DESIGN AND IMPLEMENTATION OF OUT-OF-ORDER SUPERSCALAR PROCESSOR

A PROJECT REPORT

submitted by

SIREESH N

in partial fulfillment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY (DUAL DEGREE)
in
ELECTRICAL ENGINEERING

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS
MAY 2015

CERTIFICATE

This is to certify that the project entitled "**Design and implementation of out-of-order superscalar processor**", submitted by **Sireesh N**, in partial fulfillment of the requirements for the award of the degree of MASTER OF TECHNOLOGY (Dual Degree) in Electrical Engineering, at Indian Institute of Technology, Madras, is a record of bonafide work carried out by him during the academic year **2014-2015**.

Prof. V. Kamakoti
Designation and Guide
Department of Computer Science and Engineering
Indian Institute of Technology, Madras
Chennai 600 036

Date: 12th May, 2015

ACKNOWLEDGEMENT

I owe my sincere gratitude to all those people who are responsible for the successful completion of this project and because of whom my graduate experience has been one that I would cherish forever. I would like to express my deepest gratitude to my guide, ***Prof. V. Kamakoti*** for his valuable guidance, encouragement and advice. His immense motivation helped me in making firm commitment towards my project work.

My special thanks to ***Mr. G.S. Madhusudan*** for his encouragement and motivation throughout the project. His valuable suggestions and constructive feedback were very helpful in moving ahead in my project work. I would like thank Rahul, Neel, Santosh, Ruthvik and other project-mates at RISE lab for their valuable inputs and discussions. I am also very grateful for my friends for their technical inputs and making my life at IITM an exhilarating experience. I would like to thank my family for their valuable support and encouragement which made my life wonderful.

Sireesh N

ABSTRACT

Most of the computer architecture research in academia is centered on using architectural simulators to evaluate and benchmark new techniques. This sometimes does not give an accurate picture of performance as throughput is dependent clock frequency too, not just on IPC. Only possible way to know about the impact of any architectural technique on clock speed is to actually implementing it in the hardware. This project aims at design and implementation of an out of order superscalar processor aimed at industrial control / general purpose applications running at 200 MHz – 1 GHz which can also be used as a generic out-of-order platform for evaluating new techniques. It is implemented in Bluespec systemverilog, a high level functional hardware description language which offers very good parameterization capability and fast development time so that it will be easy to add new hardware features. This processor is a part of open source processor cores development (named as I-class processor as it is aimed at industrial applications) at RISE Lab, IIT Madras. It is based on RISCV instruction set architecture which is an open source ISA.

Following features are implemented: a) Fetch width is 2 b) Renaming using merged register file c) Parametrized pipeline and Issue Queue d) Has two ALUs, one MUL/DIV unit, a Load Store and a branch unit (Issue width is 5) e) Has a speculative Load Store unit which issues both the loads and stores out of order. f) Bypass network to forward the results between functional units g) Branch predictor. h) Single cycle mispredict recovery mechanism.

Keywords*: I-Class, Superscalar processor, out-of-order, speculative load-store unit, merged register file renamer, tournament branch prediction*

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Computer has become the essential part of human lives now a days. Microprocessor is the central component of any computer. Since its advent, microprocessor throughput has been increasing continuously. It's either because of technology scaling or the advancement of architectural techniques which exploit the workload structure. Transistor density on a chip is approximately doubled every two years due to technology scaling, this is often referred to as Moore's law. With every new generation process node, the transistors get smaller, faster and more power efficient. This results in increase of processor clock speed without increasing power or area.

On the other hand, processors adapt to new workloads which evolve over time. For example, many new architectural features have been added to current day processors because of the tremendous increase in multimedia applications in recent years.

1.1 MICROARCHITECTURE CLASSIFICATIONS

Processor microarchitectures can be classified in different orthogonal dimensions. The most common classifications are the following.

1.1.1 Pipelined/Non-Pipelined Processors

A pipelined processor splits the instruction execution into different phases and allow multiple instructions to be processed in different phases simultaneously. Pipelining increases instruction level parallelism (ILP) and simple to implement. All the practical processors now a days are pipelined.

1.1.2 In-order/Out-of-order Processors

An in-order processor processes the instructions in the order they appear in binary code. Whereas in an out-of-order processor, instructions need not be executed in their order in the binary code. Instructions which do not depend on the previous ones can be sent for execution

earlier. It increases the instruction level parallelism (ILP) by allowing the hardware to choose the instructions to processes in a given cycle. But, it adds a large amount of extra logic and higher power consumption compared to the in-order ones.

1.1.3 Scalar/Superscalar Processors

A scalar processor processes one instruction in each stage of the pipeline in each cycle. So, the maximum achievable instructions per cycle (IPC) from a scalar processor is 1. Whereas, a superscalar processor can process more than one instruction in each stage of the pipeline in one cycle. This allows the maximum possible throughput to be greater than 1 instruction per cycle.

A very large instruction word (VLIW) processor is one type of superscalar processor which processes multiple instructions in each stage of the pipeline. Following features make a superscalar processor VLIW.

- It executes instruction in-order.
- Instructions which have to be executed in parallel are indicated in binary code.
- Execution latencies are exposed to the programmer and are the part of ISA itself. Programmer should maintain the proper distance between the instructions to ensure correct execution.

These constraints makes the hardware simple as it doesn't have to check for operand availability at run time and doesn't have to choose which instructions to be sent to execution. But, the programmer should know the hardware of VLIW processor (latencies of functional units) to write the code. So, a code written for one processor may not be compatible with the other.

1.1.4 Vector processors

A vector processor includes significant number of instruction in its ISA to operate on vectors. Traditionally, vector processors operate on vectors of very large lengths. Many of the recent processors also included these kind of instructions as the part of their ISAs but with small vector lengths of 4 to 8. These instruction are commonly known as SIMD instructions (examples are Intel AVX and ARM Neon instructions). According to the definition, many processors now a days can be called as vector processors.

1.1.5 Multicore Processors

A core is a unit which processes a sequence of instructions (usually called as thread). A processor can have one or more cores. Many of the current day processors are in fact multicore. A multicore processor can process multiple threads simultaneously using different hardware resources for each thread and includes support for synchronization and communication among the threads under the control of programmer. This support typically includes some type of interconnect between the cores and some primitives to communicate through this interconnect.

1.1.6 Multithreaded processors

A multithreaded processor is the one which has the support to execute multiple threads on the same core. Both the multicore and multithreaded processors can execute multiple threads simultaneously. The key difference is that a multicore processor uses entirely different hardware resources to execute different threads whereas most of the hardware resources are shared among different threads in a multithreaded processor.

1.2 PROCESSOR CLASSIFICATION BASED ON MARKET SEGMENTS

Processors characteristics also depend on the market segments they are intended to work.

- *Servers*: Refers to high performance systems used in data centers, which are shared by many users and have large number of processors. Parameters of interest for these processors are processing power and power dissipation.
- *Desktops*: Refers to home computers, normally used by a single user. Important parameters are processing power and noise of cooling solution.
- *Mobiles*: Refers to processors in notebooks, tablets and smartphones whose main feature is mobility. Power consumption is the parameter of most importance in these class of processors because of limited battery power. But, processing power is important too.
- *Embedded*: Refers to the processors in embedded systems. The processing power and power consumption requirements vary depending upon the nature of job the given system is performing. Another important parameter is the cost as many embedded

systems require less computing power, hence the processor's cost should be minimal so as not to affect the overall product cost.

All of the present day high performance processors have all the features mentioned in section 1.1. The processor we aim to implement is pipelined, out-of-order and superscalar, remaining features can be added in the future. It is named as I-Class processor which is a part of open source cores project at RISE Lab, IIT Madras.

CHAPTER 2

BACKGROUND

This chapter explains the basic material required to understand the architecture of modern day processors.

The function of any processor is to take instructions and data from memory, perform some operation and store it back into the memory. As discussed in introduction, its throughput/performance is rapidly increasing since its invention. Throughput of a processor is defined as the number of instructions it can execute per unit time.

$$Throughput = IPC \times frequency$$

Where IPC stands for 'instructions per cycle' executed. Throughput of a processor hence can be increased by either increasing the frequency or IPC. Another important parameter in measuring the performance of the processor is power consumption. As we will see, the above three terms are actually interdependent, increasing one term affects the other two. The only independent contributors for frequency and power consumption are advancements in semiconductor manufacturing technology and gate level implementation of hardware.

In this chapter, we start with basics of instruction set architecture and then discuss various techniques to increase IPC.

2.1 INSTRUCTION SET ARCHITECTURE

Instruction set refers to the set of operations processor can perform on given data and their formatting. It is the portion the computer visible to the programmer. Every instruction has two parts, one to describe operands and another is the type of operation performed on the given operands. Following are the main features of instruction set architectures:

- *Class of ISA:* Most of the today's ISA are general purpose register architectures where the operands are either register or memory locations. Again, there are two popular version of this class:

- Register-memory ISAs in which many instructions access memory as a part of instructions. These type of ISAs are referred to as CISC (complex instruction set computing) architectures. Ex: 80x86
- Load-store ISA in which there are separate instructions to access memory. This kind of architectures are also called RISC (Reduced instruction set computing) architectures. Ex: ARM, MIPS, RISCV.

- *Memory addressing:* All the present day architectures use byte addressing to access memory. Some architectures like ARM and RISCV requires memory addresses to be aligned i.e., an access to object of size s at memory address A requires $A\%s = 0$. Whereas 80x86 doesn't require memory alignment but non-aligned access is normally slower.
- *Addressing modes:* Refers to the way the addresses to memory objects are specified. RISC ISAs like ARM and RISCV use register (when an operand is in register), immediate (operand is an immediate field in instruction) and displacement (operand is in a memory location defined by register value + immediate offset) addressing modes whereas 80x86 has extra addressing modes.
- *Types and sizes of operands:* Most of the ISAs supports operand sizes of byte (8 bits), half-word (16 bits), word (32 bits) and double-word (64 bits).
- *Operations:* There are mainly three types of operations that can be performed by instructions a) ALU/FPU operations b) data transfer operations and c) control operations.
- *Control flow instructions:* All the ISAs implement control flow instructions which include conditional branches, unconditional jumps, procedural calls and returns. Also, all of the ISAs use PC (program counter) relative addressing where the jump address is specified by PC added to a signed offset value.
- *ISA encoding:* Refers to how an instruction is represented in bits. Normally, RISC architectures use fixed length encoding which simplifies decode logic. CISC architectures normally use variable length instruction encoding.

Any instruction is verbally represented in the form in this thesis

OPERATION	DEST	OP1	OP2/IMM

Where ‘OPERATION’ means the type of operation needs to be performed on operands, ‘DEST’ stands for destination register, ‘OP1’ for operand 1 and ‘OP2/IMM’ for operand 2 or the immediate value. For example ADD R3 R2 R1 means the instruction operation is ‘Add’, the source operands are the values in registers R2 and R1 and the destination register is R3.

As we are implementing a processor based on RISCV ISA which belongs to RISC architecture, we focus on the architectural aspects related to this ISA.

2.2 PIPELINING BASICS

Pipelining is a technique used by all the processors to increase processor frequency where multiple instructions are overlapped in execution. It takes advantage of parallelism that exists among the different actions needed during executing an instruction. More are the stages of pipeline, lesser will be the average execution time of instruction. Following are the typical pipeline stages of a processor

- *Instruction fetch:* Fetches instruction from instruction memory by sending PC.
- *Instruction decode/operand fetch:* Decodes instruction to find the type of operation to be performed and extract immediate operands. If the register operands stay in fixed position for every instruction, then the register access can be done in parallel in the same cycle.
- *Execute/Effective address calculation:* Perform the ALU/FPU operation on the operands in the case of arithmetic instructions or calculate effective address in the case of memory access instructions.
- *Memory access:* Access data memory in the case of load/store instructions.
- *Write-back:* Write back the calculated result/accessed memory value on to the destination register.

Figure 2.1 shows the implementation details of five stage pipeline described above. Table 2.1 shows the overlapping of instructions in the pipeline. Column at clock cycle 5 shows the overlapping of 5 instructions in different stages of their execution.

Table 2.1: Pipeline flow diagram for five stage pipeline

Instr. No.	Clock cycles								
	1	2	3	4	5	6	7	8	9
1	IF	ID	EX	MEM	WB				
2		IF	ID	EX	MEM	WB			
3			IF	ID	EX	MEM	WB		
4				IF	ID	EX	MEM		
5					IF	ID	EX	MEM	WB

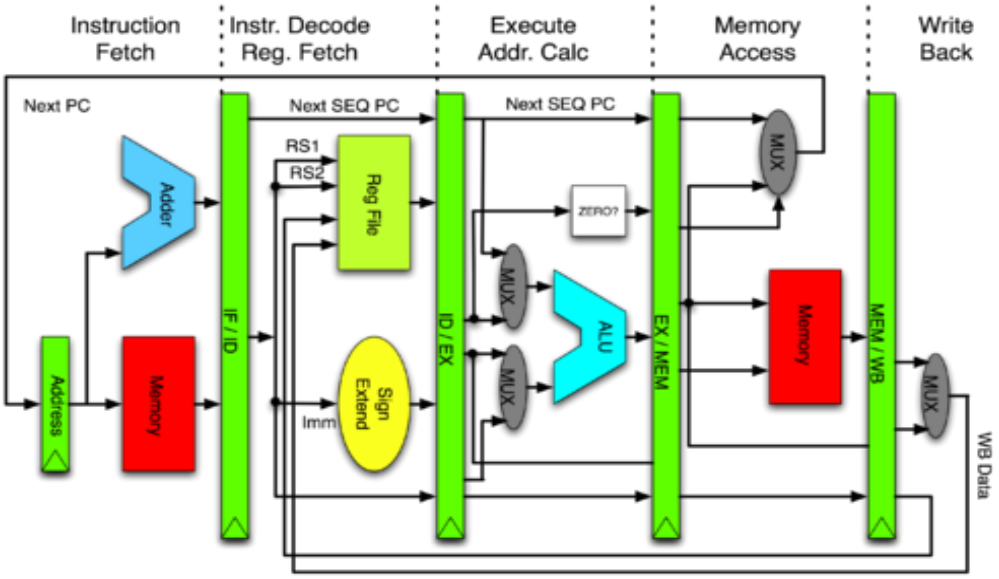

Figure 2.1: Five stage pipeline implementation (source: [wikipedia)]

If there are no stalls in the pipeline, the number of cycles taken for executing N instructions for an M cycle pipeline is M-1+N. For a large N, average execution time of each instruction is almost 1. As only a part of an instruction execution takes place every cycle, the processor now can run at higher frequency. This results in the great reduction of average execution time of an instruction in pipelined processors.

We cannot keep increasing the pipeline stages to get better performance because of the following reasons.

- *Pipeline overhead:* Pipeline overhead is caused due to the delay in inter-stage buffers (setup time of the registers) and clock skew (maximum delay between the clock arrivals at two registers). As we increase the number of pipeline stages, pipeline overhead delay becomes more dominant and limits the maximum frequency that can be achieved.
- *Pipeline latency:* As the frequency is always decided by the combinational path with maximum delay, no further pipelining in other parts the processor is useful.
- *Pipeline stalls:* Pipeline stalls/flushes in deep pipelines incur huge penalties in the case of mispredictions and dependencies.

A pipeline stall stops the instruction pipeline because of hazards which are discussed in the next section.

2.3 PIPELINE HAZARDS

A hazard is a situation where an instruction is prevented from execution in its designated clock cycle. There are three types of hazards:

1. *Structural hazards:* A structural hazard arises due to hardware resource conflicts i.e., when two instructions are trying to access same hardware. For example, it can occur in a processor with a multiple cycle latency functional unit which is not fully pipelined. When two consecutive instructions use that functional unit, second instruction must wait till the execution of the first is completed. This results in a pipeline stall.
2. *Data hazards:* Pipelining allows multiple instructions to execute simultaneously by overlapping different stages of their execution. This leads to data hazards which arise due to difference in read/write access order to registers (operands) from their access order in sequential execution. For example, look at the following instruction sequence:
    i. DIV R3 R1 R2
    ii. ADD R5 R3 R1
    iii. SUB R6 R3 R2

If DIV instruction takes multiple cycles to give out the result, ADD instruction must wait in ID stage of the pipeline till the result from that instruction is available. This stops SUB instruction to stall in IF stage and no other instruction in the sequence can be fetched until DIV instruction result is written into the destination register.

