VERIFICATION OF RISC-V DESIGN WITH UNIVERSAL VERIFICATION METHODOLOGY (UVM)

LIEW YOU HONG

A project report submitted in partial fulfilment of the requirements for the award of the degree of Bachelor of Engineering (Hons) Electronic Engineering

Faculty of Engineering and Green Technology
Universiti Tunku Abdul Rahman

May 2022

DECLARATION

I hereby declare that this project report is based on my original work except for citations and quotations which have been duly acknowledged. I also declare that it has not been previously and concurrently submitted for any other degree or award at UTAR or other institutions.

Signature : 

Name : Liew You Hong

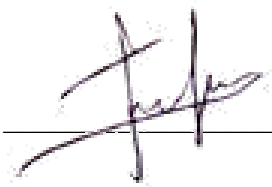
ID No. : 17AGB04120

Date : 2022-05-09

APPROVAL FOR SUBMISSION

I certify that this project report entitled **"VERIFICATION OF RISC-V DESIGN WITH UNIVERSAL VERIFICATION METHODOLOGY"** was prepared by **LIEW YOU HONG** has met the required standard for submission in partial fulfilment of the requirements for the award of Bachelor of Engineering (Hons) Electronic Engineering at Universiti Tunku Abdul Rahman.

Approved by,

Signature : 

Supervisor : Ir. Dr. Loh Siu Hong

Date : 2022/05/09

The copyright of this report belongs to the author under the terms of the copyright Act 1987 as qualified by Intellectual Property Policy of Universiti Tunku Abdul Rahman. Due acknowledgement shall always be made of the use of any material contained in, or derived from, this report.

© 2022, Liew You Hong. All right reserved.

ACKNOWLEDGEMENTS

I would like to thank everyone who had contributed to the successful completion of this project. I would like to express my gratitude to my research supervisor, Ir. Dr. Loh Siu Hong for his invaluable advice, guidance and his enormous patience throughout the development of the research.

In addition, I would also like to express my gratitude to my loving parents and friends who have supported me and given me encouragement. I would also like to express my appreciation for the various challenges posed throughout the project that pushed me to greater heights. Lastly, I would like to express my gratitude to this project that served as a bridge, connecting me to the reality that is our society and industry, helping me realize the natures of human beings and my shortcomings.

Thank you.

Liew You Hong

VERIFICATION OF RISC-V DESIGN WITH UNIVERSAL VERIFICATION METHODLOGY (UVM)

ABSTRACT

Throughout the design life cycle of a processor, verification plays a crucial part in affirming the functionalities of the features implemented based on the computer architecture used. Functional verification increases the level of confidence in conformance of the processor design to its specification. In the case of a processor with advanced microarchitectural features implemented, a simulation-based approach is taken for its functional verification. More specifically, Universal Verification Methodology (UVM) is utilized for the verification methodology of the RISC-V processor implementation in this report. UVM provides a set of guidelines for the verification testbenches to be generated. With a well-defined testbench structure, UVM allows for a standardized approach towards verification works and verifications of systems to be performed consistently and uniformly, greatly improving verification quality and reusability of testbenches. For the verification approach, constrained-random verification and direct verification approaches will be taken to verify the functionality of the RISC-V processor. In the verification methodology, *results validation* has been utilized whereby the output data of the simulation model is compared with comparable output data from an existing system. For verification purpose, a reference model is developed and will be utilized for the results validation methodology mentioned. On verification simulations, discrepancies between the output data from the simulation models and the reference model are identified as design bugs in the system and debugs will be performed to fix the design bugs in the system. Through numerous test runs on the RISC-V processor implementation, the bugs on the RTL design of the processor designed are reduced to a minimum and the processor can function as specified by the computer architecture.

TABLE OF CONTENTS

DECLARATION ii
APPROVAL FOR SUBMISSION iii
ACKNOWLEDGEMENTS v
ABSTRACT vi
TABLE OF CONTENTS vii
LIST OF TABLES x
LIST OF FIGURES xi
LIST OF SYMBOLS / ABBREVIATIONS xix
LIST OF APPENDICES xx

CHAPTER

1 INTRODUCTION 1
1.1 Background 1
1.1.1 IC Design Flow 2
1.1.2 Verification and Validation in IC Design Flow 4
1.2 Problem Statements 5
1.3 Aims and Objectives 6
1.4 Report Overview 6

2 LITERATURE REVIEW 7
2.1 RISC-V Computer Architecture 7
2.1.1 RISC-V Base Instruction Set Architecture 8
2.1.2 RISC-V Computer Organization 12
2.1.3 Datapath Flow 22

2.1.4 Processor Pipelining 29
2.1.5 Summarized Review for RISC-V Computer Architecture 45
2.2 Functional Verification 46
2.2.1 SystemVerilog as Functional Verification Language 46
2.2.2 Functional Verification Requirements 47
2.2.3 Functional Verification Technologies 48
2.2.4 Functional Verification Approaches 49
2.2.5 Summarized Review for Functional Verification 50
2.3 Universal Verification Methodology 51
2.3.1 UVM Testbench Architecture 52
2.3.2 UVM Component Class 53
2.3.3 UVM Transaction Base Class 56

3 METHODOLOGY 57
3.1 Verification Flow 57
3.1.1 Design Specification 59
3.1.2 Testbench Architecture Planning 59
3.1.3 Functional Verification Environment 60
3.1.4 Reference Model Development 60
3.1.5 Simulation and Verification 61
3.1.6 Verification Analysis 61
3.1.7 Verification Closure 62
3.2 Project Timeline 62
3.3 Verification Simulation Flow 63
3.3.1 Verification Specification 64
3.3.2 Instruction Code Generation 65
3.3.3 Directed Test Assembly Language Translation 70
3.3.4 Instruction Memory Setup 73
3.3.5 Step Simulation 74
3.3.6 Scoreboard Checking 76
3.3.7 Coverage Collection 80

3.3.8 Mismatch Documentation 82
3.3.9 Macro Check 85

4 RESULTS AND DISCUSSIONS 86
4.1 Self-Checking Bug Detection 86
4.2 Design Bug Detection 89
4.3 Directed Verification 99
4.4 Constrained-Random Verification 110

5 CONCLUSION AND RECOMMENDATIONS 116
5.1 Conclusion 116
5.2 Recommendations 117

REFERENCES 119

APPENDICES 122

LIST OF TABLES

TABLE	TITLE	PAGE
2.1	RISC-V Base Instructions	8
2.2	RISC-V Base Instruction Encoding Formats	9
2.3	RISC-V Instruction Field Specifications of Different Instruction Types	10
2.4	Alternate Names and Functionality of Base RISC-V General Purpose Registers	11
2.5	Multiplexing of Instruction Address to be updated	13
2.6	Control Signal Values based on Instruction Type	16
2.7	Control Signals from the Control Unit	17
2.8	ALU Control Signal based on Instruction	18
2.9	Immediate Value Generated corresponding to Instruction Type	21
2.10	Forwarding Control Signal and their Description	39
3.1	Test Arguments for Verification Specification	64
4.1	Directed Verification Test Program	99
4.2	Constrained-Random Verification Test Case Specifications	110

LIST OF FIGURES

FIGURE	TITLE	PAGE
1.1	Integrated Circuit Design Flow	2
2.1	Program Counter Unit	13
2.2	Instruction Memory Unit	14
2.3	Register File Unit	15
2.4	Control Unit	16
2.5	ALU Control Unit	18
2.6	Arithemtic Logic Unit	19
2.7	Data Memory Unit	20
2.8	Immediate Generation Unit	21
2.9	Simple Datapath of the Base RISC-V Processor	22
2.10	Separation of Single-Cycle Datapath for Pipeline Implementation	30
2.11	Multiple Instructions executed with Pipeline Implementation	30
2.12	Pipelined Datapath	31

2.13	Pipelined Datapath with Control Elements integrated	32
2.14	Pipeline flows at full speed when Branch Prediction is correct	37
2.15	Pipeline flushes only when Branch Prediction is incorrect	37
2.16	Graphical Representation of Forwarding	38
2.17	Implementation of Data Forwarding on the Pipelined Datapath	39
2.18	Graphicaal Representation of Stalling and Forwarding	40
2.19	Implementation of Data Forwarding and Data Stalling on the Pipelined Datapath	41
2.20	Pipelined Datapath with Branch Comaprison forwarded to Instruction Decode Stage	44
2.21	UVM Testbench Architecture	52
3.1	Design Verification Flow	57
3.2	Revised Vereification Testbench Architecture	60
3.3	Gantt Chart for Phase 1 of Final Year Project	62
3.4	Gantt Chart for Phase 2 of Final Year Project	62
3.5	Verification Simulation Flow	63
3.6	Randomization of specified Instruction Type	66
3.7	Declaration of Randomized Instruction Fields	66

3.8	Assigning valid funct3 field using randcase	67
3.9	Assigning valid funct7 field using randcase	67
3.10	Concatenation of Instruction Fields Generated into Instruction Codes	68
3.11	Storing of Instruction Code and Address onto ASM.txt and PROM.txt	68
3.12	Instruction Address and Instruction Code on ASM.txt	69
3.13	Segmented Instruction Code on PROM.txt	69
3.14	Assembly Language Instruction Codes in TEST.txt	70
3.15	Translation of Assembly Code Instruction Operation	70
3.16	Translation of Assembly Code First Operand	71
3.17	Translation of Assembly Code Second Operand	71
3.18	Translation of Assembly Code Third Operand	72
3.19	Concatenation of Translated Information to form valid Instruciton Codes	72
3.20	Loading of Instruction Code from PROM.txt initiated by UVM Driver component	73
3.21	Instruction Codes stored on Instruction Memory	73
3.22	Reference Model to Interface Data Transaction	74
3.23	Monitor Data Transaction Relaying	75
3.24	Register Read Address Self-Checking	76

3.25	Instruction Execution Output Self-Checking	76
3.26	Register Write Address and Data Self-Checking	77
3.27	Stalling on Load-use Cases Self-Checking	77
3.28	Flushing on Branch or Jump Instructions Self-Checking	78
3.29	Decoding of Instruction being executed	78
3.30	Display of Decoded Instruction Code on ModelSim Transcript	79
3.31	Comparison of Data between Reference Model and Design Under Test	79
3.32	Functional Coverage Cover Points	80
3.33	Functional Coverage Statistics for a Test Case Simulation	81
3.34	Mismatch Message generated on ModelSim Transcript	82
3.35	Pipeline Filling and Null Information on Pipeline Stages	82
3.36	Removing invalid data from Pipeline Register Data Record	83
3.37	Formatting and Logging of Information	83
3.38	Test Log Documentation on Mismatch Instruction	84
3.39	Macro Status Data Record	85
3.40	Macro Status at end of Test Log Document	85

4.1	Modification to Reference Unit ALU Source Code	87
4.2	Reference Model Instruction Functionality Bug Detection ModelSim Transcript Message	87
4.3	Modification to Reference Unit Hazard Detection Unit Flush Assertion Source Code	87
4.4	Reference Model Flush Nonassertion Bug Detection ModelSim Transcript Message	87
4.5	Modification to Reference Unit Hazard Detection Unit Stall Assertion Source Code	88
4.6	Reference Model Stall Nonassertion Bug Detection ModelSim Transcript Message	88
4.7	Modification to Design Under Test Program Counter Source Code	89
4.8	Program Counter Mismatch Detection ModelSim Transcript Message	89
4.9	Modification to Design Under Test Program Counter Target Address Branching Source Code	90
4.10	Program Counter Branch Target Address Mismatch Detection ModelSim Transcript Message	90
4.11	Modification to Design Under Test Program Counter Jump Register Target Address Source Code	90
4.12	Program Counter Jump Register Target Address Mismatch Detection ModelSim Transcript Message	91

4.13	Modification to Design Under Test Instruction Memory Source Code	91
4.14	Instruction Code Mismatch Detection ModelSim Transcript Message	91
4.15	Modification to Design Under Test Register File Source Code	92
4.16	System Register Read Register Address Mismatch Detection ModelSim Transcript Message	92
4.17	Modification to Design Under Test Immediate Generation Unit Source Code	92
4.18	Immediate Value Mismatch Detection ModelSim Transcript Message	93
4.19	Modification to Design Under Test Control Unit Load Instruction Control Signal Source Code	93
4.20	Control Signal Mismatch Detection ModelSim Transcript Message	93
4.21	Modification to Design Under Test ALU Source Code	94
4.22	ALU Output Mismatch Detection ModelSim Transcript Message	94
4.23	Modification to Design Under Test Forwarding Unit Source Code	94
4.24	Forwarded Operand Mismatch ModelSim Transcript Message	95
4.25	Modification to Design Under Test Control Unit Flush Control Source Code	95

4.26	Flush Nonassertion Mismatch ModelSim Transcript Message	95
4.27	Modification to Design Under Test Control Unit Stall Control Source Code	96
4.28	Stall Nonassertion Mismatch ModelSim Transcript Message	96
4.29	Modification to Design Under Test Data Memory Load Data Source Code	97
4.30	Load Data Mismatch ModelSim Transcript Message	97
4.31	Modification to Design Under Test Data Memory Store Data Source Code	98
4.32	Store Data Mismatch ModelSim Transcript Message	98
4.33	General-Purpose Register Signature Values	102
4.34	Memory Signature Values	102
4.35	Directed Verification Waveform Simulation Results Part 1	103
4.36	Directed Verification Waveform Simulation Results Part 2	104
4.37	Directed Verification Waveform Simulation Results Part 3	105
4.38	Directed Verification Waveform Simulation Results Part 4	106

4.39	Directed Verification Waveform Simulation Results Part 5	107
4.40	Directed Verification Waveform Simulation Results Part 6	108
4.41	Directed Verification Waveform Simulation Results Part 7	109
4.42	Simulation Completion Message on ModelSim Transcript	111
4.43	Constrained-Random Verification Functional Coverage Report	112
4.44	Coverage Report for RISC-V Constrained-Random Verification	113
4.45	Focused Expression Condition Coverage Miss Analysis	115

LIST OF SYMBOLS / ABBREVIATIONS

ABI	application binary interface
ALU	arithmetic logic unit
EX	Execution stage
IC	integrated circuit
ID	Instruction decode
IF	Instruction fetch
IP	intellectual property
ISA	instruction set architecture
MEM	Data memory access
PROM	Program Read-Only Memory
RISC	reduced instruction set computer
RTL	register transfer level
SVA	SystemVerilog assertion
UVM	universal verification methodology
VIP	verification intellectual property
WB	Write back

LIST OF APPENDICES

APPENDIX	TITLE	PAGE
A	Source Code of Reference Model – Top Module	122
B	Source Code of Reference Model – Program Counter	126
C	Source Code of Reference Model – Instruction Memory	127
D	Source Code of Reference Model – IF/ID Pipeline Register	128
E	Source Code of Reference Model – Register File	129
F	Source Code of Reference Model – Control Unit	130
G	Source Code of Reference Model – Immediate Generate Unit	131
H	Source Code of Reference Model – ID/EX Pipeline Register	132
I	Source Code of Reference Model – ALU Control Unit	133
J	Source Code of Reference Model – ALU	134
K	Source Code of Reference Model – Immediate Address Unit	135
L	Source Code of Reference Model – EX/MEM Pipeline Register	136
M	Source Code of Reference Model – Data Memory Unit	137
N	Source Code of Reference Model – MEM/WB Pipeline Register	138

O	Source Code of Reference Model – Forwarding Unit	139
P	Source Code of Reference Model – Hazard Detection Unit	140
Q	Source Code of Parameter List	141
P	Source Code of Verification Environment – UVM Testbench	144
S	Source Code of Verification Environment – UVM Test	145
T	Source Code of Verification Environment – Interface	147
U	Source Code of Verification Environment – UVM Environment	149
V	Source Code of Verification Environment – UVM Agent	150
W	Source Code of Verification Environment – UVM Driver	151
X	Source Code of Verification Environment – UVM Sequencer	152
Y	Source Code of Verification Environment – UVM Sequence Item	155
Z	Source Code of Verification Environment – UVM Monitor	161
AA	Source Code of Verification Environment – UVM Coverage	163
BB	Source Code of Verification Environment – UVM Scoreboard	165
CC	Instruction Set Manual	177

Chapter 1

INTRODUCTION

1.1 Background

Integrated circuits (ICs) are microchips with miniaturized electronic components such as transistors, resistors, and diodes fabricated into a single unit (Saint, 2020). These microchips can perform simple functions such as amplifying voltage to complex functions such as operating as a microprocessor for a complex electrical system. In today's life, nearly all electronic device uses ICs for its high reliability and efficiency along with its small size. The compact design of an IC allowed our modern electronic gadgets to be much smaller and capable of performing in one-millionth of a second.

In this era of rapid advancement whereby ICs continue to shrink in size and at the same time improving in processing power and speed, Moore's law, a prediction named after the cofounder of Intel, Gordon E. Moore is now truer than ever. According to Moore's Law, the number of transistors be fitted onto a microchip increases by twice of its amount every two years whereas the cost of computers is cut by half. With this rapid rate of growth in a microchip's speed and capability, the complexity and difficulty of IC design becomes even more prevalent than before. Back in 2013, the executive vice president and general manager of Intel's Technology Manufacturing Group, William Holt states that as the technology becomes smaller and smaller, the effort taken to design them becomes increasingly difficult and more effort needs to be taken to optimize the technology (Shah, 2013). The effort required may include introduction of new tools and innovations to compensate for this uprising challenge.

1.1.1 IC Design Flow

IC design is a very complex process that involves multiple stages. The following flowchart shows the typical design flow for an integrated circuit:

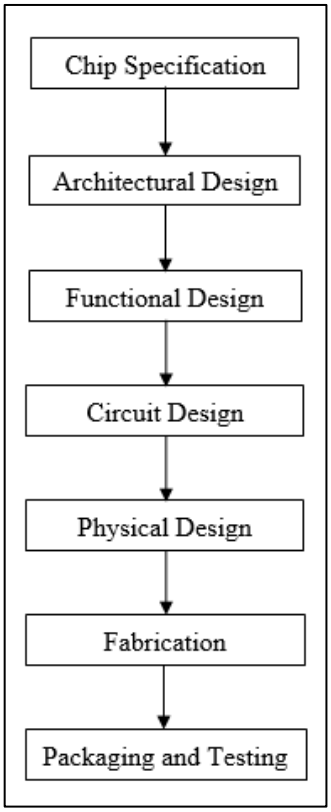


Figure 1.1: Integrated Circuit Design Flow.

In a typical design flow for an integrated circuit, the design flow begins with **chip specification**, whereby the features, microarchitecture, functionalities, and specifications of the chip are defined. These system specifications are often provided by the customers and are known as the high-level representation of the system. These specifications help the design team understand the specific requirements for the chip design (Vij, 2013).

The **architectural design** further defines the IC's required functionality and partitions them into various functional blocks. The relationship between each functional block for hardware allocation and scheduling is defined. Interface and

signals between each functional block are also defined, and a time budget is assigned to each functional block (University of Texas at Dallas, 2011).

Functional design codes the functional blocks specified in the architectural specification into register transfer level (RTL) descriptions, including the specification of the interconnections between each block and the exact behaviour of the respective functional blocks. The design team works alongside the verification team and performs behavioural simulations to verify the functional and logical behaviour of the circuit. Through the verification performed, various test vectors are generated and utilized to verify the RTL's functional behavior (Chauhan, 2020).

In **circuit design**, the high-level functional descriptions of circuit elements are further defined and decomposed into low-level circuit elements through the process known as logic synthesis. RTL code elements are converted into pre-existing building blocks such as memory units and multiplexing units with the help of synthesis tools. Upon successful logic synthesis, a gate-level netlist that contains information on the gates and the connections between each gate is produced (Synopsys, n.d). The gate-level netlist can also be known as the gate-level representation of the architectural specification of the system, providing insight into the physical implementation of the system.

Physical design converts the gate-level netlist into a manufacturable physical layout through several processes of optimization, which include floorplanning, partitioning, placement, clock tree synthesis, and routing. Floorplanning places relevant structures at particular locations with consideration of various constraints, requirements, and restrictions specified (Semiconductor Engineering, n.d.). Through effective and efficient partitioning, the complex design is divided into small blocks through a divide and conquer strategy, resulting in a system with better performance as well as lowering the production cost (Chen and Cheng, 2000). Placement determines the specific locations of the circuit modules in the netlist, optimizing the performance as well as timing delays introduced by interconnecting wires (Lavagno, Scheffer and Martin, eds., 2018). Clock tree synthesis involves the insertion of buffers to ensure an even distribution of clock signals to the sequential elements in the design to minimize

the clock skew and latency and ensure the proper timing closure is attained (Monteiro and Van Leuken, eds., 2010.). Lastly, based on logical connections between each cell, routing is performed on the design to create physical connections through metal interconnects and through the use of various routing algorithms, ensuring the best timing performance and adhering to the design rule.

Upon completion of the physical design, the physical layout of the system is obtained and physical verification is performed to validate the design functional behaviour. When the design layout is verified, the chip is then ready for **fabrication**. The layout data is converted into layers of masks which are then through the processes of deposition, diffusion, and removal, eventually transforming a silicon wafer into a prototype and tested. When the prototype passes the verifications performed, the design flow enters the last stage, where packaging and testing are performed. Wafers are mass-fabricated and converted into individual chips, packaged, and tested before delivering the chips to the customers (Vij, 2013).

1.1.2 Verification and Validation in IC Design Flow

In integrated circuit design, verification is crucial to a large-scale integrated circuit design life cycle. Verification aims to perform design functional correctness checking, detecting and debugging functional bugs in the system, eliminating human errors introduced in the design through various functional simulation tests (Ackland and Weste, 1981). Pre-silicon verification performs a functional check and identifies bugs before tape-out. In contrast, post-silicon validation captures bugs missed by pre-silicon verification through functional validation of the silicon manufactured system (Adir *et al*., 2011). In complex designs, a significant challenge is posed to design validation. The most challenging validation problem is the affirmation of the correctness of the ever-increasing amount of microarchitectural features implemented in the RTL description (Shen and Abraham, 1999). In the functional verification of a design, coverage is responsible for measuring the verification progress, assisting design engineers in identifying and understanding the progress towards design completion

(Pizialim, 2006). For a general-purpose processor design, coverage of the functional verification performed should include all functionalities implemented through multiple stages of simulation, verification, and evaluation before tape-out (Gupta and Harakchand, 2014). As processor design and verification progress through the design flow, the cost of identifying and fixing bugs increases significantly, thereby making it advisable for earlier detection and fixing of the design bugs (Gupta and Harakchand, 2014).

1.2 Problem Statements

In designing the RTL code for a processor, human errors are often introduced to the system. Functional verifications are crucial in identifying and eliminating these design bugs in the system and ensuring the system conforms to the design specifications specified in the computer architecture utilized. However, due to the complexity of a processor with millions of test cases to be considered, functional verifications with complete coverage of the design functionality are difficult to be executed and often spans for a long duration throughout the design flow due to the necessity of designing the testbench and test environment from scratch. To ease the process of functional verification, a reusable approach needs to be taken in the functional verification process.

For a standardized and reusable approach towards verification methodology, the guidelines and the complete testbench structure provided by Universal Verification Methodology (UVM) are to be utilized for the functional verification. By integrating UVM alongside functional verification of a RISC-V processor, a UVM testbench capable of performing test set generation, test driving, test monitoring, and test reporting can be constructed. Test set generation refers to the generation of random sets of instruction defined in the base set of the RISC instruction set. Test driving refers to the proper driving of the random sets of instructions generated to the design under test. Test monitoring refers to monitoring the output values from the design under test for validation purposes. Test reporting is to report the success or failure of a test run

and provide sufficient information for debug procedures. In UVM, the standardized approach supports reusability by allowing UVM-standardized Intellectual Properties to be obtained from other sources and used in the user's environment. By designing components of functional verification in modular components such as sequence, the functional verification components, otherwise known as Verification Intellectual Property (VIP), can be reused for the verification on various levels and even across different projects.

1.3 Aims and Objectives

The objectives of the thesis are shown as following:

i) To utilize Universal Verification Methodology (UVM) for the functional verification of a system.

ii) To perform thorough verification of a RISC-V processor with pipeline implementation.

1.4 Report Overview

The following chapter will discuss the overall literature review regarding RISC-V computer architecture, SystemVerilog as functional verification language, and the Universal Verification Methodology. In Chapter 3, the methodology of this project which includes the process of development of the UVM functional verification environment and the functional verification of a RISC-V processor with pipeline implementation will be explained. Chapter 4 will showcase the various results obtained from the project and explain the results obtained. Lastly, Chapter 5 will conclude the project and provide insight into future improvements.

Chapter 2

LITERATURE REVIEW

2.1 RISC-V Computer Architecture

A computer architecture, also known as instruction set architecture (ISA), is the attributes of a computer system visible to a programmer and the system's characteristics that directly affect the logical execution of a program. The ISA of a computer system specifies the instruction format, instruction opcode, registers, instruction operations, data memory, and the effect of the instructions executed on the registers and memory alongside the control mechanism for the instruction execution.

Reduced Instruction Set Computer (RISC) is an instruction set architecture renowned for its performance and capability. It is capable of handling a wide range of applications, ranging from powering micro-power embedded devices up to high-performance cloud server microprocessors. Contrasting against most instruction set architecture, RISC is an open-source ISA, free to be used by anyone, thus allowing its use for the project. RISC provides a complete set of base ISA with minimal capabilities such as arithmetic, loads and stores, branch, whereby additional extensions are available for more advanced capabilities (Ledin, 2020). The minimal yet complete set of capabilities set a proper scope for the project, thus making the RISC-V base ISA a perfect choice.

2.1.1 RISC-V Base Instruction Set Architecture

The base RISC-V ISA utilizes a 32-bit system and features 32-bits instructions that perform **arithmetic**, **data transfer**, **logical**, **data-shifting**, **conditional branching**, and **unconditional branching** operations. The following table shows the base instructions and their corresponding assembly code instruction example:

Table 2.1: RISC-V Base Instructions (Patterson and Hennessy, 2017).

Category	Instruction	Example	Meaning
Arithmetic	Add	add x5, x6, x7	x5 = x6 + x7
	Subtract	sub x5, x6, x7	x5 = x6 – x7
	Add immediate	addi x5, x6, 20	x5 = x6 + 20
Data Transfer	Load word	lw x5, 40(x6)	x5 = Memory [x6 + 40]
	Store word	sw x5, 40(x6)	Memory [x6 + 40] = x5
	Load halfword	lh x5, 40(x6)	x5 = Memory [x6 + 40]
	Load halfword, unsigned	lhu x5, 40(x6)	x5 = Memory [x6 + 40]
	Store halfword	sh x5, 40(x6)	Memory [x6 + 40] = x5
	Load byte	lb x5, 40(x6)	x5 = Memory [x6 + 40]
	Load byte, unsigned	lbu x5, 40(x6)	x5 = Memory [x6 + 40]
	Store byte	sb x5, 40(x6)	Memory [x6 + 40] = x5
	Load upper immediate	lui, x5, 0x12345	x5 = 0x12345000
Logical	And	and x5, x6, x7	x5 = x6 & x7
	Inclusive or	or x5, x6, x7	x5 = x6 | x7
	Exclusive or	xor x5, x6, x7	x5 = x6 ^ x7
	And immediate	andi x5, x6, 20	x5 = x6 & 20
	Inclusive or immediate	ori x5, x6, 20	x5 = x6 | 20
	Exclusive or immediate	xori x5, x6, 20	x5 = x6 ^ 20
	Set if less than	slt x5, x6, x7	If (x6 < x7), x5 = 1 otherwise x5 = 0
	Set if less than, unsigned	sltu x5, x6, x7	If (x6 < x7), x5 = 1 otherwise x5 = 0
	Set if less than immediate	slti x5, x6, x7	If (x6 < x7), x5 = 1 otherwise x5 = 0
	Set if less than immediate, unsigned	sltiu x5, x6, x7	If (x6 < x7), x5 = 1 otherwise x5 = 0
Shift	Shift left logical	sll x5, x6, x7	x5 = x6 << x7
	Shift right logical	srl x5, x6, x7	x5 = x6 >> x7
	Shift right arithmetic	sra x5, x6, x7	x5 = x6 >>> x7
	Shift left logical immediate	slli x5, x6, 3	x5 = x6 << 3
	Shift right logical immediate	srli x5, x6, 3	x5 = x6 >> 3
	Shift right arithmetic immediate	srai x5, x6, 3	x5 = x6 >>> 3
Conditional Branch	Branch if equal	beq x5, x6, 100	if (x5 == x6) go to PC + 100
	Branch if not equal	bne x5, x6, 100	if (x5 != x6) go to PC + 100
	Branch if greater or equal	bge x5, x6, 100	if (x5 >= x6) go to PC + 100
	Branch if greater or equal, unsigned	bgeu x5, x6, 100	if (x5 >= x6) go to PC + 100
	Branch if less than	blt x5, x6, 100	if (x5 < x6) go to PC + 100
	Branch if less than, unsigned	bltu x5, x6, 100	if (x5 < x6) go to PC + 100
Unconditional Branch	Jump and link	jal x1, 100	x1 = PC + 4; go to PC +100
	Jump and link register	jalr x1, 100(x5)	x1 = PC + 4; go to x5 +100

These instructions can also be differentiated into several types based on the encoding formats used by the instructions, such as R-type (arithmetic and logical), I-type (immediate), S-type (store), SB-type (conditional branch), U-type (load upper immediate), and UJ-type (jump and link). The following table shows the type categorization for the instructions listed in the previous table:

Table 2.2: RISC-V Base Instruction Encoding Formats (Waterman and Asanovic, 2017).

Type	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	slt	0110011	010	0000000
	sltu	0110011	011	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0100000
	or	0110011	110	0000000
	and	0110011	111	0000000
I-type	lb	0000011	000	n.a.
	lh	0000011	001	n.a.
	lw	0000011	010	n.a.
	lbu	0000011	100	n.a.
	lhu	0000011	101	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	slti	0010011	010	n.a.
	sltiu	0010011	011	n.a.
	xori	0010011	100	n.a.
	srli	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
SB-type	beq	1100011	000	n.a.
	bne	1100011	001	n.a.
	blt	1100011	100	n.a.
	bge	1100011	101	n.a.
	bltu	1100011	110	n.a.
	bgeu	1100011	111	n.a.
U-type	lui	0110111	n.a.	n.a.
UJ-type	jal	1101111	n.a.	n.a.

The instruction encoding format utilized is different for different types of instructions, whereby specific fields within the 32-bit instruction code may signify different information. The information specified within an instruction code may include the destination register address, the source register address, an immediate value or an offset value, and the opcode, funct3, and funct7 to specify the operation to be carried out. The following table shows the different encoding formats based on the instruction type for the RISC-V base instruction set:

Table 2.3: RISC-V Instruction Field Specifications of Different Instruction Types (Patterson and Hennessy, 2017).

Type	Field						Comments
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic Instruction Format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads/Immediate Arithmetic
S-type	immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode	Stores
SB-type	immediate[12,10:5]	rs2	rs1	funct3	immediate[4:1,11]	opcode	Conditional Branch Format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional Jump Format
U-type	immediate[31:12]				rd	opcode	Upper Immediate Format

The base RISC-V instruction set architecture also features 32 general-purpose registers in the system that are 32-bits wide and can be used without any restrictions, with the exception of the register x0 being physically grounded, returning zero whenever it is read. Each general-purpose register among the 32 registers has an alternate name that corresponds to their usage in a standard RISC-V application binary interface (ABI). Due to the interchangeability of the functionalities of the general-purpose registers, the ABI is crucial for dictating the roles of the registers (Ledin, 2020). The following table provides detailed information for the 32 general-purpose registers:

Table 2.4: Alternate Names and Functionality of Base RISC-V General Purpose Registers.

Register	Alternate Name	Alternate Functionality
x0	zero	-
x1	ra	Function return address
x2	sp	Stack pointer
x3	gp	Global data pointer
x4	tp	Thread-local data pointer
x5	t0	Temporary storage
x6	t1	
x7	t2	
x8	fp	Frame pointer for function-local stack data
	s0	Saved registers
x9	s1	
x10	a0	Arguments passed to functions. Additional arguments are passed onto stack. Function return values are stored in a0 and a1.
x11	a1	
x12	a2	
x13	a3	
x14	a4	
x15	a5	
x16	a6	
x17	a7	
x18	s2	Saved registers
x19	s3	
x20	s4	
x21	s5	
x22	s6	
x23	s7	
x24	s8	
x25	s9	
x26	s10	
x27	s11	
x28	t3	Temporary storage
x29	t4	
x30	t5	
x31	t6	

2.1.2 RISC-V Computer Organization

Computer organization refers to the operational units and linkages that allow the architectural standards to be realized. Organizational characteristics refer to the hardware elements of a computer system such as the control signals, the interfaces between computer and peripherals, the memory technology employed. The architectural design of a computer system defines the operations to be performed by a computer and the fundamental principles applied in the creation and design of the datapath and its control system. In contrast, organizational design determines the implementation of various functions, whether through hardware or software implementation.

The two main logic elements utilized in computer systems are combinational elements and state elements. Combinational elements operate on data values and provide output data asynchronously. In contrast, state elements have internal storage, and data is only written into the storage when a proper clock signal is applied. State elements can also be described as sequential elements in which the output (next state) of the element depends on external inputs and the current state of the state element. An example of a combinational element within the RISC-V datapath would be the ALU unit, whereas a state element would be the general-purpose register used for storing useful information in the register file. For a standard clocking methodology, edge-triggered clocking is commonly used whereby data are only written when a positive or negative edge of a clock signal arrives at the sequential element. On every clock cycle, information from state elements is inputted to combinational elements, and the processed information is sent to a subsequent state element for storing. Signals need to arrive at subsequent state elements before the next clock cycle. Failing to do so will result in a loss of information.

The detailed explanation for the RISC-V computer organization will be divided into sections, each describing a specific functional block, otherwise known as a datapath element. These datapath elements work together to process an instruction, producing a desired outcome based on the instruction code supplied.

2.1.2.1 Program Counter

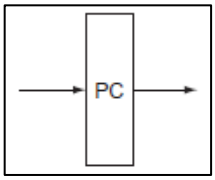


Figure 2.1: Program Counter Unit (Patterson and Hennessy, 2017).

The program counter is a simple 32-bit register that holds the instruction address, pointing towards the instruction to be executed by the microprocessor. The instruction address is sent to the instruction memory to fetch the corresponding instruction code from the program memory. On normal operations, the instruction address is incremented by 4 on each clock cycle. If a jump instruction is executed and the branch condition is fulfilled (**zero** flag is set), the program counter will be updated with a new effective target address specified by the sign-extended immediate value within the instruction. For a jump and link register instruction (JALR), the new effective target address is obtained through the sum of an offset and the content of a register both specified by the instruction. On the other hand, the effective target address of other jump or branch instructions are obtained from the immediate generate unit. The multiplexing of the new address to be updated onto the program counter is performed based on the **Branch** as well as the **JumpReg** control signals. The following table shows the new instruction address to be updated on the next cycle:

Table 2.5: Multiplexing of Instruction Address to be updated.

Control Signal	New Instruction Address
Branch and **Zero**	Effective target address is the sign-extended and left-shifted by 1 bit immediate value specified within instruction code
JumpReg (JALR instructions)	Effective target address is the sum of the register content (**rs1**) and the sign-extended offset (**Instruction [31:20]**) specified by the instruction
Otherwise	Instruction address is incremented by 4

2.1.2.2 Instruction Memory

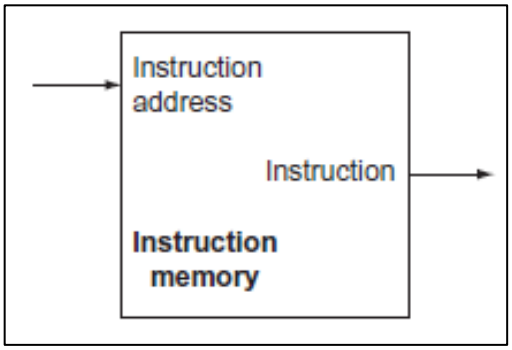


Figure 2.2: Instruction Memory Unit (Patterson and Hennessy, 2017).

The instruction memory block is a read-only memory block that contains all of the program instruction codes. The instruction address obtained from the program counter is used to fetch a 32-bit instruction code. The 32-bit instruction code contains useful information such as the opcode, source and destination register address, function code, immediate value or offset depending on the instruction type. The fetched instruction code is sent to several functional blocks in the datapath for further action. For a standard memory technology, each address points towards an 8-bit register, storing a byte (8 bits) of data. Thus, a 32-bit instruction code would require access to 4 registers in the memory to fetch the complete instruction code.

2.1.2.3 Register File

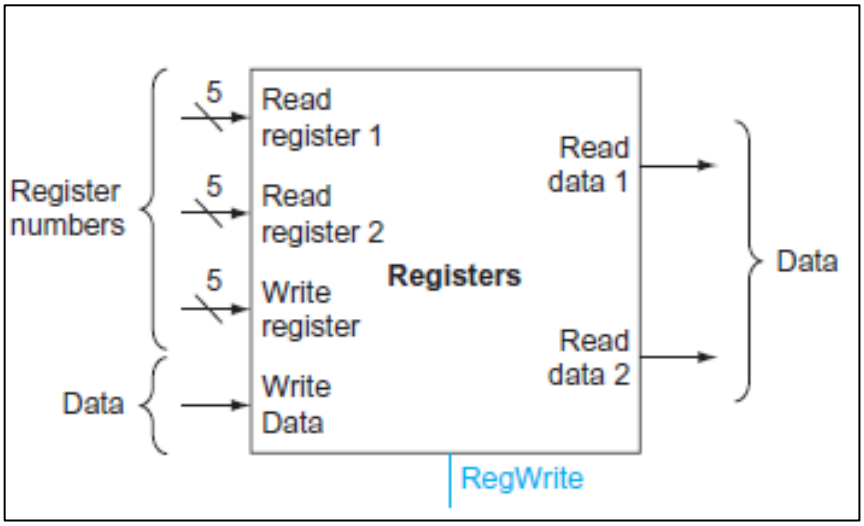


Figure 2.3: Register File Unit (Patterson and Hennessy, 2017).

The register file for a base RISC-V ISA contains 32 general-purpose registers that are each 32 bits wide. These general-purpose registers can be read or written and are accessed based on the register address specified in the instruction code. On register read operation, one or two data from the registers are read and sent to the ALU for further operations. The write operations of the registers are performed on clock edges whereby processed data from the ALU is rewritten onto the destination register or information from other sources are written onto the register. The multiplexing of the information to be written onto the register is controlled by explicit control signals such as **RegWrite** and **LinkReg**, which will be discussed in the control unit section.

2.1.2.4 Control Unit

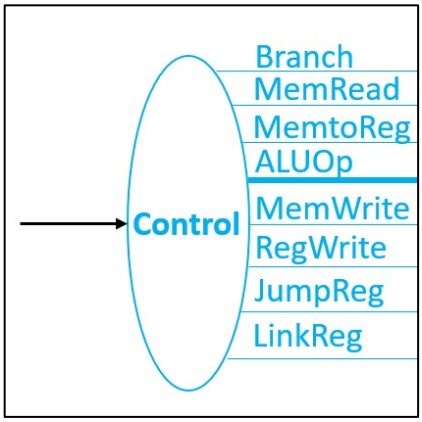


Figure 2.4: Control Unit (Patterson and Hennessy, 2017).

Control unit serves as the main decoding and control centre for the computer system. The fetched instruction code is decoded based on its opcode, and various control signals are subsequently adjusted to ensure proper functioning of the hardware. The control unit also outputs a 2-bit ALUOp control signal to the ALU control unit which will be further decoded to specify the instruction to be executed for the ALU. The following table shows the control signal values based on the instruction type decoded:

Table 2.6: Control Signal Values based on Instruction Type.

Instruction	JumpReg	LinkReg	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch
R-type	0	0	0	0	1	0	0	0
Load	0	0	1	1	1	1	0	0
I-type U-type	0	0	1	0	1	0	0	0
JALR	1	1	1	0	1	0	0	0
S-type	0	0	1	0	0	0	1	0
SB-type	0	0	0	0	0	0	0	1
UJ-type	0	1	0	0	1	0	0	0

The following table shows the various control signals utilized for controlling the hardware within the datapath and their corresponding description:

Table 2.7: Control Signals from the Control Unit (Patterson and Hennessy, 2017).

Control Signal	Description
RegWrite	Allows data on the Write Data input to be written onto the register specified by Write Register when asserted.
ALUSrc	Determines the source of second ALU operand. If asserted, the second ALU operand comes from the second register file output (Read Data 2). Otherwise, the second ALU operand is the sign-extended immediate specified in the instruction code.
Branch	Determines the instruction address to be updated for the next cycle. If asserted, and the condition is fulfilled (signified by assertion of the **zero** flag), the program counter is updated with the computed branch target address. Otherwise, the program counter is updated with the instruction address incremented by 4.
MemRead	Allows data memory contents designated by the memory address input to be read and placed onto the Read Data output when asserted.
MemWrite	Allows data memory contents designated by the memory address input to be replaced by the value placed on the Write Data input when asserted.
MemtoReg	Determines the source of the Write Data input to the register file. If asserted, the value fed to the register Write Data input is the data loaded from the data memory. Otherwise, the ALU output is written onto the register.
JumpReg	Determines the instruction address to be updated for the next cycle. If asserted, the program counter is updated with the target address formed by summing a register content and an offset specified in the instruction. Otherwise, the program counter is updated with the result determined from the **branch** control signal.
LinkReg	Allows return address (current instruction address + 4) to be utilized as input data for the Write Data input to the register when asserted. If de-asserted, the results from **MemtoReg** control signal is used as data for Write Data input.

2.1.2.5 ALU Control Unit

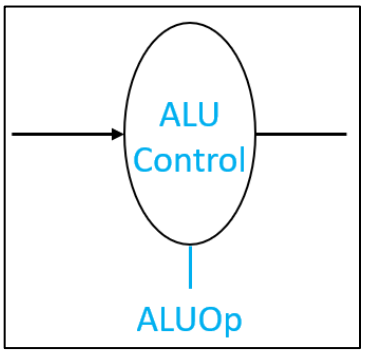


Figure 2.5: ALU Control Unit (Patterson and Hennessy, 2017).

ALU control unit is a functional unit that specifies the operation to be carried out by the arithmetic logic unit (ALU). From the **ALUOp** control signal received from the control unit as well as the funct3 and funct7 information specified in the instruction code, ALU control unit outputs a corresponding ALU control signal to the ALU. The following table shows the ALU control signal for several instructions:

Table 2.8: ALU Control Signal based on Instruction.

Instruction	ALU Op	Operation	funct7	funct3	ALU Action	ALU Control Signal
lw	00	load word	-	-	add	0010
sw	00	store word	-	-	add	0010
add	10	add	0000000	000	add	0010
addi	10	add immediate	0000000	000	add	0010
sra	10	shift right arithmetic	0100000	101	shift right arithmetic	0110
blt	01	branch if less than	-	100	compare and set (<)	1101
beq	01	branch if not equal	-	001	compare and set (=)	1000

2.1.2.6 Arithmetic Logic Unit

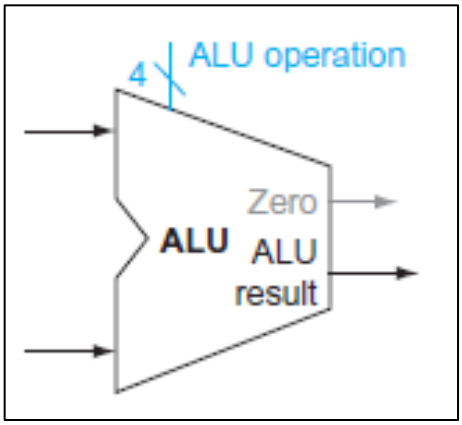


Figure 2.6: Arithmetic Logic Unit (Patterson and Hennessy, 2017).

The Arithmetic Logic Unit (ALU) performs arithmetic or logical operation on the data inputs. Depending on the instruction code decoded, different operations are performed by the ALU on the input data. Depending on the instruction type, the input data may originate from the register file or immediate generate unit. The multiplexing of the input data is controlled by **ALUSrc** control signal. The control unit first decodes the instruction code into a 2-bit **ALUOp** control signal followed by further specification by the ALU control unit into a 4-bit **ALU control** signal. The 4-bit ALU control signal specifies the specific operation on the input data. The processed 32-bit data is outputted to the register file for update, or it may be sent to the data memory to be used as memory address. Aside from the 32-bit data, an additional flag known as **zero** is also asserted if the processed output is zero. This zero flag is utilized for conditional branch instructions to signify condition fulfilment. The zero flag is asserted when the condition specified in the branch instruction is fulfilled.

