ISSUE QUEUE OPTIMISATION OF OUT-OF-ORDER SUPERSCALAR PROCESSOR

A PROJECT REPORT

Submitted by

MOHAMMED UNNISA HULUVALLAY

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

MASTER OF TECHNOLOGY
in
ELECTRICAL ENGINEERING (VLSI)

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MADRAS

JUNE 2016

CERTIFICATE

This is to certify that the project entitled "**Issue Queue optimisation of out-of-order superscalar processor**", submitted by **Mohammed unnisa Huluvallay**, in partial fulfillment of the requirements for the award of the degree of MASTER OF TECHNOLOGY (VLSI) in Electrical Engineering, at Indian Institute of Technology, Madras, is a record of bonafide work carried out by her during the academic year **2015-2016**.

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

Date: 18th June, 2016

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 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.

Mohammed unnisa Huluvallay

ABSTRACT

Most of the today's computer architecture research is focussed around improving the performance of a processor. Instruction level parallelism is one among the processor design techniques that speed up the processor by allowing individual machine operations such as integer addition subtractions, floating point operations, memory operations to execute in parallel. As a result, most modern processors aim at getting better performance through parallel instruction processing. This project aims at improving the speed of the processor by processing three instructions simulteneously. It is implemented in Bluespec systemverilog, a high level HDL (Hardware Description Language) with good modularity, flexibility, and with good ease of testing and debugging and hence easy to add new design modules and features. This processor is included in open source cores development project at RISE lab, IIT Madras.

Following features are implemented in I-Class processor. i) Fetch width of 3 ii) Parameterised FTB (Fetch Target Buffer) size, entry ROB size, issue queue size, load store queue size etc iii) Merged Register file renaming iv) Operand forwarding and wake up logic v) Speculative load store unit to achieve out of order execution of load store instructions vi) Unified issue queue vii) Fetch target buffer to store fetched instruction block viii) Tournament branch predictor unit.

Keywords: *I-Class, Fetch target buffer, Superscalar processor, speculative load-store unit, unified issue queue, tree based priority encoder logic, merged register file renamer, tournament branch prediction .*

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter-1
Introduction

Computers have become essential component of our day to day life. Microprocessor is the heart of any computer which incorporates the functions of CPU in a single integrated circuit or a few integrated circuits. A processor is a programmable electronic device that accepts binary data as input, processes it as per the instructions stored in memory and provides the output result.

1.1 Generations of Microprocessor:

1.1.1 *First Generation:*

Microprocessors introduced in 1971-1972 comes under this category. They processed their instructions serially. Each instruction is fetched, decoded and executed serially. After execution of one instruction next instruction will be fetched by the microprocessor.

1.1.2 *Second Generation:*

Pipelined instruction processing and 16 bit arithmetic were introduced in this generation. Newer semiconductor technology was introduced in this generation which resulted in high speed and high chip densities.

1.1.3 *Third Generation:*

It was introduced in 1978 when the IC transistor count reached 2,50,000. Onchip cache was implemented and pipeline depth is also raised to more than 5 stages.

1.1.4 *Fourth Generation:*

These Processors can retire more than one instruction per clock cycle and the IC transistor count crossed million. Intel's 80960CA and Motorola's 88100 comes under this generation.

1.1.5 *Fifth Generation:*

The design of these processors exceeded ten million transistors. They implemented decoupled superscalar processing for the first time.

1.2 Classification of Microprocessors:

Microprocessors can be classified into three categories based on their characterstics:

a)RISC processors

b)CISC processors

c)Special processors

1.2.1 ***RISC processor***: RISC stands for Reduced Instruction Set Computer.

The distinguished characteristics of a RISC processors are:

- Few instructions
- Relatively less addressing modes
- All ALU operations done within the CPU registers
- Memory access allowed only for load and store instructions
- Hardwired control Unit
- Fixed Instruction length.

Some architectural features of RISCs are:

- Relatively large number of registers are present in the processing unit
- Good compiler support for translation of high level language programs into machine level language programs.
- Efficient instruction pipeline mechanism is employed
- Overlapped register windows are used to speedup procedure, call and return.

1.2.2 ***CISC processor*****:** CISC stands for Complex Instruction Set computer.

The distinguished characteristics of a CISC processors are:

- More than one cycle may be required to execute one instruction
- Relatively more addressing modes
- Microprogrammed control unit
- Variable instruction length
- Control transfer instructions.
- Different execution times for different instructions.
- Some instructions that perform specialized tasks.

1.2.3 ***Special Processors:*** These are application specific processors.Core Processors,Input/output processor ,Transputer,Digital signal processors comes under this category.

1.3 Classification of Processor Microarchitecture:

Processor microarchitecture can be classified along several dimensions.The most common ones are discussed here.

1.3.1 Pipelined/Nonpipelined Processors: Execution of each instruction is furthur divided into various phases and allow several instructions to be processed simulteneosly. Pipelining is implemented in almost all the processors.

1.3.2 In-Order/Out-of-Order Processors: An inorder processor processes the instructions in the same order as they appear in the binary code where as an out of order processor can process the instuctions in different order. The instructions which are independent of other instructions can be sent to execution earlier than those which are prior to them in the binary. The purpose of executing instructions out of order is to increase the instruction level parallelism by giving more freedom to hardware to choose which instruction to process in each clock cycle.

1.3.3 Scalar/Superscalar Processors: A scalar processor is the one which cannot execute more than one instruction in aleast one of its pipeline stages. The maximum throughput of a scalar processor is one instruction per clock cycle. Whereas a superscalar processor can execute more than one instruction per cycle in all its stages and hence can achieve a throughput greater than one.

VLIW(very long instruction word) processors are a special kind of superscalar processors. They can process multiple instructions in all its pipeline stages.

The special features of VLIW processor are:

- It processes the instructions in the same order as binary.
- The instructions to be executed in parallel is indicated by the binary code.
- Execution latencies are exposed to the programmer and hence constraints regarding the distance between particular types of instructions need to be satisfied in order to ensure correct execution.

1.3.4 Vector Processors: The ISA of a vector processor includes significant number of instructions that perform vector operations. Traditionally vector processors contained instructions that operate on long vector lengths.But now most processors include large set of instructions that operate on small vectors. These are often refered to as SIMD(single instruction multiple data) instructions.Now a days many processors are termed as vector processors but their support for vector instructions vary significantly among them.

1.3.5 Multicore processors: Processors containing more than one core are termed as multicore processors. A core is a unit that processes a sequence of instructions. Most of the current day processors have multiple cores.A multicore processor can process different sets of instructions parallelly and can allow them to syncronize with each other. Multicore processors normally support interconnects among the cores to communicate with each other and to share data among them.

1.3.6 Multithreaded processors: A Multithreaded processor can support more than one thread on atleast one of its cores.Multicores are different from multithreaded ia a way that multicores use different hardware resources whereas multithreaded share same hardware resources.

The Processor which we aim to implement is out of order, superscalar with a fetch width of 3 and pipelined one.It is called I-Class processor which is included in open source cores project in Rise Lab,IIT Madras.

CHAPTER-2

OVERVIEW OF PIPELINE AND ITS HAZARDS

This chapter deals with the details of processor architecture and the basics of pipelining. Generally pipeline stages of a processor are fetch,decode,execute and write back.

These are not necessarily the pipeline stages in every processor.A particular implementation can divide each of these into several stages or can group many of them into one single stage.The latency of each stage may vary slightly. Its quite common to add buffers between the pipeline stages which allows the processor to hide some of the stalls like operand not ready etc.

2.1 Pipeline Basics:

Pipelining is a technique through which instruction level parallelism can be employed in a processor.The instruction cycle is split into several steps so that different steps can be executed parallelly and hence instructions can be processed concurrently rather than serially. Pipelining cannot reduce instruction latency as it has to go through all the stages of pipeline but increases throughput by allowing multiple instructions to get processed concurrently.

2.1.1 Design considerations:

- **Speed:** Pipelining keeps all the stages of the processor busy all the time and hence speed of the processor increases.Hazards reduces the speed of the pipeline.
- **Economy:** Pipelining enables complex operations to perform more economically than adding complex circuitry.
- **Predictability:**Comparitively it is easier for the non pipelined processor to predict the exact sequence of instructions.

2.2.2 Overview of pipeline flow:

CPU is driven by a clock.Each clock period need not perform the same task.There are so many effects that cannot happen at the same time. Pipelining modulates those effects as different stages of pipeline.

Number of steps: Number of steps varies with machine architecture. IBM proposed the terms fetch,decode,execute which has become common.

Some processors like Intel pentium 4 have pipelines as long as 10 and 20 stages also.

The Xelerated X10q Network Processor has a pipeline of thousand stages.

As the number of pipeline stages increases each step can be implemented using a simple circuitry. This enables the processor clock to run faster. Such pipelines are termed as superpipelines.

The classic RISC pipeline comprises:

1. Instruction fetch
2. Instruction decode and register fetch
3. Execute
4. Memory access
5. Register write back

The general description of each of the stages is given below.Details of our I_Class processor pipeline stages is discussed in later sections.

- *Instruction fetch*:Getting single or group of instructions from memory.
- *Instruction Decode*:This stage of pipeline performs decode operation to find the addressing mode of instruction and extract the immediate operands.
- *Execute/effective address calculation*: This stage consists of Arthematic and logic unit and a bit shifter.It may include multiply and divide unit. Calculate the effective address in case of memory access instructions. Performs ALU operations in case of arthematic instructions.
- *Memory access*:Data memory is accessed in case of load/store instructions.
- *Write back*:Write the calculated result or loaded memory value to the destination register.

The figure 2.1 shows the implementation details of five stage pipeline discussed above. Table 2.1 shows the overlapping of instructions in the five stage pipeline. The coloured column indicates parallel execution of five instructions.

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

Table 2.1: Five stage pipeline flow diagram

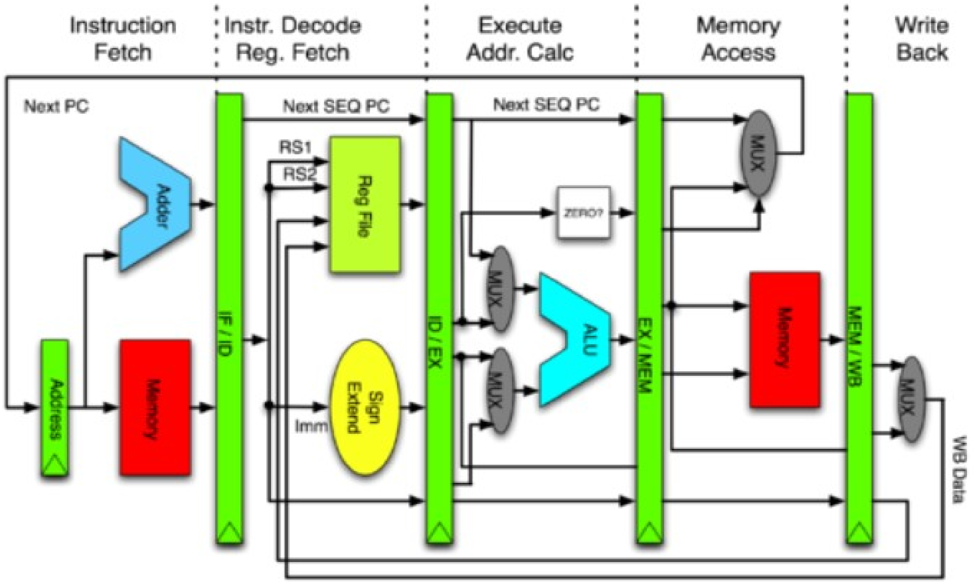


Figure 2.1: Five stage pipeline implementation

If there are N instructions and M pipeline stages then the number of clock periods required for the execution=M+N-1.We cannot increase the pipeline stages infinitely because of the following reasons.

- ***Pipeline overhead***: Interstage buffer delay (set up and hold times) and clock skew (difference in arrival of the clock at different registers) resulted in pipeline overhead. As we increase the number of pipeline stages this delay becomes dominant and increases the clock period.
- ***Pipeline latency***: The pipeline frequency depends on the stage with maximum delay.If that stage cannot be split furthur,there is no use of splitting the other stages of the pipeline.
- ***Pipeline stalls***: Branch mispredictions and data dependencies results in Pipeline stalls/flushes which incur huge penalties in the case deep pipelines.

A pipeline stall stops the instruction pipeline because of pipeline hazards.

2.2 Pipeline Hazards:

Hazard is a situation in which the next instruction is prevented from getting executed in its designated clock period. There are basically three kinds of hazards which are discussed below.