3. *Control hazards:* Control hazards arise due to jump/branch instructions. There are two types of branch instructions a) Conditional and b) Unconditional. A conditional branch instructions normally compares two operands and jumps to a target PC value based on the comparison result. Unconditional branch instructions directly change PC value to some target value without checking for any condition. As the result of the comparison is not known till the EXE stage of the pipeline, simplest method to deal with branches is to stall the execution of next instruction till then.

As the result of hazards, number of cycles per instruction (CPI) now increases and given by

$$pipeline\ CPI = ideal\ CPI + \sum hazard\ stalls$$

Where '*ideal CPI*' refers to the maximum attainable CPI by an implementation and 'hazard stalls' is the sum of stalls due to all the three hazards discussed above.

All the modern day processors use different techniques to exploit the workload structure to decrease ideal CPI and hazard stalls. The qualities of workloads that can be made use of are classified into three types, they are:

a) *Instruction level parallelism:* Measure of how many instruction executions can be overlapped. Every application has some level of instruction level parallelism that can be exploited.

b) *Data level parallelism:* When a same operation is performed on large set of independent data, for example performing an operation like increasing intensity of all the pixels in an image. Other types of operations that can give rise to data level parallelism are the operations on vectors, matrices and arrays.

c) *Thread level parallelism:* When there are independent threads running simultaneously. These types of applications are often found in the machines that have high workload, like web servers.

Now we discuss different techniques that actually exploit the above mentioned workload structures to improve IPC of the processors.

2.4 EXPLOITING INSTRUCTION LEVEL PARALLELISM

Key idea behind exploiting instruction level parallelism is determining how one instruction is dependent on another. If it is known, all the functional units can be kept busy by executing instructions (not necessarily in program order) whose operands are already available simultaneously. Even if the instructions are dependent and must execute sequentially, they can be partially overlapped. In both the cases, it is needed to determine whether an instruction is dependent on another one. Different types of dependences are:

i. *Data dependency:* An instruction *i* is data dependent on an instruction *j* if one of the following condition holds
    a. If result of instruction *j* is used by instruction *i*.
    b. If the result of instruction *j* is used by another instruction *k* whose result is used by instruction *i*.

    An instruction is said to be data dependent on another if there exists a chain of data dependencies between them. Instructions which are data dependent must execute sequentially (in the program order) and cannot execute simultaneously. These dependencies are also called true data dependencies.

ii. *Name dependency:* Two instructions are said to be name dependent if both of them use same memory location/register address (called a name), but there is no dataflow between them. Two types of name dependencies are:
    a. *Anti-dependency:* Anti-dependency is said to exist between two instructions *i* and *j* if instruction *i* writes a memory location/register location and instruction *j* reads it. Original program order must be preserved so that instruction *j* reads the correct value.
    b. *Output dependency:* An output dependency is said to exist between the instructions *i* and *j* if both of them use same destination register or memory location. Original program order must be preserved so that final value written into the destination register is correct.

Both the above dependencies are not true dependencies as no value is transferred between the instructions. So, they can be executed simultaneously or reordered if the names of the registers/memory locations are changed.

iii. *Control dependency:* It determines the ordering of an instruction *i* with respect to a branch instruction. An instruction that is control dependent on a branch instruction cannot be moved out of the branch as its execution is no longer controlled by the branch now. In the same way, an instruction before a branch should not be moved after the branch.

We must ensure that all the dependencies are maintained for correct execution of any program.

A data hazard exists between two instructions that are either data dependent or name dependent. Hazards are named by the ordering in the program that must be preserved by the pipeline. Data hazards are classified into following types.

Consider two instructions *i* and *j* in which *i* appears first in program order.

i. RAW – (read after write) When *j* tries to read the source before it is written by *i*, then it reads the wrong value. This hazard corresponds to true data dependency and is the most common type of hazard in pipeline.
ii. WAR – (write after read) When *j* writes the destination register before it is read by instruction *i.* This corresponds to anti-dependence. It occurs when the instructions are reordered in the pipeline.
iii. WAW – (write after write) When *j* tries to write the destination register before it is written by *i.* This corresponds to output dependence. It occurs in the pipelines in which more than one stage can write the register file.

2.4.1 Result Forwarding/bypassing:

As we have seen in the previous section, data hazards stall the pipeline till the result of the producer instruction (an instruction whose result is used by subsequent instructions) is written into the register file. Instead of waiting for this to happen, result calculated in the EXE or MEM stages of the pipeline can be passed to ID stage of consumer instruction in the same

cycle. This is called as result forwarding/bypassing. Result bypassing in a simple five stage pipeline is shown in the figure 2.2.

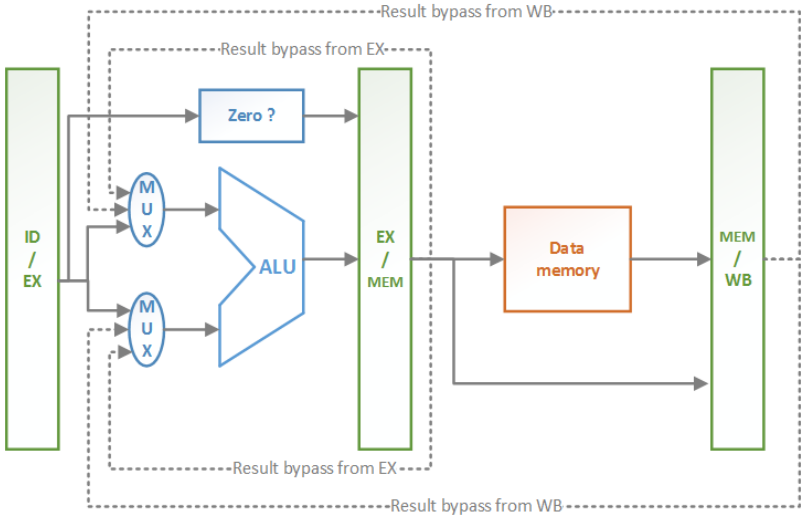

Figure 2.2: Result bypassing

Advantage: Reduces potential data hazards.

2.4.2 Simple branch scheduling

The simplest method to execute conditional branches (control hazards) is to stall the pipeline till the branch outcome is known. As the branch instructions are quite frequent in all kinds of workloads (with the share of around 10%), reducing the stalls due to control hazards is very important to get high performances. Following are the simple techniques to deal with branches:

i. Treat every branch as *not taken*: Fetch the subsequent instructions assuming that the branch is not taken, flush the wrongly fetched instruction in the pipeline if the branch result is *taken.*

ii. Treat every branch as *taken*: Fetch the instructions from the target address assuming that the branch is taken, flush the wrongly fetched instructions in the pipeline if the branch outcome is *not taken*.

iii. *Insert branch delay slots*: Branch delay slot is an extra instruction inserted by the compiler after every conditional branch instruction as shown in the pipeline flow diagram below. Job of the compiler is to make the instruction in branch delay slot useful.

Table 2.2: Branch delay slot in pipeline

Branch instruction (i)	IF	ID	EXE	MEM	WB		
Branch delay slot		IF	ID	EXE	MEM	WB	
Target address or (i+1)			IF	ID	EXE	MEM	WB

For example, the instruction sequence ADD R3 R2 R1; BNE R2 R1 #5 can be reordered by the compiler to BNE R2 R1 #5; ADD R3 R2 R1 in which case ADD instruction is executed in branch delay slot. This doesn't result in pipeline flush/stall because the new instruction after branch is fetched only after the branch outcome is known. But, inserting useful branch delay slot is not possible in all the cases.

Advantage: Reduces stalls due to control hazards.

2.4.3 Branch prediction

As the pipelines got deeper, necessity to predicting whether the branch is taken or not has grown high. General property of the workloads is that the branches are bimodally distributed i.e. a branch is highly biased towards either *taken* or *not taken*. This property of the workloads can be used to predict branches with high accuracy. Branch prediction can be classified into two types:

i. Static branch prediction: It is done during compile time of a program using the workload profile information from the previous runs. Profile based predictors have higher misprediction rates for integer instructions which typically have higher conditional branch frequency.
ii. Dynamic branch prediction: Dynamic branch prediction is done in hardware which is a table (memory) in which entries are indexed by the program counter of branch instructions. Contents of each entry is a state (prediction) which says whether branch

is taken or not. This state can be changed by the processor depending on the prediction result. Most of the processors use 2 bit prediction schemes which are shown to be optimum for different workloads. A two bit predictor is a state machine shown in the figure 2.3.

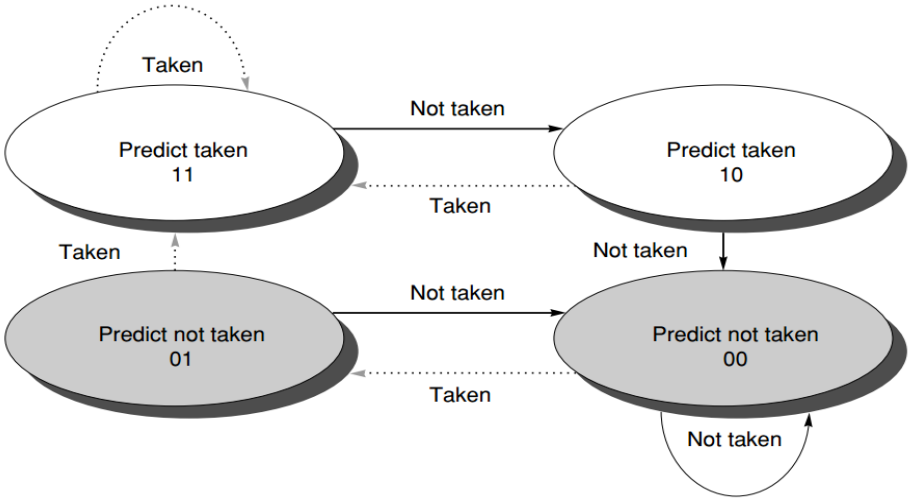

Figure 2.3: Two bit predictor state machine [1]

Advantage: reduces stalls due to control hazards.

2.4.4 Basic compiler pipeline scheduling

All the stalls due to RAW hazards are because of the data dependencies between them. To keep the pipeline full, parallelism among the instructions must be exploited and instructions must be ordered by compiler in such a way that stalls are lesser. Such stalls in the pipeline can be avoided if the producer-consumer instructions are separated by the number of cycles equal to the latency of producer instruction.

Advantage: Reduces stalls due to data hazards

2.4.5 Dynamic scheduling

A simple statically scheduled pipeline fetches and issues instructions unless there is a dependency between the instructions already in the pipeline and the newly fetched instruction.

But, if there are data dependencies that cannot be hidden by compiler or forwarding/bypassing, the pipeline will be stalled until they are resolved.

In this section, a technique used in all the modern day high performance processors called dynamic scheduling is discussed. In dynamic scheduling, hardware reorders the instructions and sends them to execution to reduce the stalls while maintaining data-flow and exception behavior. It has several advantages over the compile time scheduling, they are

a. Now, the code need not be recompiled for different processor architectures. This advantage is significant as much of the software now a days is made by third parties and is distributed in binary form.
b. Enables handling the cases when the dependencies are unknown at compile time.
c. Enable the processor to tolerate unpredictable delays, like the cache misses, by allowing other independent instructions to execute while waiting for the cache response.

A major limitation of simple pipelining is that the instructions are issued and executed in the program order. That is the major source of stalls in pipeline due to hazards. If one instruction in the pipeline is stalled, no other instructions can proceed. In the processors with multiple functional units to reduce structural hazard stalls, data hazards cause the functional unit to stay idle till the hazards are resolved. For example, in the code below

```
DIV R4 R1 R2
ADD R5 R4 R3
SUB R6 R2 R1
```

'SUB' instruction will be stalled even though it has both the operands available until the high latency DIV is done execution. This limitation can be eliminated if the instructions are allowed to execute out-of-order. In the five stage pipeline discussed in section 2.2, instruction issue was done in ID stage where it waits till both the structural and data hazards are resolved.

To begin executing 'SUB', instruction issue must be divided into two stages. First, it should be checked for structural hazards and then in the next stage, wait till the data hazards are resolved. Thus, the instructions are still issued in program order but allowed to execute out-

of-order. This results in out-of-order completion of instructions which gives rise to the following complications in the pipeline:

a. Possibility of WAR and WAW hazards. As the instructions are now completed out-of-order, there is a possibility that an instruction lower in program order writes a register before it is read/write by an instruction above in program order. This can be taken care of by renaming the instructions.
b. Out-of-order completion may result in imprecise exceptions i.e., processor state when an exception is raised is not exactly the same as if the instructions are executed in strict program order. This can be resolved by delaying the notification of exception till all the instructions above the instruction generated exception are completed. A processor is said to ensure precise exceptions when an exception raised is exactly same as if the instructions are executed in program order.
c. Handling speculative instructions. In the processors with branch predictors, all the instructions issued after a branch are speculative till the branch outcome is known. Register file must be updated only when the instruction is no longer speculative to maintain precise exceptions. This can be handled by making instructions update the register file in the program order. This stage of instruction execution is called 'instruction commit'.

Now, we look in detail how these are actually implemented in hardware.

Register renaming

Register renaming is done to eliminate WAW and WAR hazards, each destination register is given a new register name. Look at the example code below

```
DIV R2 R1 R0
ADD R3 R2 R1
STR R3 10(R0)
SUB R1 R7 R8
MUL R3 R5 R1
```

WAW hazard pairs: ADD, MUL

WAR hazard pairs: ADD, SUB; MUL, STR

After renaming, the instructions look like

DIV R2 R1 R0
ADD X R2 R1
STR X 10(R0)
SUB Y R7 R8
MUL R3 R5 Y

Where X and Y are temporary registers to store the results of ADD and SUB instructions. Destination register of ADD which is R3 is now renamed to a temporary register X. Also, all the instructions below it which uses R3 as source are now renamed to X. Renaming sources to X should be done till another instruction which has the same destination register is encountered (MUL in this example). Renaming R3 to X in ADD instruction removed WAW hazard between ADD, MUL instructions and also a WAR hazard between STR, MUL. Similarly, renaming destination register of SUB instruction to Y removed the WAR hazards between ADD, SUB pairs.

Instruction commit

An instruction must not write to register file until it becomes non-speculative. Results of these instructions after out-of-order execution must be stored in a temporary registers until they are written to register file. So, instructions must commit in program order to maintain precise exceptions.

As we've seen from the above discussion, both the register renaming and instruction commit need temporary locations. An extra register set whose indices can be used as rename registers as well as to store the speculated results of instructions can serve this purpose. This is called as *'reorder buffer'*. Another buffer is also needed to store the instructions which are waiting for data hazards to get resolved. This is called as *'reservation station'* or *'instruction queue'*.

