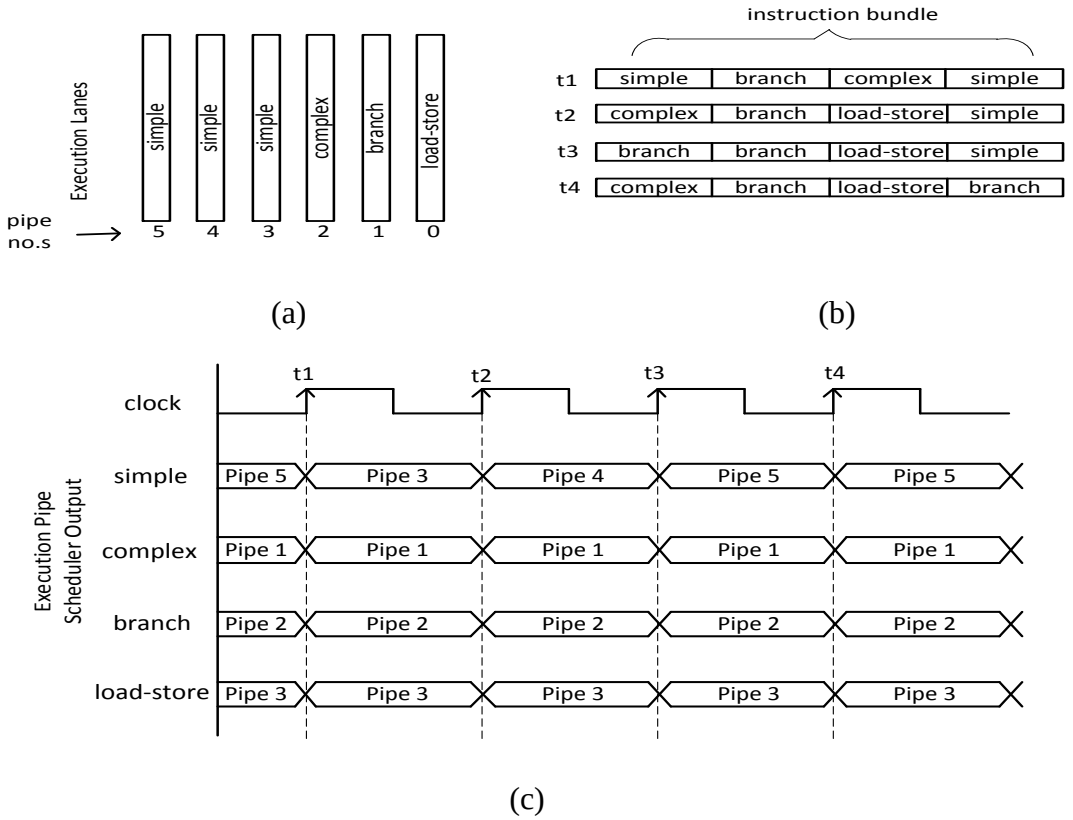


Figure 5.1: (a) Execution lanes (b) Instruction bundle input to the scheduler (c) Timing diagram showing output of the Execution-pipe Scheduler for a stream of instructions.

As shown the Figure 5.1, there are multiple lanes that execute simple instructions. The 2 simple instructions at time t1, will be assigned to lane number 3. The simple instruction at time t2 gets assigned lane 4 and at time t3 gets assigned lane 5. But since there are no simple instructions in the input bundle at time t4, the scheduler does not change its output for simple instructions. Instruction types which have only one lane to go to are always assigned the same lane number. Floating-point instructions are also assigned lane numbers based on the same principle.

5.2 Organization of Issue

The dispatched instructions get written into the issue queues and wait till their source operands are ready. The select logic selects one instruction per execution lane each cycle. The issued instructions broadcast their destination tags, if any, to wake-up their dependents in the issue queue. This enables us to do back-to-back execution of producer and consumer instructions with the help of bypasses from the Writeback stage.

We distinguish between integer and floating-point instructions based on the type of their source operands. Instructions with integer source operands get written into the integer issue queue and those with floating-point source operands are written into the floating-point issue queue. Since our mechanism to distinguish between integer and floating-point instructions is based on the type of source operands, instructions having two types of sources (*e.g. floating-point stores*) get split in the Decode stage into two or more micro-instructions. Each micro-instruction is such that it has only one type of source operand.

For instructions which have source(s) of one type and destination of the other, we issue them to the load-store lane because this lane has read and write ports to both the integer and floating-point PRFs. This lane is dedicated to perform any cross-communication between the two sides. Table 5.1 shows the classes of floating-point instructions which have different 'type' of source(s) and destination and hence are issued to the load-store lane.

Table 5.1: Instructions with different type of source and destination

Instruction class	Source type	Destination type
Floating-point load	Integer	Floating-point
Moves to floating-point	Integer	Floating-point
Moves from floating-point	Floating-point	Integer

All loads, integer or floating-point, do not speculatively wake-up their dependents. They do so only after they have executed. Since the moves from and to the floating-point unit have mixed source and destination types they are handled by the load-store lane. Thus, we force the move instructions to have the same latency as a load instruction and do not support pre-wake-up for the move instructions also.

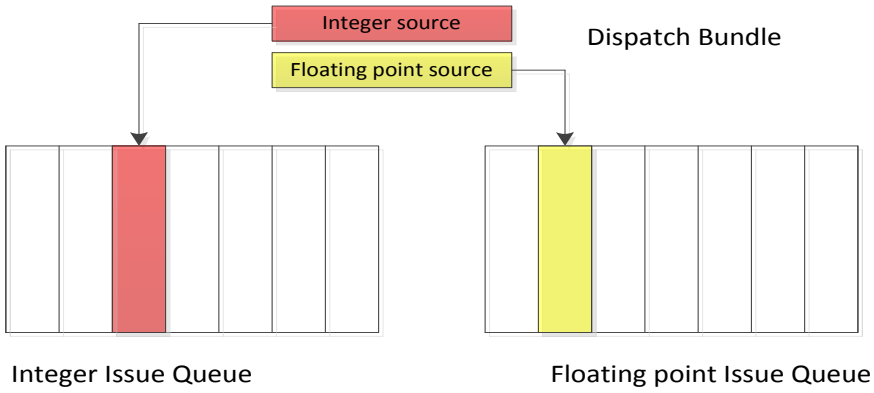


Figure 5.2: Instruction writing into the Issue Queues

As shown in Figure 5.2, the red instruction has integer sources hence it is written into the integer issue queue and the yellow instruction with floating-point sources updates the floating-point issue queue.

Each issue queue maintains a list of the free issue queue entries, called the issue queue free list. The dispatched instructions get written at free entries popped from the head of the issue queue free list. Instructions leaving the issue queue push their issue queue entry numbers at the tail. As dispatched instructions are a mix of integer and floating-point instructions, all the free list entries of either of the free lists, may or may not be used. Hence we collapse the dispatched instruction bundle to form two bundles, one containing floating-point instructions and the other containing integer instructions. The collapsed bundles of instructions are written into the respective issue queues. Figure 5.3 shows the dispatch bundle and the collapsed integer and floating-point bundles.