2.1.2.7 Data Memory Unit

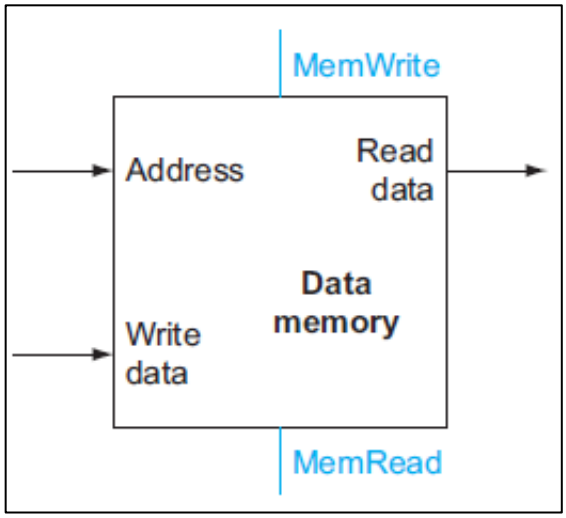


Figure 2.7: Data Memory Unit (Patterson and Hennessy, 2017).

Data memory is also known as the system's random-access memory (RAM). Based on the standard memory technology, the memory block comes with 8-bit registers that can be used as temporary data storage. The register within the memory block is accessed by first providing a memory address. Then, based on the control signals, the register content in the memory block can be updated (**MemWrite**) or read and used to update the system registers (**MemRead**). The data memory can only perform read or write operations one at a time and never both simultaneously.

2.1.2.8 Immediate Generation Unit

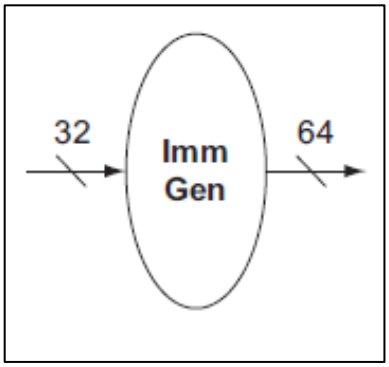


Figure 2.8: Immediate Generation Unit (Patterson and Hennessy, 2017).

The immediate generation unit constructs the immediate value or address from the instruction code based on the instruction's opcode. Depending on the instruction, the immediate value may be shifted left by 1 bit (Jump and Branch instructions), or sign-extended to 32 bits. The immediate value is sent to the arithmetic logic unit as data input for instructions that utilizes immediate values such as addi (add immediate). The following table shows the immediate value generated for different types of instructions:

Table 2.9: Immediate Value Generated corresponding to Instruction Type.

Instruction Type	Immediate Value
Load	{20{Instruction [31]}, Instruction [31:20]}
Store	{20{Instruction [31]}, Instruction [31:25], Instruction [11:8]}
I-type	{20{Instruction [31]}, Instruction [31:20]}
J-type	{19{Instruction [31]}, Instruction [31:20], 0}
SB-type	{19{Instruction [31]}, Instruction [31], Instruction [7], Instruction [30:25], Instruction [11:8] ,0}
UJ-type	{Instruction [31], Instruction [19:12], Instruction [20], Instruction [30:21]}
U-type	{Instruction [31:12], 12{0}}

2.1.3 Datapath Flow

By interconnecting the functional units, the full datapath for the processor can be visualized as shown in the figure below:

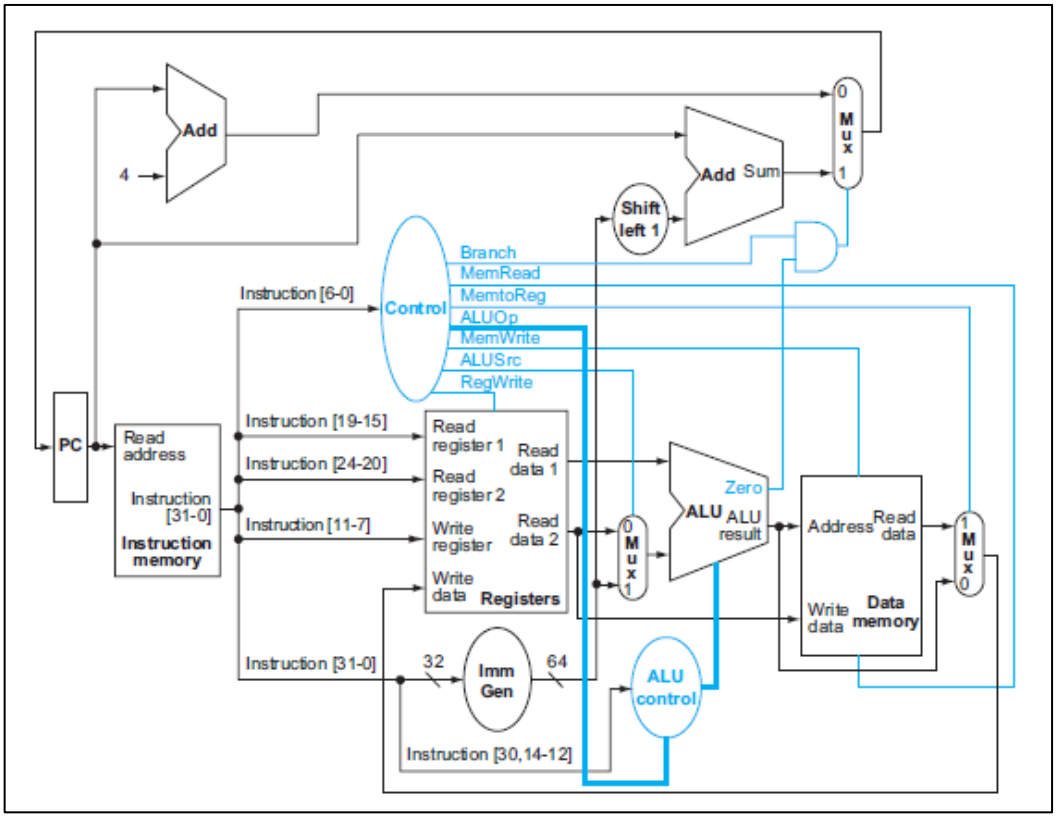


Figure 2.9: Simple Datapath of the Base RISC-V Processor (Patterson and Hennessy, 2017).

The functional units shown in the datapath interact through the interconnections and carries out the instruction fetched from the instruction memory. The datapath flow may differ depending on the instruction executed. Some instructions, however, may exhibit similar datapath flow with only minor differences such as the operation performed by the ALU. The datapath flow of various types of instruction for a single-cycle processor implementation will be discussed in the following section.

2.1.3.1 R-Type Instruction Datapath Flow

R-type instructions comprise of arithmetic and logical operations that utilize registers as operands. Upon performing the specified operation, the result from the ALU is to be written into the register specified by the instruction. The following shows an example datapath flow for an *add* instruction:

```
add     x3,     x6,     x7
```

1. The program counter provides the instruction address which is utilized to access and fetch the corresponding instruction (**add**) from the instruction memory. The program counter is incremented by 4.

2. The two registers (**x6** and **x7**) specified in the instruction are accessed and their corresponding data are read and sent to ALU for processing. The control unit decodes the instruction opcode, funct7 and funct3 fields and generates the required control signal (**RegWrite**) to control the hardware in the datapath.

3. The ALU performs the operation (**add**) specified by the ALU control unit on the data read from the registers (**x6** and **x7**).

4. The ALU output is written onto the destination register specified (**x3**).

2.1.3.2 I-Type Instruction Datapath Flow

I-type instructions comprise of load and immediate arithmetic and logical operations. Instead of a second register as an operand, these instructions have an immediate value or offset as the second operand. After performing the specified operation, the destination register is to be updated with a new data from ALU or data memory unit depending on the type of instruction. If the instruction is a load instruction, the data loaded from the data memory is used as register write data. Otherwise, the register write data for immediate arithmetic and logical instruction would be the ALU output data. The following shows an example datapath flow for a *load* instruction:

lw, x10, 100(x5)

1. The program counter provides the instruction address which is utilized to access and fetch the corresponding instruction (**load word**) from the instruction memory. The program counter is incremented by 4.

2. The register specified (**x5**) is accessed and the data read is sent to the ALU for further processing. The immediate value offset (**100**) specified by the instruction is generated by the immediate generation unit and sent to the ALU as the second operand. The control unit decodes the instruction opcode and funct3 fields and generates the required control signal (**MemRead**, **ALUSrc**, **MemtoReg**) to control the hardware in the datapath.

3. The ALU performs the operation (**add**) specified by the ALU control unit on the operands (**x5** and **100**). The resulting ALU output is utilized as the memory address for accessing the data memory unit.

4. The data stored on the data memory register specified by the memory address is read.

5. The data loaded from the data memory is written onto the destination register specified (**x10**).

2.1.3.3 S-Type Instruction Datapath Flow

S-type instruction consists of mainly store operations. Store operations stores the content of a general-purpose register onto the data memory unit, which requires MemWrite control signal to be set. The following example shows the datapath flow of a store instruction:

sb x5, 040(x6)

1. The program counter provides the instruction address which is utilized to access and fetch the corresponding instruction (**store byte**) from the instruction memory. The program counter is incremented by 4.

2. The two registers (**x5** and **x6**) specified in the instruction are accessed. The content of the first source register (**x6**) is to be added to the immediate offset (**40**) specified by the instruction and generated by the immediate generation unit whereas the content of the second source register (**x5**) is to be stored onto the data memory. The control unit decodes the instruction opcode and funct3 fields and generates the required control signal (**MemWrite**, **ALUSrc**) to control the hardware in the datapath.

3. The ALU performs the operation (**add**) specified by the ALU control unit on the operands (**x6** and **40**). The resulting ALU output is utilized as the memory address for accessing the data memory unit.

4. The contents in the second source register (**x5**) are stored onto the data memory register specified by the memory address computed.

2.1.3.4 SB-Type Instruction Datapath Flow

Conditional branch operations are categorized as SB-type instructions. Depending on the conditions specified by the instruction, the contents of the registers accessed are compared. If the condition is fulfilled, a branch occurs and the program counter is updated with the PC-relative effective target address specified by the instruction. The following shows an example datapath flow of a conditional branch instruction:

beq x20, x22, 100

1. The program counter provides the instruction address which is utilized to access and fetch the corresponding instruction (**branch if equal**) from the instruction memory.

2. The two registers (**x20** and **x22**) specified in the instruction are accessed and their corresponding data are read and sent to ALU for processing. The immediate generation unit generates the effective target address by summing the immediate offset (**100**) specified by the instruction with the current program counter address. The control unit decodes the instruction opcode and funct3 fields and generates the required control signal (**Branch**) to control the hardware in the datapath.

3. The ALU performs the operation (**compare**) specified by the ALU control unit on the operands (**x20** and **x22**). If the condition is true (**x20 == x22**), the result is set to zero and the zero flag is set. Otherwise the result is set to one and the zero flag is not set.

4. If the branch condition is fulfilled, the effective target address (**PC** + **100**) is updated into the program counter. Otherwise, the program counter is updated with the instruction address incremented by 4.

2.1.3.5 U-Type Instruction Datapath Flow

U-type instructions are special data transfer instructions that provides a 20-bit immediate value as an operand. The two instructions in this type are load upper immediate (lui) and add upper immediate to program counter (auipc) instructions. The following shows the datapath of a load upper immediate instruction:

lui x7, 0x12345

1. The program counter provides the instruction address which is utilized to access and fetch the corresponding instruction (**load upper immediate**) from the instruction memory. The program counter is incremented by 4.

2. The immediate generation unit forms the data from the immediate value specified (**12345000_{hex}**). The control unit decodes the instruction opcode and funct3 fields and generates the required control signal (**ALUSrc**, **RegWrite**) to control the hardware in the datapath.

3. The ALU loads the immediate value (**12345000_{hex}**) as its output.

4. The ALU output (**12345000_{hex}**) is used as the register write data and the data is written into the destination register specified (**x7**).

2.1.3.6 UJ-Type Instruction Datapath Flow

Unconditional branch instructions are categorized as UJ-type instructions. In the latest base version of RISC-V ISA, the UJ-type instruction comprises of only jump and link instruction (jal). The following shows an example datapath flow for a jump and link instruction:

jal x20, 100

1. The program counter provides the instruction address which is utilized to access and fetch the corresponding instruction (**jump and link**) from the instruction memory.

2. The immediate generation unit generates the effective target address by summing the immediate offset (**100**) specified by the instruction with the current program counter address. The control unit decodes the instruction opcode and funct3 fields and generates the required control signal (**LinkReg**, **RegWrite**) to control the hardware in the datapath.

3. The ALU output is set to zero and the zero flag is raised.

4. Instruction address of the instruction following the jump instruction (**PC** + **4**) is utilized as the register write data and written into the register specified by the instruction (**x20**). The program counter is updated with the PC-relative effective target address computed by the immediate generation unit (**PC** + **100**).

2.1.4 Processor Pipelining

For a single-cycle processor system, the datapath can only process one instruction at a time. The performance is then limited by load type instructions with the longest signal path, accessing up to five functional units (program counter, instruction memory, register file, ALU, data memory). The limitation imposed on the clock rate and the inefficiency regarding the usage of the functional units can be resolved through pipeline implementation which will be discussed in this section.

2.1.4.1 Pipeline Implementation

Pipelining aims to improve the efficiency and throughput of the processor data flow by overlapping instruction executions. Instruction execution in a computer system can be categorized into the following five stages:

Instruction fetch / IF:	Fetch instruction from memory.
Instruction decode / ID:	Read registers and decode the instruction. Generate the corresponding control signal to control the hardware. Also generate the immediate value or offset if necessary.
Execution or address calculation / EX:	Execute the operation or calculate the target address based on the control signal provided.
Data memory access / MEM:	Access an operand in data memory for read or write operation if necessary.
Writeback / WB:	Write result into register if necessary.

By separating the single-clock cycle datapath flow into five pipeline stages, the performance of the processor is improved by approximately four times. The following figure shows the concept of separating the datapath flow into several stages:

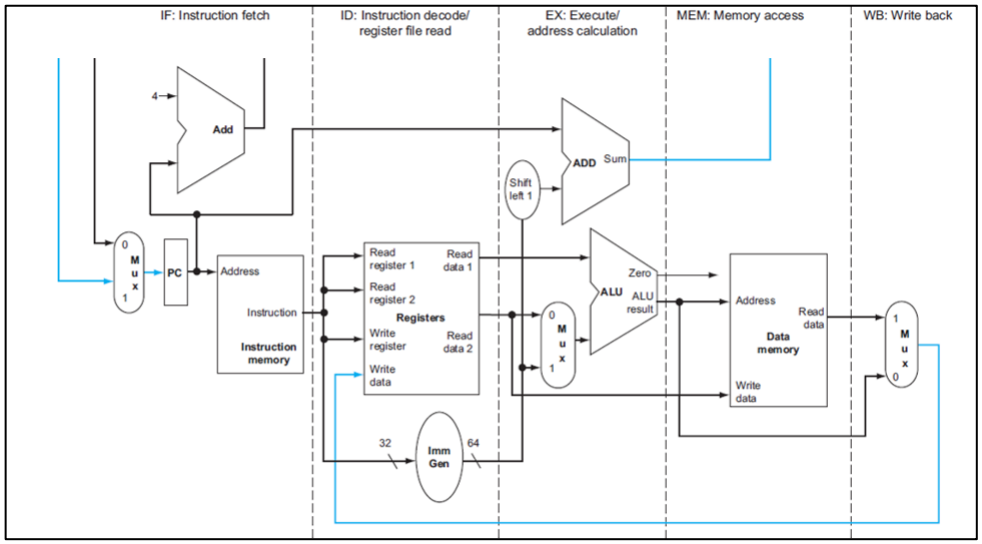


Figure 2.10: Separation of Single-Cycle Datapath for Pipeline Implementation (Patterson and Hennessy, 2017).

Pipelining segregates the different stages of instruction execution and utilize each of the stages to execute different instructions. This allows the processor to process multiple instructions at different stages at a given time, significantly increasing its throughput. The following graphical pipeline diagram showcases the instruction execution of different instructions at a given clock cycle:

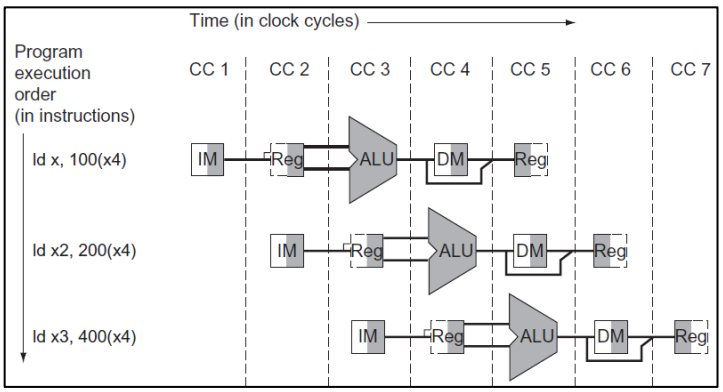


Figure 2.11: Multiple Instructions executed with Pipeline Implementation (Patterson and Hennessy, 2017).

To retain the information of an instruction and pass it down the pipeline stages, several registers known as pipeline registers will be required to be placed between stages. By placing the pipeline registers between the pipeline stages, the information processed on a pipeline stage will be stored onto the pipeline register on the next cycle and utilized for processing on the subsequent pipeline stage. This effectively advances the execution of an instruction from one pipeline stage to another on each clock cycle. The naming convention for these pipeline registers are based on the pipeline stages separated by the pipeline registers. For an example, the pipeline register separating the instruction fetch (IF) and instruction decode (ID) stages is named as IF/ID pipeline register. The following figure shows the pipelined datapath with the pipeline registers:

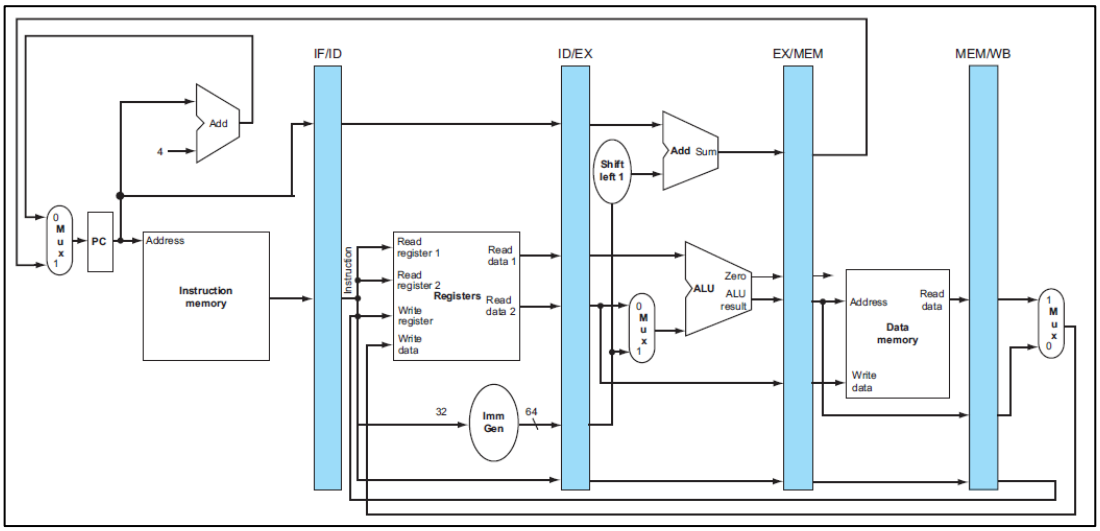


Figure 2.12: Pipelined Datapath (Patterson and Hennessy, 2017).

As the control signals decoded from an instruction by the control unit on the instruction decode (ID) stage transcends multiple pipeline stages, the control signals decoded for a given instruction will have to be passed down the pipeline register to ensure the hardware control signals for a given instruction is passed down alongside the execution of the instruction. The following pipelined datapath shows the addition of control elements to the pipeline system and the passing of the control signals down the pipeline stages:

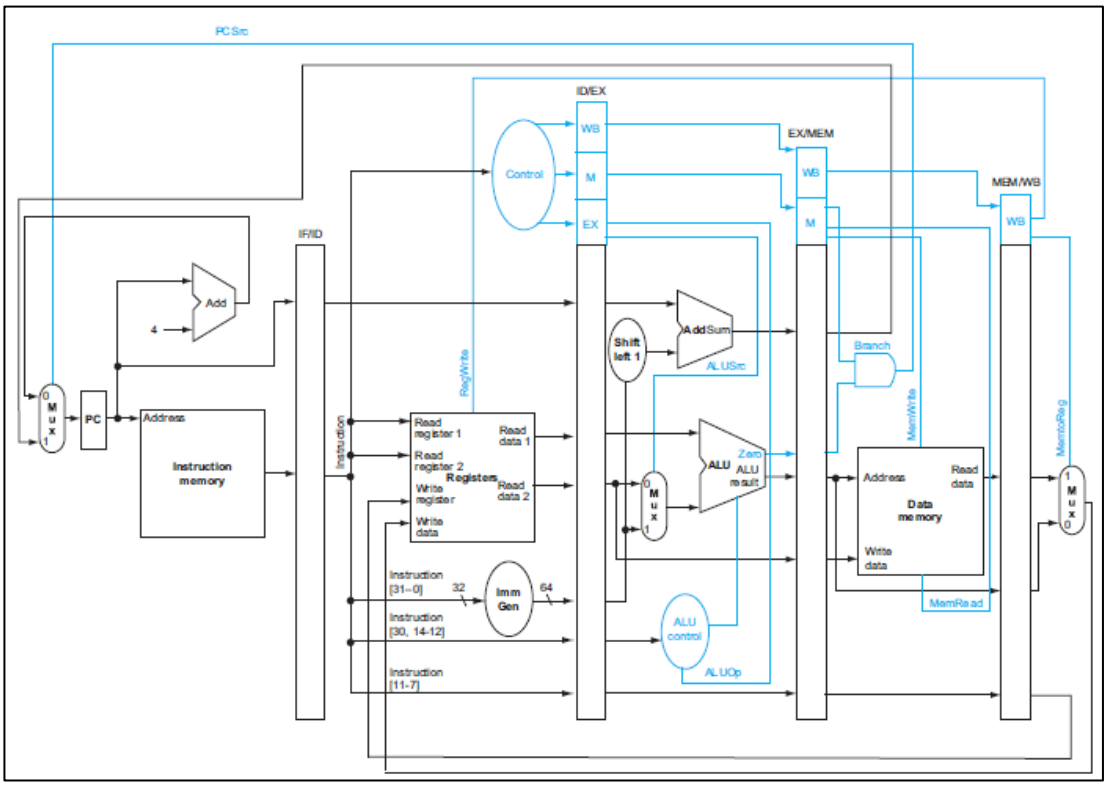


Figure 2.13: Pipelined Datapath with Control Elements integrated (Patterson and Hennessy, 2017).

The following section provides a thorough explanation of the operations at each pipeline stages:

Instruction Fetch / IF: The program counter provides the instruction address which is utilized to access and fetch the corresponding instruction code from the instruction memory. Instruction address and instruction code are stored onto the IF/ID pipeline register.

Instruction Decode / ID: The instruction code obtained from the IF/ID pipeline register is used to access registers (register file), generate immediate value (immediate generation unit), as well as generating control signals (control unit). The outputs from the functional units are stored onto the ID/EX pipeline register.

Execution / EX: The operands to be processed, alongside the control signal specifying the operation to be executed are obtained from the ID/EX pipeline register and sent to the ALU. The processed output is then stored onto the EX/MEM pipeline register.

Memory Access / MEM: The register content to be stored for store operations and the memory address for data memory access are obtained from EX/MEM pipeline register. Data loaded from the data memory and the ALU output from EX/MEM pipeline register is written onto MEM/WB pipeline register.

Write back / WB: The data to be written into the register file is obtained from the MEM/WB pipeline register and written into the register destination specified by the instruction stored on the MEM/WB pipeline register.

Implementing pipeline in RISC-V architecture is relatively straightforward than x86 computer architecture due to the fixed 32-bit length of the instructions. The few variants of instruction formats with the same fields for defining information, such as destination register address and source register addresses, also made it easy for pipeline implementation. Aside from that, the simplicity of the base instructions in which memory operands are only utilized in loads or stores allowed the use of the execution stage to calculate the memory address and immediately access the memory address in the following memory access stage.

Along with improving performance and efficiency, pipeline implementation can also bring complicated situations whereby the pipeline flow needs to be halted due to hardware limitations. These events are known as pipeline hazards, and there is a total of three different types of hazards: structural **hazards**, **data hazards**, and **control hazards**. The following section provides information for each hazard that needs to be considered for the pipeline implementation.

2.1.4.2 Structural Hazards

Structural hazards occur when there is multiple access to the same hardware by different instructions in the pipeline at a given time. Due to hardware limitations, hardware in the datapath can only be accessed by one instruction at a time, thus necessitating the halting of the datapath for multiple hardware accesses. This form of hazard can be seen in cases where Von-Neumann Architecture is employed, whereby the same memory unit is utilized for storing program instructions and data. When the memory unit is accessed for fetching instructions, memory access for memory write or memory read would be halted. However, this hazard can be resolved if Harvard Architecture is employed whereby separate memory units are utilized for storing program instructions and data. The instruction fetch and data memory access can then be performed simultaneously as each operation accesses different hardware.

2.1.4.3 Data Hazards

Data hazard occurs when there is data dependency between different instructions processed at different pipeline stages at a given time. With pipeline implementation, the datapath processes multiple instructions at a given time. In situations where there is a data dependency between the instructions in the pipeline datapath, the processed data needs to be forwarded to the corresponding pipeline stage to ensure the correct data is processed. Consider the following instruction segment:

```
and    x11,   x3,    x16
add    x12,   x5,    x11
```

From the instruction segment shown, the result of the *and* operation which is to be updated to the register *x11* is to be immediately be used as an operand of the subsequent *add* instruction. To ensure the correct information is utilized for the *add* instruction, the pipeline will have to be halted for three clock cycles to ensure completion of execution of the *and* instruction up to the writeback stage. Alternatively, forwarding of the data from a later pipeline stage to the pipeline stage requiring the data can resolve the hazard through additional hardware.

In situations where the processed data arrives at a later time, pipeline stalling becomes necessary. These situations often arise from load instructions followed by instructions with data dependency on the data to be loaded from data memory, known as load-use cases. Consider the following example instruction segment:

```
lw     x11,        02a(x5)
add    x12,   x5,    x11
```

The updated data for the register *x11* only arrives at MEM stage upon memory read by the *load word* instruction, which necessitates pipeline stalling for at least one cycle. Through the combination of pipeline stalling and data forwarding mechanisms, the pipeline stall can be minimized to only one cycle.

2.1.4.4 Control Hazards

When branch or jump instructions are introduced to the pipeline, the instructions following the branch or jump instructions may originate from a new address or remain the fetched instructions. In cases where branch or jump is executed, the fetched instructions are invalidated and need to be removed from the pipeline. This form of hazard is known as control hazards, also known as branch hazards. Consider the following instruction segment:

```
                add   x4,   x6,   x6
                beq   x1,   x0,   40
                lw    x3,   400(x0)
                sw    x3,   400(x0)

Branch Address: or    x7,   x8,   x9
```

The decision of whether the instruction (*lw*) following the conditional branch instruction (*beq*) or the instruction from the new target address (*or*) is to be executed can only be known upon the condition checking by ALU on the execution pipeline stage (EX). If the branch condition is fulfilled, the subsequent instruction (*lw*) in the instruction decode (ID) stage and the following instruction (*sw*) in the instruction fetch (IF) stage would need to be removed. One method of overcoming this hazard is to stall whenever a conditional branch instruction is fetched from the instruction memory. This, however, would cause a significant reduction in the processor's performance, especially when there is a large number of conditional branch instructions in the program.

Alternatively, branch prediction can be utilized. By predicting conditional branches are untaken by default, pipeline flow will be at full speed if the prediction is correct. Only if the prediction is incorrect whereby the branches are taken, the pipeline flushes the incorrect information from the pipeline and fetches instructions from the new address. The following figures show the utilization of branch prediction as a solution to control hazard:

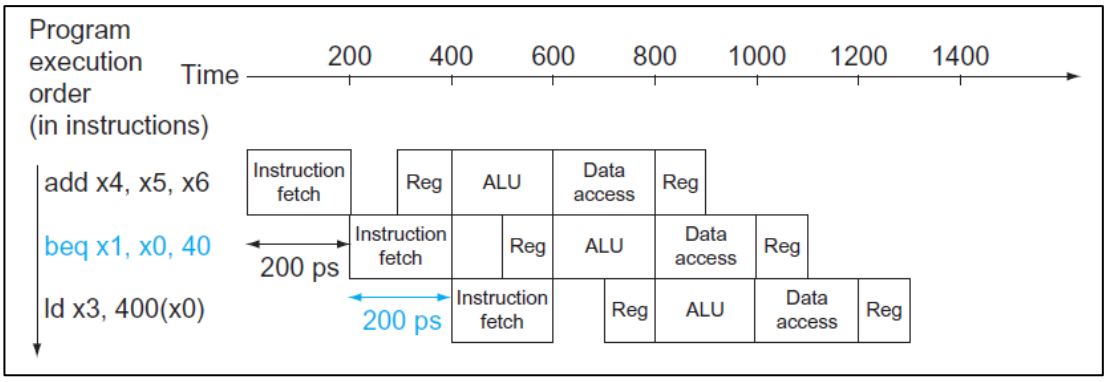


Figure 2.14: Pipeline flows at full speed when Branch Prediction is correct (Patterson and Hennessy, 2017).

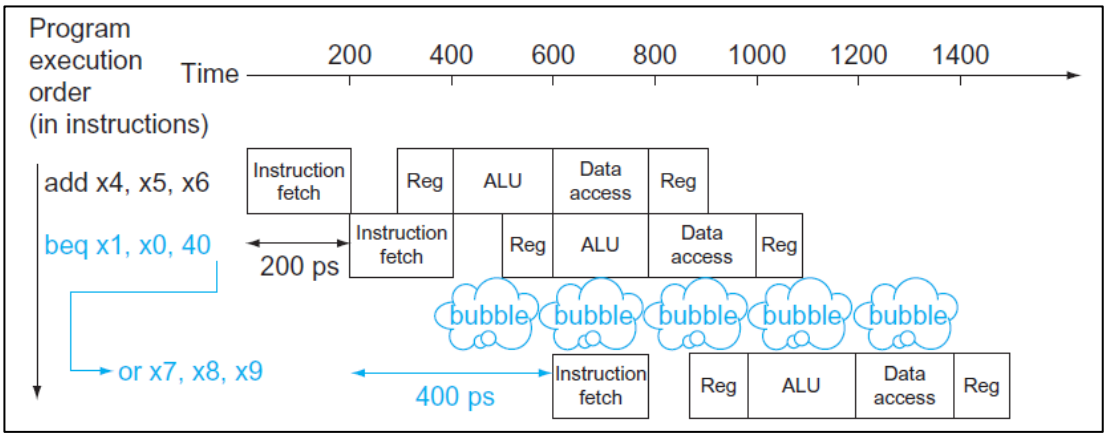


Figure 2.15: Pipeline flushes only when Branch Prediction is incorrect (Patterson and Hennessy, 2017).

The branch prediction algorithm for can be coded in a sophisticated manner to further enhance the processor's overall performance. This would however, require an advanced implementation for the branch prediction.

2.1.4.5 Data Forwarding

Data forwarding refers to the passing of processed data from a later pipeline stages to the pipeline stage requiring the updated data. Through hardware implementation, data forwarding serves as the primary solution for data hazards. Considering the following instruction segment with data hazard:

```
and   x11,  x3,   x16
add   x12,  x5,   x11
```

When the *add* instruction enters the instruction decode (ID) stage, the information required for the execution (EX) stage would require the processed information from *x11* register which is still in the execution (EX) stage. As the *and* instruction only updates the register *x11* on writeback (WB) stage, the *add* instruction will utilize the outdated data for *x11*. Data forwarding forwards the output result of the *and* instruction from the EX/MEM pipeline register to the ALU for the execution (EX) stage of *add* instruction. The following figure provides graphical representation of the data forwarding process:

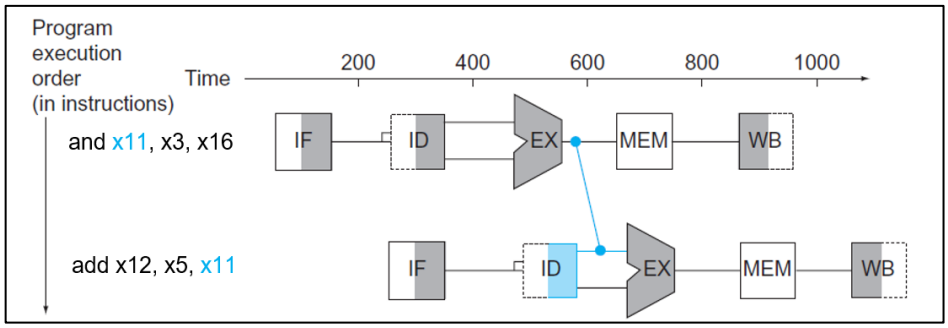


Figure 2.16: Graphical Representation of Forwarding (Patterson and Hennessy, 2017).

Through data forwarding, the data required can be forwarded, bypassing the memory access (MEM) and writeback (WB) pipeline stages. This allows the pipeline data flow to be correct and instructions to be executed using the updated information. The hardware implementation for the data forwarding unit can be done by checking

the source register addresses from the ID/EX pipeline register and the destination register address from both EX/MEM and MEM/WB pipeline registers. If the source register address at the ID/EX pipeline register matches the destination register of previous instructions, forwarding of data from either EX/MEM or MEM/WB pipeline registers can then be performed, sending the updated information from the respective pipeline to the ALU as input operands. Such implementation would also require additional multiplexing to be performed at the ALU input, as shown in the following figure:

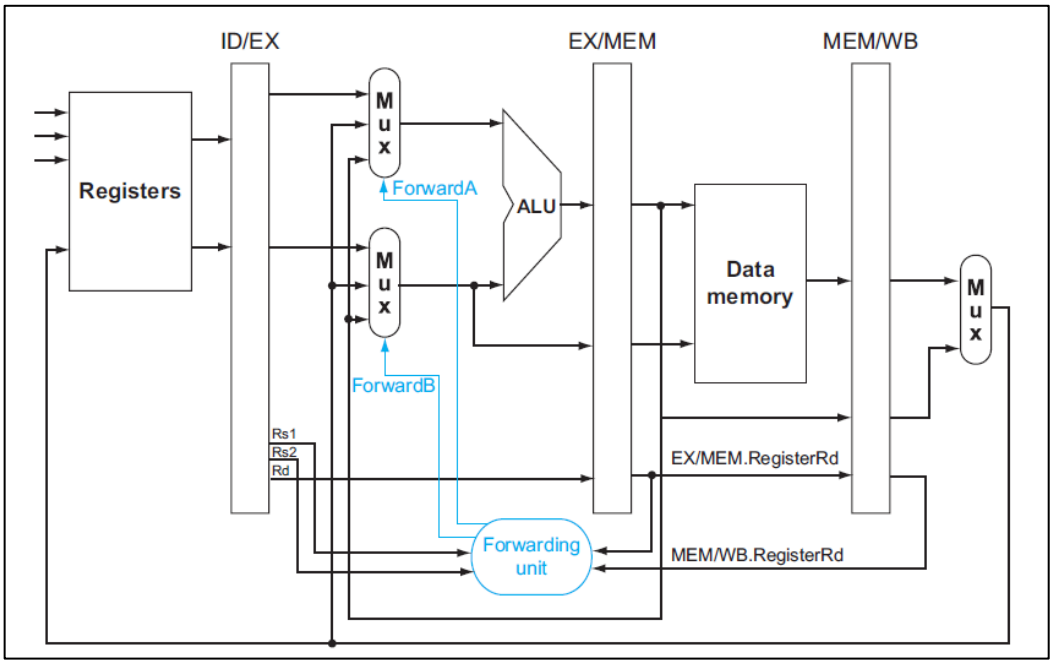


Figure 2.17: Implementation of Data Forwarding on the Pipelined Datapath (Patterson and Hennessy, 2017).

Table 2.10: Forwarding Control Signal and their Description.

Forwarding Control Signal	Value	Description
ForwardA	00	Data from first source register is sent to ALU.
	01	Data from MEM/WB pipeline register is forwarded to ALU.
	10	Data from EX/MEM pipeline register is forwarded to ALU.
ForwardB	00	Data from ALUSrc multiplexing is sent to ALU.
	01	Data from MEM/WB pipeline register is forwarded to ALU.
	10	Data from EX/MEM pipeline register is forwarded to ALU.

2.1.4.6 Pipeline Stalling

Pipeline stalls are a crucial mechanism in pipelining due to the unpreventable circumstances whereby stalling is necessitated to ensure the correct data or instruction is processed. Upon stalling, writes to program counter and IF/ID, ID/EX and EX/MEM pipeline registers are halted whereas MEM/WB pipeline register continue executing the instructions they contain. Consider the following load-use case:

```
lw      x11,    020(x5)
add     x12,    x5,     x11
```

As the updated information of *x11* register can only be obtained when the *load word* instruction reaches memory access (MEM) stage, the pipeline needs to be stalled for one cycle to accommodate for the address calculation (execution stage). The following figure shows the graphical representation of the stalling mechanism of the pipeline:

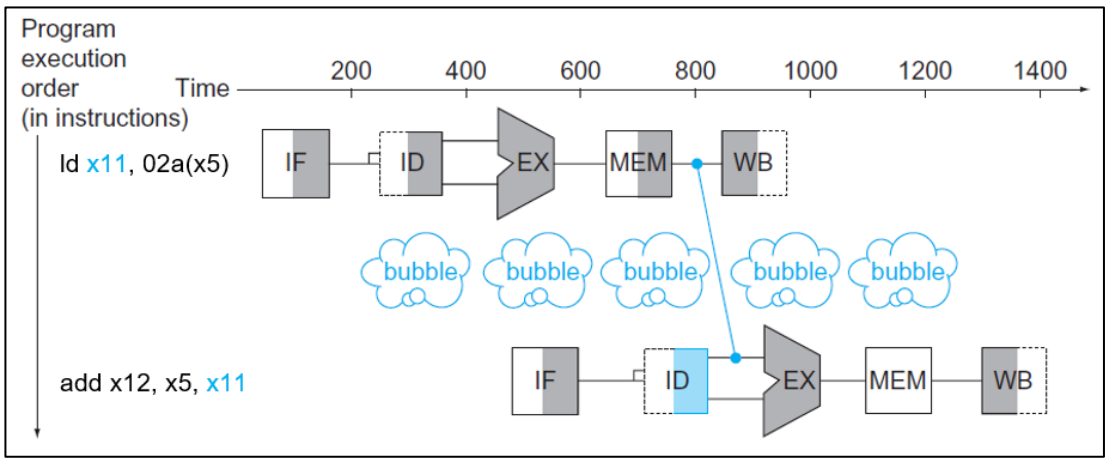


Figure 2.18: Graphical Representation of Stalling and Forwarding (Patterson and Hennessy, 2017).

Upon stalling, bubbles are inserted into the pipeline datapath, which represents no operation (nops) in terms of instruction execution. For RISC-V ISA, pipeline stalling halts the writing of new data onto program counter, IF/ID and ID/EX pipeline

registers whereby the pipeline registers retain the old instruction until the stall signal is removed. The EX/MEM pipeline register is inserted with bubble by setting the instruction code and control signal to be written to zero. The hardware implementation of pipeline stall mechanism is done through a hazard detection unit. The hazard detection unit detects for load use cases by checking the **MemRead** control signal on the EX/MEM pipeline register and comparing the destination register address on the EX/MEM pipeline register against the source register addresses stored on ID/EX pipeline register. If data is to be read from the data memory unit onto the destination register (**MemRead**) and the destination register address matches the source register addresses, the hazard detection unit asserts the stall control signal. The information stored on the program counter, IF/ID, ID/EX and EX/MEM pipeline registers will be retained whereas MEM/WB pipeline register continue with the instruction execution. When the load instruction proceeds to the memory access (MEM) stage and stores the data loaded onto MEM/WB pipeline register, the updated information can then be forwarded to the execution stage for the subsequent instruction and the stall signal can then be removed. The following figure shows the datapath with the hazard detection unit implemented:

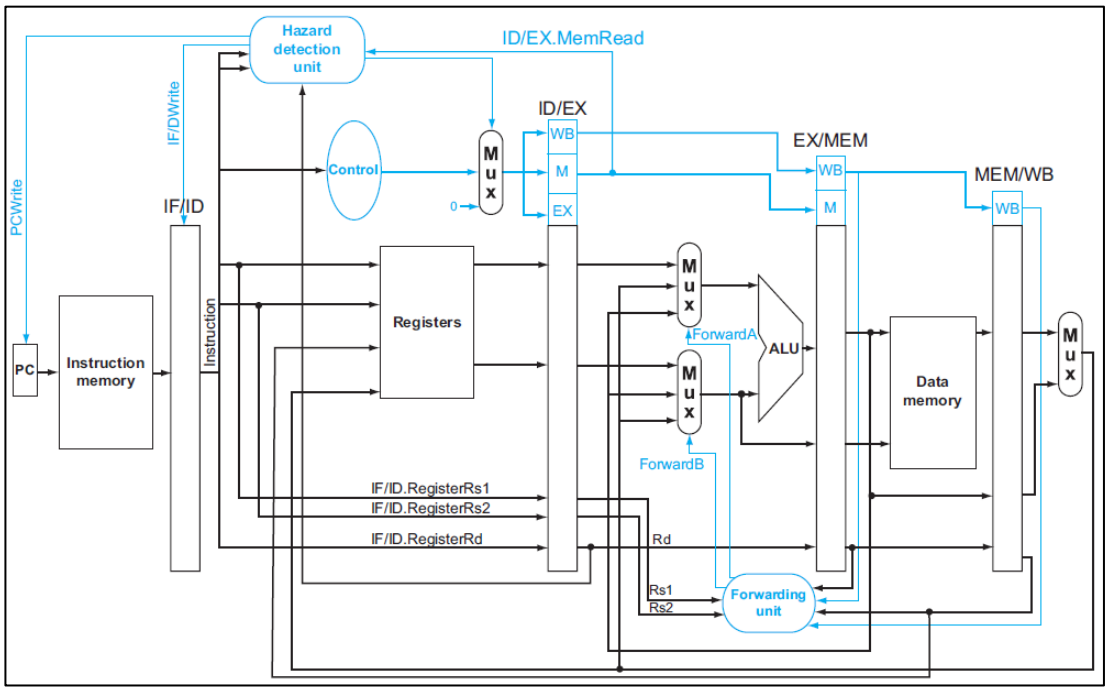


Figure 2.19: Implementation of Data Forwarding and Data Stalling on the Pipelined Datapath (Patterson and Hennessy, 2017).

2.1.4.7 Pipeline Flushing

Aside from data forwarding and data stalling, another key mechanism known as pipeline flushing is required for the proper functioning of branching in a pipelined datapath. Pipeline flushing resolves control hazards by flushing the invalidated instructions out of the pipeline registers when a branch condition is fulfilled, preventing the system from executing the invalidated instructions fetched.

Flushing of the information on the pipeline can be performed by loading zero values onto the pipeline registers. By loading zero values as instruction code and control signals, hardware components in other stages of the pipeline will perform no action. As the branch comparison result is only known at the execution stage (EX), the pipeline registers prior to the execution stage containing the invalidated instructions will be flushed. The pipeline flush mechanism can be implemented alongside the hazard detection unit whereby control signals from ID/EX pipeline register (**Branch, RegLink**) and the zero flag from ALU will trigger the flushing mechanism.

To provide a thorough understanding on the flushing mechanism, consider the following instruction segment:

```
beq    x1,    x1,    40
add    x4,    x6,    x6
lw     x3,      400(x0)
```

Clock Cycle 1: The branch instruction is fetched from the instruction memory and stored onto the IF/ID pipeline register on the next cycle.

Clock Cycle 2: The information of the branch instruction is decoded and passed to the ID/EX pipeline register on the next cycle. The *add* instruction is fetched and stored onto IF/ID pipeline register on the next cycle.

Clock Cycle 3: The information of branch instruction from ID/EX pipeline register are processed. As the branch condition is fulfilled, ALU sets the zero flag to HIGH. The hazard detection unit receives HIGH value for both **Branch** and **Zero**, thus asserting flush control signal. The information of *add* instruction is processed and passed onto ID/EX pipeline register on the next cycle whereas *lw* instruction is fetched and stored onto IF/ID pipeline register on the next cycle.

Clock Cycle 4: The assertion of flush control signal loads the contents within IF/ID, ID/EX and EX/MEM pipeline registers with zero values to ensure no operation, removing the invalidated instructions. The program counter is updated with the new target effective address on the next cycle.

Clock Cycle 5: Instruction from the new target address is fetched and stored onto IF/ID pipeline register.