Different pipeline stages of out-of-order execution scheme are as follows.

i. *Instruction fetch:* This is same as that of simple five stage pipeline where PC is sent to instruction memory and the instruction is fetched into an inter-stage buffer.

ii. *Instruction decode:* This is also same as that of simple five stage pipeline where instruction semantics are understood.

iii. *Instruction dispatch/rename:*

a. Each instruction is allotted an entry in re-order buffer whose index is used as a new renamed register for its destination register.

b. Then, it is put in issue queue where the instruction waits till the data dependencies are cleared.

c. Instruction should be dispatched only when there is an empty entry in both re-order buffer and issue queue.

d. If an operand is available, read it from register file and put it in issue queue entry. Otherwise, store the ROB tag of the producer instruction.

iv. *Instruction wakeup:* Once the result of the producer instruction is calculated, its tag is then broadcasted on a common data bus to tell its consumer instructions to mark their operands as 'ready' and get the data.

v. *Instruction issue:* If both the operands of an instruction are 'ready' and the functional unit is available i.e., when both the data and structural hazards are resolved, instruction is sent for execution to functional units.

vi. *Execute/write result:* Execute the instruction and broadcast the ROB tag of instruction on common data bus once execution is finished to inform its consumer instructions. This stage may take multiple cycles to complete execution. Also, mark the ROB entry as valid and the corresponding issue queue entry as free/available.

vii. *Instruction commit:* An instruction commits only after all the instruction above it are completed and done execution. There are three types of actions in this stage depending on the type of instruction committing.

viii. Normal action during commit is to write the result in re-order buffer to register file and remove the entry from re-order buffer.

a. For committing store instruction, same thing is done except writing the register file.

b. For wrongly speculated branch, ROB and IQ are flushed and the instructions from correct PC are fetched.

Figure 4 shows the typical hardware of an out-of-order execution engine. Only floating point units (FPU) and load store unit are shown in the figure. Also, note that there are different reservation stations for each of the functional unit which are filled during rename stage of the instruction pipeline. Information stored in reorder buffer and reservation are described below.

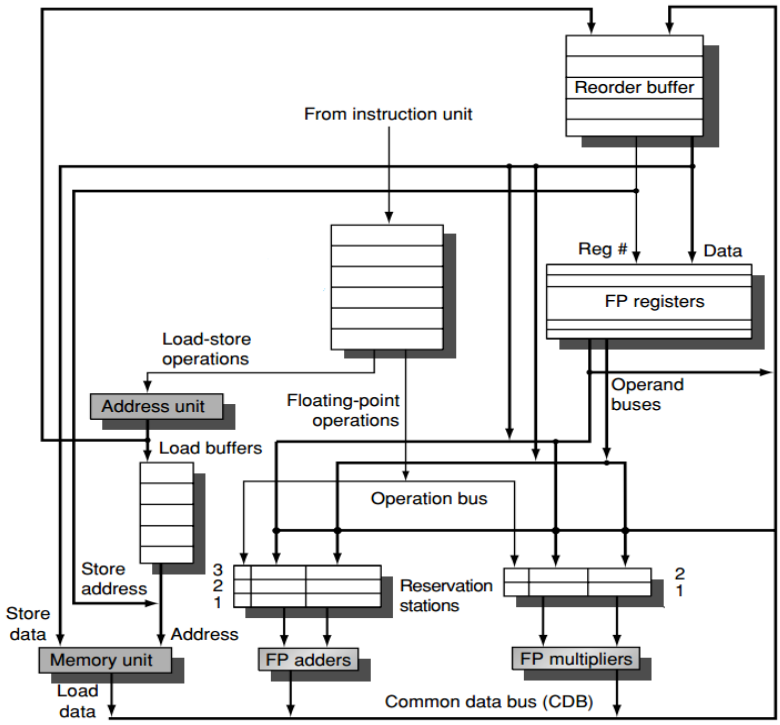

Figure 2.4: Out-of-order execution hardware [1]

Re-order buffer

Data structure of a reorder buffer entry is of the form

Valid	Data	Destination Register

- *Valid:* valid field is a Boolean value indicating whether the outcome of the instruction is calculated. This filed is filled with 'False' during the rename stage of an instruction and made 'True' when the result is broadcasted by functional unit. Note that result should be broadcasted along with the ROB tag of the instruction.

- *Data:* This field stores the calculated result of the instruction till all the instructions above it are committed. Filled during broadcast by functional unit.
- *Destination reg:* It holds the destination register of the instruction. It is the address of the register file to which the result has to be written after instruction commit. Filled during the rename stage of the pipeline.

Reservation station

Data structure of reservation station/instruction queue entry is follows

Valid	Operation	Op1 data/ ROB #	Op1 ready	Op2 data/ ROB #	Op2 ready	Destination ROB #

- Valid: Tells if the reservation station entry is valid.
- Operation: Contains functional unit type and the operation to be performed on the operands.
- Op1 data/ROB #: Holds the data of operand 1. If it is not yet calculated, it stores the ROB index of the producer instruction.
- Op1 ready: Tells if the above field is holding operand data or ROB number.
- Op2 data/ROB #: Holds the data of operand 2 (or immediate value). If it is not yet calculated, it stores index of the producer instruction.
- Op2 ready: Tells if the above field is holding operand data or ROB number. An instruction is sent to execution if there is no structural hazard and both Op1 ready and Op2 ready fields are ‘True’.
- Destination ROB #: Holds the renamed destination address

This out-of-order execution scheme is invented by Robert Tomasulo, hence it is often called as Tomasulo algorithm.

There can also be hazards in memory which should be taken care of during out-of-order execution. Loads and stores have the latency of more than one cycle. Effective address is calculated in the first cycle and memory access is performed in the next cycles. One way of dealing with the memory hazards is as follows. As the stores are always written to memory

during the commit stage of the pipeline, they happen in the program order. Hence, WAW and WAR hazards are eliminated. RAW hazards can be dealt with as follows:

- Not allowing the loads to execute their second stage if any active ROB entry is occupied by a store whose effective address is same as that of load.
- Effective addresses of loads are computed in the program order with respect to all the previous stores.

Above two restrictions ensure that the loads that memory location written by an earlier in-flight store instruction cannot perform memory access till the store instruction is writes the memory.

Advantages: Reduces stalls due to data, control and structural hazards.

2.4.6 Multiple issue

All the techniques above can be combined to achieve a maximum IPC of 1 by eliminating the hazards. Such kind of processors are called 'scalar processors' which issue only one instruction per cycle. To further improve the performance, CPI (clocks per instructions) should be made less than 1. Multiple issue in a processor allows issuing more than one instruction per cycle. Processors which issue more than one instruction per cycle are called 'superscalar processors'. There are three primary approaches in superscalar processor design.

1. *Statically scheduled superscalar processors:* Which instructions to issue simultaneously is decided by compiler and they are executed in program order in the pipeline. Number of instructions issued per cycle is not necessarily the same every cycle. Ex: Some embedded processors like MIPS and ARM cortex A8.
2. *VLIW processors:* As discussed in section 1.1, VLIW processors are like statically scheduled superscalar processors except that the fixed number of instructions are issued every cycle. Ex: Signal processing processors like TI C6x
3. *Dynamically scheduled superscalar processors:* These processors normally issue fixed number of instructions every cycle, but the execution is done out-of-order. Ex: Most of the high performance modern day processors like Intel core i series, AMD bulldozer and IBM power 8.

Most important complication arises due to superscalar execution is handling the dependencies among the instructions to be issued in the same cycle. In statically scheduled superscalar processors and VLIW processors, compiler ensures no dependencies among the instructions issued in same cycle. But, in the case of dynamically scheduled superscalar processors, hardware should take care of these dependencies.

Advantages: Reduces Ideal CPI

Although exploiting instruction level parallelism increases performance of the processor significantly, it has the following limitations:

1. Number of instructions in-flight is limited and is dependent on the size of issue queue and reorder buffer. Increasing the size of these buffers increase the hardware complexity, thereby reducing the clock frequency and increasing power dissipation.
2. Branch predictors cannot be made one hundred percent perfect.
3. It is not possible to do perfect memory address alias analysis (eliminating memory hazards) for a huge instruction window as it adds up high hardware complexity.

These above constraints makes it difficult to utilize complete instruction level parallelism of any application.

2.5 EXPLOITING DATA LEVEL PARALLELISM

All the applications which uses vectors and matrices have data level parallelism where same operation is performed on all their elements. Examples of this kind of applications are scientific computing and multimedia processing.

To exploit data level parallelism, same operation must be performed on all the independent data simultaneously. This can be done by adding an extra functional unit which takes a vector of data and operate on all the elements in parallel. Such kind of functional units are called SIMD units (single instruction multiple data). By the increase in the use of multimedia application now a days, most of the ISAs incorporated SIMD instructions. Examples are Intel AVX and ARM NEON. Graphic processing units are another class of computing resources which are built to handle the applications with high data level parallelism.

2.6 EXPLOITING THREAD LEVEL PARALLELISM

Thread level parallelism implies the existence of multiple instruction sequences which have to be run simultaneously. Instruction level parallelism thus can be primarily exploited through MIMDs (multiple instructions multiple data). Two different techniques in the processors which execute MIMDs are:

- *Multithreaded processors:* Different threads make use of same hardware resources of the processor to keep it busy.
- *Multicore processors:* Entire processor is replicated and used to run multiple threads.

Most of the modern day high performance processors use both the above techniques to exploit TLP.

The processor we aim to implement uses all the techniques described in the section 2.4 to exploit instruction level parallelism to the highest extent possible.

CHAPTER 3

ARCHITECTURAL DESIGN AND IMPLEMENTATION

This chapter explains the architectural design, design decisions and implementation details of the I-Class processor. We begin with short descriptions about RISCV instruction set architecture and Bluespec systemverilog in which the whole design is implemented. Then, we move on to detailed architecture description and implementation details.

3.1 RISCV ISA

RISC-V is a new instruction set architecture (ISA) that was originally designed to support computer architecture research and education which has a good potential to become a standard open architecture for industry applications. RISC-V was developed in the Computer science division of EECS department at the University of California, Berkeley. Following are main features of RISCV ISA [2].

- It is completely open ISA that is freely available to academia and industry unlike the commercial ISAs like x86 and ARM.
- It is suitable for direct native hardware implementation, not just simulation or binary translation.
- This ISA avoids 'over-architecting' for a particular microarchitecture style (eg., microcoded, in-order, out-of-order) or implementation technology (eg., full-custom, ASIC, FPGA), but allows efficient implementation in any of these.
- It supports extensive user-level ISA extensions and specialized variants.
- It has 32, 64 and 128-bit address space variants for applications, operating system kernels and hardware implementations.
- It has a support for highly-parallel multicore or manycore implementations.
- All the instructions are 32 bit wide and there is a scope for variable length encoding too.

RISC-V ISA is defined as a base integer ISA and optional extensions to it. Base ISA is restricted to the minimal set of instructions sufficient to provide a reasonable target for compliers, assemblers, linkers and operating systems. Currently there are two variants of base integer ISA based on the width of integer registers and address space, RV32I and RV64I. RV32I provides user address space of 32 bits and RV64I provides 64 bits of user address space. There is also RV128I which is a 128 bit version of base integer ISA.

3.1.1 Overview of base integer ISA

Base integer ISA has following types of instructions:

- *Integer computational instructions:* Contains all the arithmetic operations except 'multiply' and 'divide' instructions. One of the operand in these instructions can be an immediate field.
- *Integer load store instructions:* Instructions to loads from the data memory and stores to the data memory. Load and store instructions in RISCV ISA are of the form

Load/Store	Size of access	Destination register	Base register	Offset

  - Where 'size of access' means the size of the memory access which can be any of 'byte, half word, word and double word'.
  - Destination register is the index of register file to which loaded memory value is written. In the case of store instruction, this field stores the index of register file whose contents should be written to the memory.
  - Base register is the index of register file whose content is base address. It is added to the offset (signed addition) to get the effective memory address.
- *Control flow instructions:* Comprises of both unconditional jumps and conditional branch instructions. Conditional branch instructions compare contents of two registers and add an offset to program counter depending on the comparison result.

3.1.2 Extensions to base integer ISA

Following are the current extensions to base integer ISA:

- *M Standard extension:* Adds all the multiply and divide instructions on integers.

- *A Standard extension:* Adds instructions that atomically read, modify and write memory for inter-processor synchronization.
- *F Standard extension:* Adds floating point registers, single-precision computational instructions and single-precision loads and stores.
- *D Standard extension:* Expands the floating-point registers, and adds double-precision computational instructions, loads and stores.

Processor implemented as a part of my thesis implements RV64I (64 bit address space and integer register width) along with M standard extension (all the multiply, divide instructions) except fence and system calls.

3.1.3 Memory model

Memory supported by RISCV ISA is byte addressable and little endian i.e., least significant bytes of an integer are stored in the lower memory locations. Base ISA supports multiple concurrent threads of execution within a single user address space. RISCV threads can synchronize and communicate with other threads either through calls to execution environment or directly through shared memory system.

3.2 BLUESPEC SYSTEMVERILOG

Bluespec systemverilog [3] (BSV) is a high level hardware design language based on systemverilog. This language is based on a new model of computation for hardware, where all the hardware behavior is described as a set of rewrite rules, or guarded atomic actions. BSV borrows powerful abstraction mechanisms from advanced programming languages such as rich user-defined polymorphic types and overloading, strong basic type-checking, first class parameterization, high-order programming and object-oriented interfaces. Core BSV tool synthesizes BSV code into high-quality RTL (Verilog) which can be further synthesized into netlists for ASICs and FPGAs.

Advantages of Bluespec over the other available hardware description languages are [4]:

- *Design implementation time:* Because of its high level nature and library modules, it is easier and faster to code a design in Bluespec than any other HDLs like VHDL and Verilog.

- *Test and debug:* Bluespec generates a standalone simulator named Bluesim as an alternative to the Verilog modules which can be used to run the design on clock cycle basis. Bluesim is much faster than Verilog/VHDL simulators which makes verification faster.
- *Modulariy:* Bluespec offers more powerful parameterization for types, modules and functions allowing for better reuse and more modular designs.
- *Flexibility:* Adding extra features to the design is much simpler in Bluespec because it takes much fewer lines of code when compared to Verilog or VHDL for implementing a design.

Although it has many advantages over other HDLs in the above mentioned ways, it does have drawbacks in terms of design area, timing and power consumption. Even though Bluespec claims that it produces high quality Verilog code, not all designs are proved to be as efficient as hand written Verilog/VHDL in terms of area and timing [5].

3.3 ARCHITECTURAL OVERVIEW

Specs of the processor implemented are as follows:

- *Fetch width 2:* Fetch width means the number of instructions fetched from instruction memory to the processor per cycle. This makes the processor superscalar.
- *Branch predictor:* Tournament branch predictor used.
- *Merged register file renamer:* As we discussed in section 2.4.5, renaming instructions eliminates name dependencies.
- *Parameterized issue queue:* Holds the instructions in flight, parameter is the length of issue queue.
- *Issue width 5:* Issue queue can issue up to 5 instruction per cycle to execution units.
- *Functional units:* 2 ALUs, 1 Multiply divide, 1 Load store and 1 branch unit.
- *Bypass network:* To bypass the results from one functional unit to another.
- *Speculative LS unit:* Uses CAM based speculative load store unit.

Block diagram of the out of order engine is shown in the figure 3.1. Broadcast wires, branch predictor unit internals, inter-stage buffers and load store queues are not shown in the figure for simplicity. Following are the pipeline stages of the I-Class processor.

