Design Of A 5-Stage Dual Issue Processor

A Project Report

submitted by

MANISH KUMAR MAURYA

in partial fulfilment of the requirements

for the award of the degree of

BACHELOR OF TECHNOLOGY

AND

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

MAY 2017

THESIS CERTIFICATE

This is to certify that the thesis titled **Design Of A 5-Stage Dual Issue Processor**, submitted by **Manish Kumar Maurya**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor Of Technology and Master Of Technology**, is a bonafide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. V. Kamakoti
Research Guide
Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date: May 2017

ACKNOWLEDGEMENTS

Successful completion of this project work would not have been possible without the guidance, support and encouragement of many people. I take this opportunity to express my sincere gratitude and appreciation to them.

First and foremost I offer my earnest gratitude to my guide, **Dr. V. Kamakoti** who has supported me throughout my project work with his patience and knowledge while allowing me freedom and flexibility to follow my own thought process.

My special thanks and deepest gratitude to **Neel Gala** and **Rahul Bodduna** who have been very supportive. They have enriched the project experience with their knowledge of the subject matter, active participation, deep understanding and invaluable suggestions.

ABSTRACT

KEYWORDS: Superscalar; Dual-issue; 5-stage pipeline; Data Dependencies

The number of transistors on a chip is increasing every year (according to Moore's law it doubles in every 2 years) and designers are developing different techniques and architectures to utilize all this hardware to increase the throughput of a processor. This project aims to design a superscalar processor which in simple terms is like replicating the hardware units of a processor to execute more than one instruction in the same time unit which means increasing the throughput. The processor designed in this project is a dual issue and has 5-stages that are Fetch, Decode, Execute, Memory and Writeback. Ideally a superscalar processor should give the improvement in multiples of the units replicated, like a dual issue processor should double the throughput but as this project will show in real scenario this is not possible due to data dependencies between the instructions and many multiple cycle operations which stalls the pipeline, a superscalar processor exploits the Instruction Level Parallelism of the program so it depends on the program how many of its instructions can be executed in parallel. We will see different results for different test cases executed on this processor.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

BSV	Bluespec System Verilog
ISA	Instruction Set Architecture
RISC	Reduced Instruction Set Computing
CISC	Complex Instruction Set Computing
HDL	Hardware Description Language
IPC	Instructions Per Cycle
PC	Program Counter
IF	Instruction Fetch
ID	Instruction Decode
WB	Write Back
WAR	Write After Read
RAW	Read After Write
WAW	Write After Write

CHAPTER 1

INTRODUCTION

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

1.1 Overview of current processor

The Processor design team of Reconfigurable and Intelligent Systems Engineering (RISE) Lab in the Computer Science Department of IIT Madras has been actively involved in building different variants of processors based on RISC-V Instruction Set Architecture(ISA). One of these processors is a 5-stage In-order scalar processor named " C-Class processor ". But being a scalar processor the maximum instructions it can execute in a single cycle is 1. This project describes the micro-architecture of a 5-stage in-order dual issue superscalar processor which is designed using C-Class processor as base. A superscalar processor will have higher throughput than a scalar processor but on the cost of more hardware. The entire design of the processor is done using a Hardware Description Language (HDL) called Bluespec SystemVerilog (BSV).

1.2 Changes in the current processor

Here are the main changes which I implemented in the current processor these will be explained in detail in coming chapters.

• I-Cache changed to supply two instructions.

• Fetch unit changed to receive two instructions attach token to them and then enqueue in decode buffers.

• A Completion buffer design and implementation.

• Register file changes to handle inter-dependencies between instructions and to handle other hazards.

1.3 Organization of the thesis

Chapter 2 contains the theory needed for this project. It starts with the single cycle processor design then deals with Pipelining and how it improves the the processor throughput with little hardware cost then we go to a supescalar design to further improve the throughput. It also briefly discuss about the ISA, mainly about RISC-V ISA which is used in this processor.Finally it describes about the HDL called Bluespec System Verilog, its key features.

Chapter 3 describes the changes implemented in the current processor using Bluespec System Verilog and also different problems and errors faced during implementation.

Chapter 4 describes the verification process and results.

Chapter 5 contains a conclusion and description on the future work.

CHAPTER 2

BACKGROUND

2.1 Micro-architecture Basics

2.1.1 How Does a Machine Process Instructions

What does processing an instruction means? Processing an instruction means transforming present Architectural State to next Architectural State according to the ISA specification of the instruction.

AS = Architectural (programmer visible) state before an instruction is processed

$\Downarrow$

Process instruction

$\Downarrow$

AS' = Architectural (programmer visible) state after an instruction is processed

ISA specifies abstractly what AS' should be, given an instruction and AS.

2.1.2 What is ISA

The instruction set, also called instruction set architecture (ISA), is part of a computer that pertains to programming, which is basically machine language. The instruction set provides commands to the processor, to tell it what it needs to do. The instruction set consists of addressing modes, instructions, native data types, registers, memory architecture, interrupt, and exception handling, and external I/O. Most of the processors in the present use either Reduced Instruction Set Computer(RISC) ISA or Complex Instruction Set Computer(CISC) ISA, both of these have their merits and demerits.

2.1.2.1 The RISC

A RISC was realized in the late 1970s by IBM. Researchers discovered that most programs did not take advantage of all the various address modes that could be used with the instructions. By reducing the number of address modes and breaking down multi-cycle instructions into multiple single-cycle instructions several advantages were realized:[2]

- compilers were easier to write (easier to optimize)
- performance is increased for programs that did simple operations

Evaluation of RISC

- Emphasis on software
- Register to register
- "LOAD" and "STORE" are independent instructions
- Low cycles per second,large code sizes

2.1.2.2 The CISC

Complex instruction set computing(CISC) is a processor design, where single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) or are capable of multi-step operations or addressing modes within single instructions.

Evaluation of CISC

- Emphasis on hardware
- Memory-to-memory
- "LOAD" and "STORE" incorporated in instructions
- Small code sizes, high cycles per second

The Overall RISC Advantage

Today, the Intel x86 is arguable the only chip which retains CISC architecture. This is primarily due to advancements in other areas of computer technology. The price of RAM has decreased dramatically. In 1977, 1MB of DRAM cost about $5,000. By 1994, the same amount of memory cost only $6 (when adjusted for inflation). Compiler technology has also become more sophisticated, so that the RISC use of RAM and emphasis on software has become ideal.

Our processor is based on RISC-V ISA which is a new instruction set architecture (ISA) that was originally designed to support computer architecture research and education, it's a completely open ISA that is freely available to academia and industry.

2.1.3 Single-cycle and Multi-cycle Machines

This section contains theory on single cycle and multicycle machine and why a multicycle machine is better than single cycle, this multicycle machine is the base for our dual issue supersclar processor.

2.1.3.1 Single-cycle machine

A Single-cycle processor does all the instruction processing for a instruction in single cycle i.e. it fetches instruction from memory, executes, and the results are stored all in a single clock cycle. The benefits of single-cycle processors is that they tend to be the most simple in terms of hardware requirements, and they are easy to design. Unfortunately, they tend to have poor data throughput, and require long clock cycles (slow clock rate) in order to perform all the necessary computations in time.

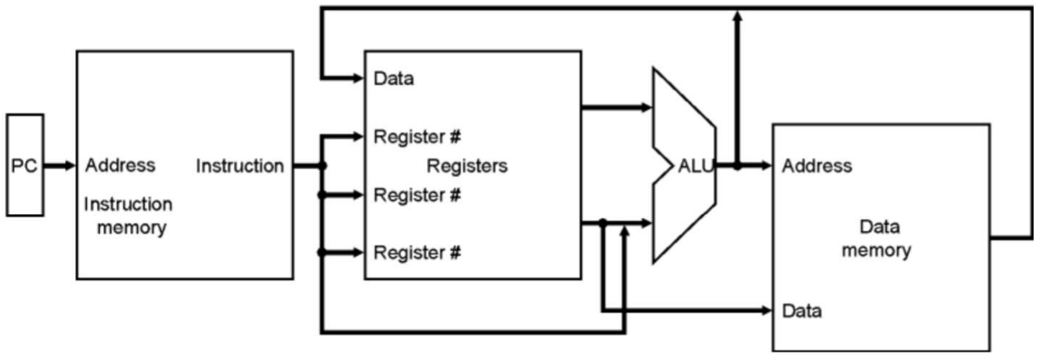


Figure 2.1: Single cycle processor datapath

The length of the cycle must be long enough to accommodate the longest possible propagation delay in the processor. This means that some instructions (typically the arithmetic instructions) will complete quickly, and time will be wasted each cycle. Other instructions (typically memory read or write instructions) will have a much longer propagation delay.