2.2.1 Structural Hazards: They arise due to resource conflicts.When a single piece of hardware is used in more than one stage, many instructions may require it at the same time which it cannot support leading to stall in the pipeline.

For instance if we use single unit of memory both for instructions and data then fetch stage might be fetching instruction when memory access stage wants to read/write data for load and store instructions. This can be solved by splitting the memory into orthogonal units like instruction cache and data cache.

2.2.2 Data Hazards: This arises when write and read operations occur in different order in the pipeline than in the binary. Data hazard can occur in three situations.

1.Read after Write data dependency (RAW): This occurs when an instruction is supposed to read a location after it is written by the earlier instruction,but in the pipeline write operation is happening after the read that means the read instruction is getting the stale data.

2.Write after Read data dependency (WAR): This is the reverse case of RAW dependency. According to the code write should occur after read but read occurs before write in the pipeline.

3.Write after Write data dependency (WAW): This situation occurs when two instructions are supposed to write the same location but write opeartion is happening out of order.

Consider the example

```
R2 <- R1 + R3
R4 <- R2 + R3

R3 <- R1 + R5
```

Here R2 should not be read before it is written by adding R1 and R3.While R3 should not be written before being read to write R4.

2.2.3 Control Hazards: Control hazards occur due to branch mispredicrions. Sometimes the processor will not know the outcome of a branch and it predicts based on past history which need not be correct all the time.On branch mispredictions the pipeline needs to be flushed and a new instruction should be fetched from instruction cache.

2.3 Eliminating Hazards:

2.3.1 Add extra hardware: This technique resolves resource conflicts.If a single unit of hardware has to be used twice in an instruction, we should maintain a replica of it so that two instructions can access it at the same time.

2.3.2 Pipeline Bubbling: Pipeline bubbling helps in preventing all kinds of hazards. After fetching instructions from instruction cache the control logic finds whether a hazard could occur. If it comes true the control logic introduces NOP(no operation) in the pipeline. Hence before the next instruction (ie., the instruction which would have caused hazard) comes for execution the earliar instruction would have completed and the hazard can be prevented. But this technique introduces delay in the pipeline.

2.4 WAYS OF EXPLOITING INSTRUCTION LEVEL PARALLELISM:

There are a wide variety of techniques to improve instruction level parallelism. Some of them are discussed below.

2.4.1 Out of order execution: Instructions which are independent of earliar instructions can be scheduled between the producer and consumer instructions so that the pipeline continues without introducing NOPs. All the pipeline stalls due to RAW hazards are due to data dependencies among them. We can exploit the parallelism of instructions to keep the pipeline full. Such stalls can be avoided if the producer-consumer instructions are separated by number of cycles equal to the latency of producer instructions.

Consider the examle

```
DIV R5 R1 R2

SUB R6 R5 R3

ADD R4 R2 R1

MUL R7 R4 R2

LOAD R10 A
```

In the above sequence of instructions, DIV operation takes more number of clock cycles and SUB instruction should stall for DIV operation to finish but ADD instruction has both its operands ready and MUL should wait for ADD to finish to resolve data dependency.Here the ADD can be scheduled between DIV and SUB and Load can be scheduled before MUL to avoid stall in the pipeline. As soon as R4 is available MUL can be sent for execution.Hence the overall clock frequency is improved since the out of order execution is hiding the latency caused by DIV and MUL instructions.

But this out of order execution of instructions results in the following impediments in the pipeline.

- *Probability of WAW and WAR hazards*: As the instructions are now executed out of order there is a possibility that an instruction present lower in the binary code writes a register before it is read/written by an instruction present above in the binary. This can be eliminated by renaming the registers before sent for execution.

- *Handling precise Exception:* Out of order instruction execution may cause imprecise exceptions ie.,when an exception is arised the state of the processor is not exactly the same as if the instructions were executed in program order.We can get precise exception by delaying the exception notification till all the instructions above the instruction generated exection are finished.
- *Managing speculative instructions*: Processors using branch predictors issues instructions after branch eventhough the branch outcome is not known.Such instructions are termed as speculative instructions.In order to maintain precise exception,the register file should be updated only when the instructioin is no longer speculative.This can be done if the instructions write the reg file in the same order as in the binary code. This stage of pipeline is called 'commit' stage.

Now we see how the above discussed methods are actually implemented in hardware.

2.4.1.1 Register rename:

Register rename is generally seen in out of order processors. In an out of order processor, instructions are reordered to improve the amount of instruction level parallelism that is exploited. But this out of order execution gives incorrect results because of data dependencies among the instructions which force them to get executed in program order. Register renaming is a well known scheme for out of order execution. It effectively resolves the data dependencies by allocating a free register to every instruction. The destination register of every instruction is given a new name when it comes to map stage.This eliminates Write after Read and Write after Write hazards.

Consider the example given below

```
ADD R1 R2 R3
SUB  R5 R1 R4
MUL R1 R7 R8
DIV R11 R1 R12
STR R1 6(R10)
```

WAW hazard can occur if MUL is sent for exection before ADD. WAR hazard can occur if MUL is sent for execution before SUB.similarly STR and DIV are also WAR hazard pairs.The renamed instructions look like the following:

ADD R1 R2 R3

SUB T5 R1 R4

MUL T1 R7 R8

DIV T11 T1 R12

STR T2 6(R10)

T1,T2,T5,T11 are introduced to store the short term results of MUL,STR,SUB,DIV instructions WAW hazard is eliminated by using T1.WAR hazard between MUL and SUB is also eliminated using T1.WAR hazard is eliminated by using T2. Hence the false dependencies are removed.

2.4.2 Operand Forwarding:

Sometimes it happens that the data is available somewhere but not exactly where we want.In such situations we create extra paths to transfer the data to the location where it is needed. This technique sees advantages over other techniques because it does not slow down the processor and does not affect the semantics of the instruction set.

As we have seen that the data hazards stall the pipeline till the result of the producer instruction is written into the destination register. Instead of waiting for that, result calculated in the MEM and EXE stages of the pipeline can be sent to the Id stage of consumer instruction in the same cycle.

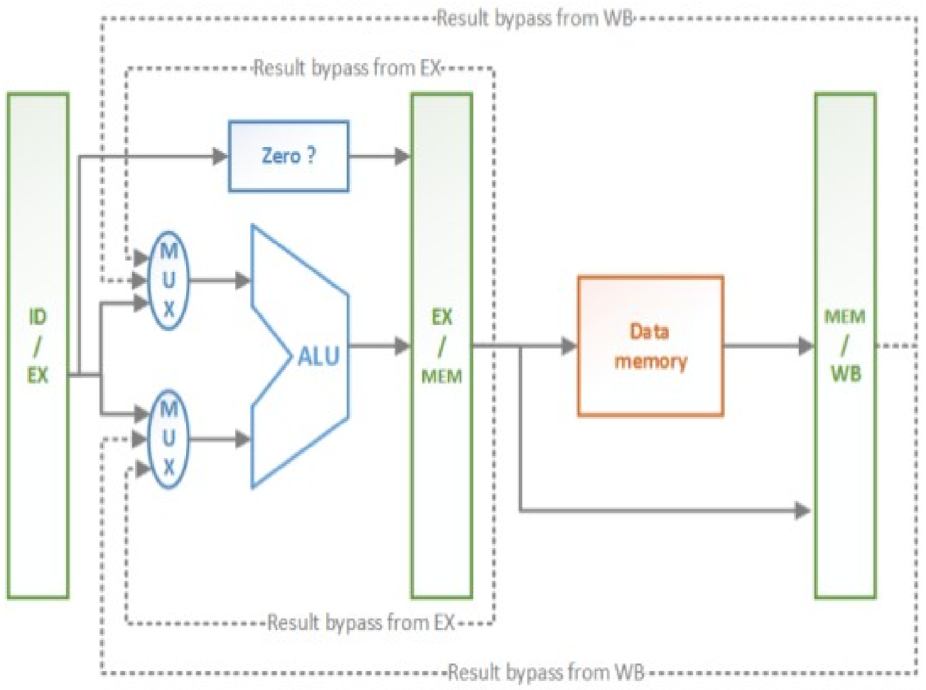


Figure 2.2: Operand Forwarding

2.4.3 Instruction Prefetcher:

Instructiion level parallelism can be improved effectively by employing prefetchers. Cache miss stalls the instruction pipeline flow for one or more clock cycles.To nullify this effect, prefetchers fetch instructions before they are needed. These prefetched instructions are enqued into a special queue named instruction queue. Typically the instruction queue size is around 16. By implementing instruction prefetcher,we can hide the memory latency and memory appears to have a performance equal to processor register.

To hide latency effectively, a prefetcher should:

1.Predict the address of memory access for being accurate.

2.Predict when to issue a prefetch ie., it must be timely

3.Effectively select where to place the prefetched data and which other data to replace.

Various kinds of instruction prefetchers are discussed here

2.4.3.1 Next line prefetcher:

Next line prefetching is simplest kind of instruction prefetching which is widely used in current day processors. Since code is laid sequentially in consecutive memory locations,over half of the cache look up is for sequential address. The logic of generating sequential address is relatively simple and is easy to incorporate it into processor and cache hierarchy.

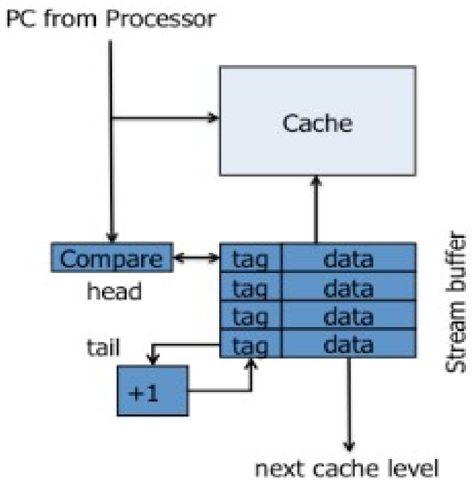


Figure 2.3 : A Next line prefetcher

Figure 2.3 dipicts the anatomy of next line prefetcher used in current day processors. An instruction prefetch buffer strores the instruction cache blocks prefetched from lower cache hierarchy levels.Each time a block from the stream buffer/prefetch buffer is explicitely required by the processor, it is sent to cache and a new block is fetched from memory.

2.4.3.2 Fetch Directed Prefetching:

Next line prefetchers are quite simple to implement but only half of the cache lookups are sequential. Control transfer instructions break the sequential flow and create discontinuities. Hence a prefetcher which predicts future control flow is needed.

Branch predictor directed prefetchers use branch predictors to detect the control flow. Branch predictors recursively make predictions to explore instruction addresses for prefetch. Fetch directed instruction prefetching is one of the best branch prediction based instruction prefetcher.

Fetch directed architecture implements a decoupled branch predictor and an instruction cache. The decoupled branch predictor runs ahead of execution pc and make predictions. The other components of this architecture makes use of these predictions and helps supplying instruction addresses to the fetch engine. The predicted instruction addresses are stored in the FTQ whose corresponding instructions are fetched from L2 cache and then placed in a fully associative buffer.The buffer is accessed by the fetch unit in parallel with the instruction cache. FDIP utilizes idle instruction cache ports to probe the cache for the addresses in the FTQ to see whether they are already present and inserts only the missing addresses in the prefetch instruction queue(PIQ).

The anatomy of FDIP is shown below.

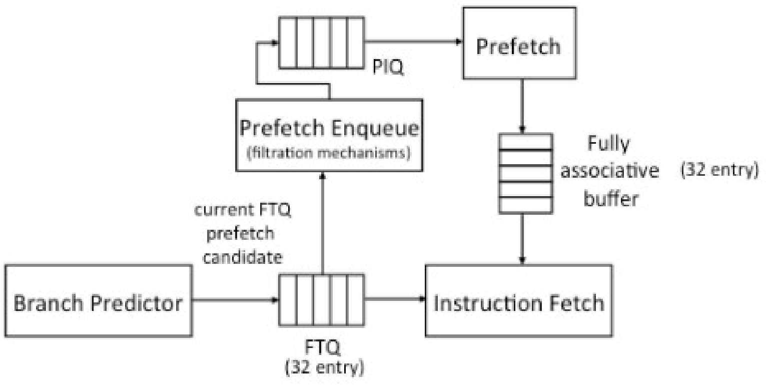


Figure 2.4: Fetch-directed instruction prefetching

2.4.3.3 Discontinuity prefetching:

Next line and fetch directed prefetchers fail to address control flow discontinuities. Prefetching instructions when there are function calls, taken branches and traps is a great challenge. Discontinuity predictor addresses this discontinuity to some extent. It maintains a table of fetch discontinuities which maps the program counter of the block containing the taken branch to its corresponding branch target. Even though this predictor is simple and easy to implement, it can only bridge single fetch discontinuity. Further the coverage is limited since the table can record single discontinuity per cache block. In reality there can be multiple taken branches in a single instruction block.