- Instruction fetch
- Instruction decode
- Instruction rename/map
- Instruction issue (may take more than one cycle)
- Data read
- Execute (may take more than one cycle)
- Commit

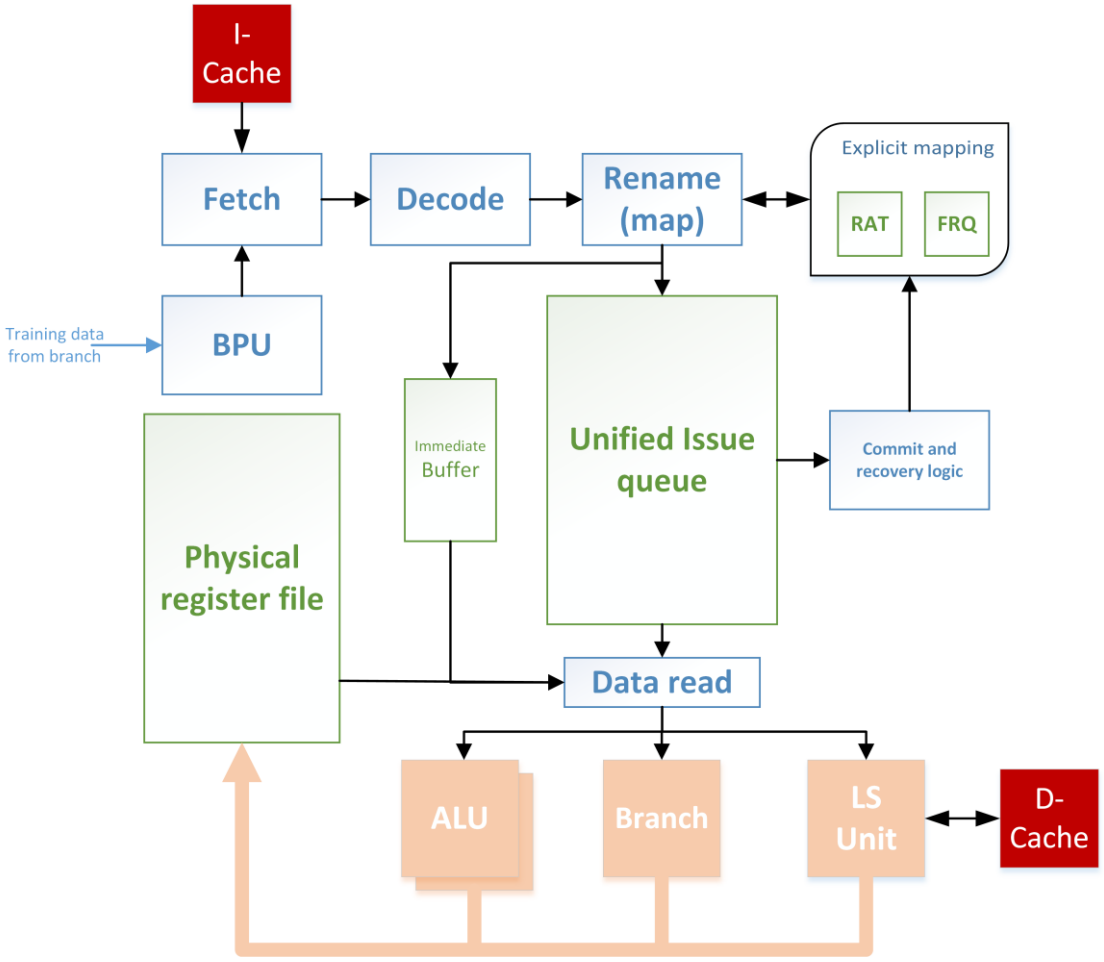

Figure 3.1: Overview of I-class processor pipeline

Design tradeoffs and implementation details of each of the pipeline stages are described in the rest of the chapter. ***All the code for above figure except ALUs, branch unit, BPU is written by myself.***

3.4 INSTRUCTION FETCH UNIT

Instruction fetch unit is responsible for feeding the processor with the instructions to execute, hence is the first block where the instructions are processed.

Instruction fetch unit sends program counter to instruction cache and receives the instructions. Fetch width of I-class processor is 2, two instructions must be fetched from instruction cache every cycle. As the instructions are being fetched every cycle, calculating next PC should be done along with cache access. Significant complexity is introduced by branch instructions as the target address to jump is not known until it completes execute stage of the pipeline. Branch predictor helps here predicting the next PC to fetch.

So, the fetch unit takes the predicted PC from branch unit and sends request to instruction cache whose line width is 64 bits (each line holds two instructions with consecutive PC values). Instruction cache responds with instruction packet which contains two consecutive instructions. For example, if fetch unit requests i-cache with PC 0, i-cache returns a packet of instructions with PC=0 and PC=4. Last three bits of PC are made zero in i-cache request as the line width is 64 bits and the memory is byte addressable. Data structure of IF/ID inter-stage buffer is

Instr0	Instr1	Valid1

Following bookkeeping actions must be performed on the instruction packet received before it is sent to next stage. It is also showin in the figure 3.2, packet[0] corresponds to lower 32 bits of cache response and packet[1] corresponds to higher 32 bits in the figure.

- If the PC is not a multiple of 8 i.e., if the last three bits of PC are not zeros, packet[1] from instruciton packet must be discarded.
- If packet[0] of instruction packet is a branch which is predicted as taken, then discard packet[1].

Valid instruction is always in instr0 field. Whereas instr1 may be valid or invalid. Keeping the instructions this way makes rename stage of the pipeline simpler.

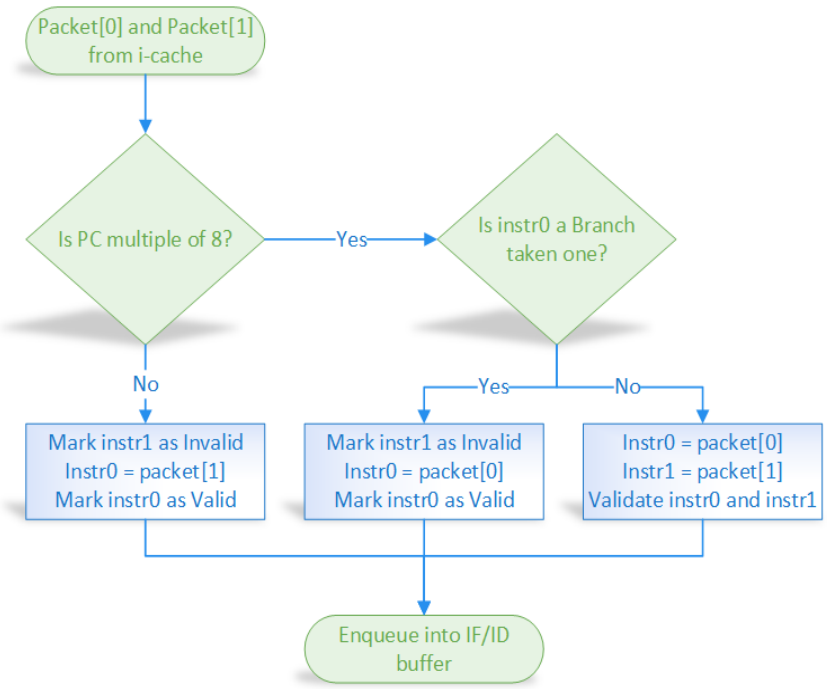

Figure 3.2: Dual instruction fetch

Branch predictor used is a tournament branch predictor which takes decisions based on both global and bimodal predictors.

3.5 INSTRUCTION DECODE UNIT

The purpose of instruction decode logic is to understand the semantics of an instruction and to define how this instruction should be executed by the processor. It identifies:

- Type of instruction: arithmetic, memory, control etc.
- Operation to be performed: ADD, SUB, bit-wise operations etc. in the case of arithmetic instructions, BNE,BEQ etc. in the case of control instructions and LB, LW, SB, SW etc. in the case of memory instructions.
- Resources the instruction requires: what the input operands and destination operand are.

Unconditional jump instruction JAL (jump and link) is specially processed in instruction decode stage. JAL changes the program counter by an offset and stores PC+4 in the destination register specified. If a JAL instruction is encountered in the decode stage target PC value is calculated and written to program counter register. Data structure of decode packet is same as

that of instruction packet in the fetch stage except that it contains decoded version of instructions.

Decoded instr0	Decoded instr1	Valid1

If instr0 is a JAL instruction, decoded instr1 should be made ‘Invalid’ in the decode packet.

3.6 INSTRUCTION RENAME AND DISPATCH

This stage of the pipeline as the name indicates performs two actions: rename the instructions to get rid of name dependencies and reserve the entries in instructions queue and related buffers. Pipeline should be stalled if issue queue or related buffers are full.

There are three ways of renaming instructions.

- Renaming through reorder buffer: This is discussed in section 2.4.5 where a buffer is used to store the results of in-flight instructions and also to rename them.
- Renaming through rename buffer: Good number of instructions in any application do not produce output (unconditional branches and stores in RISCV ISA). This results in wastage of reorder buffer entries for such instructions. This scheme is same as that of previous one except that the results are stored in a separate buffer and a link (pointer) to that buffer is stored in rename buffer.
- Renaming through merged register file: It is the renaming approach implemented in I-class processor. It is described below.

3.6.1 Renaming through merged register file

This scheme maintains a single register file which is named merged register file or physical register file to hold both the architectural registers and the results of non-committed in-flight instructions. Size of this register file is bigger than the number of architectural registers. Each register in this register file can either be free or allotted. List of all the free register is stored in another buffer called ‘free register queue’ (FRQ). Allotted register can be in any of the following states:

- Holds a committed value which means it is an architectural register.

- Holds a speculative value when result of the instruction is available but it is not committed yet.
- Holds no value if an in-flight instruction result is not yet available.

A map is maintained to store the indices of physical register file to which architectural registers are mapped.

The free register queue (FRQ) can be implemented as a circular buffer whose contents are indices of all the available registers in merged register file. A register is removed from the head of the circular buffer to rename the destination operand of an instruction. If FRQ becomes empty, pipeline is stalled.

Mapping can be implemented as a simple table of length equal to number of architectural registers. When an instruction is renamed, map table is looked up to find the mappings of source operands. Also, a register (index of a free register in merged register file) from FRQ is removed and used as a renamed register and the map table is updated to reflect this change. Merged register file implementation is shown in the figure 3.3. 'Valid' bit in physical register file indicates whether the instruction has completed execution.

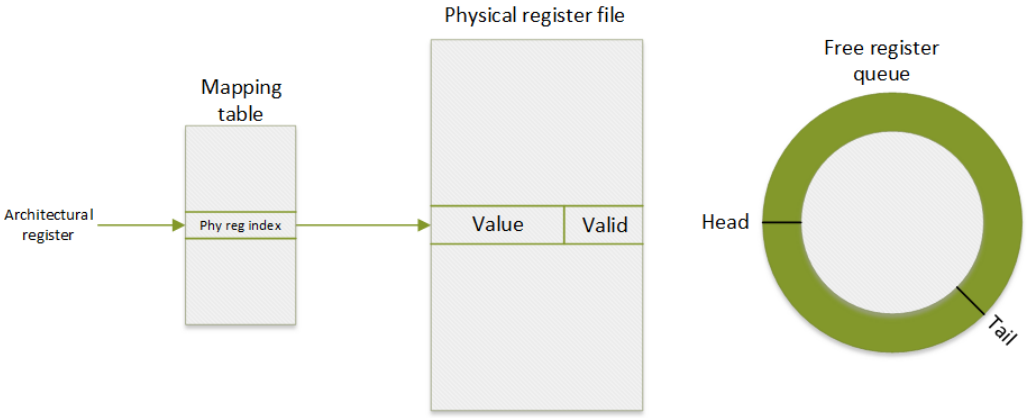

Figure 3.3: Merged register file

A physical register is freed when no instruction is going to use it anymore. Ideally, this should be done when the last instruction that uses this register commits. But, it's not possible for hardware to identify it. Hence a safe, conservative approach is used. Let us consider a destination register R_i of an instruction i is renamed to T_i. T_i is freed only when the first

instruction below instruction *i* which uses R_i as a destination operand is committed because it is guaranteed that every instruction below it uses new map of R_i. Table 3.1 shows an example of how it is done. Let us assume that the initial mapping is $R_i=T_i$ and free register queue has the register indices starting from T_8. At every commit, physical register which has the map from previous instruction with same destination architectural register is added back to FRQ. This is done in the commit stage of pipeline.

Table 3.1: Example of merged register file renaming

Instructions before mapping	Instructions after mapping	Register freed at commit
ADD R3 R2 R1	ADD T8 T2 T1	T3
SUB R4 R3 R2	SUB T9 T8 T2	T4
MUL R5 R3 R4	MUL T10 T8 T9	T5
DIV R5 R5 R2	DIV T11 T10 T2	T10
OR R6 R5 R1	OR T12 T11 T1	T6

As the mapping table changes with rename of every instruction, it needs to be reverted back to the previous state in case of mispredictions. This can be done by maintaining another rename table which is delayed version of the main one and updating it only during the commit stage of the instructions. We name them FRAM (frontend RAM) and RRAM (retirement RAM). So, RRAM contents are the of all the architectural register indices. While the entries which are there in FRAM but not in RRAM are indices in PRF holding the results of in-flight non committed instructions.

3.6.2 Register file read

Register file read is another aspect which has implications on key design aspects. Register file can be read either before issuing instruction for execution or before it.

If the register file is read before issue, the operand data should be stored inside issue queue. This takes up a lot of area especially in the processors which use large register widths. Also, there will be many replicated copies of same data in the issue as the same operands can be present in more than one instruction.

If the register file is read after issue, only the identifiers to the operands should be stored in the issue queue, not the entire data. But, it requires more number of read ports in register files because multiple number of instructions can be issued in one cycle.

Design decision

Now we have four choices which are the combination of two renaming styles (considering first two methods use same approach) and two ways of register file read. ROB based renaming requires a FIFO structure to store the speculative results. Every instruction is allotted an entry at tail of ROB and an entry at head is freed when the instruction completes execution. On the other hand, merged register file based renaming requires more complex mechanism. It has following advantages over ROB based renaming:

- Results are written only once in this scheme. Whereas in ROB based renaming, results are first written into the ROB and then into register file during commit. This results in extra power consumption.
- In merged register file based renaming, operands come from a single location (merged register file) whereas in the other case, operands can come either from architectural register file or ROB. This increases the amount of interconnect needed.

In the case of ROB based renaming, read before issue is more appropriate. If the pointer to ROB or architectural register file is stored, we need to do associative search every time an instruction commits and change the pointer value from ROB number to architecture register file index which results in complex hardware.

In the case of merged register file based renaming, either the read before issue or read after issue can be used. As the read after issue saves area in issue queue, it is more attractive choice. Because of these advantages, merged register file based renaming with read after issue is chosen for I-class processor implementation.

After renaming the instructions, they wait in the instruction queue till data and structural hazards are resolved. Data structure of instruction queue is:

Table 1.2: Issue queue data structure

Field	*Purpose*
Operation, functional unit details	Holds the operation type, functional unit details
Op1	Operand 1 pointer
Op1 ready	Whether Op1 is available
Op2	Operand 2 pointer
Op2 ready	Whether Op2 is available
Imm valid	If instruction has immediate field
Imm buffer index	Pointer to immediate buffer
Destination architectural register	To add back registers to PRF
Memory queue index	Holds the instruction position in load/store queues
Prediction	Prediction bit in the case of branch instruction
Program counter	Needed for branch and AUIPC instructions

Immediate fields can be up to 20 bits wide in RISCV ISA (LUI, AUIPC and JAL instructions). Normally, only one third of instructions have immediate field. Storing these values in a separate buffer and using pointers to the buffer in issue queue saves area. Hence, a separate buffer named *immediate buffer* is used to store the immediate fields. The field 'Imm valid' indicates if an instruction uses immediate field.

If the instruction is a memory access instruction, it is allotted an entry in load/store queues too which are described in the later sections. Its index is stored in 'Mem queue index' field.

Updating Op1 ready and Op2 ready

As we have seen in section 2.4.6, destination tag is broadcasted on to issue queue to say that producer instruction has completed execution and the consumer instructions mark their operand as 'ready'. This also sets the 'Valid' bit in PRF. But, if a consumer instruction is in rename stage when the destination tag of producer instruction is broadcasted, 'ready' field

will not be updated correctly. To avoid this, the destination tag is also broadcasted to map stage.

3.6.3 Renaming multiple instructions

In superscalar processors, multiple number of instructions should be renamed each cycle. In I-class processor two instructions are renamed each cycle. While renaming multiple instructions in a single cycle, dependencies between these instructions must be checked. Now, two registers are taken from FRQ and two entries are allotted in issue queue if the second instruction is valid. Additional bookkeeping actions that must be performed when renaming two instructions per cycle (*instr0* and *instr1*) are:

- If a source operand of *instr1* is same as that of destination operand of *instr0*, then change its identifier and mark the 'ready' field as 'False'.
- If the destination operand of *instr0* is same as that of destination operand of *instr1*, FRAM is updated only by *instr1*.
- Take care of the inter-dependencies when memory access instructions are involved. This will be discussed later in this chapter.