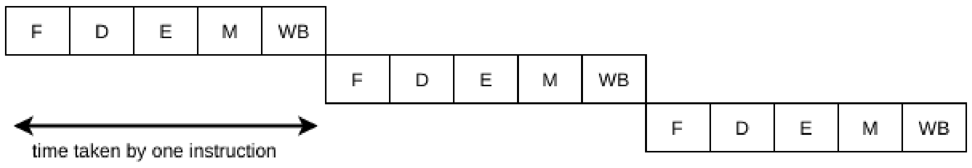


Figure 2.2: Instruction processing in single cycle machine

2.1.3.2 Multi-Cycle pipelined machine

In single cycle machine even when an instruction has completed one stage that stage is still not used till the present instruction completes the processing. In multicycle machine this idle stage can process another instruction so at a time more than one instructions are being processed in the processor but these instructions are in different stages. Here we have 5 stages in our processor these are Fetch(F), Decode(D), Execute(E), Memory operation (M) and Write-back(WB).

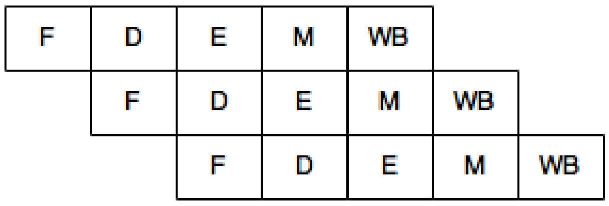


Figure 2.3: Instruction processing in multi cycle machine

In figure 2.3 we can see more than one instructions are processing in different stages in a clock cycle.

2.1.3.3 Improvements through pipeling

First we need to understand difference between Latency and Throughput.

Latency - After being fetched each instruction takes certain amount of time to complete all the stages and update the register file this is called the latency of that instruction.

Throughput - It is the number of instructions executed(completed) per unit of time. In a pipelined processor the latency to complete an instruction is same or slightly greater than the single cycle machine but the throughput increases because in pipelined processor after 1st instruction we get the output in each cycle.

2.1.3.4 Single cycle vs pipeline example

In an ideal pipeline all the stages take same time to execute and the pipe continues without any interrupts.Making these assumptions here is an example to compare advantage of pipeline over single cycle machine.

A non pipelined machine with 5 stages, each stage takes 50ns to execute, we want to execute 100 instructions.

Sol. Instruction latency = 50 x 5 = 250 ns

Time to execute 100 instructions = 100x250 = 25000ns

Now lets take a 5 stage pipelined processor with same execution time of each stage and 5ns overhead due to pipeline registers.

Sol. Latency = 50x5 + 5 = 255ns

Time to execute 100 instructions = time to execute 1st instruction + 50x rest of the instructions = 255 + 50x99 = 5205ns

Speedup obtained for 100 instructions = 25000/5205 = 4.8

2.1.4 Pipeline in detail

In this section I'll explain different stages of pipelined processor and their function.

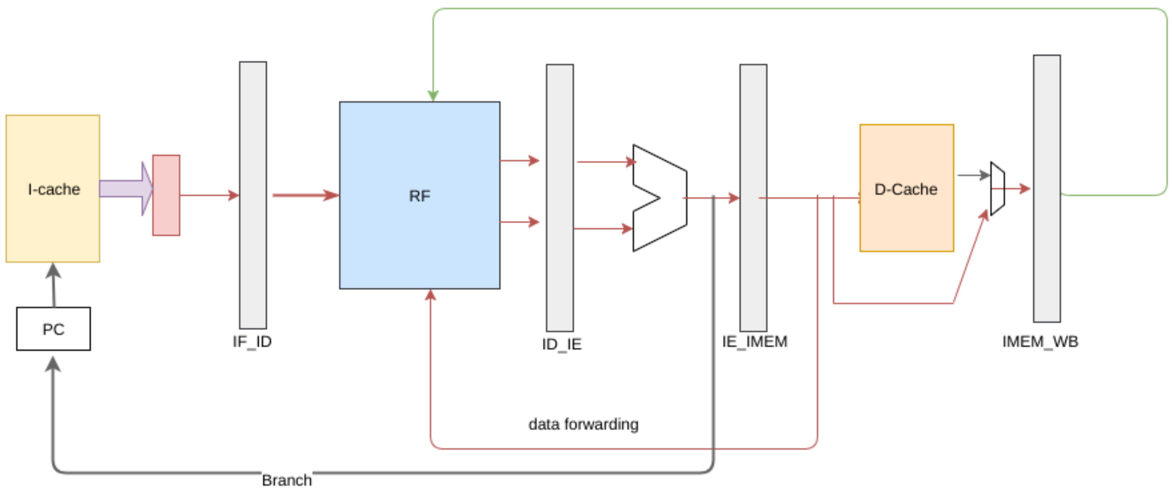


Figure 2.4: 5-stage pipelined processor

2.1.4.1 Fetch Unit

The instruction fetch unit takes instruction from the I-cache and feeds it into the processor. And it also calculates the next PC.

2.1.4.2 Decode Unit

The purpose of the instruction decode stage is to understand the semantics of an instruction and this will define how the instruction will be executed by the processor. In this stage processor identifies

- Which registers to read and write
- Type of instruction- ALU, memory, control etc.

2.1.4.3 Execution Unit

In this stage the operands are send to processor's computational unit along with the operation to be performed. In this stage

- arithmetic operations (addition,multiplication, etc.) are performed.

• effective address calculation is done for Load/Store instruction.
• for control-flow instructions change the value of the Program Counter (PC) register.

2.1.4.4 Memory access

In this stage Load and Store operations are performed. Load and store operation are generally multicycle in that case we have to stall the pipe till these operations are completed. If the instruction is not load/store then it will just bypass the memory stage and go to write-back buffer directly.

2.1.4.5 Write-back

This reads the data from imem_wb buffer and updates the register file. If any instruction is waiting for the updated result of a register then that instruction can only execute when the register is updated from write-back stage.

2.1.5 Pipeline hazards

In the previous section we assumed that our processor is ideal each stage takes equal time and processor never stalled but in reality a processor will have many stalls, a stall is when the processor can not execute the next instruction because of some exceptions or dependence of present instruction on previous instructions.

2.1.5.1 Types of Hazard

• **Data Hazard** - Occurs when present instruction depends on the result previous instruction.
• **Structural Hazards** - Occurs when there is resource conflict.
• **Control Hazards**- Arises due to branches and other instructions which can change program counter.
We will discuss mainly on the Data Hazards.

2.1.5.2 Data Hazards

These occurs when there are some register dependency between present and past/future instructions. There are three types of data hazard, a hazard can only occur if the register value is being updated and that register is being used by future/past instruction, that's why RAR is not a hazard.
Lets assume two instruction A and B where A should execute before B. Now
• **RAW (Read After Write)** hazard occurs when B tries to read a register in which A has yet to update the value so B gets the old value which is incorrect so there should be stall till A updates the register.
A: R3 <– R1 op R2
B: R5 <– R3 op R4
• **WAR (Write After Read)** hazard occurs when B tries to update a register from which A has yet to read. So B should not update the value till A has read the register.
A: R3 <– R1 op R2
B: R1 <– R4 op R5
• **WAW (Write After Write)** occurs when B updates the same register before A.
A: R3 <– R1 op R2
B: R3 <– R4 op R5
• RAR(Read After Read) is not a hazard and it wont stall the pipe.
RAW is also called True dependency and they are always need to be obeyed.
WAW is called Output dependency and WAR is called Anti dependency, these exits due to limited number of architectural registers. They are dependence on a name, not a value.

2.1.5.3 Dependence handling

Anti and output dependencies are easier to handle: write to the destination in one stage and in program order.
RAW or True dependencies are little difficult, few ways to handle them are:
• Detect and wait until value is available in register file, this will create many bubbles in the pipeline.
• Detect and forward/bypass data to dependent instruction, we forward the data after execution back to the register decode stage and if any other instruction is waiting for the

register value then that instruction can execute now, this way we save bubbles in pipe.