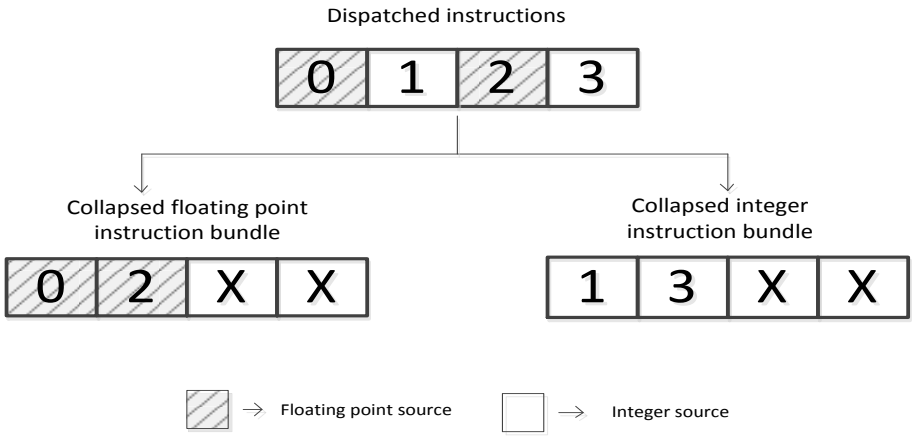


Figure 5.3: Collapsing of dispatch packet inside issue stage

An alternative to collapsing the issue queue packets is to alter the operation each of the free lists. That would involve pushing back the unused free entries at the tail, if any. Both the approaches used would affect cycle time; collapsing approach would use up more cycle time because of the added logic whereas the alteration to the free list would mean adding extra write ports which will indirectly impact cycle time.

After the instruction issues from the issue queue, it broadcasts its destination tag to wake-up its consumers. However, instructions which execute for multiple cycles, would wake-up dependents after a number of cycles equal to the execute latency of the instruction. Floating-point operations are known for their longer times of execution. However, as we have focused more on the micro-architectural design in the thesis, floating-point arithmetic is emulated and the floating-point instructions execute in one cycle. Hence, they wake-up their dependents after they leave the floating-point issue queue. The issued instructions go to their assigned execution lanes where they have dedicated resources to execute.

CHAPTER 6

EXECUTION LANES

All the major back-end stages are organized as a lane. Each lane comprises of the Register-read (R-R), Execute (EX) and Writeback (WB) stages. When an instruction issues, it has all the resources it needs to flow freely through the lane.

The lanes are classified based on the type of instructions they execute: (a) Branch, (b) Simple (single-cycle operations), (c) Complex (multiple-cycle operations), (d) Loads and Stores, (e) Floating-point.

To avoid a conflict on the write port to the integer Physical Register File accessed by the modified load-store lane, an extra write port was added. Hence, the integer side was changed to handle 'issue width plus one' bypasses, unlike before where there was one bypass per execution lane.

To support conditional moves, an extra bit was added to both the integer and floating-point Physical Register Files. Hence, all the lanes were modified to pass the extra bit with each value.

6.1 Branch Lane

The Fetch stage predicts the directions, taken or not-taken, of all branch instructions in the fetch bundle and continues fetching newer instructions from the predicted target PC. A branch instruction flows through the pipeline and executes in the Branch lane. The result, which is nothing but the target PC, is matched with the predicted target PC and in the case of

a mismatch the branch is marked as mispredicted in the Active List. When the branch reaches the head of the Active List it causes a flush of the pipeline and processor state is restored to the committed state. The fetch unit is now re-directed to the correct target address of the branch.

The Mips-ISA defines two floating-point branch instructions. Both instructions are conditional branches which depend on the FCCR. As branch instructions initiate recovery in program order, we could also have another branch lane on the floating-point side. But if ever checkpoint-based recovery was implemented, having another branch lane would create many complexities like two branches initiating recovery at the same time. Hence to keep the checkpoint recovery option open, we execute all control-flow instructions in one lane.

6.2 Simple and Complex lane

The simple lane handles integer instructions which execute in one cycle and the complex lane executes instructions with multi-cycle execute latency. The Simple ALU was extended to support integer conditional move instructions. Both micro-instructions of the conditional move are executed in a simple lane. There were no changes made to the complex lane.

6.2.1 Implementation of integer conditional moves

The integer conditional move (cmove) is a simple operation. It gets split into two micro-instructions, as described in Section 3.2.3, because the destination register is conditionally modified. All conditional moves are implemented using the ‘extra bit’ approach described in Section 3.2.3.

The alternative solution that surfaced was to split the conditional move as shown in Figure 6.1. The approach, however, would require that both the split parts are assigned the same physical destination register. This would imply changes to the rename stage and the decode stage.

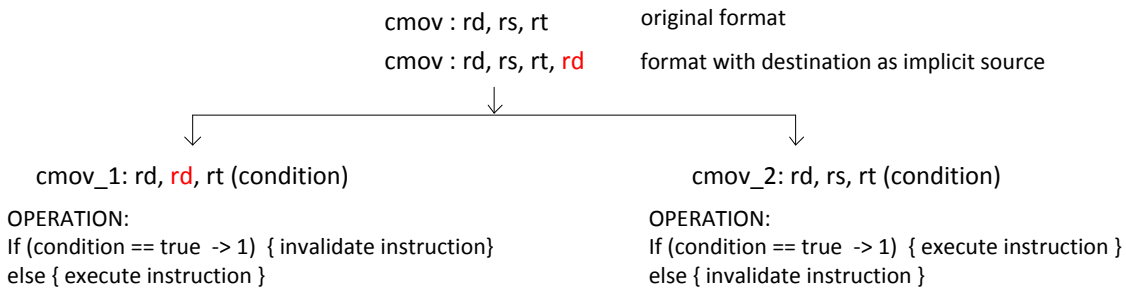


Figure 6.1: Alternative approach to split conditional moves

Cmov_2 would have to be marked, to indicate that it would always use the same physical destination register as cmov_1. If the two micro-instructions were to be in different rename bundles, it would mean saving the previous micro-instruction's destination in a pipeline register. On the other hand, this approach would save the extra PRF bit. Also, unlike now, the two micro-instructions would be able to pre-wake-up their dependent instructions. This solution appeared to be very specific in nature; hence the general 'extra bit' approach was adopted to implement the conditional moves. The floating-point conditional move implementation follows on similar lines except that it is handled in the floating-point lane.

6.3 Load-Store lane

Loads and stores get written into the load/store queue when they are dispatched. When a load/store instruction issues, from the issue queue to the load-store lane, it reads its source operands from the register file or from the bypasses. The address generation unit in the execute stage calculates the memory address for the instruction. After the address is computed, the load/store instruction is sent to the load-store unit where a load deposit its address in the load queue and searches the store queue and a store deposits its address and value in the store queue and searches the load queue. If there is a data cache miss, the load replays when it reaches the head of the load queue. A precise memory state is maintained by committing stores from the store queue in program order. If a load instruction follows a store to the same address, the memory state will be stale if the store has not committed yet. In such a case we forward the value of the store from the store queue to the load instruction (when the load searches the store queue).

The overall operation of floating-point loads and stores is the same as that of integer loads and stores. The difference is that a floating-point load loads a value from memory into a floating-point register and the floating-point store stores a value from a floating-point register to a memory address. The Mips-ISA also defines indexed floating-point load and store instructions when a 64-bit floating-point unit is used.