3.7 INSTRUCTION ISSUE

This stage of the pipeline is responsible for issuing instructions to execution units. It is the key component that determines the amount of instruction level parallelism that an out-of-order processor can exploit.

3.7.1 Issue queue

As we've discussed in sections 2.4.5 and 3.6, instruction queue holds the instructions which are waiting for the data dependencies to resolve. In I-class processor, instructions enter issue queue in the program order but they are issued for execution out-of-order. Entry in the instruction queue is freed when that instruction commits which again happens in the program order. Hence, it can be implemented as a circular buffer.

Issue queue can be implemented in two ways: One is unified issue queue and another is distributed issue queue. They are shown in figure 3.4.

- *Unified issue queue:* It is a single issue queue to store all the instructions of the program. Issue stage should have a logic to figure out to which functional unit an instruction should be sent. This logic increases with the size of issue queue and the number of functional units.

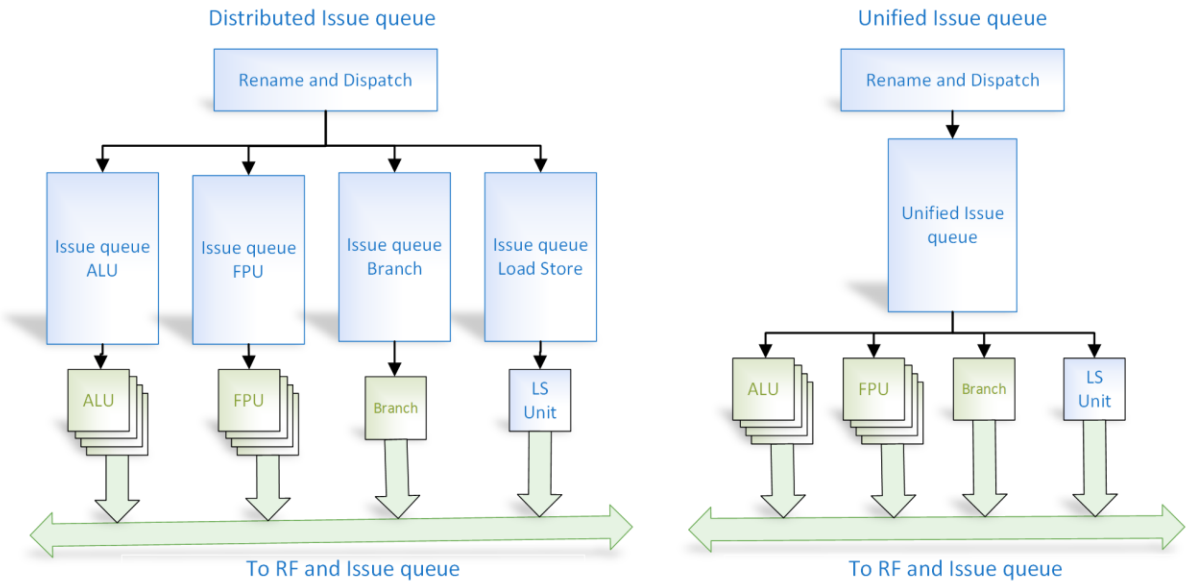

Figure 3.4: Distributed and unified issue queues

- *Distributed issue queue:* Multiple issue queues are maintained which are bound to one or more functional units. In this case, rename/dispatch stage does the job of differentiating the instruction type and enqueue into respective issue queue. This reduces the logic in the more critical issue logic.

If different issue queues are used for each type of functional unit, all the other issue queues lay unused when the workloads are not balanced (when one type of instructions dominates) which is the case with many general purpose workloads. This problem doesn't arise in unified issue queue. Hence, we decided to go with unified issue queue in I-class processor.

Instruction issue logic lies in the critical path of the pipeline which decides the clock frequency of the processor. Instruction issue is a multi-cycle operation which comprises of following stages:

3.7.2 Instruction Wakeup

When a producer instruction completes execution, its destination operand tag is broadcasted to all the instructions in the issue queue. Each instruction compares that tag with the tags of its source operands. If there is a match, source operand is marked 'ready' by setting Op1 ready or Op2 ready bit in issue queue entry. Figure 3.5 shows the wakeup logic implementation.

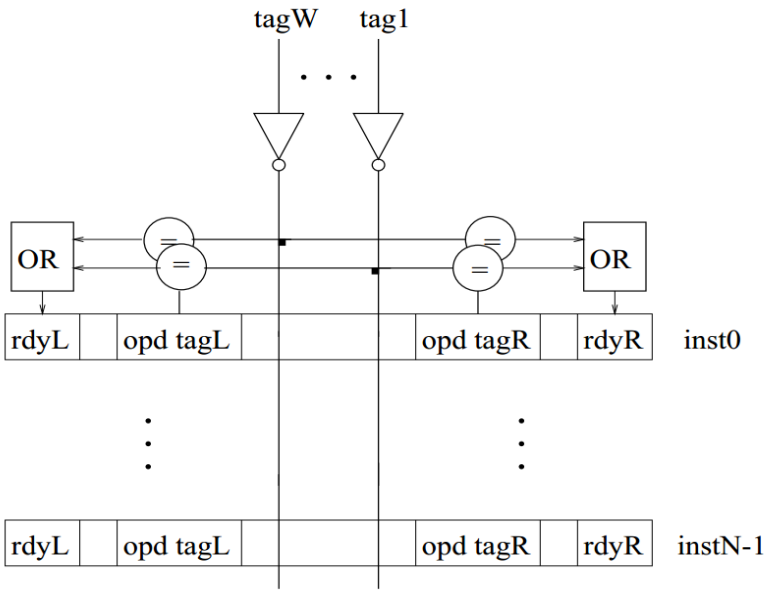

Figure 3.5 Wakeup logic [6]

Note that there will be multiple broadcast buses from different functional units, tag matching should be done with each of these buses. 'Ready' bit is set if one there is a match with one broadcast tag. As each instruction is allotted a unique destination operand from FRQ, each broadcast bus broadcasts unique tag. So, if there is a tag match, it happens with only one broadcast bus.

Delay of wakeup logic increases with increase in number of functional units and issue queue size.

3.7.3 Instruction select

Instruction select is in charge of selecting instructions from a poll of 'ready' instructions to be sent for execution i.e., instructions whose all operands are available. There should be a select logic because the number of 'ready' instructions in issue can be more than the number of available functional units. Inputs to the select logic are requests form instructions in issue queue. This is named as '*request vector*' with is an array of bits with size of the array equal to number of entries in issue queue. If an instruction in the issue has both the operands 'ready' and it is not yet selected for execution, then the bit of the request vector in the corresponding location is set. Outputs of select logic are the grant signals named as *grant vector* which also has the size equal to issue queue size. It indicates if an entry in issue queue is granted a functional unit. A selection policy should be used to decide which of the 'ready' instructions should be granted functional units. There are mainly two kinds of selections policies:

- Age based: Older instructions in the issue queue are given more priority i.e., the instructions which entered the issue queue earlier. Implementing age based selection policy requires more hardware to keep track of age information.
- Position based: Instructions in the top of issue queue are given more priority. This policy is relatively simple to implement as it only requires priority encoders.

As this stage of the pipeline lies in the critical path, design decision should be made based on the final throughput which largely depends on the kind of workloads the processor is running. As we don't have the workload data, it is designed in such a way that the selection policy can be set in the compile time.

Position based policy can be implemented using a simple priority encoder based arbiter. An encoder is a combinational circuit with 2^N inputs and N outputs. The output of the encoder is the binary representation of the input position activated. In priority encoder, each position has a weight. In the case of several inputs are set, the position with highest weight is codified in the output. Priority encoder based arbiter is shown in the figure 3.6a. It takes request vector and enable as the inputs and outputs grant vector. This circuit has the number of levels equal to the size of issue queue resulting in the delay linearly dependent on the size of issue queue.

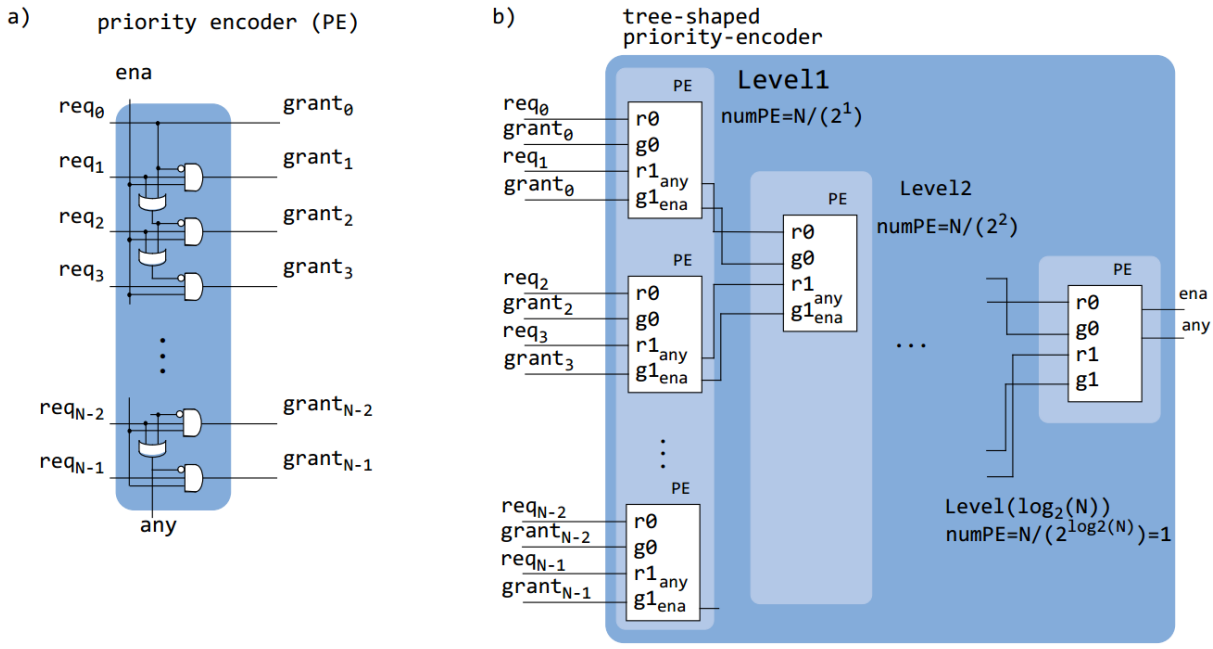

Figure 3.6: Select logic implementation [7]

Another approach which is much more efficient in terms of delay is shown in the figure 3.6b. It uses a tree based priority logic named as encoder tree works this way. Each block in the tree is a priority encoder shown in 3.6a with N=2. Requests from the top level are first sent to the root of the tree. This is done by 'any' output of the PE block which is set in any one of the bits of request vector are set. 'Enable' to the root PE is the functional unit status, it is 'set' if the functional unit is free. Grants from the root are used as 'enable' inputs of the PEs in the next higher level. This way, the grants are passed back to the branches. In this implementation, number of levels of logic is $log_2 N$. Hence, the select logic delay is logarithmically dependent on the size of the issue queue. Tree based select logic is used in I-class processor implementation.

Instructions in the I-class processor enter issue queue in the program order and leave the issue queue in program order. Hence, the instructions nearer to the head of issue queue are the older compared to the ones which are farther from it. To implement age based selection policy, the request vector is rotated down by the amount which is equal to the head position. And the grant vector from the select logic is again rotated up by the same amount. As this requires rotation by arbitrary amount, two barrel shifters is required to perform this operation, one for

rotating request vector and another for rotating grant vector. Adding them to the select logic significantly increases the delay in that path.

There are as many number of encoders as the number of functional units in the processor. As I-Class processor implemented with 2 ALUs, a multiply divide unit, a load store unit and a branch unit four encoder trees, five encoder trees are needed. Also, four request vectors are made one for each type of functional unit.

Select logic for multiple ALUs:

I-class processor has 2 ALUs. Up to 2 ALU instructions can be selected for execution every cycle. Select logic should make sure that the same instruction is not selected for execution in both ALUs. This is done as shown in the figure 3.7.

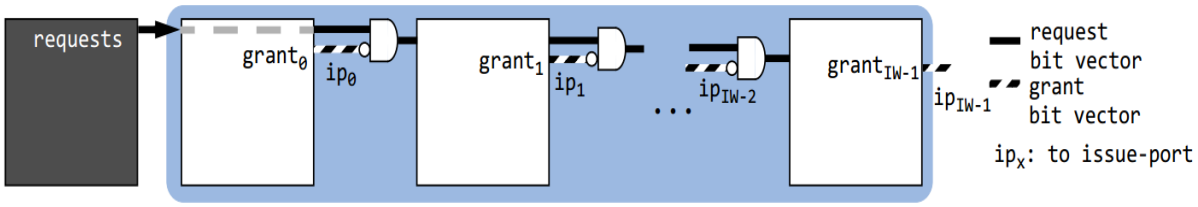

Figure 3.7: Select logic for multiple ALUs [7]

Grant from ALU0 is negated and bitwise and operation is performed with original request vector. This result is used as the request vector for encoder tree of ALU1. The same logic extends if there are more functional units.

Selected instructions are enqueued into the inter-stage buffers (select/data-read buffer). If both the source operands of the instruction are ‘ready’ before entering the issue queue, there is no need to wake up the operands. Such instruction will proceed to the next stage of the pipeline once all the instructions which use same functional unit above it are selected. To keep track of which instructions are selected for execution, a Boolean array with size of issue queue is maintained. Once an instruction is selected for execution, corresponding entry in that array is made ‘True’ and it is will no longer generates the request.

3.8 INSTRUCTION DATA READ

As discussed in section 3.6 instructions read register file after issue. Selected instructions are taken from the select/data-read inter stage buffer and the register file is read. I-class processor can issue up to 5 instructions per cycle which results in 10 read ports for physical register file.

Immediate fields of instructions are stored in separate location (immediate buffer) and the pointer to the immediate buffer location is stored in issue queue entry. If the instruction has 'imm valid' field set, then the immediate buffer is accessed to get the immediate field data.

Instructions along with the data are enqueued into the next inter-stage buffer (data-read/drive buffer).

3.9 INSTRUCTION EXECUTE

This is the stage where the instruction execution takes place. This stage may take multiple cycles depending on the type of operation. Once the result is calculated, it is broadcasted to all the instruction queue and rename stage to update the status of source operands. Branch unit also broadcasts the training packet, the branch result (whether prediction is correct or not) and the value to which PC should jump when there is a branch misprediction (named as *squash program counter*). Branch result and the squash program counter are stored in separate array (named as *squash vector*) of size equal to the size of issue queue. Squash program counter field is filled with program counter value in the rename stage which is used in committing load instructions as we will see in next section.

As discussed in section 2.4.6, data dependencies through memory cannot be identified in the rename stage. A memory data dependency exists between two instructions if both of them access the same memory location i.e., when there exists an aliasing or collision between two memory addresses. These dependencies can be checked only once these instructions have been issued to execution and the effective address is computed. The mechanism in charge of handling data dependencies through memory is called '*memory disambiguation policy*'. There are two types of memory disambiguation policies: *Speculative memory disambiguation* and *non-speculative memory disambiguation*. First scheme does not allow executing memory operation until we are sure that it doesn't have dependencies with any previous memory

operation. On the other hand, speculative memory disambiguation schemes predict whether a memory operation will have dependence on other in-flight memory operations.

As around 30% of instructions in general purpose workloads are memory accesses. Implementing a conservative memory disambiguation mechanism may produce unnecessary serialization that could significantly limit the instruction level parallelism that can be exploited. Next section explains the architectural details of load store unit implemented in I-class processor.