2.1.6 From scalar to superscalar

As we know the Moore's law is still being realized by the engineers so the number of transistors in the same are is increasing, that means we have more hardware in the same are area it enables us to design different architectures which will improve the processor throughput. In superscalar pipelined architecture we replicate the hardware of single issue pipeline processor so that we can issue and execute more than one instruction per cycle this increases the throughput and so the performance of processor.

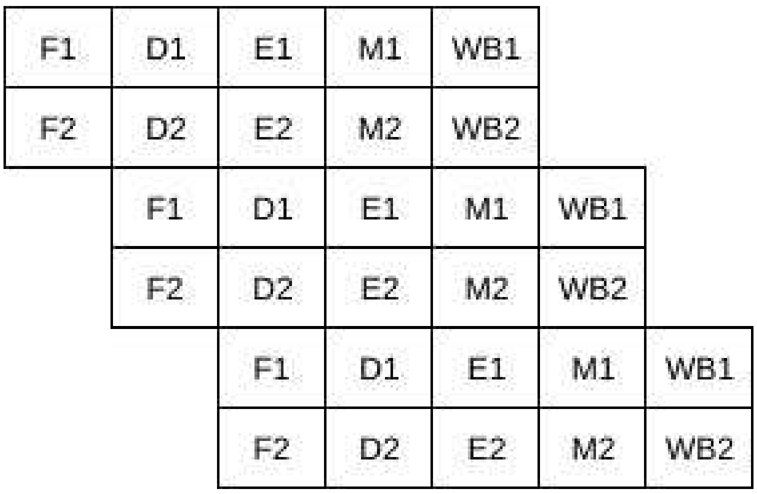


Figure 2.5: Instruction processing in a pipelined superscalar processor

Figure 2.5 shows an ideal superscalar processor in which each stage of the pipeline is duplicated and instruction are independent of each other so pipe is not stalling, but in reality its very hard to replicate all the stages and also there will be many stalls due to the instruction dependencies.

2.1.7 Superscalar processor design

We will now look into details how do we design a superscalar pipeline from a single issue pipeline and also problems in doing so. We are designing an in-order superscalar processor, these are less complex so have less hardware overhead but still can give some improvement over single issue.

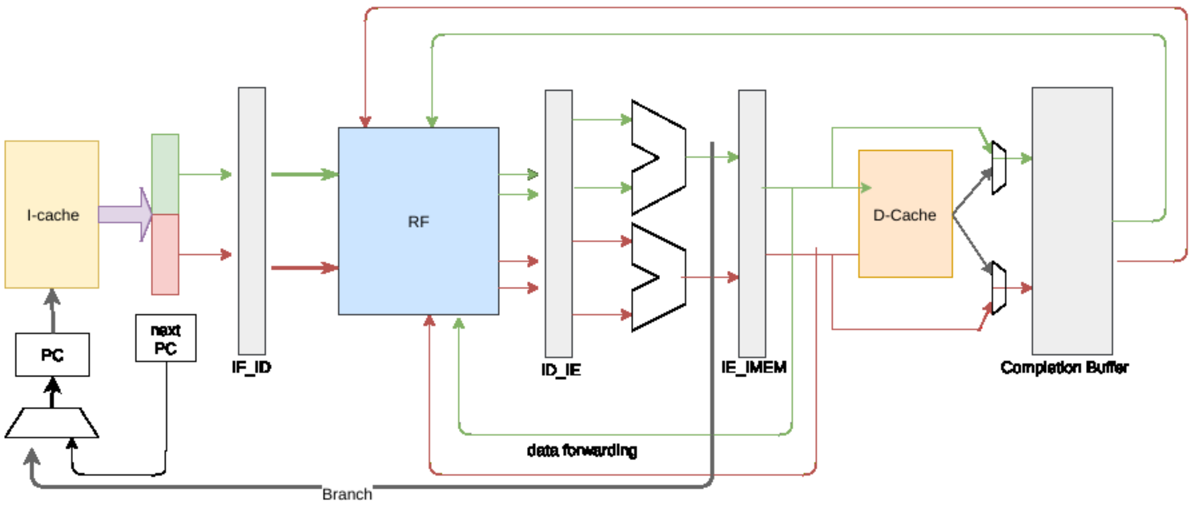


Figure 2.6: A dual issue pipelined processor

Figure 2.6 shows 5-stage dual issue superscalar processor we will discuss the problems in implementing each stage.

2.1.7.1 Dual Issue Fetch Unit

The first step in making a dual issue processor is to fetch two instruction from the instruction cache and enqueue them into the next stage buffer. But fetching two instruction from cache is not so simple, if the PC is for the last block of the cache line then the cache can only send one instruction because for next instruction it will have to read from memory. There are some cache designs to overcome this problem but in our processor we only take one instruction if the PC is for last block of the cache.

2.1.7.2 Dual Issue Decode

Problem is decoding two instruction- decoding two instructions is easy if we have fix length instruction i.e. RISC, we just need two decoders.

2.1.7.3 Dual Execution

We need two ALUs its not so expensive to add ALU but if there are floating point multiplier of divider then its very costly to duplicate them as multipliers and dividers are very big in term of hardware, in those cases we make one execution unit to have both ALU and FP units and other execution unit to have only ALU, but in this case we will need a dispatch unit to send the instruction appropriately into the correct execution unit. In our processor we are fetching operands in execution stage so the Second problem is dependence check between the instructions which generates the stall signal.It is difficult if there is data forwarding from memory and writeback stage, we will have to compare source registers of 1st and 2nd instruction with the destination register forwarded and also the we need to check the dependence of the 2nd instruction on the first one, because only if the source of 2nd instruction is not using the destination register of 1st then only 2nd can be executed. The data forwarded from memory will be compared first then the data from the writeback stage and if the register is not in these stages then the data will be taken from register file, and this will be done for all the 4 sources so it uses a lot of comparators.
Example -
if(rs1(D1) = rd(M1)) then rs1(D1) gets the rd(M1) data
else if (rs1(D1) = rd(M2)) then rs1(D1) gets the rd(M2) data
else if (rs1(D1) = rd(WB1)) then rs1(D1) gets the rd(WB1) data
else if (rs1(D1) = rd(WB2)) then rs1(D1) gets the rd(WB2)
else rs1(D1) gets the data from the register file
this will be done for rs1(D1),rs2(D1),rs1(D2) and rs2(D2), along with these comparisons we also need to compare the rs1(D2) and rs2(D2) with the destination of 1st stage.

2.1.7.4 Dual Memory access

For dual memory access we need different type of D-cache, but in our processor we are only using single memory access we have superscalar design till execute stage only as dual memory access in same cycle is not yet possible for our cache design.