The active list signals the load-store queue to commit the head load/store instruction if the instruction committing is a load or store. There can be multiple store instructions committing in a cycle, but the logic in the store queue writes only one store at a time to memory.

The load queue and store queue must handle any type of load instruction or store instruction, respectively. Thus the load-store lane was extended to support floating-point loads and stores. The store queue data structure was made 64-bits wide. The R-R stage was expanded as the lane now has access to both the register files. In the EX stage, logic was added to form the floating-point packet going to the load-store unit.

As shown in Figure 6.2, the load-store lane is connected to both of the issue queues and hence gets up to two issued instructions per cycle. The yellow stars show the path of a floating-point load. After it gets issued from the integer issue queue it reads the source values from either the integer PRF or the bypasses in the operand bypass module. AGEN calculates the address and sends the information packet to the load-store unit which updates the load queue with the load's address. In parallel, the load reads the value from the data cache or from the store queue in case of a prior store to the same address. This is called store-load forwarding. Floating-point loads can load up to a double-word. The load-store unit sends the execution flags associated with the load instruction to the Writeback stage. The Writeback stage broadcasts the load's destination tag and value on the bypass. The yellow line shows the bypass used by floating-point load instructions. The Active List is updated with the execution flags.

The FabScalar template, previously, supported only word-aligned accesses to the data cache. If the interface to the data cache was to remain unchanged to allow only word accesses, the 'floating-point load double word' instruction would have to be split into two 'load word' instructions both having the same destination register specifier. This would lead to alterations in the rename circuitry. Hence, to avoid the same, the interface to the data

cache was modified to support double-word accesses. This decision also helped in committing double-word store values to the data cache.

If the load is marked to have previously violated, it is stalled and becomes a candidate for replay when it reaches the load queue head.

The floating-point store gets split into two micro-instructions. The first micro-instruction, deposits the address and the second micro-instruction deposits the value into the store queue. Since an integer store's address and value update the store queue together, the store can forward the value by just validating the address match with a younger load. But, unlike the integer store, the floating-point store value is not read along with the address. Hence, when the dependent load is executed, the 'Store v' micro-instruction may not have executed even if the 'Store @' micro-instruction has. Thus, the value is forwarded to the dependent load only if it is valid. The valid bit is set by the 'Store v' micro-instruction. The 'Store @' instruction only updates the address related structures. If the store value is invalid the dependent load is stalled and replays when it reaches the head of the load queue. This has to be done because the 'Store v' instruction, if executed later, does not have the address to mark the speculative load as violated.

The blue stars in Figure 6.2 show the path followed by the floating-point store instruction. The 'Store @' micro-instruction is issued from the integer issue queue. It reads its source operand(s) in the R-R stage followed by address computation in AGEN. The store packet is sent to the load-store unit where it updates the store queue and checks for any younger loads that may have violated.

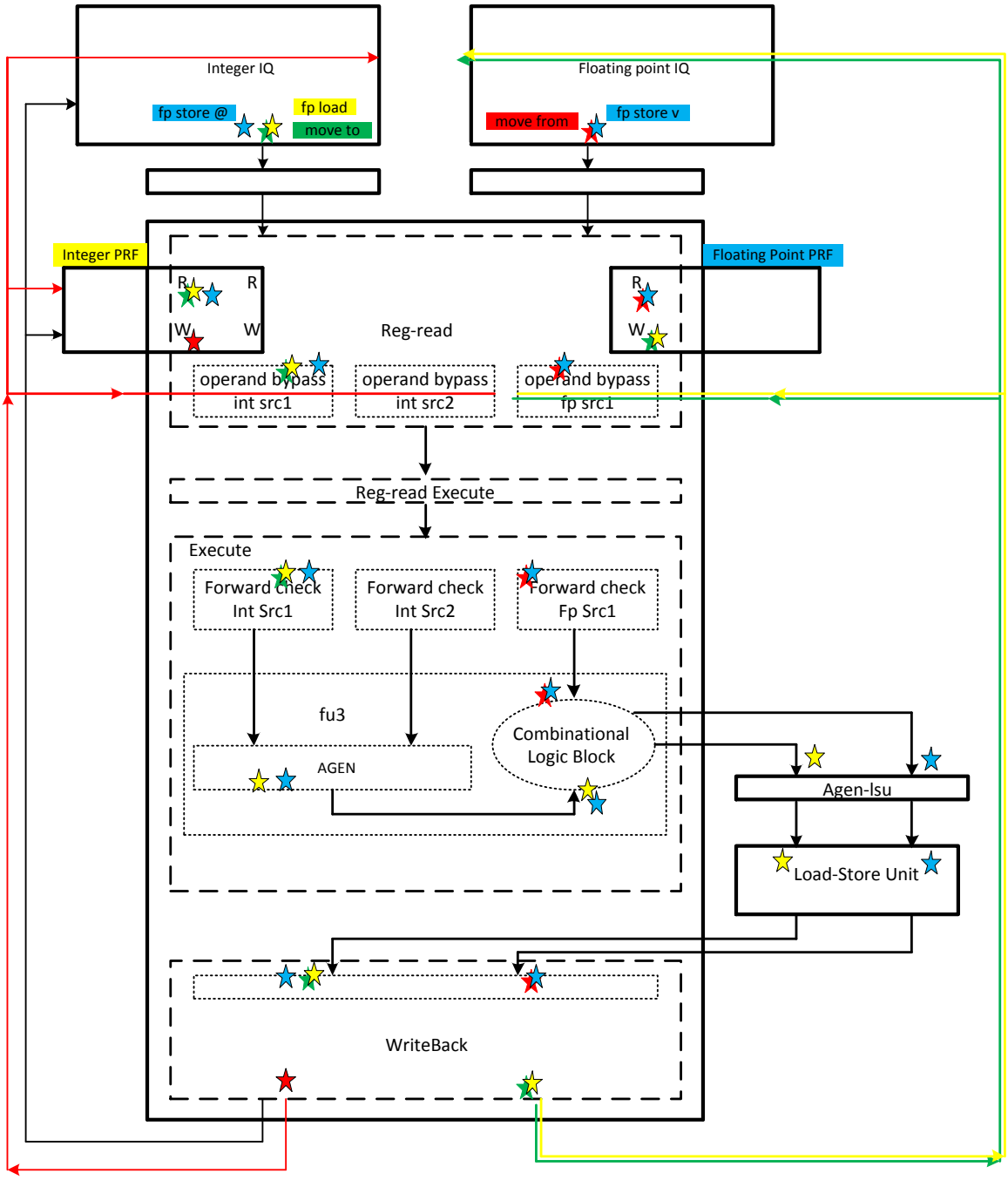


Figure 6.2: Modified load-store lane

A flag is added to distinguish between the ‘Store @’ and integer store. This prevents the ‘Store @’ from writing a value into the store queue, which it does not have.

The ‘Store v’ micro-instruction issues from the floating-point issue queue and reads the source value from the floating-point PRF or from the bypass. The value of the store is then sent to the load-store unit, which writes the value into the store queue. The store queue data structure has a dedicated write port for the floating-point store value part. This avoids a conflict for the store-value write port if an integer store and ‘Store v’ were issued at the same time, from the integer issue queue and floating-point issue queue respectively.