3.10 CAM BASED SPECULATIVE LOAD STORE UNIT

In the conservative memory disambiguation schemes, both the loads and stores are issued for execution in program order. To gain performance improvement, memory access instructions should be executed out-of-order without violating memory data dependencies. Memory models impose certain constraints on out-of-order execution of these instructions by the processor. To facilitate the recovery from exceptions, stores to the memory must be done in the program order. By executing stores in the program order, WAW and WAR data hazards through the memory are eliminated. Only RAW hazards should be handled by processor's load store unit. The approach used to handle these dependencies is similar to result bypassing/forwarding in five stage pipeline discussed in section 2.4.1.

3.10.1 Load bypassing and load forwarding

In load bypassing, if a load instruction is not dependent on the trailing stores in the pipeline, it is sent to execution. This enforces a constraint that a load instruction can be issued for execution only after all the in-flight stores are issued. This is because, for the load to check for aliases with in-flight store instructions, their effective addresses for stores must have already been computed.

In load forwarding, store data is forwarded to load instruction if there is an aliasing with an earlier store. It is also called as store forwarding since the store data is forwarded. We still need to execute the load only after all the earlier stores are issued. Implementing this in the hardware requires separate buffer which holds all the store results and the effective addresses. When a load is sent for execution, it associatively check that buffer and get the value directly from it if there is a match. One complication in this scheme arises in the case when there is

more than one match during associative search i.e. when there is more than one store in the pipeline which access the same memory location as that of load. One needs to have extra hardware to ensure that the load is forwarded with latest aliased store.

Using these two techniques, loads can be executed out of order with instructions other than stores. Although implementing these techniques takes extra hardware resources, performance gain of up to 25 % can be achieved [8].

3.10.2 Speculative load execution

If the loads are executed out-of-order with stores when using above techniques, we may fail to forward the store because the effective address of that store might not have been calculated yet. But, both aliasing with a previous store and the load being issued earlier than that store happens very rarely in any practical workloads. This fact is used to further exploit the instruction level parallelism among memory access instructions.

In this scheme, load is issued even if the in-flight stores earlier in the pipeline are not yet issued. This is called speculative load execution because as we predict that no memory violation occurs because of that load. Memory violation occurs when the load is not forwarded or when it is forwarded from wrong store. If a memory violation occurs, the entire pipeline should be flushed and the instructions starting from the mispredicted load are executed again. Hardware implementation details of this scheme are described below.

Two buffers are maintained to hold the memory disambiguation status of load store instructions, one for loads and one for stores. Every memory access instruction is allotted an entry in these buffers. Allotted entry is freed only during commit stage of that instruction. As these buffers are filled and freed in program order, they are named as load/store queues. Data structures of load and store queue entries are given the tables 3.3 and 3.4. Here is how load and store instructions are processed in rename and issue stage of the pipeline:

- An entry in load/store queue is allotted in the rename stage of the pipeline and ‘Filled’ bit is set and ‘Valid’ bit is reset in the entry. If the queue is full, pipeline is stalled. In the case of load instruction, store mask field is filled. Store mask is a bit vector of the size equal to the size of issue queue. As all the instructions are renamed in program order, all the in-flight stores that the load being renamed depends on can be found from

the ‘Filled’ field in store queue. Store mask field is just a copy of ‘Valid’ bit array in store queue. While renaming multiple instructions in a cycle, store mask should be appropriately updated if the first instruction is store.

- Index of load/store queues is stored in the field ‘memory queue index’ along with the instruction in issue queue.
- Instructions are issued to load store unit once all the data and structural dependencies among registers are resolved. Maximum of one memory instruction can be issued for execution in each cycle. In the load store unit, store and loads are handled separately.

Table 3.3: Load queue data structure

Field	*Purpose*
Filled	Tells if an entry is allotted to a load instruction
Valid	If the data is load queue is valid
Store mask	List of all stores a load is dependent on
Load address	Memory access location of load instruction
Load size	Size of access (B,HW,W,DW)
Forwarded	To indicate if a store forwarding has happened for a load
Forward acknowledge	To indicate if the aliased store has committed
Aliased	Indicates that a memory violation has occured

Table 3.2: Store queue data structure

Field	*Purpose*
Filled	To indicate if the store queue entry is allotted
Valid	To indicate if the filled entry has valid data
Store address	Memory location address to which the data should be stored
Store data	Data to be stored
Store size	Size of memory access (B,HW,W,D)

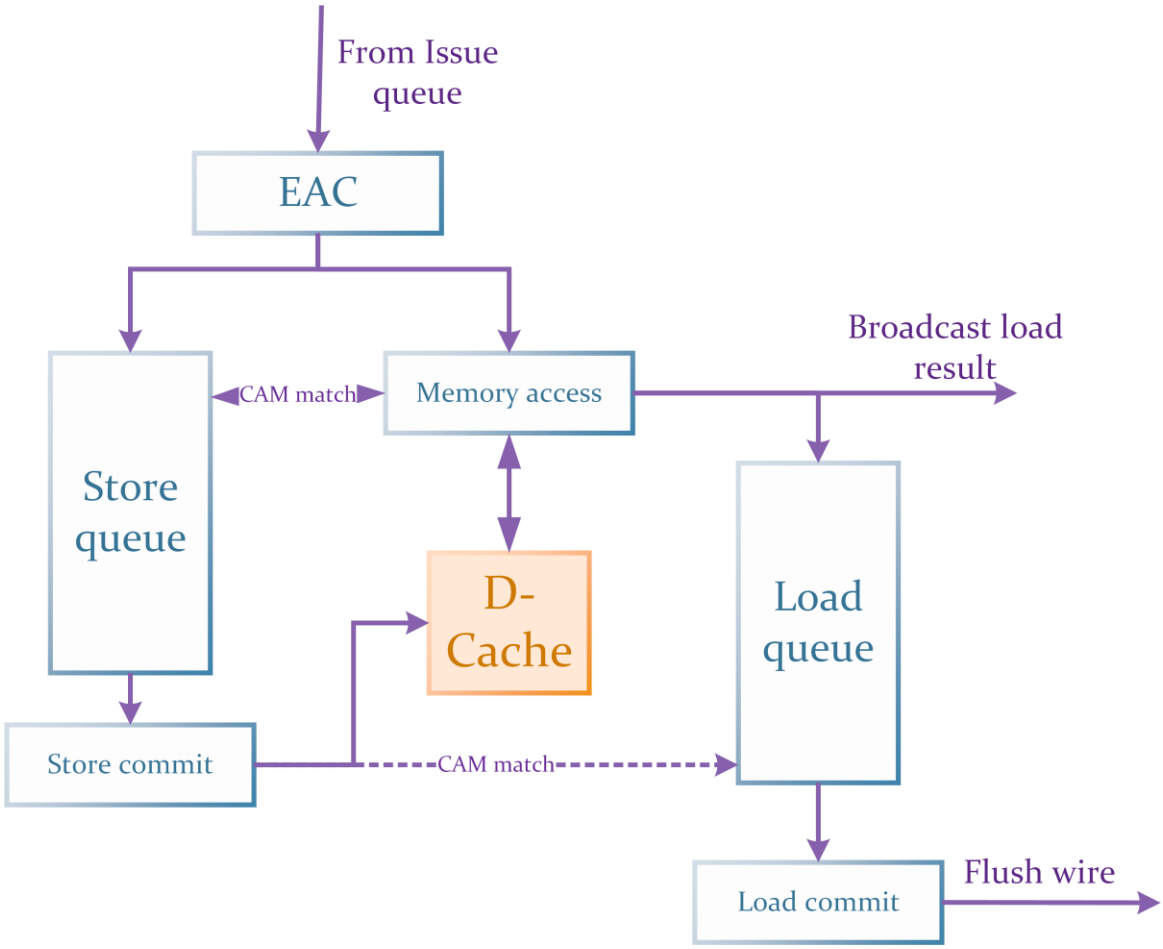

Figure 3.8: CAM based speculative load store unit

Load and store execution

Block diagram of CAM based speculative load store unit implemented in this thesis is shown in the figure 3.8. Store instruction execution is finished in one cycle whereas a load instruction can take any number of cycles depending on cache hierarchy. First cycle of execution for both load and store is effective address calculation in which signed offset is added to the value in base register. After effective address calculation, store instruction updates valid (is set), store address, store data and store size fields of the store queue. Note that the index of the store queue comes from issue stage where memory queue index is stored.

After effective address calculation, load instruction updates Valid (is set), load address fields. Also, the instruction is enqueued into an inter-stage buffer. In the next cycle, associative

search is performed on store queue to check for address match. Search is performed only on the store queue entries which are there in store mask of the load instruction. If there is a match, that result is broadcasted to issue queue and 'forwarded' bit in load queue is set. If there is no match, read request is sent to data cache.

Instruction commit

Main part of memory disambiguation is done in the commit stage of the instruction. In the commit stage store sends the write request to data cache. Along with it, it checks if any in-flight loads aliased with it and update this information in load queue. A flow chart of how this is done is shown in the figure 3.9.

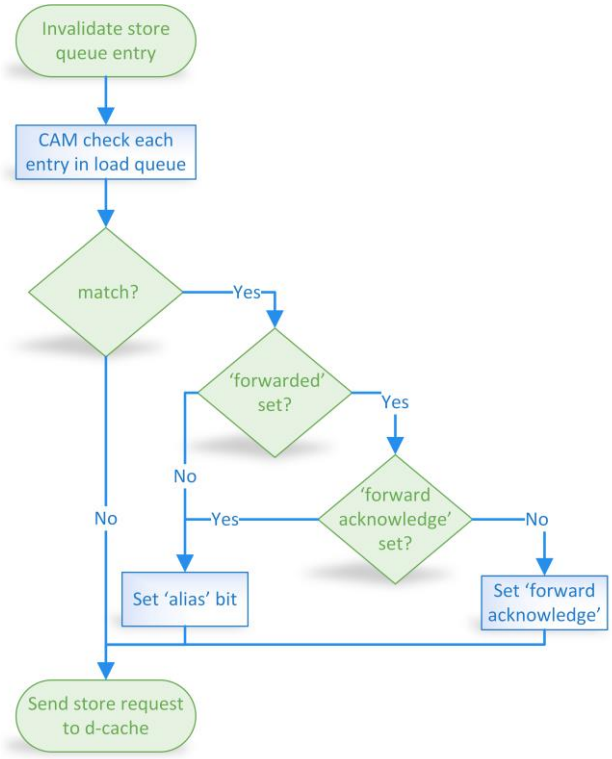

Figure 3.9: Store commit flowchart

'Alias' bit in the load queue is set in the following cases:

- If there is address match but the store forwarding has not happened.
- If more than one store has memory data dependency with a load instruction. As age based forwarding is not done, a more conservative approach is taken to ensure no memory violation, pipeline is flushed and the load is re-executed that case. 'forward

acknowledge' bit is used to find such cases. First aliased store sets 'forward-acknowledge' bit during commit. If another aliased store commits, it sees 'forward-acknowledge' bit set and sets 'alias' bit because the wrong store might have forwarded.

Load commit is relatively simple, load queue entry is invalidated and pipeline flush signal is sent (corresponding bit in the squash vector is set) to processor if the 'aliased' bit is set.

Handing multiple access sizes

A conservative approach is followed. Store is forwarded only when both the address and access size are matched.

3.11 RESULT FORWARDING/BYPASS NETWORK

As the issue stage normally lies in critical path of the pipeline, it is pipelined as much as possible to reduce its complexity. Dividing issue stage into two cycles, namely wakeup and select stages has implication on IPC.

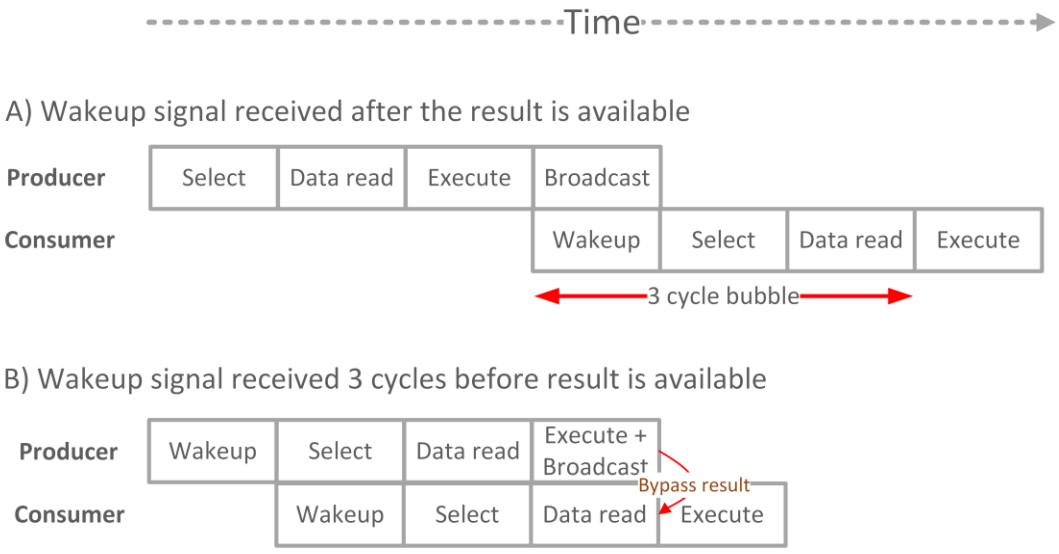

Figure 3.10: Pipeline bubble

As shown in the figure 3.10.A, a three cycle bubble results in pipeline when producer and consumer instructions are back to back in the program because the consumer instruction is woken up only at the broadcast stage of the pipeline. As we are executing instructions out-of-order, other 'ready' instructions can be executed in this time. But, if a work load

has significant number of producer consumer pairs consecutively, IPC is noticeably reduced.

To eliminate three cycle bubble, consumer instructions can be notified thee cycles in advance to the result broadcast stage. If producer instruction is of single cycle latency, instruction select and wakeup must happen in the same cycle to avoid pipeline bubble as shown in figure 3.10.B. As both the wakeup and select logic have high hardware complexity, it further increases critical path delay in the processor. Also, extra hardware should be added in other parts of the pipeline to keep track of when the result of the producer result will be broadcasted.

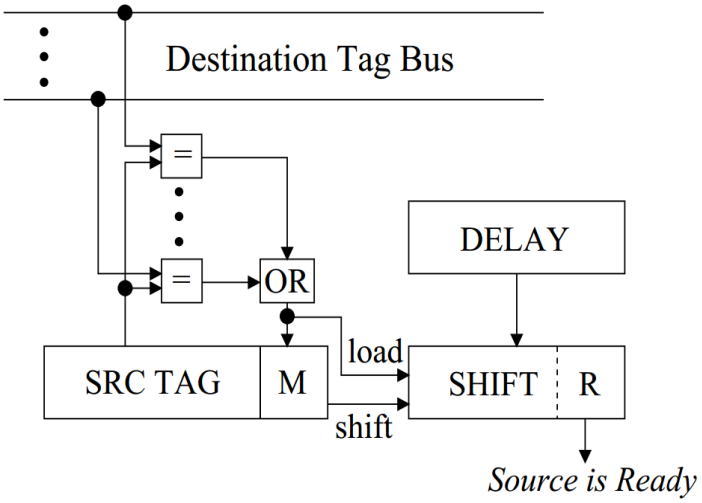

Figure 3.11 Wakeup logic for issue stage with bypass network [9]

Now, instead of having a bit for ‘Operand ready’ and ‘Valid’ fields in PRF, block shown in figure 3.11 is used. Idea here is to use a shift register which counts the number of cycles after an instruction is issued for execution. Note that the consumer instructions of loads cannot be woken up earlier because of their variable execution latency. We know the latencies of all the instructions except loads, this latency is stored in ‘delay’ field. This delay is coded in inverted radix-1 value, in which 1 is 11..10, 2 is 11…100 etc. We want to wake up the consumer instructions three cycles before the result of producer is available. If an instruction has latency N, delay field is filled with N-1 in inverted radix-1

representation. When an instruction is selected for execution, its tag is broadcasted on to all the instructions in issue queue and rename stage. If there is a tag match with any source operands, then the delay is loaded into the 'shift' field and match bit (represented by M) is set which acts as an enable for shift operation. The least significant bit of the shift register is used as the 'ready' field for the operands. Now, source operands consumer instructions of single cycle latency producer instruction are made 'ready' in the same cycle it is selected. By using this approach, consumer may be in the data read stage by the time result of the producer instruction is broadcasted. This value is directly passed between the functional units by a bypass network. Bypass network is a set of wires connecting all the possible pairs of functional units. Bypass network for two functional units is shown in the figure 3.12.