2.2 Bluespec System Verilog

2.2.1 Building a design in BSV

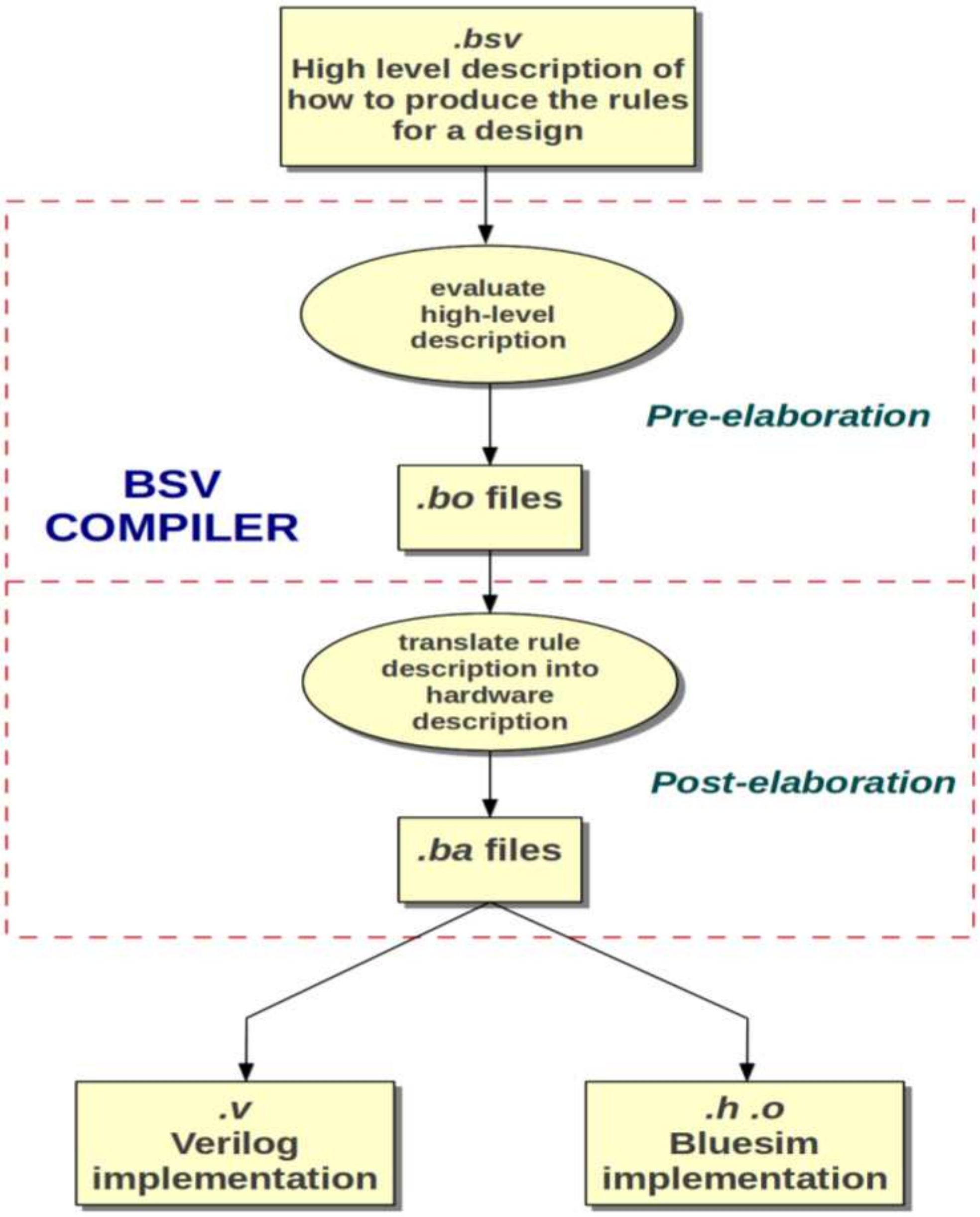

.bsv
High level description of how to produce the rules for a design
evaluate high-level description
Pre-elaboration
BSV COMPILER
.bo files
translate rule description into hardware description
Post-elaboration
.ba files
.v
Verilog implementation
.h .o
Bluesim implementation

Figure 2.7: Building a design in BSV

• Designer writes a BSV program. It may optionally include Verilog, SystemVerilog, VHDL,and C components.
• The BSV program is compiled into a Verilog or Bluesim specification. This step has two distinct stages: 1. pre-elaboration - parsing and type checking 2. post-elaboration - code generation
• The compilation output is either linked into a simulation environment or processed by a synthesis tool.

2.2.2 Rules

BSV does not have always blocks like Verilog. Instead, rules are used to describe all behavior (how state evolves over time). Rules are made up of two components: **Rule Condition:** a Boolean expression which determines if the rule body is allowed to execute("fire").
Rule Body: a set of actions which describe the state updates that occur when the rule fires.

We can logically think of a rule's execution as instantaneous, complete and ordered w.r.t execution of all other rules.
Instantaneous:
• Conceptually, all the actions in the rule body occur at a single,common instant - there is no sequencing of actions within a rule.
Complete:
• When fired, the entire rule body executes. There is no concept of "partial" execution of a rule body.
Ordered:
• Each rule execution conceptually occurs either before or after every other rule execution, but never simultaneously.

Some constraints are imposed on rules. These constraints are:

• Each rule fires at most once within a clock.

• Certain pairs of rules, which we will call conflicting, cannot both fire in the same clock.

2.2.3 Module hierarchy and Interfaces

In BSV, a module's interface is an abstraction of its verilog port list. In BSV, the interface declaration and module declaration are separate. So, a common interface can be used by several modules, with out having to repeat the declaration in each of its implementation modules.

An interface declaration specifies the methods provided by every module that provides the interface,but does not specify the methods implementation. The implementation of the interface methods can be different in each module that provides that interface. The definition of the interface and its methods is contained in the providing module. BSV classifies interface methods into three types:

• **Value Methods:** These are methods which return a value to the caller, and have no "actions"(i.e., when these methods are called, there is no change of state, no side effect).

• **Action Methods:** These are methods which cause actions (state changes) to occur. One may consider these as input methods, since they typically take data into the module.

• **Action Value Methods:** These methods couple Action and Value methods, causing an action (state change) to occur and they return a value to the caller.

Every module uses the interface(s) just below it in the hierarchy. Every module provides an interface to the module above it in the hierarchy.

CHAPTER 3

IMPLEMENTATION

In this chapter I'll discuss in detail the design and implementation of different units to change the C-Class single issue processor into a 5-stage in-order dual issue processor.

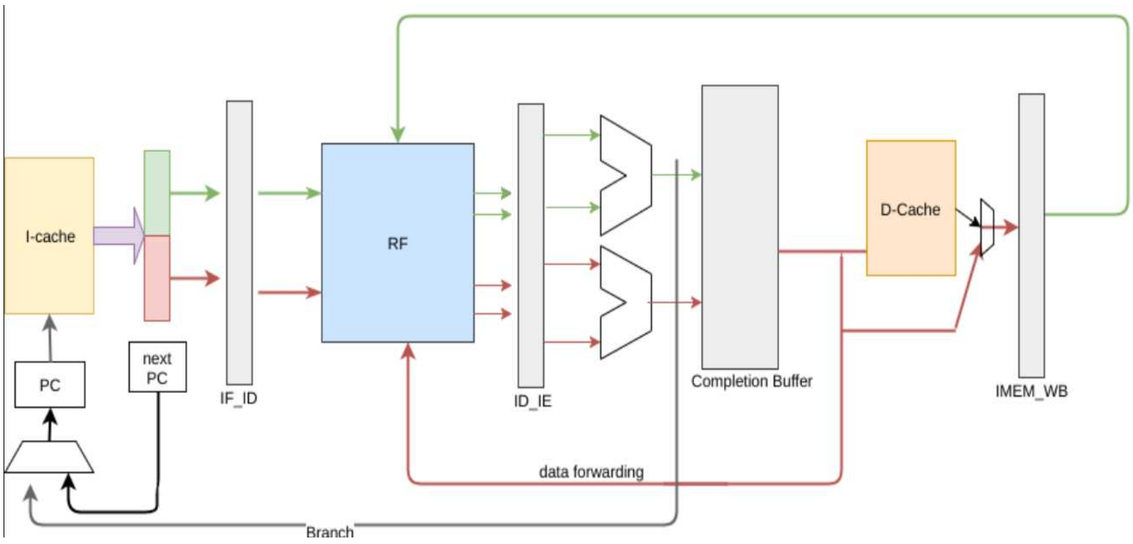


Figure 3.1: Designed dual issue processor pipeline

Figure 3.1 shows the dual issue pipeline which I implemented. It's not an ideal dual issue, it has dual superscalar pipeline till execution stage only. This is because we can only handle one memory operation at a time and also we are committing only one instruction at a time. But we are issuing and executing two instructions in parallel. The design and implementation of each stage will be discussed in detail in the next sections.

3.1 Fetch Unit Design

For making a dual issue processor first we need two instruction from the cache and then the processor should move these instruction into the decode stage and also the next PC calculations should be done perfectly, so let's start with the design changes I made in instruction cache to send two instructions.

3.1.1 Cache Design for Dual issue

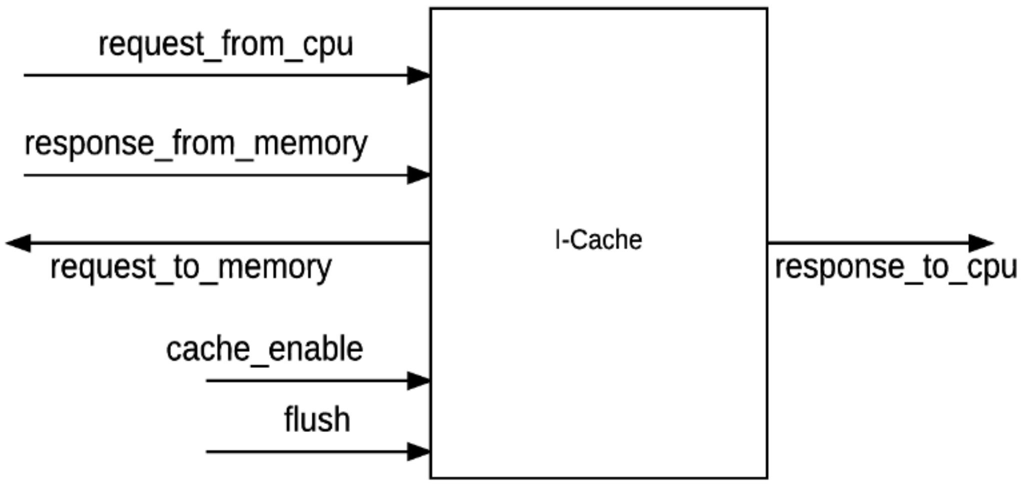


Figure 3.2: Cache interface

Changes in I-Cache

1. Data line for CPU response from i-cache changed to 64 bit , to receive two instructions.
2. Upper and lower offset calculation algorithm changed.
3. Simultaneous_word_send algorithm changed and added a 64 bit register in simultaneous_word_send rule to store and send two instructions.