The Writeback stage for the load-store lane broadcasts values on up to three bypasses at a time. Table 6.1 shows the bypasses and which instruction types use them to broadcast their values.

The type of destination is passed down till the Writeback stage to distinguish between the integer and the floating-point bypasses. The integer bypass is valid if the destination is of integer type and floating-point bypass is valid if the destination is of floating-point type.

Table 6.1: Load-store lane bypass usage

Load-store lane bypasses	Used by
Integer bypass 1	Integer loads
Integer bypass 2	Move from floating-point side
Floating-point bypass	Floating-point loads, Moves to floating-point side

Since the load-store lane was modified to support floating-point loads and stores which communicate between the integer and the floating-point sides, we chose to use this lane to

also execute the moves to and from the floating-point side. The idea was to have only one lane with access to both of the register files. The other option was to have a dedicated lane to handle the cross-communication for moves. The conjecture is that, this would complicate the physical design of the back-end by virtue of having multiple lanes with access to both the register files. Hence the load-store lane was used.

The potential disadvantage of this approach is that, the move instructions will not be able to pre-wake-up dependents.

The ‘move to’ and ‘move from’ the floating-point side get issued from the integer issue queue and floating-point issue queue, respectively. Figure 6.2 shows the path of the ‘move to’ with green stars. After grabbing the source value from the integer register file or from the bypass, in the Execute stage, the instruction is sent to the load-store unit as an invalid instruction. This is done so that the instruction does not update the load/store structures and yet has the same latency as that of a load/store. To make sure that the move instruction, even though invalid for the load-store unit, writes its destination value, the instruction carriers an ‘isMove’ flag along with it. The writeback is done if the ‘isMove’ flag is set. The ‘move to’ broadcasts its tag and value on the floating-point bypass (shown by green lines) whereas the ‘move from’ broadcasts its tag and value on the integer bypass (shown by red lines), in the Writeback stage.

An ‘integer load’ instruction and a ‘move from’ issue from separate issue queues, which means it is possible that they issue in the same cycle. Thus, to avoid a conflict for the integer PRF write port and corresponding bypass, in case the instructions issue at the same time, we add an extra write port to the integer PRF and an extra bypass. On the other hand, the

floating-point PRF does not need the additional write port because the only instructions accessing that resource are the ‘floating-point load’ and ‘move to’ instructions. Since both of these instructions are issued from the integer issue queue, there would never be a conflict for that resource.

The alternative approach to adding an extra write port would have been to make sure that a ‘move from’ instruction does not issue in the same cycle as that of an ‘integer load’. But this would involve having a global select tree to arbitrate between the integer and floating-point issue queues’ select trees. The approach would save us an extra write port, but would slow down the selection process and impact cycle time more than adding the write port would.

6.4 Floating-point lane

A floating-point lane is connected only to the floating-point issue queue. All floating-point instructions, except floating-point loads, stores and moves, are executed in this lane. The lane’s structure is similar to the simple lane on the integer side, except that the datapath is 64-bits wide.

The floating-point arithmetic is emulated in software. Since the main idea of the thesis was to focus on microarchitectural design, floating-point arithmetic was not implemented. Also understanding that floating-point arithmetic implementation is, in itself, a tough task, the scope was kept restricted. For now, the instructions are implemented as C++ functions. The System Verilog language allows the use of C++ functions in hardware with the help of the Direct Programming Interface (DPI). The memory system in the FabScalar template is

emulated in a similar manner. Figure 6.3 shows a snapshot of some of the DPI declarations used in the code.

```
//*******import declarations **********************
import "DPI-C" function int     add_s           (int data1, int data2);
import "DPI-C" function int     sub_s           (int data1, int data2);
import "DPI-C" function int     mul_s           (int data1, int data2);
import "DPI-C" function int     div_s           (int data1, int data2);
import "DPI-C" function int     sqrt_s          (int data1);
import "DPI-C" function int     abs_s           (int data1);
```

```
`ADD_S:
begin
        result[31:0]            = add_s(data1[31:0],data2[31:0]);
        result[63:32]           = 32'h0;
        result[64]              = 1'h0;
        flags_o.executed        = 1'h1;
        flags_o.destValid       = 1'h1;
end

`SUB_S:
begin
        result[31:0]            = sub_s(data1[31:0],data2[31:0]);
        result[63:32]           = 32'h0;
        result[64]              = 1'h0;
        flags_o.executed        = 1'h1;
        flags_o.destValid       = 1'h1;
end
```

Figure 6.3: Direct programming interface (DPI) used in floating-point ALU

6.4.1 Implementation of Floating-point compare

Section 3.2.2 describes how and why the compare instructions get split into 3 micro-instructions. The first two micro-instruction are executed by the floating-point ALU. The third micro-instruction is a ‘move from’ instruction and is executed in the load-store lane.

For the compare instruction, FCCR is an implicit source as the instruction modifies only parts of it. The compare instruction could alternatively be split into only two micro-instructions, shown in Figure 6.4. The third source operand is FCCR.

```
compare_1  : FCCR_fp , fs, ft , FCCR_fp
move from  : FCCR_int, FCCR_fp
```

Figure 6.4: Alternative way to split floating-point compare

If this approach was adopted for the compare instruction's implementation, we would need to have a lane which has 3 read ports into the floating-point register file. The floating-point RMT would be required to have 3 read ports per instruction. The decode logic would have to change. The floating-point issue queue would need to have a wake-up CAM for the third source. The approach would thus affect many pipeline stages. It would have been a beneficial approach had there been many instructions which would require using these added resources. One such class is the multiply-add fusion instruction. But the Mips-ISA document clearly mentions that, the results and flags are as if separate floating-point multiply and add instructions were executed [8]. It means that we would add the resources only for one type of instruction. Hence, we opted for the splitting of the compare instruction into three micro-instructions.

CHAPTER 7

RETIRE

The Active List maintains all inflight instructions in their program order. After instructions complete out-of-order, their execution flags are updated in the Active List. The instructions commit in program order from the Active List.

The Architectural Map Table (AMT) in the retire stage maintains the committed physical register mappings of the logical registers. There are separate AMT's for the integer and floating-point sides.

7.1 Retire circuitry

The Active List is common for both integer and floating-point instructions. The Dispatch stage writes instructions in their program order at the Active List tail. This is the instruction's index into the Active List and flows with the instruction in the pipeline. After an instruction completes, it updates the Active List with its execution status. Instructions retire from the Active List in their program order. If the Active List commits a load or store instruction, the load-store unit is signaled to commit the head load or store instruction from the load or store queue, respectively. The respective AMT, floating-point or integer is updated with the current physical mapping of an instruction's destination if the committing instruction has a valid destination register. The freed mapping from the AMT is pushed onto the respective free list.

The commit logic in the Retire stage checks for fission instructions and always commits them together. Thus, if the first micro-instruction is ready but not the second one, the logic makes sure that the first micro-instruction does not commit. A branch and its delay slot also retire as a fission instruction. If the instruction in the branch's delay slot is a fission instruction, the branch and all parts of the fission instruction are retired together.

The Active List sends the commit information containing the logical destination and its current physical mapping to the AMT after the instruction retires from the Active List. The floating-point AMT is updated if the retiring instruction's destination is of floating-point type and the integer AMT is updated if the destination is of integer type.