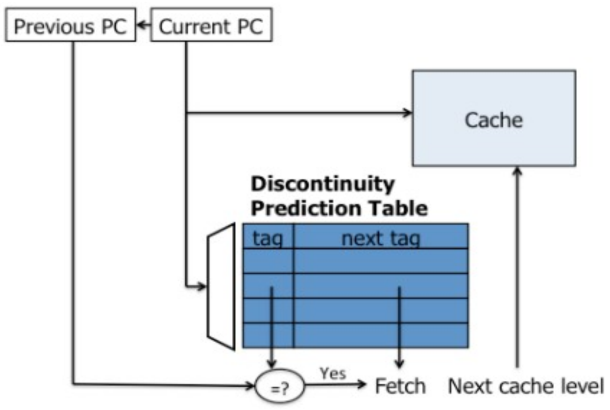


Figure 2.5: A Discontinuity predictor

2.4.3.4 Return address stack directed instruction prefetching:

The pitfall of Fetch directed branch predictors is that they cannot past loop return branches. Dicontinuity prefetchers address these limitations but they rely on a single PC to predict an upcoming fetch discontinuity. Hence both of these techniques fail when there are multiple control flow paths out of a particular cache block.

Return address stack directed prefetching makes use of additional program context information to improve lookahead and prediction accuracy. This kind of prefetching is based on two obsravations:

1.Program context as recorded in the call stack correlates with L1 instruction cache misses.

2.The return address stack briefly summarizes the program context.

Proactive instruction fetch, Temporal instruction fetch streaming, Prescient fetch are some of the other notable instruction prefetchers.

Prefetching needs to be done to data in order to hide the latency due to data miss patterns. Some of the most common data prefetchers are *stride and stream* prefetchers and *Address correlating* prefetchers.

1.Stride and stream prefetchers generalize next line instruction prefetching concepts to data.

2.Address correlating prefetchers are specially designed to target pointer chasing access patterns of data structures.

To exploit instruction level parallelism branch prediction is must. Otherwise conditional branch instructions causes stall in the pipeline.

2.4.4 Branch scheduling:

Conditional branch instructions decide whether to take a branch or not based on run time data. This can be computed only in the execution stage of the pipeline. It takes atleast 10 clock cycles from the time a branch is fetched till it gets executed. So waiting for the result of that computation is not correct option in current day processors. Branch prediction can be done statistically,dynamically or a combination of both.

2.4.4.1 Static branch prediction:

Static prediction collects the most frequent outcome of each branch and uses it as a prediction.Static prediction is the simplest form of branch prediction as it requires only few bits(one may be enough) to say whether the branch should be taken or not.

2.4.4.2 Dynamic branch prediction:

Dynamic prediction is based on a particular hardware that stores the previous history of running application and uses it to predict every branch.A simple and quite commonly used branch predictor contains a table of 2^n entries of two bit each. The table is indexed with n least significant bits of the program counter. The corresponding entry is then used to predict whether the branch should be taken or not and it is updated once the branch outcome is available. In this way the most recent history of the branch is reflected. The two bit entry implements a finite state machine which is often referred to as saturating counter. The prediction of each conditiional branch is made using the past history of the same branch (ignoring alaising) and hence this is named as local branch predictor.

The branches that are almost always taken will be in 11 state wheeas the branches that are almost always not taken will be 00 state.The branches which recently changed their bais will be in 10 or 01 state.

Since the table contains finite number of entries, more than one branch uses the same entry which sometimes gives wrong prediction.This is called alaising effect.

The anatomy of Dynamic branch predictor is shown below.

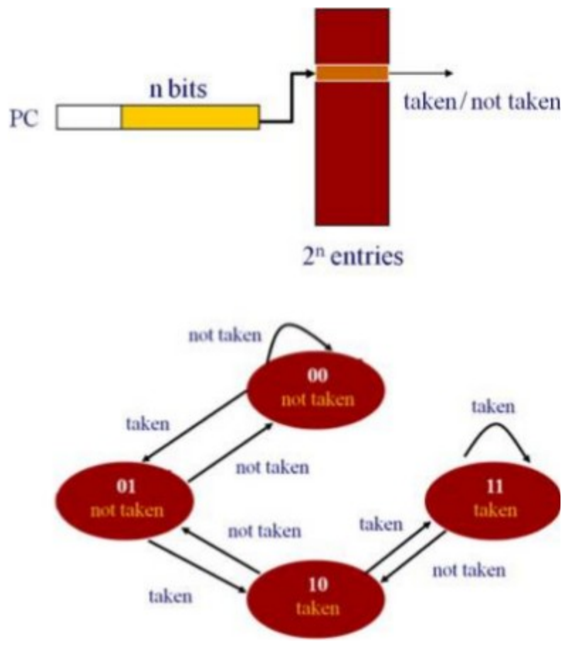


Figure 2.6:   A Dynamic branch predictor

2.4.4.3 Gshare predictor:

Current microprocessors are using correlating predictor called gshare predictor to furthur reduce the branch misprediction penalty. Correlating predictor not only uses history of a branch itself but it makes prediction considering neighbouring branches also.

It contains a register called global branch history register which stores the branch outcome of most recent branches.The PC of the branch is combined with this history through a hashing function to generate an index to the table 2^n entries.Each entry is a 2 bit saturating counter. The prediction is made by this entry and is updated with the branch outcome in the same manner as for the local predictor discussed above.The anatomy of Gshare predictor is shown below.

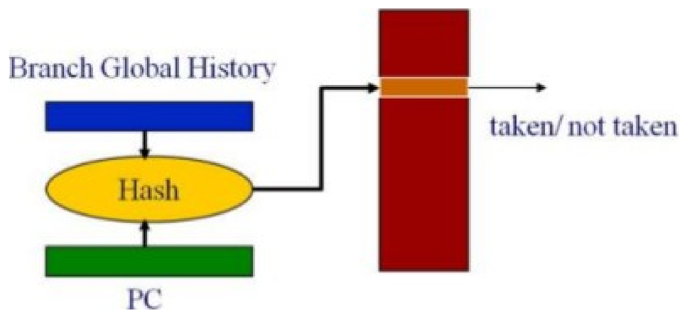


Fig AGshare correlating predictor

There are so many other predictors like the hybrid predictors,L-TAGE predictor,PPM based predictors which use complex circuitry to furthur reduce branch misprediction penalty.

Some of the above discussed methods like operand forwarding, branch scheduling, out of order execution, register renaming are implemented in I_Class processor and the details of implementation are explained in later sections. Instead of instruction prefetcher, a slightly different method is used, which is explained below.

2.5 Implementation of Fetch Target Buffer in I_Class processor:

When the fetch unit requests the icache with a PC, it responds with a block of eight consecutive instructions which are stored in a special buffer named Fetch Target Buffer. First three amoung these eight instructions are send to decode unit in a way discussed in section 4.1. If the predicted program counter from branch predictor in the next cycle is the same as icache address of the fourth instruction in the FTB then the next three instructions are send to decode unit otherwise new instruction block is fetched from icache with the predicted program counter and stored in FTB and the same cycle is repeated.

Chapter-3
RISC-V INSTRUCTION SET ARCHITECTURE

This chapter deals with the basics of RISCV ISA and a short discription of Bluespec system verilog in which the whole design is implemented.

3.1 RISCV ISA:

RISCV is an insruction set architecture that was originally designed to support education and computer architecture but now became a standard architecture for industrial applications. RISCV ISA is like a base integer ISA which is must in any implementation in addition to optional extensions to base ISA. The base ISA is similar to early RISC processors except that it supports variable length instruction encodings. Our goals in defining RISCV are the following:

- Its an open instruction set architecture available both for industry and academia.
- It is suitable for native hardware implementation not just simulation and binary translation.
- It supports revised 2008 IEEE-754 floating point standard.
- Its completely virtualizable and supports hypervisor development.
- It supports user level extensions and specialized variants.
- It encourages both 32 bit and 64 bit variants for hardware implementations and operting system kernals.
- It allows efficient implementation but avoids over architecting for a particular design.

The base ISA supports a limited set of instructions sufficient to provide a reasonable platform for compilers, linkers, operating systems and hence provides a convinient base around which advanced processor ISAs can be built.The width of integer registers and corresponding user address spaces are different for different base ISAs. RV32I and RV64I are two base integer variants that provide 32 bit and 64 bit user level address spaces respectively.

RISCV can support extensive customization and specialization. The base ISA can be extended with optional instruction set extensions but the instructions in base ISA can't be redefined.

RISCV instructions are categorized into two types.

1.Standard extensions

2.Non standard extensions

Standard extensions should not conflict with other extensions whereas non standard extensions are highly specialized and may conflict with other standard or non standard extensions.

The base ISA is prefixed by RV32 or RV64 depending on register width and is called I. It supports integer load and store instructions, integer computational instuctions and control flow instructions.

M standard extension: The extension M adds instructions for integer multiplication and division operatons.

A standard extension: The standard atomic instruction extension A extends base integer ISA by adding instructions that atomically read, modify and write memory for interprocessor synchronization.

*F standard extension:*The extension F is for single precision floating point operations.It supports single precision load and store instructions, single precision computational instructions etc.

D standard extention: D is a double precision floating point extension and it supports double precion floating point load store and computational instructions.

An integer base along with these four standard extensions is named the abbrevation G and serves as the general purpose scalar instruction set.

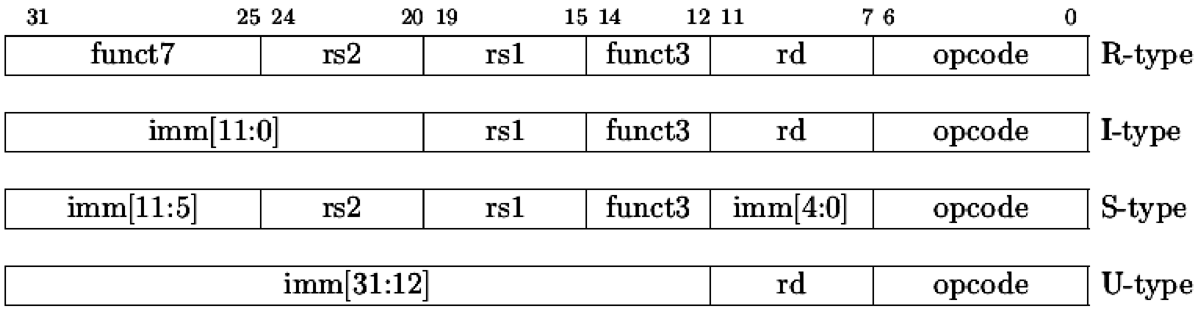
3.2 Exceptions, Traps and Interrupts:

- Exception refers to an unusual condition occuring at run time.
- Trap refers to the synchronous transfer of control to a supervisor environment due to an exceptional condition occcuring within RISCV thread.
- Interrupt is similar to trap but it refers to the asynchronous transfer of control toa supervisor environment caused by an event occuring outside the current RISCV thread.

3.3 RV32I Base Integer Instruction Set:

There are 31 general purpose registers named x1 to x31 to hold integer values. Register x0 is hardwired to zero. Register x1 is assigned to hold the return address on a call. The register width is 32 for RV32 where as it is 64 for RV64. The program counter holds the current instruction address.

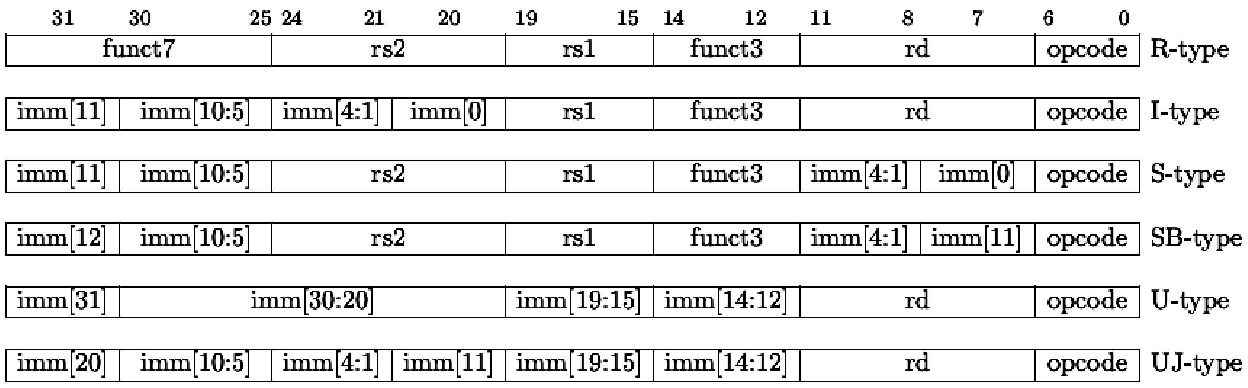
3.3.1 Base Instruction Formats: There are four core instruction formats in the base ISA. All instructions are of fixed 32 bit length. Each instruction is aligned on a four byte boundary in memory. Instruction misaligned exception occurs if the pc is not four byte aligned on an instruction fetch.