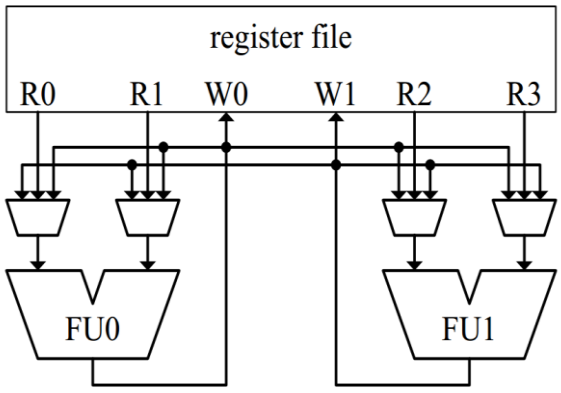

Figure 3.12: Bypass network between two functional units

Bypass network complexity increases exponentially with number of functional units and is a potential contributor for critical path delay in modern day processors.

I-class processor is implemented in two versions, one with bypass network and one without it.

3.12 INSTRUCTION COMMIT

It is the final stage of the instruction in the pipeline. To preserve the precise exception behavior, all the instructions are committed in the program order. All the instructions in issue queue are stored in program order. The job of the commit logic is to look at the oldest instructions i.e. the instructions at the head of issue queue and do all the bookkeeping actions

if they have finished execution. This is known by maintaining an array of bits with size equal to the size of issue queue. When an instruction tag is broadcasted, the bit corresponding to the issue queue entry with the same destination operand is set. Following bookkeeping actions are performed by commit logic:

- Add back the register to free register queue in the way described in section 3.6.1. In I-class processor, even the instructions which do not produce any result (conditional branches and stores) are allotted a free registers from FRQ. They are used to just mark the instruction as 'execute done' by broadcasting the tag. For these instructions, the free registers allotted are added back to FRQ in commit stage.
- If an instruction uses an immediate field, it is allotted an entry in immediate buffer. Immediate buffer is freed in commit stage for such instructions.
- If the instruction is a memory access instruction, perform the actions described in section 3.10.
- Update RRAM index of destination architectural register with the destination operand (the free register it is allotted during rename) for all the instructions which produce result (all instructions except stores and conditional branches).
- If the instruction being committed is a mispredicted branch, send a signal to flush the pipeline.
- If the instruction being committed is a wrongly speculated load, signal the pipeline flush. Note that this load should be re-executed.

3.12.1 Committing multiple instructions

I-class processor can commit up to two instructions per cycle. Additional actions that have to be performed which committing two instructions per cycle are:

- If the first instruction is a mispredicted instruction, do not commit second instruction.
- If the first instruction is store, then do not commit the second instruction if it is a store or load. We assumed that cache can take only one write request per cycle. If we commit a load and a store in a single cycle memory dependency between these two have to be checked in the same cycle which is not done to keep the design simple.
- If the destination architectural registers of both the instruction in commit are same, then only the second instruction will update RRAM.

3.12.2 Recovery mechanism

Processor state must be reverted back when an instruction is mispredicted. Reverting the processor state involves following steps:

- Clear all the inter-stage buffers in the processor.
- Empty load and store queues (invalidate all the entries and change head and tail to zero).
- Empty issue queue (invalidate all the entries in issue queue and change head and tail to zero).
- Copy RRAM contents to FRAM to restore the previous mapping.
- As each instruction in the pipeline is allotted one free register, flushing the entire pipeline should add back all the registers to FRQ. As FRQ is implemented as circular buffer, all we need to do is to mark all the FRQ entries as valid and change tail and head to zero.
- Change the program counter to the value present in squash program counter of the mispredicted instruction. For conditional branches, this value is broadcasted by branch unit. In the case of wrongly speculated load instructions, instructions from that load should be re-executed. As the squash program counter is filled by the instruction program counter in rename stage, it can be used as new PC.

As all these operations just update register values, recovery takes single cycle.

In RISCV ISA spec, R0 is always hardwired to zero. The map of R0 in FRAM and RRAM are always maintained constant (R0→T0 in PRF). If an instruction has R0 as destination operand, do not update FRAM and RRAM maps in rename and commit stages respectively.

CHAPTER 4

VERIFICATION, RESULTS AND CONCLUSIONS

This chapter explains how I-class processor is verified and the performance results.

4.1 VERIFICATION FRAMEWORK

All the modules of I-class processor are written in Bluespec systemverilog. It has separate memories for storing instructions and data. Interface between processor and caches is shown in the figure 4.1.

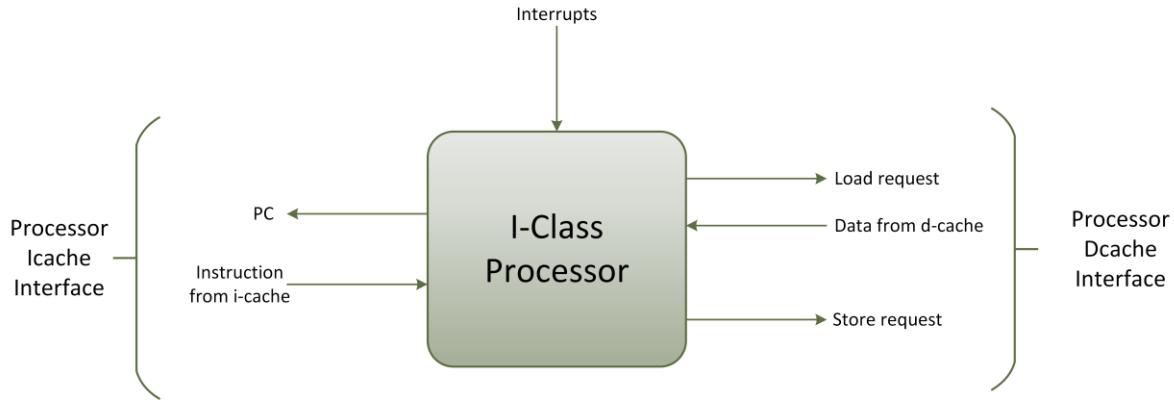

Figure 4.1 Interface of I-Class processor

Instructions from automatic test case generator (also called as automatic program generator, APG) are first loaded into the instruction cache. Initial memory state is also generated by APG which is loaded data cache. Instructions in i-cache are fed into the processor. Instruction and data caches are single level caches with load-to-use latency of one cycle. After commit of each instruction, the contents of register file along with the program counter are dumped into an output file. Also, the contents of different buffers in the processor are also dumped into output logs every cycle for debugging purposes.

The register dump is then compared with the register dump generated by instruction set simulator (ISA). ISA takes the same instructions and data memory initialization files from APG, simulate the output in software and dump the register contents after each instruction execution into a dump file. Both the register dumps are then compared to see if there is any mismatch. Simulation environment is shown in the figure 4.2.

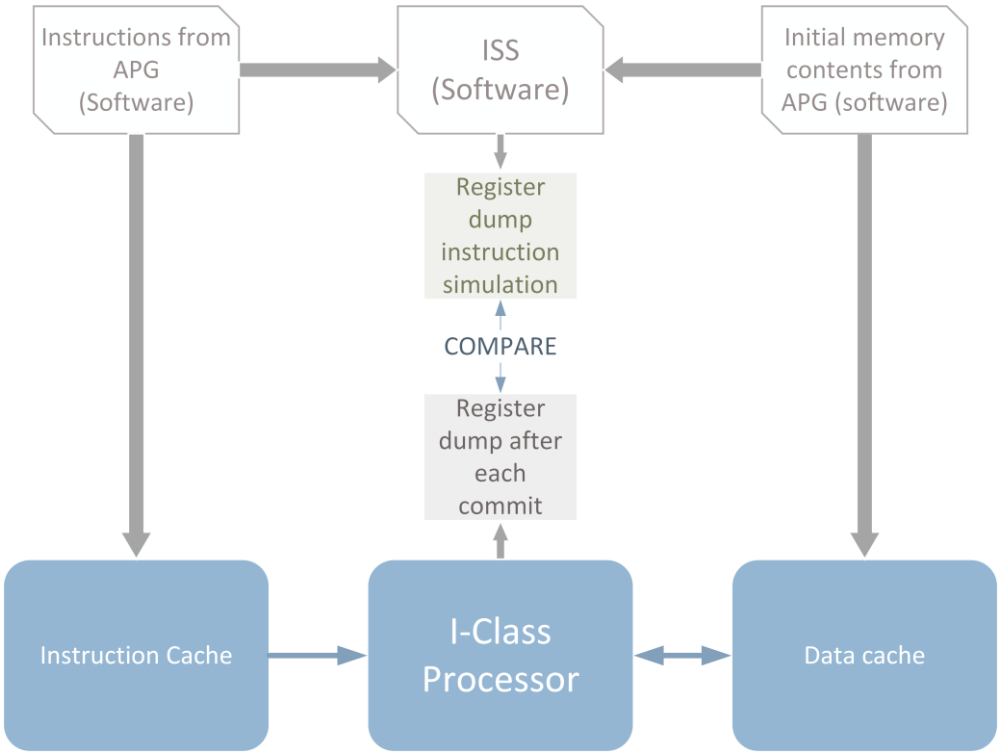

Figure 4.2 Verification environment

As APG generates large number of instructions with random operations, source and destination operands, all possible combinations of instruction are expected to be covered. Several hundreds of test cases were run on processor and the processor core was debugged successfully. All the corner cases of memory disambiguation are verified manually as the memory space is vast and there is less chance that all those cases are covered by APG generated test cases. Also the instructions (Multiply and divides) which were not working on ISS are verified manually.

Same verification has been carried out after converting Bluespec codes to Verilog modules. Verilog simulations are done in Xilinx ISA.

4.2 RESULTS

One of the main advantages of Blusepec systemverilog over other hardware languages is that it is more flexible and supports extensive parameterization. Any architectural feature can be added to the existing code without much effort as Bluespec takes care of generating all the control logic. Impact on performance of the processor for different configurations is tested. Parameters that can be varied are the issue queue size, selection policy used and the bypass network.

4.2.1 IPC

Same framework shown in the figure 4.2 is used for measuring the IPC (instructions per cycles committed) of the processor. Along with the register dump, number of cycles taken and statuses of different processor buffers are also dumped into a file. As the actual benchmarks were not available in machine readable format, hence IPC is measured with APG test cases. APG generates random instructions from the set of RV64I instructions which are integer arithmetic instructions, variable size loads and stores, conditional and conditional branches. Load and store instructions have latency of two cycles whereas all the other instructions have latency of single cycle. Percentage of each type of instructions can be set in APG. Distribution of instructions was set to be 30% percent loads and stores, 60% arithmetic instructions and 10% branch instructions which is the case with general purpose workloads. But APG doesn't guarantee the generation of same kind of instruction dependencies as that of general purpose workloads. Also, branches generated by APG instructions jump to random addresses which results in poor branch prediction. Hence, running APG instructions is not the actual measure of attainable IPC.

Size of instruction queue decides the size of in-flight instruction window. More is the size of instruction window, more is the exploitable instruction level parallelism. Figure 4.3 shows the IPC variation with size of the issue queue in the processor with no bypass network and position based selection policy. IPC increases with the increasing the size of issue queue.

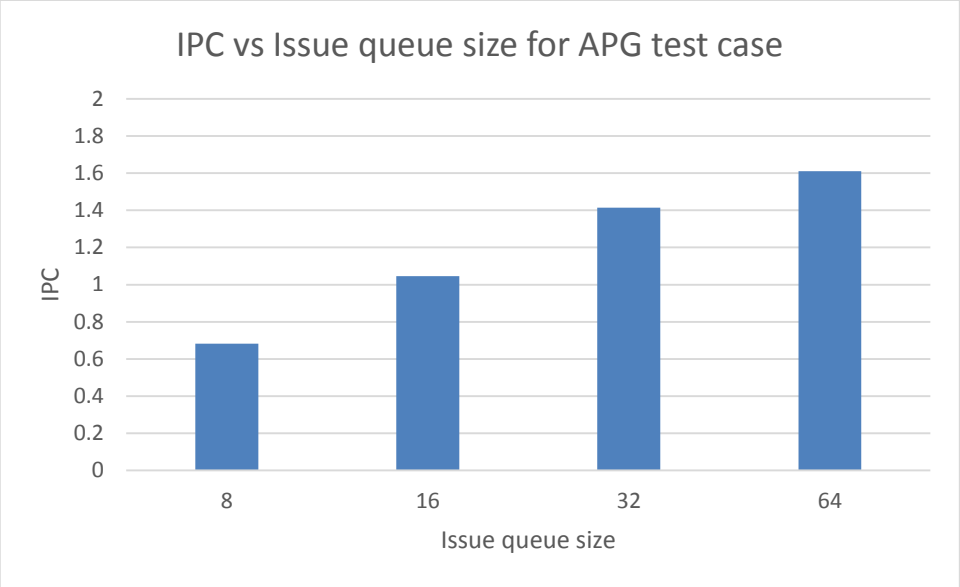

Figure 4.3: IPC vs Issue queue size

Now, the selection policy is changed for all the sizes of issue queue sizes. Age based selection policy should have higher IPC as it resolves the data dependencies quicker than position based selection policy. Its impact on IPC of the processor is shown in figure 4.4. As we can see, there is no noticeable improvement in IPC with selection policy being used for APG test case.

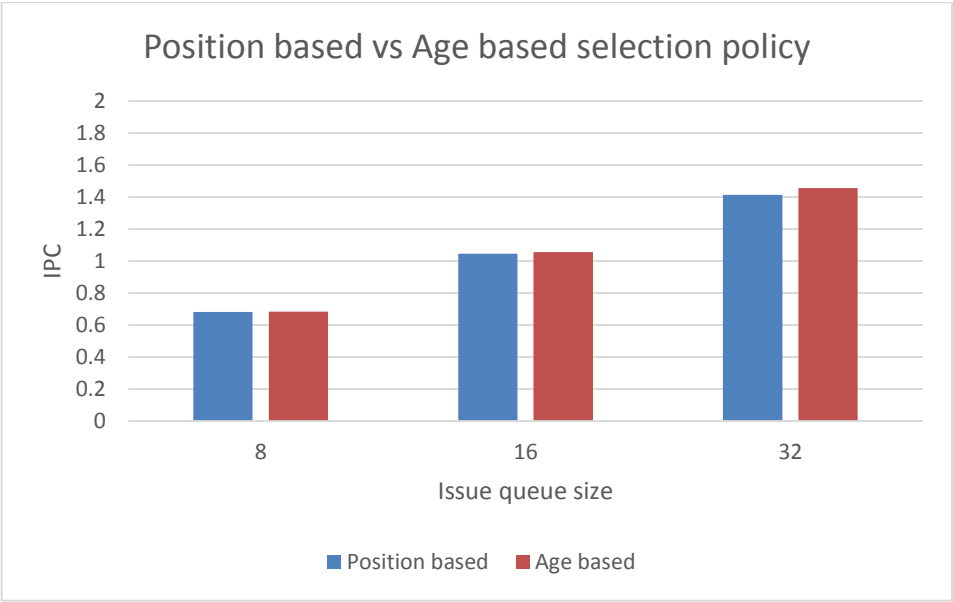

Figure 4.4: IPC improvement with selection policy

As discussed in section 3.11, bypass network eliminates three cycle bubble between producer and consumer instructions. Impact of bypass network on IPC in the processor with position based selection policy is shown in the figure 4.5. Huge IPC improvement can be seen with bypass network.

