Design and Implementation of a 5-stage Scalar In-order Pipelined RISC Processor

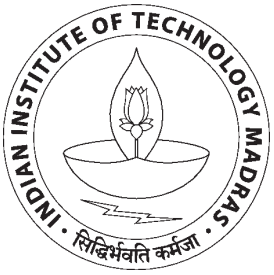
A Project Report

submitted by

SARATH YADAV S

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

MASTER OF TECHNOLOGY



DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.

MAY 2014

THESIS CERTIFICATE

This is to certify that the thesis entitled **Design and Implementation of a 5-stage Scalar In-order Pipelined RISC Processor**, submitted by **SARATH YADAV S**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bonafide record of the research work carried out by him under my 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:

ACKNOWLEDGEMENTS

I Would like to express my deepest gratitude to my guide, ***Dr. 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 through out the project. His valuable suggestions and constructive feedback were very helpful in moving ahead in my project work.

I would like to thank my faculty advisor ***Dr. Deleep R.Nair*** who patiently listened, evaluated, and guided us through out our course.

My special thanks to project team members Neel, Rishi Naidu, Arjun ,Naveen, Chidambaranathan, Sachin, Senthil, Chaitanya, Keerthi for their help and support.

I would like to thank my classmates who made my life in IIT a memorable experience.

Finally, I take this opportunity to thank my parents for their support and encouragement without which learning in and becoming a part of such a prestigious institution would not have been possible.

ABSTRACT

KEYWORDS: Scalar Inorder, Data Forwarding or Bypassing, Hazard Detection

The aim of the project is to Design a 5-stage Scalar Inorder pipelined processor, using Bluespec Verilog HDL. The project involves design of a simple Reduced Instruction Set Compiler (RISC) processor and simulation of it. The 5 stages involved in instruction execution are Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM) and Write Back (WB). The instruction set being used is of 32-bit base integer variant of RISC-V ISA. The RISC-V ISA is a completely open ISA which is freely available for academia and industry. The modules used are Instruction Memory, Data Memory, ALU, Registers etc. The present project work deals with design of Decoder unit, Execute unit, Control path and Data path. I also have implemented a Forwarding (Bypassing) Unit and a hazard detection unit to overcome hazards.

This report focuses upon, basics of RISC, CISC Architectures, concept of pipelining ,Design of RISC Processor, Hazard detection in RISC processor and also the synthesis based results.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

CPU	Central Processing Unit
RISC	Reduced Instruction Set Computer
BSV	Bluespec System Verilog
IF	Instruction Fetch
ID	Instruction Decode
EXE	Execution Stage
MEM	Memory Access Stage
RAW	Read After Write
PC	Program Counter
HDL	Hardware Description Language
ISA	Instruction set Architecture
CISC	Complex Instruction Set Computer
ALU	Arithmetic Logic Unit
RISC	Reduced Instruction Set Architecture
BSW	Bluespec Development Workstation
RV	RISC-V

CHAPTER 1

INTRODUCTION

1.1 Overview of Processor

The Processor design team of Reconfigurable and Intelligent Systems Engineering (RISE) Lab in the Computer Science Dept. of IIT Madras has been actively involved in building a scalar processor. The proposed processor is a single core 5 stage in-order architecture. A number of RISC-V base instructions have been implemented and care has been taken to provide a control mechanism for data, structural and control hazards. It has Data Forwarding and hazard detection logics to avoid different hazard situations in which instructions in a pipeline would produce wrong results. The project work involves designing a 5-stage pipelined RISC processor, using Bluespec System Verilog HDL and to synthesize it using Xilinx ISE. The 5 stages being used are Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM) and Write Back (WB).The focus is to understand the processor design, pipelining and RISC-V ISA.

1.2 Objectives and Problem Statement

- Implementation of ALU for base 32 bit instructions of RISC-V ISA.
- Pipelining of the stages using intermediate register buffers.
- Detection of hazards.
- Implementation of Data Forwarding and Pipeline Interlock logics for resolving hazards.

1.3 Organization of the thesis

Chapter 2 describes about the basics of CISC and RISC Architectures, also the features of both RISC and CISC processors. It also explains about Single Cycle implementation and Multi Cycle implementation of CPU and also comparison between them. It also describes about the HDL called Bluespec System Verilog, its key features, BSV constructs.

Chapter 3 describes about concept of Pipelining and brief description of different stages of Pipelining. It also explains about challenges in pipelining i.e different types of hazards.

Chapter 4 describes about design of Instruction Set Architecture and design of control and data path and also implementation details of 5 stages and also describes about how the processor design is verified in Bluespec System Verilog. It also describes about the test cases generated to verify the design.

Chapter 5 concludes with a short description about the future work.

CHAPTER 2

BACKGROUND

This project describes the design and functional behaviour of RISC processor. Before describing implementation and design of processor, this chapter explains about basics of CPU.There are basically two types of CPU namely RISC based CPU and the other one is CISC based CPU. It also deals about the key features of Bluespec System Verilog (BSV), why Bluespec is used, how to build a design in the BSV.

2.1 Basics of CISC and RISC architecture

Complex Instruction Set Computer (CISC), name itself suggests it has a more number of different multi-clock complex instructions. CISC processor emphasizes on hardware. LOAD and STORE instructions are based on memory to memory transfer operation. CISC processors are relatively slow in comparison to RISC (Reduced Instruction Set Computer) but it gives benefit of usage of less number of instructions.

Compared to CISC architecture, RISC processors are faster. RISC processor emphasizes on software. Now a days CISC processors are very rarely in use. RISC uses simpler and faster instructions that are typically of same size which make RISC processor easier and less expensive to design. All operations are performed data in registers. The only operations that uses memory access are load and store instructions that move data to memory and move data from memory.

2.1.1 CISC Architecture

CISC is based on the idea of integrating Computer Architecture with Computer Science. CISC is basically designed to decrease gap between low-level language and high-level languages. CISC CPUs instructions executed by using Microprogrammed control.

2.1.1.1 Features of CISC Architecture

- They use a 2 operand format basically that of source and destination. Source and destinations can either between register and register (or) register and memory (or) memory and register.
- The instructions are of variable length. The length varies with respect to addressing mode.
- Single instruction can support many addressing modes. Hence complex instruction decoding logic is required.
- The task can be completed by using fewer instructions.

2.1.2 RISC Architecture

RISC is a microprocessor type architecture which uses small and highly optimized set of instructions.In a RISC machine, the instruction set contains simple, basic instructions, from which more complex instructions can be composed.An abstract view of RISC processor is shown in below figure.

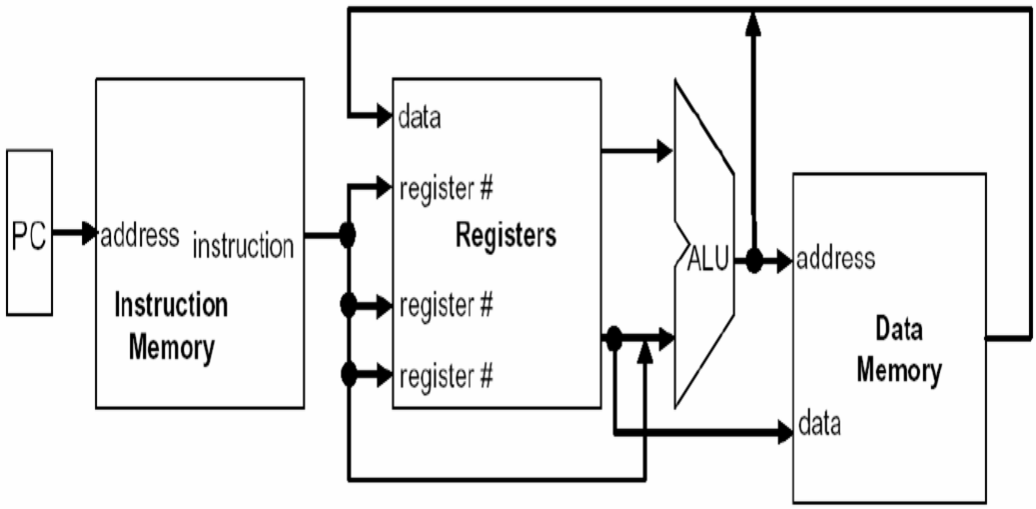


Figure 2.1: Abstract view of RISC Processor

2.1.2.1 Features of RISC Architecture

- Most instructions complete in one machine cycle, which allows the processor to handle several instructions at the same time.
- To avoid more number of memory accesses, RISC makes use of more number of registers.
- RISC makes use of PIPELINING (this is what we basically focus upon in this project). Pipelining allows simultaneous execution of instructions, making the process more efficient.

2.2 Introduction to Single cycle and Multi cycle Implementation of CPU

In order to understand how one can implement the RISC instruction set in pipelined fashion, we should understand how it can be implemented without pipelining and therefore here we will go through the basics of single and multi clock cycle CPU approach [1].