The AMT circuitry was replicated. The size of the AMT is equal to the size of the respective RMT and hence, that is the only difference between the integer and floating-point AMTs.

7.2 Interface with the testbench

The instructions retiring from the Active List are matched with the functional simulator to verify functional correctness. Since the Active List retires instructions, it forms the interface with the testbench. Currently, the checker only checks destination values (if any) and program counters of the retiring instructions.

Fission instructions get split into two or three micro-instructions and get dispatched into the Active List at the time of dispatch. But as the notion of splitting is hidden from the functional simulator, only one out of the multiple micro-instructions is checked against the functional simulator's instruction.

Micro-instructions that should not be checked are marked as such in the Decode stage. This information is maintained in the Retire stage in a similar way as the other Active List information is maintained. When an instruction retires, the checker only checks it if the retired instruction is not marked as per above.

All the code that interfaces with the testbench, anywhere in the pipeline, is used only for simulation purposes. Preprocessor macros surround the code and hence it gets stripped away and will not get synthesized.

CHAPTER 8

RESULTS

This chapter presents the results and performance analysis for the augmented superset core. We measure the performance in terms of the instructions per cycle (IPC). We also present instruction statistics to show the ability of the benchmarks used to stress floating-point operation.

8.1 Simulation setup

We use the Cadence® NC-Sim simulation environment. We verify the functional correctness of the Verilog by matching its results with that of the functional simulator. The functional simulator is a C++ model implementing the MIPS 32 Release 2 ISA. It is not a cycle-accurate model. Figure 8.1 shows the diagrammatic representation of the simulation setup.

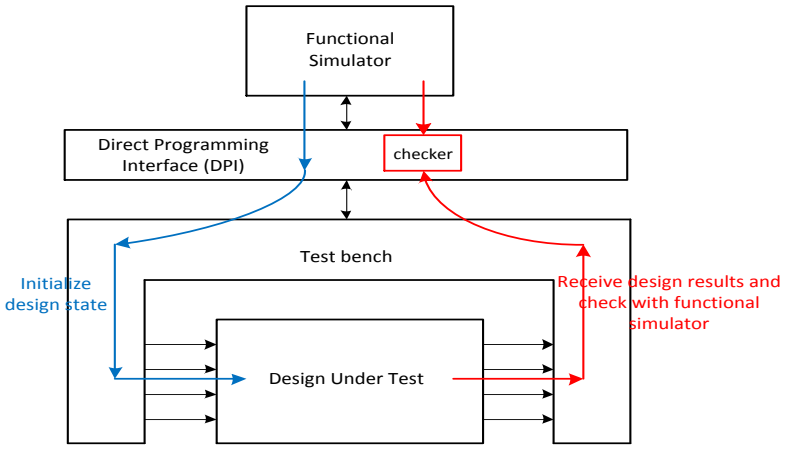


Figure 8.1: Block diagram showing the simulation setup

When an instruction retires from the Active List head, the testbench calls a checker function using the DPI. The testbench passes the result values of the retired instruction as arguments to the checker function. The checker then checks the result values with the functional simulator values. It returns a success if the results match. The simulation is exited if there is a mismatch in the program counters or the result values between the design and the functional simulator.

We have tested the design with micro-benchmarks which stress floating-point computations. We have also tested the design with one SPEC CPU2006 floating-point benchmark to measure the performance of the implemented design. Some of the limitations of the functional simulator prevented further testing with other SPEC benchmarks.

8.2 Results and analysis

We present the results obtained by running 10 million dynamic instructions of a SPEC floating-point benchmark - 470.lbm. We vary one structure size at a time to find the inflection point for each of the varied resources. We obtain two optimal resource configurations, for two, different floating-point issue widths. First, we vary the issue queue size, keeping the Active List, Physical Register File (PRF) and Load/Store queue sizes constant and large and obtain the issue queue size after which any increase in size yields diminishing returns. Second, we vary the load/store queue size, with the new issue queue size and unchanged Active List/PRF size. Finally, we vary the Active List/PRF size, with the inflection-point sizes for the issue queue and load/store queue. Thus, by varying one structure size at a time, we obtain the best possible window configuration to suite the benchmark.

The plot in Figure 8.2 shows the IPC variation with changing issue queue size. We vary the issue queue size while keeping the load-store queue and the Active List/ PRF constant. The sizes of the load-store queue and Active List/ RPF are kept large, so as to find the largest issue queue before getting diminishing returns.

With one floating-point lane, an issue queue size > 16 causes a dip in the IPC. This anomaly can be attributed to the fact that as window size increases, the number of speculative loads increases. This increases the chances of load violations. As a load violation causes a flush of the entire pipeline, it has a very bad impact on IPC.

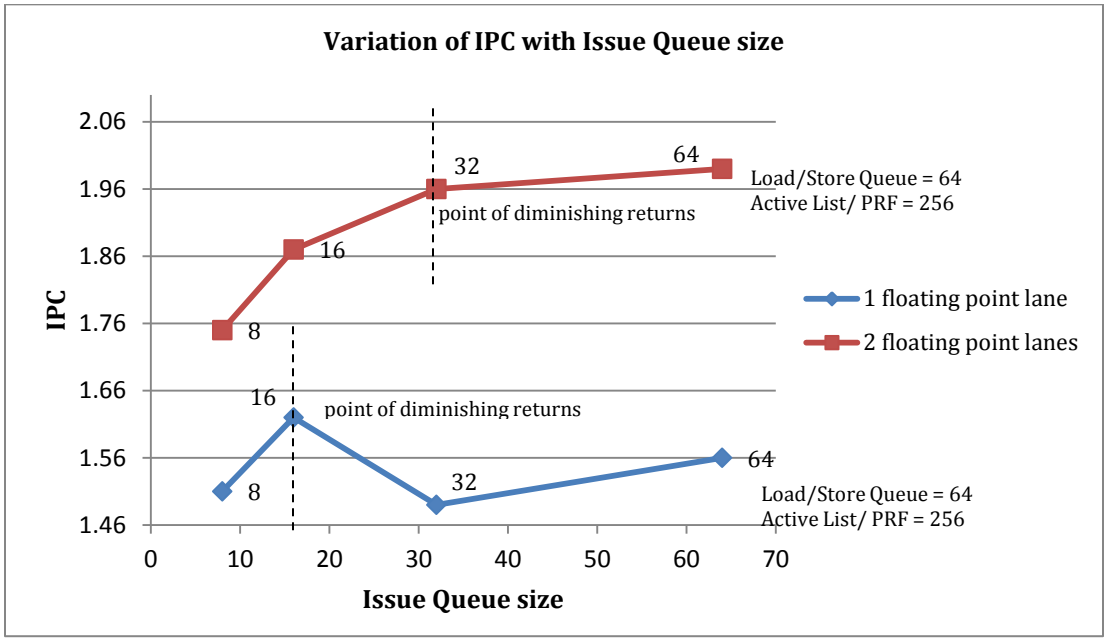


Figure 8.2: Graph showing variation of IPC with Issue queue size

Figure 8.3 shows the number of load violations and branch mispredictions with varying issue queue size. It clearly shows that when the issue queue size is increased from 16 to 32 there is a very large increase in the number of load violations.