From the description provided, the pipeline flush mechanism can resolve control hazards at the cost of two clock cycles, which can take a heavy toll on the processor performance if there are many conditional branch instructions in the program. Alternatively, the conditional branch comparison execution can be advanced to the instruction decode pipeline stage to further reduce the branch flush delay by one clock cycle. However, advancing the branch comparison would require data forwarding and pipeline stalling to be reworked and more complicated control. Nevertheless, reducing one clock cycle for branch execution would significantly improve the overall processor performance. The following figure shows the datapath implemented with the branch comparison forwarded to the instruction decode stage:

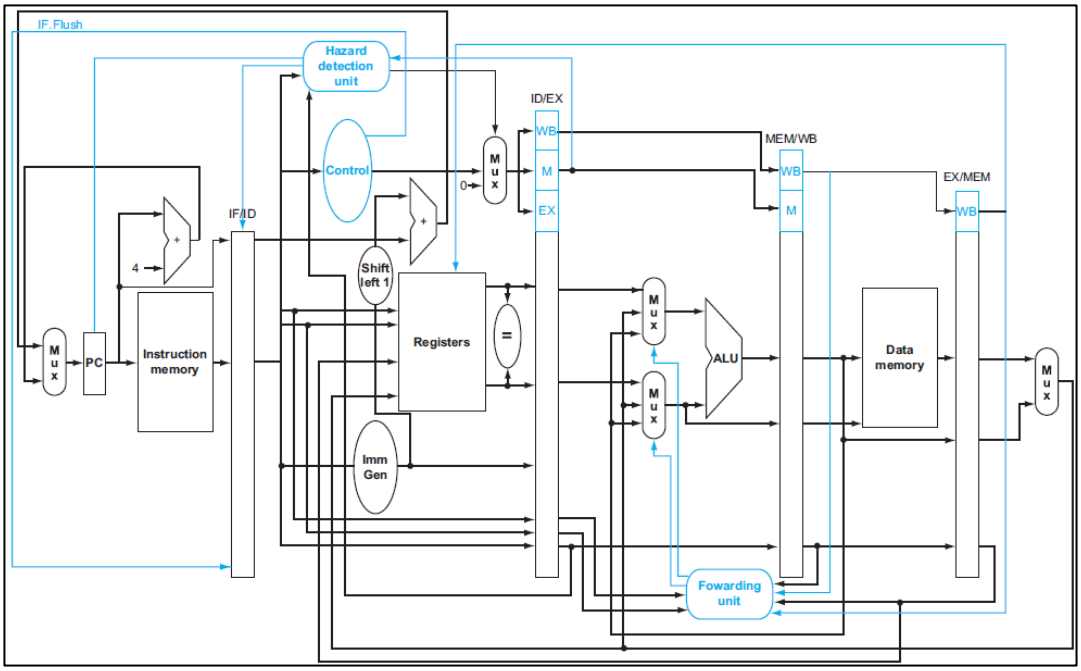


Figure 2.20: Pipelined Datapath with Branch Comparison forwarded to Instruction Decode Stage (Patterson and Hennessy, 2017).

2.1.5 Summarized Review for RISC-V Computer Architecture

From the detailed description for the RISC-V computer architecture provided by the textbook “Computer Organization and Design RISC-V Edition” published by Patterson and Hennessy, a well-established fundamental understanding of the datapath flow of RISC-V computer architecture with pipeline implementation is achieved. Despite not providing the full description for all the workings of a RISC-V system, such as examples for all the instructions within the RISC-V base instruction set and the complete RTL coding for the RISC-V processor, the textbook “Computer Organization and Design: RISC-V Edition” has done well in conveying information on the datapath flow of a RISC-V pipeline implementation with the detailed description for several examples.

Although the project's primary focus is the verification portion, a well-established understanding of the computer architecture is just as crucial as verifying the system. With a well-established understanding of how the system works, a thorough verification can be performed with the test engineer understanding the architecture flow and greatly aiding the debugging process. Aside from verifying the design under test with the fundamental knowledge on the datapath flow, the knowledge on the RISC-V computer architecture is also helpful for implementing a reference model to be compared with the design under test. A reference model is a model that produces the expected outcome in a simulation whereby the results from a design under verification will be compared to. Despite the nature of the reference model to be deemed as the model that provides the expected outputs, in the industry, the reference model may have bugs within the model. Therefore, a verification engineer must have a well-established understanding of the computer architecture such that such cases whereby the reference model is at fault can be detected and fixed.

The detailed information obtained from the textbook has encouraged and provided sufficient information to build a reference model from scratch. As such, the information listed in this report is the main reference document for the architecture of the RISC-V reference model alongside the RISC-V instruction set architecture manual.

2.2 Functional Verification

Functional defects are often introduced to the system design during RTL code design. These functional defects can be caused by logical errors in the coding, miscommunication between the design team, complexity of the design, and more. Verification is a much-needed process in the design flow as it ensures these functional defects are captured and maintains the integrity of the design functionality with the design specifications. Capturing these functional defects at an earlier stage can help prevent the manufacture and deployment of functionally defective designs, losing many resources, money, and time (Kaeslin, 2014). Therefore, design verification is a much-emphasized process in product development whereby design verification often consumes as much as 80% of the total product development time (Wang, Chang and Cheng, K.T.T., 2009).

2.2.1 SystemVerilog as Functional Verification Language

Verilog is an industrial standard Hardware Description Language (HDL) used mainly to describe circuits and systems. In electronic design, Verilog is utilized for simple verification of digital circuits at RTL abstraction level, timing analysis, test analysis, and logic synthesis (Doulos, n.d.). Despite the capability to perform verification, Verilog has very limited features, which is insufficient to meet the verification requirement for today's standards. In today's design complexity, a tool better than Verilog needs to be utilized to verify systems with a complex design.

SystemVerilog, an extension of Verilog that supports object-oriented programming, allows for advanced functional verification constructs, further opening up possibilities for incorporating advanced functional verification methodologies such as universal verification methodology (UVM) and more. With the added capability to perform constrained random stimuli generation and incorporate object-oriented programming (OOP) in test environment construction, SystemVerilog is a much-developed functional verification language compared to Verilog (Chip Verify, n.d.).

2.2.2 Functional Verification Requirements

The basis of functional verification is to verify the features implemented in a design and capture all functional defects. With the increasing complexity of modern-day system designs, incorporating additional criteria towards the longevity of functional verification is much needed.

Reusability of the verification methodology is one of the highly-focused aspects of functional verification. Manual design of verification testbench for complex designs often consumes a lot of time. Incorporating reusability in verification testbench with object-oriented programming through reusable verification components can allow the verification environment to be designed much shorter and robustly. Verification components designed for reusability allow verification intellectual properties (VIP) to be reused across components, multiple chips, and in different organizations.

Automation is another critical aspect of design verification that can significantly enhance verification effectiveness. Automation of test case execution allows verification to be performed without manually driving the inputs to the design under test. Automation of result analysis through the implementation of self-checking testbench helps identify discrepancies between results obtained and the expected outcome, removing the requirement of manual inspection on the results obtained and significantly improving the efficiency of the verification process. Automation of functional coverage analysis helps track and measure the progress of functional verification by providing insights on the design features that have and have yet been tested.

Standardized coding guidelines for verification component and environment development is another crucial aspect for design verification. A standardized approach towards verification environment design ensures a consistent working design and aids with the debugging process of the verification environment (Singhal, 2015). The guidelines provided also allow codes to be written and maintained easily.

2.2.3 Functional Verification Technologies

Simulation-based and formal verification are the two main functional verification technologies used in the industry. Simulation-based verifications generate and drive stimuli to a design under test and a reference model. The outputs from both models are then obtained and compared. Discrepancies between the results obtained are categorized as functional defects within the design. On the other hand, formal verification does not require input vectors but instead is an output-driven form of verification. Formal verification first defines the output behaviour for the design and identifies the possible inputs and state conditions for failures.

The main difference between the two types of verification stated is the requirement of input vectors for the verification process. As aforementioned, simulation-based verification is **input-driven**, whereas formal verification is **output-driven**. In simulation-based verification, inputs are driven to the design under test one at a time, whereby a scoreboard checks for the correctness of the design behaviour. Formal verification utilizes constraints to identify the legal input behaviours. Through sufficient runtime, input patterns corresponding to the constraints set can be identified and verified by the scoreboard for behavioural correctness of the design (Oski Technology, 2020). Abstractly speaking, simulation-based verification checks for one output point at a time, whereas formal verification checks groups of points at a time. By performing verification in groups of points at a time, the set properties of the groups of points tested must be further verified against the design specifications, thus making formal verification less intuitive and harder to use (Lam, 2005.).

Due to the lack of intuitiveness, formal verification is applicable for designs of moderate complexity. As the project is concerned with verifying a processor system with a large number of blocks integrated, functional verification becomes unsuitable to be utilized. Therefore, the project will use **simulation-based verification** whereby input stimuli are generated and driven to the design. The results obtained from the design are subsequently compared with a reference model.

2.2.4 Functional Verification Approaches

For simulation-based verification, a total of five different verification approaches can be taken. Multiple approaches are often required for a complex design to achieve sufficient functional coverage.

Directed verification manually generates test stimulus and test cases and drives them to design under test for verification. As manual test stimulus generation is involved, this form of verification allows specific functionality of the design to be tested. It is, however, unsuitable and inefficient to be used as the sole approach taken for designs with many functionalities to be tested (Singhal, 2015). On the other hand, **constrained random verification** generates user-defined constrained-random stimuli through automation. This form of verification provides broad functional coverage for complex designs and is often used with directed verifications to further provide coverage for corner cases (Singhal, 2015).

Coverage-driven verification identifies holes in the verification progress. It provides insight on features of the design that has yet to be sufficiently verified, tracking the functional coverage progress of the verification process. Coverage is an essential metric in design verification, whereby most functional verifications are guided by the metrics provided by coverage-driven verification (Singhal, 2015). **Assertion-based verification** is a useful form of verification approach for pinpointing the sources of error and significantly reducing debugging time. Assertions are executable specifications that control the execution of passive code segments, providing controllability and observability to the design. However, the implementation of assertion-based verification poses challenges in increased coding and debugging complexity and customization limitation (Tech Design Forum, n.d.).

Emulation-based verification verifies the gate-level model or RTL representation of the design mapped onto an FPGA through emulation. Proper emulation of the system allows for a high-performance system for verification. However, the long time required for setup and compilation can pose challenges towards the time-to-market aspect of the design flow (Singhal, 2016).

2.2.5 Summarized Review for Functional Verification

From the literature review performed for the requirements, technologies, and approaches of functional verification, a general idea for the verification of a RISC-V pipelined processor implementation is established.

The verification environment is coded using SystemVerilog to utilize object-oriented programming to incorporate reusability in design verification. Other capabilities such as the randomization capability in SystemVerilog will also play a considerable role in constrained-random stimuli generation for functional verification. From the discussion of requirements for functional verification, the basis of the functional verification is to verify and validate the features implemented by design under test. When constructing the verification environment, additional criteria such as reusability in the verification components, automation in the process execution, and standardization of coding guidelines will also be considered. From the verification technologies discussed, simulation-based verification has been deemed to be preferable for the RISC-V processor design with many functionalities to be tested. Among the functional verification approaches discussed, a combination of verification approaches that include constrained-random, directed, and coverage-driven verification will be utilized. Using a variety of verification approaches, a broader functional coverage can be achieved.

2.3 Universal Verification Methodology

Universal Verification Methodology (UVM) is a standardized methodology for verification that emphasizes reusability. UVM provides a standardized approach towards designing verification environment that promotes reusability and compatibility. UVM is widely used in the industry. It dramatically helps companies develop a modular, reusable and scalable testbench structure, encouraging the growth of a verification intellectual property (VIP) marketplace (Francesconi, Rodriguez and Julian, 2014). Through the standardized approach provided by the UVM, a layer of abstraction is integrated into the verification environment, whereby each verification component has a specific role. The layer of abstraction, in turn, helps the verification testbench to be more efficiently coded and maintainable. The following section provides a detailed analysis of the verification components within the UVM verification environment and insight into the verification flow.

2.3.1 UVM Testbench Architecture

UVM provides class libraries that allow for generic utilities such as configuration databases, transaction library modelling, and component hierarchy. The generic utilities provided to the user allow for creating a dynamic testbench structure. The building blocks allow for the rapid development of well-constructed, reusable verification components and test environments. In a typical UVM verification environment, the verification environment can be built using readily available UVM classes. The UVM classes components have a well-established standard communication infrastructure, allowing the verification components to send data packets between each other and work synchronously (Chip Verify, n.d.). The following figure showcases an example of a testbench architecture:

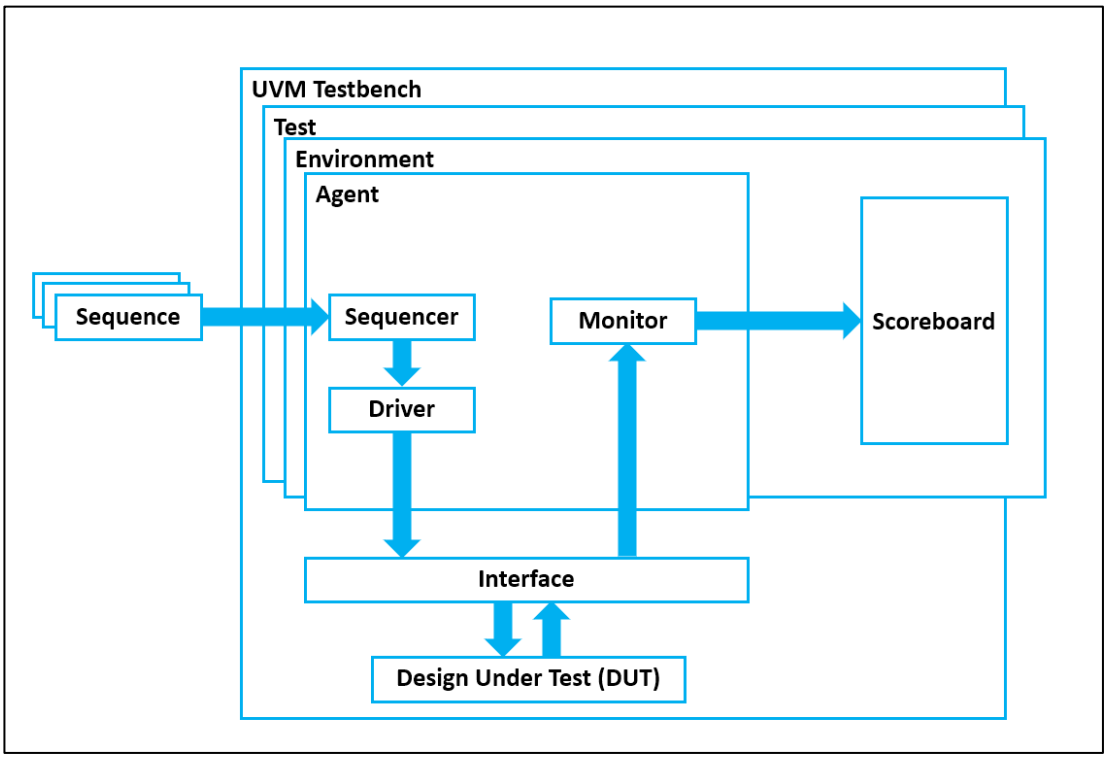


Figure 2.21 UVM Testbench Architecture.

2.3.2 UVM Component Class

UVM components are verification components that construct the complete hierarchical verification environment for UVM. These verification components are used for processing UVM objects, passing transactional data from one component to another. The hierarchical environment of the components also allows each component to be configured for specific features and different test scenarios. The various components of the UVM component class will be described in this section.

2.3.2.1 UVM Testbench

In UVM, a typical testbench is the root node, otherwise known as the top-level module. which serves as a static container that holds and instantiates all the verification components, interfaces, and the design under test. It is responsible for invoking the test to be performed on the design.

2.3.2.2 UVM Test

The test component is the top-level verification component in the component hierarchy. It instantiates and configures the environment component, the next level down in the component hierarchy. It is also responsible for initiating stimuli generation by starting virtual sequences. A test case is the specification of a verification test whereby the stimuli and conditions for the test run are set to test out specific design features under the specified condition. Through the configuration made to the environment, the test component can configure the environment component to generate different test cases, therefore exhibiting the aspect of reusability in the verification environment.

2.3.2.3 UVM Environment

The environment component is a hierarchical component that groups and instantiates interrelated verification components such as the agent, scoreboard, and other components. It has several configuration parameters set by the test component, allowing the environment to be configured for different test scenarios.

2.3.2.4 UVM Agent

The agent component is another hierarchical component that encapsulates the verification components dealing with a specific design under test interface. These components include a sequencer, a driver, and a monitor. The verification components encapsulated are instantiated and interconnected through transaction-level modelling interfaces. Like the environment component, the agent component also has configuration options to enable or disable features or even set the agent component as an active driving component or a passive monitoring component.

2.3.2.5 UVM Sequencer

The sequencer is a verification component that generates sequence items as data transactions and sends them to the driver component for further execution. Upon receiving the request for sequence items made by the driver component, the sequencer initializes sequence item generation and sends them upon finishing the item generation.

2.3.2.6 UVM Driver

The driver is an active component within the verification environment that actively drives the sequence items obtained from the sequencer to the design under test via the interface. The driver pulls sequence items downstream on a test run by sending a request to the sequencer component to generate sequence items. The sequence items generated and received by the driver component are further mapped to signal level formats compatible with the interface to be driven to the design under test.

2.3.2.7 UVM Monitor

Monitor captures information from the design under test from the interface and converts the captured signals to transaction level sequence items. These transactions containing the captured information are then sent to other components such as the scoreboard for functionality checking. It can also perform internal processing such as coverage collection on the data received.

2.3.2.8 UVM Scoreboard

The scoreboard component is the verification component within the testbench that performs the functionality checking. From the data transactions received from the monitor component via an analysis port, the actual values from the design under test and the expected values are compared. One methodology often used for generating expected values to be compared is through the use of a reference model. The input stimuli to be driven to the design under test are also sent to the reference model. The obtained result for the given stimuli by both reference model and design under test can then be checked for functionality correctness.

2.3.3 UVM Transaction Base Class

UVM transaction class contains objects that represent data within the verification environment.

2.3.3.1 UVM Sequence Items

Sequence items are the information or data transactions passed between verification components. They may also include the stimuli to be driven to or monitored from the design under test. On an abstract level, sequence items can be viewed as the communication data between the components in the UVM environment.

2.3.3.2 UVM Sequence

Sequences are a set of sequence items often initiated by the sequencer component to be driven to the driver component. The set of sequence items are assembled to form a stimuli for the verification process.

Chapter 3

METHODOLOGY

3.1 Verification Flow

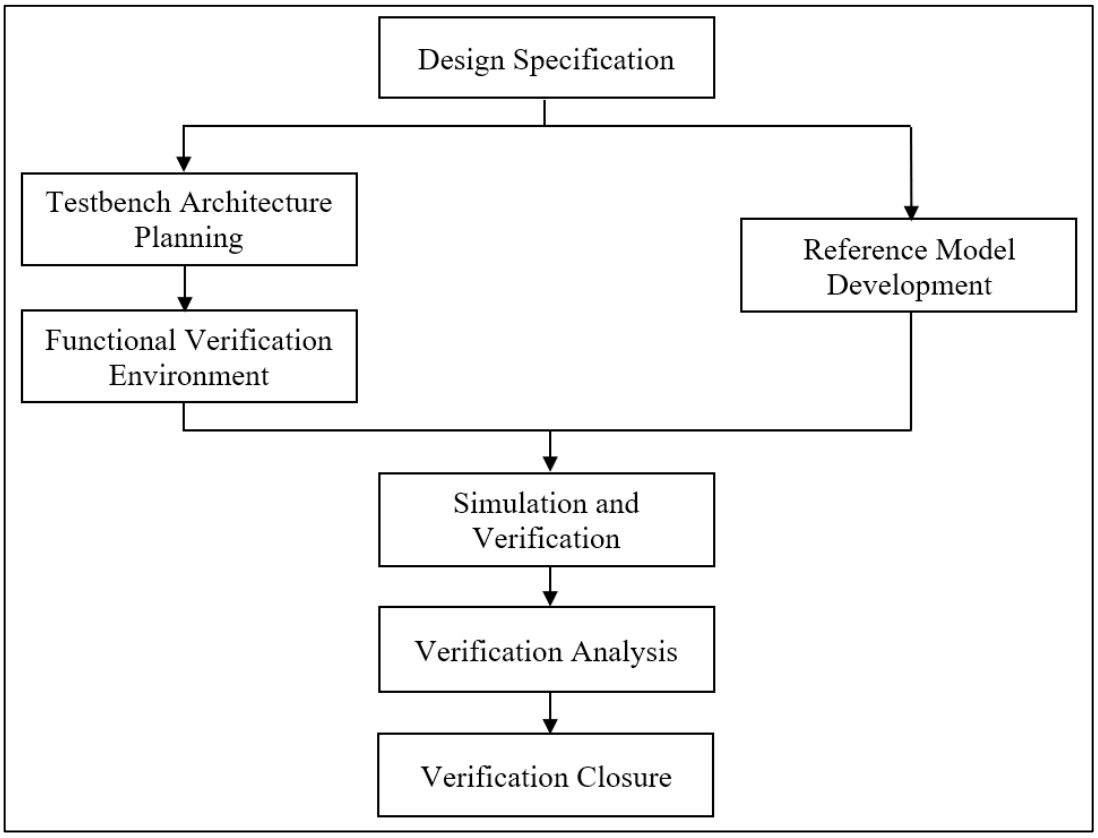


Figure 3.1: Design Verification Flow.

A project flow has been created to provide a systematic approach towards the design verification project. A well-planned project flow can give complete coverage and encapsulate the tasks needed to be done.

For design verification, the first stage is to **capture the design specification**. Reference documents such as the textbook “Computer Organization and Design RISC-V Edition” and RISC-V's instruction set architecture manual are studied to understand the design's functioning properly. Documents and articles on UVM and design verification are also read to generate ideas on how the design verification should be done.

The project flow then separates into two parallel paths upon document study and research. One of the paths involves verification environment development and the other consists of reference model development. On the verification environment development path, the **testbench architecture planning stage** is performed to define the verification testbench architecture. As the project utilizes UVM for verification, a UVM testbench architecture is used for the design verification. The UVM components that constructs the UVM testbench architecture planned are then coded to **construct the functional verification environment**. On the other hand, **reference model development** is to develop a reference model based on the reference documents studied. The reference model provides the expected data for functionality checking of the design under test. Therefore, extensive verification needs to be performed to ensure its functionality correctness.

When both reference model and verification environment are constructed and verified, the project flow proceeds to the **simulation and verification** stage which comprises of the main design verification work. Simulation-based verification is performed to check the functionality correctness of the design under test. Bugs encountered during simulations are debugged, and the simulation runs are reiterated. The cycle of simulation and debugging are repeated until the design is bug-free.

Following the simulation-based verification stage, **verification analysis** is performed to determine the sufficiency and thoroughness of the design verification. Through a well-planned functional coverage plan, the functional coverage analysis

helps identify the design functionalities verified and help provide insight on the overall design verification progress. When the functional verification of the design is sufficiently performed, **verification closure** can be performed to file the necessary documentation for future reference purposes.

3.1.1 Design Specification

Design verification requires extensive knowledge and understanding of the functionalities of the design to be verified. This project's subject to be verified is a RISC-V base instruction set architecture pipelined processor. As such, RISC-V base instruction set architecture and pipeline implementation need to be thoroughly studied and understood. The design principles in the reference documents utilized for the processor design studied must be aligned with the design principles used by the design team to ensure a mutual understanding of the architecture of the design. Discussion also needs to be held with the design team to determine the design functionalities to be implemented, providing information on design functionalities that require verification.

3.1.2 Testbench Architecture Planning

A well-planned testbench architecture can help create a reusable testbench architecture that caters to various functional verification scenarios and approaches. Providing configurability to the testbench components allows the aforementioned functional verification approaches such as constrained-random verification and directed verification to be implemented using the same testbench. As a reference model is to be utilized for the functionality checking, the UVM testbench architecture shown in **Figure 2.21** has been modified and is shown on the following page. The revised implementation allows synchronized operation between the reference model and the design under test. The synchronized operation allows the UVM monitor component to

capture the synchronized output and send it to the UVM scoreboard component for further analysis.

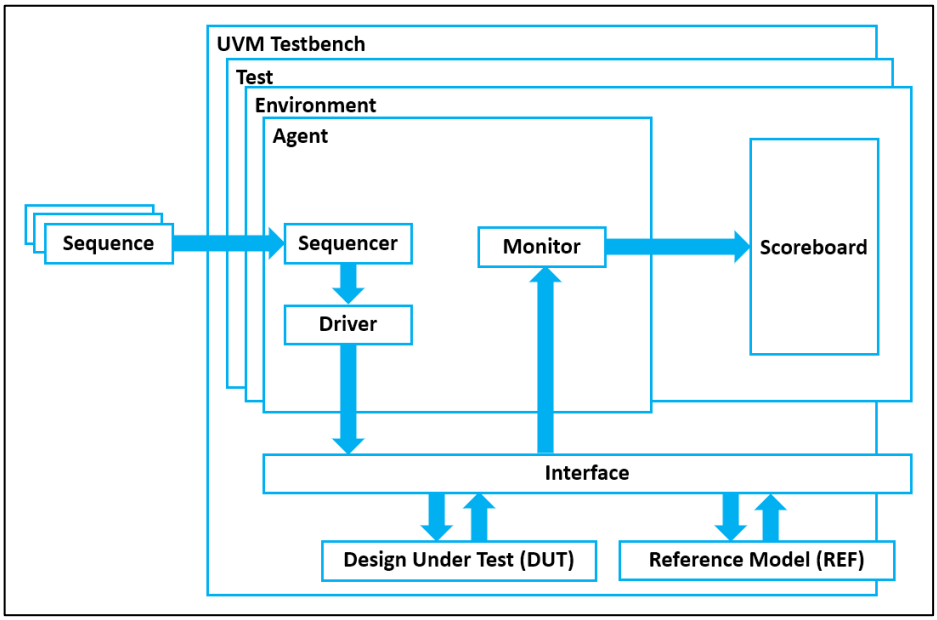


Figure 3.2: Revised Verification Testbench Architecture.

3.1.3 Functional Verification Environment

The UVM components that construct the functional environment and the testbench architecture are developed. The verification components developed are derived from the UVM standardized class library and further defined based on design verification requirements. ModelSim has been selected as the platform for the code designing, simulation, and verification process due to its capability to handle system simulations and produce waveforms for detailed debugging.

3.1.4 Reference Model Development

Simulation-based verification for a complex system usually deploys a reference model. A reference model has been developed based on the reference documents on RISC-V

base instruction set architecture studied. The reference model design is done based on the functionalities implemented by the design team to allow for synchronized operation between the reference model and the design under test. The functionality correctness of the reference model developed is verified using the verification testbench developed. The verifications provide insight into improvement opportunities on the functional verification environment and possible features to be added. The reference model development is completed when all agreed design functionalities are implemented and verified.

3.1.5 Simulation and Verification

The simulation-based verification is executed when the design prototype, reference model, and functional verification environment are constructed. Bugs identified in the verification models or discrepancies identified in the models are debugged. Communication is established with the design team to discuss the bugs identified. The debugged design provided by the design team is sent for regression test to ensure the debug fix does not introduce new bugs to the system. The process of simulation and debugging is repeated until the design is bug-free.

3.1.6 Verification Analysis

A functional coverage plan written for the design verification is utilized for functional coverage analysis. Functional coverage analysis provides a coverage metric that offers insight into design verification progress. Additional test cases are generated for the verification process if insufficient testing is performed. Directed verification is also utilized to verify corner cases. If additional features are added to the verification environment or the design, the reference model and testbench development, simulation, and verification stages are repeated. When sufficient verification is performed on the design, the design verification proceeds to the verification closure stage.

3.1.7 Verification Closure

The final stage of the design verification flow involves the documentation of the design verification, such as the functionalities tested and reports on the overall verification coverage to determine the robustness of the verification performed.

3.2 Project Timeline

Gantt charts have been created to schedule the tasks to be carried out for the project. The task scheduling allows the project to progress without unwanted delays and to complete on time. The following Gantt charts show the planning for the first phase and second phase of the project:

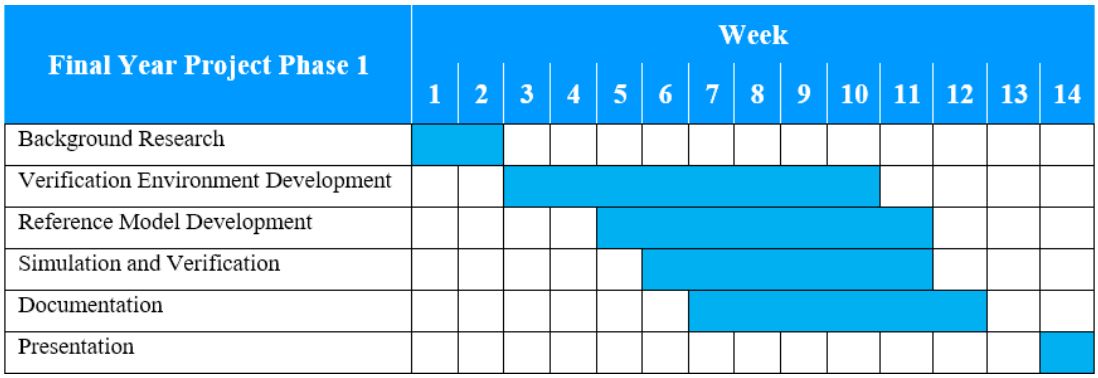


Figure 3.3: Gantt Chart for Phase 1 of Final Year Project.

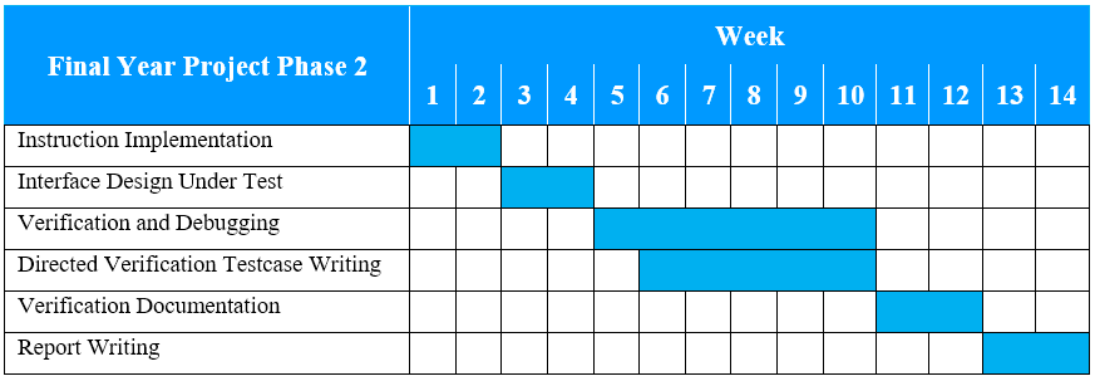


Figure 3.4: Gantt Chart for Phase 2 of Final Year Project.

3.3 Verification Simulation Flow

This section provides a detailed explanation on the simulation-based verification flow.

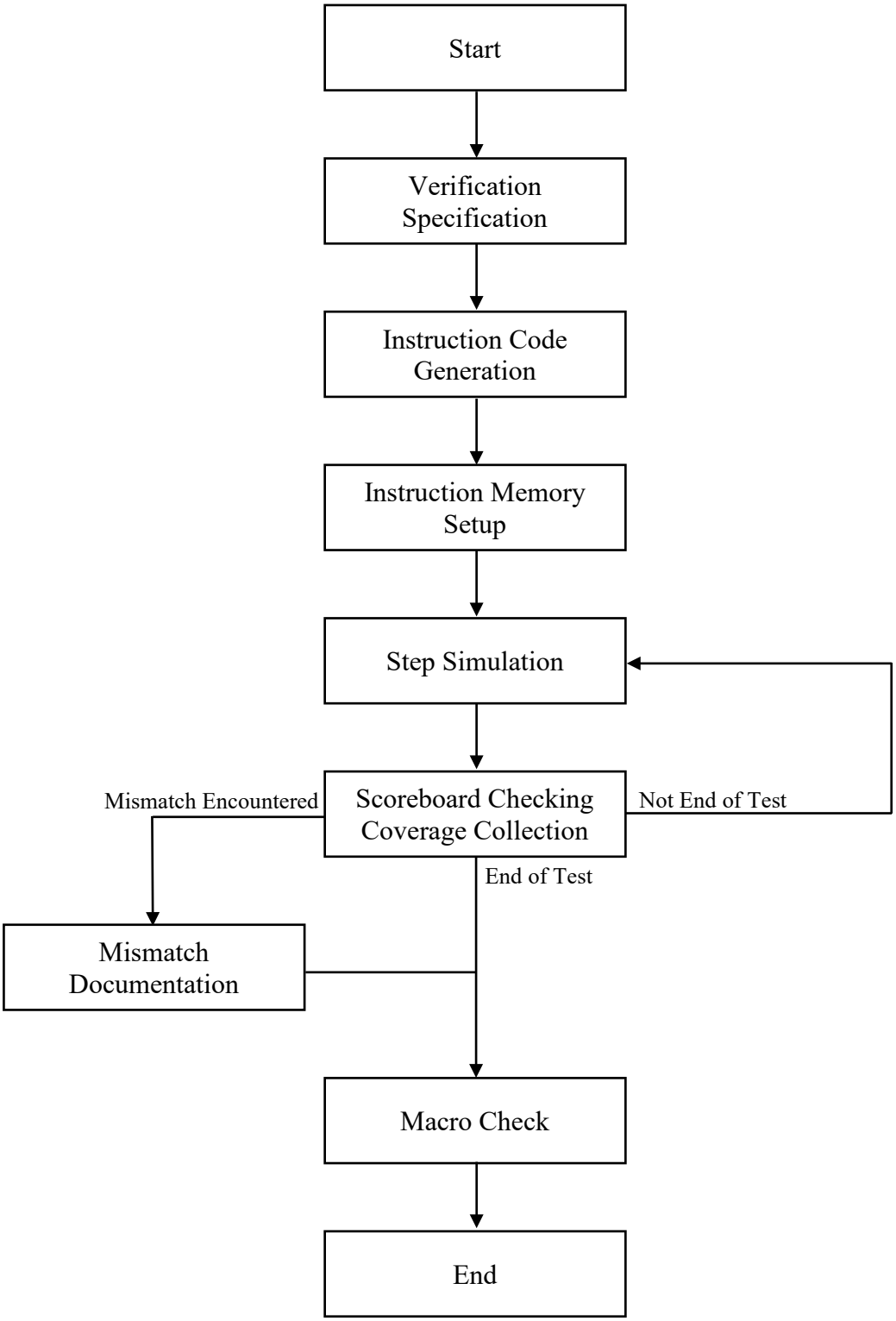


Figure 3.5: Verification Simulation Flow.

3.3.1 Verification Specification

The verification environment alongside the reference model and the design under test is first compiled and checked for syntax errors. Through the use of command line arguments, specifications can be provided to configure the testbench components and the nature of the test case, allowing the verification environment to be reused for various test scenarios. The following table provides information on the specifications that can be inputted to modify the verification test:

Table 3.1: Test Arguments for Verification Specification.

Test Arguments	Description
+TESTLOG	Configures the verification environment to generate test log
+SEED=<val>	Specifies the test seed to be tested. The verification environment clones pre-existing test case or creates the test case using instruction code generation and stores the created test case onto repository. If unspecified, a default seed number of "0000" is used.
+INSTR=<val>	Specifies the amount of instructions to be generated for a given test case. If unspecified, a default amount of 500 instructions is generated.
+FORCE_GEN	Configures verification environment to generate new instruction set for specified test seeds, renewing pre-existing test case in the repository.
+INSTR_TYPE=<val>	Specifies the instruction types (R-type, SB-type) to be generated. If unspecified, all instruction types will be generated by default.
+BATCH_TEST	Configures the verification environment to execute multiple test cases specified in a seed file.
+CONT	Configures the verification environment to run all the test cases provided in a batch test. Instead of ending the simulation upon encountering mismatch, the mismatch and the test case failed are recorded onto a file named "FAILED.txt" for further debug.
+BATCH_SEED=<val>	Specifies the seed file containing the test cases for multiple test case run. If unspecified, a default file named "SEED.txt" is accessed.
+DIRECTED_TEST	Configures the verification environment to perform directed verification. The test instructions written in a file named "TEST.txt" are translated to machine language and driven to the test models for simulation,
+SKIP_MACRO_CHECK	Bypasses macro checking of stall and flush conditions.
+MACRO_OVERWRITE	Configures the verification environment to renew the historical values of stall and flush conditions executed for a test case.

3.3.2 Instruction Code Generation

Upon starting the simulation, the testbench first instantiates all the verification components and models. The models are first applied with a reset signal, halting them until the verification environment finishes setting up the test case. While the reset signal is asserted, the driver component sends requests for sequence generation, requesting for generation of sequence items which are the instruction codes to be sent to the models. Upon completion of instruction code generation, the instruction code transactions are stored externally onto files named "ASM.txt" and "PROM.txt". A data transaction signalling the completion of instruction generation will also be sent to the driver to inform the completion of instruction generation.

The methodology utilized ensures that the full program instructions are generated before the models start operating. In contrast to the traditional transaction-to-transaction simulation, the availability of the entire program allows the pipelined processor to execute jump or branch instructions without issue. In the case of jump or branch instructions, the instruction code at the target address specified by the jump or branch instruction needs to be available to the instruction memory on the next clock cycle. In traditional transaction to transaction simulation, instruction codes are generated and driven to the models upon prompt, which may result in a bad test case when the models attempt to access an instruction code at an instruction address that is yet to be generated by the testbench.

The instruction code generation function represents the constrained-random verification for the design verification. The instruction codes are randomized but are constrained such that they remain as valid instruction codes that the processor models can process. Among the specifications that can be provided to the system, the “+INSTR_TYPE=” argument specifies the instruction type to be generated. When specific instruction types are provided, the corresponding opcode for the instruction type is added to a pool from which the instruction generator will randomly select an opcode. If no instruction type is specified, all instruction types will be added to the pool, allowing the generator to generate any valid instruction type. The following code segment shows how the randomization of specified instruction type is performed:

```
for(int pointer = 0; pointer < instr_type.len(); pointer ++) begin
        case(instr_type[pointer])
        "R"     : begin
                        possible_opcode[randomizer] = `R_OPCODE;
                        randomizer ++;
                  end
        "I"     : begin
                        possible_opcode[randomizer] = `I_OPCODE;
                        randomizer ++;
                  end
        "L"     : begin
                        possible_opcode[randomizer] = `LOAD_OPCODE;
                        randomizer ++;
                  end
        "J"     : begin
                        possible_opcode[randomizer] = `JALR_OPCODE;
                        randomizer ++;
                  end
        "U"     : begin
                        case(instr_type[pointer + 1])
                        "J"     : begin
                                        possible_opcode[randomizer] = `J_OPCODE;
                                        randomizer ++;
                                        pointer ++;
                                  end
                        default : begin
                                        possible_opcode[randomizer] = `U_OPCODE;
                                        randomizer ++;
                                  end
                        endcase
                  end
        "S"     : begin
                        case(instr_type[pointer + 1])
                        "B"     : begin
                                        possible_opcode[randomizer] = `SB_OPCODE;
                                        randomizer ++;
                                        pointer ++;
                                  end
                        default : begin
                                        possible_opcode[randomizer] = `S_OPCODE;
                                        randomizer ++;
                                  end
                        endcase
                  end
        "_"     : continue;
        default : `uvm_fatal("ERROR","SEQ_ITEM_ERROR: invalid INSTR_TYPE specified")
        endcase
end
opcode = possible_opcode[$urandom_range((randomizer-1),0)];
```

Figure 3.6: Randomization of specified Instruction Type.

For instruction fields requiring fewer constraints, the instruction field is specified as a randomized variable, allowing a randomized value within the specified range to be assigned to the instruction field. The following figure shows the declaration of randomized instruction fields and randomizer variables:

```
rand bit [`FUNCT3_WIDTH-1:0]    funct3          = $urandom_range(7,0);
rand bit [`FUNCT7_WIDTH-1:0]    funct7          = $urandom_range(127,0);
rand bit [`REG_ADDR_WIDTH-1:0]  rs1             = $urandom_range(31,0);
rand bit [`REG_ADDR_WIDTH-1:0]  rs2             = $urandom_range(31,0);
rand bit [`REG_ADDR_WIDTH-1:0]  rd              = $urandom_range(31,1);
```

Figure 3.7: Declaration of Randomized Instruction Fields.

For the specification of funct3 and funct7 fields for the opcode generated from the valid pool, specific values are assigned through **randcase**, a randomized case statement that randomly selects one of its statements based on the probability assigned. The following shows how the funct3 and funct7 fields are selected based on the opcode generated and the randcase statement:

```
case(opcode)
`R_OPCODE       : begin
                        randcase
                        1: funct3 = `ADD_FUNCT3;
                        1: funct3 = `SLL_FUNCT3;
                        1: funct3 = `SLT_FUNCT3;
                        1: funct3 = `SLTU_FUNCT3;
                        1: funct3 = `XOR_FUNCT3;
                        1: funct3 = `SR_FUNCT3;
                        1: funct3 = `OR_FUNCT3;
                        1: funct3 = `AND_FUNCT3;
                        endcase
                  end
`LOAD_OPCODE    : begin
                        randcase
                        1: funct3 = `LB_FUNCT3;
                        1: funct3 = `LH_FUNCT3;
                        1: funct3 = `LW_FUNCT3;
                        1: funct3 = `LBU_FUNCT3;
                        1: funct3 = `LHU_FUNCT3;
                        endcase
                  end
```

Figure 3.8: Assigning valid funct3 field using randcase.

```
case(opcode)
`R_OPCODE       : begin
                        case (funct3)
                        `ADD_FUNCT3,
                        `SR_FUNCT3: begin
                                        randcase
                                        1: funct7 = `DEFAULT_FUNCT7;
                                        1: funct7 = `ALT_FUNCT7;
                                        endcase
                                    end
                        default:        funct7 = `DEFAULT_FUNCT7;
                        endcase
                  end
`I_OPCODE       : begin
                        case (funct3)
                        `SLL_FUNCT3:    funct7 = `DEFAULT_FUNCT7;
                        `SR_FUNCT3:     begin
                                        randcase
                                        1: funct7 = `DEFAULT_FUNCT7;
                                        1: funct7 = `ALT_FUNCT7;
                                        endcase
                                    end
                        endcase
                  end
endcase
```

Figure 3.9: Assigning valid funct7 field using randcase.

Lastly, the instruction code is constructed using the instruction fields generated by concatenating the appropriate fields in the correct order based on the instruction type. An additional constraint is imposed for branch instructions to ensure a constrained branch range for better test case quality. The following code segment shows how the instruction code is formed from the instruction fields generated:

```
// Branch Address Constraint
case(opcode)
`J_OPCODE:    begin
                {instr_code[31],instr_code[19:12],instr_code[20],instr_code[30:21]} =
                $urandom_range(8,4) * `INST_ADDR_SUM;
                rd = $urandom_range(31,0);
                instr_code[11:0] = {rd, opcode};
              end
`SB_OPCODE:   begin
                {instr_code[31],instr_code[7],instr_code[30:25],instr_code[11:8]} =
                $urandom_range(8,4) * `INST_ADDR_SUM;
                instr_code[24:12] = {rs2,rs1,funct3};
                instr_code[6:0] = opcode;
              end
`JALR_OPCODE: begin
                immediate_value = $urandom_range(current_instr_number + 8, current_instr_number + 4) * `INST_ADDR_SUM;
                rs1 = 0;
                instr_code = {immediate_value[11:0],rs1,funct3,rd,opcode};
              end
default:      instr_code = {funct7,rs2,rs1,funct3,rd,opcode};
endcase
```

Figure 3.10: Concatenation of Instruction Fields into Instruction Codes.

The constructed instruction codes are then stored into ASM.txt and PROM.txt, as shown in the code segment below:

```
// Output instruction code and address
fh = $fopen("ASM.txt", "a+");
$fdisplay(fh, "%8h %8h", instr_addr, instr_code);
$fclose(fh);
// Output instruction code in bytes
fh = $fopen("PROM.txt","a+");
$fdisplay(fh, "%2h %2h %2h %2h", instr_code[31:24], instr_code[23:16], instr_code[15:8], instr_code[7:0]);
$fclose(fh);
```

Figure 3.11: Storing of Instruction Code and Address onto ASM.txt and PROM.txt.

This instruction generation process is reiterated until the specified number of generated instruction codes is achieved. The number of instruction codes to be generated can be specified in the command line through the argument “+INSTR=”. The following figures show the instruction codes generated alongside their corresponding instruction address in “ASM.txt” and the segmented instruction codes in “PROM.txt”:

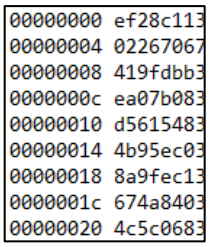


Figure 3.12: Instruction Address and Instruction Code on ASM.txt.