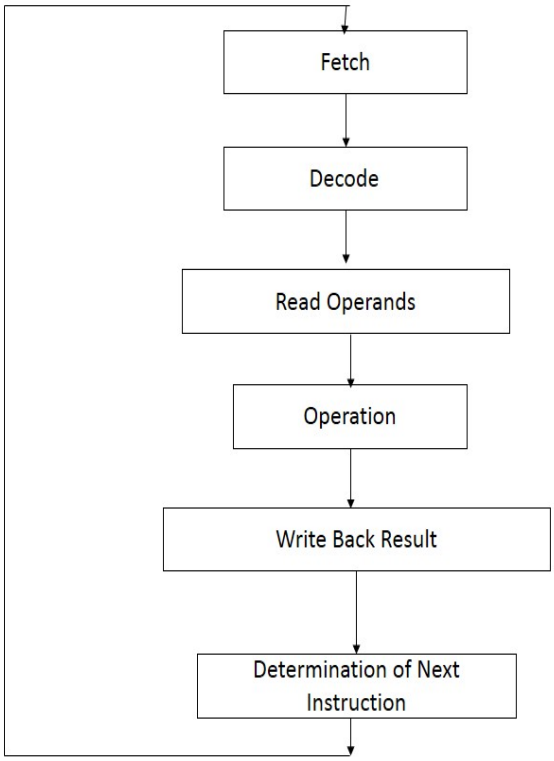


Figure 2.2: Steps of Instruction Execution

2.2.1 Basic of Single Cycle CPU

As name suggests in this type of CPU, it executes all instructions in one clock cycle. In reality each cycle requires a certain amount of time and this mean single cycle CPU spends same amount of time to execute each instruction, basically one cycle no matter how complex is the instruction. In order to ensure the correct operation, the slowest instruction

should be completed within one clock tick e.g. load , which means single cycle CPU operates at the speed of slowest instruction in ISA. Another aspect of this CPU is, since it has to complete all the instructions in one clock cycle means any element must be used once only. So duplication of such an element has to be available.

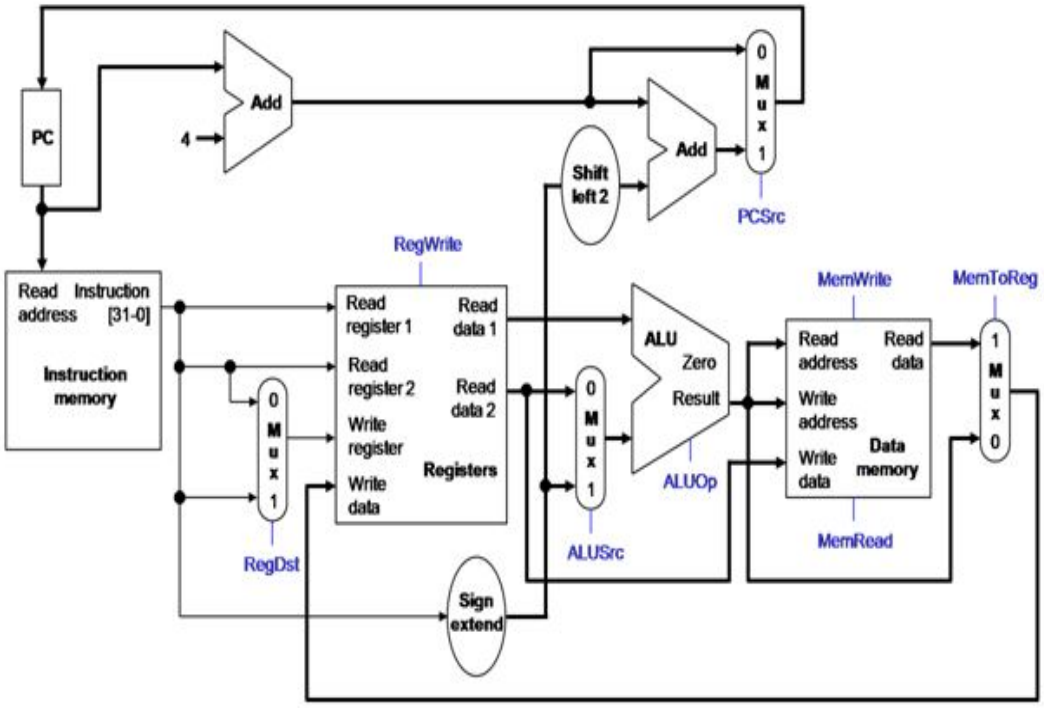


Figure 2.3: Single cycle Implementation

2.2.2 Basic of Multi cycle CPU

As name implies, this kind of CPU requires multiple cycles to execute each instruction, of course this means the CPI will be more than one in this case. The advantage of such kind of CPU over single cycle CPU is that depends upon the complexity of the instruction, more and less number of clock cycles can be used, e.g. load instruction needs 5 cycles in comparison to 3 cycles for branch instruction.

Since the complexity of operation is increased, there must be a control unit and this can be developed using Finite State Machine where as in the case of single cycle CPU it was multiplexers. Like Single Cycle CPU, now in this case since a complexity is increased, total cycle time is determined by slowest operation unit.

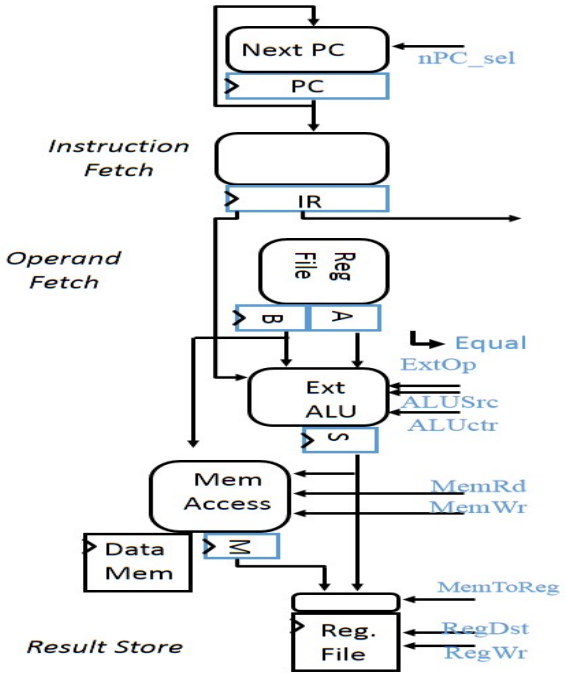


Figure 2.4: Multi cycle Implementation

2.2.3 Comparison between Single Cycle and Multi Cycle

Now let us differentiate between Single cycle and Multi cycle performance with the help of an example.

Example:Assume 10ns is the required time to perform any operations: memory access, register file access and ALU operation. Since 10ns is the required time to perform any operation and we are considering 5 stages of operation for RISC architecture, the total time for single cycle CPU operation will take 50ns.

For Single cycle:

So, Cycles Per Instruction CPI = 1

Performance = 50ns/instruction

For Multicycle:, let us assume a CPU is executing following instructions Branch:20 percentage,Load: 20 percentage and ALU:60 percentage.

Here clock period =11ns(10ns per stage +register overhead)

CPI = (0.2*3+0.2*5+0.6*4) = 4

Performance = 44ns/instruction

2.3 Definitions of Throughput and Latency

Latency: Latency is the total time required to complete one instruction cycle i.e compete instruction execution.

Throughput: Throughput is the number of such actions executed or results produced per unit of time. This is measured in units of Instructions Per Cycle (IPC). Throughput is the reverse of CPI.Goal is to make programs, not individual instructions, go faster.

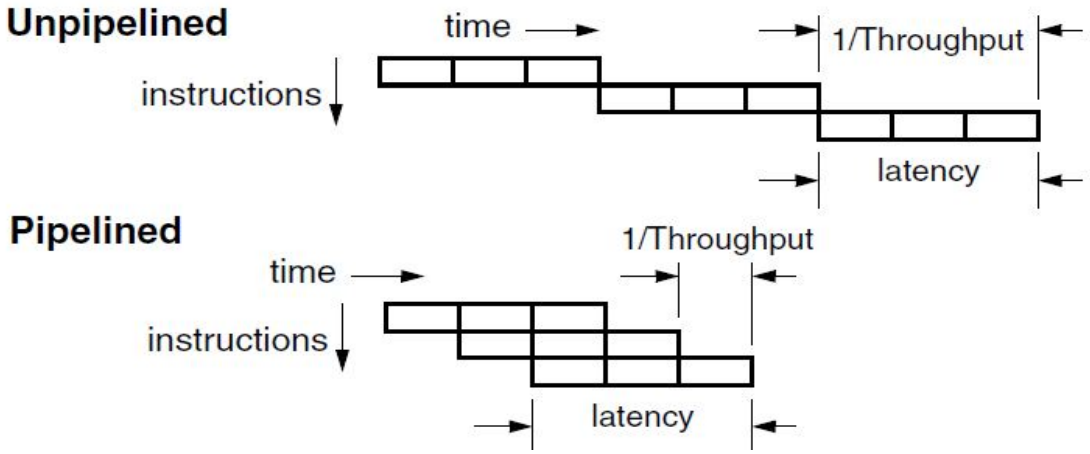


Figure 2.5: Difference between Latency and Throughput

2.4 Bluespec System Verilog

2.4.1 Building a design in BSV

Figure: 2.6 illustrates the various steps involved in building a design in Bluespec System Verilog [4].