The issue queue's select policy is not oldest-first. This increases the chance of younger loads issuing before their dependent stores. Moreover, a larger issue queue increases the chance even further. As we shall see, adding a second floating-point lane helps drain the larger issue queue faster, eliminating the anomaly.

When we increase the number of floating-point lanes, as expected, for each of the issue queue sizes we see an IPC increase. However when we increase the issue queue size after 32 we do not yield much benefit.

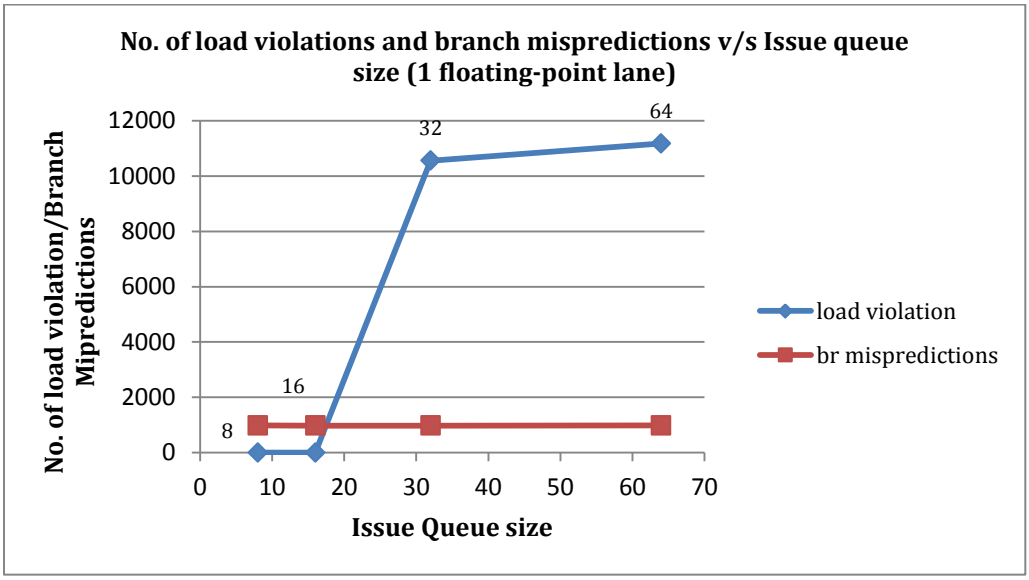


Figure 8.3: Graph showing no. of load violations and branch mispredictions with varying issue queue size

Figure 8.4 shows the IPC variation with the load/store queue size. For this plot, we choose the issue queue size obtained from Figure 8.2. The Active List size and the size of the PRF are kept constant and large. With 1 or 2 floating-point lanes, the IPC increases with load-store queue size until 32.

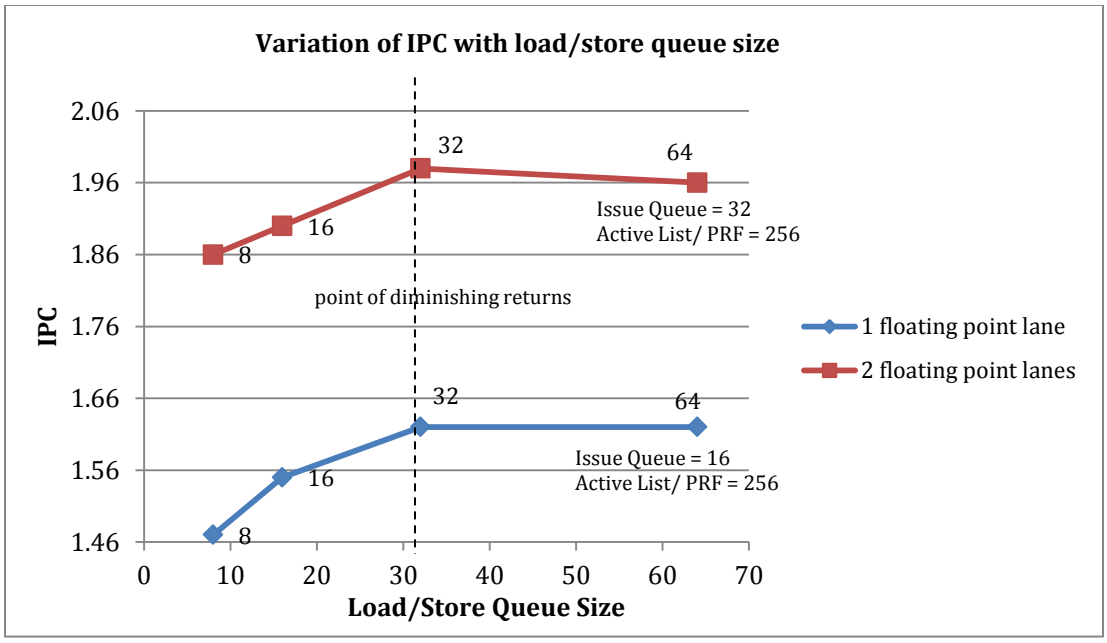


Figure 8.4: Graph showing variation of IPC with load/store queue size

We now plot the IPC variation by varying Active List/PRF size and keeping the issue queue size and load/store queue size constant. We chose the values obtained in Figure 8.2 and Figure 8.4 as the structure sizes. As can be seen in Figure 8.5, with 1 floating-point lane the IPC increases with the Active List/PRF size. With 2 floating-point lanes an increase in Active List/PRF size after 128 does not offer any performance boost.

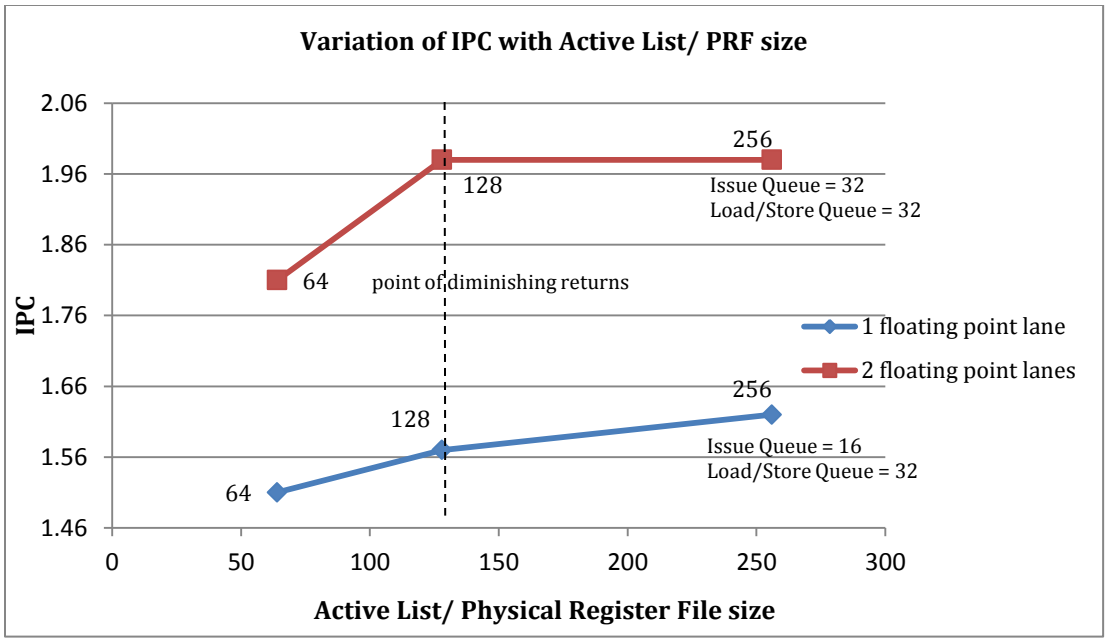


Figure 8.5: Graph showing variation of IPC with Active List/PRF size

Table 8.1 shows the inflections points for the window for the 470.lbm benchmark.