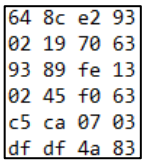


Figure 3.13: Segmented Instruction Code on PROM.txt.

As instruction code generation for test cases may consume a lot of simulation time, test case simulations can become tedious if instruction code generation needs to be executed each time. Therefore, the text files containing the instruction codes generated are cloned onto the test repository. An additional feature whereby the system checks through the test repository for existing test cases is implemented. The verification environment checks through the test repository for pre-existing test cases on repeated test case simulation. If pre-existing test case is detected, the test case will be cloned, and instruction code generation will be bypassed, saving a lot of simulation time. If a pre-existing test case is not found, instruction code generation will then be executed, and the files generated will be cloned to the test repository. This added feature can also be bypassed, forcing the verification environment to execute the instruction code generation function and update the test repository through the command line argument “+FORCE_GEN”.

3.3.3 Directed Test Assembly Language Translation

A feature to translate manually written test cases from assembly language to machine language is introduced to the verification environment to incorporate directed verification into the environment. When the command line argument "+DIRECTED_TEST" is provided, instruction code written in assembly language on a text file named "TEST.txt" will be translated into machine language instruction codes. The following figure shows a sample program written in assembly language:

```
addi    x3,     x1,     3000
add     x2,     x1,     x3
and     x1,     x2,     x3
lui     x4,     1
```

Figure 3.14: Assembly Language Instruction Codes in TEST.txt.

The following code segment showcases how the file is accessed for the instruction operation and compares it with a list of instructions defined. If a correct instruction operation is matched, the corresponding values for opcode, funct3, funct7 fields are assigned, and the instruction type is defined for encoding and operand processing purposes.

```
fh = $fopen("TEST.txt","r");
if(fh == 0)
        `uvm_fatal("ERROR","Unable to access TEST.txt");
instr_addr = 0;
while(!$feof(fh))       begin
        shift_type = 0;
        store_type = 0;
        load_type = 0;
        dv_instr_type = "";
        assembly_code = "";
        code = $fscanf(fh,"%s",assembly_code);
        case(assembly_code)
        "add":  begin
                        opcode = `R_OPCODE;
                        funct3 = `ADD_FUNCT3;
                        funct7 = `DEFAULT_FUNCT7;
                        dv_instr_type = "R";
                end
        "sub":  begin
                        opcode = `R_OPCODE;
                        funct3 = `ADD_FUNCT3;
                        funct7 = `ALT_FUNCT7;
                        dv_instr_type = "R";
                end
        "sll":  begin
                        opcode = `R_OPCODE;
                        funct3 = `SLL_FUNCT3;
                        funct7 = `DEFAULT_FUNCT7;
                        dv_instr_type = "R";
                end
```

Figure 3.15: Translation of Assembly Code Instruction Operation.

Following the identification of the instruction operation, the first operand is subsequently obtained and processed for its information before the processed information is placed into the correct field based on the instruction type currently being executed. The following code segment shows how the first, second, and third operands are accessed, processed, and placed into the correct instruction fields:

```
code = $fscanf(fh,"%s",operand);
min_range = -1;
for(int i = 0; i < operand.len(); i++)  begin
        case(operand[i])
        "x":    continue;
        " ":    continue;
        ",":    max_range = i - 1;
        default:begin
                        if(min_range < 0)
                                min_range = i;
                end
        endcase
end
operand = operand.substr(min_range,max_range);
if(dv_instr_type  == "S")
        rs2 = operand.atoi();
else if(dv_instr_type == "B")
        rs1 = operand.atoi();
else
        rd = operand.atoi();
```

Figure 3.16: Translation of Assembly Code First Operand.

```
code = $fscanf(fh,"%s",operand);
if(dv_instr_type != "J" && dv_instr_type != "U")        begin
        min_range = -1;
        for(int i = 0; i < operand.len(); i++)  begin
                case(operand[i])
                "x":    continue;
                " ":    continue;
                "(":    begin
                                max_range = i - 1;
                                hold = operand.substr(min_range,max_range);
                                immediate_value [11:0] = hold.atoi();
                                min_range = -1;
                        end
                ")",
                ",":    max_range = i - 1;
                default:begin
                                if(min_range < 0)
                                        min_range = i;
                        end
                endcase
        end
        operand = operand.substr(min_range,max_range);
        if(dv_instr_type == "B")
                rs2 = operand.atoi();
        else
                rs1 = operand.atoi();
end
else
        immediate_value [20:1] = operand.atoi();
```

Figure 3.17: Translation of Assembly Code Second Operand.

```
if(load_type == 0 && store_type == 0 && dv_instr_type != "J" && dv_instr_type != "U")   begin
        code = $fscanf(fh,"%s",operand);
        min_range = -1;
        max_range = operand.len() - 1;
        for(int i = 0; i < operand.len(); i++)  begin
                case(operand[i])
                "x",
                " ":    continue;
                default:begin
                                if(min_range < 0)
                                        min_range = i;
                        end
                endcase
        end
        operand = operand.substr(min_range,max_range);
        case(dv_instr_type)
        "R":    rs2 = operand.atoi();
        "I":    begin
                        if(shift_type == 1)
                                immediate_value [4:0] = operand.atoi();
                        else
                                immediate_value [11:0] = operand.atoi();
                end
        "B":    immediate_value [13:1] = operand.atoi();
        endcase
end
```

Figure 3.18: Translation of Assembly Code Third Operand.

After performing translation for all the operands, the information obtained is concatenated into a valid instruction and stored externally onto “ASM.txt” and “PROM.txt”. The translation process is repeated for all the instructions contained in the test file. The following figure shows the concatenation of the instructions:

```
case(dv_instr_type)
"R":    instr_code = {funct7,rs2,rs1,funct3,rd,opcode};
"I":    instr_code = {immediate_value[11:0],rs1,funct3,rd,opcode};
"S":    instr_code = {immediate_value[11:5],rs2,rs1,funct3,
                      immediate_value[4:0],opcode};
"B":    instr_code = {immediate_value[12],immediate_value[10:5],
                      rs2,rs1,funct3,immediate_value[4:1],immediate_value[11],opcode};
"J":    instr_code = {immediate_value[20],immediate_value[10:1],immediate_value[11],
                      immediate_value[19:12],rd,opcode};
"U":    instr_code = {immediate_value[19:0],rd,opcode};
endcase
f_out = $fopen("ASM.txt", "a+");
$fdisplay(f_out, "%8h %8h", instr_addr, instr_code);
$fclose(f_out);
f_out = $fopen("PROM.txt","a+");
$fdisplay(f_out, "%2h %2h %2h %2h", instr_code[31:24], instr_code[23:16], instr_code[15:8], instr_code[7:0]);
$fclose(f_out);
instr_addr = instr_addr + 4;
```

Figure 3.19: Concatenation of Translated Information into Instruction Code.

3.3.4 Instruction Memory Setup

Upon completing generation of instructions, cloning of test case, or translation of assembly language, a transaction signal is sent to the UVM driver component signifying the instruction codes are ready to be loaded onto the instruction memory. The driver component then loads the instruction code from PROM.txt onto the instruction memory of the models through the top testbench module. The following code segment shows how instruction codes are loaded onto a dynamic memory structure from the driver component:

```
// Load test program onto processor instruction memory
virtual task load_program(seq_item transaction);
        @(interface_instance.cb);
                if(transaction.instr_gen_completion)    begin
                        $display("Loading Program into ROM");
                        $readmemh("PROM.txt",testbench.riscv.rom);
                        $readmemh("PROM.txt",testbench.riscv_ref.rom);
                        #100;
                        $display("Program Successfully Loaded");
                        transaction.instr_gen_completion = 0;
                end
endtask
```

Figure 3.20: Loading of Instruction Code from PROM.txt initiated by UVM Driver component.

Based on standard memory technology, each memory register stores a byte of data. For a 32-bit RISC-V processor, each instruction is 32 bits (4 bytes) long and is stored in 4 memory registers. The following figure shows the instruction codes stored on the memory structure in the instruction memory unit:

00000000	bb 80 81 93
00000004	00 30 81 33
00000008	00 31 70 b3
0000000c	00 00 12 37
00000010	40 12 00 b3
00000014	00 c2 52 93

Figure 3.21: Instructions Codes stored on Instruction Memory.

After loading the instruction codes onto the models, the test run is initiated by the test component by releasing the reset signal.

3.3.5 Step Simulation

The simulation runs in steps, allowing the UVM monitor to capture the outputs of the models at every clock cycle and send the transactional data received to the scoreboard and coverage component for functionality checking and coverage checking respectively. The synchronous functioning of the pipeline models allows outputs to be compared against each other on every cycle. The following code segment shows the transaction of model outputs to the interface for design verification:

```
// Drive signals to the testbench for further verification
always @(posedge interface_instance.clk) begin: interface_block
                interface_instance.ref_stall <= stall;
                interface_instance.ref_flush <= flush;
                interface_instance.ref_jump_link <= idex_ctl_op[`JUMP_LINK];
                if(!stall) begin
                        interface_instance.ref_pc <= ifid_instr_addr;
                        interface_instance.ref_instr <= ifid_instr_code;
                end
                interface_instance.ref_reg_read_addr_1 <= idex_reg_addr_1;
                interface_instance.ref_reg_read_addr_2 <= idex_reg_addr_2;
                case(fwrd_mux_1)
                `FWRD_ALU:      interface_instance.ref_reg_read_data_1 <= exmem_alu_output;
                `FWRD_MEM:      begin
                        if(memwb_ctl_op[`MEM_TO_REG])
                                interface_instance.ref_reg_read_data_1 <= memwb_mem_data;
                        else
                                interface_instance.ref_reg_read_data_1 <= memwb_alu_data;
                        end
                `NO_FWRD:       interface_instance.ref_reg_read_data_1 <= idex_reg_data_1;
                endcase
                case(fwrd_mux_2)
                `FWRD_ALU:      interface_instance.ref_reg_read_data_2 <= exmem_alu_output;
                `FWRD_MEM:      begin
                        if(memwb_ctl_op[`MEM_TO_REG])
                                interface_instance.ref_reg_read_data_2 <= memwb_mem_data;
                        else
                                interface_instance.ref_reg_read_data_2 <= memwb_alu_data;
                        end
                `NO_FWRD:       interface_instance.ref_reg_read_data_2 <= idex_reg_data_2;
                endcase
                interface_instance.ref_imm_val <= idex_imm_val;
                interface_instance.ref_alu_output <= exmem_alu_output;
                interface_instance.ref_alu_zero <= exmem_zero;
                interface_instance.ref_ctl_op <= exmem_ctl_op;
                interface_instance.ref_reg_write_addr <= memwb_write_addr;
                interface_instance.ref_mem_addr <= memwb_mem_addr;
                interface_instance.ref_mem_write_data <= memwb_mem_write_data;
                interface_instance.ref_mem_write <= memwb_ctl_op[`MEM_WRITE];
                interface_instance.ref_mem_read <= memwb_ctl_op[`MEM_READ];
                interface_instance.ref_reg_write <= memwb_ctl_op[`REG_WRITE];
                interface_instance.ref_reg_write_data <= memwb_write_data;

                // Delayed alignment instruction executions for coverage checking
                interface_instance.ref_ID_instr <= idex_instr_code;
                interface_instance.ref_EX_instr <= exmem_instr_code;
                interface_instance.ref_EX_pc <= exmem_instr_addr;
end: interface_block
```

Figure 3.22: Reference Model to Interface Data Transaction.

The UVM monitor component then captures the information sent to the interface. The UVM monitor component converts the signals captured to transactional data and exports them to the scoreboard and coverage components. The following code segment shows the UVM monitor component capturing the signals from the models and exporting them to other components via an analysis port:

```
virtual task run_phase(uvm_phase phase);
        super.run_phase(phase);
        forever begin
        @(interface_instance.cb);
                if (interface_instance.monitor_start) begin
                        seq_item transaction = seq_item::type_id::create("transaction");
                        // DUT Transactions
                        transaction.dut_instr           = interface_instance.dut_instr;
                        transaction.dut_pc              = interface_instance.dut_pc;
                        transaction.dut_reg_read_addr_1 = interface_instance.dut_reg_read_addr_1;
                        transaction.dut_reg_read_addr_2 = interface_instance.dut_reg_read_addr_2;
                        transaction.dut_reg_read_data_1 = interface_instance.dut_reg_read_data_1;
                        transaction.dut_reg_read_data_2 = interface_instance.dut_reg_read_data_2;
                        transaction.dut_imm_val         = interface_instance.dut_imm_val;
                        transaction.dut_alu_output      = interface_instance.dut_alu_output;
                        transaction.dut_alu_zero        = interface_instance.dut_alu_zero;
                        transaction.dut_ctl_op          = interface_instance.dut_ctl_op;
                        transaction.dut_reg_write_addr  = interface_instance.dut_reg_write_addr;
                        transaction.dut_reg_write_data  = interface_instance.dut_reg_write_data;
                        transaction.dut_mem_write_data  = interface_instance.dut_mem_write_data;
                        transaction.dut_mem_addr        = interface_instance.dut_mem_addr;
                        // REF Transactions
                        transaction.instr_addr          = interface_instance.instr_addr;
                        transaction.ref_instr           = interface_instance.ref_instr;
                        transaction.ref_pc              = interface_instance.ref_pc;
                        transaction.ref_reg_read_addr_1 = interface_instance.ref_reg_read_addr_1;
                        transaction.ref_reg_read_addr_2 = interface_instance.ref_reg_read_addr_2;
                        transaction.ref_reg_read_data_1 = interface_instance.ref_reg_read_data_1;
                        transaction.ref_reg_read_data_2 = interface_instance.ref_reg_read_data_2;
                        transaction.ref_imm_val         = interface_instance.ref_imm_val;
                        transaction.ref_alu_output      = interface_instance.ref_alu_output;
                        transaction.ref_alu_zero        = interface_instance.ref_alu_zero;
                        transaction.ref_ctl_op          = interface_instance.ref_ctl_op;
                        transaction.ref_reg_write_addr  = interface_instance.ref_reg_write_addr;
                        transaction.ref_reg_write_data  = interface_instance.ref_reg_write_data;
                        transaction.ref_reg_write       = interface_instance.ref_reg_write;
                        transaction.ref_mem_write_data  = interface_instance.ref_mem_write_data;
                        transaction.ref_mem_addr        = interface_instance.ref_mem_addr;
                        transaction.ref_mem_read        = interface_instance.ref_mem_read;
                        transaction.ref_mem_write       = interface_instance.ref_mem_write;
                        transaction.ref_stall           = interface_instance.ref_stall;
                        transaction.ref_flush           = interface_instance.ref_flush;
                        transaction.ref_jump_link       = interface_instance.ref_jump_link;
                        transaction.ref_ID_instr        = interface_instance.ref_ID_instr;
                        transaction.ref_EX_instr        = interface_instance.ref_EX_instr;
                        transaction.ref_EX_pc           = interface_instance.ref_EX_pc;
                        transaction.end_of_test         = interface_instance.end_of_test;

                        // Write to analysis port (scoreboard and coverage checker)
                        analysis_port.write(transaction);
```

Figure 3.23: Monitor Data Transaction Relaying.

3.3.6 Scoreboard Checking

The UVM scoreboard component performs the primary functionality correctness checking for the models in the verification environment. Signals obtained from the reference model and design under test are compared for discrepancies. The reference model undergoes partial self-checking testing before the model result comparison. Specific outputs such as the source register address, ALU output, and destination register address and data are checked. Self-checking ensures the functional correctness of the reference model, which provides more confidence in the comparison results produced. The following code segment shows the self-checking mechanism implemented:

```
if(transaction.ref_ID_instr[`RS1_ADDR_HI:`RS1_ADDR_LO] != transaction.ref_reg_read_addr_1) begin
        `uvm_fatal("REF MODEL ERROR",$sformatf
        ("Incorrect 1st register read: Behaviour: x%2d     Model: x%2d",
        transaction.ref_ID_instr[`RS1_ADDR_HI:`RS1_ADDR_LO], transaction.ref_reg_read_addr_1))
end
if(transaction.ref_ID_instr[`RS2_ADDR_HI:`RS2_ADDR_LO] != transaction.ref_reg_read_addr_2) begin
        `uvm_fatal("REF MODEL ERROR",$sformatf
        ("Incorrect 2nd register read: Behaviour: x%2d     Model: x%2d",
        transaction.ref_ID_instr[`RS2_ADDR_HI:`RS2_ADDR_LO], transaction.ref_reg_read_addr_2))
end
```

Figure 3.24: Register Read Address Self-Checking.

```
// Check EX Stage Execution Correctness
case(transaction.ref_EX_instr[`OPCODE_HI:`OPCODE_LO])
`R_OPCODE: begin
        case(transaction.ref_EX_instr[`FUNCT3_HI:`FUNCT3_LO])
        `ADD_FUNCT3: begin
                case(transaction.ref_EX_instr[`FUNCT7_HI:`FUNCT7_LO])
                `DEFAULT_FUNCT7: begin
                        behaviour_result = data_1 + data_2;
                        if(transaction.ref_alu_output != behaviour_result)
                        `uvm_fatal("REF MODEL ERROR",$sformatf
                        ("Incorrect ADD Result: Behaviour: %8h     Model: %8h",
                        behaviour_result, transaction.ref_alu_output))
                                end
                `ALT_FUNCT7: begin
                        behaviour_result = data_1 - data_2;
                        if(transaction.ref_alu_output != behaviour_result)
                        `uvm_fatal("REF MODEL ERROR",$sformatf
                        ("Incorrect SUB Result: Behaviour: %8h     Model: %8h",
                        behaviour_result, transaction.ref_alu_output))
                            end
                endcase
                    end
        `SLL_FUNCT3: begin
                behaviour_result = data_1 << data_2[4:0];
                if(transaction.ref_alu_output != behaviour_result)
                `uvm_fatal("REF MODEL ERROR",$sformatf
                ("Incorrect SLL Result: Behaviour: %8h     Model: %8h",
                behaviour_result, transaction.ref_alu_output))
                    end
```

Figure 3.25: Instruction Execution Output Self-Checking.

```
// EXMEM Stage to MEMWB Stage Pipeline
reg_write_check = reg_write_check_buffer;
// Check instruction type at EX Stage
case(transaction.ref_EX_instr[`OPCODE_HI:`OPCODE_LO])
`R_OPCODE,
`J_OPCODE,
`JALR_OPCODE,
`U_OPCODE,
`I_OPCODE: begin
                reg_write_check_buffer = 1;
            end
default: reg_write_check_buffer = 0;
endcase

// EXMEM Stage to MEMWB Stage Pipeline
write_address_check = write_address_buffer;
write_data_check = write_data_buffer;
write_address_buffer = transaction.ref_EX_instr[`RD_ADDR_HI:`RD_ADDR_LO];
// For JAL and JALR
if(transaction.ref_EX_instr[`OPCODE_HI:`OPCODE_LO] == `J_OPCODE ||
   transaction.ref_EX_instr[`OPCODE_HI:`OPCODE_LO] == `JALR_OPCODE)
        write_data_buffer = transaction.ref_EX_pc + `INST_ADDR_SUM;
else
        write_data_buffer = transaction.ref_alu_output;

if(reg_write_check) begin
        //Check Address
        if(write_address_check != transaction.ref_reg_write_addr)
                `uvm_fatal("REF MODEL ERROR",$sformatf
                ("Incorrect write register address: Behaviour: x%2d     Model: x%2d",
                write_address_check, transaction.ref_reg_write_addr))
        //Check Data
        if(write_data_check != transaction.ref_reg_write_data)
                `uvm_fatal("REF MODEL ERROR",$sformatf
                ("Incorrect write register data: Behaviour: %8h    Model: %8h",
                write_data_check, transaction.ref_reg_write_data))
end
```

Figure 3.26: Register Write Address and Data Self-Checking.

Aside from checking on specific outputs of the reference model, self-checking also checks for assertions of stall and flush control signals. The stall assertion checking is performed on detecting load-use cases, whereas flush assertion checking is performed on detecting branch condition fulfilment or jump instruction. The following code segments showcase the self-checking mechanism for stall and flush control signal assertions:

```
// Check for Load-use case and Stall assertion
if(load_flag &&
   (transaction.ref_ID_instr[`RS2_ADDR_HI:`RS2_ADDR_LO] == transaction.ref_EX_instr[`RD_ADDR_HI:`RD_ADDR_LO] ||
    transaction.ref_ID_instr[`RS1_ADDR_HI:`RS1_ADDR_LO] == transaction.ref_EX_instr[`RD_ADDR_HI:`RD_ADDR_LO]) &&
   transaction.ref_EX_instr[`RD_ADDR_HI:`RD_ADDR_LO] != 0)
        load_use_flag = 1;
else
        load_use_flag = 0;
// ID Stage Load Detection
if(transaction.ref_ID_instr[`OPCODE_HI:`OPCODE_LO] == `LOAD_OPCODE)
        load_flag = 1;
else
        load_flag = 0;
// Check for Stall Combinational Output upon EX Stage Load ID Stage Use
if(load_use_flag) begin
        if(!transaction.ref_stall)
                `uvm_fatal("REF MODEL ERROR","Load-use Case Not Stalled")
        else
                load_use_flag = 0;
end
```

Figure 3.27: Stalling on Load-use Cases Self-Checking.

```
// Check for Branch or Jump Execution
flush_assertion = flush_buffer;
flush_buffer = transaction.ref_flush;
case(transaction.ref_EX_instr[`OPCODE_HI:`OPCODE_LO])
`SB_OPCODE: branch_check_flag = 1;
`J_OPCODE,
`JALR_OPCODE: jump_check_flag = 1;
endcase
if((branch_check_flag && transaction.ref_alu_zero) || jump_check_flag) begin
        if(!flush_assertion)
                `uvm_fatal("REF MODEL ERROR","Branch or Jump not Executed")
end
branch_check_flag = 0;
jump_check_flag = 0;
```

Figure 3.28: Flushing on Branch or Jump Instructions Self-Checking.

After self-checking, the instruction received is decoded to assembly language for user reference and instruction validity checking. The decoded instructions are also stored externally on a file for log documentation purposes. The following code segment shows part of the instruction code decoding process:

```
fl = $fopen("INSTR.txt", "a+");
case (opcode)
`R_OPCODE: begin
        case(funct3)
        `ADD_FUNCT3: begin
                case (funct7)
                `DEFAULT_FUNCT7: begin
                        `uvm_info("SCBD",$sformatf("add     x%2d,   x%2d,   x%2d",
                         rd, rs1, rs2), UVM_MEDIUM)
                        $fdisplay(fl, "add     x%2d,   x%2d,   x%2d", rd, rs1, rs2);
                                end
                `ALT_FUNCT7: begin
                        `uvm_info("SCBD",$sformatf("sub     x%2d,   x%2d,   x%2d",
                         rd, rs1, rs2), UVM_MEDIUM)
                        $fdisplay(fl, "sub     x%2d,   x%2d,   x%2d", rd, rs1, rs2);
                             end
                default: `uvm_fatal("ERROR",$sformatf
                         ("Unknown funct7 field Failing Field: %7b      Failing Instruction: %8h",
                         funct7, transaction.ref_instr))
                endcase
                        end
        `SLL_FUNCT3: begin
                `uvm_info("SCBD",$sformatf("sll     x%2d,   x%2d,   x%2d",
                 rd, rs1, rs2), UVM_MEDIUM)
                $fdisplay(fl, "sll     x%2d,   x%2d,   x%2d", rd, rs1, rs2);
                     end
        `SLT_FUNCT3: begin
                `uvm_info("SCBD",$sformatf("slt     x%2d,   x%2d,   x%2d",
                 rd, rs1, rs2), UVM_MEDIUM)
                $fdisplay(fl, "slt     x%2d,   x%2d,   x%2d", rd, rs1, rs2);
                     end
        `SLTU_FUNCT3: begin
                `uvm_info("SCBD",$sformatf("sltu     x%2d,   x%2d,   x%2d",
                 rd, rs1, rs2), UVM_MEDIUM)
                $fdisplay(fl, "sltu     x%2d,   x%2d,   x%2d", rd, rs1, rs2);
                      end
        `XOR_FUNCT3: begin
                `uvm_info("SCBD",$sformatf("xor     x%2d,   x%2d,   x%2d",
                 rd, rs1, rs2), UVM_MEDIUM)
                $fdisplay(fl, "xor     x%2d,   x%2d,   x%2d", rd, rs1, rs2);
                     end
```

Figure 3.29: Decoding of Instruction Code being executed.

The decoded instruction will also be displayed to the user on the transcript, as shown in the figure below:

```
[SCBD] addi    x 3,   x 1,   0xbb8
[SCBD] add     x 2,   x 1,   x 3
[SCBD] and     x 1,   x 2,   x 3
[SCBD] lui     x 4,   0x00001
[SCBD] sub     x 1,   x 4,   x 1
[SCBD] srli    x 5,   x 4,   12
```

Figure 3.30: Display of Decoded Instruction Code on ModelSim Transcript.

After performing a validity check on the instruction codes received, the information on each pipeline register is stored externally onto temporary text files, which will be accessed for test log documentation later. The information from the reference model and design under test are then compared. The following figure shows the storing and comparing of pipeline register values:

```
fh = $fopen("IFID.txt", "a+");
$fdisplay(fh, "%8h %8h", transaction.ref_pc, transaction.dut_pc);
$fdisplay(fh, "%8h %8h", transaction.ref_instr, transaction.dut_instr);
$fclose(fh);

if(transaction.ref_pc != transaction.dut_pc) begin
        mismatch = 1;
        `uvm_info("MISMATCH","Mismatch Encountered at IF/ID Pipeline Register",UVM_LOW);
        `uvm_info("MISMATCH","Mismatching Program Counter",UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("REF PC: %8h", transaction.ref_pc),UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("DUT PC: %8h", transaction.dut_pc),UVM_LOW);
end
if(transaction.ref_instr != transaction.dut_instr) begin
        mismatch = 1;
        `uvm_info("MISMATCH","Mismatch Encountered at IF/ID Pipeline Register",UVM_LOW);
        `uvm_info("MISMATCH","Mismatching Instruction Code",UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("REF Instr Code: %8h", transaction.ref_instr),UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("DUT Instr Code: %8h", transaction.dut_instr),UVM_LOW);
end
```

Figure 3.31: Comparison of Data between Reference Model and Design Under Test.

If any discrepancy is identified, the scoreboard asserts a mismatch flag and halts the test execution. The scoreboard will then perform the test log documentation process to capture information regarding the mismatch. If no discrepancies are found between the models, the scoreboard reiterates the checking of information received for every clock cycle until the end of the test execution.

3.3.7 Coverage Collection

The UVM coverage component has been integrated as part of the verification environment, allowing functional coverage analysis to be performed on the various test case executed. By obtaining the transaction data from the monitor component, the coverage component can perform coverage collection on the instructions being executed. A coverage plan is first written to specify the conditions to be captured for the coverage checking. The following code segment shows the conditions for functional coverage analysis:

```
covergroup                    functional_cover;
        option.per_instance = 1;
        option.get_inst_coverage = 1;
        stall:          coverpoint      transaction.ref_stall           {
        flush:          coverpoint      transaction.ref_flush           {
        uncond_jump:    coverpoint      transaction.ref_ID_instr[`OPCODE_HI:`OPCODE_LO] {
                                bins    jal     =       {`J_OPCODE};
                                bins    jalr    =       {`JALR_OPCODE};
                        }
        cond_jumps:     coverpoint      {transaction.ref_ID_instr[`FUNCT3_HI:`FUNCT3_LO],
                                         transaction.ref_ID_instr[`OPCODE_HI:`OPCODE_LO]} {
                                bins    beq_    =       {`BEQ_CVR};
                                bins    bne_    =       {`BNE_CVR};
                                bins    blt_    =       {`BLT_CVR};
                                bins    bge_    =       {`BGE_CVR};
                                bins    bltu_   =       {`BLTU_CVR};
                                bins    bgeu_   =       {`BGEU_CVR};
                        }
        loads:          coverpoint      {transaction.ref_EX_instr[`FUNCT3_HI:`FUNCT3_LO],
                                         transaction.ref_EX_instr[`OPCODE_HI:`OPCODE_LO]} {
                                bins    lb_     =       {`LB_CVR};
                                bins    lh_     =       {`LH_CVR};
                                bins    lw_     =       {`LW_CVR};
                                bins    lbu_    =       {`LBU_CVR};
                                bins    lhu_    =       {`LHU_CVR};
                        }
        instructions_A: coverpoint      {transaction.ref_EX_instr[`FUNCT3_HI:`FUNCT3_LO],
                                         transaction.ref_EX_instr[`OPCODE_HI:`OPCODE_LO]} {
                                bins    sb_     =       {`SB_CVR};
                                bins    sh_     =       {`SH_CVR};
                                bins    sw_     =       {`SW_CVR};
                                bins    addi_   =       {`ADDI_CVR};
                                bins    slli_   =       {`SLLI_CVR};
                                bins    xori_   =       {`XORI_CVR};
                                bins    ori_    =       {`ORI_CVR};
                                bins    andi_   =       {`ANDI_CVR};
                                bins    slti_   =       {`SLTI_CVR};
                                bins    sltiu_  =       {`SLTIU_CVR};
                        }
        instructions_B: coverpoint      {transaction.ref_EX_instr[`FUNCT7_HI:`FUNCT7_LO],
                                         transaction.ref_EX_instr[`FUNCT3_HI:`FUNCT3_LO],
                                         transaction.ref_EX_instr[`OPCODE_HI:`OPCODE_LO]} {
                                bins    srli_   =       {`SRLI_CVR};
                                bins    srai_   =       {`SRAI_CVR};
                                bins    add_    =       {`ADD_CVR};
                                bins    sub     =       {`SUB_CVR};
                                bins    sll_    =       {`SLL_CVR};
                                bins    xor_    =       {`XOR_CVR};
                                bins    srl_    =       {`SRL_CVR};
                                bins    sra_    =       {`SRA_CVR};
                                bins    or_     =       {`OR_CVR};
                                bins    and_    =       {`AND_CVR};
                                bins    slt_    =       {`SLT_CVR};
                                bins    sltu_   =       {`SLTU_CVR};
                        }
        instruction_C:  coverpoint      {transaction.ref_EX_instr[`OPCODE_HI:`OPCODE_LO]}       {
                                bins    lui     =       {`U_OPCODE};
                        }
        load_use_stalls:        cross   loads, stall;
        cond_jump_flushes:      cross   flush, cond_jumps;
```

Figure 3.32: Functional Coverage Cover Points.

Coverage collection provides insight towards design functionalities that are yet to be tested in test cases, allowing directed verification to be performed to verify these untested functionalities. Crossed coverage points can further increase the complexity and thoroughness of the functional coverage analysis. A well-planned coverage plan can provide accurate insight into the verification progress. When the coverage argument, "-coverage" is inputted as a command line argument, the detailed coverage information can be displayed as shown in the following figure:

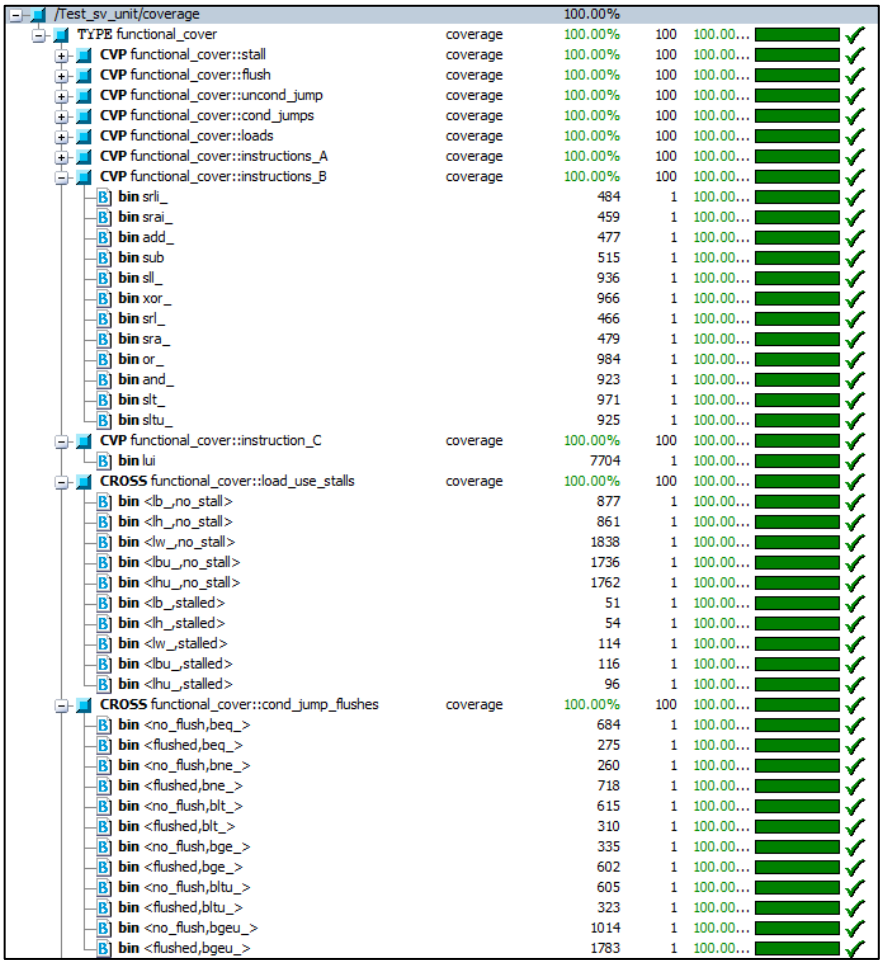


Figure 3.33: Functional Coverage Statistics for a Test Case Simulation.

The functional coverage statistics shown provide insights on the specific types of instruction that have been executed as well as the stall and flush conditions encountered during the execution of the instruction. The crossed conditions also provide information on the conditions of specific instructions, such as load-use case stalling and conditional branch flushing.

3.3.8 Mismatch Documentation

When a mismatch is encountered, the test execution is halted, and information regarding the mismatch is provided on the transcript interface as shown below:

```
[MISMATCH] Mismatch Encountered at EX/MEM Pipeline Register
[MISMATCH] Mismatching ALU Output
[MISMATCH] REF ALU Out: 0000000000000000
[MISMATCH] DUT ALU Out: 0000000000000ca0
[MISMATCH] Mismatch Encountered at EX/MEM Pipeline Register
[MISMATCH] Mismatching ALU Zero
[MISMATCH] REF ALU Zero: 1
[MISMATCH] DUT ALU Zero: 0
[MISMATCH] Mismatch encountered, Logging Test Information
```

Figure 3.34: Mismatch Message generated on ModelSim Transcript.

The information displayed provides insight into the pipeline register that provided mismatching values as well as the mismatching parameter and values. Next, the scoreboard component has been implemented with a feature to perform test log documentation, which can be used for debugging purposes. The test log documentation includes instructions executed and the internal states of the pipeline registers.

The logging process begins by clearing several cycles of null information stored on different pipeline registers. These null information stored on the pipeline registers are due to the pipeline filling process whereby for a 5-stage pipeline, 4 clock cycles are required for all the pipeline registers to be filled with instructions. Clearing the null information allows the pipeline registers to be aligned in instruction execution which eases the logging process. The following figure shows the pipeline filling process and the null information on the pipeline stages:

	Instruction Fetch	Instruction Decode	Execution	Memory Access	Writeback
Clock Cycle 1	Instruction 1	**NULL**	**NULL**	**NULL**	**NULL**
Clock Cycle 2	Instruction 2	Instruction 1	**NULL**	**NULL**	**NULL**
Clock Cycle 3	Instruction 3	Instruction 2	Instruction 1	**NULL**	**NULL**
Clock Cycle 4	Instruction 4	Instruction 3	Instruction 2	Instruction 1	**NULL**
Clock Cycle 5	Instruction 5	Instruction 4	Instruction 3	Instruction 2	Instruction 1

Figure 3.35: Pipeline Filling and Null Information on Pipeline Stages.

The following code segment shows the removal of invalid information from the temporary text file used to store pipeline register data:

```
fh_ifid = $fopen("IFID.txt", "r");
fh_idex = $fopen("IDEX.txt", "r");
fh_exmem = $fopen("EXMEM.txt", "r");
fh_memwb = $fopen("MEMWB.txt", "r");
fh_instr = $fopen("INSTR.txt", "r");
fh_stall = $fopen("PIPELINE.txt", "r");
fh_flush = $fopen("FLUSH.txt","r");
fh_jump = $fopen("JUMP.txt","r");
fh_log = $fopen("LOG.txt", "w");

// Dump NULL information
// Clear 1 cycle for ID/EX and macro status
for(int i = 0;  i < 1;  i++)    begin
        for(int k = 0; k < 10; k++)
                code = $fscanf(fh_idex,"%s",dump);
        code = $fscanf(fh_stall,"%s",dump);
        code = $fscanf(fh_flush,"%s",dump);
        code = $fscanf(fh_jump,"%s",dump);
end
// Clear 2 cycles for EX/MEM
for(int i = 0; i < 2;   i++)    begin
        for(int k = 0; k < 6; k++)
                code = $fscanf(fh_exmem,"%s",dump);
end
// Clear 3 cycles for MEM/WB
for(int i = 0; i < 3;   i++)    begin
        for(int k = 0; k < 11; k++)
                code = $fscanf(fh_memwb,"%s",dump);
end
```

Figure 3.36: Removing invalid data from Pipeline Register Data Record.

After removing the invalid pipeline filling data, the pointers pointing towards the data on each pipeline register data record are aligned and accessed, formatted, and outputted into the test log documentation. The following code segment shows the formatting and logging of test results:

```
code = $fscanf(fh_exmem,"%s",file_transact_1);  //Control Signal
code = $fscanf(fh_exmem,"%s",file_transact_2);
code = $fscanf(fh_exmem,"%s",file_transact_3);  //ALU Output
code = $fscanf(fh_exmem,"%s",file_transact_4);
code = $fscanf(fh_exmem,"%s",file_transact_5);  //Zero
code = $fscanf(fh_exmem,"%s",file_transact_6);
$fdisplay(fh_log, "-------------------------ALU Operation----------------------------");

$fdisplay(fh_log,       "Control Signal     : REF: %8b\n                    DUT: %8b",
                file_transact_1, file_transact_2);
$fdisplay(fh_log,       "ALU Output         : REF: %8h\n                    DUT: %8h",
                file_transact_1, file_transact_2);
$fdisplay(fh_log,       "ALU Zero Signal    : REF: %1b\n                    DUT: %1b",
                file_transact_1, file_transact_2);
if((file_transact_1 != file_transact_2) || (file_transact_3 != file_transact_4) ||
   (file_transact_5 != file_transact_6)) begin
        mismatch_found = 1;
        break;
end
```

Figure 3.37: Formatting and Logging of Information.

The following figure shows an example of test log output for a mismatch case:

```
=========================Instruction    6=========================
Instruction Address: REF: 0000004c
                     DUT: 0000004c
Instruction Code   : REF: cc938da3
                     DUT: cc938da3
Instruction Decode : sb      x 9,   cdb(x 7)

------------------------Register Access--------------------------
Register Read      : REF: x 7:00000000
                          x 9:00000000
Register Read      : DUT: x 7:00000000
                          x 9:00000000
Immediate  Value   : REF:     00000cdb
                     DUT:     00000cdb
------------------------ALU Operation---------------------------
Control Signal     : REF: 100010
                     DUT: 100010
ALU Control Signal : REF: 0010
                     DUT: 0010
ALU Output         : REF: 00000000
                     DUT: 00000cdb
```

Figure 3.38: Test Log Documentation on Mismatch Instruction.

The information documented in the test log can provide context on mismatching cases, allowing a thorough analysis to be performed. The test log documenting feature can also be configured such that test log documentation is also performed for passing test cases. This is achieved through the command line argument "+TESTLOG".

File handling of the test log documents generated is performed after test execution. The test log documents are relocated to relevant test case folders in the test result directory. The temporary text files created for the test log documenting process are also removed to ensure the system is clutter-free.

3.3.9 Macro Check

Another additional feature implemented is macro status consistency checking. This feature ensures that the macro status, such as the number of stall occurrences and flush occurrences during a regression test, is consistent with the previously stored macro status data. If the macro status of the new design is inconsistent with previous macro status data, the debugging performed has introduced other bugs that have altered the functioning of the system and need to be further analysed.

This feature can be bypassed by using the "+SKIP_MACRO_CHECK" command line argument. For cases where the new macro status data is correct and the recorded status data is to be updated, the command line argument "+MACRO_OVERWRITE" can be inputted to configure the verification environment to update the previously stored macro status data. The following figure shows the macro status data record of a sample test case:

```
Stall:           4
Flush:          55
Last Updated on Thu 12/23/2021 01:03
```

Figure 3.39: Macro Status Data Record.

The macro status of a test case will also be recorded on the test log document at the end of the test, as shown in the following figure:

```
=========================End of Testlog============================
Test Completion Time:                        Mon 12/27/2021 12:26
Total Stall Encountered:                                        1
Total Flush Encountered:                                       70
```

Figure 3.40: Macro Status at end of Test Log Document.

Chapter 4

RESULTS AND DISCUSSIONS

4.1 Self-Checking Bug Detection

Several tests have been performed to verify the bug detection capabilities of the verification methodology implemented. Bugs were intentionally introduced to the reference model to test the implemented self-checking mechanism. Due to the complexity of the design, the self-checking mechanism is only implemented for a limited number of characteristics of the RISC-V pipelined processor design listed below:

- Pipeline Stalling
- Pipeline Flushing
- Implemented Instruction Functionality Correctness

A self-checking mechanism is crucial for ensuring the functional correctness of the reference model. This mechanism is even more significant when the reference model is used for output comparing against a design. Ensuring the functional correctness of the reference model can increase the overall confidence in the results of the verification performed.

For the first testing performed, part of the reference model ALU source code is altered as shown in the figure below:

```
always @(*) begin: main_alu_block
        case(alu_ctl)
        `AND_CTL                : alu_output = data_1 & data_2;
        //`OR_CTL               : alu_output = data_1 | data_2;
        //`ADD_CTL              : alu_output = data_1 + data_2;
```

Figure 4.1: Modification to Reference Unit ALU Source Code.

```
[SCBD] addi    x 3,   x 0,   0x07d
[SCBD] add     x 2,   x 3,   x 0
 [REF MODEL ERROR] Incorrect ADDI Result: Behaviour: 0000007d     Model: 00000000
```

Figure 4.2: Reference Model Instruction Functionality Bug Detection ModelSim Transcript Message.

From the UVM message displayed on the transcript interface, the bug introduced has been detected. The expected outcome of the instruction *addi x3, x0, 0x07d* of *0000007d* differs from the modified model outcome of *00000000*. The discrepancy encountered allows the system to identify this error as a logical bug on the reference model.

For the subsequent testing, the hazard detection unit is modified such that the flush signal is never asserted as shown in the modified code below:

```
//if((zero && branch) || jump)
//      flush = 1;
//else
//      flush = 0;
```

Figure 4.3: Modification to Reference Unit Hazard Detection Unit Flush Assertion Source Code.

```
[SCBD] jal     x31,   0x000a8
[SCBD] lui     x30,   0x11111
 [REF MODEL ERROR] Branch or Jump not Executed
```

Figure 4.4: Reference Model Flush Nonassertion Bug Detection ModelSim Transcript Message.

As observed from the figure shown, the self-checking mechanism detects the nonassertion of the flush signal. When the instruction *jal x31, 0x000a8* is executed, the control unit is to assert flush control signal to flush out nulled information on the pipeline registers. However, it fails to do so due to the modification to the source code introduced. The nonassertion of the flush signal on the reference model is then identified as a logical bug on the reference model.

Similarly, the stall control signal of the hazard detection unit is modified to check for nonassertion of stall control signal on load-use case detection. The following figure shows the modified code:

```
if(mem_read && (idex_reg_addr_1 == exmem_rd || idex_reg_addr_2 == exmem_rd) && exmem_rd != 0) begin
        if ((idex_reg_addr_1 == exmem_rd && fwrd_mux_1 != `FWRD_MEM) ||
            (idex_reg_addr_2 == exmem_rd && fwrd_mux_2 != `FWRD_MEM))
                stall = 0;      // modified
end
else
        stall = 0;
```

Figure 4.5: Modification to Reference Unit Hazard Detection Unit Stall Assertion Source Code.

```
[SCBD] lw      x 4,    0x101(x 1)
[SCBD] addi    x14,    x 4,    0x000
 [REF MODEL ERROR] Load-use Case Not Stalled
```

Figure 4.6: Reference Model Stall Nonassertion Bug Detection ModelSim Transcript Message.

From the figure shown, the instruction sequence of *lw x4, 0x101(x1)* followed by *addi x14, x4, 0x000* which depicted a load-use case of the register *x4*. The self-checking mechanism detects the nonassertion of the stall control signal and identifies the logical bug introduced on the reference model.

4.2 Design Bug Detection

The design verification is performed by comparing various internal states of the design under test against the internal states of the reference model. For the testing of the bug detection capabilities of the design verification methodology implemented, bugs are intentionally introduced to the design under test.