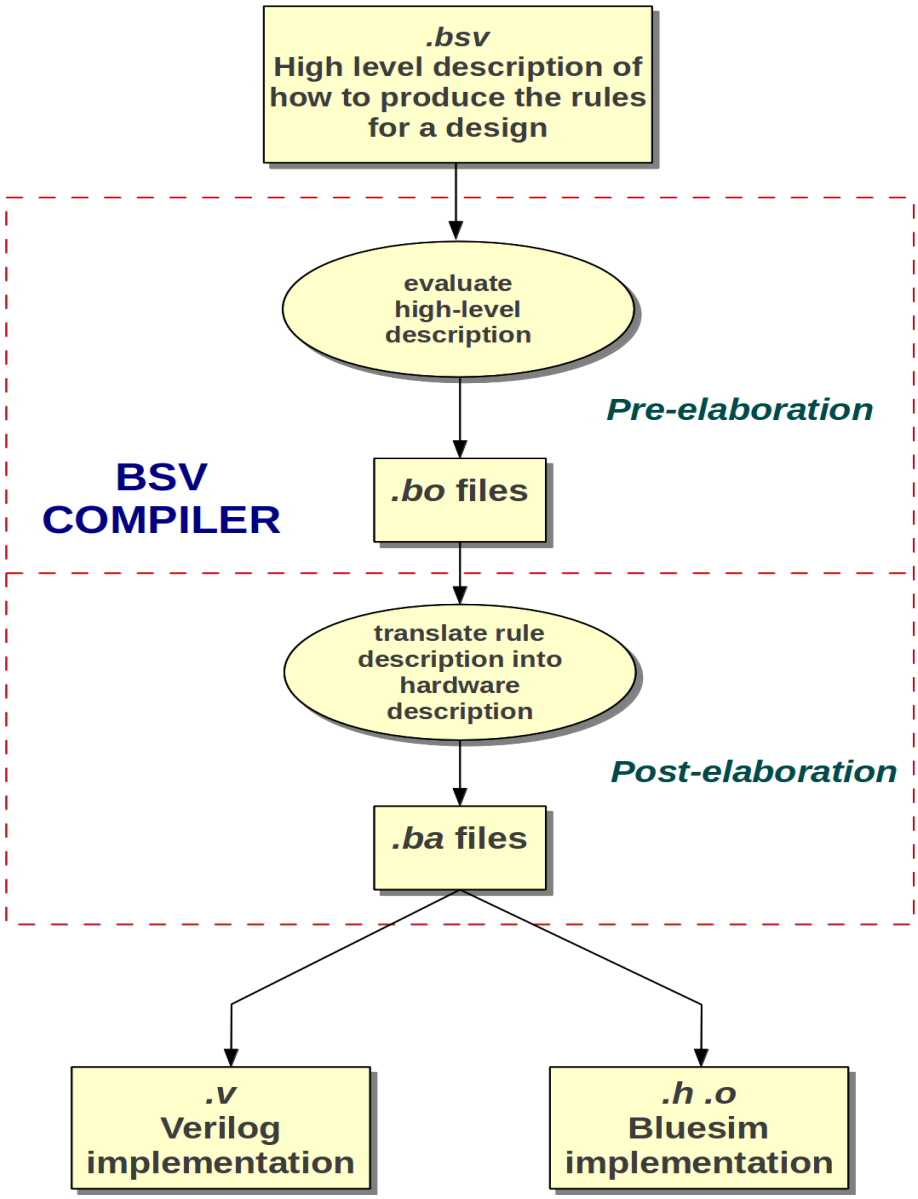


Figure 2.6: Building a design in BSV

- 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.

2.4.2 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.

2.4.3 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

Concept of Pipelining and its Performance Issues

Before going into implementation details of Processor, we need to know what is pipelining and how it speeds up the execution of instructions and also the issues involved in implementation of pipelining.

3.1 Pipelining

Pipelining is nothing but doing more than one operation, in a single data path. A multicycle CPU consists of many processes. For example load might take up to 5 clock cycles, but branch takes only 3 clock cycles. So if one process is taking place, instead of waiting for the process to complete, we can simultaneously start a new process in the same data path, without disturbing the previous process.

For this to happen, each part of the process is divided into various pipelined stages. So after every clock, the process is stored into next pipelined stage, enabling another operation to start in that stage without disturbing the previous process. Hence all the stages in the path can be used simultaneously. This in turn can increase the throughput of your design.

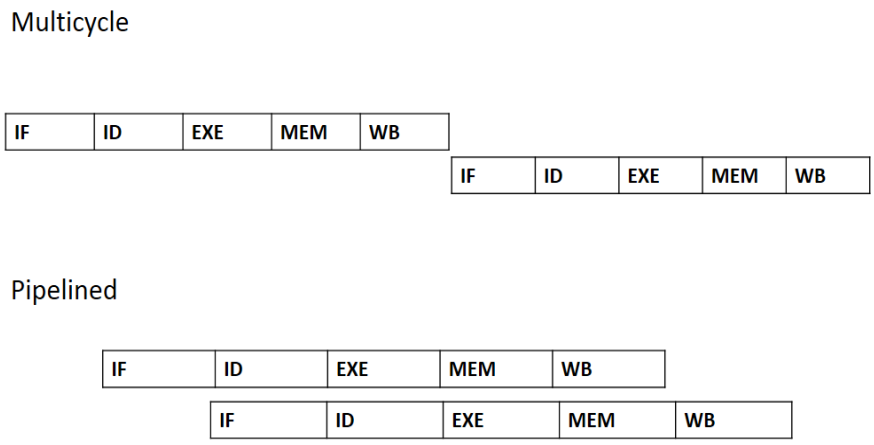


Figure 3.1: Difference between Multi cycle and Pipelined CPU

Now let us briefly discuss about 5 stage pipelined architecture

Instruction Fetch

- Sending PC to memory and fetching the current instruction from memory as well update the PC to next in sequence.

Instruction Decode

- Decoding the instruction and reading the registers as specified in register file.
- For branch instruction, necessary actions have to be taken like calculation of effective address in case of Unconditional Jump instruction.
- Decoding can be done in parallel with reading the registers since the register specifiers at a fixed location, this is called is fixed field decoding.

Execution Unit

- In this stage, mainly ALU operations based on the instruction type.
- For branch instruction, necessary actions have to be taken like calculation of effective address and also condition checking of Conditional Jump instructions.
- For Load/STORE instructions calculation of effective address is done.
- It performs operation for register immediate ALU instructions.

Memory access

- In this particular stage, load and store instructions are being performed.
- If it is a load instruction then it reads an effective address from the memory and in the case of store instruction it writes the data in to memory.

Write Back

- This is the last stage and it performs register register ALU instruction or LOAD instruction to write the result in to register file (at ID stage), to check whether it comes through load instruction or from ALU when it is a case of ALU instruction.

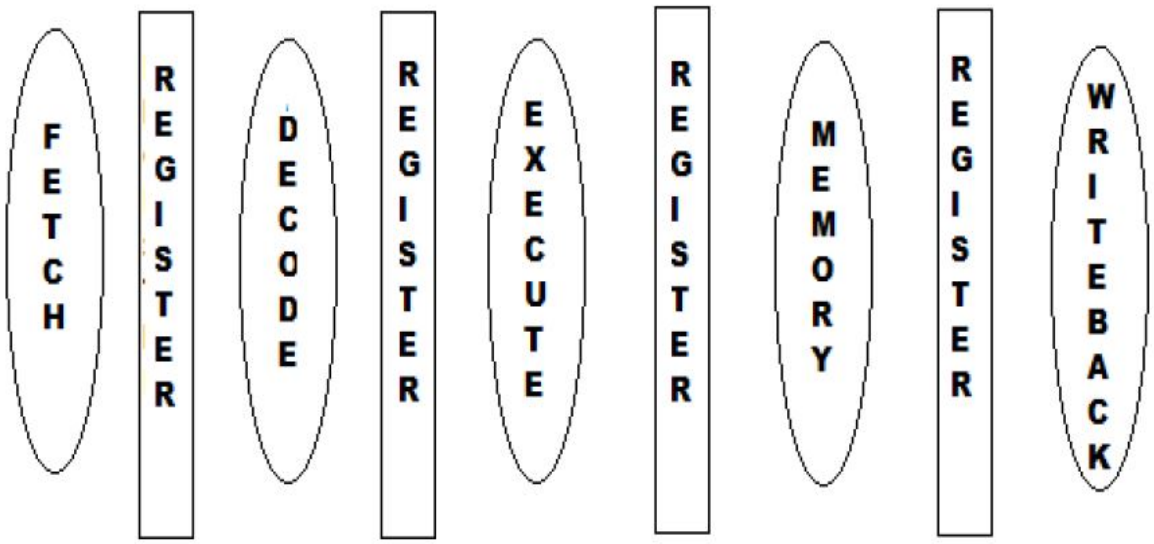


Figure 3.2: Five stage Pipeline Structure

All the five stages are connected through registers as shown in above Fig .3.2. The throughput of the pipeline is determined by the consideration of a fact that how often an instruction exits. As all the stages are connected, all of them should be ready to perform at the same time. The time required to move an instruction one step down to another stage among five stages sequentially is known as Processor Cycle. The slowest pipeline stage decides the length of the processor cycle. It is designer's responsibility to balance the length of processor cycle of each stage.