31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[31:12]				rd	opcode	U-type

Table 3.1: RISCV base instruction formats

3.3.2 Immediate Encoding Variants: There are two other varieties of instruction formats lik SB and UJ. RISCV instruction formats based on immediate encoding variants is shown below.



31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11]	imm[10:5]	imm[4:1]	imm[0]	rs1	funct3	rd		opcode	I-type
imm[11]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[0]	opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	SB-type
imm[31]	imm[30:20]			imm[19:15]	imm[14:12]	rd		opcode	U-type
imm[20]	imm[10:5]	imm[4:1]	imm[11]	imm[19:15]	imm[14:12]	rd		opcode	UJ-type

Table 3.2: RISCV base instruction formats showing immediate variants

3.3.3 Integer computational instructions: Integer computational instructions are either encoded as register immediate operations or register register operations.The destination register is rd for both of them.

3.3.3.1 ***Integer register immediate instruction:***All these operations are performed on a register and sign extended immediate number.

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

ADDI adds the 12 bit sign extended immediate to register rs1.

SLTI stands for Set Less Than Immediate. SLTI places 1 in rd if rs1 is less than sign extended immediate,othervise it keeps 0 in it.

SLTI treats both the numbers as signed integers where as SLTIU treats them as unsigned numbers.

ANDI/ORI/XORI perform logical operatios like and,or and xor respectively on register rs1 and sign extended 12 bit immediate and places the result in rd.

3.3.3.2 ***Integer register register operations:*** All operations read the registers rs1 and rs2 as source operands and writes the result in destination register rd.

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

ADD and SUB perform addition and subtraction action.
The operation of SLT and SLTU is same as discussed above but here they compare registers rs1 and rs2 instead of immediate value.
SLL/SRL perform logical left and right shift operstions where as SRA perform arthematic right shift.They shift the value in register rs1 by a value equal to the lower 5 bits of register rs2.

3.3.3.3 ***NOP Instruction:*** NOP just advances the pc without changing any register values.

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
0	0	ADDI	0	OP-IMM

3.3.4 Control transfer instructions: It supports two types of control transfer instructions. They are unconditional jump and conditional branches.

3.3.4.1 ***Unconditional jumps:*** The JAL instruction stores the address of the instruction following the jump instruction in the register rd.The jump target address is formed by adding pc to the sign extended offset.It uses UJ type encoding format.

31	30 21	20	19 12	11 7	6 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1	10	1	8	5	7
	offset[20:1]			dest	JAL

The indirect Jump instruction JALR uses I type encoding format. The jump target address is formed by adding the 12 bit signed I immediate to the value in register rs1 and then making the least significant bit of the result to zero. JALR stores the address of the next instruction following the jump instruction in the register rd.

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	0	dest	JALR

3.3.4.2 ***Conditional branches:*** All conditional branch intructions use SB instruction encoding format. The 12 bit immediate encodes signed offsets in multiples of two and is added to the program counter to get the target address.

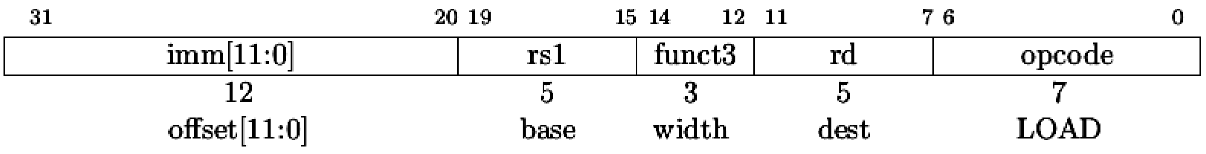
31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
	offset[12,10:5]	src2	src1	BEQ/BNE	offset[11,4:1]		BRANCH
	offset[12,10:5]	src2	src1	BLT[U]	offset[11,4:1]		BRANCH
	offset[12,10:5]	src2	src1	BGE[U]	offset[11,4:1]		BRANCH

BEQ and BNE takes the branch if the register values of rs1 and rs2 are equal or unequal respectively.

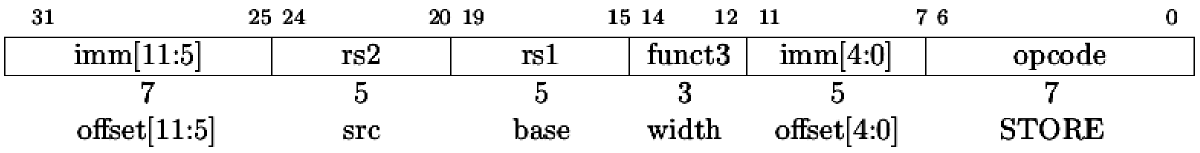
BLT and BLTU takes the branch if rs1 is less than rs2 with signed and unsigned comparision.

BGEU and BGE takes the branch if rs1 is greater than or equal to rs2 with unsigned and signed comparision.

3.3.5 ***Load and Store instructions:*** Load and Store instructions are used to transfer data between registers and memory. Loads use I type instruction encoding where as stores use S type instruction encoding.
Loads copy data from memory to register rd where as stores copy data from register rs2 to memory. The value in register rs1 is added to sign extended 12 bit offset to get effective byte address.



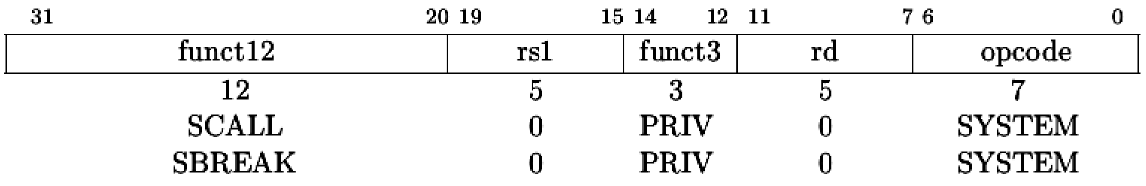
31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD



31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE

3.3.6 ***System Instructions:*** These are encoded using I type instruction encoding. These are used to access system functionality that require previliged access.

1. *SCALL and SBREAK Instructions:* SCALL instruction makes a request to operating system environment. *SBREAK in*struction transfers the control back to debugging environment.



31 20	19 15	14 12	11 7	6 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
SCALL	0	PRIV	0	SYSTEM
SBREAK	0	PRIV	0	SYSTEM

3.4 RV64I Base Integer Instruction Set:
RV64I supports user address space of 64 bit.

3.4.1 Integer Computational Instructions: Register immediate and register operations comes in this category.

3.4.1.1 ***Integer Register Immediate Instructions:***

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDIW	dest	OP-IMM-32

ADDIW is a 64 bit instruction that adds sign extended 12 bit immediate to register rs1 and stores the result in destination register rd.

31 26	25	24 20	19 15	14 12	11 7	6 0
imm[11:6]	imm[5]	imm[4:0]	rs1	funct3	rd	opcode
6	1	5	5	3	5	7
000000	shamt[5]	shamt[4:0]	src	SLLI	dest	OP-IMM
000000	shamt[5]	shamt[4:0]	src	SRLI	dest	OP-IMM
010000	shamt[5]	shamt[4:0]	src	SRAI	dest	OP-IMM
000000	0	shamt[4:0]	src	SLLIW	dest	OP-IMM-32
000000	0	shamt[4:0]	src	SRLIW	dest	OP-IMM-32
010000	0	shamt[4:0]	src	SRAIW	dest	OP-IMM-32

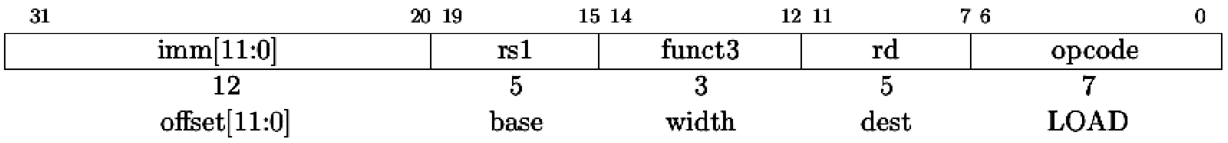
SLLI stands for logical left shift and it shifts zeroes into the lower bits.
SRLI stands for logical right shift and it shifts zeroes into the upper bits.
SRAI stands for arthematic right shift and original sign bit is extended to vacated upper bits.
RV64I supports SLLIW, SRLIW, SRAIW that are analogously defined but operate on 32 bit values and signed 32 bit results.

3.4.2 *Integer register register operations:*

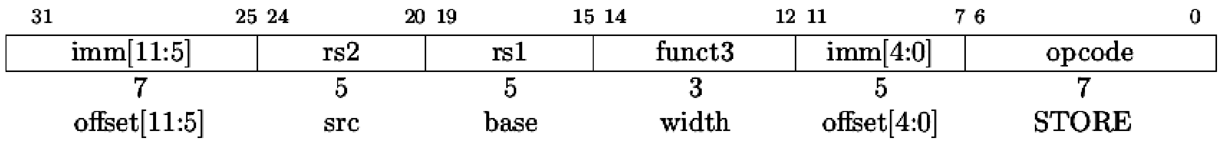
31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SRA	dest	OP
0000000	src2	src1	ADDW	dest	OP-32
0000000	src2	src1	SLLW/SRLW	dest	OP-32
0100000	src2	src1	SUBW/SRAW	dest	OP-32

ADDW and SUBW are RV64I instructions that are defined autonomously to ADD and SUB but operate on 32 bit values to get 32 bit result. As discussed above SLL,SRL,SRA perform shifting operations on the values in register rs1 by an amount equal to lower 6 bits of register rs2.
SLLW, SRLW and SRAW are RV64I only instructions but operate on 32-bit values and produce signed 32 bit results.The shift is obtaines by rs2[4:0].

3.4.3 *Load and Store Instruction:*



31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD



31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE

The LW instruction loads a 32 bit data from memory and sign extends that to 64 bit and then stores it in register rd.
The LD instruction loads a 64 bit value from memory into destination register rd.

In addition to these instructions there are some additional instructios called system instructions like RDCYCLE to count the number of clock cycles, RDTIME to write the clock real time that has passed from an arbitrary start time in the past an RDINSTRET to write the number of retires instructions from an arbitrary start point in the past.

So far we have seen the instruction varieties in base RV32I and RV64I. RV128I is also defined to support 128 bit user address space. In addition to these base ISA instructions standard extensions like M,A,F and D support additional instructions for multiplication,division,single precision,double precision floating point operations.

All the Instruction Formats shown above are taken from " THE RISCV INSTRUCTION SET MANUAL VOLUME 1: USER LEVEL ISA."

3.5 BLUESPEC SYSTEM VERILOG:

Bluespec system verilog abbrevated as BSV is a hardware design language used in the design of ASICs, FPGAs and systems.BSV is used accross several applications like memory subsystems,processors, signal processing accelarators, multimedia and communication codecs and processors, DMAs and data movers etc.

3.5.1 APPLICATIONS OF BLUESPEC

BSV is used in many design activities as discussed below.

- **Executable specifications:** For most of the current complex systems a specification written in human language is likely to be imprecise and sometimes self contradictory in its requirements. Using precise semantics BSV addresses these concerns.
- **Design and Implementation:** BSV enables designing complex subsystems at a much higher level of abstraction and with better maintainability.
- **Virtual platforms:** Current day chips are dominated by the complexity of the software that runs on them. Waiting for chips before developing software is no longer acceptable. Virtual platforms enable testing and software development to begin as early as possible. As we know that BSV is synthesizable, Virtual platforms written in BSV run on FPGAs at much higher speeds than traditional software platforms resulting in greater software development productivity.
- **Verification environments:** Verification environment is a model of the rest of the system. It faces several challenges like similar speed of execution issues, similar issues of reusability, evolvability etc. Bluespec System Verilog is used for coding test benches, synthesizable transactors and both system and reference models.