In the first testing, a bug is introduced to the program counter source code. The increment of 4 is altered to an increment of 2 and tested as shown in the following code segment:

```
//instr_addr <= instr_addr + `INST_ADDR_SUM;
instr_addr <= instr_addr + 2;
```

Figure 4.7: Modification to Design Under Test Program Counter Source Code.

```
[SCBD] addi     x 3,    x 0,    0x07d
[SCBD] add      x 2,    x 3,    x 0
[MISMATCH] Mismatch Encountered at IF/ID Pipeline Register
[MISMATCH] Mismatching Program Counter
[MISMATCH] REF PC: 00000004
[MISMATCH] DUT PC: 00000002
```

Figure 4.8: Program Counter Mismatch Detection ModelSim Transcript Message.

When the instruction *addi x3, x0, 0x07d* is fetched, the program counter increments by 4. Due to the alteration to the source code, the program counter of the design under test only increments by 2. When the second instruction, *add x2, x3, x0* is fetched, the instruction address information stored on the IF/ID pipeline register mismatches. The verification testbench identifies the discrepancy and provides relevant information to the user for debug.

In the subsequent testing, the updating of the program counter with effective target address generated from immediate generation unit is altered as shown below:

```
if(jump_reg)
        instr_addr <= {alu_output[`DATA_WIDTH-1:2],2'b00};
//else
//      instr_addr <= imm_addr;
```

Figure 4.9: Modification to Design Under Test Program Counter Target Address Branching Source Code.

```
[SCBD] jal      x31,    0x000a8
[SCBD] lui      x30,    0x11111
[SCBD] nop
[SCBD] addi     x30,    x 0,    0xf96
[MISMATCH] Mismatch Encountered at IF/ID Pipeline Register
[MISMATCH] Mismatching Program Counter
[MISMATCH] REF PC: 00000138
[MISMATCH] DUT PC: 00000098
```

Figure 4.10: Program Counter Branch Target Address Mismatch Detection ModelSim Transcript Message.

When an unconditional branch instruction (*jal x31, 0x000a8*) is executed, instruction is to be fetched from the target address. Due to the modification performed on the program counter source code, the design under test does not update the program counter with a new program counter value. The verification testbench identifies the discrepancy and provides relevant information to the user.

For a similar case, the updating of program counter with effective target address read from register by the instruction *jump and link register* (*jalr*) is modified as shown in the code segment below:

```
if(flush) begin: flush_pc
        //if(jump_reg)
        //      instr_addr <= {alu_output[`DATA_WIDTH-1:2],2'b00};
        //else
                instr_addr <= imm_addr;
```

Figure 4.11: Modification to Design Under Test Program Counter Jump Register Target Address Source Code.

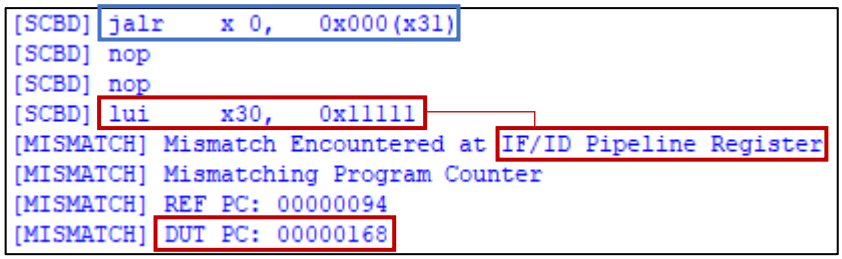


Figure 4.12: Program Counter Jump Register Target Address Mismatch Detection ModelSim Transcript Message.

When *jump and link register* instruction is executed, the program counter is to be updated with the target address read from a register. From the alteration performed to the program counter, the design under test provides an incorrect target address. The verification testbench identifies the discrepancy, and relevant information are provided to the user for debugging to be performed.

For the memory technology utilized, each memory register holds a byte (8-bit) of information. For a 32-bit instruction code to be read from the instruction memory, aligned read access need to be performed to 4 instruction memory registers. The following test ignores the memory technology implemented and performs a singular read access to a memory location for instruction code fetches as shown in the code segment below:

```
always @(*)     begin: instruction_fetch
        //instr_code = {rom[instr_addr],
        //              rom[instr_addr + 1],
        //              rom[instr_addr + 2],
        //              rom[instr_addr + 3]};
        instr_code = rom[instr_addr];
end: instruction_fetch
```

Figure 4.13: Modification to Design Under Test Instruction Memory Source Code.

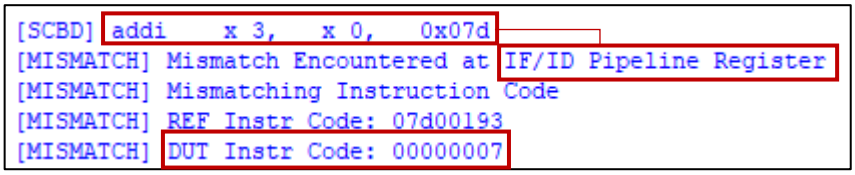


Figure 4.14: Instruction Code Mismatch Detection ModelSim Transcript Message.

As a result of the modification, only a byte of instruction code information is fetched. The verification testbench identifies the discrepancy in the instruction code fetched and provides the user relevant information for debugging to be performed.

The following test alters the read register address information read from the instruction code:

```
// assign  reg_read_addr_1 = reset ? 0 : instr_code[`RS1_ADDR_HI:`RS1_ADDR_LO];
   assign  reg_read_addr_1 = reset ? 0 : instr_code[20:16];
   assign  reg_read_addr_2 = reset ? 0 : instr_code[`RS2_ADDR_HI:`RS2_ADDR_LO];
```

Figure 4.15: Modification to Design Under Test Register File Source Code.

```
[SCBD] addi     x 3,    x 0,    0x07d
[SCBD] add      x 2,    x 3,    x 0
[MISMATCH] Mismatch Encountered at ID/EX Pipeline Register
[MISMATCH] Mismatching Register 01 Address
[MISMATCH] REF Reg01 Addr:  0
[MISMATCH] DUT Reg01 Addr: 16
```

Figure 4.16: System Register Read Register Address Mismatch Detection ModelSim Transcript Message.

The testbench identifies the discrepancy in the register address accessed by design under test and provides relevant information for debugging.

When the immediate generation unit source code is altered such that the immediate value generated is inconsistent with the instruction set architecture specification, the following results are obtained:

```
    `I_OPCODE   : imm_val = {{20{instr_code[31]}},instr_code[11:0]};  //modified
// `I_OPCODE   : imm_val = {{20{instr_code[31]}},instr_code[31:20]}; //original
```

Figure 4.17: Modification to Design Under Test Immediate Generation Unit Source Code.

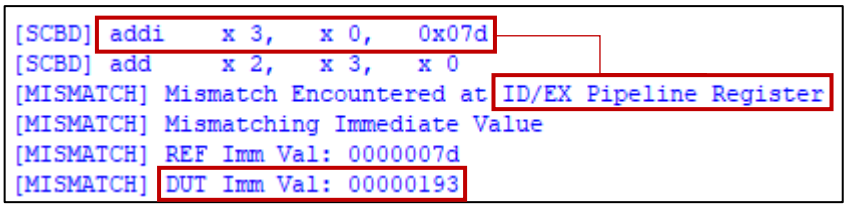


Figure 4.18: Immediate Value Mismatch Detection ModelSim Transcript Message.

The verification testbench identifies the incorrect immediate value generated (*00000193*) by design under test.

For the following testing, the control unit is altered such that an incorrect control signal is provided for load instructions as shown in the code segment below:

```
`LOAD_OPCODE: begin
                //ctl_op = `LOAD_CTL_SGNL;
                ctl_op = `NOP_CTL_SGNL;
                alu_op = `IMM_ADDR_CALC_ALU_OP;
              end
```

Figure 4.19: Modification to Design Under Test Control Unit Load Instruction Control Signal Source Code.

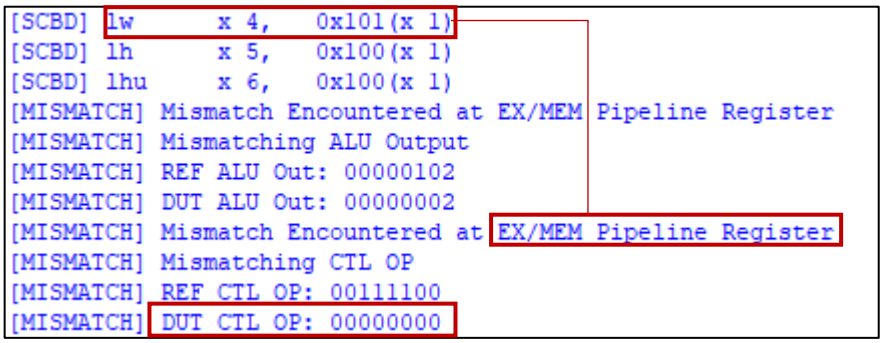


Figure 4.20: Control Signal Mismatch Detection ModelSim Transcript Message.

When a load instruction (*lw x4, 0x101(x1)*) is executed, the testbench detects the discrepancy in the control signal of the instruction. The ALU operation is also altered due to the alteration of the control signal (*ALUSrc*), resulting in a mismatched output as a side effect.

In the following alteration, the *shift right arithmetic* operation coded in the ALU is altered to have a similar effect as *shift right logical* operation:

```
    `SRA_CTL                : alu_output = data_1 >> data_2[4:0];           //modified
//  `SRA_CTL                : alu_output = $signed(data_1) >>> data_2[4:0]; //original
```

Figure 4.21: Modification to Design Under Test ALU Source Code.

```
[SCBD] sra      x11,    x 4,    x 3
[SCBD] srai     x12,    x 5,     8
[SCBD] bne      x11,    x12,    0x10c
[MISMATCH] Mismatch Encountered at EX/MEM Pipeline Register
[MISMATCH] Mismatching ALU Output
[MISMATCH] REF ALU Out: ffffffff
[MISMATCH] DUT ALU Out: 00000007
```

Figure 4.22: ALU Output Mismatch Detection ModelSim Transcript Message.

When a *shift right arithmetic* instruction is executed, the design under test ALU produces an incorrect outcome. The testbench detects the discrepancy and provides information to the user for debugging to be performed.

When the data forwarding functionality is altered as shown in the code segment below, the following results are obtained:

```
if(exmem_opcode != `LOAD_OPCODE && exmem_reg_write && exmem_rd != 0) begin: exmem_fwrd
        //if(reg_1 == exmem_rd)
        //      fwrd_mux_1 = `FWRD_ALU;
        //else if (reg_1 == memwb_rd && memwb_reg_write) //CHECKME
        //      fwrd_mux_1 = `FWRD_MEM;
        //else
        //      fwrd_mux_1 = `NO_FWRD;
        //if(reg_2 == exmem_rd)
        //      fwrd_mux_2 = `FWRD_ALU;
        //else if (reg_2 == memwb_rd && memwb_reg_write)
        //      fwrd_mux_2 = `FWRD_MEM;
        //else
                fwrd_mux_2 = `NO_FWRD;
end: exmem_fwrd
```

Figure 4.23: Modification to Design Under Test Forwarding Unit Source Code.

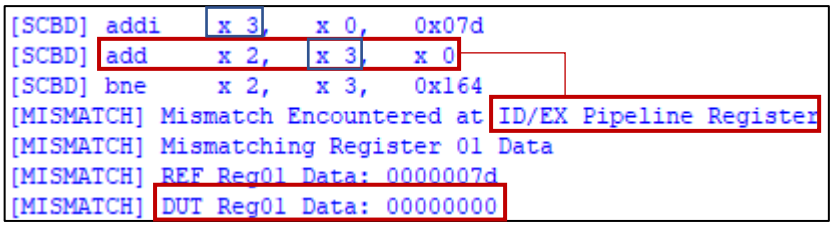


Figure 4.24: Forwarded Operand Mismatch ModelSim Transcript Message.

In the instruction sequence above, the data dependency on the register *x3* warrants data forwarding. Data forwarding ensures the updated information is utilized as an operand for the subsequent instruction operation. Due to the alteration performed, the data forwarding on design under test is not executed, resulting in an incorrect operand value. The discrepancy is detected by the verification testbench and relevant information is provided to the user for debugging.

The next testing modifies the hazard detection unit, ensuring the nonassertion of flush control signal.

```
if((zero && branch) || jump)
        //flush = 1;
        flush = 0;
else
        flush = 0;
```

Figure 4.25: Modification to Design Under Test Control Unit Flush Control Source Code.

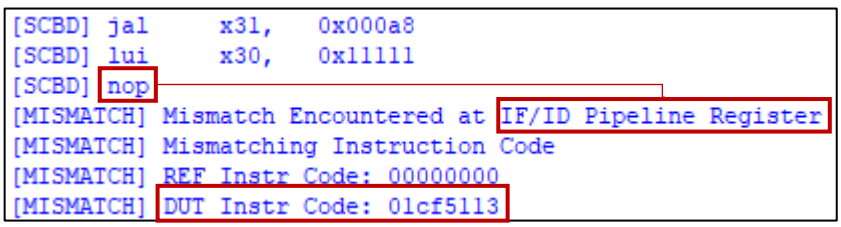


Figure 4.26: Flush Nonassertion Mismatch ModelSim Transcript Message.

When flush control signal is asserted, the pipeline registers are to flush the invalidated instructions by discarding the information of the instructions. From the results obtained,

due to the nonassertion of the flush signal by design under test hazard detection unit, the design under test IF/ID pipeline register still contains the invalidated instruction information. The verification testbench detects the discrepancy, and relevant information is provided to the user for debugging.

Similarly, testing for the stall nonassertion detection can be performed by ensuring the nonassertion of the stall control signal on the design under test hazard detection unit as shown in the code segment below:

```
if(mem_read && (idex_reg_addr_1 == exmem_rd || idex_reg_addr_2 == exmem_rd) && exmem_rd != 0) begin
        if ((idex_reg_addr_1 == exmem_rd && fwrd_mux_1 != `FWRD_MEM) ||
            (idex_reg_addr_2 == exmem_rd && fwrd_mux_2 != `FWRD_MEM))
                //stall = 1;
                stall = 0;
end
else
        stall = 0;
```

Figure 4.27: Modification to Design Under Test Control Unit Stall Control Source Code.

```
[SCBD] lw      x 4,    0x101(x 1)
[SCBD] addi    x14,    x 4,    0x000
[SCBD] addi    x14,    x 4,    0x000
[MISMATCH] Mismatch Encountered at IF/ID Pipeline Register
[MISMATCH] Mismatching Program Counter
[MISMATCH] REF PC: 000000b0
[MISMATCH] DUT PC: 000000b4
```

Figure 4.28: Stall Nonassertion Mismatch ModelSim Transcript Message.

From the results obtained, the instruction sequence of *lw x4, 0x101(x1)* to *addi x14, x4, 0x000* showcases a load-use case with a data dependency on the register *x4*. As a result of a load-use case, stall control signal needs to be asserted to allow the pipeline flow to be partially halted. Due to the nonassertion of the stall control signal, the pipeline flow of the design under test is not halted is observed in the mismatching program counter. The testbench detects the bug introduced by the stall nonassertion and provides relevant information for debugging.

For the data memory, similar memory technology has been utilized. Each data memory register holds a byte (8-bit) of information. Aligned read access needs to be performed for proper data memory access. In the following testing, the memory technology implemented is ignored, and read access is performed to only one location as shown in the code segment below:

```
   assign  read_data = ram[address];
// assign  read_data = {(ram[address+3] === 8'bx ? 8'b0 : ram[address+3]),
//                      (ram[address+2] === 8'bx ? 8'b0 : ram[address+2]),
//                      (ram[address+1] === 8'bx ? 8'b0 : ram[address+1]),
//                      (ram[address]   === 8'bx ? 8'b0 : ram[address])};
```

Figure 4.29: Modification to Design Under Test Data Memory Load Data Source Code.

```
[SCBD] lw       x 4,    0x101(x 1)
[SCBD] lh       x 5,    0x100(x 1)
[SCBD] lhu      x 6,    0x100(x 1)
[SCBD] lb       x 7,    0x0ff(x 1)
[MISMATCH] Mismatch Encountered at MEM/WB Pipeline Register
[MISMATCH] Mismatching Register Write Data
[MISMATCH] REF Reg Wr.Data: ff96ffff
[MISMATCH] DUT Reg Wr.Data: 000000ff
```

Figure 4.30: Load Data Mismatch ModelSim Transcript Message.

From the result obtained, when a load word instruction is executed, 4 memory locations need to be accessed for the word (32-bit) information. As a result of disregard towards the memory technology implemented, the design under test reads only a byte of information. The discrepancy is detected by the testbench and shown to the user for debugging to be performed.

Lastly, the following alteration made to the data store functionality of the data memory unit disregards the amount of data to be stored:

```
   assign  stored_data = mem_write ? reg_data : 0;
// assign  stored_data = mem_write ? (funct3 == `SB_FUNCT3 ? {24'b0,reg_data[7:0]} :
//                                   (funct3 == `SH_FUNCT3 ? {16'b0,reg_data[15:0]} : reg_data)) : 0;
```

Figure 4.31: Modification to Design Under Test Data Memory Store Data Source Code.

```
[SCBD] sh       x30,    0x0ff(x 2)
[SCBD] sb       x30,    0x0ff(x 3)
[SCBD] lw       x 4,    0x101(x 1)
[SCBD] lh       x 5,    0x100(x 1)
[MISMATCH] Mismatch Encountered at MEM/WB Pipeline Register
[MISMATCH] Mismatching Memory Write Data
[MISMATCH] REF Mem Wr.Data: 0000ff96
[MISMATCH] DUT Mem Wr.Data: ffffff96
```

Figure 4.32: Store Data Mismatch ModelSim Transcript Message.

The instruction store halfword is to store 16 bits of information onto the memory. Due to the modification performed, the design under test stores a word instead. The testbench detects the discrepancy, and relevant information is provided to the user for debugging.

The verification testbench has been programmed to monitor most internal states of the design from the results provided. When a discrepancy is detected from the model outcome comparison simulation, the verification testbench provides information on the mismatching values, providing an automated logical error detection to the verification. This methodology effectively saves an immeasurable amount of time and provides the user with a more straightforward debugging process with the information provided. The challenges of this form of verification methodology would be the strict requirements of adherence to the specifications and the interfacing work required for proper synchronized operation of both reference model and design under test.

4.3 Directed Verification

For directed verification, a specific verification scheme has been employed to perform testing on specific criteria listed below:

- Functionalities of all implemented instructions
- Data Forwarding
- Pipeline Stalling
- Pipeline Flushing
- Misaligned Data Memory Access

The following table shows the instructions of the directed test and their expected outcome:

Table 4.1: Directed Verification Test Program.

Instruction Address	Instruction Code (Assembly Language)				Comment
0x00000000	addi	x3,	x0,	125	x3 = 125 ($7D_{16}$)
0x00000004	add	x2,	x3,	x0	Forward data from ALU Output x2 = 125 ($7D_{16}$)
0x00000008	bne	x2,	x3,	END	Forward data from ALU Output Forward data from MEM/WB.Reg If x2 ≠ x3, jump to END
0x0000000C	addi	x4,	x0,	3971	x4 = -125(FFFF $FF83_{16}$) <sign-extended>
0x00000010	sub	x5,	x0,	x2	x5 = -125(FFFF $FF83_{16}$)
0x00000014	bne	x4,	x5,	END	Forward data from ALU Output Forward data from MEM/WB.Reg If x4 ≠ x5, jump to END
0x00000018	addi	x6,	x0,	1	x6 = 1
0x0000001C	srli	x7,	x3,	6	x7 = 1
0x00000020	bne	x6,	x7,	END	Forward data from ALU Output Forward data from MEM/WB.Reg If x6 ≠ x7, jump to END
0x00000024	srl	x8,	x6,	x6	x8 = 0
0x00000028	bne	x8,	x0,	END	Forward data from ALU Output If x8 ≠ 0, jump to END
0x0000002C	slli	x9,	x7,	1	x9 = 2
0x00000030	sll	x10,	x7,	x6	x10 = 2
0x00000034	bne	x9,	x10,	END	Forward data from ALU Output Forward data from MEM/WB.Reg If x9 ≠ x10, jump to END
0x00000038	slt	x1,	x6,	x9	x1 = 1
0x0000003C	beq	x1,	x0,	END	Forward data from ALU Output

					If x1 = x0, jump to END
0x00000040	slt	x1,	x4,	x3	x1 = 1
0x00000044	beq	x1,	x0,	END	Forward data from ALU Output If x1 = x0, jump to END
0x00000048	sltu	x1,	x4,	x3	x1 = 0
0x0000004C	bne	x1,	x0,	END	Forward data from ALU Output If x1 ≠ x0, jump to END
0x00000050	xor	x2,	x2,	x3	x2 = 0
0x00000054	bne	x2,	x0,	END	Forward data from ALU Output If x2 ≠ x0, jump to END
0x00000058	sra	x11,	x4,	x3	x11 = -1 (FFFF FFFF$_{16}$)
0x0000005C	srai	x12,	x5,	8	x12 = -1 (FFFF FFFF$_{16}$)
0x00000060	bne	x11,	x12,	END	Forward data from ALU Output Forward data from MEM/WB.Reg If x11 ≠ x12, jump to END
0x00000064	ori	x13,	x0,	3	x13 = 3
0x00000068	or	x14,	x9,	x6	x14 = 3
0x0000006C	bne	x13,	x14,	END	Forward data from ALU Output Forward data from MEM/WB.Reg If x13 ≠ x14, jump to END
0x00000070	and	x15,	x13,	x11	x15 = 3
0x00000074	andi	x16,	x14,	15	x16 = 3
0x00000078	bne	x15,	x16,	END	Forward data from ALU Output Forward data from MEM/WB.Reg If x15 ≠ x16, jump to END
0x0000007C	slti	x17,	x15,	4	x17 = 1
0x00000080	sltiu	x18,	x5,	4095	x18 = 1
0x00000084	bne	x17,	x18,	END	Forward data from ALU Output Forward data from MEM/WB.Reg If x17 ≠ x18, jump to END
0x00000088	xori	x1,	x0,	1	x1 = 1
0x0000008C	beq	x1,	x0,	END	Forward data from ALU Output If x1 = x0, jump to END
0x00000090	jal	x31,	STORE_ROUT		x31 = 00000094$_{16}$ Jump to STORE_ROUT
0x00000094	lui	x30,	69905		x30 = 11111000$_{16}$
0x00000098	srli	x2,	x30,	28	x2 = 1
0x0000009C	blt	x1,	x2,	END	Forward data from ALU Output If x1 < x2, jump to END
0x000000A0	bge	x0,	x1,	END	If x0 > x1, jump to END
0x000000A4	bltu	x1,	x2,	END	If x1 < x2, jump to END
0x000000A8	bgeu	x0,	x1,	END	If x0 > x1, jump to END
STALL_CHECK:					
0x000000AC	lw	x4,	257(x1)		x4 = FF96 FFFF$_{16}$
0x000000B0	addi	x14,	x4,	0	Stall x14 = FF96 FFFF$_{16}$
0x000000B4	lh	x5,	256(x1)		x5 = FFFF FFFF$_{16}$
0x000000B8	addi	x15,	x5,	0	Stall x15 = FFFF FFFF$_{16}$
0x000000BC	lhu	x6,	256(x1)		x6 = 0000 FFFF$_{16}$
0x000000C0	addi	x16,	x6,	0	Stall

					x16 = 0000 $FFFF_{16}$
0x000000C4	lb	x7,	255(x1)		x7 = FFFF $FF96_{16}$
0x000000C8	addi	x17,	x7,	0	Stall x17 = FFFF $FF96_{16}$
0x000000CC	lbu	x8,	255(x1)		x8 = 0000 0096_{16}
0x000000D0	addi	x18,	x8,	0	Stall x18 = 0000 0096_{16}
0x000000D4	beq	x20,	x0,	SKIP1	If x20 = x0, jump to SKIP1
0x000000D8	lui	x28,	912095		x28 = DEAD $F000_{16}$
0x000000DC	lui	x29,	912095		x29 = DEAD $F000_{16}$
0x000000E0	lui	x30,	912095		X30 = DEAD $F000_{16}$
SKIP1					
0x000000E4	bne	x1,	x0,	SKIP2	If x1 ≠ x0, jump to SKIP2
0x000000E8	lui	x28,	912095		x28 = DEAD $F000_{16}$
0x000000EC	lui	x29,	912095		x29 = DEAD $F000_{16}$
0x000000F0	lui	x30,	912095		X30 = DEAD $F000_{16}$
SKIP2					
0x000000F4	bge	x1,	x14,	SKIP3	If x1 > x14, jump to SKIP3
0x000000F8	lui	x28,	912095		x28 = DEAD $F000_{16}$
0x000000FC	lui	x29,	912095		x29 = DEAD $F000_{16}$
0x00000100	lui	x30,	912095		X30 = DEAD $F000_{16}$
SKIP3					
0x00000104	blt	x11,	x1,	SKIP4	If x11 < x1, jump to SKIP4
0x00000108	lui	x28,	912095		x28 = DEAD $F000_{16}$
0x0000010C	lui	x29,	912095		x29 = DEAD $F000_{16}$
0x00000110	lui	x30,	912095		X30 = DEAD $F000_{16}$
SKIP4					
0x00000114	bgeu	x5,	x4,	SKIP5	If x5 > x4, jump to SKIP5
0x00000118	lui	x28,	912095		x28 = DEAD $F000_{16}$
0x0000011C	lui	x29,	912095		x29 = DEAD $F000_{16}$
0x00000120	lui	x30,	912095		X30 = DEAD $F000_{16}$
SKIP5					
0x00000124	bltu	x17,	x5,	SKIP6	If x17 < x5, jump to SKIP6
0x00000128	lui	x28,	912095		x28 = DEAD $F000_{16}$
0x0000012C	lui	x29,	912095		x29 = DEAD $F000_{16}$
0x00000130	lui	x30,	912095		X30 = DEAD $F000_{16}$
SKIP6					
0x00000134	jal	x31,	END		x31 = $000000D8_{16}$ Jump to END
STORE_ROUT:					
0x00000138	addi	x30,	x0,	3990	x30 = 3990(FFFF $FF96_{16}$)
0x0000013C	addi	x1,	x0,	1	x1 = 1
0x00000140	addi	x2,	x1,	4	x2 = 5
0x00000144	addi	x3,	x2,	4	x3 = 9
0x00000148	sw	x30,	255(x1)		Memory[0000 0100_{16}] = FF FF FF 96_{16}
0x0000014C	sh	x30,	255(x2)		Memory[0000 0104_{16}] = FF 96_{16}
0x00000150	sb	x30,	255(x3)		Memory[0000 0108_{16}] = 96_{16}
0x00000154	lw	x4,	257(x1)		x4 = FF96 $FFFF_{16}$
0x00000158	lh	x5,	256(x1)		x5 = FFFF $FFFF_{16}$
0x0000015C	lhu	x6,	256(x1)		x6 = 0000 $FFFF_{16}$
0x00000160	lb	x7,	255(x1)		x7 = FFFF $FF96_{16}$

0x00000164	lbu x8, 255(x1)	x8 = 0000 0096_{16}
0x00000168	jalr x0, 0(x31)	Return to 0000 0094_{16}
END:		
0x0000016C	nop	No Operation
0x00000170	nop	No Operation
0x00000174	nop	No Operation
0x00000178	nop	No Operation
0x0000017C	end	End of Test

Test Program Instruction Highlight Indication

Pass Signature Value Instructions
Fail Signature Value Instructions
Branch/Jump Executed
Return Address

For verification of the test program results, specific signature values can be checked from the general-purpose registers and data memory. The simulated results shown in the figure below tallies with the expected outcome:

```
   4    ff96ffff
   5    ffffffff
   6    0000ffff
   7    ffffff96
   8    00000096
  14    ff96ffff
  15    ffffffff
  16    0000ffff
  17    ffffff96
  18    00000096
```

Figure 4.33: General-Purpose Register Signature Values.

```
00000100    96 ff ff ff
00000104    96 ff xx xx
00000108    96 xx xx xx
```

Figure 4.34 Memory Signature Values.

For a detailed approach towards test verification, the primitive methodology of waveform debugging is performed as shown in the following figures:

When the second instruction (*add x2, x3, x0*) is stored on the ID/EX pipeline register, the correct forwarding mechanism is executed, forwarding the ALU output from EX/MEM pipeline register to the ALU unit. The forwarding mechanism is triggered by the data dependence of the register *x3* on the first instruction (*addi x3, x0, 125*).

Both register *x2* and *x3* are data forwarded from EX/MEM pipeline register and MEM/WB pipeline register for the third instruction (*bne x2, x3, END*), providing the updated data to the ALU unit.

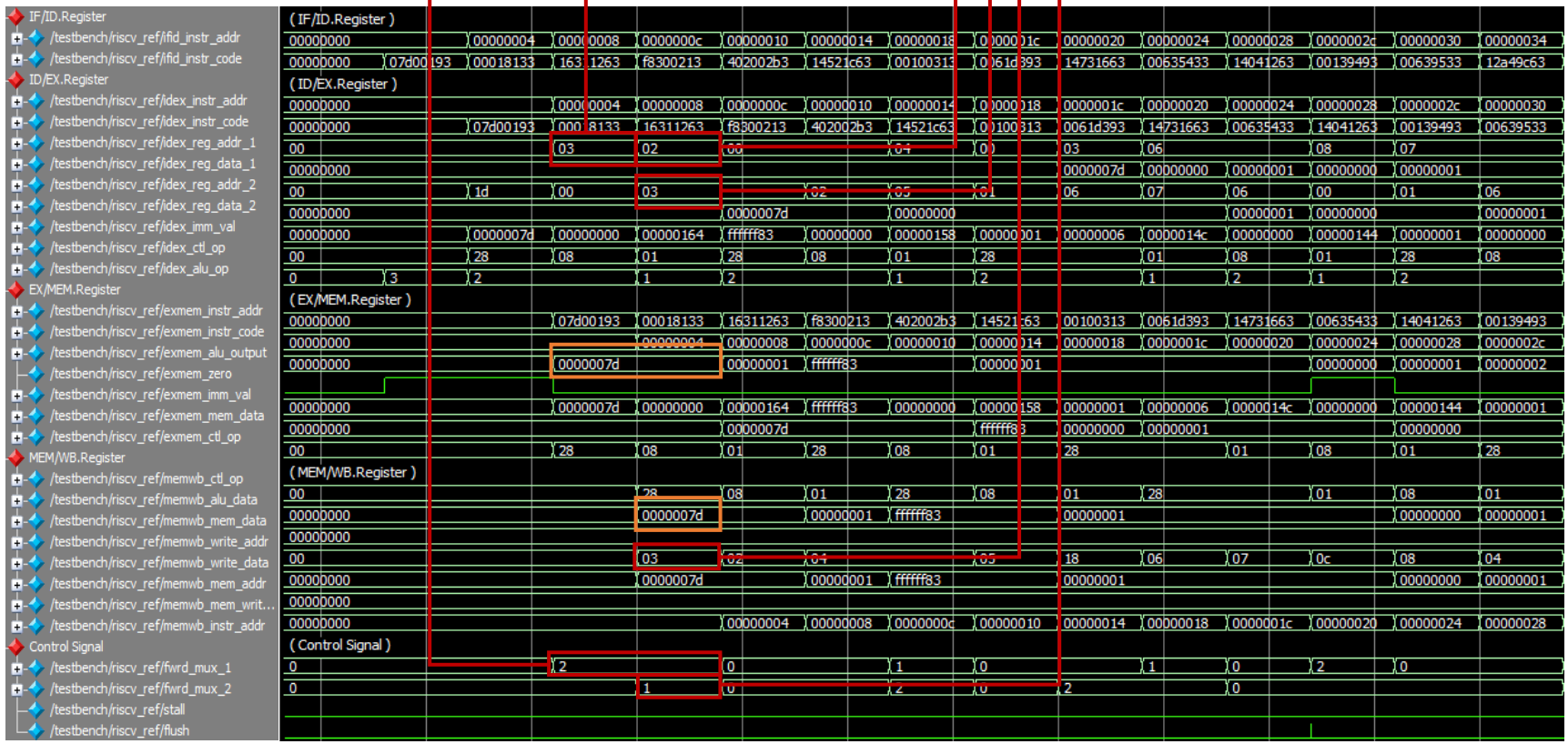


Figure 4.35: Directed Verification Waveform Simulation Results Part 1.

The verification of the functionalities of the instructions can be performed by observing the ALU output and the control signals (*EX_ctl_op*) on the EX/MEM pipeline register.

Zero-width glitches can be observed on the flush control signal. These zero-width glitch signals are caused by static zero hazards. However, as flush signals are only effective if the assertions are held HIGH for the full clock cycle (pipeline registers are only updated on clock edges), these zero-width glitches will not cause issue to the functioning of the pipeline.

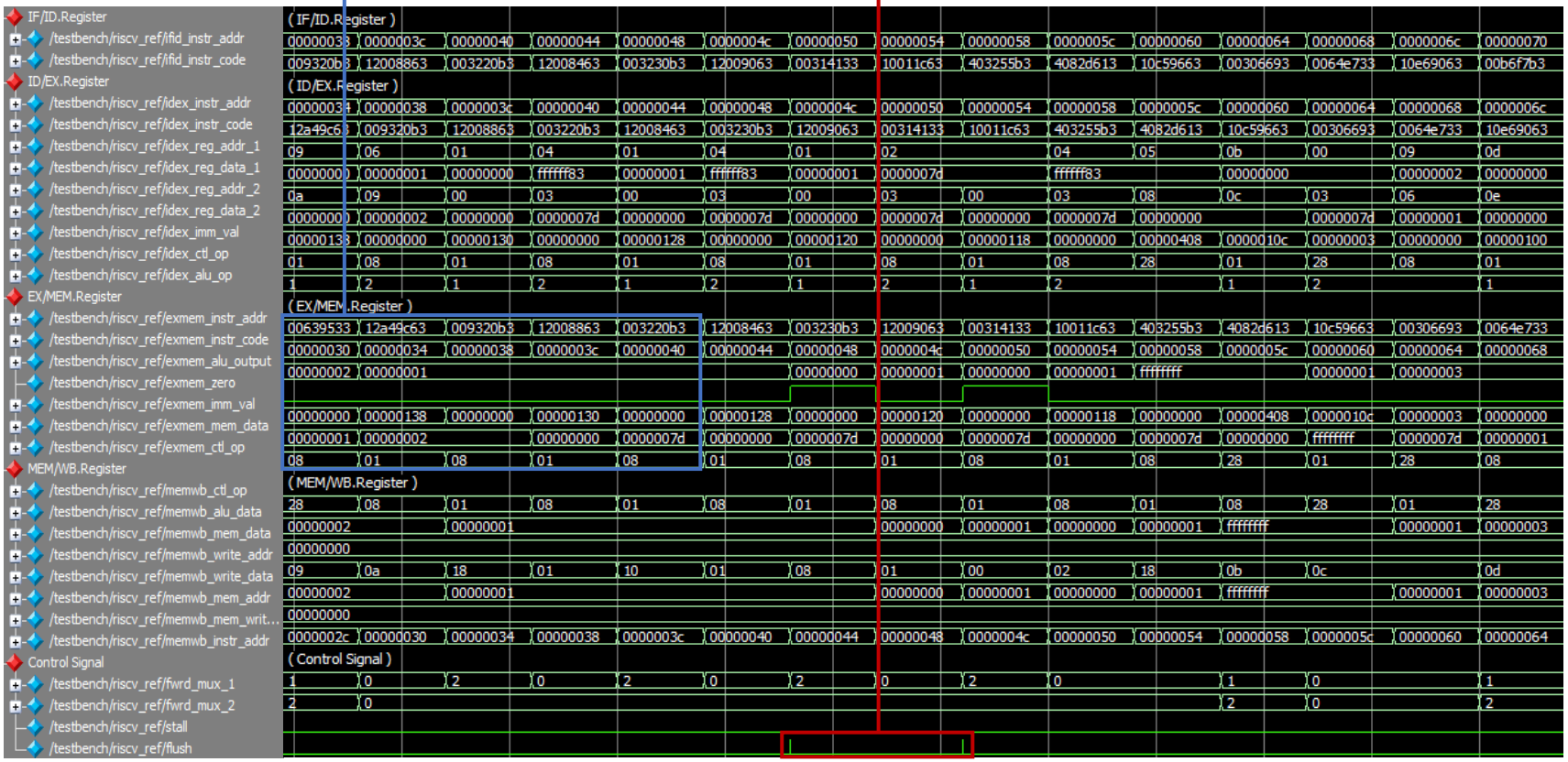


Figure 4.36: Directed Verification Waveform Simulation Results Part 2.

When the jump instruction (*jal x31, STORE_ROUT*) is executed, flush signal is asserted. The assertion of flush signal clears the instruction code stored on the IF/ID and ID/EX pipeline registers. The register *x31* is also updated with the return address when the instruction progresses towards writeback stage (WB). The instruction address of the jump instruction stored on the MEM/WB pipeline register is added by 4 before updating the register *x31*.

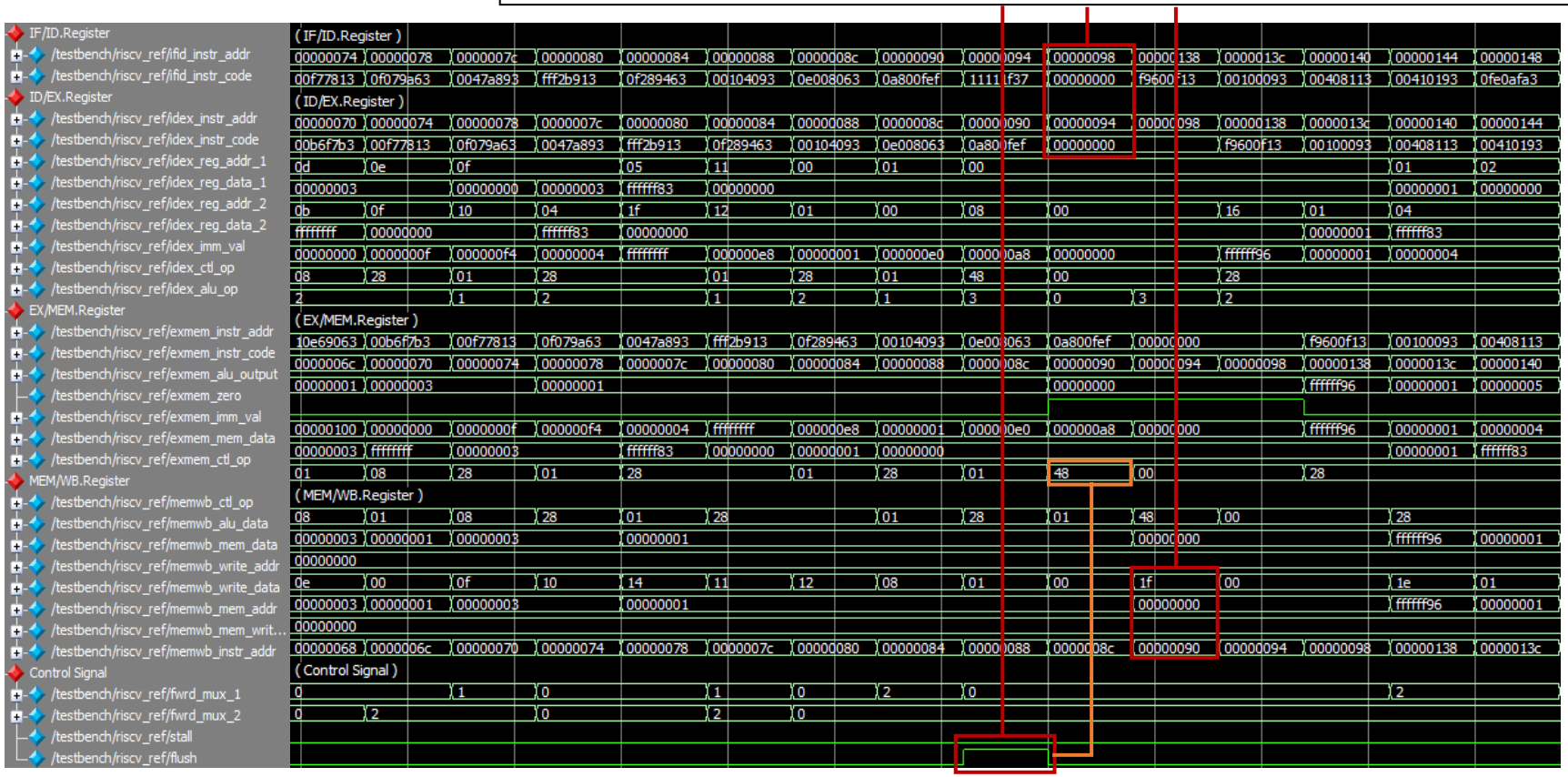


Figure 4.37: Directed Verification Waveform Simulation Results Part 3.

For store instructions, the data stored onto the data memory can be observed on the MEM/WB pipeline register. The store instructions can be checked on the EX/MEM pipeline register (one cycle earlier).

For load instructions, the data read from the data memory can be observed on the MEM/WB pipeline register. Similarly, the load instructions can be checked on the EX/MEM pipeline register (one cycle earlier).

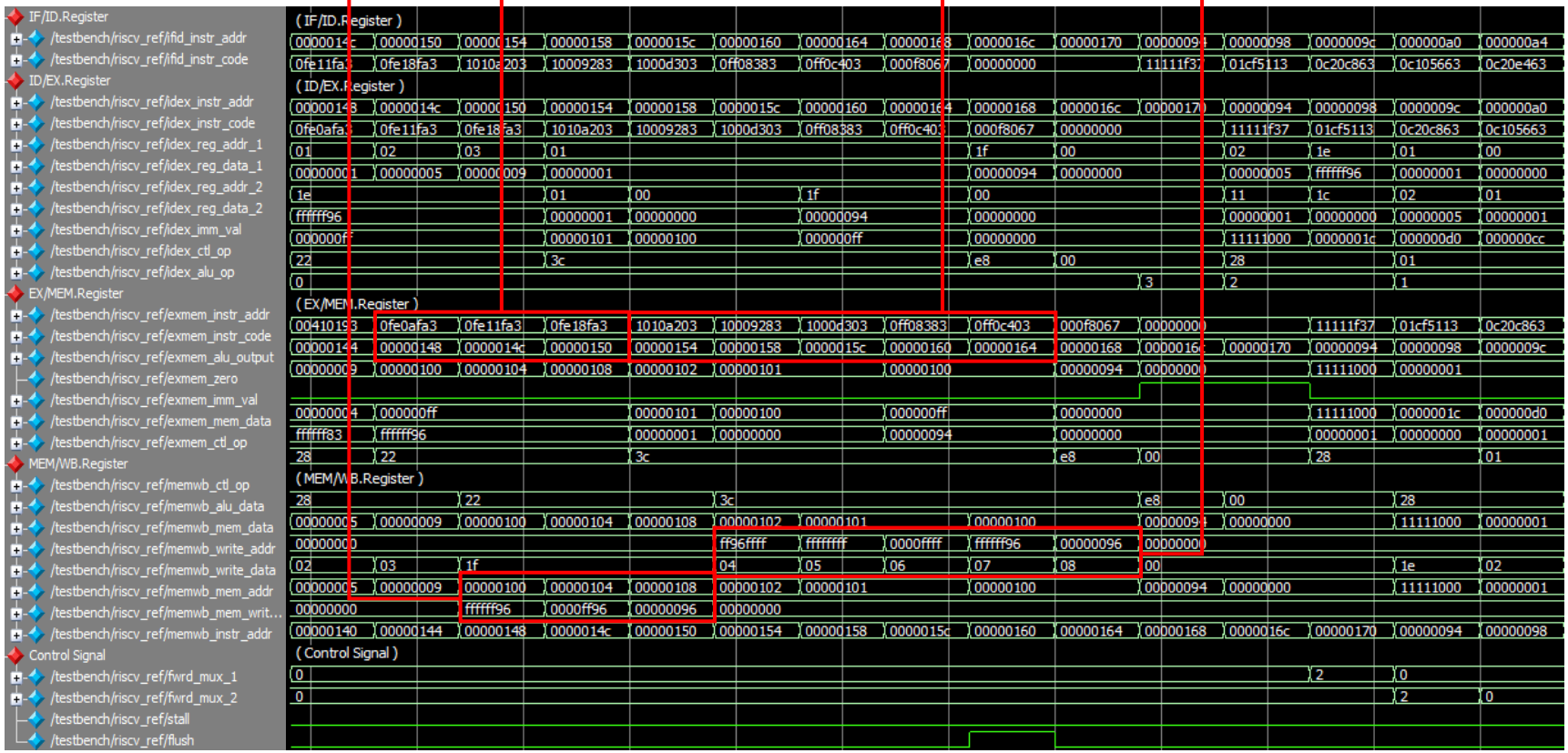


Figure 4.38: Directed Verification Waveform Simulation Results Part 4.