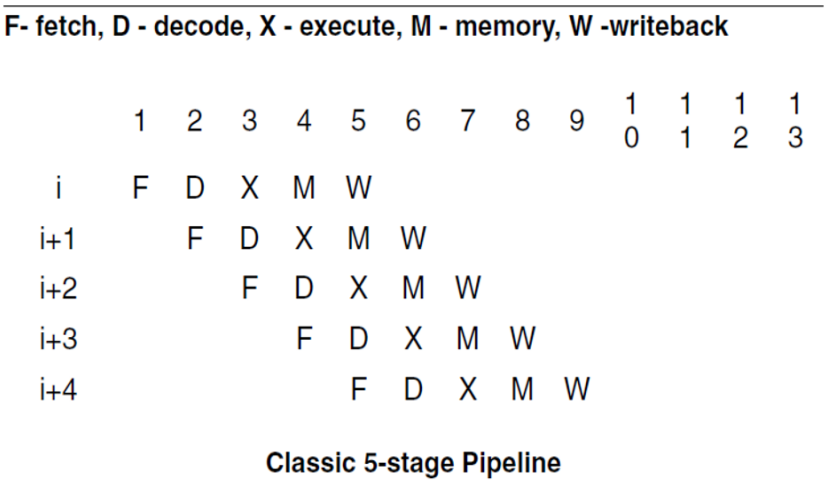


Figure 3.3: Simple Pipeline

3.2 Performance Issues in Pipelined Systems

After a certain number of stages benefits level off and later they start diminishing.A pipelined processor can stall for a variety of reasons, including delays in reading information from memory, a poor instruction set design, or dependencies between instructions.

3.2.1 Hazards in Pipelined Inorder processor

Pipelining in processors come with its own challenges known as hazards. An hazard occurs when an instruction has to stall before continuing execution due to the behaviour of other instructions in the pipeline and therefore could result in incorrect execution. The stalls introduced by hazards reduce the performance gained through pipelining.

Types of hazards

Structural Hazard: Structural hazards occur when multiple instructions try to use the same hardware unit in the same clock cycle. This often occurs when the pipelining is not done properly in the harware thereby resulting in the consecutive use of resources by instructions requiring more than one clock cycle. This type of hazards could also occur if different instructions try to access the same resource in the same clock cycle, for example two instructions attempting to read data from the same register in the same cycle. Structural hazards can be avoided by adding more hardware which could also lead to increased cost and unit latency.

Data Hazard: Data hazards occur when the data needed for the completion of an instruction is dependent on the data produced by another instruction in the pipeline and as a result, the instruction has to stall because all instructions must be completed in-order. This type of hazards could occur if a subsequent instruction tries to read from memory before a previous instruction writes to it, tries to write to memory before a previous instruction reads from it or tries to write to memory before a previous instruction writes to it.

Data hazards are usually avoided by forwarding data from an instruction to instructions needing it as soon as it is available. Another simple technique that can be used to avoid data hazards is writing instructions in the first half of a cycle and reading in the second half of the cycle. Only RAW (Read After Write) is occured in Inorder Processor.Others WAW and WAR occurs only in case of Superscalar Processors where Out of Order execution is done.

Control Hazard: Control hazards are hazards that occur due to branching instructions. This usually happens when the processor has to determine whether or not to execute a subsequent instruction based on the outcome of a previous branching instruction. This outcome of a branch cant be determined until the second stage of the instruction execution known as instruction decode stage and by then, the subsequent instruction would have already been issued. Control hazards have big impact on performance as all programs are essentially loops and also stalling a fully pipelined machine, that is a machine with a cycle per instruction of one, on encountering of any control hazards would result in three-cycle stalls for each of all the loops in the program. Control hazards are usually avoided through branch prediction mechanisms.

CHAPTER 4

Implementation of Scalar Inorder Processor

This chapter begins with the description of RISC-V Instruction Set Architecture. Different types of instruction formats are presented. Then it explains about the data path of execution of different instructions and finally explains about implementation details of 5 stages of proposed design.

Design Rules and Assumptions 1. Design is to be generated for a 32 bit processor for 32 bit Base RISC-V. That is both the data-path and the instruction word are 32 bits long. 2. Processor design is based on the Harvard architecture of separate memories for data and instructions.

4.1 Design of Instruction Set Architecture

I have implemented 32 bit Base level subset of RISC-V ISA [5]. Here I present details of those Base level Instructions formats. There are 31 general-purpose registers r1-r31, which hold integer values. Register r0 is hardwired to the constant 0. For RV32, registers are of 32 bits wide. There is one more register: the program counter PC holds the address of the current instruction. In the Base ISA, there are five basic instruction formats. These are fixed 32 bits in length.

4.1.1 Basic Instruction Formats

R-Type: R-type instructions has two source registers (rs1 and rs2) and a destination register (rd). The funct10 is an additional opcode field.

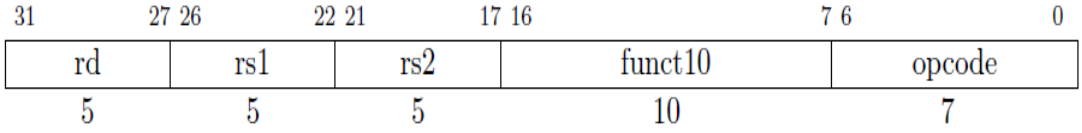


Figure 4.1: R-Type

I-Type: I-type instructions has one source register (rs1) and a destination register (rd). The second source operand is a sign-extended 12-bit immediate, encoded contiguously in bits 21-10.The funct3 field is a second opcode field.

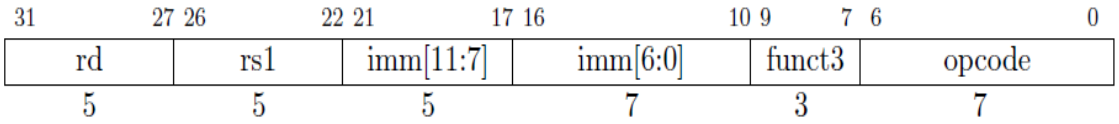


Figure 4.2: I-Type

B-Type: B-type instructions specify two source registers (rs1 and rs2) and a third source operand encoded as a sign-extended 12-bit immediate. The immediate is encoded as the concatenation of the upper 5 bits in bits 31-27, and a lower 7 bits in bits 16-10. The funct3 field is a second opcode field.

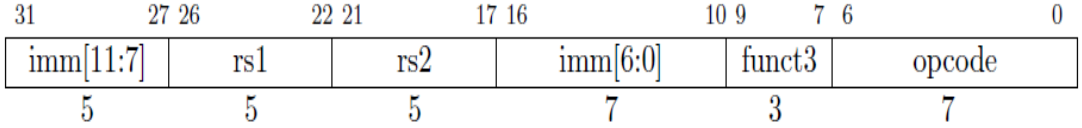


Figure 4.3: B-Type

U-Type: U-type instructions specify a destination register (rd) and a 20-bit upper immediate value that represents bits 31-12 of a 32-bit signed integer.

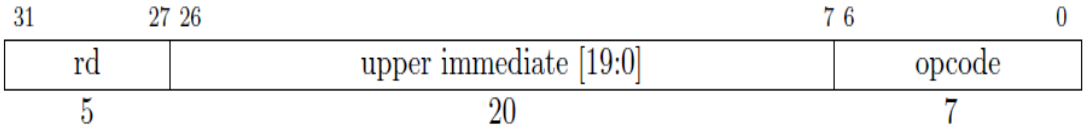


Figure 4.4: U-Type

J-Type: J-type instructions encode a 25-bit jump target address as a PC-relative offset. The 25-bit immediate value is shifted left one bit and added to the current PC to form the target address.