Changes in detail

1. CPU data line width is increased to accommodate two instructions from i-cache. In terms of hardware number of wires increase.

2. After getting the address from the processor we need fetch instruction at that address but now we need two instruction so the lower offset and and upper offset of the cache line should be 64 bit wide, previously it was 32.
So now we have, lower_offset=fromInteger(v_word_size)*8*fromInteger(i)
and the upper_offset = lower_offset + 63
but this is true only if the address is not for the last block of the cache line because then we can only give one instruction so in that case what I'm doing is, I'm just sending the 6th block and 7th block data but in the fetch unit of the processor it will only take the upper 32 bits if the PC is for last block of the cache and for detecting the address if it is for last block we compare the [4:2] bit of PC to 3'b111, if [4:2] bits are 111 then the request is for 7th block of the cache line and in that case we only get one valid instruction.

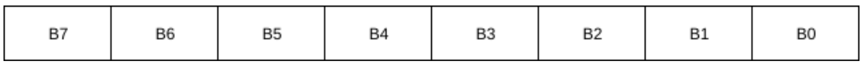


Figure 3.3: Single cache line and blocks

3. If there is miss for the address then we have to fetch the instruction from the memory, and for that we send the current address to the memory and fetch the 8 blocks of data to the cache. While doing so we can directly send the data for the requested address to the processor when we get that block but now we need to send two instructions so we have to wait for one cycle and also store the previous value in a register so that when next instruction comes we send both to the processor. Here also we have to take into account if the address is for the last block then we can only send one valid instruction. These changes are implemented in rule called simultaneous_word_send in the i-cache. In this I have added a 64 bit register " rg_mem_resp_data " of Maybe type. When there is miss and cache is reading from the memory in blocks, when the correct block matches with the address, it stores that instruction into the register and tag the register valid so when in the next cycle it checks if the register is tagged valid it will get a hit and then it puts the 2nd instruction in register too and enq the data into the response_to_cpu fifo.

3.1.2 Fetch Unit changes

The fetch unit changed to take two instructions from the I-cache and enq them into the IF_ID buffers (FIFO between Fetch and Decode stage), as now we need two FIFOs for two instructions they are named ff_if_id_1 and ff_if_id_2.

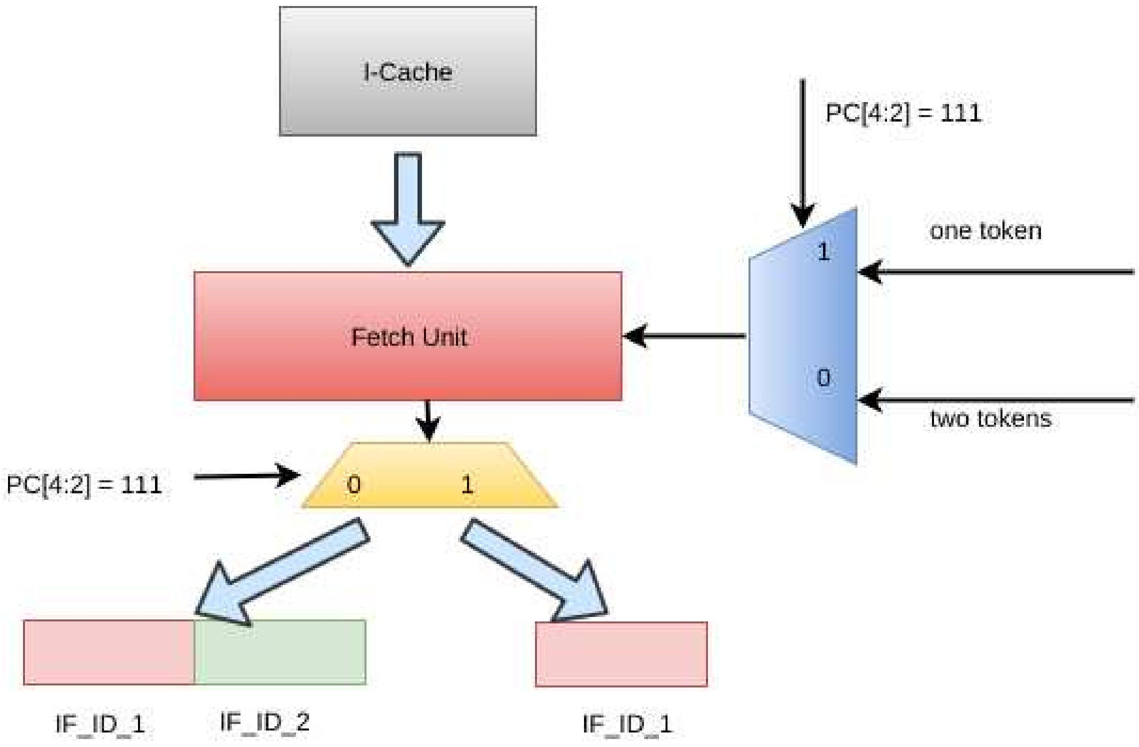


Figure 3.4: Fetch Unit

Figure 3.4 shows the design of my fetch unit, as we can see there are tokens coming to the fetch unit, these tokens are attached to the instructions and when the execution unit gives the result we write back the result at the specific token. These tokens are issued from a Completion buffer whose design I'll explain next. The result from the completion buffer can only be retrieved in the token order and the token order matches the PC order so this makes sure that we write in correct program order into the register file.

Here also we check if the PC[4:2] = 111 then we only get one valid instruction from the cache and so we only enq one instruction in decode FIFO and PC also in this case will increase only by 4 but if PC[4:2] != 111 then the next PC will be current PC + 8 if there is no branch.

3.2 Completion Buffer Design

A CompletionBuffer is like a FIFO except that the order of the elements in the buffer is independent of the order in which the elements are entered. Each element obtains a token, which reserves a slot in the buffer. Once the element is ready to be entered into the buffer, the token is used to place the element in the correct position. When removing elements from the buffer, the elements are delivered in the order specified by the tokens, not in the order that the elements were written.
The Completion Buffer in bluespec library can issue only one token at a time and also there is not method to partially clear the buffer and for our design we wanted a CompletionBuffer which can issue two tokens, can write results at two tokens, can give results of one or two registers depending on method called and also has a method to clear it partially which is used in case of branch to clear the result already stored in the buffer.