Table 8.1: Inflection points of various resources for 470.lbm benchmark

No. of floating-point lanes	Issue queue size	Load-store queue size	Active List/PRF size
1	16	32	128
2	32	32	128

Thus the results obtained for the 470.lbm benchmark show an increasing trend. The results, in a way, demonstrate that the decisions that went into making the design did not cause a degradation of the performance and hence the implemented techniques are good. Notice, that with 2 floating-point lanes IPC nearly reaches 2.0. For all experiments, the

processor's fetch/dispatch width is 2. Hence, we have nearly reached the machine's peak IPC.

We plot the workload characterization of the 470.lbm benchmark. The pie chart shown in Figure 8.6 shows the distribution of the integer and floating-point instructions over a total of 1 billion instructions. Thus we can see that the benchmark stresses many classes of floating-point operations and is a good platform to test and measure performance of the implementation.

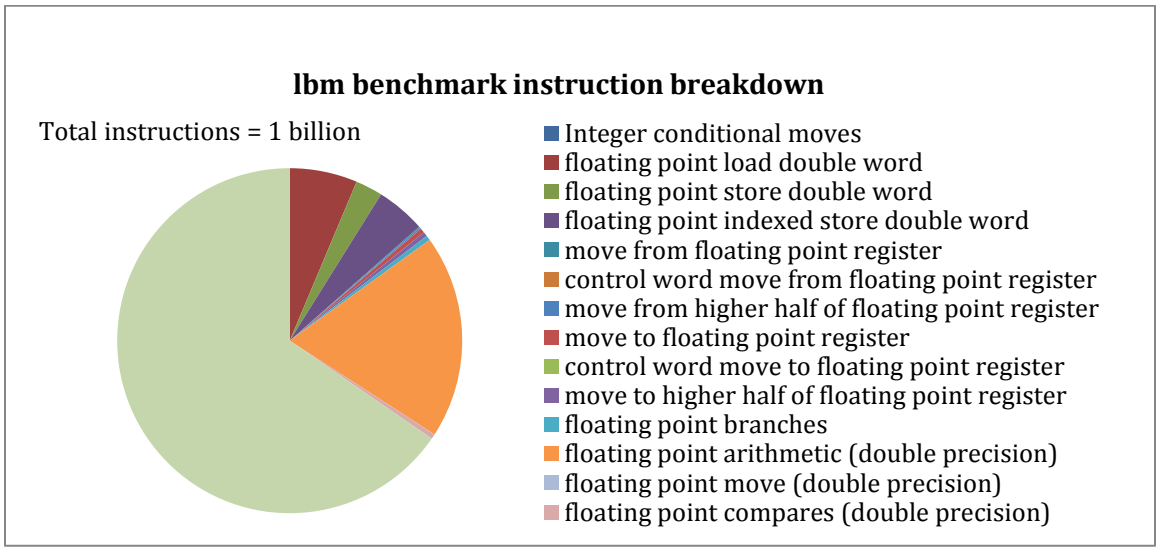


Figure 8.6: Pie chart showing the distribution of instructions for the 470.lbm benchmark

CHAPTER 9

SUMMARY AND FUTURE WORK

The thesis presented many interesting aspects of a floating-point ISA implementation. The work demonstrates the extensibility of the FabScalar toolset, which made it possible to extend it to have full blown floating-point support.

The processor front-end is modified to support floating-point instructions. Most notably, the instruction rename and issue components, present in the Rename and Issue stages, respectively, are replicated for the floating-point side and logic is added to correctly steer instructions to components based on their source and destination operand types. In the FabScalar toolset, the back-end is organized in the form of parallel execution lanes with each instruction having dedicated resources for it when it executes. The floating-point execution lane has a similar structure as that of the integer simple lane, only that it accesses different sets of resources.

The load-store lane is extended to handle floating-point loads and stores and thus establishes cross-interaction between the two sides (loads and stores use the integer side for addresses and the floating-point side for values). Since there are other instructions in the ISA that communicate between the integer and floating-point sides, we use the load-store lane to also execute these instructions. The thesis also considers alternative solutions for the same, to put into perspective the efficiency of our unified load-store lane approach.

As a commercial grade ISA, MIPS has some sophisticated instructions. The thesis describes the implementation of these instructions. Alternative implementation approaches are also discussed.

We verify the functional correctness of the Verilog code by checking with a C++ model of the ISA. We present results of the performance by testing on a few of the SPEC CPU 2006 benchmarks. Floating-point benchmarks tend to be rich in instruction-level parallelism. The quality of the superscalar floating-point implementation is evident in the fact that close to peak IPC of the tested configuration was achieved, and the required sizes of the issue queues, load/store queues, and physical register files were determined in the process.

There are several ways in which this work can be improved and extended. The microarchitectural implementation of the floating-point ISA could be tested further (for both correctness and IPC) by running more benchmarks and more superscalar configurations. It would also be good to augment the performance experimentation done in the thesis with synthesis results of the core to understand the cycle-time, area and power impact. It would also be interesting to explore the alternative implementations of some of the instructions, that were discussed but not implemented in the thesis.

REFERENCES

[1] N. K. Choudhary, S. V. Wadhavkar, T. A. Shah, H. Mayukh, J. Gandhi, B. H. Dwiel, S. Navada, H. H. Najaf-abadi, and E. Rotenberg, "FabScalar: Composing Synthesizable RTL Designs of Arbitrary Cores Within a Canonical Superscalar Template", Proceedings of the *38th Annual International Symposium on Computer Architecture. ACM, 2011, pp. 11*. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000067

[2] T. Nakabayashi, T. Sasaki, E. Rotenberg, K. Ohno and T. Kondo, "Research for Transporting Alpha ISA and Adopting Multi-processor to FabScalar". Proceedings of the *Symposium on Advanced Computing Systems and Infrastructures 2012 (SACSIS2012)*, pp. 374-381, May 2012. (*in Japanese*)

[3] Daniel Boley, Gene H. Golub, Samy Makar, Nirmal Saxena and Edward J. McCluskey, "Floating-point Fault Tolerance with Backward Error Assertions". Proceedings of the *Ieee transactions on computers, Vol. 44, No. 2. February 1995*. [Online]. Available:- http://crc.stanford.edu/crc_papers/Boley_paper.pdf

[4] OpenSPARC™ T1 Microarchitecture Specification. [Online]. Available:- http://users.ece.utexas.edu/~mcdermot/vlsi-2/OpenSPARCT1_Micro_Arch.pdf

[5] OpenSPARC™ T2 Core Microarchitecture Specification. [Online]. Available:- http://www.opensparc.net/pubs/t2/docs/OpenSPARCT2_Core_Micro_Arch.pdf