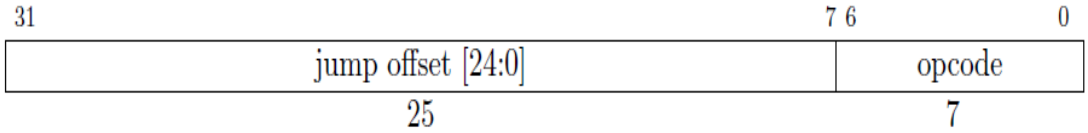


Figure 4.5: J-Type

4.1.2 Integer Computational Instructions

Integer computational instructions are either encoded as register-immediate operations using the I-type format or as register-register operations using the R-type format. The destination is register rd for both register-immediate and register-register instructions. No integer computational instructions cause arithmetic exceptions.

Register Immediate Operations: ADDI adds the sign-extended 12-bit immediate to register ADDI rd, rs1, 0 is used to implement the MV rd, rs1 assembler pseudo-instruction.

SLTI (Set Less than Immediate) places the value 1 in register rd if register rs1 is less than the sign-extended immediate when both are treated as signed numbers, else 0 is written to rd. SLTIU is similar but compares the values as unsigned numbers (i.e., the immediate is first sign-extended to 32-bits then treated as an unsigned number). Note, SLTIU rd, rs1, 1 sets rd to 1 if rs1 equals zero, otherwise sets rd to 0.

ANDI, ORI, XORI are logical operations that perform bitwise AND, OR, and XOR on register rs1 and the sign-extended 12-bit immediate and place the result in rd. Note, XORI rd, rs1, -1 performs a bitwise logical inversion (NOT) of register rs1.

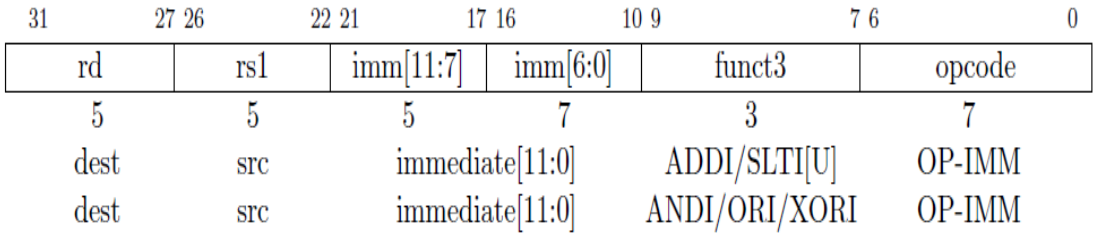
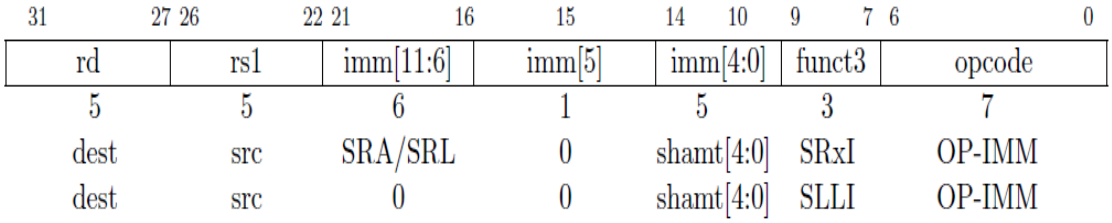


Figure 4.6: Register-Immediate type 1

Shifts by a constant are encoded as a specialization of the I-type format. The operand to be shifted is in rs1, and the shift amount is encoded in the lower 5 bits of the immediate field. The shift type is encoded in the upper bits of the immediate field. SLLI is a logical left shift (zeros are shifted into the lower bits); SRLI is a logical right shift (zeros are shifted into the upper bits); and SRAI is an arithmetic right shift (the original sign bit is copied into the vacated upper bits).

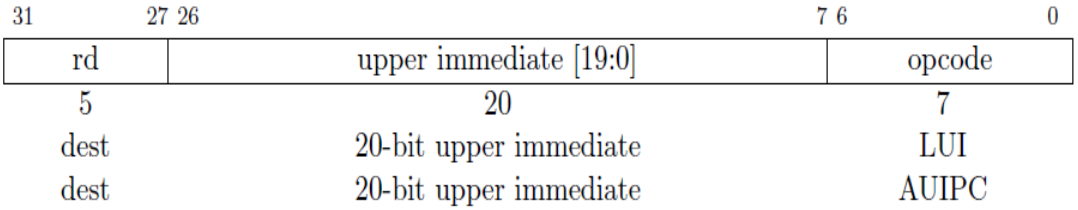


31–27	26–22	21–16	15	14–10	9–7	6–0
rd	rs1	imm[11:6]	imm[5]	imm[4:0]	funct3	opcode
5	5	6	1	5	3	7
dest	src	SRA/SRL	0	shamt[4:0]	SRxI	OP-IMM
dest	src	0	0	shamt[4:0]	SLLI	OP-IMM

Figure 4.7: Register-Immediate type 2

LUI (load upper immediate) is used to build 32-bit constants. LUI shifts the 20-bit immediate left 12 bits, filling in the vacated bits with zeros, then places the result in register rd.

AUIPC (add upper immediate to pc) is used to build pc-relative addresses. AUIPC shifts the 20-bit immediate left 12 bits, adds the pc, then places the result in register rd.

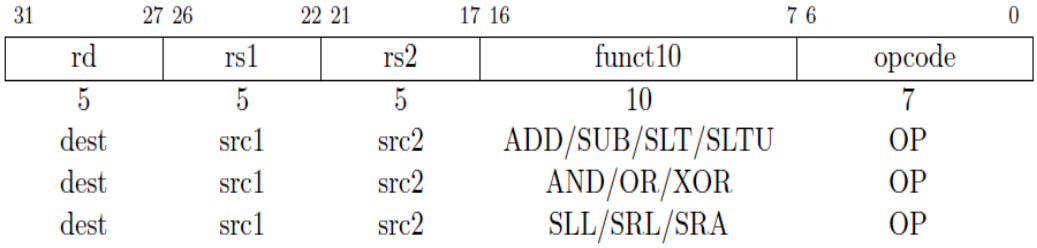


31–27	26–7	6–0
rd	upper immediate [19:0]	opcode
5	20	7
dest	20-bit upper immediate	LUI
dest	20-bit upper immediate	AUIPC

Figure 4.8: Register-Immediate type 3

Integer Register-Register Operations RV32I defines several arithmetic R-type operations. All operations read the rs1 and rs2 registers as source operands and write the result into register rd. The function field selects the type of operation.

ADD and SUB perform addition and subtraction respectively. Overows are ignored and the low 32 bits of results are written to the destination. SLT and SLTU perform signed and



31 27	26 22	21 17	16 7	6 0
rd	rs1	rs2	funct10	opcode
5	5	5	10	7
dest	src1	src2	ADD/SUB/SLT/SLTU	OP
dest	src1	src2	AND/OR/XOR	OP
dest	src1	src2	SLL/SRL/SRA	OP

Figure 4.9: Register-Register type

unsigned compares respectively, writing 1 to rd if rs1 ¡ rs2, 0 otherwise. Note, SLTU rd, r0, rs2 sets rd to 1 if rs2 is not equal to zero, otherwise sets rd to zero. AND, OR, and XOR perform bitwise logical operations. SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.

4.1.3 Control Transfer Instructions

RV32I provides two types of control transfer instructions: unconditional jumps and conditional branches. Control transfer instructions in RV32I do not have architecturally visible delay slots.

Unconditional Jumps Absolute jumps (J) and jump and link (JAL) instructions use the J-type format. The 25-bit jump target offset is sign-extended and shifted left one bit to form a byte offset, then added to the pc to form the jump target address.JAL stores the address of the instruction following the jump (pc+1) into register r1.

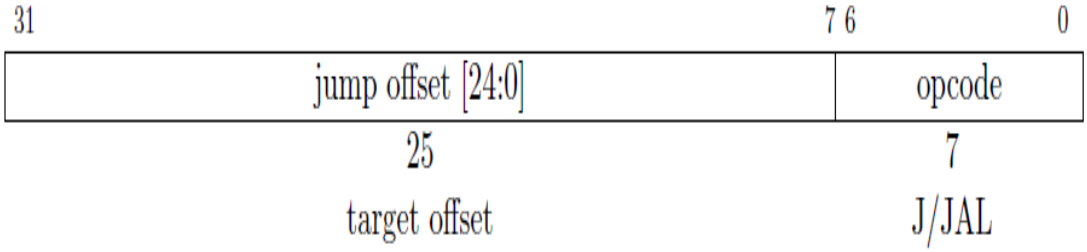


Figure 4.10: Unconditional Jump type

Conditional Jumps All branch instructions use the B-type encoding. The 12-bit immediate is sign-extended, shifted left one bit, then added to the current pc to give the target address.Branch instructions compare two registers. BEQ and BNE take the branch if registers rs1 and rs2 are equal or unequal respectively. BLT and BLTU take the branch if rs1 is less than rs2, using signed and unsigned comparison respectively. BGE and BGEU take the branch if rs1 is greater than or equal to rs2, using signed and unsigned comparison respectively. Note, BGT, BGTU, BLE, and BLEU can be synthesized by reversing the operands to BLT, BLTU, BGE, and BGEU, respectively.