3.5.2 Structure of BSV:

Bluespec basically borrows its ideas from two sources for its sructural abstractions.

i)System verilog: For modules and module hierarchy, defining interfaces, syntax for literals, scalars, loops, blocks and expressions; syntax for user defined types like structs,enums,arrays etc.

ii)Haskell: Its a modern pure functioning programming language that addresses software complexity. It is a promising basis for attacking the parallel programming software crisis that has been precipitated by the advent of multi threaded and multicore processors.Bluespec borrows ideas from haskell for parametarization,static elaboration and more advanced types.

3.5.3 Components of Bluespec:

BSV is realized using the following components.

- *BSV language syntax:* BSV helps a designer to develop a high level,hardware design utilizing methods and rules which can be compiled to a verilog RTL design.

- *BSV compiler:* The BSV syntax is given to the compiler to generate hardware description for either verilog or bluesim.
- *BSV library packages*: BSV contains a set of libraries to provide programming idioms and hardware structures.
- *Primitive Bluespec elements*: BSV elements like registers and FIFOs are expressed as verilog primities.
- Bluesim: A simulator for BSV designs.
- Bluetcl: A set of Tcl extensions, packages and scripts to link into a bluespec design.
- Bluespec Workstation: An integrated environment containing all Bluespec components as well as third party design tools, including simulators and editors.

3.5.4 Overview of Bluespec:

There are three distinct stages in designing with BSV.

- *Coding a specifcation in BSV:* A designer writes a BSV program, including VHDL,Verilog and ***c*** components as desired.
- *Compiling the BSV program:* Compiling a BSV program is comprized of two stages.
  *a)*Pre-elaboration: Type checking and parsing
  b)Post-elaboration: Code generation
- *Simulation and Synthesis:* The compilation output is either simulated orprocessed by a synthesis tool.

Advantages of Bluespec over other programming languages are:

- *Flexibility:* Adding extra features to the design is much simplier in Bluespec because it takes much fewer lines of code when compared to other programming languages for implementing a design.
- *Test and Debug*: Bluespec generates a simulator called Bluesim as an alternative to verilog modules which can be used to run design on clock cycle basis. Verification is faster in BSV because Bluesim is much faster than verilog/VHDL simulators.
- *Design and Implementation Time*: It is easier and faster to code a design in Bluespec than other programming languages like VHDL/Verilog because of its high level nature and library modules.
- *Modularity*: Bluespec offers much higher levels of parametarization for types, modules and functions enabling better modular designs.

The compilation stages of Bluespec is shown in diagram below:

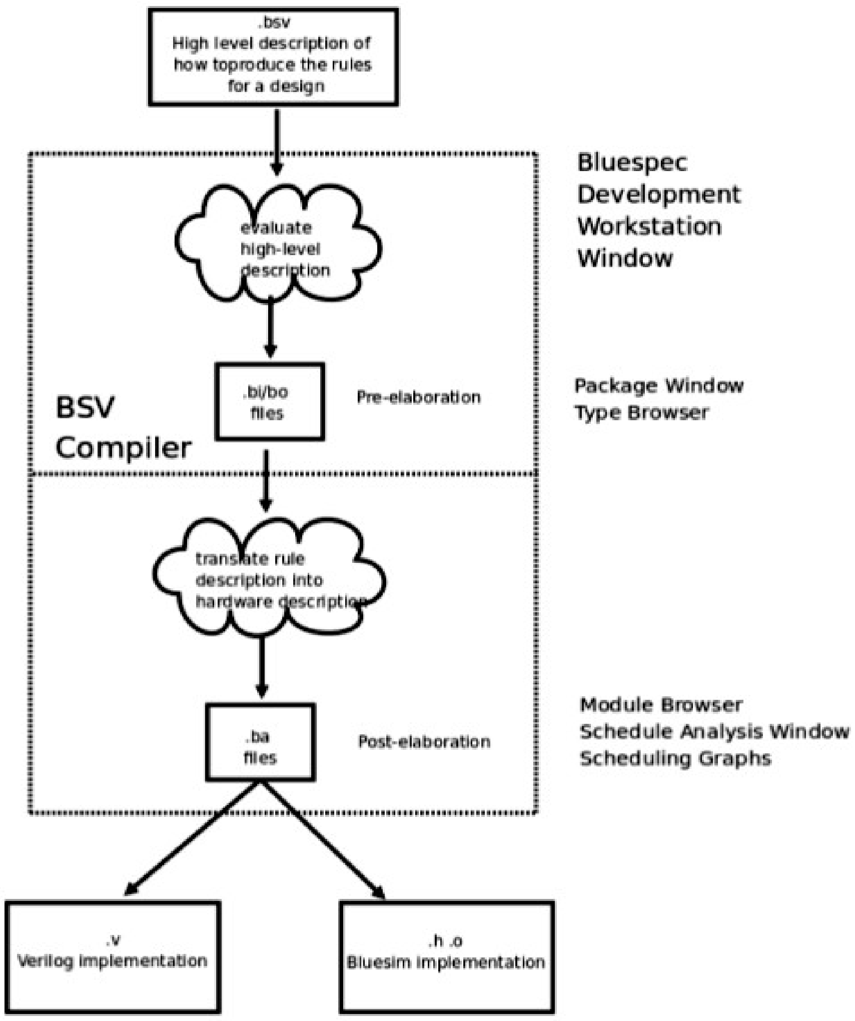

.bsv
High level description of how toproduce the rules for a design
evaluate high-level description
.bi/bo files
Pre-elaboration
BSV Compiler
translate rule description into hardware description
.ba files
Post-elaboration
.v Verilog implementation
.h .o Bluesim implementation
Bluespec Development Workstation Window
Package Window
Type Browser
Module Browser
Schedule Analysis Window
Scheduling Graphs

Fig 3.1: BSV compilation stages

Though Bluespec has many advantages as discussed above it has its own disadvantages in terms of area, power and timing.Eventhough Bluespec says that it outputs high quality verilog code, In many cases it turns out that handwritten verilog/VHDL code is better in terms of area and timing.

Chapter-4

Architectural Design and Implementation

This chapter explains the architectural design,Implementation details of I class processor.

4.1 Instruction Fetch unit

Instruction fetch is unit is the first block where instructions are processed. It is responsible for feeding the processor with instructions to execute. It mainly comprizes of instuction cache and necessary logic to compute fetch address.

Fetch unit sends program counter to instruction cache and gets the instructions. Fetch width of I class processor is 3 ie., three instructions must be fetched from instruction cache per clock period. Hence the calculation of next pc should be done along with cache access. Branch instructions introduce significant complexity as the target address is not known until the branch is executed. Branch predictor predicts the next pc using past history of that branch.

Fetch unit takes the predicted PC from branch predictor unit and sends request to instruction cache. Instruction cache responds with a cache block containing eight consecutive instructions. For instance, if fetch unit requests i cache with PC 0, i-cache returns a packet of instructions with PC 0 to PC 8.

Following book keeping actions must be done before this block is stored in fetch target buffer.

- If PC is not a multiple of 8 ie., if the last three bits of PC are not zero, packet[1] and packet[2] are discarded.
- If packet[0] of instruction packet is a branch which is predicted as taken, then discard packet[1] and packet[2].
- If packet[1] of instruction packet is a branch which is predicted as taken, then discard packet[2].

  Instruction 0 is always a valid instruction.where the other instructions can be valid or invalid.

Tournament branch predictor is used which takes branching decisions based on both global and bimodal predictors.

4.2 Instruction Decode Unit:

The purpose of instruction decode unit is to know the semantics of an instruction and to define how this instruction should be executed by the processsor. In decode stage the processor basically identifies the following aspects of an instruction.

- Type of instruction:memory,control,arthematic etc.
- Operation to be performed: ADD,SUB,OR,AND,XOR etc., in case of arthematic instructions, BEQ, BNE, BLT, BLTU, BGE, BGEU etc., in case of control transfer instructions, LD, LW, SW, SH, SB etc., in case of load and store instructions.
- Resources required by the Instruction: Finds the source and destination registers.

Typically the input to the decode unit is a stream of bytes that contains the three instructions to be decoded. The decode unit splits the byte stream into three instructions based on instruction boundaries and then generate a series of control signals for each valid instruction. The complexity of decode unit increases as we increase the number of instructions being decoded in parallel.

Unconditional Jump instructions like JAL (Jump and Link) is specially processed in decode stage. Jump and Link instruction changes the program counter by an offset and stores PC+4 in the specified destination register. If a JAL instruction is encountered in the decode stage, target PC is calculated and stored in the program counter register.
If instr0 is a JAL, then instr1 and instr2 are made invalid in the decode packet.
If instr1 is a JAL, then instr2 is made invalid in the decode packet.

4.3 ALLOCATION Unit:

This pipeline phase performs two main activities. They are

1. Register renaming
2. Instruction dispatch

4.3.1 Register Renaming:

Register renaming removes false dependencies (WAR and WAW hazards) due to reuse of registers. Dispatch stage reserves some resources like ROB entry, Issue queue entry, Load store queue entry etc. These are required by the instruction for getting executed. If any of the buffers are full the instruction is stalled till they become available. Current day micrprocessors are mainly using three kinds of renaming schemes. Thay are

4.3.1.1 *Renaming through Reorder Buffer*:

Data structure of the reorder buffer entry is of the following form.

Valid	Data	Destination Register

- Valid: It is a boolean field that indicates whether a outcome of an instruction is calculated. It is made false in the raname stage and made true when the result is broadcasted by the functional unit.
- Data: It stores the result broadcasted by the functional unit until all the instructions above it are committed.
- Destination Register: This field is filled during rename stage of the pipeline. It stores the address of the register file to which the result should be written during instruction commit stage.

In this scheme of register renaming, the register values are present both in the ROB and in the architectural register file. The architectural reg file stores the latest committed value for each register whereas the ROB holds results of non committed instructions. When an instruction is committed, corresponding entry in ROB is freed ie., it can be made available for other instructions in rename stage. A table called rename table indicates for every architectural register whether its latest value is present in ROB or regfile. The rename table contains one more field called ROB pointer to indicate the location of the operand in the ROB.

There is no need of maintaining a free list since the physical registers are part of ROB entry assignment which is basically a FIFO.Hence the allocation and release of physical registers is simple in case of ROB based renaming scheme.

Since an operand can reside in two different locations in its lifetime it adds extra complexity to the scheme to read operands.

The anatomy of ROB renaming is shown below.

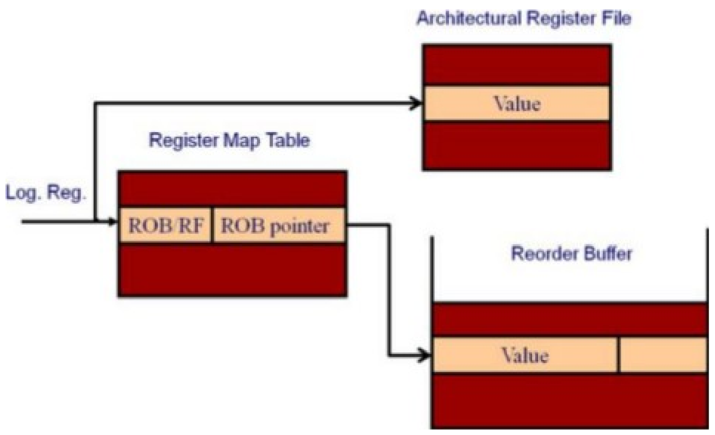


Fig 4.1: Renaming through Reorder Buffer

4.3.1.2 ***Renaming through a Rename Buffer:***

Around one third of the executed instuctions does not produce any register result. In the above discussed renaming scheme, every instruction is given a slot in the ROB which means that one third of the storage is not getting used.

Renaming through rename buffer is slightly modified version of renaming through ROB. The idea behind this scheme is to have a separate buffer to store the result of inflight instructions. So, only the instructions that produce a result consumes a slot for storage. Similar to the ROB scheme, results are first stored in rename buffer then copied to architectural regfile while committing the instructions.

4.3.1.3 ***Renaming through merged register file:***

As the name indicates, this renaming scheme contains a merged reister file to hold the results of both speculative and committed instructions. Size of this single register file is bigger than the number of architectural registers. Each register in this register file can either be allotted or free. Free registers are stored in a queue called free register queue. There is a register map table to store the latest physical register assignment for each architectural register.

Allocated register may contain any of the following values.

- A committed value which means that it is acting like an architectural register.
- A speculative value when the result of an instruction is available but the particular instruction is not committed yet.
- No value at all when that register is allotted but the instruction has not produced any result.

Free queue can be implemented using a circular buffer to indicate the identifiers of free registers. Reneming is done using a map table whose size is equal to the number of architectural registers.When an instruction is renamed, the rename table is looked up to know its source operand mappings. If the instruction produces a register result, a register from the head of the free queue is removed to rename the destination operand of that instruction and the rename table is updated to reflect this change. The pipeline is stalled if the free queue becomes empty.