[6] N.J. Wang, J. Quek, T.M. Rafacz, and S.J. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline". Proceedings of the *International Conference on Dependable Systems and Networks. IEEE Computer Society, Jun 2004.* [Online]. Available:- http://www.crhc.illinois.edu/ACS/pub/dsn04.pdf

[7] MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS32® Architecture

[8] MIPS® Architecture For Programmer Volume II-A: The MIPS32® Instruction Set

[9] [Online]. Available:- http://steve.hollasch.net/cgindex/coding/ieeefloat.html

[10] Jack D. Mills,"Floating-point and integer condition compatibilty for conditional branches and conditional moves." *U.S. Patent 5,889,984, issued March 30 1999*. [Online]. Available:- http://www.freepatentsonline.com/5889984.pdf

[11] Karagada Ramarao Kishore, Xing Yu Jiang, Vidya Rajagopalan, Maria Ukanwa. "Conditional move instruction formed into one decoded instruction to be graduated and another decoded instruction to be invalidated", *U.S. Patent 8,078,846 B2, issued Dec 13, 2011*. [Online]. Available:- http://www.freepatentsonline.com/8078846.pdf

[12] SystemVerilog 3.1a Language Reference Manual Accellera's Extensions to Verilog®. [Online]. Available :- http://www.eda.org/sv/SystemVerilog_3.1a.pdf

[13] Cadence® NC-Verilog Simulator Help. [Online]. Available:- http://www.eet.bme.hu/~benedek/Asic_FPGA/Manuals/ncvlog.pdf

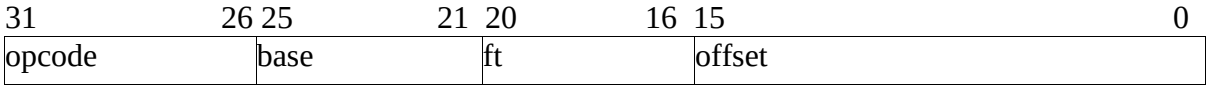
[14] DPI tutorial. [Online]. Available:- http://www.doulos.com/knowhow/sysverilog/tutorial/dpi/

APPENDICES

Appendix A

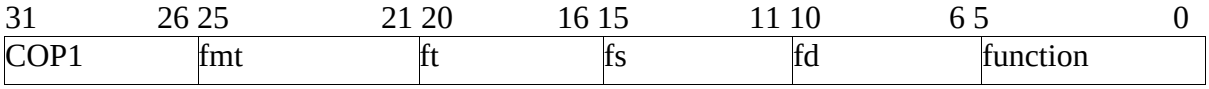
MIPS FPU INSTRUCTION FORMATS

I-Type (Immediate) FPU Instruction Format



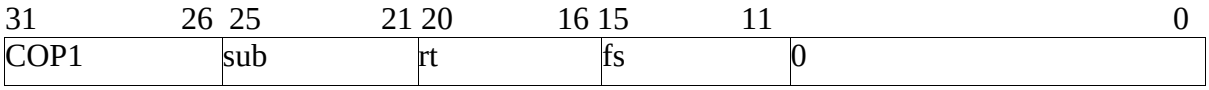
Immediate: Load/Store using register + offset addressing

R-Type (Register) FPU Instruction Format



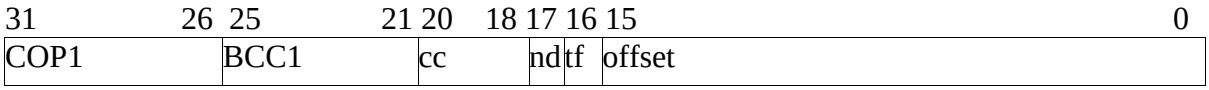
Register: Two-register and Three-register formatted arithmetic operations

Register-Immediate FPU Instruction Format



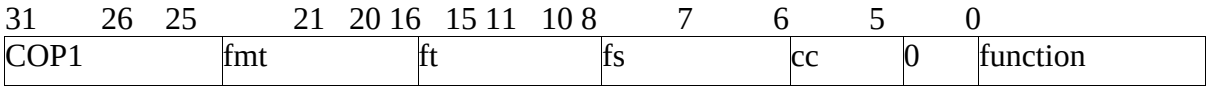
Register Immediate: Data transfer, CPU ↔ FPU register

Condition Code, Immediate FPU Instruction Format



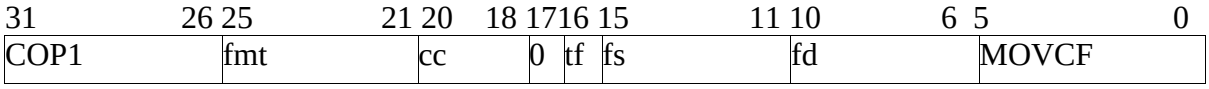
Condition Code, Immediate: Conditional branches on FPU cc using PC + offset

Formatted FPU Compare Instruction Format



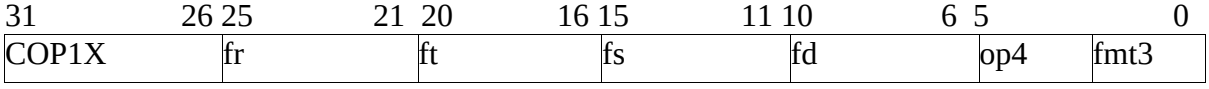
Register to Condition Code: Formatted FP compare

FP RegisterMove, Conditional Instruction Format



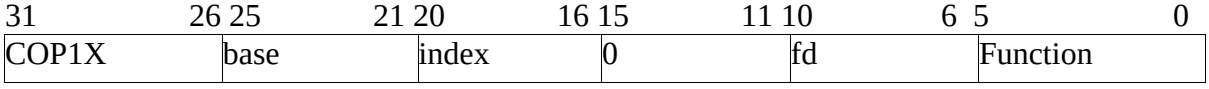
Condition Code, Register FP: FPU register move-conditional on FP, cc

Four-Register Formatted Arithmetic FPU Instruction Format



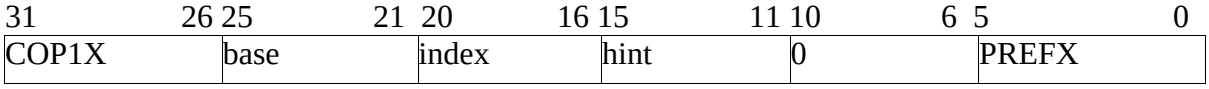
Register-4: Four-register formatted arithmetic operations

Register Index FPU Instruction Format



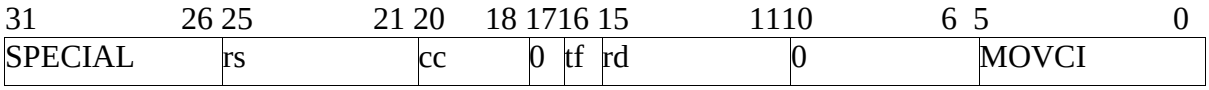
Register Index: Load and Store using register + register addressing

Register Index Hint FPU Instruction Format



Register Index Hint: Prefetch using register + register addressing

Condition Code, Register Integer FPU Instruction Format



Condition Code, Register Integer: CPU register move-conditional on FP, cc