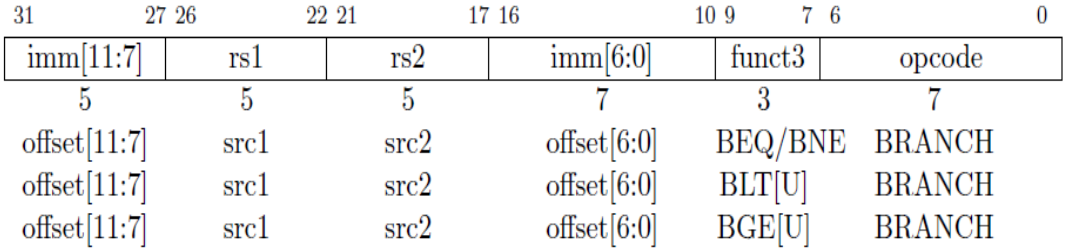


Figure 4.11: Conditional Jump type

4.1.4 Load and Store Instructions

RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers.Load and store instructions transfer a value between the registers and memory.

Loads are encoded in the I-type format, and stores are B-type. The effective byte address is obtained by adding register rs1 to the sign-extended immediate. Loads copy a value from memory to register rd. Stores copy the value in register rs2 to memory.

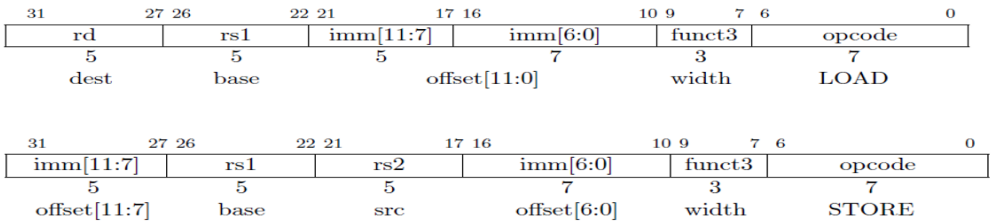


Figure 4.12: Load and Store Instructions

4.2 Data Path Design

The design of the instruction set was followed by the analysis of data-paths which are taken by a data set upon instruction execution. These are the paths in which data will flow around the processor. The paths for each instruction are combined to form the overall data-path for the processor. The most common data-paths are as follows:

R-Type Data-path The R-type data-path is also known as the arithmetic data-path. In the R-type data-path the instruction is fetched from memory and broken up into its various parts. The two read registers from the instruction are fetched from the Register File and the ALU performs the operation given to it by the instruction. The result from the ALU is then written back into the register file.

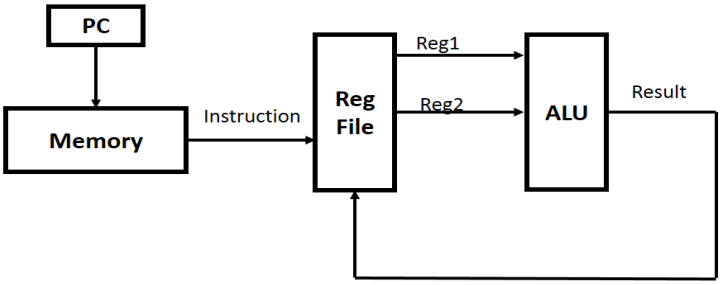


Figure 4.13: R-Type Data path

Immediate-Type Data-path Also known as the Register Immediate Data-path or the RI Data-path, it is similar to the R-type except the second read register is replaced with a value that is actually inside the instruction.

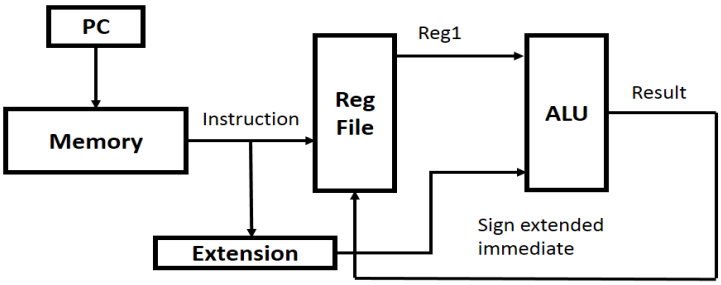


Figure 4.14: Immediate-Type Data path

Load Word Data-path The data-path for a load word is identical to the RI-type data-path with the exception that the result from the ALU is sent to fetch a value from memory instead of being written to the register file. The value that is fetched from memory is then loaded into the register file.

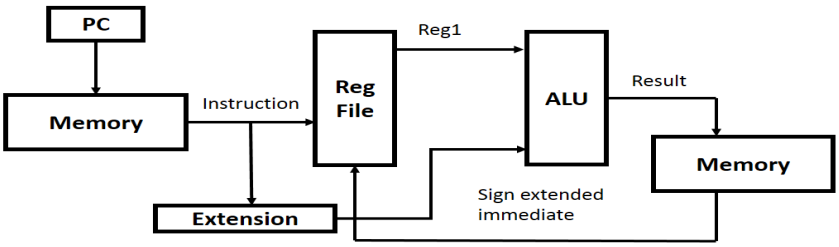


Figure 4.15: Load Word Data path

Store Word Data-path The store word data-path is similar to the load word with the exception that the write register actually specifies which register to write to memory and not the register file.

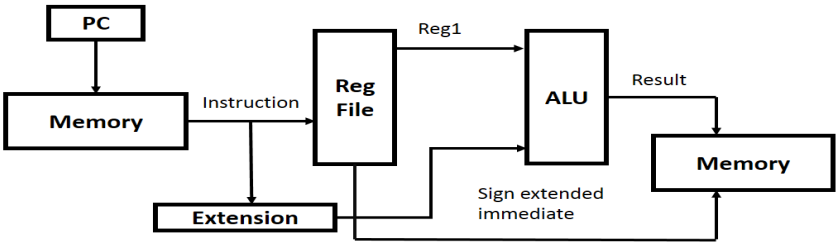


Figure 4.16: Store Word Data path

4.3 Implementation Details

Scalar pipeline: The term Scalar pipeline means one instructionn per stage per cycle.

In-order pipeline: It means all the instructions enter into execution stage in an order.

So, this section describes about different issues in each stage and how are they overcomed.

The overview of processor is seen in below figure.

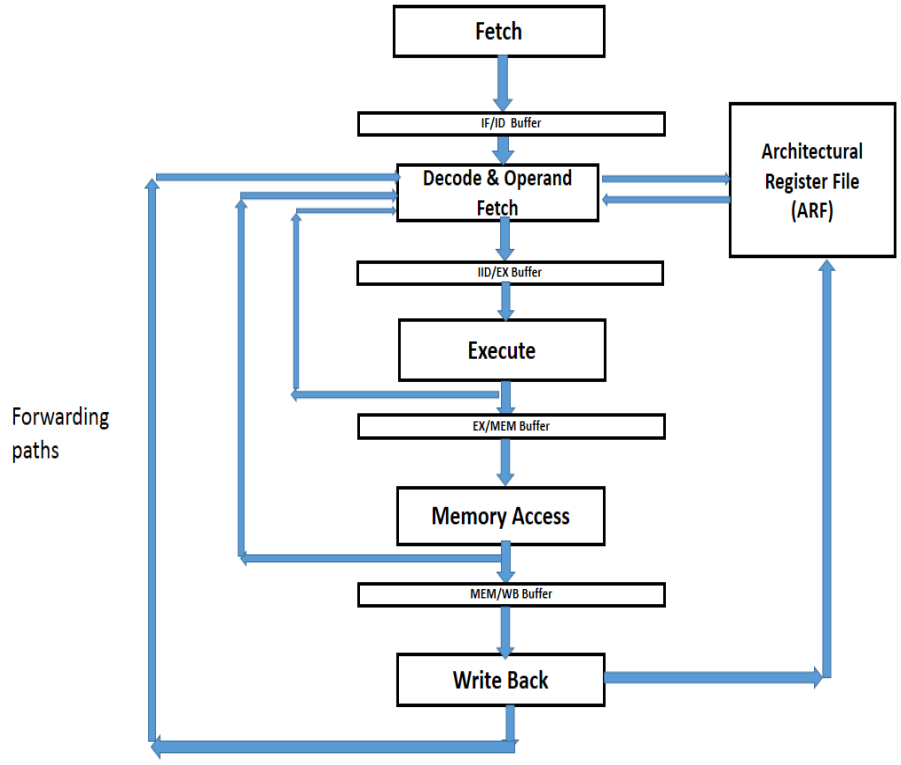


Figure 4.17: Proposed Architecture

4.3.1 Instruction Fetch (IF) Stage

PC is used to fetch the instruction from the Instruction Memory and is stored in the Instruction Register (IF/ID) at the next clock. This stage has various modules like Instruction Memory, which holds the instructions needed. PC holds the address of the current instruction, which is used as address to the Instruction Memory.