Stall control signal is set to HIGH when load-use case is detected. The destination register *x4* of the load instruction (*lw x4, 257(x1)*) at 0x000000AC is to be used by the following instruction (*addi x14, x4, 0*) at 0x000000B0 which resulted in a load-use case.

The load-use case is resolved when the correct data (from data memory) is forwarded.

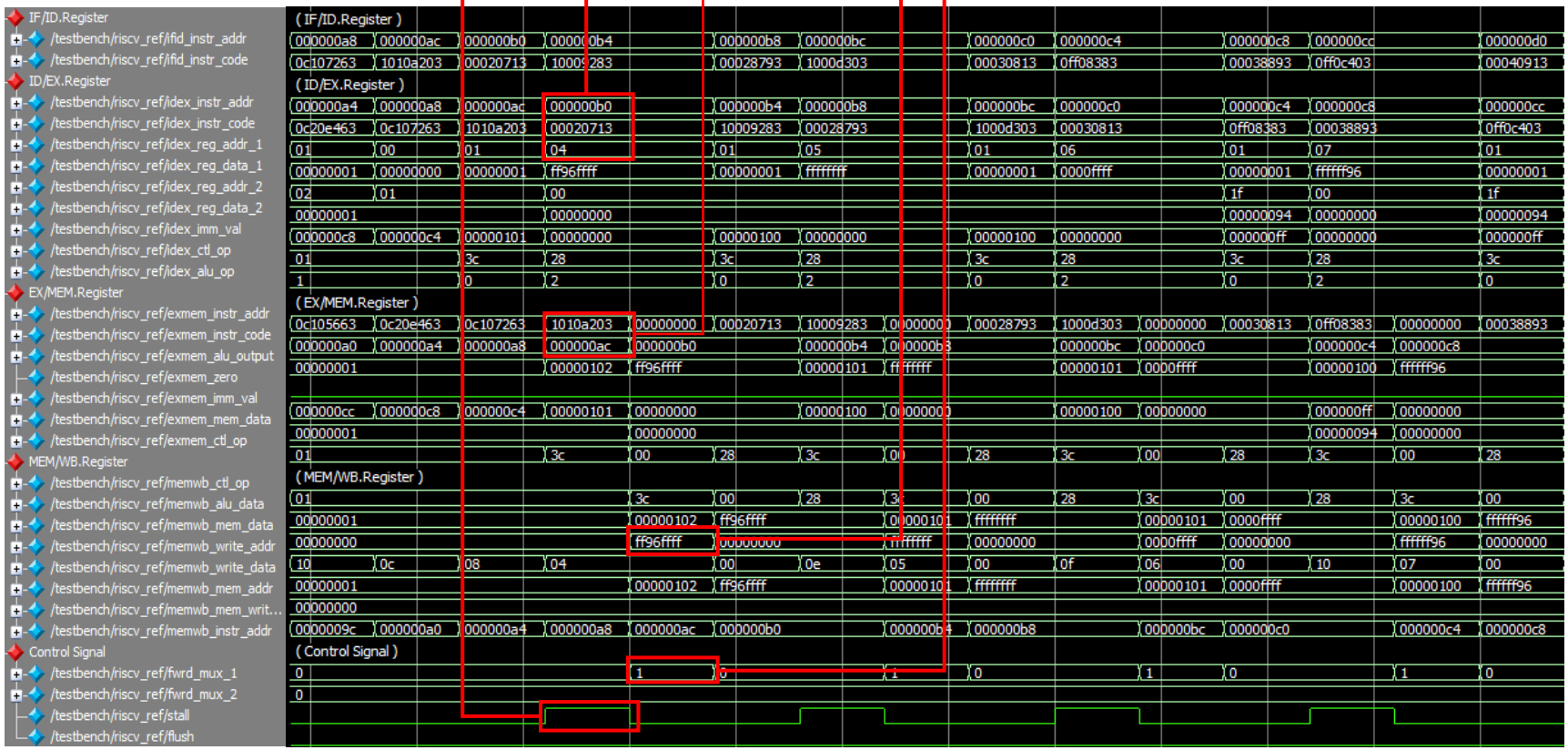


Figure 4.39: Directed Verification Waveform Simulation Results Part 5.

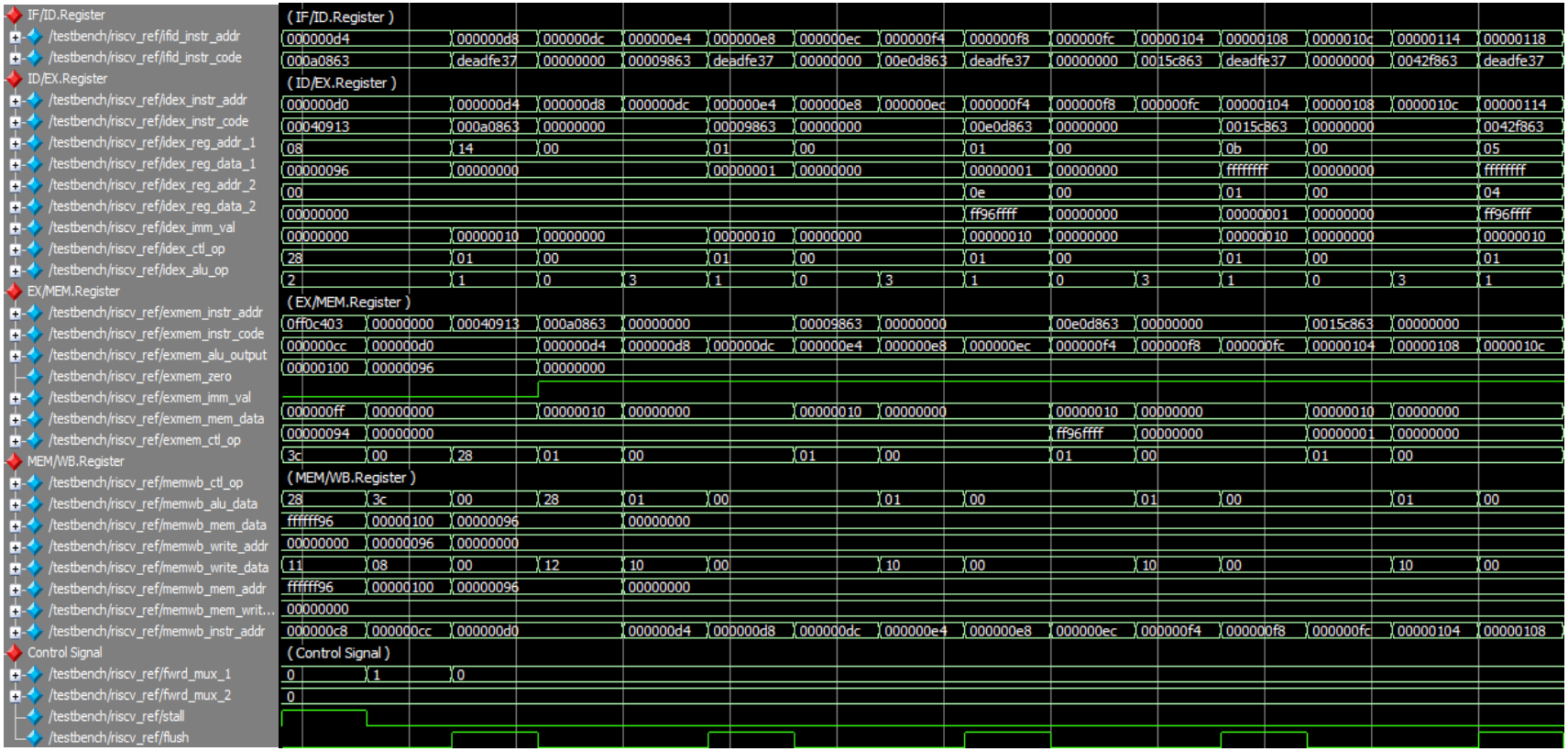


Figure 4.40: Directed Verification Waveform Simulation Results Part 6.

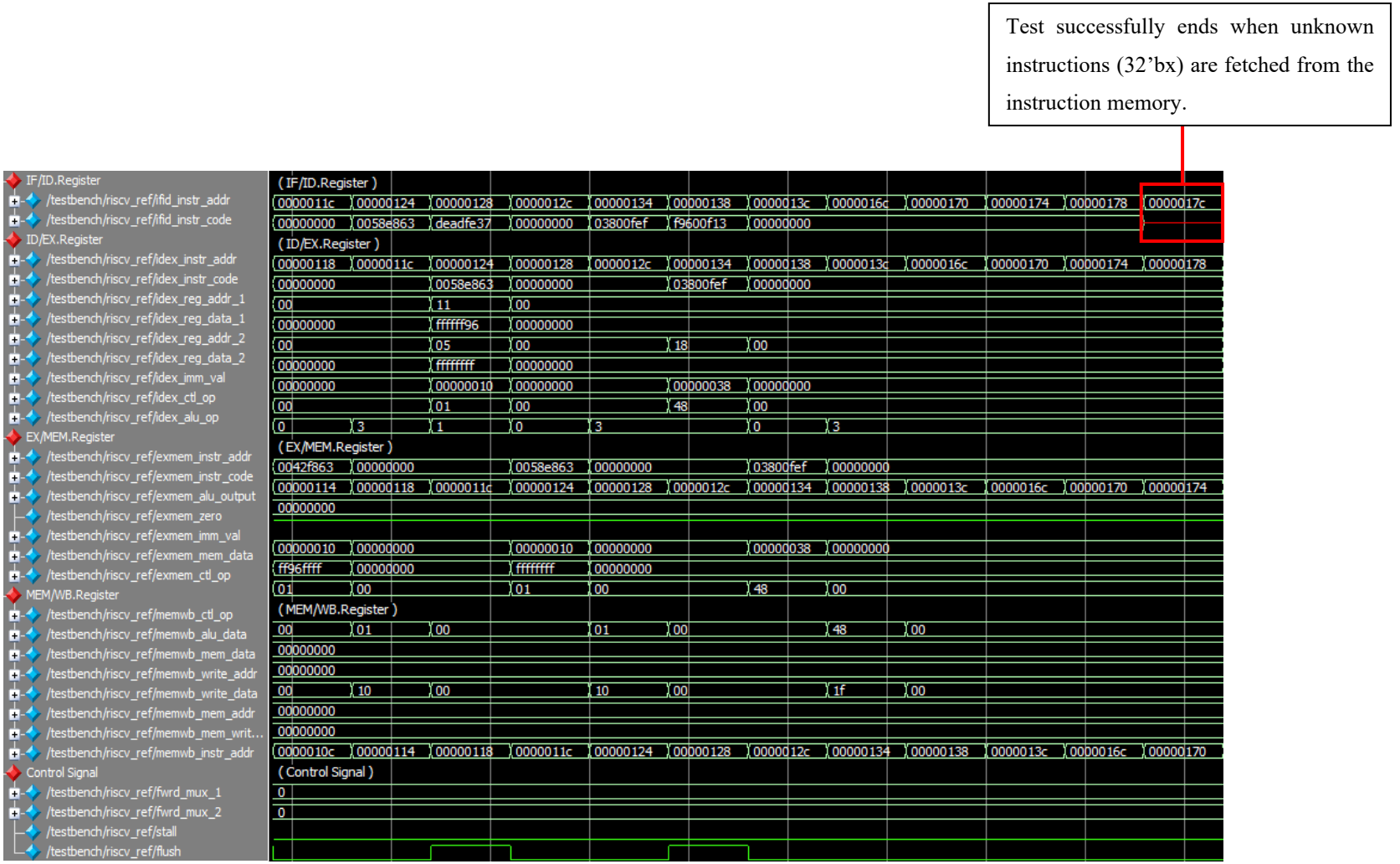


Figure 4.41: Directed Verification Waveform Simulation Results Part 7.

Waveform debugging provides a much thorough analysis and debugging process. In cases where no logical bugs are detected on the outcome, such detailed analysis may be unnecessary. Test log provides a more accessible analysis towards specific internal states of the design, allowing test results analysis and verification to be performed without performing waveform debugging.

From the directed verification and test results analysis performed, the RISC-V pipeline processor is functioning in accordance to the specifications and therefore is accepted.

4.4 Constrained-Random Verification

The following table shows the test cases with specified instruction types utilized for constrained-random verification of the RISC-V processor design:

Table 4.2: Constrained-Random Verification Test Case Specifications.

Test Seeds	R-type	I-type	I-type Load	S-type	SB-type	U-type	UJ-type
3301 to **3320**		/					
3401 to **3420**	/	/					
3501 to **3520**		/	/	/			
3601 to **3620**	/	/	/	/			
3701 to **3720**	/	/	/	/	/	/	
3801 to **3820**	/	/			/	/	
3901 to **3920**							/
2506 to **3006** **0107** to **3107** **0108** to **3108** **0109** to **3009** **0110** to **3110** **0111** to **3011** **0112** to **3112** **0101** to **3101** **0102** to **2802** **0103** to **3103** **0104** to **3004** **0105** to **1005**	/	/	/	/	/	/	/

The test cases with specific instruction types are generated through the following instructions:

```
vsim -gui -onfinish stop work.testbench +TESTLOG +MEMLOG +FORCE_GEN
+INSTR_TYPE=<instruction type> +BATCH_SEED=<seed file>
```

Test cases are generated and stored onto the test repository by listing the test seed values on the <*seed file*> text file and specifying the instruction type to be generated for the test seeds on the field <*instruction type*>.

The generated test seeds are then compiled and listed on "*SEED.txt"*, and the following command is executed:

```
vsim -gui -onfinish stop work.testbench -coverage +TESTLOG +MEMLOG +BATCH_TEST
```

Inclusion of the argument "*-coverage*" allows functional coverage and code coverage analysis to be performed on the simulation executed. Inclusion of the specification "*+BATCH_TEST*" allows all test seeds generated and listed in "*SEED.txt"* to be executed in a batch test. The inclusion of all test seeds allows a comprehensive coverage analysis on the constrained-random verification performed. The simulation is then executed by selecting "Simulate > Run > Run All". Upon completion of the simulation, the following message is generated:

```
** Report counts by severity
UVM_INFO :  925
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[LOGGING]   461
[PASS]   461
[RNTST]     1
[UVM/COMP/NAMECHECK]     1
[UVM/RELNOTES]     1

** Note: $finish    : D:/FinalYearProject/Coding/uvm/uvm_include/base/uvm_root.svh(578)
   Time: 3394850 ns  Iteration: 68  Instance: /testbench
```

Figure 4.42: Simulation Completion Message on ModelSim Transcript.

The following functional coverage analysis provides insight on the functionalities that have been tested using the constrained-random verification:

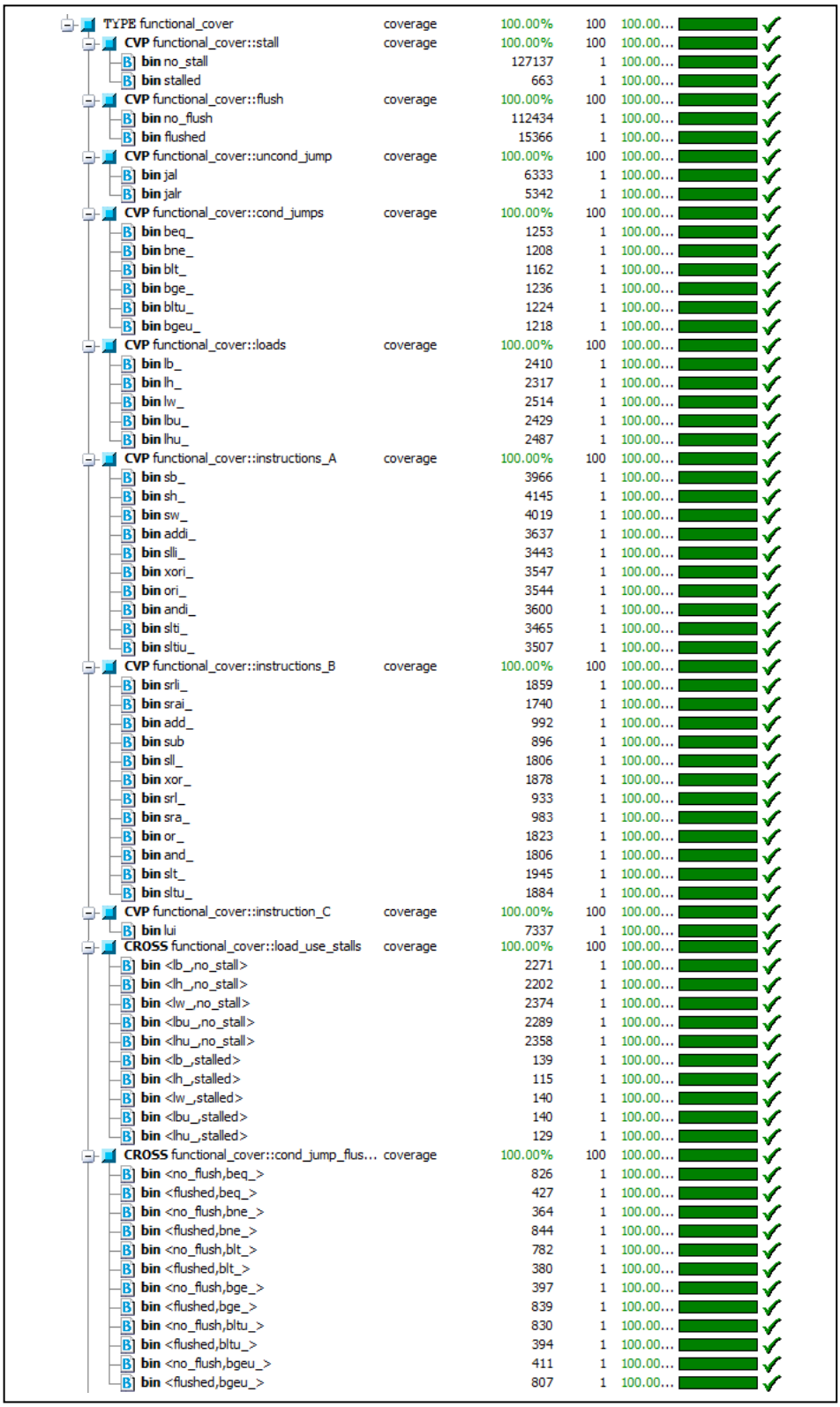


Figure 4.43: Constrained-Random Verification Functional Coverage Report.

The following coverage report provides code coverage analysis which shows source codes on the model designed that has yet to be tested:

```
Coverage Report Summary Data by file

================================================================================
=== File: D:/FinalYearProject/Coding/uvm/ref_model/ref_alu.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                           20        20         0    100.00
    Branches                        39        39         0    100.00
    FEC Condition Terms              8         8         0    100.00
    FEC Expression Terms             1         1         0    100.00

================================================================================
=== File: D:/FinalYearProject/Coding/uvm/ref_model/ref_alu_ctl.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                           32        32         0    100.00
    Branches                        43        36         7     83.72

================================================================================
=== File: D:/FinalYearProject/Coding/uvm/ref_model/ref_ctl_unit.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                           18        18         0    100.00
    Branches                         9         9         0    100.00

================================================================================
=== File: D:/FinalYearProject/Coding/uvm/ref_model/ref_d_mem.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                           12        12         0    100.00
    Branches                        32        31         1     96.87
    FEC Condition Terms             10        10         0    100.00

================================================================================
=== File: D:/FinalYearProject/Coding/uvm/ref_model/ref_exmem_pipeline_reg.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                           17        17         0    100.00
    Branches                        10        10         0    100.00
    FEC Condition Terms              2         2         0    100.00

================================================================================
=== File: D:/FinalYearProject/Coding/uvm/ref_model/ref_fwrd_unit.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                           15        15         0    100.00
    Branches                        15        15         0    100.00
    FEC Condition Terms             13        13         0    100.00

================================================================================
=== File: D:/FinalYearProject/Coding/uvm/ref_model/ref_hzrd_unit.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                            7         7         0    100.00
    Branches                         8         8         0    100.00
    FEC Condition Terms             11        10         1     90.90

================================================================================
=== File: D:/FinalYearProject/Coding/uvm/ref_model/ref_i_mem.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                            2         2         0    100.00
```

```
================================================================================
=== File: D:/FinalYearProject/Coding/uvm/ref_model/ref_idex_pipeline_reg.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                           28        28         0    100.00
    Branches                         5         5         0    100.00
    FEC Condition Terms              2         2         0    100.00

================================================================================
=== File: D:/FinalYearProject/Coding/uvm/ref_model/ref_ifid_pipeline_reg.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                            7         7         0    100.00
    Branches                         4         4         0    100.00
    FEC Condition Terms              2         2         0    100.00

================================================================================
=== File: D:/FinalYearProject/Coding/uvm/ref_model/ref_imm_addr.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                            2         2         0    100.00

================================================================================
=== File: D:/FinalYearProject/Coding/uvm/ref_model/ref_imm_gen.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                           10        10         0    100.00
    Branches                         8         8         0    100.00

================================================================================
=== File: D:/FinalYearProject/Coding/uvm/ref_model/ref_memwb_pipeline_reg.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                           16        16         0    100.00
    Branches                         6         6         0    100.00

================================================================================
=== File: D:/FinalYearProject/Coding/uvm/ref_model/ref_model.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                           30        30         0    100.00
    Branches                        14        12         2     85.71

================================================================================
=== File: D:/FinalYearProject/Coding/uvm/ref_model/ref_pc.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                            5         5         0    100.00
    Branches                         7         7         0    100.00

================================================================================
=== File: D:/FinalYearProject/Coding/uvm/ref_model/ref_reg_file.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                            9         9         0    100.00
    Branches                        20        20         0    100.00
    FEC Condition Terms              4         4         0    100.00


TOTAL COVERGROUP COVERAGE: 100.00%  COVERGROUP TYPES: 1

Total Coverage By File (code coverage only, filtered view): 98.38%
```

Figure 4.44: Coverage Report for RISC-V Constrained-Random Verification.

A total of 461 test seeds have been tested in constrained-random verification. All the specified cover points from the functional coverage report have been covered and tested.

From the code coverage analysis, some branches are not taken in several blocks of the reference model designed: ALU control unit block, memory unit block, and the top module. Upon analysis, these unexercised branches are the default statement for case statements. As the case statements for the reference model design are all assigned a specific value, it is expected for the default statement to be untaken. Aside from that, there is a miss from the focused expression condition coverage from the hazard unit block. Upon inspection, this condition miss is due to the condition of ensuring the write register is not register *x0* as shown in the following figure:

```
Condition Coverage for:
((mem_read && ((idex_reg_addr_1 == exmem_rd) || (idex_reg_addr_2 == exmem_rd))) && (exmem_rd != 0))
FEC Coverage:  3 out of 4 input terms covered = 75.00%

      Input Terminal               Covered  Reason         Hint
      ---------------------------- -------  ------------   --------
      mem_read                     Y
      (idex_reg_addr_1 == exmem_rd) Y
      (idex_reg_addr_2 == exmem_rd) Y
      (exmem_rd != 0)              N        '_0' not hit   Hit '_0'
```

Figure 4.45: Focused Expression Condition Coverage Miss Analysis.

As writes to register *x0* has been disabled in the test generator, this condition is not checked. However, as RISC-V instruction set architecture specifies writes to register *x0* are inhibited, the coverage miss can be dismissed as it serves as a precautionary condition. With the justifications provided, it can be stated that the reference model designed has been adequately tested.

Even though constrained-random verification contributes greater efficiency towards achieving full functional coverage, the vastly randomized instruction flow may miss out on corner cases that require specific instruction sequences. Developing a better instruction sequence generation algorithm will be needed to test such corner cases with specific instruction sequences. Alternatively, these corner cases can be manually written and tested.

Chapter 5

CONCLUSION AND RECOMMENDATIONS

5.1 Conclusion

Functional verification of a **pipelined RISC-V processor** has been successfully performed using **Universal Verification Methodology** (UVM). UVM is a testbench architecture that emphasizes a standardized and reusable approach towards a verification environment. The verification subject of this project is a RISC-V processor with a 5-stage pipeline implementation, with **data forwarding**, **pipeline stalling**, and **pipeline flushing** implemented.

The reference model, design under test, and testbench environment have been modelled using **SystemVerilog** and are simulated using **ModelSim**. **Directed verification** and **constrained-random** verification are the two main forms of verification performed. Both directed and constrained-random verification have been integrated into the UVM verification environment **translation** and **specification** tasks in the UVM sequence item class. Through the UVM configuration database, the user can select the intended verification form through the command line argument "+DIRECTED_TEST". Directed verification provided a much more well-planned test case scenario. In contrast, constrained-random verification proves to be a much more efficient approach towards achieving full functional coverage. The UVM verification environment has also introduced **regression testing** capability through programmed capabilities to store test cases and perform multiple test case testing in one simulation via command-line argument "+BATCH_TEST". These introduced capabilities

assisted with the verification process through **shorter simulation time**, **consistent test case reproduction**, and **cumulative coverage collection** for multiple test cases.

For the functional verification of the reference model, a **self-checking mechanism** has been introduced in the UVM scoreboard component. The self-checking mechanism performs functional verification for major design functionalities, including pipeline stalling, pipeline flushing, and implemented instruction functionality. Intentional bugs have also been introduced in the design under test and reference model to test out the bug detection capability of the verification environment. The simulation results for self-checking mechanism testing and bug detection capability testing have been compiled and explained. For directed verification, a sample program has been written in assembly language. The written program is translated to machine language and simulated. The simulation results are compiled and analysed for any logical errors. For constrained-random verification, various test seeds with varying specifications have been generated and tested. The compiled results can be found in **Chapter 4**.

From the results compiled, **code coverage** is at **98.38%**, whereas **functional coverage** is at **100%**. The unexercised code is due to default cases for several case statements, whereby all expected case statements have been appropriately assigned. The project is said to have been completed with sufficient functional verification performed as indicated by the full functional coverage and high code coverage.

5.2 Recommendations

For future enhancement of the project, several recommendations can be made. The utilized simulation verification performs lock-step comparison between the reference model and design under test. It is suggested that a **reference model from Imperas can be used for the lock-step comparison** to further **enhance the confidence in the verification performed** due to the maturity of the Imperas RISC-V reference model. Usage of a high confidence reference model would remove the requirement of a self-checking mechanism, allowing more effort to be placed on other verification

components. The utilization of a reference model would also offer a learning opportunity for verification intellectual property (VIP) interfacing and usage.

Aside from the utilization of a higher confidence reference model, another recommendation that can be made is the **fragmentation of the verification process**. The verification methodology focuses on **chip level verification** whereby verification is performed on the RISC-V processor as a whole. Fragmenting the verification process to several levels such as **unit level verification**, **block level verification**, and **chip level verification** can ease the overall verification process, especially when the verification subject is a complex system. Verification at lower levels can place more emphasis on functionality correctness of the unit whereas verification at higher level can place more emphasis on overall functionality correctness and interconnection of the lower level components.

Another future enhancement of the project would be to **complexify the functional coverage criteria**. In this project, the functional coverage criteria include pipeline stalling, pipeline flushing, and the instructions executed. The lack of complex functional coverage cover point or cross cover point makes it easy to achieve full functional coverage. A well-planned functional coverage would **allow more complex test case scenarios** to be included in the test plan, leading to a better verification. A complexified functional coverage criteria would also push forward the necessity for a complexified test generator algorithm. The test generator mainly used for constrained-random verification provides randomized test cases with valid random instructions. A recommendation that can be made is to include a **better algorithm for instruction generation** that results in a **sensible instruction flow**.

Lastly, **formal property verification** can also be included for specific properties of the RISC-V microprocessor architecture, such as pipeline stalling and pipeline flushing. Compared to the unconventional approach taken (self-checking mechanism) for assertion checks of pipeline stalls and pipeline flushes, standardized SystemVerilog assertions provide a much more comprehensible approach. The inclusion of formal property verification would also provide learning and practical opportunity for SystemVerilog assertions (SVA), a widely used verification methodology in the industry.

REFERENCES

Ackland, B. and Weste, N. (1981). *Functional verification in an interactive symbolic IC design environment.*

Adir, A., Copty, S., Landa, S., Nahir, A., Shurek, G., Ziv, A., Meissner, C. and Schumann, J. (2011). *A unified methodology for pre-silicon verification and post-silicon validation*. Design, Automation & Test in Europe, pp. 1-6.

Andrew, W., Krste, A., and SiFive Inc., 2017. *The RISC-V Instruction Set Manual Volume I: User Level ISA.*

Chauhan, K., 2020. *ASIC Design Flow in VLSI Engineering Services – A Quick Guide*. [online] Available at: https://www.einfochips.com/blog/asic-design-flow-in-vlsi-engineering-services-a-quick-guide/ [Accessed 26 July 2021].

Chen, S.J. and Cheng, C.K. (2000). *Tutorial on VLSI partitioning*. VLSI design, 11(3), pp.175-218.

Chip Verify, n.d. *SystemVerilog Tutorial*. [online] Available at: https://www.chipverify.com/systemverilog/systemverilog-tutorial [Accessed 30 July 2021].

Doulos, n.d. *What is Verilog?* [online] Available at: https://www.doulos.com/httpswwwdouloscomknowhow/verilog/what-is-verilog/ [Accessed 30 July 2021].

Francesconi, J., Rodriguez, J.A. and Julian, P.M. (2014). *UVM based testbench architecture for unit verification*. Argentine Conference on Micro-Nanoelectronics, Technology and Applications (EAMTA), pp. 89-94.

Kaeslin, H. (2014). *Top-down digital VLSI design: from architectures to gate-level circuits and FPGAs*. Morgan Kaufmann. pp. 301 – 357.

Lam, W.K. (2005). *Hardware design verification: simulation and formal method-based approaches*. Prentice Hall Professional Technical Reference.

Lavagno, L., Scheffer, L. and Martin, G. eds. (2018). *EDA for IC implementation, circuit design, and process technology*. CRC press. pp. 635 – 685.

Law, A. M., 2008. *How to Build Valid and Credible Simulation Models*. [online] Available at: https://www.informs-sim.org/wsc08papers/007.pdf [Accessed 14 July 2021].

Ledin, J. (2020). *Modern Computer Architecture and Organization*. Packt Publishing.

Monteiro, J. and Van Leuken, R. eds. (2010). *Integrated Circuit and System Design: Power and Timing Modeling, Optimization and Simulation: 19th International Workshop, PATMOS 2009, Delft, The Netherlands, September 9-11, 2009, Revised Selected Papers (Vol. 5953).* Springer Science & Business Media. pp. 511 – 529.

Nuwen, P., 2020. *What is RISC-V? 10 Things You Should Know About RISC-V in 2020.* [online] Available at: https://www.seeedstudio.com/blog/2020/06/01/10-things-you-should-know-about-risc-v-in-2020-m/ [Accessed 14 July 2021].

Oski Technology, 2020. *Formal vs. Simulation Testbenches: Architecting for End-to-End Verification.* [online] Available at: https://www.oskitechnology.com/formal-vs-simulation-testbenches-architecting-for-end-to-end-verification/ [Accessed 24 December 2021].

Patterson, D. and Hennessy, J. eds. (2017). *Computer Organization and Design. RISC-V Edition*. Morgan Kaufmann, Burlington. pp. 234 – 307.

Saint, C., 2020. *Integrated Circuit.* [online] Available at: https://www.britannica.com/technology/integrated-circuit [Accessed 26 July 2021].

Semiconductor Engineering, n.d. *Physical Design.* [online] Available at: https://semiengineering.com/knowledge_centers/eda-design/definitions/physical-design/ [Accessed 26 July 2021].

Shah, A., 2013. *Keeping up with Moore's Law is getting harder.* [online] Available at: https://www.computerworld.com/article/2497175/keeping-up-with-moore-s-law-is-getting-harder.html [Accessed 28th July 2021].

Shen, J. and Abraham, J.A. (1999). *Verification of processor microarchitectures.* Proceedings 17th IEEE VLSI Test Symposium (Cat. No. PR00146), pp. 189-194.

Singhal, M., 2015. *Introduction about Advanced Functional Verification.* [online] Available at: https://www.learnuvmverification.com/index.php/2015/07/04/187/ [Accessed 30 July 2021].

Synopsys, n.d. *What is IC Design?* [online] Available at: https://www.synopsys.com/glossary/what-is-ic-design.html [Accessed 26 July 2021].

Tech Design Forum, n.d. *Assertion-based verification.* [online] Available at: https://www.techdesignforums.com/practice/guides/guide-to-assertion-based-verification/ [Accessed 30 July 2021].

The University of Texas at Dallas, 2011. *Modern ASIC Design – Architectural Design* [online] Available at: https://personal.utdallas.edu/~zhoud/EE6306/lecture%20slides/ASIC%202011%20Chapter%204%20Architecture%20Design.pdf [Accessed 26 July 2021].

Tranter, J., 2021. *What is RISC-V and Why is it Important?* [online] Available at: https://www.ics.com/blog/what-risc-v-and-why-it-important [Accessed 14 July 2021].

Vij, V. (2013). *VLSI Design Theory and Practice.* Laxmi Publications. pp. 8 – 12.

Wang, L.T., Chang, Y.W. and Cheng, K.T.T. eds. (2009). *Electronic design automation: synthesis, verification, and test*. Morgan Kaufmann. pp. 14 – 17.

APPENDICES

APPENDIX A: Source Code of Reference Model – Top Module

```
`include "parameter_list.sv"
`ifndef ref_model
        `define ref_model
module  ref_model (virtual_interface interface_instance);

        logic   [`INST_ADDR_WIDTH-1:0]  instr_addr;
        logic   [`INST_CODE_WIDTH-1:0]  instr_code;
        logic   [`REG_ADDR_WIDTH-1:0]   reg_read_addr_1;
        logic   [`DATA_WIDTH-1:0]       reg_data_1;
        logic   [`REG_ADDR_WIDTH-1:0]   reg_read_addr_2;
        logic   [`DATA_WIDTH-1:0]       reg_data_2;
        logic   [`DATA_WIDTH-1:0]       imm_val;
        logic   [`INST_ADDR_WIDTH-1:0]  imm_addr;
        logic   [`CTL_SGNL_WIDTH-1:0]   ctl_op;
        logic   [`ALU_OP_WIDTH-1:0]     alu_op;
        logic   [`ALU_CTL_WIDTH-1:0]    alu_ctl;
        logic   [`DATA_WIDTH-1:0]       alu_output;
        logic                           alu_zero;
        logic   [`DATA_WIDTH-1:0]       loaded_data;
        logic   [`DATA_WIDTH-1:0]       stored_data;
        logic   [`FWRD_MUX_WIDTH-1:0]   fwrd_mux_1;
        logic   [`FWRD_MUX_WIDTH-1:0]   fwrd_mux_2;
        logic                           stall;
        logic                           flush;

        logic   [`INST_ADDR_WIDTH-1:0]  ifid_instr_addr;
        logic   [`INST_CODE_WIDTH-1:0]  ifid_instr_code;
        logic   [`INST_ADDR_WIDTH-1:0]  idex_instr_addr;
        logic   [`INST_CODE_WIDTH-1:0]  idex_instr_code;
        logic   [`REG_ADDR_WIDTH-1:0]   idex_reg_addr_1;
        logic   [`DATA_WIDTH-1:0]       idex_reg_data_1;
        logic   [`REG_ADDR_WIDTH-1:0]   idex_reg_addr_2;
        logic   [`DATA_WIDTH-1:0]       idex_reg_data_2;
        logic   [`DATA_WIDTH-1:0]       idex_imm_val;
        logic   [`CTL_SGNL_WIDTH-1:0]   idex_ctl_op;
        logic   [`ALU_OP_WIDTH-1:0]     idex_alu_op;
        logic   [`INST_ADDR_WIDTH-1:0]  exmem_instr_addr;
        logic   [`INST_CODE_WIDTH-1:0]  exmem_instr_code;
        logic   [`DATA_WIDTH-1:0]       exmem_alu_output;
        logic                           exmem_zero;
        logic   [`DATA_WIDTH-1:0]       exmem_imm_val;
        logic   [`DATA_WIDTH-1:0]       exmem_mem_data;
        logic   [`CTL_SGNL_WIDTH-1:0]   exmem_ctl_op;
        logic   [`CTL_SGNL_WIDTH-1:0]   memwb_ctl_op;
        logic   [`DATA_WIDTH-1:0]       memwb_alu_data;
        logic   [`DATA_WIDTH-1:0]       memwb_mem_data;
        logic   [`REG_ADDR_WIDTH-1:0]   memwb_write_addr;
        logic   [`DATA_WIDTH-1:0]       memwb_write_data;
        logic   [`DATA_WIDTH-1:0]       memwb_mem_addr;
        logic   [`DATA_WIDTH-1:0]       memwb_mem_write_data;
        logic   [`INST_ADDR_WIDTH-1:0]  memwb_instr_addr;

        logic   [`DATA_REG_WIDTH-1:0]   rom     [0:((2**`MEM_ROWS) - 1)];

        ref_pc                          REF_PC          (.clock(interface_instance.clk),
                                                         .reset(interface_instance.reset),
                                                         .stall(stall),
                                                         .flush(flush),
                                                         .imm_addr(imm_addr),
                                                         .alu_output(alu_output),
                                                         .jump_reg(idex_ctl_op[`JUMP_REG]),
                                                         .instr_addr(instr_addr));
```

```
ref_i_mem                   REF_I_MEM       (.reset(interface_instance.reset),
                                             .rom(rom),
                                             .instr_addr(instr_addr),
                                             .instr_code(instr_code));

ref_ifid_pipeline_reg       REF_IF          (.clock(interface_instance.clk),
                                             .reset(interface_instance.reset),
                                             .stall(stall),
                                             .flush(flush),
                                             .instr_addr(instr_addr),
                                             .instr_code(instr_code),
                                             .ifid_addr(ifid_instr_addr),
                                             .ifid_instr(ifid_instr_code));

ref_reg_file                REF_REG         (.clock(interface_instance.clk),
                                             .reset(interface_instance.reset),
                                             .instr_code(ifid_instr_code),
                                             .reg_write(memwb_ctl_op[`REG_WRITE]),
                                             .reg_write_addr(memwb_write_addr),
                                             .reg_write_data(memwb_write_data),
                                             .reg_read_addr_1(reg_read_addr_1),
                                             .reg_data_1(reg_data_1),
                                             .reg_read_addr_2(reg_read_addr_2),
                                             .reg_data_2(reg_data_2));

ref_imm_gen                 REF_IMM_GEN     (.instr_code(ifid_instr_code),
                                             .imm_val(imm_val));

ref_imm_addr_unit           REF_IMM_ADDR    (.instr_addr(idex_instr_addr),
                                             .imm_val(idex_imm_val),
                                             .imm_addr(imm_addr));

ref_ctl_unit                REF_CTL         (.instr_code(ifid_instr_code),
                                             .ctl_op(ctl_op),
                                             .alu_op(alu_op));

ref_idex_pipeline_reg       REF_ID          (.clock(interface_instance.clk),
                                             .reset(interface_instance.reset),
                                             .stall(stall),
                                             .flush(flush),
                                             .instr_code(ifid_instr_code),
                                             .instr_addr(ifid_instr_addr),
                                             .reg_addr_1(reg_read_addr_1),
                                             .reg_data_1(reg_data_1),
                                             .reg_addr_2(reg_read_addr_2),
                                             .reg_data_2(reg_data_2),
                                             .imm_val(imm_val),
                                             .ctl_op(ctl_op),
                                             .alu_op(alu_op),
                                             .idex_instr_code(idex_instr_code),
                                             .idex_instr_addr(idex_instr_addr),
                                             .idex_imm_val(idex_imm_val),
                                             .idex_ctl_op(idex_ctl_op),
                                             .idex_alu_op(idex_alu_op),
                                             .idex_reg_addr_1(idex_reg_addr_1),
                                             .idex_reg_data_1(idex_reg_data_1),
                                             .idex_reg_addr_2(idex_reg_addr_2),
                                             .idex_reg_data_2(idex_reg_data_2));

ref_alu_ctl                 REF_ALU_CTL     (.alu_op(idex_alu_op),
                                             .instr_code(idex_instr_code),
                                             .alu_ctl(alu_ctl));

ref_alu                     REF_ALU         (.mem_to_reg(memwb_ctl_op[`MEM_TO_REG]),
                                             .reg_data_1(idex_reg_data_1),
                                             .reg_data_2(idex_reg_data_2),
                                             .imm_val(idex_imm_val),
                                             .alu_ctl(alu_ctl),
                                             .alu_src(idex_ctl_op[`ALU_SRC]),
                                             .fwrd_mux_1(fwrd_mux_1),
                                             .fwrd_mux_2(fwrd_mux_2),
                                             .exmem_alu_data(exmem_alu_output),
                                             .memwb_alu_data(memwb_alu_data),
                                             .memwb_mem_data(memwb_mem_data),
                                             .alu_output(alu_output),
                                             .alu_zero(alu_zero));

ref_exmem_pipeline_reg      REF_EX          (.clock(interface_instance.clk),
                                             .reset(interface_instance.reset),
                                             .alu_zero(alu_zero),
                                             .stall(stall),
                                             .flush(flush),
                                             .mem_to_reg(memwb_ctl_op[`MEM_TO_REG]),
                                             .imm_val(idex_imm_val),
                                             .fwrd_mux_2(fwrd_mux_2),
                                             .memwb_mem_data(memwb_mem_data),
                                             .memwb_alu_data(memwb_alu_data),
                                             .idex_mem_data(idex_reg_data_2),
                                             .alu_output(alu_output),
                                             .instr_code(idex_instr_code),
                                             .instr_addr(idex_instr_addr),
```

```
                                        .ctl_op(idex_ctl_op),
                                        .exmem_instr_code(exmem_instr_code),
                                        .exmem_instr_addr(exmem_instr_addr),
                                        .exmem_mem_data(exmem_mem_data),
                                        .exmem_alu_output(exmem_alu_output),
                                        .exmem_zero(exmem_zero),
                                        .exmem_imm_val(exmem_imm_val),
                                        .exmem_ctl_op(exmem_ctl_op));

ref_d_mem                   REF_D_MEM   (.clock(interface_instance.clk),
                                        .reset(interface_instance.reset),
                                        .mem_write(exmem_ctl_op[`MEM_WRITE]),
                                        .mem_read(exmem_ctl_op[`MEM_READ]),
                                        .instr_code(exmem_instr_code),
                                        .address(exmem_alu_output[`MEM_ADDR_WIDTH-1:0]),
                                        .reg_data(exmem_mem_data),
                                        .loaded_data(loaded_data),
                                        .stored_data(stored_data));

ref_memwb_pipeline_reg      REF_WB      (.clock(interface_instance.clk),
                                        .reset(interface_instance.reset),
                                        .alu_output(exmem_alu_output),
                                        .ctl_op(exmem_ctl_op),
                                        .loaded_data(loaded_data),
                                        .stored_data(stored_data),
                                        .rd(exmem_instr_code[`RD_ADDR_HI:`RD_ADDR_LO]),
                                        .instr_addr(exmem_instr_addr),
                                        .memwb_ctl_op(memwb_ctl_op),
                                        .memwb_write_addr(memwb_write_addr),
                                        .memwb_write_data(memwb_write_data),
                                        .memwb_mem_data(memwb_mem_data),
                                        .memwb_alu_data(memwb_alu_data),
                                        .memwb_mem_addr(memwb_mem_addr),
                                        .memwb_mem_write_data(memwb_mem_write_data),
                                        .memwb_instr_addr(memwb_instr_addr));

ref_fwrd_unit               REF_FWRD    (.reset(interface_instance.reset),
                                        .exmem_reg_write(exmem_ctl_op[`REG_WRITE]),
                                        .memwb_reg_write(memwb_ctl_op[`REG_WRITE]),
                                        .stall(stall),
                                        .exmem_rd(exmem_instr_code[`RD_ADDR_HI:`RD_ADDR_LO]),
                                        .memwb_rd(memwb_write_addr),
                                        .exmem_opcode(exmem_instr_code[`OPCODE_HI:`OPCODE_LO]),
                                        .reg_1(idex_reg_addr_1),
                                        .reg_2(idex_reg_addr_2),
                                        .fwrd_mux_1(fwrd_mux_1),
                                        .fwrd_mux_2(fwrd_mux_2));

ref_hzrd_unit               REF_HZRD    (.reset(interface_instance.reset),
                                        .mem_read(exmem_ctl_op[`MEM_READ]),
                                        .zero(alu_zero),
                                        .branch(idex_ctl_op[`BRANCH]),
                                        .jump(idex_ctl_op[`JUMP_LINK]),
                                        .exmem_rd(exmem_instr_code[`RD_ADDR_HI:`RD_ADDR_LO]),
                                        .idex_reg_addr_1(idex_reg_addr_1),
                                        .idex_reg_addr_2(idex_reg_addr_2),
                                        .fwrd_mux_1(fwrd_mux_1),
                                        .fwrd_mux_2(fwrd_mux_2),
                                        .stall(stall),
                                        .flush(flush));

// Drive signals to the testbench for further verification
always @(posedge interface_instance.clk) begin: interface_block
                interface_instance.ref_stall <= stall;
                interface_instance.ref_flush <= flush;
                interface_instance.ref_jump_link <= idex_ctl_op[`JUMP_LINK];
                if(!stall) begin
                        interface_instance.ref_pc <= ifid_instr_addr;
                        interface_instance.ref_instr <= ifid_instr_code;
                end
                interface_instance.ref_reg_read_addr_1 <= idex_reg_addr_1;
                interface_instance.ref_reg_read_addr_2 <= idex_reg_addr_2;
                case(fwrd_mux_1)
                `FWRD_ALU:      interface_instance.ref_reg_read_data_1 <= exmem_alu_output;
                `FWRD_MEM:      begin
                        if(memwb_ctl_op[`MEM_TO_REG])
                                interface_instance.ref_reg_read_data_1 <= memwb_mem_data;
                        else
                                interface_instance.ref_reg_read_data_1 <= memwb_alu_data;
                        end
                `NO_FWRD:       interface_instance.ref_reg_read_data_1 <= idex_reg_data_1;
                endcase
                case(fwrd_mux_2)
                `FWRD_ALU:      interface_instance.ref_reg_read_data_2 <= exmem_alu_output;
                `FWRD_MEM:      begin
                        if(memwb_ctl_op[`MEM_TO_REG])
                                interface_instance.ref_reg_read_data_2 <= memwb_mem_data;
                        else
                                interface_instance.ref_reg_read_data_2 <= memwb_alu_data;
                        end
                `NO_FWRD:       interface_instance.ref_reg_read_data_2 <= idex_reg_data_2;
```

```
                        endcase
                        case(fwrd_mux_2)
                        `FWRD_ALU:      interface_instance.ref_reg_read_data_2 <= exmem_alu_output;
                        `FWRD_MEM:      begin
                                if(memwb_ctl_op[`MEM_TO_REG])
                                        interface_instance.ref_reg_read_data_2 <= memwb_mem_data;
                                else
                                        interface_instance.ref_reg_read_data_2 <= memwb_alu_data;
                                end
                        `NO_FWRD:       interface_instance.ref_reg_read_data_2 <= idex_reg_data_2;
                        endcase
                        interface_instance.ref_imm_val <= idex_imm_val;
                        interface_instance.ref_alu_output <= exmem_alu_output;
                        interface_instance.ref_alu_zero <= exmem_zero;
                        interface_instance.ref_ctl_op <= exmem_ctl_op;
                        interface_instance.ref_reg_write_addr <= memwb_write_addr;
                        interface_instance.ref_mem_addr <= memwb_mem_addr;
                        interface_instance.ref_mem_write_data <= memwb_mem_write_data;
                        interface_instance.ref_mem_write <= memwb_ctl_op[`MEM_WRITE];
                        interface_instance.ref_mem_read <= memwb_ctl_op[`MEM_READ];
                        interface_instance.ref_reg_write <= memwb_ctl_op[`REG_WRITE];
                        interface_instance.ref_reg_write_data <= memwb_write_data;

                        // Delayed alignment instruction executions for coverage checking
                        interface_instance.ref_ID_instr <= idex_instr_code;
                        interface_instance.ref_EX_instr <= exmem_instr_code;
                        interface_instance.ref_EX_pc <= exmem_instr_addr;
        end: interface_block
endmodule
`endif
```