Merged register file implementation is shown in the figure below.

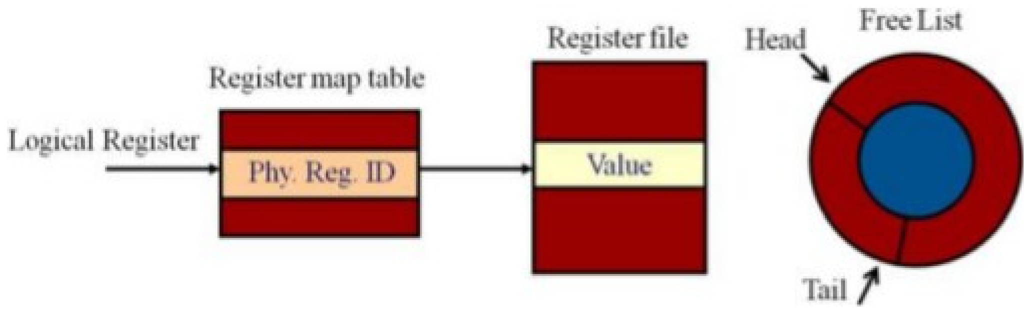


Figure 4.2: Merged Register file implementation

A physical register is freed when no other instruction is going to use it anymore ie., when the last instruction that uses this register commits. However, it is difficult for the hardware to know the last usage of a register. So, the processor uses the safe and conservative way. A physical rgister is freed when the following instruction that uses the same architectural register as destination commits.

For instance, consider the following instructions.

R1= R2 + R3

R1= R6 * R7

Renamed instructions are the following

P1= P2 + P3

P4= R6 * R7

P1 is freed when instruction 2 is committed.

4.3.2 Register File Read:

Reading a register file can be done in two different ways.

1.Read before issue

2.Read after issue

In Read before issue case, Register file is read before the instructions are dispatched to issue queue and the values are stored in the issue queue. The operands which are not available at that time are marked as non available and can be obtained later from the bypass networks of the ALUs. As register file is accessed before issue, all the operands are not provided by the regfile. Hence less number of read ports are enough as only a portion of operands is provided by it.

In this method, the issue queue stores the operands similar to regfile and hence expensive in terms of area. Also when the same operand is present in more than one instruction, there will be replicated copies of same data in the issue queue which is again a waste of area.

In Read after issue case, the operands are read after the instruction is issued to be executed. Here only the identifiers of the source operands are stored in the issue queue. The drawback of this method is that it requires more number of read ports to the register file since large portion of operands is provided by it. On the otherhand, the source operands are read only once and need not be copied anywhere.

Design Alternatives:

So far we have seen three different renaming schemes and two different register read methods. ROB based renaming duplicates the register values in both ROB and architectural register file. When an instruction comes to rename stage it is allotted an entry in the ROB and when it commits corresponding ROB entry is freed.

On the other hand, merged Register file renaming uses a single regfile to store both the speculative and committed results. Its a complex mechanism but has the following advantages over other renaming schemes discussed above.

- No need of copying the results from one location to another because results are written just once in merged register file renaming scheme. Whereas in ROB renaming scheme results are copied from ROB to architectural register file while committing an instruction which causes extra power consumption.
- In ROB based renaming operands can come either from architectural register file or from ROB which increases the amount of interconnect needed. On the otherhand, operands come only from a single location in merged register file based renaming.

In merged register file based renaming scheme, both the read before issue and read after issue can be done as there are no significant differences in their implementation but in case of ROB based renaming scheme, read before issue is more appropriate than read after issue.

Since the register values are moved from one location to another, in read after issue scheme the issue queue stores the pointer of the source operands. If when an instructioin is renamed, the source operand is in the ROB, the issue queue will store its pointer. If the producer of the source operand is committed before it is used by the consumer instruction, the value is copied to architectural regfile and the pointer stored in the issue queue will not be valid anymore since that particular entry might have been allocated to some new instruction. In order to avoid this, we need to do an associative search in the issue queue for each committed instruction to see if any entry is directing to its destination register. If this is happening, the pointer needs to be changed to the corresponding architectural regfile location. Its very difficult to implement this in hardware. Hence read before issue is preferred to read after issue for ROB based renaming scheme.

As we have seen that the read after issue is area efficient scheme, merged register file renaming with read after issue scheme is used in I-class processor implementation.

4.3.3 Renaming Multiple Instructions:

In our I-class processor, three instructions are renamed every cycle. While renaming three instructions in single cycle, dependencies between them should be checked. If all the three instructions are valid, three registers are taken from free queue and three entries are allotted in the issue queue.

Some additional actions to be performed while renaming three instructions parallelly are the following

- If the source operand of instruction-2 is same as that of destination operand of instruction-1 or destination operand of instuction-0, then change its identifier and mark the ready field as 'False'.
- If the source operand of instruction-1 is same as that of destination operand of instruction-0, then change its identifier and mark its ready field as 'False'.
- If the destination operand of instruction-0 is same as that of destination operand of instruction-2, then update the FRAM with instruction-2 only and if the destination operand of instruction 0 is same as that of destination operand of instruction-1, then update the FRAM with instruction-1 only. If destination operands of all the three instructions are the same then update the FRAM with instruction-2.
- Interdependencies amoung the instructions should be handled when load store instructions are involved.

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

4.4 The Issue Stage:

This stage is responsible for issueing the instructions to the functional units for execution. There are two different kinds issueing methods.
1.In order issue
2.Out of order issue

As discussed above, in order scheme issues the instructions in program order whereas out of order scheme issues the instructions as soon as their source operands are available. Out of order issue is used in I-class processor to improve the clock speed.

After instructions are renamed they wait in the issue queue till all the data and structural hazards are resolved and the corresponding functional unit becomes free. Data structure of issue queue is shown below:

Table 4.1: Issue queue structure

Field	*Work*
Functional unit details and operation	Holds the operation to be performed and nameof the functional unit to be sent for execution
Op1	Source Operand-1 pointer
Op1 ready	Denote whether the operand-1 is ready or not
Op2	Source operand 2 pointer
Op2 ready	Denote whether the operand-2 is available or not
Imm valid	Field denoting whether the instruction has immediate field
Imm buffer index	Holds the instruction position in immediate buffer
Destination architectural register	To back registers to free queue
Memory queue index	Load store queue pointer
Prediction	To find whether the branch should be taken or not in case of branch instructions
Program counter	Holds the instruction address and needed for branch and AUIPC instructions.

Generally thirty percent of the instructions have immedate field and hence using a separate buffer called immediate buffer and storing only the pointer to the buffer saves the area in the issue queue. The *imm valid field* indicates whether the instruction have immediate field or not.

If the instruction is load store instruction, it is allotted an entry in a separate queue called load store queue. The pointer to the load store queue isheld in the *memory queue index* field.

Marking op-1 ready and op-2 ready:

The desination tag of the producer instruction is broadcasted to the issue queue to say that the producer instruction hs completed execution and corresponding consumer instruction may use it. Then the *op-1* ready and *op-2* ready fields of the consumer instruction are set ie., marked ready.

But if the consumer instruction is in rename stage when the producer is broadcasting results, ready field will not be marked correctly. To avoid this destination tag is also sent to map stage.

There are two different varieties of issue queue implementations. They are
1.Unified issue queue
2.Distributed issue queue

4.4.1 *Unified Issue Queue:*

This scheme maintains a single issue queue to store all the instructions of the code.It requires a separate logic to know which functional unit an instruction should be sent. It is difficult to implement in hardware as the complexity of the logic increases with size of issue queue and the number of functional units present in the processor.

Clock frequency can be increased with the size of the issue queue as more number of instructions can be renamed and stored in it. But, due to the complex logic unified issue is offering contradictioin results after a certain size of issue queue. Adding delay in the critical stage of pipeline is the drawback of unified issue queue.

4.4.2 *Distributed Issue Queue:*

Separate issue queues are maintained for different functional units. Some functional units can share a common issue queue. In this case, the map stage figures out to which functional unit the particular instruction should be sent and enqueues it into corresponding issue queue. This reduces the delay in the most critical stage of the processor pipeline.

Distributed issue queue has the following drawback. When the workloads are not balenced one issue queue will be full and the pipeline will be stalled and all the other issue queues will be empty.

The anatomy of unified and distributed issue queue is shown below:

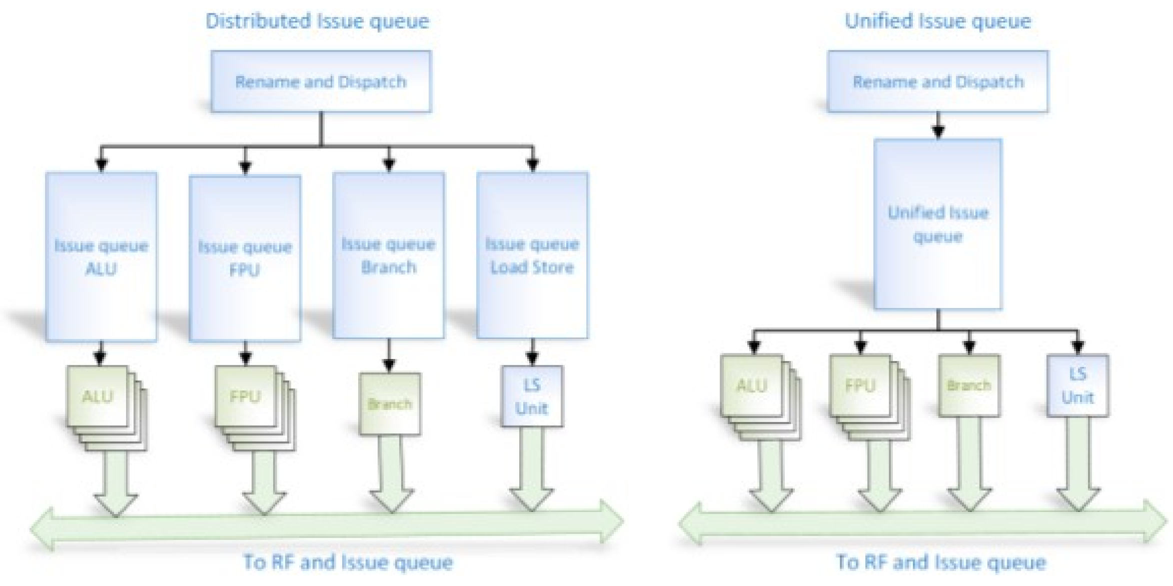


Figure 4.3: Distributed and Unified Issue Queues

We use unified issue queue in our I-class processor in order to avoid stall in the pipeline when the workloads are not balenced.

Instructiion issue is again a two istage operation which are discussed below.
1.Instruction Wake-Up
2.Instruction Select Grant

4.4.3 Instruction Wake_up

Wake up is the event that notifies that one of the source operands of the consumer instruction has been produced by the producer instruction. The destination tag of the producer instruction is broadcasted to all the instructions in the issue queue. It is then compared with the tags of the source operands of all the instructions. If the destination tag matches with any of the source operand tags then corresponding *op-1 ready* or *op-2 ready* is marked in the issue queue entry.

Wake up logic adds more delay as the number of instructions and size of issue queue increases. The anatomy of issue queue wake up logic is shown below.

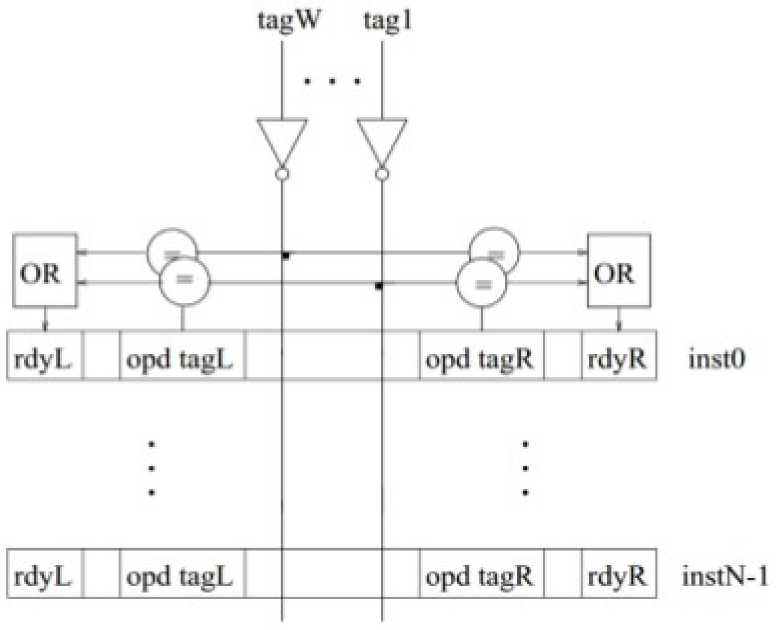


Figure 4.4: Instruction wake up logic

4.4.4 Instruction Select Grant:

Select grant unit selects the ready instructions(instructions whose source operands are ready and corresponding functional unit is also free) in the issue queue and sends them to appropriate functional unit. There should be a provision for select logic because number of ready instructions are greater than number of functional units available. Inputs to the select logic are requests for acccess to particular functional unit from the instructions in the issue queue. This is called *request vector* which is a vector of bits where the size of the vector is equal to the number of instructions in the issue queue. If an instruction in the issue queue has both its operands ready and not not yet selected for execution, then the corresponding bit in the rquest vector is set. The output of the select logic is also a vector named *grant vector* whose size is equal to the number of instructions in the issue queue. If an instruction is granted functional unit then the corresponding bit in the grant vector is set.