The instructions read out from the Instruction memory are stored in the Instruction Register, which is a part of IF/ID Register. Instruction Memory is built using in-built constructs available in Bluespec library.

The following Figure 4.18 is the data path for Instruction Fetch. The other part of IF/ID is PC itself. In this stage logic has to be implemented to determine next PC. So a MUX is used to select next PC which is either PC+1 or PC + offset in case of Unconditional Jump instructions like J or PC + offset in case of Conditional branch instruction whose prediction validation is done in Execution Unit.

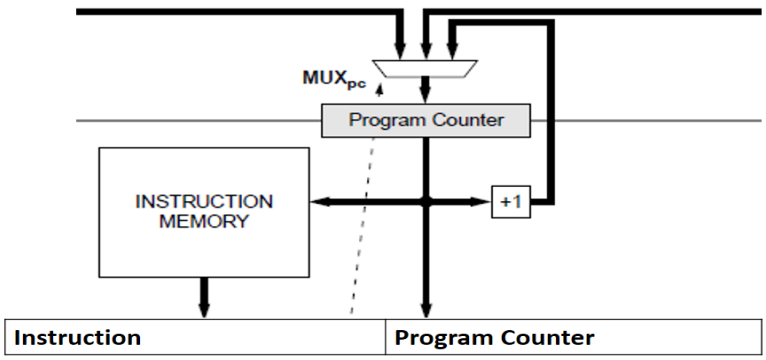


Figure 4.18: Data path for Instruction Fetch

4.3.2 Instruction Decode (ID) and Operand Fetch Stage

In this stage the instruction fetched is first decoded to decide the type of instruction it belongs to like branch, alu ,load or store. If the instruction fetched is found to be Unconditional Jump then PC + offset is calculated and forwarded to IF stage also a control signal is send to select PC + offset as next PC. Inclusion of Branch Prediction Table is also done for case of Conditional Branch instructions. I have designed a decoder which is capable of decoding upto 152 instructions of RISC-V ISA.

4.3.2.1 Operand Fetching

Here operands are read from Register File. There is a valid bit associated with each register to determine whether it is valid or not. In decode stage, the destination register of current instruction into which the result is to be written is made invalid, so that the next instructions do not read the wrong values. Here the Data Hazards are resolved. The Figure 4.21 explains the details of Operand Fetching technique.

Handling of Data Hazards: Data hazards are resolved using Operand Forwarding and Stalling techniques.Data Hazards occur when an instruction attempts to use a register whose value depends on the result of previous instructions that have not yet finished.Since our Processor Scalar In-order, We encounter only one type of hazard i.e RAW (Read After Write).

Stalling: Stalling involves halting the flow of instructions until the required result is ready to be used. It is the simplest way to resolve a data hazard. However, stalling wastes processor time by doing nothing while waiting for the result.

Operand Forwarding or Bypassing: The forwarding method is best described through the use of an example. The figure below shows two instructions in the pipeline, It can be seen that the instruction I2 needs the result of the instruction I1 in the I2's EX stage but the instruction I1 does not write the result until the I1's WB stage. However it can also be seen that the result for the I1 instruction is actually computed before the I2 instruction needs it so the result is forwarded from the EX/MEM stage back to the EX stage of the I2 instruction.

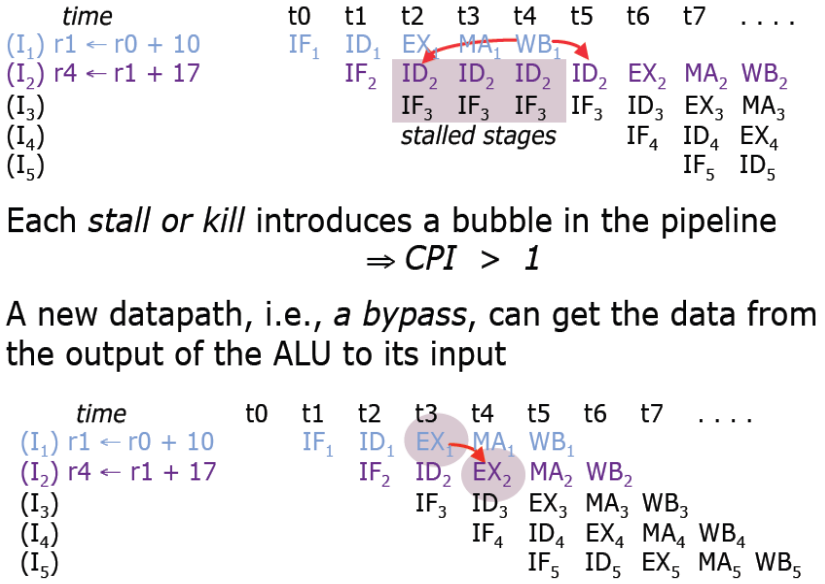


Figure 4.19: Stalling and Bypassing techniques

Similarly it is required to have a path which forwards data from Memory Access stage and Write back stage. If ADD instruction in which one of the operands is the destination register of Load instruction is followed by ADD, we can forward the data to ADD instruction from MEM stage of Load instruction because actual writing into register happens in WB stage. Also a path is required to forward data from WB stage to Decode stage while writing into Register File is occuring simultaneously.

Structural Hazards: A structural hazard occurs when the hardware is unable to handle certain combinations of instructions simultaneously. Multi cycle operations causes structural hazards. But here all instructions are implemented in single cycle, so there is no possibility of occurring structural hazard. If it occurs then we use the same method Stalling as in case of Data Hazard, until the required hardware is free.

Control Hazards: These occurs in cases of branches and exceptions, where dependency for next instruction's address arises. So, these are handled using prediction techniques.There are many branch prediction techniques. There are of two types namely Static and Dynamic branch prediction. Static branch prediction is made by the compiler whereas Dynamic prediction is made during run time based on history. Here I have implemented static branch prediction technique to avoid control hazards. Here I have considered Branch Not Taken technique and based on the condition becoming true or false in Execution unit, the necessary steps are taken. This technique helps to reduce overhead of stalls.

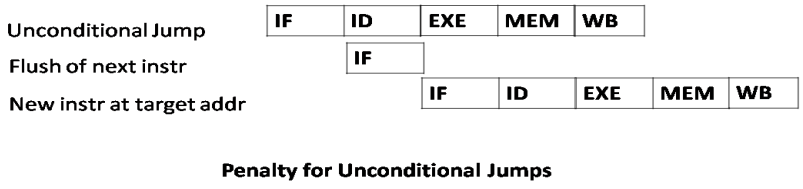


Figure 4.20: Penalty for Unconditional Jumps

So all kinds of Hazards are detected and necessary control signals are generated in this stage So finally the ID/EX register consists of two operands data, PC, Instruction and Instruction type.

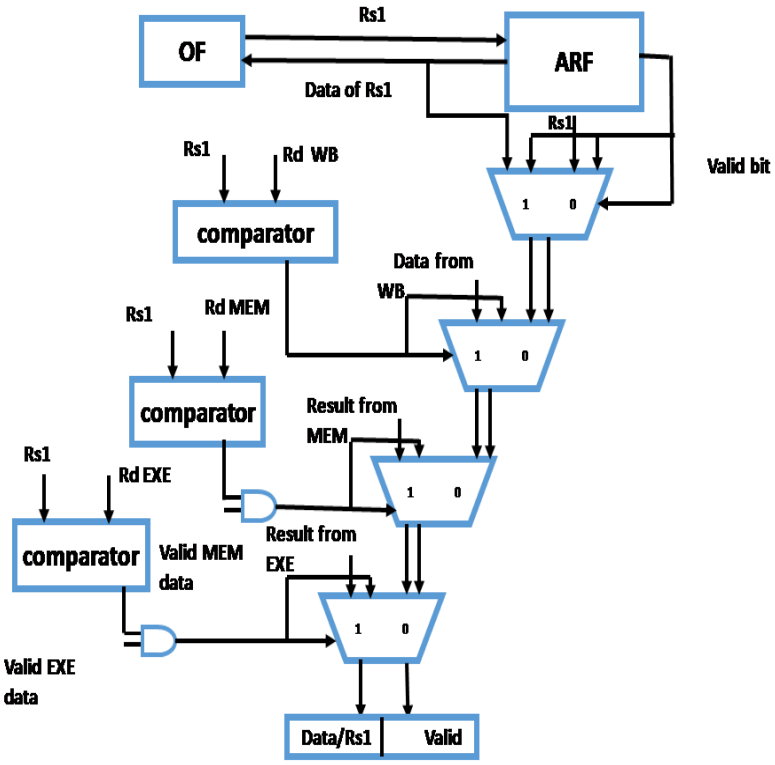


Figure 4.21: Operand Fetching

4.3.3 Execution Unit:

In this I have implemented a single cycle execution unit where it can handle all base 32 bit Integer Instructions. In case of ALU instructions the two operands are taken from ID/EX stage and based on Instruction type the required operation is performed. I have implemented all ADD, SUB, XOR, AND, Shift instructions considering all combinations of signed and unsigned operands. Also I have implemented a Carry Save Based multiplier for MUL operations.