APPENDIX B: Source Code of Reference Model – Program Counter

```
`include "parameter_list.sv"
`ifndef ref_pc
        `define ref_pc
module  ref_pc  (input  wire    logic                           clock,
                 input  wire    logic                           reset,
                 input  wire    logic                           stall,
                 input  wire    logic                           flush,
                 input  wire    logic   [`INST_ADDR_WIDTH-1:0]  imm_addr,
                 input  wire    logic   [`DATA_WIDTH-1:0]       alu_output,
                 input  wire    logic                           jump_reg,
                 output logic           [`INST_ADDR_WIDTH-1:0]  instr_addr);


        always @(posedge clock) begin: always_block
                if(reset)
                        instr_addr <= `RESET_VALUE;
                else if(!stall) begin: update_pc
                        if(flush) begin: flush_pc
                                if(jump_reg)
                                        instr_addr <= {alu_output[`DATA_WIDTH-1:2],2'b00};
                                else
                                        instr_addr <= imm_addr;
                        end: flush_pc
                        else
                                instr_addr <= instr_addr + `INST_ADDR_SUM;
                end: update_pc
        end: always_block
endmodule
`endif
```

APPENDIX C: Source Code of Reference Model – Instruction Memory

```
`include "parameter_list.sv"
`ifndef ref_i_mem
        `define ref_i_mem
module  ref_i_mem       (input  wire    logic                           reset,
                         ref    logic           [`DATA_REG_WIDTH-1:0]   rom     [0:((2**`MEM_ROWS) - 1)],
                         input  wire    logic   [`INST_ADDR_WIDTH-1:0]  instr_addr,
                         output logic           [`INST_CODE_WIDTH-1:0]  instr_code);

        // Setup program in instruction memory during reset
        // Fetch instruction from instruction memory based on provided instruction address
        always @(*)     begin: instruction_fetch
                instr_code = {rom[instr_addr],
                              rom[instr_addr + 1],
                              rom[instr_addr + 2],
                              rom[instr_addr + 3]};
        end: instruction_fetch
endmodule
`endif
```

APPENDIX D: Source Code of Reference Model – IF/ID Pipeline Register

```
`include "parameter_list.sv"
`ifndef ref_ifid_pipeline_reg
        `define ref_ifid_pipeline_reg
module  ref_ifid_pipeline_reg   (input  wire    logic                           clock,
                                 input  wire    logic                           reset,
                                 input  wire    logic                           stall,
                                 input  wire    logic                           flush,
                                 input  wire    logic   [`INST_ADDR_WIDTH-1:0]  instr_addr,
                                 input  wire    logic   [`INST_CODE_WIDTH-1:0]  instr_code,
                                 output logic           [`INST_ADDR_WIDTH-1:0]  ifid_addr,
                                 output logic           [`INST_CODE_WIDTH-1:0]  ifid_instr);

        // On positive clock edge, update the pipeline registers if pipeline is not stalled or flushed
        // For IF/ID pipeline register, store instruction code and address
        always @(posedge clock) begin: always_block
                if(reset) begin: system_reset
                        ifid_instr <= `RESET_VALUE;
                        ifid_addr <= `RESET_VALUE;
                end: system_reset
                else if(flush && !stall) begin: pipeline_flush
                        ifid_instr <= `NOP_INST_CODE;
                        ifid_addr <= instr_addr;
                end: pipeline_flush
                else if(!stall) begin: normal_operation
                        ifid_instr <= instr_code;
                        ifid_addr <= instr_addr;
                end: normal_operation
        end: always_block
endmodule
`endif
```

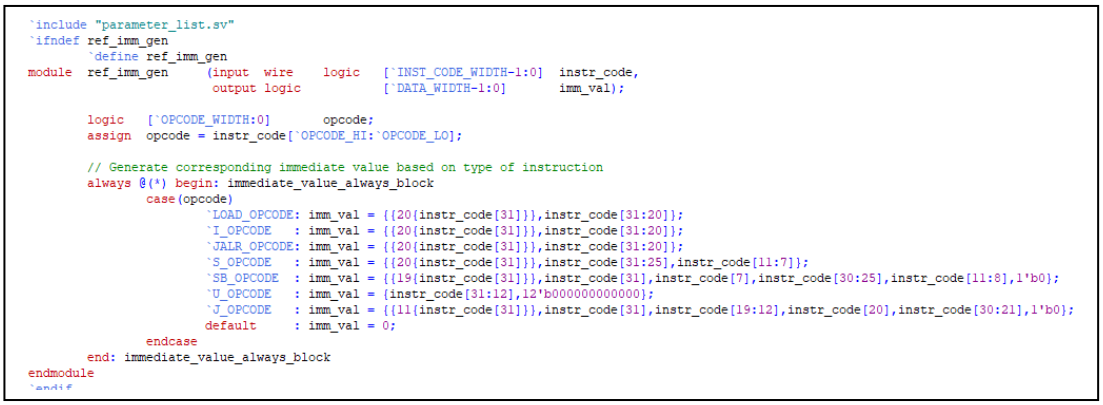
APPENDIX E: Source Code of Reference Model – Register File

```
`include "parameter_list.sv"
`ifndef ref_reg_file
        `define ref_reg_file
module  ref_reg_file    (input  wire    logic                           clock,
                         input  wire    logic                           reset,
                         input  wire    logic   [`INST_CODE_WIDTH-1:0]  instr_code,
                         input  wire    logic                           reg_write,
                         input  wire    logic   [`REG_ADDR_WIDTH-1:0]   reg_write_addr,
                         input  wire    logic   [`DATA_WIDTH-1:0]       reg_write_data,
                         output logic           [`REG_ADDR_WIDTH-1:0]   reg_read_addr_1,
                         output logic           [`DATA_WIDTH-1:0]       reg_data_1,
                         output logic           [`REG_ADDR_WIDTH-1:0]   reg_read_addr_2,
                         output logic           [`DATA_WIDTH-1:0]       reg_data_2      );

        logic   [`DATA_WIDTH-1:0]       register        [0:(2**`REG_ADDR_WIDTH)-1];

        assign  reg_read_addr_1 = reset ? 0 : instr_code[`RS1_ADDR_HI:`RS1_ADDR_LO];
        assign  reg_read_addr_2 = reset ? 0 : instr_code[`RS2_ADDR_HI:`RS2_ADDR_LO];
        assign  reg_data_1      = (reset ? 0 : (reg_write ? (reg_read_addr_1 == reg_write_addr ?
                                   reg_write_data : register[reg_read_addr_1]) : register[reg_read_addr_1]));
        assign  reg_data_2      = (reset ? 0 : (reg_write ? (reg_read_addr_2 == reg_write_addr ?
                                   reg_write_data : register[reg_read_addr_2]) : register[reg_read_addr_2]));

        always @(posedge clock) begin: always_block
                if(reset)       begin
                        for     (int i = 0; i < 32; i++)
                                register[i] <= 32'b0;
                end
                else begin
                        if(reg_write && reg_write_addr != 0)
                                register[reg_write_addr] <= reg_write_data;
                end
        end: always_block
endmodule
`endif
```

APPENDIX F: Source Code of Reference Model – Control Unit

```
`include "parameter_list.sv"
`ifndef ref_ctl_unit
        `define ref_ctl_unit
module  ref_ctl_unit    (input  wire    logic   [`INST_CODE_WIDTH-1:0]  instr_code,
                         output logic           [`CTL_SGNL_WIDTH-1:0]   ctl_op,
                         output logic           [`ALU_OP_WIDTH-1:0]     alu_op);

        logic   [`OPCODE_WIDTH:0]       opcode;
        assign  opcode = instr_code[`OPCODE_HI:`OPCODE_LO];

        // Assign control signal and ALU OP based on instruction type
        // Ensuring proper hardware functioning based on instruction
        always @(*) begin: main_control_signal_block
                case(opcode)
                `R_OPCODE   : begin
                                ctl_op = `R_CTL_SGNL;
                                alu_op = `ARITH_LOGIC_ALU_OP;
                              end
                `LOAD_OPCODE: begin
                                ctl_op = `LOAD_CTL_SGNL;
                                alu_op = `IMM_ADDR_CALC_ALU_OP;
                              end
                `U_OPCODE,
                `I_OPCODE   : begin
                                ctl_op = `I_CTL_SGNL;
                                alu_op = `ARITH_LOGIC_ALU_OP;
                              end
                `S_OPCODE   : begin
                                ctl_op = `S_CTL_SGNL;
                                alu_op = `IMM_ADDR_CALC_ALU_OP;
                              end
                `SB_OPCODE  : begin
                                ctl_op = `SB_CTL_SGNL;
                                alu_op = `COND_BRANCH_ALU_OP;
                              end
                `J_OPCODE   : begin
                                ctl_op = `J_CTL_SGNL;
                                alu_op = `JAL_ALU_OP;
                              end
                `JALR_OPCODE: begin
                                ctl_op = `JALR_CTL_SGNL;
                                alu_op = `IMM_ADDR_CALC_ALU_OP;
                              end
                default     : begin
                                ctl_op = `NOP_CTL_SGNL;
                                alu_op = `JAL_ALU_OP;
                              end
                endcase
        end: main_control_signal_block
endmodule
`endif
```

APPENDIX G: Source Code of Reference Model – Immediate Generate Unit



```
`include "parameter_list.sv"
`ifndef ref_imm_gen
        `define ref_imm_gen
module  ref_imm_gen     (input  wire    logic   [`INST_CODE_WIDTH-1:0]  instr_code,
                         output logic           [`DATA_WIDTH-1:0]       imm_val);

        logic   [`OPCODE_WIDTH:0]       opcode;
        assign  opcode = instr_code[`OPCODE_HI:`OPCODE_LO];

        // Generate corresponding immediate value based on type of instruction
        always @(*) begin: immediate_value_always_block
                case(opcode)
                        `LOAD_OPCODE: imm_val = {{20{instr_code[31]}},instr_code[31:20]};
                        `I_OPCODE   : imm_val = {{20{instr_code[31]}},instr_code[31:20]};
                        `JALR_OPCODE: imm_val = {{20{instr_code[31]}},instr_code[31:20]};
                        `S_OPCODE   : imm_val = {{20{instr_code[31]}},instr_code[31:25],instr_code[11:7]};
                        `SB_OPCODE  : imm_val = {{19{instr_code[31]}},instr_code[31],instr_code[7],instr_code[30:25],instr_code[11:8],1'b0};
                        `U_OPCODE   : imm_val = {instr_code[31:12],12'b000000000000};
                        `J_OPCODE   : imm_val = {{11{instr_code[31]}},instr_code[31],instr_code[19:12],instr_code[20],instr_code[30:21],1'b0};
                       default      : imm_val = 0;
                endcase
        end: immediate_value_always_block
endmodule
`endif
```

APPENDIX H: Source Code of Reference Model – ID/EX Pipeline Register

```
`include "parameter_list.sv"
`ifndef ref_idex_pipeline_reg
        `define ref_idex_pipeline_reg
module  ref_idex_pipeline_reg   (input  wire    logic                           clock,
                                 input  wire    logic                           reset,
                                 input  wire    logic                           stall,
                                 input  wire    logic                           flush,
                                 input  wire    logic   [`INST_CODE_WIDTH-1:0]  instr_code,
                                 input  wire    logic   [`INST_ADDR_WIDTH-1:0]  instr_addr,
                                 input  wire    logic   [`REG_ADDR_WIDTH-1:0]   reg_addr_1,
                                 input  wire    logic   [`DATA_WIDTH-1:0]       reg_data_1,
                                 input  wire    logic   [`REG_ADDR_WIDTH-1:0]   reg_addr_2,
                                 input  wire    logic   [`DATA_WIDTH-1:0]       reg_data_2,
                                 input  wire    logic   [`DATA_WIDTH-1:0]       imm_val,
                                 input  wire    logic   [`CTL_SGNL_WIDTH-1:0]   ctl_op,
                                 input  wire    logic   [`ALU_OP_WIDTH-1:0]     alu_op,
                                 output logic           [`INST_CODE_WIDTH-1:0]  idex_instr_code,
                                 output logic           [`INST_ADDR_WIDTH-1:0]  idex_instr_addr,
                                 output logic           [`DATA_WIDTH-1:0]       idex_imm_val,
                                 output logic           [`CTL_SGNL_WIDTH-1:0]   idex_ctl_op,
                                 output logic           [`ALU_OP_WIDTH-1:0]     idex_alu_op,
                                 output logic           [`REG_ADDR_WIDTH-1:0]   idex_reg_addr_1,
                                 output logic           [`DATA_WIDTH-1:0]       idex_reg_data_1,
                                 output logic           [`REG_ADDR_WIDTH-1:0]   idex_reg_addr_2,
                                 output logic           [`DATA_WIDTH-1:0]       idex_reg_data_2 );

        // On positive clock edge, update the pipeline registers if pipeline is not stalled or flushed
        // For ID/EX pipeline register, pass register read info, immediate value, control signal and ALU OP
        always @(posedge clock) begin: always_block
                if(reset) begin: system_reset
                        idex_instr_code <= `RESET_VALUE;
                        idex_instr_addr <= `RESET_VALUE;
                        idex_imm_val <= `RESET_VALUE;
                        idex_reg_data_1 <= `RESET_VALUE;
                        idex_reg_data_2 <= `RESET_VALUE;
                        idex_reg_addr_1 <= `RESET_VALUE;
                        idex_reg_addr_2 <= `RESET_VALUE;
                        idex_ctl_op <= `RESET_VALUE;
                        idex_alu_op <= `RESET_VALUE;
                end: system_reset
                else begin
                        if(flush && !stall) begin: pipeline_flush
                                idex_instr_code <= `NOP_INST_CODE;
                                idex_instr_addr <= instr_addr;
                                idex_ctl_op <= `NOP_CTL_SGNL;
                                idex_alu_op <= `RESET_VALUE;
                                idex_imm_val <= `RESET_VALUE;
                                idex_reg_data_1 <= `RESET_VALUE;
                                idex_reg_data_2 <= `RESET_VALUE;
                                idex_reg_addr_1 <= `RESET_VALUE;
                                idex_reg_addr_2 <= `RESET_VALUE;
                        end: pipeline_flush
                        else if(!stall) begin: normal_operation
                                idex_instr_code <= instr_code;
                                idex_instr_addr <= instr_addr;
                                idex_imm_val <= imm_val;
                                idex_reg_data_1 <= reg_data_1;
                                idex_reg_data_2 <= reg_data_2;
                                idex_reg_addr_1 <= reg_addr_1;
                                idex_reg_addr_2 <= reg_addr_2;
                                idex_ctl_op <= ctl_op;
                                idex_alu_op <= alu_op;
                        end: normal_operation
                end
        end: always_block
endmodule
`endif
```

APPENDIX I: Source Code of Reference Model – ALU Control Unit

```
`include "parameter_list.sv"
`ifndef ref_alu_ctl
        `define ref_alu_ctl
module  ref_alu_ctl     (input  wire    logic   [`ALU_OP_WIDTH-1:0]     alu_op,
                         input  wire    logic   [`INST_CODE_WIDTH-1:0]  instr_code,
                         output logic           [`ALU_CTL_WIDTH-1:0]    alu_ctl);

        logic   [`OPCODE_WIDTH:0]       opcode;
        logic   [`FUNCT3_WIDTH:0]       funct3;
        logic   [`FUNCT7_WIDTH:0]       funct7;
        assign  opcode = instr_code[`OPCODE_HI:`OPCODE_LO];
        assign  funct3 = instr_code[`FUNCT3_HI:`FUNCT3_LO];
        assign  funct7 = instr_code[`FUNCT7_HI:`FUNCT7_LO];

        // Assign ALU control signal correspond to the instruciton operation
        always @(*) begin: main_alu_ctl_block
                case(alu_op)
                `IMM_ADDR_CALC_ALU_OP: alu_ctl = `ADD_CTL;
                `COND_BRANCH_ALU_OP: begin: sb_alu_ctl_block
                                        case(funct3)
                                        `BEQ_FUNCT3     : alu_ctl = `COMP_EQ_CTL;
                                        `BNE_FUNCT3     : alu_ctl = `COMP_NEQ_CTL;
                                        `BLT_FUNCT3     : alu_ctl = `COMP_LESS_CTL;
                                        `BGE_FUNCT3     : alu_ctl = `COMP_GEQ_CTL;
                                        `BLTU_FUNCT3    : alu_ctl = `COMP_LESS_UNSIGNED_CTL;
                                        `BGEU_FUNCT3    : alu_ctl = `COMP_GEQ_UNSIGNED_CTL;
                                        endcase
                                    end: sb_alu_ctl_block
                `ARITH_LOGIC_ALU_OP: begin: arith_logic_alu_ctl_block
                                        case(opcode)
                                        `I_OPCODE: begin: i_type_alu_ctl_block
                                                        case(funct3)
                                                        `ADD_FUNCT3     : alu_ctl = `ADD_CTL;
                                                        `SLL_FUNCT3     : alu_ctl = `SLL_CTL;
                                                        `SLT_FUNCT3     : alu_ctl = `COMP_GEQ_CTL;
                                                        `SLTU_FUNCT3    : alu_ctl = `COMP_GEQ_UNSIGNED_CTL;
                                                        `XOR_FUNCT3     : alu_ctl = `XOR_CTL;
                                                        `SR_FUNCT3      : begin
                                                                            case(funct7)
                                                                            `DEFAULT_FUNCT7 : alu_ctl = `SRL_CTL;
                                                                            `ALT_FUNCT7     : alu_ctl = `SRA_CTL;
                                                                            endcase
                                                                          end
                                                        `OR_FUNCT3      : alu_ctl = `OR_CTL;
                                                        `AND_FUNCT3     : alu_ctl = `AND_CTL;
                                                        endcase
                                                   end: i_type_alu_ctl_block
                                        `R_OPCODE: begin: r_type_alu_ctl_block
                                                        case(funct3)
                                                        `ADD_FUNCT3     : begin
                                                                            case(funct7)
                                                                            `DEFAULT_FUNCT7 : alu_ctl = `ADD_CTL;
                                                                            `ALT_FUNCT7     : alu_ctl = `SUB_CTL;
                                                                            endcase
                                                                          end
                                                        `SLL_FUNCT3     : alu_ctl = `SLL_CTL;
                                                        `SLT_FUNCT3     : alu_ctl = `COMP_GEQ_CTL;
                                                        `SLTU_FUNCT3    : alu_ctl = `COMP_GEQ_UNSIGNED_CTL;
                                                        `XOR_FUNCT3     : alu_ctl = `XOR_CTL;
                                                        `SR_FUNCT3      : begin
                                                                            case(funct7)
                                                                            `DEFAULT_FUNCT7 : alu_ctl = `SRL_CTL;
                                                                            `ALT_FUNCT7     : alu_ctl = `SRA_CTL;
                                                                            endcase
                                                                          end
                                                        `OR_FUNCT3      : alu_ctl = `OR_CTL;
                                                        `AND_FUNCT3     : alu_ctl = `AND_CTL;
                                                        endcase
                                                   end: r_type_alu_ctl_block
                                        `U_OPCODE: alu_ctl = `LOAD_UPPER_CTL;
                                        endcase
                                    end: arith_logic_alu_ctl_block
                `JAL_ALU_OP: alu_ctl = `SET_ZERO_CTL;
                endcase
        end: main_alu_ctl_block
endmodule
`endif
```

APPENDIX J: Source Code of Reference Model – ALU

```
`include "parameter_list.sv"
`ifndef ref_alu
        `define ref_alu
module  ref_alu (input                                  mem_to_reg,
                 input          [`DATA_WIDTH-1:0]       reg_data_1,
                 input          [`DATA_WIDTH-1:0]       reg_data_2,
                 input          [`DATA_WIDTH-1:0]       imm_val,
                 input          [`ALU_CTL_WIDTH-1:0]    alu_ctl,
                 input                                  alu_src,
                 input          [`FWRD_MUX_WIDTH-1:0]   fwrd_mux_1,
                 input          [`FWRD_MUX_WIDTH-1:0]   fwrd_mux_2,
                 input          [`DATA_WIDTH-1:0]       exmem_alu_data,
                 input          [`DATA_WIDTH-1:0]       memwb_alu_data,
                 input          [`DATA_WIDTH-1:0]       memwb_mem_data,
                 output reg     [`DATA_WIDTH-1:0]       alu_output,
                 output reg                             alu_zero);

        logic   [`DATA_WIDTH-1:0]       data_1;
        logic   [`DATA_WIDTH-1:0]       data_2;
        assign  data_1 = (fwrd_mux_1 == `FWRD_ALU ? exmem_alu_data :
                         (fwrd_mux_1 == `FWRD_MEM ? (mem_to_reg ? memwb_mem_data :memwb_alu_data) : reg_data_1));
        assign  data_2 = (alu_src ? imm_val : (fwrd_mux_2 == `FWRD_ALU ? exmem_alu_data :
                         (fwrd_mux_2 == `FWRD_MEM ? (mem_to_reg ? memwb_mem_data : memwb_alu_data) : reg_data_2)));

        // Perform specific operation based on alu_ctl signal provided
        always @(*) begin: main_alu_block
                case(alu_ctl)
                `AND_CTL                : alu_output = data_1 & data_2;
                `OR_CTL                 : alu_output = data_1 | data_2;
                `ADD_CTL                : alu_output = data_1 + data_2;
                `SUB_CTL                : alu_output = data_1 - data_2;
                `SLL_CTL                : alu_output = data_1 << data_2[4:0];
                `SRL_CTL                : alu_output = data_1 >> data_2[4:0];
                `SRA_CTL                : alu_output = $signed(data_1) >>> data_2[4:0];
                `XOR_CTL                : alu_output = data_1 ^ data_2;
                `COMP_EQ_CTL            : alu_output = !(data_1 == data_2);
                `COMP_NEQ_CTL           : alu_output = (data_1 == data_2);
                `COMP_GEQ_UNSIGNED_CTL  : alu_output = ((data_1 >= data_2) ? 0 : 1);
                `COMP_LESS_UNSIGNED_CTL : alu_output = ((data_1 < data_2) ? 0 : 1);
                `COMP_GEQ_CTL           : alu_output = (($signed(data_1) >= $signed(data_2)) ? 0 : 1);
                `COMP_LESS_CTL          : alu_output = (($signed(data_1) < $signed(data_2)) ? 0 : 1);
                `LOAD_UPPER_CTL         : alu_output = data_2;
                `SET_ZERO_CTL           : alu_output = 0;
                endcase
                alu_zero <= (alu_output == 0);
        end: main_alu_block
endmodule
`endif
```

APPENDIX K: Source Code of Reference Model – Immediate Address Unit

```
`include "parameter_list.sv"
`ifndef ref_imm_addr_unit
        `define ref_imm_addr_unit
module  ref_imm_addr_unit       (input  wire    logic   [`INST_ADDR_WIDTH-1:0]  instr_addr,
                                 input  wire    logic   [`DATA_WIDTH-1:0]       imm_val,
                                 output logic           [`INST_ADDR_WIDTH-1:0]  imm_addr);

        // Calculate the correct immediate address for branches
        always @(*)
                imm_addr = instr_addr + imm_val;
endmodule
`endif
```

APPENDIX L: Source Code of Reference Model – EX/MEM Pipeline Register

```
`ifndef ref_exmem_pipeline_reg
        `define ref_exmem_pipeline_reg
module  ref_exmem_pipeline_reg  (input  wire    logic                           clock,
                                 input  wire    logic                           reset,
                                 input  wire    logic                           alu_zero,
                                 input  wire    logic                           stall,
                                 input  wire    logic                           flush,
                                 input  wire    logic                           mem_to_reg,
                                 input  wire    logic   [`DATA_WIDTH-1:0]       imm_val,
                                 input  wire    logic   [`FWRD_MUX_WIDTH-1:0]   fwrd_mux_2,
                                 input  wire    logic   [`DATA_WIDTH-1:0]       memwb_mem_data,
                                 input  wire    logic   [`DATA_WIDTH-1:0]       memwb_alu_data,
                                 input  wire    logic   [`DATA_WIDTH-1:0]       idex_mem_data,
                                 input  wire    logic   [`DATA_WIDTH-1:0]       alu_output,
                                 input  wire    logic   [`INST_CODE_WIDTH-1:0]  instr_code,
                                 input  wire    logic   [`INST_ADDR_WIDTH-1:0]  instr_addr,
                                 input  wire    logic   [`CTL_SGNL_WIDTH-1:0]   ctl_op,
                                 output logic           [`INST_CODE_WIDTH-1:0]  exmem_instr_code,
                                 output logic           [`INST_ADDR_WIDTH-1:0]  exmem_instr_addr,
                                 output logic           [`DATA_WIDTH-1:0]       exmem_mem_data,
                                 output logic           [`DATA_WIDTH-1:0]       exmem_alu_output,
                                 output logic                                   exmem_zero,
                                 output logic           [`DATA_WIDTH-1:0]       exmem_imm_val,
                                 output logic           [`CTL_SGNL_WIDTH-1:0]   exmem_ctl_op);

        // On positive clock edge, update the pipeline registers if pipeline is not stalled or flushed
        // For EX/MEM pipeline register, pass instruction and control signal info, store ALU output
        always @(posedge clock) begin: always_block
                if(reset) begin: system_reset
                        exmem_instr_code <= `RESET_VALUE;
                        exmem_instr_addr <= `RESET_VALUE;
                        exmem_alu_output <= `RESET_VALUE;
                        exmem_zero <= `RESET_VALUE;
                        exmem_imm_val <= `RESET_VALUE;
                        exmem_ctl_op <= `RESET_VALUE;
                        exmem_mem_data <= `RESET_VALUE;
                end: system_reset
                else begin
                        if(stall) begin: pipeline_stall
                                exmem_instr_code <= `NOP_INST_CODE;
                                exmem_ctl_op <= `NOP_CTL_SGNL;
                        end: pipeline_stall
                        else begin: normal_operation
                                exmem_instr_code <= instr_code;
                                exmem_ctl_op <= ctl_op;
                        end: normal_operation
                        exmem_instr_addr <= instr_addr;
                        exmem_alu_output <= alu_output;
                        exmem_zero <= alu_zero;
                        exmem_imm_val <= imm_val;
                        exmem_mem_data <= (fwrd_mux_2 == `NO_FWRD ? idex_mem_data :
                                          (fwrd_mux_2 == `FWRD_ALU ? exmem_alu_output :
                                          (mem_to_reg ? memwb_mem_data : memwb_alu_data)));
                end
        end: always_block
endmodule
`endif
```

APPENDIX M: Source Code of Reference Model – Data Memory Unit

```
`include "parameter_list.sv"
`ifndef ref_d_mem
        `define ref_d_mem
module  ref_d_mem       (input  wire    logic                           clock,
                         input  wire    logic                           reset,
                         input  wire    logic                           mem_write,
                         input  wire    logic                           mem_read,
                         input  wire    logic   [`INST_CODE_WIDTH-1:0]  instr_code,
                         input  wire    logic   [`MEM_ADDR_WIDTH-1:0]   address,
                         input  wire    logic   [`DATA_WIDTH-1:0]       reg_data,
                         output logic           [`DATA_WIDTH-1:0]       loaded_data,
                         output logic           [`DATA_WIDTH-1:0]       stored_data);

        int     fh;
        logic   [`DATA_REG_WIDTH-1:0]   ram     [0:((2**`MEM_ROWS) - 1)];
        logic   [`DATA_WIDTH-1:0]       read_data;
        logic   [`FUNCT3_WIDTH-1:0]     funct3;

        assign  funct3 = instr_code[`FUNCT3_HI:`FUNCT3_LO];

        assign  read_data = {(ram[address+3] === 8'bx ? 8'b0 : ram[address+3]),
                             (ram[address+2] === 8'bx ? 8'b0 : ram[address+2]),
                             (ram[address+1] === 8'bx ? 8'b0 : ram[address+1]),
                             (ram[address]   === 8'bx ? 8'b0 : ram[address])};

        assign  loaded_data = mem_read ? (funct3 == `LB_FUNCT3 ? {{24{read_data[7]}},read_data[7:0]} :
                                          (funct3 == `LH_FUNCT3 ? {{16{read_data[15]}},read_data[15:0]} :
                                          (funct3 == `LW_FUNCT3 ? read_data :
                                          (funct3 == `LBU_FUNCT3 ? {24'b0,read_data[7:0]} : {16'b0,read_data[15:0]})))) : 0;

        assign  stored_data = mem_write ? (funct3 == `SB_FUNCT3 ? {24'b0,reg_data[7:0]} :
                                           (funct3 == `SH_FUNCT3 ? {16'b0,reg_data[15:0]} : reg_data)) : 0;

        // Write corresponding register data to the data memory
        always @(posedge clock) begin: memory_control_block
                if(reset)
                        ram <= '{default:`RESET_VALUE};
                if(mem_write)   begin
                        case (funct3)
                        `SB_FUNCT3: ram[address] <= reg_data[7:0];
                        `SH_FUNCT3: begin
                                        ram[address+1] <= reg_data[15:8];
                                        ram[address]   <= reg_data[7:0];
                                    end
                        `SW_FUNCT3: begin
                                        ram[address+3] <= reg_data[31:24];
                                        ram[address+2] <= reg_data[23:16];
                                        ram[address+1] <= reg_data[15:8];
                                        ram[address]   <= reg_data[7:0];
                                    end
                        endcase
                end
        end: memory_control_block
endmodule
`endif
```

APPENDIX N: Source Code of Reference Model – MEM/WB Pipeline Register

```
`include "parameter_list.sv"
`ifndef ref_memwb_pipeline_reg
        `define ref_memwb_pipeline_reg
module  ref_memwb_pipeline_reg  (input  wire    logic                           clock,
                                 input  wire    logic                           reset,
                                 input  wire    logic   [`DATA_WIDTH-1:0]       alu_output,
                                 input  wire    logic   [`CTL_SGNL_WIDTH-1:0]   ctl_op,
                                 input  wire    logic   [`DATA_WIDTH-1:0]       loaded_data,
                                 input  wire    logic   [`DATA_WIDTH-1:0]       stored_data,
                                 input  wire    logic   [`REG_ADDR_WIDTH-1:0]   rd,
                                 input  wire    logic   [`INST_ADDR_WIDTH-1:0]  instr_addr,
                                 output logic           [`CTL_SGNL_WIDTH-1:0]   memwb_ctl_op,
                                 output logic           [`REG_ADDR_WIDTH-1:0]   memwb_write_addr,
                                 output logic           [`DATA_WIDTH-1:0]       memwb_write_data,
                                 output logic           [`DATA_WIDTH-1:0]       memwb_mem_data,
                                 output logic           [`DATA_WIDTH-1:0]       memwb_alu_data,
                                 output logic           [`DATA_WIDTH-1:0]       memwb_mem_addr,
                                 output logic           [`INST_ADDR_WIDTH-1:0]  memwb_mem_write_data,
                                 output logic           [`INST_ADDR_WIDTH-1:0]  memwb_instr_addr);

        always @(posedge clock) begin: always_block
                if(reset) begin: system_reset
                        memwb_ctl_op <= `RESET_VALUE;
                        memwb_write_addr <= `RESET_VALUE;
                        memwb_mem_data <= `RESET_VALUE;
                        memwb_alu_data <= `RESET_VALUE;
                        memwb_mem_addr <= `RESET_VALUE;
                        memwb_mem_write_data <= `RESET_VALUE;
                        memwb_instr_addr <= `RESET_VALUE;
                end: system_reset
                else begin: normal_operation
                        memwb_ctl_op <= ctl_op;
                        memwb_write_addr <= rd;
                        memwb_write_data <= (ctl_op[`JUMP_LINK] ? instr_addr + `INST_ADDR_SUM :
                                             (ctl_op[`MEM_TO_REG] ? loaded_data : alu_output));
                        memwb_mem_data <= loaded_data;
                        memwb_alu_data <= alu_output;
                        memwb_mem_addr <= alu_output;
                        memwb_mem_write_data <= stored_data;
                        memwb_instr_addr <= instr_addr;
                end: normal_operation
        end: always_block
endmodule
`endif
```

APPENDIX O: Source Code of Reference Model – Forwarding Unit

```
`include "parameter_list.sv"
`ifndef ref_fwrd_unit
        `define ref_fwrd_unit
module  ref_fwrd_unit   (input  wire    logic                           reset,
                         input  wire    logic                           exmem_reg_write,
                         input  wire    logic                           memwb_reg_write,
                         input  wire    logic                           stall,
                         input  wire    logic   [`REG_ADDR_WIDTH-1:0]   exmem_rd,
                         input  wire    logic   [`REG_ADDR_WIDTH-1:0]   memwb_rd,
                         input  wire    logic   [`OPCODE_WIDTH-1:0]     exmem_opcode,
                         input  wire    logic   [`REG_ADDR_WIDTH-1:0]   reg_1,
                         input  wire    logic   [`REG_ADDR_WIDTH-1:0]   reg_2,
                         output logic           [`FWRD_MUX_WIDTH-1:0]   fwrd_mux_1,
                         output logic           [`FWRD_MUX_WIDTH-1:0]   fwrd_mux_2);

        // Provides data forwarding multiplex control signal based on write register address
        // from EX/MEM and MEM/WB pipeline register and OPCODE to check for LOAD type instruction
        always@(*) begin: always_block
                if(reset) begin: system_reset
                        fwrd_mux_1 = `RESET_VALUE;
                        fwrd_mux_2 = `RESET_VALUE;
                end:  system_reset
                if(exmem_opcode != `LOAD_OPCODE && exmem_reg_write && exmem_rd != 0) begin: exmem_fwrd
                        if(reg_1 == exmem_rd)
                                fwrd_mux_1 = `FWRD_ALU;
                        else if (reg_1 == memwb_rd && memwb_reg_write) //CHECKME
                                fwrd_mux_1 = `FWRD_MEM;
                        else
                                fwrd_mux_1 = `NO_FWRD;
                        if(reg_2 == exmem_rd)
                                fwrd_mux_2 = `FWRD_ALU;
                        else if (reg_2 == memwb_rd && memwb_reg_write)
                                fwrd_mux_2 = `FWRD_MEM;
                        else
                                fwrd_mux_2 = `NO_FWRD;
                end: exmem_fwrd
                else if(memwb_reg_write && memwb_rd != 0) begin: memwb_fwrd
                        if(reg_1 == memwb_rd)
                                fwrd_mux_1 = `FWRD_MEM;
                        else
                                fwrd_mux_1 = `NO_FWRD;
                        if(reg_2 == memwb_rd)
                                fwrd_mux_2 = `FWRD_MEM;
                        else
                                fwrd_mux_2 = `NO_FWRD;
                end: memwb_fwrd
                else begin: default_case
                        fwrd_mux_1 = `NO_FWRD;
                        fwrd_mux_2 = `NO_FWRD;
                end: default_case
        end: always_block
endmodule
`endif
```

APPENDIX P: Source Code of Reference Model – Hazard Detection Unit

```
`include "parameter_list.sv"
`ifndef ref_hzrd_unit
        `define ref_hzrd_unit
module  ref_hzrd_unit   (input  wire    logic                           reset,
                         input  wire    logic                           mem_read,
                         input  wire    logic                           zero,
                         input  wire    logic                           branch,
                         input  wire    logic                           jump,
                         input  wire    logic   [`REG_ADDR_WIDTH-1:0]   exmem_rd,
                         input  wire    logic   [`REG_ADDR_WIDTH-1:0]   idex_reg_addr_1,
                         input  wire    logic   [`REG_ADDR_WIDTH-1:0]   idex_reg_addr_2,
                         input  wire    logic   [`FWRD_MUX_WIDTH-1:0]   fwrd_mux_1,
                         input  wire    logic   [`FWRD_MUX_WIDTH-1:0]   fwrd_mux_2,
                         output logic                                   stall,
                         output logic                                   flush);

        // Assign corresponding value for flush and stall control signal based on
        // load-use case and branch condition fulfillment
        always @(*) begin: hazard_detectioin_always_block
                if(reset) begin: system_reset
                        stall = `RESET_VALUE;
                        flush = `RESET_VALUE;
                end: system_reset
                if((zero && branch) || jump)
                        flush = 1;
                else
                        flush = 0;
                if(mem_read && (idex_reg_addr_1 == exmem_rd || idex_reg_addr_2 == exmem_rd) && exmem_rd != 0) begin
                        if ((idex_reg_addr_1 == exmem_rd && fwrd_mux_1 != `FWRD_MEM) ||
                            (idex_reg_addr_2 == exmem_rd && fwrd_mux_2 != `FWRD_MEM))
                                stall = 1;
                end
                else
                        stall = 0;
        end: hazard_detectioin_always_block
endmodule
`endif
```

APPENDIX Q: Source Code of Parameter List

```
// General Processor Specifications
`define INST_ADDR_WIDTH         32
`define INST_CODE_WIDTH         32
`define DATA_WIDTH              32
`define MEM_ROWS                16
`define MEM_ADDR_WIDTH          16
`define DATA_REG_WIDTH          8
`define INST_ADDR_SUM           4
`define RESET_VALUE             0
`define NOP_INST_CODE           32'h00000000

// Immediate Constant Values
`define I_IMM_WIDTH             12
`define SB_IMM_WIDTH            12
`define J_IMM_WIDTH             20
`define U_IMM_WIDTH             20

// Register Access Constant Values
`define REG_ADDR_WIDTH          5
`define RS1_ADDR_HI             19
`define RS1_ADDR_LO             15
`define RS2_ADDR_HI             24
`define RS2_ADDR_LO             20
`define RD_ADDR_HI              11
`define RD_ADDR_LO              7

// FUNCT7 Constant Values
`define FUNCT7_WIDTH            7
`define FUNCT7_HI               31
`define FUNCT7_LO               25
`define DEFAULT_FUNCT7          7'b0000000
`define ALT_FUNCT7              7'b0100000

// FUNCT3 Constant Values
`define FUNCT3_WIDTH            3
`define FUNCT3_HI               14
`define FUNCT3_LO               12
`define LB_FUNCT3               3'b000
`define LH_FUNCT3               3'b001
`define LW_FUNCT3               3'b010
`define LBU_FUNCT3              3'b100
`define LHU_FUNCT3              3'b101
`define SB_FUNCT3               3'b000
`define SH_FUNCT3               3'b001
`define SW_FUNCT3               3'b010
`define BEQ_FUNCT3              3'b000
`define BNE_FUNCT3              3'b001
`define BLT_FUNCT3              3'b100
`define BGE_FUNCT3              3'b101
`define BLTU_FUNCT3             3'b110
`define BGEU_FUNCT3             3'b111
`define ADD_FUNCT3              3'b000
`define SLL_FUNCT3              3'b001
`define SLT_FUNCT3              3'b010
`define SLTU_FUNCT3             3'b011
```

```
`define XOR_FUNCT3              3'b100
`define SR_FUNCT3               3'b101
`define OR_FUNCT3               3'b110
`define AND_FUNCT3              3'b111
`define JALR_FUNCT3             3'b000

// OPCODE Constant Values
`define OPCODE_WIDTH            7
`define OPCODE_HI               6
`define OPCODE_LO               0
`define R_OPCODE                7'b0110011
`define LOAD_OPCODE             7'b0000011
`define S_OPCODE                7'b0100011
`define U_OPCODE                7'b0110111
`define I_OPCODE                7'b0010011
`define SB_OPCODE               7'b1100011
`define J_OPCODE                7'b1101111
`define JALR_OPCODE             7'b1100111
`define NOP_OPCODE              7'b0000000

// Control Signal Constant Values
`define CTL_SGNL_WIDTH          8
`define NOP_CTL_SGNL            8'b00000000
`define R_CTL_SGNL              8'b00001000
`define LOAD_CTL_SGNL           8'b00111100
`define I_CTL_SGNL              8'b00101000
`define S_CTL_SGNL              8'b00100010
`define SB_CTL_SGNL             8'b00000001
`define J_CTL_SGNL              8'b01001000
`define JALR_CTL_SGNL           8'b11101000

// Specific Control Signal Index
`define BRANCH                  0
`define MEM_WRITE               1
`define MEM_READ                2
`define REG_WRITE               3
`define MEM_TO_REG              4
`define ALU_SRC                 5
`define JUMP_LINK               6
`define JUMP_REG                7

// ALU OP Constant Values
`define ALU_OP_WIDTH            2
`define IMM_ADDR_CALC_ALU_OP    2'b00
`define COND_BRANCH_ALU_OP      2'b01
`define ARITH_LOGIC_ALU_OP      2'b10
`define JAL_ALU_OP              2'b11

// ALU CTL OP Constant Values
`define ALU_CTL_WIDTH           4
`define AND_CTL                 4'b0000
`define OR_CTL                  4'b0001
`define ADD_CTL                 4'b0010
`define SUB_CTL                 4'b0011
`define SLL_CTL                 4'b0100
`define SRL_CTL                 4'b0101
`define SRA_CTL                 4'b0110
`define XOR_CTL                 4'b0111
`define COMP_EQ_CTL             4'b1000
`define COMP_NEQ_CTL            4'b1001
`define COMP_GEQ_UNSIGNED_CTL   4'b1010
```

```
`define COMP_LESS_UNSIGNED_CTL  4'b1011
`define COMP_GEQ_CTL            4'b1100
`define COMP_LESS_CTL           4'b1101
`define LOAD_UPPER_CTL          4'b1110
`define SET_ZERO_CTL            4'b1111


// Forwarding Constant Values
`define FWRD_MUX_WIDTH          2
`define NO_FWRD                 2'b00
`define FWRD_ALU                2'b10
`define FWRD_MEM                2'b01

// Coverage Coverpoint Constant Values
`define BEQ_CVR                 10'b0001100011
`define BNE_CVR                 10'b0011100011
`define BLT_CVR                 10'b1001100011
`define BGE_CVR                 10'b1011100011
`define BLTU_CVR                10'b1101100011
`define BGEU_CVR                10'b1111100011
`define LB_CVR                  10'b0000000011
`define LH_CVR                  10'b0010000011
`define LW_CVR                  10'b0100000011
`define LBU_CVR                 10'b1000000011
`define LHU_CVR                 10'b1010000011
`define SB_CVR                  10'b0000100011
`define SH_CVR                  10'b0010100011
`define SW_CVR                  10'b0100100011
`define ADDI_CVR                10'b0000010011
`define SLLI_CVR                10'b0010010011
`define XORI_CVR                10'b1000010011
`define ORI_CVR                 10'b1100010011
`define ANDI_CVR                10'b1110010011
`define SLTI_CVR                10'b0100010011
`define SLTIU_CVR               10'b0110010011
`define SRLI_CVR                17'b00000001010010011
`define SRAI_CVR                17'b01000001010010011
`define ADD_CVR                 17'b00000000000110011
`define SUB_CVR                 17'b01000000000110011
`define SLL_CVR                 17'b00000000010110011
`define XOR_CVR                 17'b00000001000110011
`define SRL_CVR                 17'b00000001010110011
`define SRA_CVR                 17'b01000001010110011
`define OR_CVR                  17'b00000001100110011
`define AND_CVR                 17'b00000001110110011
`define SLT_CVR                 17'b00000000100110011
`define SLTU_CVR                17'b00000000110110011
```