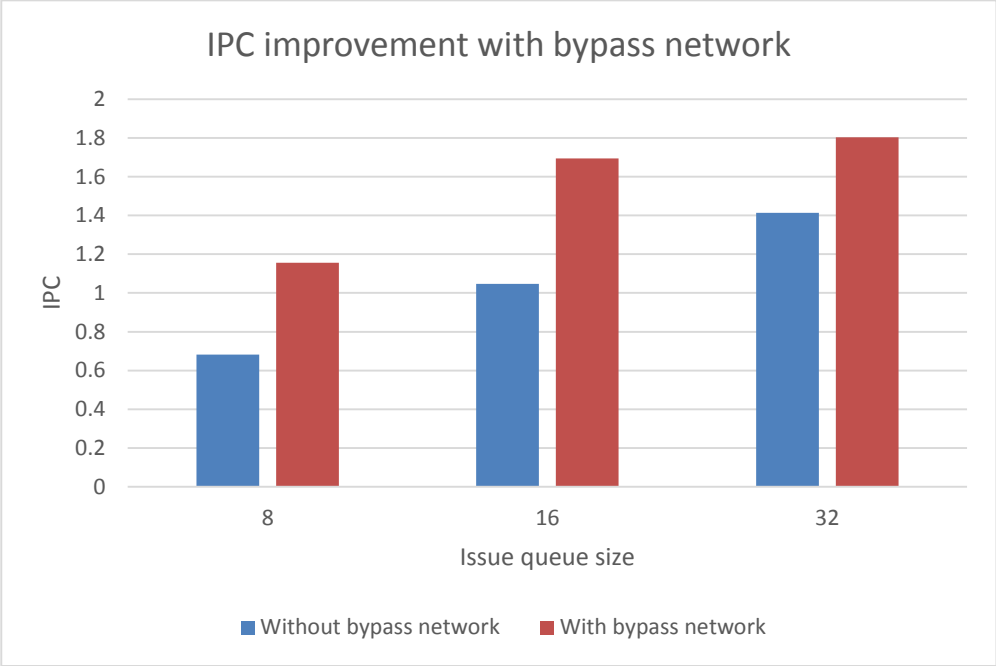

Figure 4.5: IPC improvement with bypass network

As APG does not generate code with heavy dependencies and generates only the instructions with single and two cycle latencies, IPC close to 2 is expected. As discussed above, poor branch prediction is resulting in IPC not reaching 2. Figure 4.6 shows the IPC of the processor in all the configurations with the APG test case with no branches. It can be seen that IPC almost reaches theoretical maximum of 2 (1.996) for issue queue size of 32, age based selection policy and with bypass network.

4.2.2 Clock Frequency

Impact of different architectural techniques on maximum attainable clock frequency of the processor is tested. As discussed in chapter 3, the maximum attainable frequency is limited by the logic in critical path which is normally the issue stage of the pipeline or the bypass network. Bluespec code is first compiled to Verilog modules. Clock frequency is determined

by running Cadence RTL compiler on the Verilog netlist. *65 nm UMCIP* Standard cell library with operating conditions *1.32 V* supply voltage and *110* ^{0}F is used.

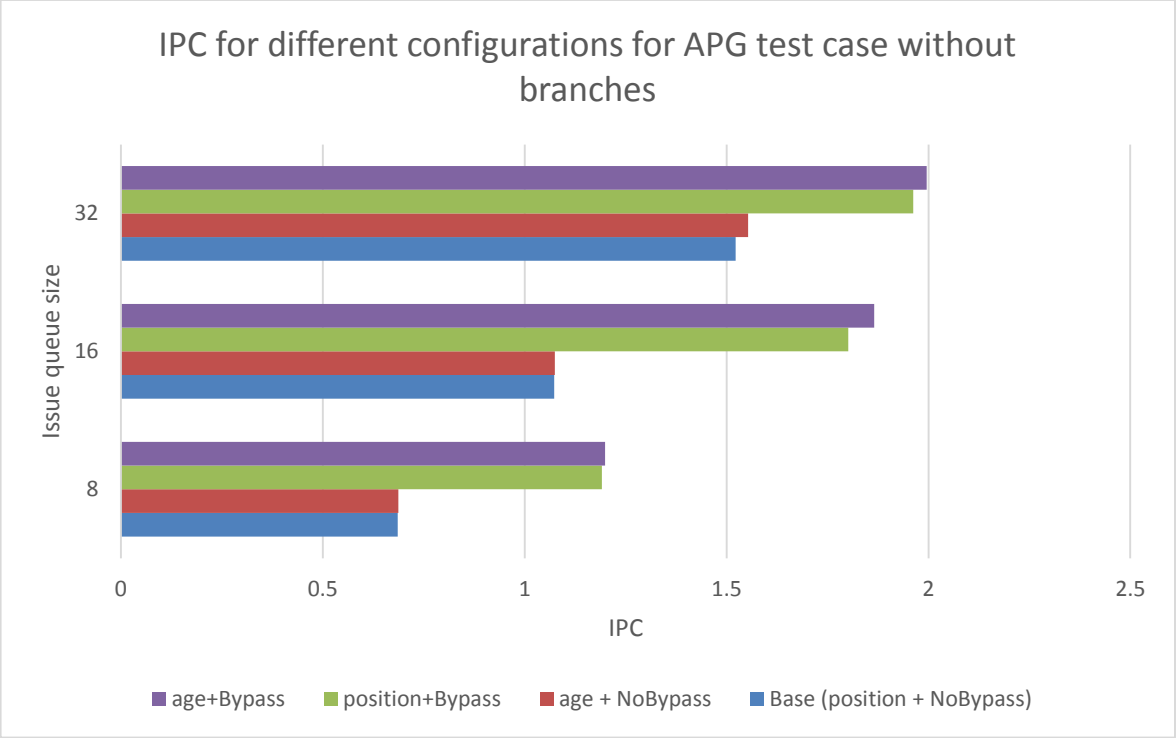

Figure 4.6: IPC in different processor configurations for APG test case without branches

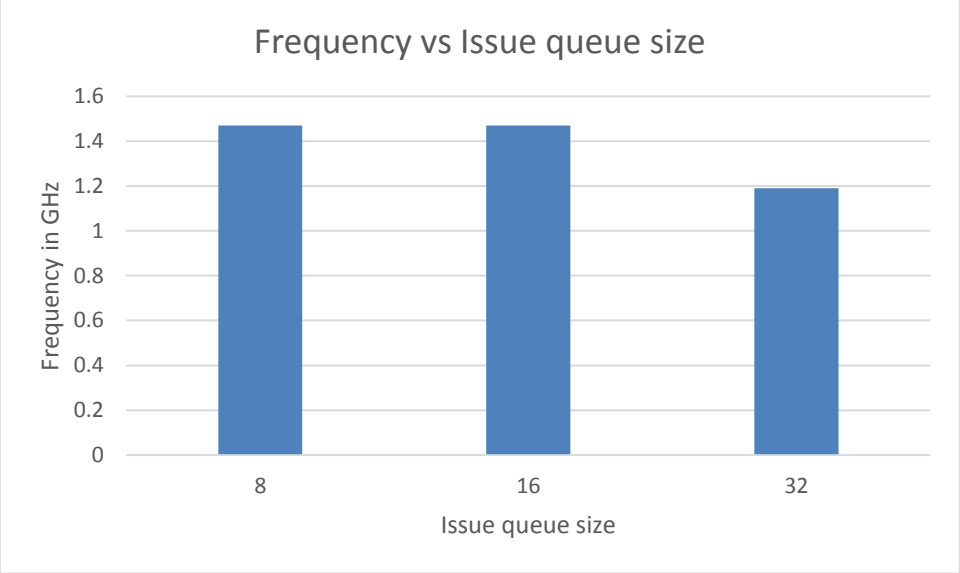

Figure 4.7: Frequency vs Issue queue size

Figure 4.7 shows the impact of issue queue size on clock frequency of the processor with position based selection policy and without bypass network. For the issue queue sizes of 16 and 32, critical path does not lie in the issue stage of the pipeline (it is in decode stage), hence the issue queue size did not affect the clock frequency. But, in the case of issue queue with size 32, critical path lies in issue queue and hence the frequency came down from 1.47 GHz to 1.19 GHz.

As discussed in chapter 3, two barrel shifters are added to the select logic which increases delay in that path. Figure 4.8 shows the impact of selection policy on processor's clock frequency. In all the three cases with age based selection policy, critical path lies in the issue stage of the pipeline. Hardware complexity of barrel shifter is increases with increase in issue queue size. That's why frequency degradation worsens as the size of issue queue is increased.

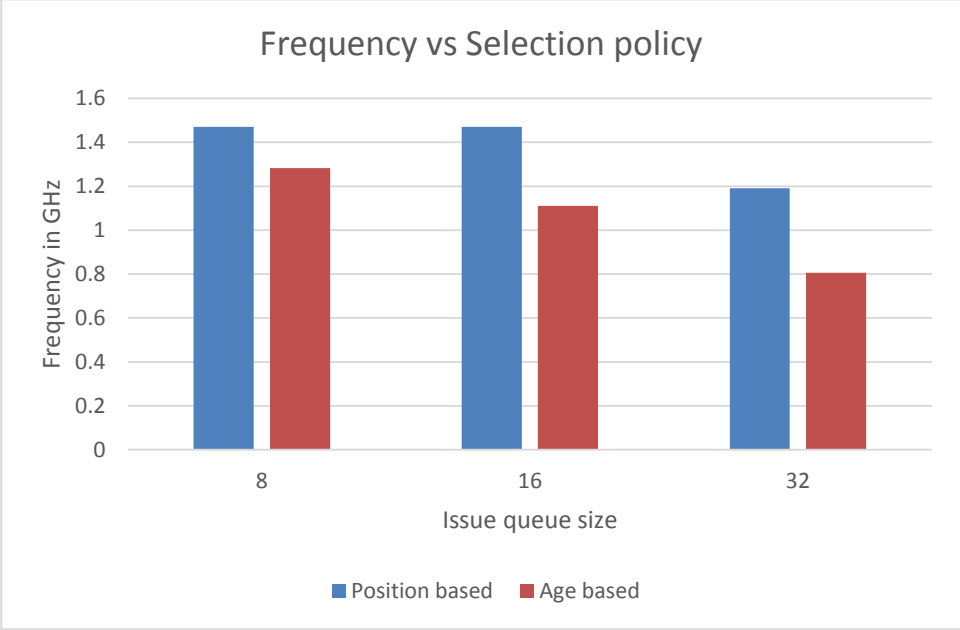

Figure 4.8: Frequency vs Selection policy used

In order to eliminate three cycle bubble between single cycle latency producer instruction and its consumers, select and wakeup must happen in the same cycle. This results in extra logic in issue stage. Figure 4.9 shows the impact of bypass network on the frequency. Also, execute and result bypassing should happen in same cycle too as discussed in section 3.11. This is another potential critical path in the processor.

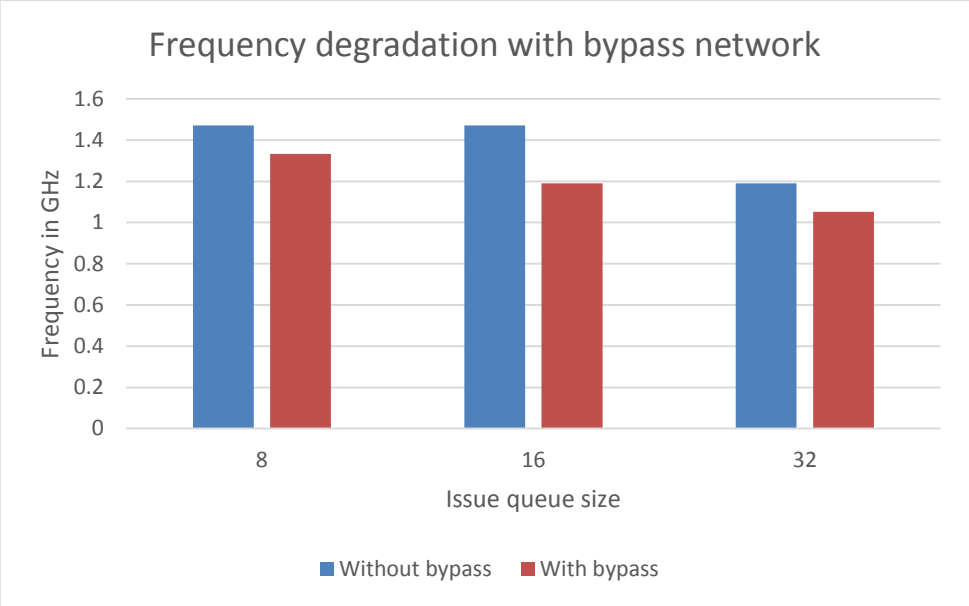

Figure 4.9: Impact of bypass network on frequency

In all the three cases with bypass network, critical path lies in the issue logic (select + wakeup) and the frequency is decreasing with increasing issue queue size.

Age based selection policy improved IPC by 1.3% and decreased clock frequency by 24% on average in all the different configurations. Whereas bypass network improved IPC by 69 and 64% for issue queue sizes of 8 and 16 respectively, 27% for issue queue size of 32. Clock was reduced by 15% and 20% for issue queue sizes of 8 and 16 respectively, 12% for issue queue size of 32. For APG test cases, bypass network increased the total throughput by 45 and 36% for issue queue sizes of 8 and 16 respectively, 14% for issue queue size of 32. Whereas age based selection policy reduced throughput by 24% on average. Although both selection policy and bypass network increased the IPC, their effect on throughput is completely different.

Again, IPC was evaluated using APG test cases which is not the accurate measure of performance.

4.3 Conclusion

In this thesis, a parameterized superscalar out-of-order processor is designed, implemented and verified. A CAM based speculative load store unit is designed, implemented and verified. Number of ALUs, issue queue size, immediate buffer size and memory queue sizes are

parameterizable. Impact of architectural techniques (selection policy and bypass network) on IPC and clock is evaluated. As the entire core was written in Bluespec which has good flexibility and parameterization support, it can also be used as a generic out-of-order platform for trying out new architectural techniques and evaluate their performance.

4.4 FUTURE WORK

Following features can be added to the processor designed and implemented in this thesis:

- Multi-level cache.
- Multi-threading and multi-core support.
- Extra functional units like FPU and SIMD unit.
- Memory management unit.
- Do functional unit clustering after new functional units are added to reduce the issue logic complexity.
- Run the actual benchmark programs and do the necessary optimizations to increase the throughput.

BIBLIOGRAPHY

[1] Hennessy, John L., and David A. Patterson. "*Computer architecture: a quantitative approach*." Elsevier, 2012.

[2] Andrew, Waterman, Lee Yunsup, Patterson David A., and Asanović Krste. "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0." *EECS Department, University of California*, Berkeley, 2014.

[3] Bluespec, Inc. http://www.bluespec.com, 2014

[4] Nikhil, Rishiyur. "Bluespec System Verilog: efficient, correct RTL from high level specifications." *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*. IEEE, 2004.

[5] Gruian, Flavius, and Mark Westmijze. "VHDL vs. Bluespec system verilog: a case study on a Java embedded architecture." *Proceedings of the 2008 ACM symposium on Applied computing*. ACM, 2008.

[6] Palacharla, Subbarao, Norman P. Jouppi, and James E. Smith. *Complexity-effective superscalar processors*. Vol. 25. No. 2. ACM, 1997.

[7] Tejero, Rubén Gran."*Non-speculative Enhancements for the Scheduling Logic*." Diss. Universitat Politécnica de Catalunya, 2010.

[8] Shen, John Paul, and Mikko H. Lipasti. "*Modern processor design: fundamentals of superscalar processors*." Waveland Press, 2013. Chapter 5.

[9] González, Antonio, Fernando Latorre, and Grigorios Magklis. "Processor microarchitecture: An implementation perspective."*Synthesis Lectures on Computer Architecture*"5.1 (2010): 1-116.

[10] Bluespec, I. "Bluespec System Verilog V3. 8." *Reference Guide* (2014).

[11] Xilinx Inc. http://www.xilinx.com/, 2014