There are basically two different kinds of selection policies. They are

1.Age based selection

2.Position based selection

Age based selection policy:

Older instructions in the issue queue are given more priority and granted functional unit earliar than latest instructions that entered the issue queue. But it needs complex hardware to hold the age information.

Position based selection policy:

Instructions are given priority based on their position in the issue queue. Instructions in the top are given preference to instructions at the bottom of the issue queue. This policy is relatively simple to implement because it just needs priority encoders.

Position based selection policy with tree encoder is used in I-class processor. Tree shaped priority encoder implementation is shown below.

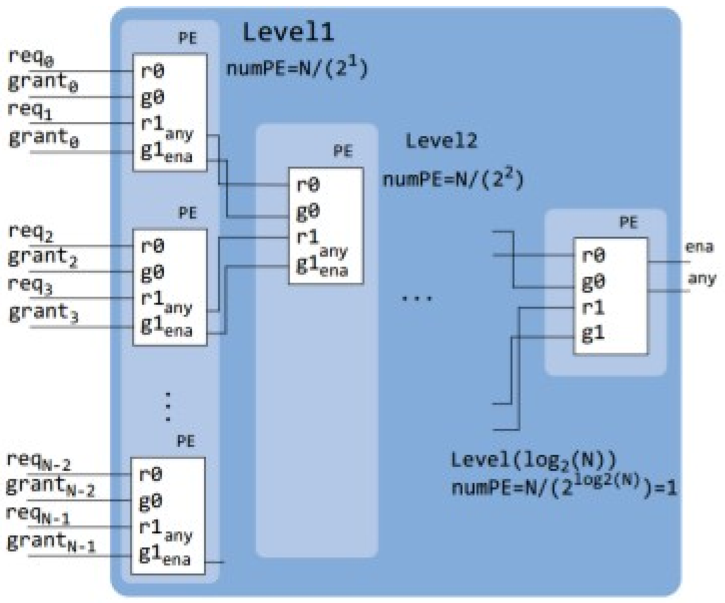


Figure 4.5: Tree shaped priority encoder implementation

Each block in the above figure is a priority encoder with four inputs and two outputs. Requests from top level are passed to the root of the tree. This is done by the any output of tthe priority encoder block which is set if any of the request vectors are set. The fuctional unit status is shown by the *enable* to the root priority encoder. *Enable* bit is set when the functional unit is free. Enble inputs of the encoders in the next higher level is given by *grants* from the root. Hence the grants are sent back to the branches.

As the instructios enter and leave the issue queue in program order, instructions at the head of the queue are older than farther instructions. Hence to implement Age based selection, two barrel shifters are needed one for roating request vector down by an amount equal to the head position, and another for rotating grant vector up by the same amount. This adds extra complexity to the hardware and hence this method is genenrally not preferred.

4.5 Data Read

Read after issue is selected in I-calss processor for reading data in order to save area of the issue queue. Immediate fields are stored in separate location called immediate buffer and its pointer is stored in the issue queue. Immediate buffer valid bit in issue indicates whether it is an immediate instruction or not.

Instructions and data are put into the next buffer named data read/drive buffer from where they sent for execution and hence the next pipeline stage is execution.

4.6 Instruction Execute

This stage is responsible for calculating the results. In this stage the source operands along with the type of operation to be performed are send to the computational units like ALU,load store unit etc. The processor operates on the source operands and the results are written to the physical regfile. The destination tag is broadcasted to the issue queue to produce source operands for consumer instructions.

There are several operations that the procssor performs in execute unit. Some of them are listed below.

- Arthematic operations like addition, multiplication, dividion etc. These are handled by ALU.
- Memory operations like loading and storing data. These operations are handled by Load Store unit.
- Control transfer operations to change the PC value. These are handled by Branch unit.

Different instructions have different complexities and hence take different time in the execute stage. In current day processors it is generally not a single pipeline stage but several.

4.7 MEMORY DISMBIGUATION SCHEME

If more than one instruction operate on same memory location then memory data dependency occurs. These dependencies can be found only in the execute stage ie., after the effective address is calculated. These dependencies are handled by memory disambiguation scheme. There are two different ways in which memory data dependencies are solved. They are speculative memory disambiguation and non speculative disambiguation.

According to non speculative disambiguation, memory access instructions are not allowed to access memory if they have dependency with any of the previous memory instructions., ie., memory access instructions are forced to execute in program order. Whereas the speculative scheme predicts whether a memory operation has any dependency with other inflight memory instructions.

4.7.1 *Load Bypassing:*

If a load instruction is independent of the previous stores in the pipeline, then it is sent for execution. Hence for a load instruction to be sent for execution all the inflight stores must have been issued. This is because for a load to figure out memory address match the effective address for stores must have already been calculated.

4.7.2 *Load Forwarding:*

The result of store will be sent to load instruction if there is an address match with the earliar store instruction. This scheme is sometimes named store forwarding because its the store data that is being forwarded. This requires a separate buffer to hold the store results and their effective addresses. Before the load is being issued, the buffer is looked up for address match and if there is alaising, it takes the value directly from it.

4.7.3 Handling Load store instructions:

Speculative load execution is used and implementation details are discussed below. In this scheme load instruction is issued even if the earliar stores in the pipeline are not. It is assumed that there are no memory violations hence called speculative load disambiguation. If a load is not forwarded or if it is forwarded from a incorrect store(not the latest store) then the pipeline will be flushed from the mispredicted load instruction.

Two queues namely load queue and store queues are maintained to hold the load and store instruction. Load and store instructions are allotted entries in their respective queues and if they are full the pipeline is stalled. Data structures of load and store queues are shown below.

Table 4.2: Store Queue Structure

Field	Work
Filled	Tells if an entry is allotted to store instruction
Valid	Tells if the store queue entry has valid data
Store address	Memory address location to which data should be stored
Store data	Result to be stored in memory
Store size	Tells the size of memory access

Table 4.3: Load queue structure

Field	Work
Filled	Indicate when an entry is alloted to load instruction
Valid	Set if the data in load queue is valid
Store mask	Record of all the stores on which load is dependent
Load address	Memory location from which data should be loaded
Load size	Size of access
Forwarded	Tells whether store forwarding is happened to laod instruction
Forward acknowledge	Specifies whether the alaised store instruction has committed
Aliased	Shows that a memory violation has occured

Since every load store instruction should be provided an entry in the load store queue, the pipeline is stalled any one of these queues become full.

Load store instuctions are processed in the following steps during rename issue stages of processor pipeline.

- Every load store instruction is allotted an entry in the respective load or store queue and filled bit is made set and the valid bit is reset. For load instructions store mask bit is also filled.
- *Mem queue index* field in the issue queue is filled with the index of the load and store queue of that memory instruction.
- Once all the dependencies are resolved, instructions are sent for execution to load store unit where load and store instructions are treated separately. Atmost one instruction can be issued for execution in one clock cycle.

One memory read or write is allowed to perform in one clock cycle in our I_Class processor in order to reduce hardware complexity. Hence a store in commit stage and a load in execute stage cannot happen parallellly.
The complexity in handling load store instructions lies in the instruction commit stage which is discussed in the furthur sections.

Hardware implementation is shown below:

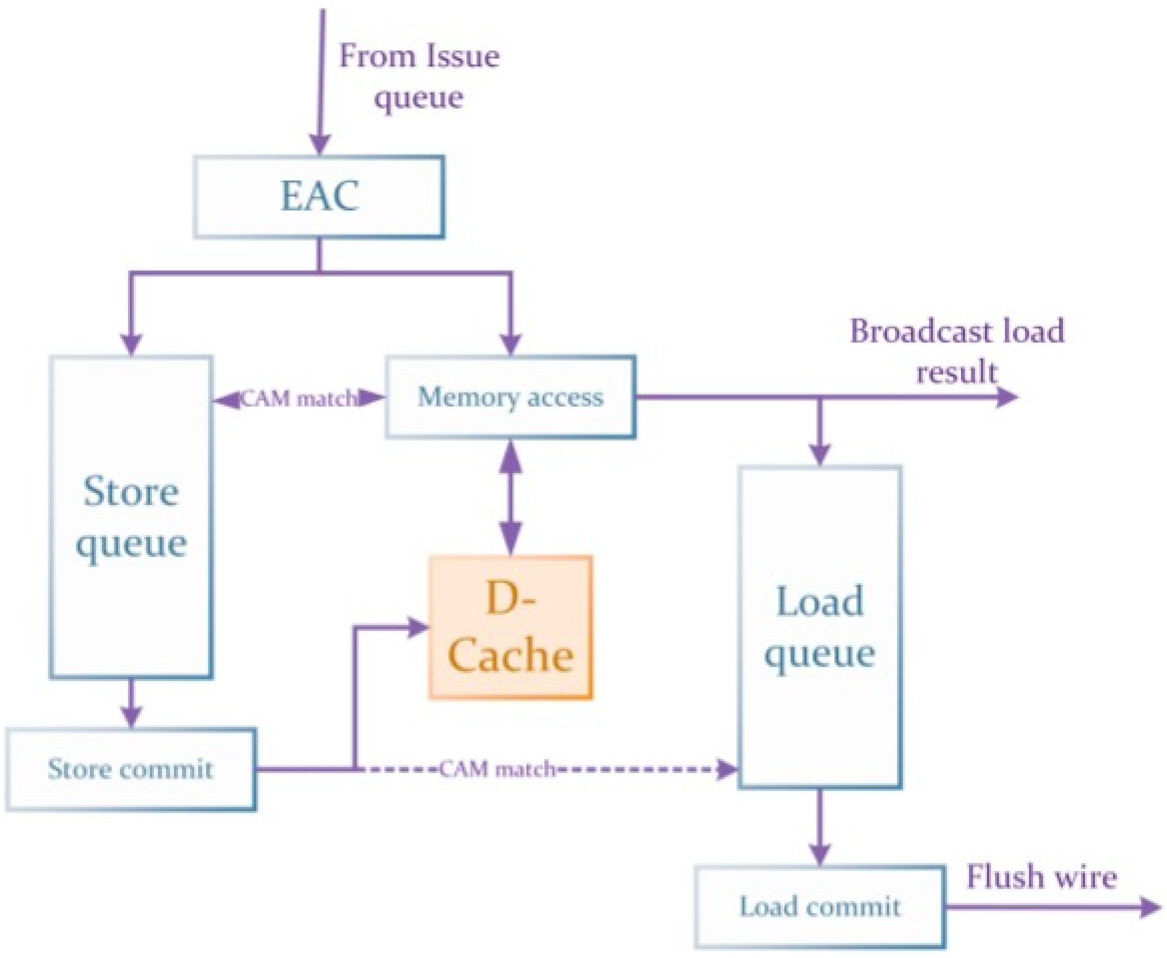


Figure 4.6: Speculative load store unit

First cycle of execution for load and store instructions is calculating the effective address. After calculating that, store instruction updates all the remaining fields of the store queue (except the *filled* field which is set in the rename stage and load instruction revises *valid* and *load address* fields of load queue.

The next step of instruction is to search store queue for address match. Search is done only on the store queue entries which are present in *store mask* field of the load queue. If there is an address match, that result is broadcasted to issue queue and '*forwarded'* bit field in load queue is set. In case, if there is no address match, read request is sent to data cache.

The core complexity of memory disambiguation scheme lies in the commit stage of the instruction.

Commit stage for load store instructions:

During the commit stage, store instruction sends the write request to the data cache. Also it looks if any non committed load matches this address, and updates that in the load queue entry of the corresponding instrcution.