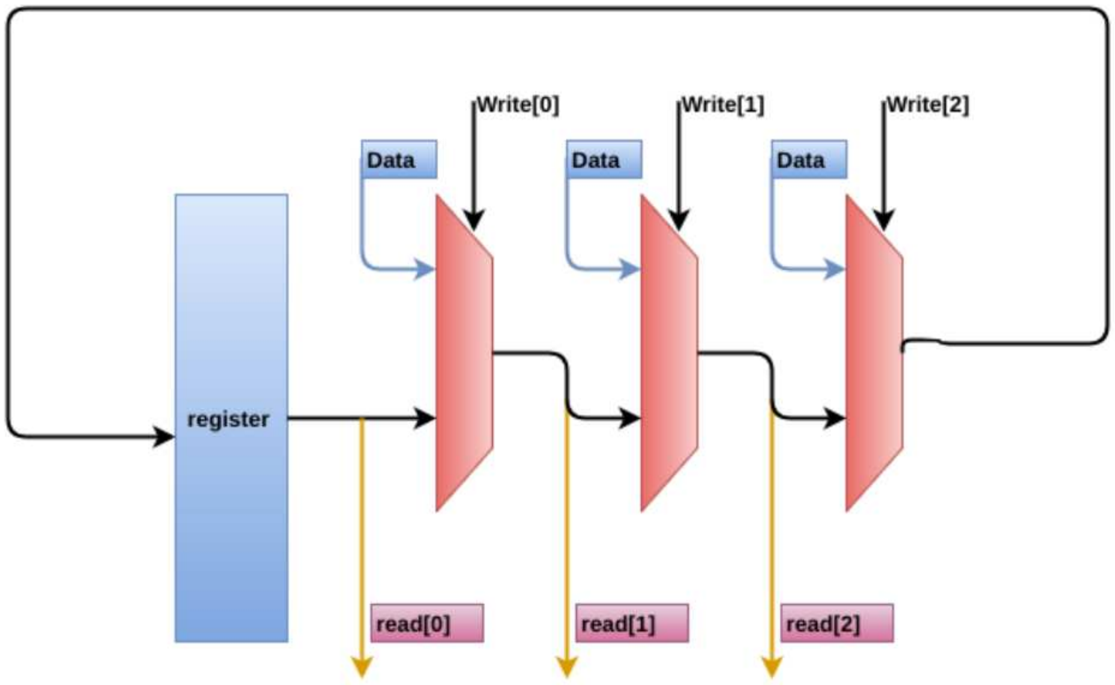


Figure 3.5: Circuit implementation of CReg

The basic register modules in bluespec have scheduling annotations that do not allow two rules to read and write a register concurrently (that is, sequentially in the same clock cycle). But we wanted our buffer to have read write access to a register in the same cycle so we used CReg family registers. These are concurrent register in these registers values written by one rule are visible to another rule in the same cycle so one rule can write and another rule can read in the same clock cycle.

CReg are used to design the required completion buffer, this section explains the implementation details.

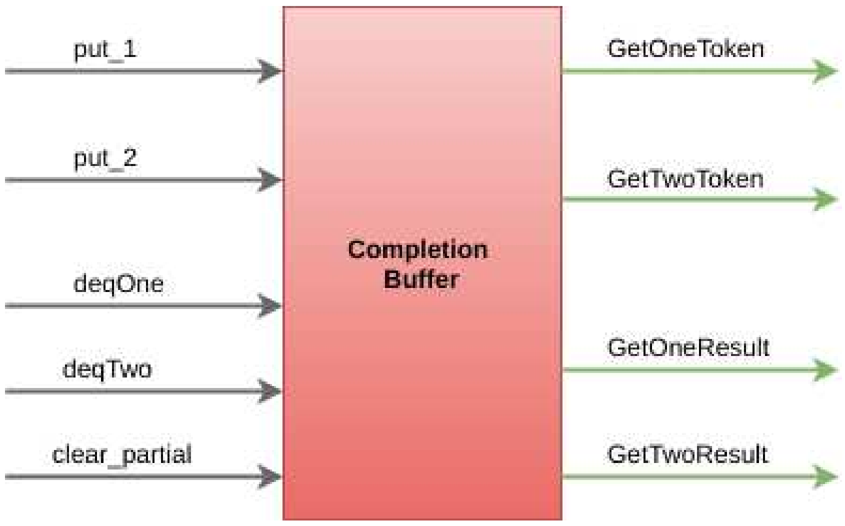


Figure 3.6: Interface of CompletionBuffer

Figure 3.6 shows the interface of the CBuff(CompletionBuffer). The buffer designed has 16 registers of CReg family and each register is of type "Maybe" with holding the data of type IE_IMEM_type as the buffer is being used between the execution stage and memory stage.
The buffer is being used in between execution and memory stage because the processor can only have one memory access at a time and also only one writeback, but it can execute two instruction in parallel so when both instruction gets executed they will write the result in the completion buffer and then the memory stage will read one value from the buffer in the program order and execute the memory operation on it and then pass in to the writeback stage which will be then written to the register file. The ideal design will have the buffer in between the memory stage and the writeback stage but its not possible with the current memory and register file design.

Details of the interface methods:
• **GetOneToken and GetTwoToken :** as name suggest these two methods are used to get the token from the buffer, there is a register pointer named iidx in the buffer which stores the latest token number of the register which has not been issued yet so if the

request is to issue one token it will give the present value and increment itself by one, it also invalidates the current register issued which will be validated when the processor writes the result at that token. Similarly if the request is for two tokens the pointer will give present value and the next value and invalidate both the registers. But this rule can only give the value if the head pointer(from which it gives the result) named "ridx" is not equal to the present tail pointer + 1 or tail pointer + 2 for two tokens.

• **put_1 and put_2 methods :** These methods are used to write the result back into the completion buffer at the specific token, these methods take data and token as the input and write the data at the token and validate the register so that the head pointer can read the register and give the data. The processor can write the value at any issued token in any order through these put rules and they will validate that registers but the result can only be read if the top token is tagged valid where the head is pointing to.

• **GetResult methods :** theses methods are used to get the result from the buffer, the result from the buffer can only come in the token ascending order processor can write result in any any order but this method will only fire if the current head token in tagged valid. Buffer can give one result or two result according to request, in present design the processor is only using to get one result but in future modifications we can use it to get two results. Two results will only be given if the present and the next registers are tagged valid.

• **clear_partial method :** this method is used if we want to clear the registers. In this case clearing the register means tagging it invalid because if the register is tagged invalid the get result rule can't read its value. This method also resets the tail pointer because the cleared register can be reissued now. The input to this method is the clear index from which we want to start cleaning. This method is called by execution unit when there is branch, then it resets the tail pointer.

3.3 Decode Stage Changes

Changes in decode stage are simple compared to other changes, in this stage we just need to add and extra decoder which decodes the 2nd instruction and then we enq the decoded data into the execution buffers. As the designed processor is completely in-order so the enq in both execution stage buffers will happen together which means both previous instructions has finished their execution and data has been stored in the completion buffer then only decode stage can enq.

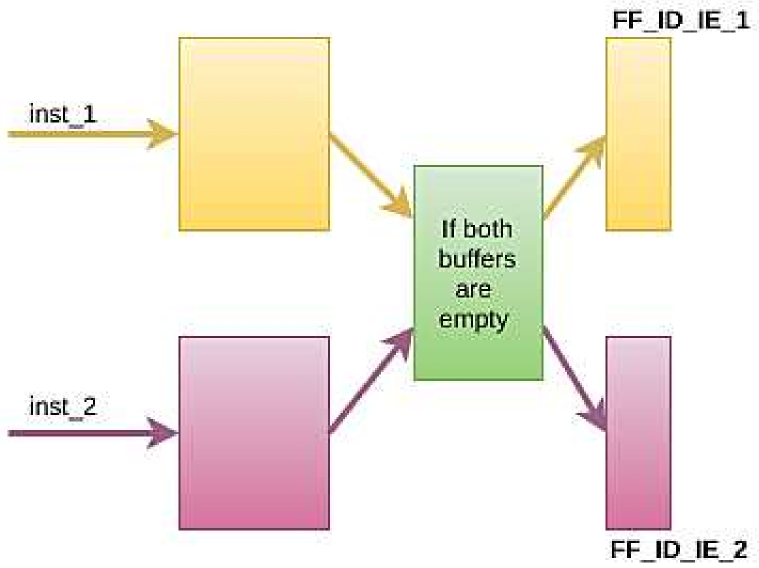


Figure 3.7: Decode stage design

3.4 Register File changes

Major changes are done in register file because it is where processor checks the dependencies, remove the hazards and get the data to the source registers. The register file has been modified to remove RAW, WAW and WAR as the modified design is a dual issue so these hazards will come while in single issue processor we don't explicitly need to take care of these, they will be automatically taken care by the pipeline as the instructions are executing one at a time and completing in strict program order.But in dual issue processor the execution is happening in parallel and the register value is being stored in the completion buffer which is not visible to the register file.