In case of Branch instructions namely Conditional Branch instructions prediction validation is done by checking the condition.So, based on condition True or False the PC is updated.In case of prediction going wrong we have to flush the next two instructions which are already in pipeline.

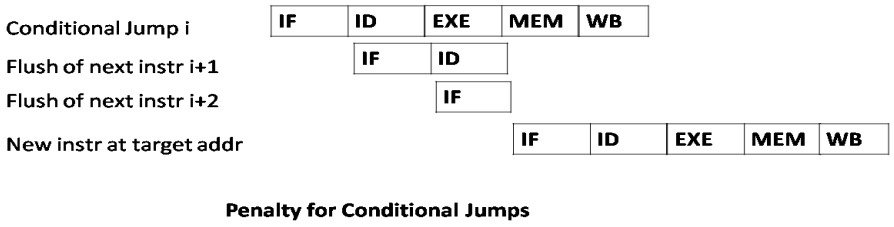


Figure 4.22: Penalty for Conditional Jumps

In case of Load/Store instructions the required effective address is calculated. Also in order to avoid data hazards the valid result is forwarded to ID stage so that the stalls are avoided.

So finally the EX/MEM register consists of ALU output, Store data in case of Store Instruction and PC and Instruction type. ALU output consists either ALU result or Effective address for Load/Store instructions.

4.3.4 Memory Accesss Stage:

In this stage Load and Store instructions access the Data Memory to read and write the data. The effective address calculated in previous stage is given to data memory and data is read in case of Load operation and the data is written into memory at the calculated effective address for Store instruction.

In case of Load here data is read from memory but it is not actually written into the destination register. Also to avoid data hazards the valid result is forwarded to ID stage so that the stalls are avoided. For ALU instructions, this stage is idle. Finally MEM/WB

register consists of destination register address and data to be written into it. Memory is implemented using 'RegFile' in built construct nothing but a block of registers.

4.3.5 Write Back Stage

In this stage the writing of the results into Registers in Architectural Register File is done. Also to reduce the stalls the result is forwarded to ID stage.

Apart from all the above modules, I also have implemented separate Branch Execution Unit for Super scalar processor implementation which will do validation of our prediction that is done during decode stage using 2-bit branch prediction technique. I also have implemented a 2-bit dynamic branch predictor as a separate module.

4.4 Verification

This section deals about verification of the processor design that is in Bluespec System Verilog. It also describes about the test cases generated to verify the design.

4.4.1 Verification Setup

The Bluespec code that is written in .bsv format is given to the Bluespec compiler in the Bluespec Development Workstation (BSW). The Bluespec Compiler compiles and generates the Verilog code in .v format. The Verilog code is simulated in the ICARUS Verilog simulator to check for the functionality of the design. The Verilog code is also synthesized in Xilinx ISE to get hardware utilization.

The functionality of the design is verified by using a test bench which provides a set of instructions given to the main module. 'display' statements were written in the code in all the necessary places to monitor the functionality of the modules in various clock cycles.

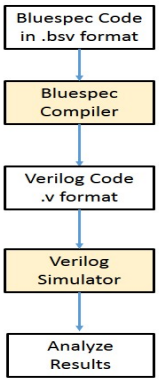


Figure 4.23: Verification steps in BSV

The verification process mainly focused on verifying the functional operations of the processor. The necessary test instructions are directly are first written into Instruction Memory. A variety of sequence of instructions are loaded so that maximum combinations are met.

A testbench is written for Execution module of processor to verify all operations.Also testbenches are written to verify following functionalities.

- Dependency check between instructions.
- Operand Fetching.
- Operand Forwarding and Hazard detection

4.4.2 Synthesis Report

The design has been synthesized using Xilinx ISE for ***Virtex 6 XC6VLX240T-FF1156***. All default settings were used. The design strategy was set to *"optimization for speed"*. The slice utilization and timing summary are provided below.

Device Utilization Summary :

Selected Device: 6vlx240tff1156-1

Slice Logic Utilization:

Number of Slice Registers : 4264 out of 301440

Number of Slice LUTs : 12578 out of 150720

Number used as Logic : 12578 out of 150720

Number of LUT Flip Flop pairs used: 3453

Timing Summary :

Minimum period: 2.743ns

Maximum Frequency: 364.45MHz

Minimum input arrival time before clock: 0.363ns

Maximum output required time after clock: 0.567ns

CHAPTER 5

CONCLUSION AND FUTURE WORK

In this practice, I have successfully accomplished building a Scalar Inorder Processor with pipeline functionalities. Data hazard, Structural hazard and control hazards are resolved successfully. This design shows the implementation of RISC-V Base ISA CPU capable of handling various R-type, J-type and I-type of instruction and each of these categories has a different format. Designing Forwarding unit and hazard detection unit to overcome the data dependencies was critical task and it was implemented successfully. This project shows the wide variety of logics to consider during the design.

Future Work:

The design does not cover a number of issues involved in real-world implementations, including data caches, instruction caches, data- and instruction-cache misses, support for precise interrupts, or branch prediction more sophisticated than predict-not-taken. In the existing design, the following are needed to be modified or added to enhance the performance of the processor:

- Development of a 32-bit RISC processor with all instructions included like Floating point instructions, Atomic Memory Operations.
- Development of separate hardware for memory and implementing memory management sub-routines.
- Development of a full cache memory

APPENDIX A

RISC-V ISA

A.1 INTRODUCTION

RISC-V is a new instruction set architecture (ISA) designed to support computer architecture research and education. Few points about RISC-V are:

- A completely open ISA that is freely available to academia and industry.
- A realistic ISA that is suitable for direct hardware implementation and which captures important details of commercial general-purpose ISA designs.
- Optional variable-length instructions to both expand available instruction encoding space and to support an optional dense instruction encoding for improved performance, static code size, and energy efficiency.
- Both 32-bit and 64-bit address space variants for applications, operating system kernels, and hardware implementations.
- Support for the revised 2008 IEEE-754 floating-point standard.
- Standard simple ISA subsets for educational purposes or for embedded systems, and to reduce the complexity of bringing up new implementations.
- An ISA supporting extensive user-level ISA extensions and specialized variants.

All RISC-V user-level ISAs are built around a base integer instruction set that must be supported by any valid RISC-V implementation. Each base integer instruction set is characterized by the width of the integer registers and the corresponding size of the user address space. There are two base integer variants, RV32I and RV64I, which provide 32-bit or 64-bit user-level address spaces respectively.

A.2 Instruction Length Encoding

The base RISC-V ISA has fixed-length 32-bit instructions that must be naturally aligned on 32-bit boundaries. However, the standard RISC-V encoding scheme is designed to support ISA extensions with variable-length instructions, where each instruction can be any number of 16-bit instruction parcels in length and parcels are naturally aligned on 16-bit boundaries.

The below figure illustrates the standard RISC-V instruction-length encoding convention. All the 32-bit instructions in the base ISA have their lowest two bits set to 11. The optional compressed 16-bit instruction-set extensions have their lowest two bits equal to 00, 01, or 10. Standard instruction-set extensions encoded with more than 32 bits have additional low-order bits set to 1, with the conventions for 48-bit and 64-bit lengths shown in Figure 1.1. Instruction lengths between 80 bits and 304 bits are encoded using a 4-bit field giving the number of 16-bit words in addition to the first 5x16-bit words. Encodings with 11 or more low-order opcode bits set to 1 are reserved for future longer instruction encodings.

Figure A.1: Instruction length encoding

REFERENCES

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

[2] J. P. Shen and M. H. Lipasti, *Modern Processor Design - Fundamentals of Superscalar processors*. TATA McGraw-Hill Publishing Company Private Limited, 2005.

[3] R. S. Nikhil and K. Czeck, *BSV by Example*. Bluespec, Inc, 2010.

[4] Bluespec, Inc, *Bluespec System Verilog Reference Guide*, revision: 17 ed., 2012.

[5] User Level RISC-V ISA, *University of California, Berkeley*, revision: 2 ed., 2014.

[6] *http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-823-computer-system-architecture-fall-2005/lecture-notes/*.

[7] *http://inst.eecs.berkeley.edu/ cs152/sp14/*.