The commit logic of load store instructions is shown in the following flowchart.

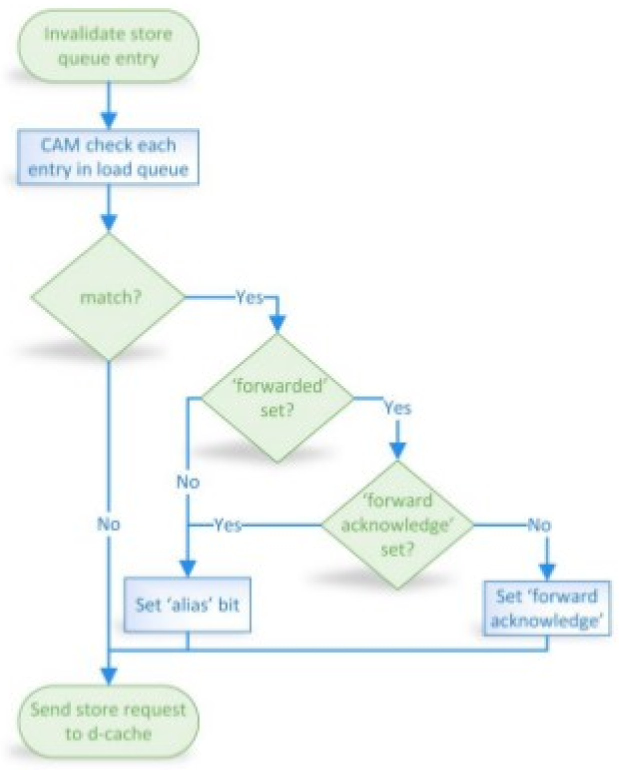

Invalidate store queue entry
CAM check each entry in load queue
match?
Yes
'forwarded' set?
Yes
No
'forward acknowledge' set?
Yes
No
No
Set 'alias' bit
Set 'forward acknowledge'
Send store request to d-cache

Figure 4.7: Commit stage of Load store instructions

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

- If the store forwarding has not happened inspite of having address match.
- If more than one store address matches with load, the first store instruction set *forward acknowledge* bit during its commit. When the second alias store is committing it sees the *forward acknowledge* bit being set and sets the *Alias* bit since wrong store information might have been forwarded. In this case, Pipeline is flushed from the wrongly speculated load instruction.

4.8 Instruction Commit

This is the last pipeline stage of I-Class processor. Instruction enter the issue queue in program order but they are issued out of order. So many speculations like branch predictions, load forwarding are implemented in order to speed up the processor but these may result in inprecise exception. In order to get precise exception, commit stage is introduced in current day processors where the instructions are handled in program order.

Commit stage is responsible to do the following actions discussed below.

- Immediate instructions are allotted entry in the immediate buffer. In their case, Immediate buffer entry is freed during commit stage.
- Load store instructions are allotted entry in load and store queues. Load store entry is freed in commit stage for such instructions and the actions described in section 4.7.3 are performed.
- If the instruction is found to be mispredicted branch in commit stage pipeline flush signal is sent and new instruction is fetched from a different location.
- If the instruction is found to be a wrongly speculated load send a signal to flush the pipeline and load operation is performed to get the latest stored data.
- Add back the allotted register in the rename stage to the free register queue in a way discussed in section 4.3.1.3.
- Update RRAM (Retirement end RAM) index of the destination architectural register with the destination operand for all instructions which produce result.

Steps to be followed while doing Multiple Instruction Commit:

I-Class processor can commit a maximum of three instructions per cycle. Following actions needs to be done while committing multiple instructions.

- If the first instruction is store do not commit the other two instructions if they are memory access instructions. If the first instuction is not memory instruction and the second instruction is a store instruction do not commit the third instruction if it is a memory instruction.

  In summary do not commit two memory instructions at a time.
- If the second instruction is mispredicted branch do not commit the third. If the first is mispredicted branch do not commit the remaning two.
- If the destination architectural registers of any two of the instructions is the same then update RRAM with the latest instruction

4.9 Exception Handling

Exceptions are generally handled in commit stage. There are two reasons behind this. First is that we should be sure that the instruction causing exception is not speculative. Second is we want to maintain precise exception ie., the architectural state similar to the way it would have been when all the instructions earliar to the one causing exception has been executed in original program order. Following book keeping actions needs to be done to get precise exception handling.

- Copy RRAM content to FRAM to provide the previous mapping.
- Clear the issue queue contents and made the head and tail to zero.
- Empty all the ISB(inter stage buffer)s in the processor pipeline.
- Clear the load and store queue contents.
- As we need to add all the architectural registers to free queue during exception, it would be enough if we make the head and tail of FRQ(circular buffer) to zero.
- Squash program counter of mispredicted instruction is copied to the Program counter. Hence it serves as new PC.

CHAPTER-5
RESULTS AND CONCLUSIONS

This units explains the performance of I-Class processor and conclusions.

5.1 Verification Framework

Entire code of I-Class processor is done in Bluespec system verilog. Cache is different for instructions and data named instruction cache and data cache to speed up the processor.
The block diagram of I-Class processor with its cache interface is shown below.

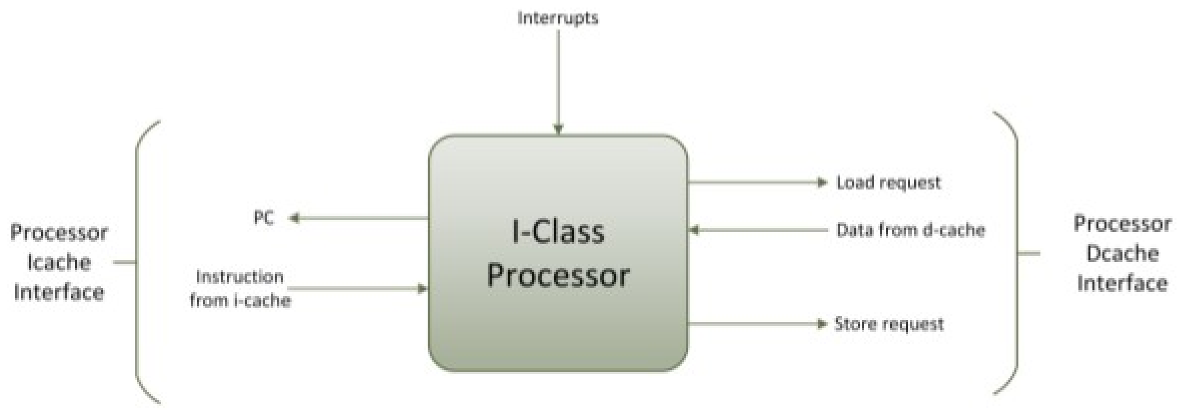


Fig 5.1: Interface between cache and I-Class processor

Automatic test case generator otherwise known as ATPG generates the test cases which are given as input instructions to be run on I-Class processor. These instructions are loaded in input.hex file which serves as icache to the processor. Initial memory status is generated by ATPG which is loaded into rtl_mem_init.txt file which serves as data cache for I-Class processor.
After commit of every instruction the content of register file and program counter are dumped into out.txt file. Also the contents of various buffers in the processor are also dumped in various output logs like store.txt, iq_status.txt etc for verification reasons. The out.txt generated by the processor is then compared with the output file generated by the simulator.
The simulation environment is shown in the figure below.

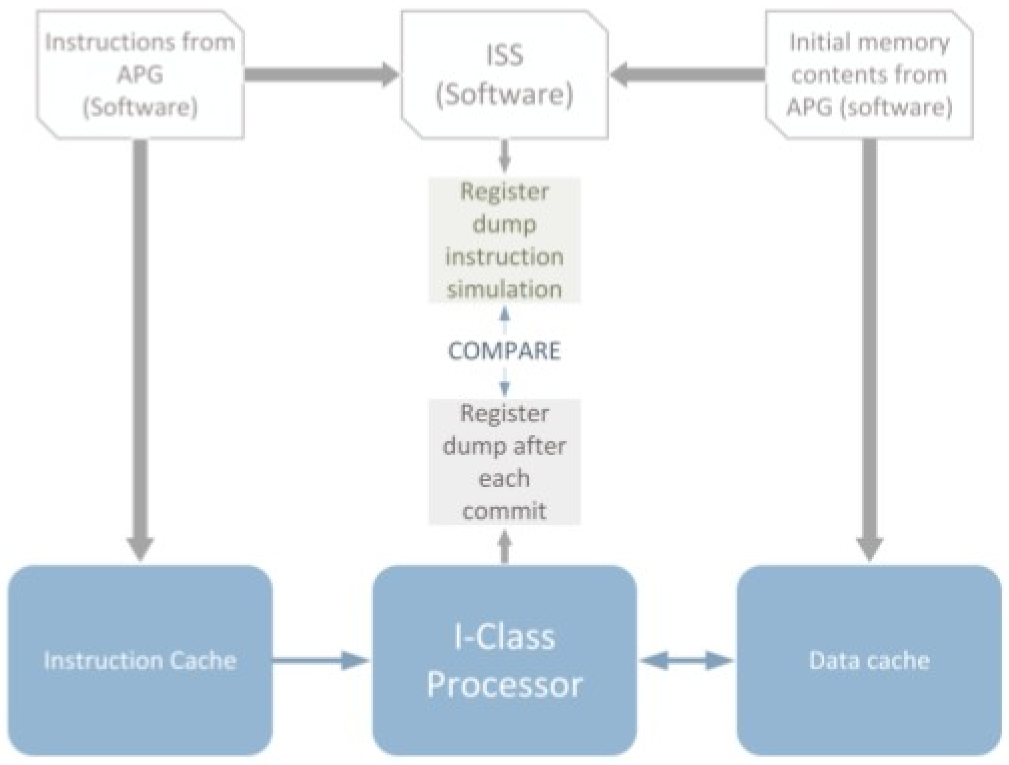

Instructions from APG (Software)
ISS (Software)
Initial memory contents from APG (software)
Register dump instruction simulation
COMPARE
Register dump after each commit
Instruction Cache
I-Class Processor
Data cache

Figure 5.2: Simulation environment of an I-Class processor

5.2 Results

I_Class processor is designed that can process three instructions parallelly in all stages of pipeline. Hence the IPC should be 3, but ideally the IPC is less than 3 because of the following reasons. Although the tournament branch predictor works well, sometimes it fails to predict correct branch flow. Then the instructions in all the stages of pipeline are flushed, interstage buffers are made empty and new instruction is fetched.

Similarly as we are using speculative load store unit, if the load happened from incorrect store, all the instructions from the load instruction are flushed and processed again.

Sometimes the pipeline delay occurs because of data dependencies among the instructions.

Sometimes the delay could be because of unavailability of functional units.

All these factors constitute decreasing the IPC of the I_Class processor.

The IPC of the designed processor is close to 2.9.

5.3 Conclusion

In this work out of order superscalar processor with fetch width of 3 is designed, implemented and verified. Issue queue size, load store queue size, immediate buffer size, number of ALUs are parametarizable. IPC of the processor (processing three instructions parallelly) is calculated. As whole coding is done in BSV which is good in modularity, fast development time, flexibility with a good ease of verification, it can also be used as a typical out of order platform to develop new architectures and to furthur enhance its speed.

CHAPTER 6
Future work

There is a wide scope of future work in this area. The following features can be added to the I_Class processor to furthur enhance its speed.

- Adding prefetchers (discussed in Section 2.4.3) to decouple the branch predictor which increses the throughput.
- Using multilevel caches to reduce memory access delay.
- Develop mutithreading and multicore processors out of it.
- Run the actual benchmark programs to furthur optimise it and to improve throughput.

BIBLIOGRAPHY

[1] Antonio González, Fernando Latorre, and Grigorios Magklis . "*Processor microarchitecture: an implementation perspective*", 2011 , Page no:(11 to 50)

[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,May 6, 2014 page no(5 to 30)

[3] Andrew, Waterman, Lee Yunsup, Patterson David A., and Asanovic Krste. "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture version 1.7." *EECS Department, University of California,* Berkeley, May 9, 2015 page no(7 to 9)

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

[5] William M. Johnson "Superscalar Processor Design." *Department of Electrical Engineering and Computer Science,Stanford University.*

[6] Babak Falsafi, Thomas F. Wenisch. " A Premier on Hardware Prefetching", *Synthesis Lectures on Computer Architecture,* 2014. *page no(7 to 15)*

[7] Victor Lee, Nghia Lam, Feng Xiao and Arun K. Somani, "*Superscalar and Superpipelined Microprocessor Design and Simulation: A Senior Project", IEEE Transactions on Education, VOL 40, NO 1, FEBRAURY 1997.*

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

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