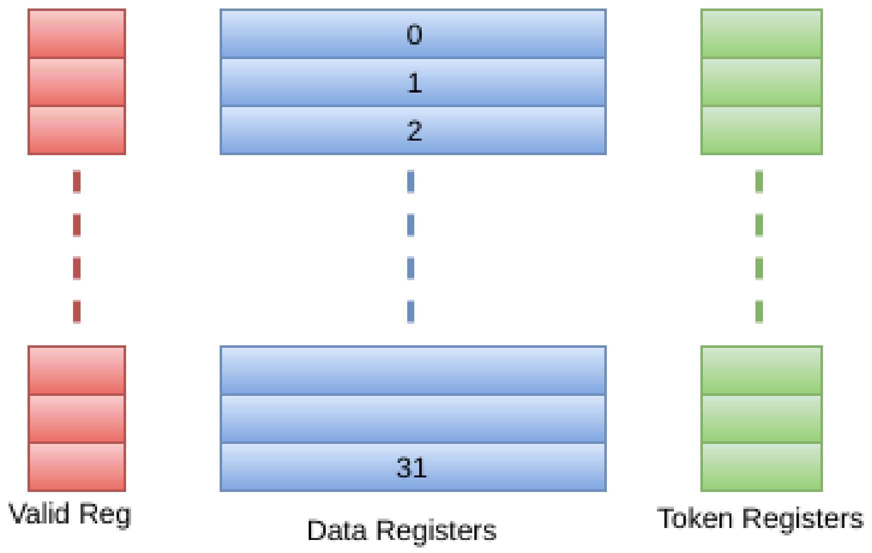


Figure 3.8: New register file design

Figure 3.8 shows the new register file design. Now all the register has been changed to CReg type to implement concurrency so that we can compare second instruction with the first instruction and some other facilities can happen in same cycle, these will be explained in working of register file. As we can see in the figure, there are three register arrays, these are: 1. Valid registers 2. data registers 3. token registers.

Data registers: these are the registers which stores the data. There are 32 register and these has been changes to CReg type. In previous processor design there was only this data register file but for dual issue design we need valid and token registers also to handle the hazards.

Valid register: These registers are necessary to eliminate the Read After Write hazard. In the present design the register file is not able to read the completion buffer so if a instruction has been completed but the value is still in the completion buffer and some other instruction needs the value of that register then it will read the old incorrect value so we need to tag the destination register invalid till the processor writes back the updated data in it and till then the present instruction will stall.

Token registers: These registers are used to eliminated the Write After Write hazards. If processor completes multiple instructions which are writing to same register then the valid bit has to be validated only when the latest instruction writes in the register, because if it validates the register with previous value then current instruction will get a wrong value.

3.4.1 Working of register file

As the design has been changed for dual issue the register file now gets request from two instructions that means 4 source register reads. For 1st instruction the read is simple it will check the forwarded data from each stage and if the destination matches the source it will get the data or else it will get the data from the data register file given that the register is tagged valid. Once both source operands of the 1st instruction are read the destination of that instruction is tagged invalid with the present token attached to it.

After reading the 1st instruction operands, the second instruction operands are fetched, but here it has to check if the destination of 1st instruction is source for the 2nd instruction then the 2nd instructions can't get the data, else we start the matching like the 1st instruction. we match the sources with the forwarded data from each stage and if they don't match we get the data from the data register file if that register is tagged valid.
If the destination of the 2nd instruction is same as of the 1st instruction the token attached to it will be of the 2nd instruction.

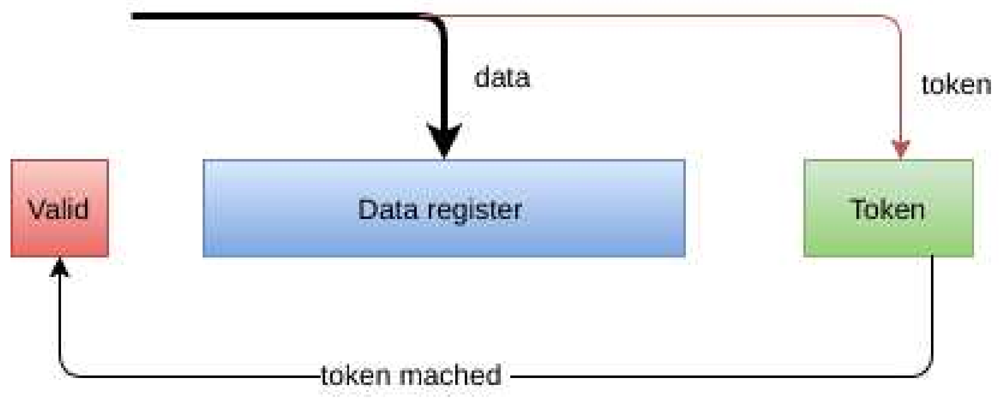


Figure 3.9: Data write back in register file

Figure 3.9 illustrate the working of the write back in register file. We can see the data will be written in all the cases but the Valid bit will only be validated if the token attached to the data coming from writeback stage matches the token stored for that register.

3.5 Execution Stage Changes

This is the stage where both instructions will be executed and get the result of the operation, the result is stored in the completion buffer. The enq in execution buffers always happen together and only after enq in execution stage the decode buffers will be dequeued so if any instruction stalls here the pipe will not move till that instruction is completed. After enq in the EXE buffers it will fetch operands from the RF and do the operation, there can be a branch misprediction by any of the two instruction but as this is completely in-order if the 1st pipe gets misprediction it will clear the 2nd execution buffer and call to the clear_partial method of the completion buffer which will reset the token issue head. But if the misprediction is in the 2nd pipe it will not clear the 1st execution buffer as the instruction in the 1st buffer is the previous instruction, this will also call to same clear_partial method. If both of them have branch then the 1st pipe will get priority and will clear the 2nd pipe. On branch misprediction both of the fetch and decode stage buffers will also be cleared.

3.6 Memory Stage Changes

The fifo between execution and memory stage has been removed, now there is completion buffer in place of it. The memory stage will read the completion buffer, one register at a time and execute the memory operation if it's Load/Store operation else it will bypass the result and enq it into the writeback buffer.

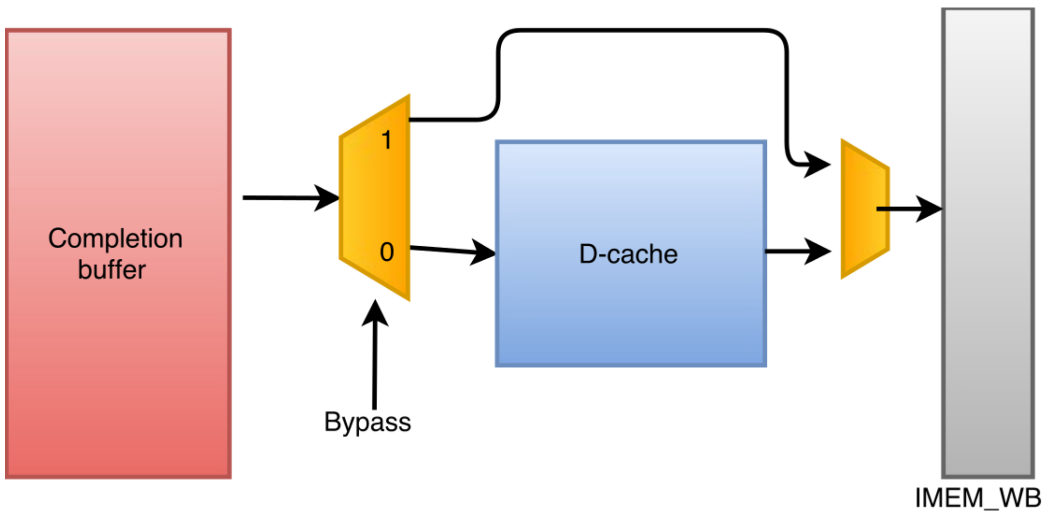


Figure 3.10: Memory stage

3.7 Write Back Stage

There are no logical changes in the write back stage, the only change is while sending the result back to the register file it will also send the token number attached to the result, in the register file it will be compared with the token stored for that register and if it matches the valid bit in register file will be validated.

CHAPTER 4

VERIFICATION AND RESULTS

This chapter describes how the new processor design is verified in Bluespec, it also compares the result of the dual issue processor with the single issue design.

4.1 Verification Flow

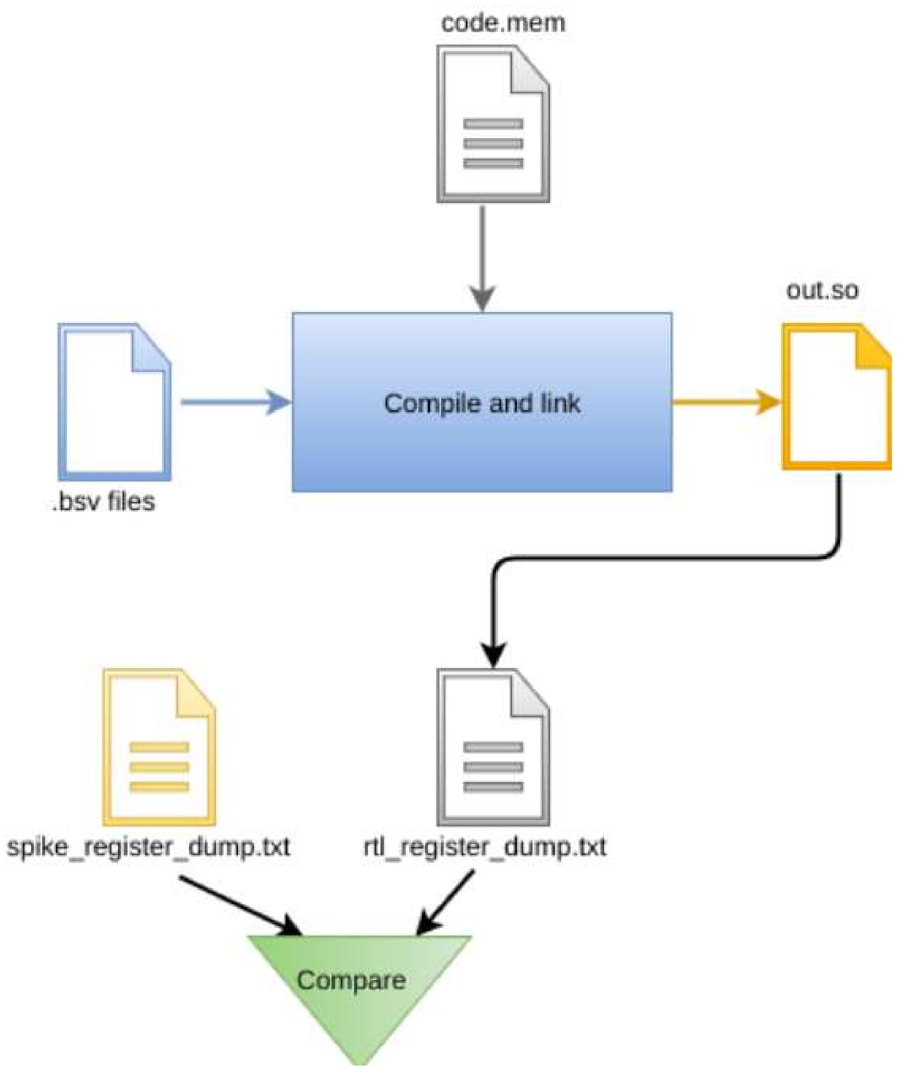


Figure 4.1: Verification flow

Figure 4.1 illustrates the verification flow. We write the code in Bluespec which has file extension of " .bsv ". The "code.mem" file contains the instruction, these are generated using AAPG. We give the .bsv files and code.mem to the Bluesim compiler and linker and it generated the "out.so" file. Now from out.so we get the register_dump.txt which contains the register file values for each instruction. Finally we compare this register_dump with the golden output generated using spike, if both are identical then our processor has worked correctly.

4.2 Debugging the code

After comparing the rtl_register_dump.txt file with the spike_register_dump.txt if I get some register mismatch between the two files then I followed these steps to debug the code :

1. Note down the program counter from where the error starts.
2. Note down the register number for which wrong value is showing.
3. Now from the log file check at that program counter why the register file is being updated with the wrong value.
4. Make changes in the code and repeat till both register files are identical.

4.3 Results

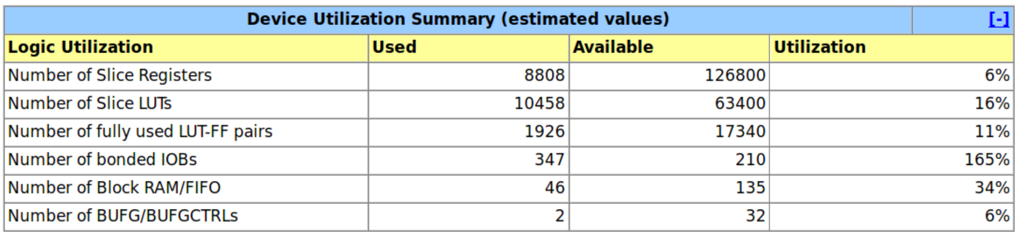
When both rtl_register_dump.txt and spike_register_dump.txt are identical then we know the processor has executed the instructions correctly. Now we compare the number of cycles it took to complete the program and compare it with the original processor. As a superscalar processor exploit the Instruction-level parallelism in the program so some program which has more ILP will give better results than other.

Test results:

	Test1	Test2	Test3
Improvement	5.6%	3.1%	4.5%

Synthesis Report:

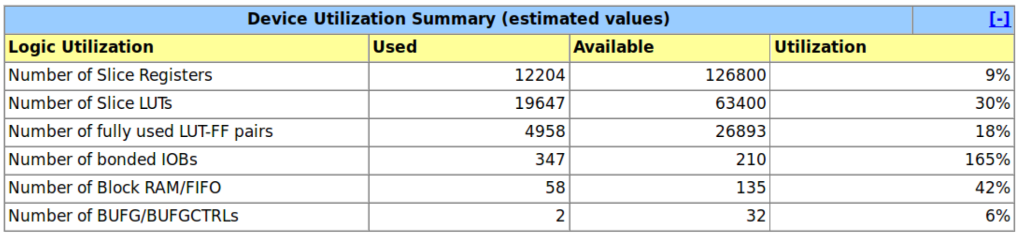
Current Processor:



Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	8808	126800	6%
Number of Slice LUTs	10458	63400	16%
Number of fully used LUT-FF pairs	1926	17340	11%
Number of bonded IOBs	347	210	165%
Number of Block RAM/FIFO	46	135	34%
Number of BUFG/BUFGCTRLs	2	32	6%

Figure 4.2: Device Utilization

Designed Dual Issue Processor:



Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	12204	126800	9%
Number of Slice LUTs	19647	63400	30%
Number of fully used LUT-FF pairs	4958	26893	18%
Number of bonded IOBs	347	210	165%
Number of Block RAM/FIFO	58	135	42%
Number of BUFG/BUFGCTRLs	2	32	6%

Figure 4.3: Device Utilization

Timing:

Current processor : Maximum frequency- 84.765 MHz

Dual issue processor : Maximum frequency- 54.811 MHz

This is due to the critical path length increase in the operand fetch stage, as now the processor needs to fetch operands for two instructions.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

An ideal superscalar pipelined processor will double the throughput compared to a single issue processor but in reality a superscalar processor exploits the Instruction Level Parallelism of the program, which varies widely depending on the type of program, so it did not show the improvements we expected but surely it showed some improvements over the single issue design. A superscalar design comes on the cost of hardware but as the Moore's law says, number of transistors double every 18 month so the hardware overhead can be managed if the performance improvement is good.

In this processor the fetch, decode and execute stages are dual issue but the memory and writeback stages are still single issue. It has a completion buffer after the execution stage which works based on token mechanism. The register file is also changed to handle different types of hazards. The processor completed all the programs correctly and showed improvements compared to its single issue design.

5.2 Future Work :

The designed processor is still not completely dual issue due to limitation of single D-cache access and Writeback stage CSR access and also as the processor has become dual issue the operand fetch from register file has increased the critical path. There are good projects to improve this design further :

1. Make an extra stage for operand fetch to reduce the critical path length.

2. Changing D-cache design and making memory stage and write back stage dual issue.

REFERENCES

[1] **J. P. Shen and M. H. Lipasti,** *Modern Processor Design - Fundamentals of Superscalar processors.* McGraw-Hill Publishing Company Private Limited, 2003.

[2] **J. L. Hennessy and D. A. Patterson,** *Computer Architecture (5th Edition).* Morgan Kaufmann, 2012.

[3] Bluespec, Inc, Bluespec System Verilog Reference Guide, revision: 30 ed., 2014.

[4] Wikipedia:ISA
https://en.wikibooks.org/wiki/Microprocessor_Design/Instruction_Set_Architectures

[5] Berkeley pdf: Superscalar Processor
http://www-inst.eecs.berkeley.edu/ cs152/fa16/lectures/superscalar.pdf

[6] Online Course: Introduction to Computer Architecture
http://www.ece.cmu.edu/ ece447/s15/doku.php?id=schedule