APPENDIX R: Source Code of Verification Environment – UVM Testbench

```
`include "uvm_pkg.sv"
`include "uvm_macros.svh"
`include "test.sv"
import    uvm_pkg::*;
`ifndef testbench
        `define testbench
module  testbench;

        logic   clk;
        always  #10     clk = ~clk;

        virtual_interface       if_instance(clk);
        dut_model               riscv(.interface_instance(if_instance));
        ref_model               riscv_ref(.interface_instance(if_instance));


        initial begin
                uvm_config_db#(virtual virtual_interface)::set(null,"uvm_test_top","virtual_interface",if_instance);
                run_test("test");
        end

        initial
                clk     <= 0;
endmodule
`endif
```

APPENDIX S: Source Code of Verification Environment – UVM Test

```
`include "uvm_pkg.sv"
`include "uvm_macros.svh"
`include "env.sv"
`include "seq.sv"
import    uvm_pkg::*;
`ifndef test
        `define test
class   test            extends uvm_test;
        `uvm_component_utils(test)

        function new(string name = "test", uvm_component parent = null);
                super.new(name, parent);
        endfunction

        env             env_inst;
        seq             seq_inst;
        virtual         virtual_interface       interface_instance;

        virtual function void   build_phase(uvm_phase phase);
                // Declare variables to store arguments from command line onto configuration
                int     test_seed;
                int     instr_amt;
                string  seed_file;
                string  instr_type;

                super.build_phase(phase);
                env_inst= env::type_id::create("env_inst",this);

                if(!uvm_config_db#(virtual virtual_interface)::get(this,"","virtual_interface",interface_instance))
                        `uvm_fatal("TEST", "Unable to access virtual interface on verification environment")
                uvm_config_db#(virtual virtual_interface)::set(this,"env_inst.agent_inst.*","virtual_interface",interface_instance);

                seq_inst = seq::type_id::create("seq_inst");

                // Pass test arguments as configuration parameters or use default values set
                if($test$plusargs("BATCH_TEST"))        begin
                        uvm_config_db#(bit)::set(null,"uvm_test_top","batch_test",1);
                        if($value$plusargs("BATCH_SEED=%s",seed_file))
                                uvm_config_db#(string)::set(null,"uvm_test_top","seed_file",seed_file);
                        if($test$plusargs("CONT"))
                                uvm_config_db#(bit)::set(null,"uvm_test_top","run_all",1);
                end
                if($value$plusargs("SEED=%s",test_seed))
                        uvm_config_db#(int)::set(null,"uvm_test_top","test_seed",test_seed);
                if($value$plusargs("INSTR_TYPE=%s",instr_type))
                        uvm_config_db#(string)::set(null,"uvm_test_top","instr_type",instr_type);
                if($test$plusargs("DIRECTED_TEST"))
                        uvm_config_db#(bit)::set(null,"uvm_test_top","directed_test",1);
                if($test$plusargs("FORCE_GEN"))
                        uvm_config_db#(bit)::set(null,"uvm_test_top","force_gen",1);
                if($value$plusargs("INSTR=%d",instr_amt))
                        uvm_config_db#(int)::set(null,"uvm_test_top","instr_amount",instr_amt);
                if($test$plusargs("MACRO_OVERWRITE"))
                        uvm_config_db#(bit)::set(null,"uvm_test_top","macro_overwrite",1);
                if($test$plusargs("SKIP_MACRO_CHECK"))
                        uvm_config_db#(bit)::set(null,"uvm_test_top","bypass_macro",1);
                if($test$plusargs("TESTLOG"))
                        uvm_config_db#(bit)::set(null,"uvm_test_top","testlog",1);
                if($test$plusargs("MEMLOG"))
                        uvm_config_db#(bit)::set(null,"uvm_test_top","memlog",1);
        endfunction

        virtual task            run_phase(uvm_phase phase);
                // Config and Database Parameter Variables
                bit     batch_test;
                bit     batch_test_in_progress;
                bit     run_all;
                string  seed_file;
                int     fh;
                int     batch_test_max;
                int     current_test_index;

                //Retrieve Parameter Values
                if(!uvm_config_db#(bit)::get(null,"uvm_test_top","batch_test",batch_test))
                        batch_test = 0;
                if(!uvm_config_db#(bit)::get(null,"uvm_test_top","run_all",run_all))
                        run_all = 0;
                if(!uvm_config_db#(string)::get(null,"uvm_test_top","seed_file",seed_file))
                        seed_file = "SEED.txt";

                if(batch_test)
                        batch_test_max = read_seed(seed_file);
                else
                        batch_test_max = 1;

                // Clear FAILED.txt file for continuous batch test
                if(batch_test && run_all) begin
                        fh = $fopen("FAILED.txt","w");
                        $fclose(fh);
                end

                // Initialize resource parameters
                current_test_index = 0;
                uvm_resource_db#(int)::set("uvm_test_top","current_test_index",0);
                uvm_resource_db#(bit)::set("uvm_test_top","batch_test_in_progress",batch_test);
                uvm_resource_db#(int)::set("uvm_test_top","batch_test_max",batch_test_max);
                if(!uvm_resource_db#(bit)::read_by_name("uvm_test_top","batch_test_in_progress",batch_test_in_progress))
                        batch_test_in_progress = 0;
                do begin
                        phase.raise_objection (this);
                        apply_reset();
                        if(batch_test) begin: display_batch_test_progress
                                $display("Batch Test Progress:                                          %d\/%d",current_test_index + 1 ,batch_test_max);
                                if(!uvm_resource_db#(int)::write_by_name("uvm_test_top","current_test_index",current_test_index)) begin
                                        `uvm_fatal("ERROR","Fail to update UVM Resource Database");
                                end
                        end: display_batch_test_progress
                        $display("Initiating Test Run");
```

```
                    seq_inst.start (env_inst.agent_inst.seqr_inst);
                    start_test();
                    while(interface_instance.test_in_progress)
                            #20;
                            #20;
                       if(!interface_instance.mismatch_detected) begin: mismatch_post_processing
                            #80;
                            interface_instance.end_of_test <= 1;
                            #20;
                       end: mismatch_post_processing
                       phase.drop_objection (this);
                       end_test();
                       if(batch_test)
                            current_test_index ++;
                       if(!uvm_resource_db#(bit)::read_by_name("uvm_test_top","batch_test_in_progress",batch_test_in_progress))
                            batch_test_in_progress = 0;
               end while (batch_test_in_progress);
       endtask

       // Apply reset (active high) and initialize clock to LOW
       virtual task apply_reset();
               interface_instance.reset <= 1;
               interface_instance.end_of_test <= 0;
       endtask

       // Release reset to initiate verification subject operation
       virtual task start_test();
               interface_instance.reset <= 0;
               interface_instance.test_in_progress <= 1;
               interface_instance.mismatch_detected <= 0;
               #40;
               interface_instance.monitor_start <= 1;
       endtask

       // Simple mechanism for cleaning up signals after test run
       virtual task end_test();
               interface_instance.reset <= 1;
               interface_instance.monitor_start <= 0;
               interface_instance.end_of_test <= 0;
       endtask

       function int read_seed(string seed_file);
               int     fh;
               int     code;
               int     batch_test_max;
               string  buffer;
               string  dump;

               fh = $fopen(seed_file,"r");
               $display("Checking for batch test max range");
               if(fh) begin: read_seed_file
                       batch_test_max = 0;
                       while(!$feof(fh)) begin: read_seed_line
                               code = $fscanf(fh,"%s",buffer);
                               if(buffer == "\//") begin: skip_comment
                                       code = $fgets(dump,fh);
                                       continue;
                               end: skip_comment
                               case(buffer[0])
                               "0",
                               "1",
                               "2",
                               "3",
                               "4",
                               "5",
                               "6",
                               "7",
                               "8",
                               "9": batch_test_max ++;
                               default:`uvm_fatal("ERROR",$sformatf("Invalid seed read from seed file: %s", buffer))
                               endcase
                       end: read_seed_line
               end: read_seed_file
               else
                       `uvm_fatal("ERROR",$sformatf("Invalid seed file: %s",seed_file))
               $fclose(fh);
               $display("Total Seeds Detected:                                                %d", batch_test_max);
               return batch_test_max;
       endfunction
endclass
`endif
```

APPENDIX T: Source Code of Verification Environment – Interface

```
`include "parameter_list.sv"
`ifndef virtual_interface
        `define virtual_interface
interface virtual_interface (input      wire    logic   clk);

                // Interface Communication Signals
                logic                               reset;
                logic                               monitor_start;
                logic                               test_in_progress;
                logic                               end_of_test;
                logic                               mismatch_detected;
                logic   [`INST_ADDR_WIDTH-1:0]      instr_addr;
                logic   [`INST_CODE_WIDTH-1:0]      instr_code;

                // Signals from Design Under Test
                logic   [`INST_CODE_WIDTH-1:0]      dut_instr;
                logic   [`INST_ADDR_WIDTH-1:0]      dut_pc;
                logic   [`REG_ADDR_WIDTH-1:0]       dut_reg_read_addr_1;
                logic   [`REG_ADDR_WIDTH-1:0]       dut_reg_read_addr_2;
                logic   [`REG_ADDR_WIDTH-1:0]       dut_reg_write_addr;
                logic   [`DATA_WIDTH-1:0]           dut_reg_read_data_1;
                logic   [`DATA_WIDTH-1:0]           dut_reg_read_data_2;
                logic   [`DATA_WIDTH-1:0]           dut_reg_write_data;
                logic   [`CTL_SGNL_WIDTH-1:0]       dut_ctl_op;
                logic                               dut_alu_zero;
                logic   [`DATA_WIDTH-1:0]           dut_alu_output;
                logic   [`DATA_WIDTH-1:0]           dut_imm_val;
                logic   [`DATA_WIDTH-1:0]           dut_mem_write_data;
                logic   [`MEM_ADDR_WIDTH-1:0]       dut_mem_addr;

                // Signals from Reference Model
                logic   [`INST_CODE_WIDTH-1:0]      ref_instr;
                logic   [`INST_ADDR_WIDTH-1:0]      ref_pc;
                logic   [`REG_ADDR_WIDTH-1:0]       ref_reg_read_addr_1;
                logic   [`REG_ADDR_WIDTH-1:0]       ref_reg_read_addr_2;
                logic   [`REG_ADDR_WIDTH-1:0]       ref_reg_write_addr;
                logic   [`DATA_WIDTH-1:0]           ref_reg_read_data_1;
                logic   [`DATA_WIDTH-1:0]           ref_reg_read_data_2;
                logic   [`DATA_WIDTH-1:0]           ref_reg_write_data;
                logic   [`CTL_SGNL_WIDTH-1:0]       ref_ctl_op;
                logic                               ref_alu_zero;
                logic   [`DATA_WIDTH-1:0]           ref_alu_output;
                logic   [`DATA_WIDTH-1:0]           ref_imm_val;
                logic   [`DATA_WIDTH-1:0]           ref_mem_write_data;
                logic   [`MEM_ADDR_WIDTH-1:0]       ref_mem_addr;
                logic                               ref_mem_write;
                logic                               ref_mem_read;
                logic                               ref_reg_write;
                logic                               ref_stall;
                logic                               ref_flush;
                logic                               ref_jump_link;
                logic   [`INST_CODE_WIDTH-1:0]      ref_ID_instr;
                logic   [`INST_CODE_WIDTH-1:0]      ref_EX_instr;
                logic   [`INST_ADDR_WIDTH-1:0]      ref_EX_pc;

        // Define interface signal input/output direction
```

```
        clocking cb @(posedge clk);
                default input #1step;

                input   dut_instr;
                input   dut_pc;
                input   dut_reg_read_addr_1;
                input   dut_reg_read_addr_2;
                input   dut_reg_write_addr;
                input   dut_reg_read_data_1;
                input   dut_reg_read_data_2;
                input   dut_reg_write_data;
                input   dut_ctl_op;
                input   dut_alu_zero;
                input   dut_alu_output;
                input   dut_imm_val;
                input   dut_mem_write_data;
                input   dut_mem_addr;

                input   ref_instr;
                input   ref_pc;
                input   ref_reg_read_addr_1;
                input   ref_reg_read_addr_2;
                input   ref_reg_write_addr;
                input   ref_reg_read_data_1;
                input   ref_reg_read_data_2;
                input   ref_reg_write_data;
                input   ref_ctl_op;
                input   ref_alu_zero;
                input   ref_alu_output;
                input   ref_imm_val;
                input   ref_mem_write_data;
                input   ref_mem_addr;
                input   ref_mem_write;
                input   ref_mem_read;
                input   ref_reg_write;
                input   ref_stall;
                input   ref_flush;
                input   ref_jump_link;
                input   ref_ID_instr;
                input   ref_EX_instr;
                input   ref_EX_pc;
        endclocking
endinterface
`endif
```

APPENDIX U: Source Code of Verification Environment – UVM Environment

```
`include        "uvm_pkg.sv"
`include        "uvm_macros.svh"
`include        "agent.sv"
`include        "scoreboard.sv"
`include        "coverage.sv"
import          uvm_pkg::*;
`ifndef env
        `define env
class   env             extends uvm_env;
        `uvm_component_utils(env)

        function new(string name="env", uvm_component parent=null);
                super.new(name,parent);
        endfunction

        agent                   agent_inst;
        scoreboard              scoreboard_inst;
        coverage                coverage_inst;

        virtual function void   build_phase(uvm_phase phase);
                super.build_phase(phase);
                agent_inst      = agent::type_id::create("agent_inst", this);
                scoreboard_inst = scoreboard::type_id::create("scoreboard_inst",this);
                coverage_inst   = coverage::type_id::create("coverage_inst",this);
        endfunction

        virtual function void   connect_phase(uvm_phase phase);
                super.connect_phase(phase);
                agent_inst.monitor_inst.analysis_port.connect(scoreboard_inst.m_analysis_imp);
                agent_inst.monitor_inst.analysis_port.connect(coverage_inst.analysis_export);
        endfunction
endclass
`endif
```

APPENDIX V: Source Code of Verification Environment – UVM Agent

```
`include        "uvm_pkg.sv"
`include        "uvm_macros.svh"
`include        "driver.sv"
`include        "monitor.sv"
import          uvm_pkg::*;
`ifndef agent
        `define agent
class   agent           extends uvm_agent;
        `uvm_component_utils(agent)

        function new(string name="agent", uvm_component parent=null);
                super.new(name, parent);
        endfunction

        driver                          driver_inst;
        monitor                         monitor_inst;
        uvm_sequencer   #(seq_item)     seqr_inst;

        virtual function void           build_phase(uvm_phase phase);
                super.build_phase(phase);
                seqr_inst       = uvm_sequencer#(seq_item)::type_id::create("seqr_inst",this);
                driver_inst     = driver::type_id::create("driver_inst",this);
                monitor_inst    = monitor::type_id::create("monitor_inst",this);
        endfunction

        virtual function void           connect_phase(uvm_phase phase);
                super.connect_phase(phase);
                driver_inst.seq_item_port.connect(seqr_inst.seq_item_export);
        endfunction
endclass
`endif
```

APPENDIX W: Source Code of Verification Environment – UVM Driver

```
`include        "uvm_pkg.sv"
`include        "uvm_macros.svh"
`include        "seq_item.sv"
import          uvm_pkg::*;
`ifndef driver
        `define driver
class   driver          extends uvm_driver #(seq_item);
        `uvm_component_utils(driver)

        function new(string name="driver", uvm_component parent=null);
                super.new(name, parent);
        endfunction

        virtual virtual_interface       interface_instance;

        virtual function void           build_phase(uvm_phase phase);
                super.build_phase(phase);
                if(!uvm_config_db#(virtual virtual_interface)::get(this,"","virtual_interface",interface_instance))
                        `uvm_fatal("DRV", "Unable to access virtual interface on verification environment")
        endfunction

        virtual task run_phase(uvm_phase phase);
                super.run_phase(phase);
                forever begin
                        // Database parameter variables
                        bit     batch_test;
                        int     current_test_index;
                        int     batch_test_max;

                        seq_item        transaction;
                        seq_item_port.get_next_item(transaction);
                        load_program(transaction);
                        seq_item_port.item_done();

                        // Obtain parameters from database for checking
                        if(!uvm_config_db#(bit)::get(null,"uvm_test_top","batch_test",batch_test))
                                batch_test = 0;
                        if(!uvm_resource_db#(int)::read_by_name("uvm_test_top","current_test_index",current_test_index))
                                current_test_index = 0;
                        if(!uvm_resource_db#(int)::read_by_name("uvm_test_top","batch_test_max",batch_test_max))
                                batch_test_max = 0;

                        // If batch test index reaches the maximum number of seeds found, end batch test
                        if((batch_test_max - 1) == current_test_index && batch_test)    begin
                                if(!uvm_resource_db#(bit)::write_by_name("uvm_test_top","batch_test_in_progress",0)) begin
                                        `uvm_fatal("ERROR","Fail to update UVM Resource Database");
                                end
                        end
                end
        endtask

        // Load test program onto processor instruction memory
        virtual task load_program(seq_item transaction);
                @(interface_instance.cb);
                        if(transaction.instr_gen_completion)    begin
                                $display("Loading Program into ROM");
                                $readmemh("PROM.txt",testbench.riscv.rom);
                                $readmemh("PROM.txt",testbench.riscv_ref.rom);
                                #100;
                                $display("Program Successfully Loaded");
                                transaction.instr_gen_completion = 0;
                        end
        endtask
endclass
`endif
```

APPENDIX X: Source Code of Verification Environment – UVM Sequencer

```
`include "uvm_pkg.sv"
`include "uvm_macros.svh"
`include "seq_item.sv"
`include "parameter_list.sv"
import    uvm_pkg::*;
`ifndef seq
        `define seq
class   seq             extends uvm_sequence;
        `uvm_object_utils(seq)


        // Declare parameter variables
        bit             force_gen;
        bit             batch_test;
        bit             directed_test;
        int             test_seed;
        int             instr_amt;
        int             current_test_index;
        string          seed_file;

        bit             skip_gen;
        string          argument_pass;

        function new(string name="seq");
                super.new(name);
        endfunction

        virtual task    body();
                int             fh;

                // Obtain config and resource parameter values
                if(!uvm_config_db#(bit)::get(null,"uvm_test_top","force_gen",force_gen))
                        force_gen = 0;
                if(!uvm_config_db#(bit)::get(null,"uvm_test_top","batch_test",batch_test))
                        batch_test = 0;
                if(!uvm_config_db#(bit)::get(null,"uvm_test_top","directed_test",directed_test))
                        directed_test = 0;
                if(!uvm_config_db#(int)::get(null,"uvm_test_top","test_seed",test_seed))
                        test_seed = 0;
                if(!uvm_config_db#(int)::get(null,"uvm_test_top","instr_amount",instr_amt))
                        instr_amt = 500;
                if(!uvm_config_db#(string)::get(null,"uvm_test_top","seed_file",seed_file))
                        seed_file = "SEED.txt";
                if(!uvm_resource_db#(int)::read_by_name("uvm_test_top","current_test_index",current_test_index))
                        current_test_index = 0;

                // Passing of Test Seed and Creation of Seed Directory
                if(directed_test) // Directed
                        directory_creation("directed_test");
                else begin
                        if(batch_test) // Batch Seed
                                read_seed();
                        else // Singular Seed
                                process::self().srandom(test_seed);
                        $display("Test Seed:                                                                    %s", test_seed);
                        $sformat(argument_pass, "%s", test_seed);
                        directory_creation(argument_pass);
                end
                clear_log();

                if(directed_test) begin // Check for directed test case test file
                        fh = $fopen("TEST.txt","r");
                        if(fh)
                                $display("Translating test instructions from direct test file");
                        else
                                `uvm_fatal("ERROR","Unable to access TEST.txt");
                        $fclose(fh);
                end
                else begin
                        if(!force_gen)
                                retrieve_repository(argument_pass);
                        else begin
                                $display("Forcing test case generation");
                                skip_gen = 0;
                        end
                end
                instruction_code_generate(); // Generate the instruciton code
        endtask

        // Simple function for creating directory
        function void directory_creation(string destination);
                string file_creation;

                file_creation = {"mkdir \"./test_results/",destination,"/\""};
                $system(file_creation);
        endfunction

        // Simple function for clearing log files
        function void clear_log();
                int     fh;

                fh = $fopen("ASM.txt", "w");
```

```
		$fclose(fh);
		fh = $fopen("PROM.txt", "w");
		$fclose(fh);
		fh = $fopen("INSTR.txt", "w");
		$fclose(fh);
		fh = $fopen("IFID.txt", "w");
		$fclose(fh);
		fh = $fopen("IDEX.txt", "w");
		$fclose(fh);
		fh = $fopen("EXMEM.txt", "w");
		$fclose(fh);
		fh = $fopen("MEMWB.txt", "w");
		$fclose(fh);
		fh = $fopen("PIPELINE.txt", "w");
		$fclose(fh);
		fh = $fopen("FLUSH.txt","w");
		$fclose(fh);
		fh = $fopen("JUMP.txt","w");
		$fclose(fh);
	endfunction

	// Simple function for reading test seed on batch test seed file
	function void read_seed();
		int	fh;
		int	code;
		string	dump;

		fh = $fopen(seed_file,"r");
		if(fh != 0) begin
			for(int i = 0; i < current_test_index + 1;)	begin
				code = $fscanf(fh,"%s",test_seed);
				if(test_seed == "\//")	begin
					code = $fgets(dump,fh);
					continue;
				end
				else	i++;
			end
			$fclose(fh);
		end
		else
			`uvm_fatal("ERROR","Unable to access seed file");
	endfunction

	// Simple function for cloning test from repository
	function void retrieve_repository(string destination);
		int	fh;
		string	file_pointer;
		string	file_path_1;
		string	file_handling_1;
		string	file_path_2;
		string	file_handling_2;

		file_pointer = {"./test_repo/",destination,"/ASM.txt"};
		file_path_1 = {"D:\\FinalYearProject\\Coding\\uvm\\test_repo\\",destination,"\\ASM.txt"};
		file_handling_1 = {"copy ", file_path_1};
		file_path_2 = {"D:\\FinalYearProject\\Coding\\uvm\\test_repo\\",destination,"\\PROM.txt"};
		file_handling_2 = {"copy ", file_path_2};
		$display("Checking repository for pre-existing test case...");
		fh = $fopen(file_pointer,"r");
		if(fh) begin
			$system(file_handling_1);
			$system(file_handling_2);
			$display("Test case cloned from repository");
			skip_gen = 1;
		end
		else begin
			$display("No existing test case found, generating new test case");
			skip_gen = 0;
		end
		$fclose(fh);
	endfunction;

	// Simple function for storing test to repositry
	function void store_repository(string destination);
		int	fh;
		string	file_pointer;
		string	file_path;
		string	file_handling_1;
		string	file_handling_2;
		string	file_creation;

		file_pointer = {"./test_repo/",destination,"/ASM.txt"};
		fh = $fopen(file_pointer,"r");
		if(!fh) begin // Create directory in repositry
			file_creation = {"mkdir \"./test_repo/",destination,"/\""};
			$system(file_creation);
		end
		$fclose(fh);
		file_path = {"D:\\FinalYearProject\\Coding\\uvm\\test_repo\\",destination,"\\"};
		file_handling_1 = {"copy ASM.txt ",file_path};
		file_handling_2 = {"copy PROM.txt ",file_path};
		$system(file_handling_1);
		$system(file_handling_2);
		$display("Testcase cloned and stored into repository");
	endfunction

	// Generate instruction code
	task instruction_code_generate();
		for (int i = 0, logic [`INST_ADDR_WIDTH-1:0] stored_instr_addr = 0, bit completion = 0; !completion; i++) begin
			seq_item transaction = seq_item::type_id::create("transaction");
			start_item(transaction);
			if(directed_test) begin // Translate and store to repository
				transaction.translation();
				store_repository("directed_test");
				completion = 1;
```

```
                                transaction.instr_gen_completion = 1;
                                $display("Completed translation of directed test");
                        end
                        else begin // Generate randomized instruction
                                if(!skip_gen)   begin
                                        transaction.instr_addr = stored_instr_addr;
                                        transaction.max_range = instr_amt;
                                        transaction.current_instr_number = i;
                                        transaction.specification();
                                        stored_instr_addr = transaction.instr_addr + `INST_ADDR_SUM;
                                end
                                if (( i == (instr_amt - 1)) || skip_gen) begin // If specified amount of instruction generated
                                        uvm_config_db#(int)::set(null,"uvm_test_top","test_seed",test_seed);
                                        if(!skip_gen)   begin
                                                $display("RISC-V Instruction Test Set Generation Completed");
                                                $display("Total of %0d Instruction Code Generated", instr_amt);
                                                $sformat(argument_pass, "%s", test_seed);
                                                store_repository(argument_pass);
                                                $display("Testcase cloned and stored into repository");
                                        end
                                        completion = 1;
                                        transaction.instr_gen_completion = 1;
                                end
                        end
                        finish_item(transaction);
                end
        endtask
endclass
`endif
```

APPENDIX Y: Source Code of Verification Environment – UVM Sequence Item

```
`include "uvm_pkg.sv"
`include "uvm_macros.svh"
`include "parameter_list.sv"
import    uvm_pkg::*;
`ifndef seq_item
        `define seq_item
class   seq_item extends uvm_sequence_item;
        `uvm_object_utils(seq_item)

        function new(string name = "seq_item");
                super.new(name);
        endfunction

        int     code;
        int     instr_count;
        int     max_range;
        int     current_instr_number;
        logic   instr_gen_completion;
        logic   end_of_test;
        logic   mismatch_log_complete;
        logic   mem_check;

        logic   [`INST_CODE_WIDTH-1:0]  instr_code;
        logic   [`INST_ADDR_WIDTH-1:0]  instr_addr;

        // Transactions receiveed from DUT
        logic   [`INST_CODE_WIDTH-1:0]  dut_instr;
        logic   [`INST_ADDR_WIDTH-1:0]  dut_pc;
        logic   [`REG_ADDR_WIDTH-1:0]   dut_reg_read_addr_1;
        logic   [`REG_ADDR_WIDTH-1:0]   dut_reg_read_addr_2;
        logic   [`REG_ADDR_WIDTH-1:0]   dut_reg_write_addr;
        logic   [`DATA_WIDTH-1:0]       dut_reg_read_data_1;
        logic   [`DATA_WIDTH-1:0]       dut_reg_read_data_2;
        logic   [`DATA_WIDTH-1:0]       dut_reg_write_data;
        logic   [`CTL_SGNL_WIDTH-1:0]   dut_ctl_op;
        logic   [`ALU_OP_WIDTH-1:0]     dut_alu_op;
        logic                           dut_alu_zero;
        logic   [`DATA_WIDTH-1:0]       dut_alu_output;
        logic   [`DATA_WIDTH-1:0]       dut_imm_val;
        logic   [`DATA_WIDTH-1:0]       dut_mem_read_data;
        logic   [`DATA_WIDTH-1:0]       dut_mem_write_data;
        logic   [`MEM_ADDR_WIDTH-1:0]   dut_mem_addr;

        // Transactions received from REF
        logic   [`INST_CODE_WIDTH-1:0]  ref_instr;
        logic   [`INST_ADDR_WIDTH-1:0]  ref_pc;
        logic   [`REG_ADDR_WIDTH-1:0]   ref_reg_read_addr_1;
        logic   [`REG_ADDR_WIDTH-1:0]   ref_reg_read_addr_2;
        logic   [`REG_ADDR_WIDTH-1:0]   ref_reg_write_addr;
        logic   [`DATA_WIDTH-1:0]       ref_reg_read_data_1;
        logic   [`DATA_WIDTH-1:0]       ref_reg_read_data_2;
        logic   [`DATA_WIDTH-1:0]       ref_reg_write_data;
        logic   [`CTL_SGNL_WIDTH-1:0]   ref_ctl_op;
        logic   [`ALU_OP_WIDTH-1:0]     ref_alu_op;
        logic                           ref_alu_zero;
        logic   [`DATA_WIDTH-1:0]       ref_alu_output;
        logic   [`DATA_WIDTH-1:0]       ref_imm_val;
        logic   [`DATA_WIDTH-1:0]       ref_mem_read_data;
        logic   [`DATA_WIDTH-1:0]       ref_mem_write_data;
        logic   [`MEM_ADDR_WIDTH-1:0]   ref_mem_addr;
        logic                           ref_mem_write;
        logic                           ref_mem_read;
        logic                           ref_reg_write;
        logic                           ref_stall;
        logic                           ref_flush;
        logic                           ref_jump_link;
        logic   [`INST_CODE_WIDTH-1:0]  ref_ID_instr;
        logic   [`INST_CODE_WIDTH-1:0]  ref_EX_instr;
        logic   [`INST_ADDR_WIDTH-1:0]  ref_EX_pc;

        // Possible fields for instructions
        bit     [`OPCODE_WIDTH-1:0]     opcode;
        bit     [21:0]                  immediate_value;
        rand bit [`FUNCT3_WIDTH-1:0]    funct3          = $urandom_range(7,0);
        rand bit [`FUNCT7_WIDTH-1:0]    funct7          = $urandom_range(127,0);
        rand bit [`REG_ADDR_WIDTH-1:0]  rs1             = $urandom_range(31,0);
        rand bit [`REG_ADDR_WIDTH-1:0]  rs2             = $urandom_range(31,0);
        rand bit [`REG_ADDR_WIDTH-1:0]  rd              = $urandom_range(31,1);

        // Miscellaneous variables
        bit     [7:0]  [`OPCODE_WIDTH-1:0]      possible_opcode;
        int                                     randomizer;
        int                                     fh;

        // Specification for a valid instruction code
        virtual function void specification();

                // Database parameter variables
                string  instr_type;

                // Obtain parameters from database
                if(!uvm_config_db#(string)::get(null,"uvm_test_top","instr_type",instr_type))
                        instr_type = "R_I_L_U_UJ_S_SB_J";

                for(int pointer = 0; pointer < instr_type.len(); pointer ++) begin
                        case(instr_type[pointer])
                        "R"     : begin
                                        possible_opcode[randomizer] = `R_OPCODE;
                                        randomizer ++;
                                  end
                        "I"     : begin
                                        possible_opcode[randomizer] = `I_OPCODE;
                                        randomizer ++;
                                  end
                        "L"     : begin
                                        possible_opcode[randomizer] = `LOAD_OPCODE;
                                        randomizer ++;
                                  end
                        "J"     : begin
```

```
                                possible_opcode[randomizer] = `JALR_OPCODE;
                                randomizer ++;
                        end
                "U"     : begin
                                case(instr_type[pointer + 1])
                                "J"     : begin
                                                possible_opcode[randomizer] = `J_OPCODE;
                                                randomizer ++;
                                                pointer ++;
                                        end
                                default : begin
                                                possible_opcode[randomizer] = `U_OPCODE;
                                                randomizer ++;
                                        end
                                endcase
                        end
                "S"     : begin
                                case(instr_type[pointer + 1])
                                "B"     : begin
                                                possible_opcode[randomizer] = `SB_OPCODE;
                                                randomizer ++;
                                                pointer ++;
                                        end
                                default : begin
                                                possible_opcode[randomizer] = `S_OPCODE;
                                                randomizer ++;
                                        end
                                endcase
                        end
                "_"     : continue;
                default : `uvm_fatal("ERROR","SEQ_ITEM_ERROR: invalid INSTR_TYPE specified")
                endcase
        end
        opcode = possible_opcode[$urandom_range((randomizer-1),0)];

        case(opcode)
        `R_OPCODE       : begin
                                randcase
                                1: funct3 = `ADD_FUNCT3;
                                1: funct3 = `SLL_FUNCT3;
                                1: funct3 = `SLT_FUNCT3;
                                1: funct3 = `SLTU_FUNCT3;
                                1: funct3 = `XOR_FUNCT3;
                                1: funct3 = `SR_FUNCT3;
                                1: funct3 = `OR_FUNCT3;
                                1: funct3 = `AND_FUNCT3;
                                endcase
                        end
        `LOAD_OPCODE    : begin
                                randcase
                                1: funct3 = `LB_FUNCT3;
                                1: funct3 = `LH_FUNCT3;
                                1: funct3 = `LW_FUNCT3;
                                1: funct3 = `LBU_FUNCT3;
                                1: funct3 = `LHU_FUNCT3;
                                endcase
                        end
        `I_OPCODE       : begin
                                randcase
                                1: funct3 = `ADD_FUNCT3;
                                1: funct3 = `SLL_FUNCT3;
                                1: funct3 = `SLT_FUNCT3;
                                1: funct3 = `SLTU_FUNCT3;
                                1: funct3 = `XOR_FUNCT3;
                                1: funct3 = `SR_FUNCT3;
                                1: funct3 = `OR_FUNCT3;
                                1: funct3 = `AND_FUNCT3;
                                endcase
                        end
        `S_OPCODE       : begin
                                randcase
                                1: funct3 = `SB_FUNCT3;
                                1: funct3 = `SH_FUNCT3;
                                1: funct3 = `SW_FUNCT3;
                                endcase
                        end
        `SB_OPCODE      : begin
                                randcase
                                1: funct3 = `BEQ_FUNCT3;
                                1: funct3 = `BNE_FUNCT3;
                                1: funct3 = `BLT_FUNCT3;
                                1: funct3 = `BGE_FUNCT3;
                                1: funct3 = `BLTU_FUNCT3;
                                1: funct3 = `BGEU_FUNCT3;
                                endcase
                        end
        `JALR_OPCODE    : funct3 = `JALR_FUNCT3;
        endcase

        case(opcode)
        `R_OPCODE       : begin
                                case (funct3)
                                `ADD_FUNCT3,
                                `SR_FUNCT3: begin
                                                randcase
                                                1: funct7 = `DEFAULT_FUNCT7;
                                                1: funct7 = `ALT_FUNCT7;
                                                endcase
                                        end
                                default:        funct7 = `DEFAULT_FUNCT7;
                                endcase
                        end
        `I_OPCODE       : begin
                                case (funct3)
                                `SLL_FUNCT3:    funct7 = `DEFAULT_FUNCT7;
                                `SR_FUNCT3:     begin
                                                randcase
                                                1: funct7 = `DEFAULT_FUNCT7;
                                                1: funct7 = `ALT_FUNCT7;
                                                endcase
                                        end
                                endcase
                        end
        endcase

        // Branch Address Constraint
        case(opcode)
        `J_OPCODE:    begin
                        {instr_code[31],instr_code[19:12],instr_code[20],instr_code[30:21]} =
                        $urandom_range(0,4) * `INST_ADDR_SUM;
                        rd = $urandom_range(31,0);
                        instr_code[11:0] = {rd, opcode};
```

```
                        end
                `SB_OPCODE:   begin
                                {instr_code[31],instr_code[7],instr_code[30:25],instr_code[11:8]} =
                                $urandom_range(8,4) * `INST_ADDR_SUM;
                                instr_code[24:12] = {rs2,rs1,funct3};
                                instr_code[6:0] = opcode;
                              end
                `JALR_OPCODE: begin
                                immediate_value = $urandom_range(current_instr_number + 8, current_instr_number + 4) * `INST_ADDR_SUM;
                                rs1 = 0;
                                instr_code = {immediate_value[11:0],rs1,funct3,rd,opcode};
                              end
                default:      instr_code = {funct7,rs2,rs1,funct3,rd,opcode};
                endcase

                // Output instruction code and address
                fh = $fopen("ASM.txt", "a+");
                $fdisplay(fh, "%8h %8h", instr_addr, instr_code);
                $fclose(fh);
                // Output instruction code in bytes
                fh = $fopen("PROM.txt","a+");
                $fdisplay(fh, "%2h %2h %2h %2h", instr_code[31:24], instr_code[23:16], instr_code[15:8], instr_code[7:0]);
                $fclose(fh);
        endfunction

        // Translation of assembly code to machine language
        virtual function void translation();

                //Directed test variables
                int             f_out;
                int             shift_type;
                int             store_type;
                int             load_type;
                int             min_range;
                int             max_range;
                string          dv_instr_type;
                string          dump;
                string          hold;
                string          assembly_code;
                string          operand;

                fh = $fopen("TEST.txt","r");
                if(fh == 0)
                        `uvm_fatal("ERROR","Unable to access TEST.txt");
                instr_addr = 0;
                while(!$feof(fh))       begin
                        shift_type = 0;
                        store_type = 0;
                        load_type = 0;
                        dv_instr_type = "";
                        assembly_code = "";
                        code = $fscanf(fh,"%s",assembly_code);
                        case(assembly_code)
                        "add":  begin
                                        opcode = `R_OPCODE;
                                        funct3 = `ADD_FUNCT3;
                                        funct7 = `DEFAULT_FUNCT7;
                                        dv_instr_type = "R";
                                end
                        "sub":  begin
                                        opcode = `R_OPCODE;
                                        funct3 = `ADD_FUNCT3;
                                        funct7 = `ALT_FUNCT7;
                                        dv_instr_type = "R";
                                end
                        "sll":  begin
                                        opcode = `R_OPCODE;
                                        funct3 = `SLL_FUNCT3;
                                        funct7 = `DEFAULT_FUNCT7;
                                        dv_instr_type = "R";
                                end
                        "slt":  begin
                                        opcode = `R_OPCODE;
                                        funct3 = `SLT_FUNCT3;
                                        funct7 = `DEFAULT_FUNCT7;
                                        dv_instr_type = "R";
                                end
                        "sltu": begin
                                        opcode = `R_OPCODE;
                                        funct3 = `SLTU_FUNCT3;
                                        funct7 = `DEFAULT_FUNCT7;
                                        dv_instr_type = "R";
                                end
                        "srl":  begin
                                        opcode = `R_OPCODE;
                                        funct3 = `SR_FUNCT3;
                                        funct7 = `DEFAULT_FUNCT7;
                                        dv_instr_type = "R";
                                end
                        "sra":  begin
                                        opcode = `R_OPCODE;
                                        funct3 = `SR_FUNCT3;
                                        funct7 = `ALT_FUNCT7;
                                        dv_instr_type = "R";
                                end
                        "xor":  begin
                                        opcode = `R_OPCODE;
                                        funct3 = `XOR_FUNCT3;
                                        funct7 = `DEFAULT_FUNCT7;
                                        dv_instr_type = "R";
                                end
                        "and":  begin
                                        opcode = `R_OPCODE;
                                        funct3 = `AND_FUNCT3;
                                        funct7 = `DEFAULT_FUNCT7;
                                        dv_instr_type = "R";
                                end
                        "or":   begin
                                        opcode = `R_OPCODE;
                                        funct3 = `OR_FUNCT3;
                                        funct7 = `DEFAULT_FUNCT7;
                                        dv_instr_type = "R";
                                end
                        "addi": begin
                                        opcode = `I_OPCODE;
                                        funct3 = `ADD_FUNCT3;
                                        dv_instr_type = "I";
                                end
                        "slti": begin
                                        opcode = `I_OPCODE;
```

```
                funct3 = `SLT_FUNCT3;
                dv_instr_type = "I";
        end
"sltiu": begin
                opcode = `I_OPCODE;
                funct3 = `SLTU_FUNCT3;
                dv_instr_type = "I";
        end
"xori": begin
                opcode = `I_OPCODE;
                funct3 = `XOR_FUNCT3;
                dv_instr_type = "I";
        end
"ori":  begin
                opcode = `I_OPCODE;
                funct3 = `OR_FUNCT3;
                dv_instr_type = "I";
        end
"andi": begin
                opcode = `I_OPCODE;
                funct3 = `AND_FUNCT3;
                dv_instr_type = "I";
        end
"slli": begin
                opcode = `I_OPCODE;
                funct3 = `SLL_FUNCT3;
                dv_instr_type = "I";
                shift_type = 1;
                immediate_value[11:5] = `DEFAULT_FUNCT7;
        end
"srli": begin
                opcode = `I_OPCODE;
                funct3 = `SR_FUNCT3;
                dv_instr_type = "I";
                shift_type = 1;
                immediate_value[11:5] = `DEFAULT_FUNCT7;
        end
"srai": begin
                opcode = `I_OPCODE;
                funct3 = `SR_FUNCT3;
                dv_instr_type = "I";
                shift_type = 1;
                immediate_value[11:5] = `ALT_FUNCT7;
        end
"lb":   begin
                opcode = `LOAD_OPCODE;
                funct3 = `LB_FUNCT3;
                dv_instr_type = "I";
                load_type = 1;
        end
"lh":   begin
                opcode = `LOAD_OPCODE;
                funct3 = `LH_FUNCT3;
                dv_instr_type = "I";
                load_type = 1;
        end
"lw":   begin
                opcode = `LOAD_OPCODE;
                funct3 = `LW_FUNCT3;
                dv_instr_type = "I";
                load_type = 1;
        end
"lbu":  begin
                opcode = `LOAD_OPCODE;
                funct3 = `LBU_FUNCT3;
                dv_instr_type = "I";
                load_type = 1;
        end
"lhu":  begin
                opcode = `LOAD_OPCODE;
                funct3 = `LHU_FUNCT3;
                dv_instr_type = "I";
                load_type = 1;
        end
"sb":   begin
                opcode = `S_OPCODE;
                funct3 = `SB_FUNCT3;
                dv_instr_type = "S";
                store_type = 1;
        end
"sh":   begin
                opcode = `S_OPCODE;
                funct3 = `SH_FUNCT3;
                dv_instr_type ="S";
                store_type = 1;
        end
"sw":   begin
                opcode = `S_OPCODE;
                funct3 = `SW_FUNCT3;
                dv_instr_type = "S";
                store_type = 1;
        end
"beq":  begin
                opcode = `SB_OPCODE;
                funct3 = `BEQ_FUNCT3;
                dv_instr_type = "B";
        end
"bne":  begin
                opcode = `SB_OPCODE;
                funct3 = `BNE_FUNCT3;
                dv_instr_type = "B";
        end
"blt":  begin
                opcode = `SB_OPCODE;
                funct3 = `BLT_FUNCT3;
                dv_instr_type = "B";
        end
"bge":  begin
                opcode = `SB_OPCODE;
                funct3 = `BGE_FUNCT3;
                dv_instr_type = "B";
        end
"bltu": begin
                opcode = `SB_OPCODE;
                funct3 = `BLTU_FUNCT3;
                dv_instr_type = "B";
        end
"bgeu": begin
                opcode = `SB_OPCODE;
```

```
                funct3 = `BGEU_FUNCT3;
                dv_instr_type = "B";
        end
"jal":  begin
                opcode = `J_OPCODE;
                dv_instr_type = "J";
        end
"jalr": begin
                opcode = `JALR_OPCODE;
                funct3 = `JALR_FUNCT3;
                dv_instr_type = "I";
                load_type = 1; //Similar assembly format
        end
"lui":  begin
                opcode = `U_OPCODE;
                dv_instr_type = "U";
        end
"\//":  begin
                code = $fgets(dump,fh);
                continue;
        end
"nop":  begin
                instr_code = `NOP_INST_CODE;
                f_out = $fopen("ASM.txt", "a+");
                $fdisplay(f_out, "%8h %8h", instr_addr, instr_code);
                $fclose(f_out);
                f_out = $fopen("PROM.txt","a+");
                $fdisplay(f_out, "%2h %2h %2h %2h", instr_code[31:24], instr_code[23:16], instr_code[15:8], instr_code[7:0]);
                $fclose(f_out);
                instr_addr = instr_addr + `INST_ADDR_SUM;
                continue;
        end
"end":  break;
default:begin
                $display("Unknown assembly operation referenced: %s", assembly_code);
                `uvm_fatal("ERROR","Unknown assembly operation referenced")
        end
endcase
code = $fscanf(fh,"%s",operand);
min_range = -1;
for(int i = 0; i < operand.len(); i++)  begin
        case(operand[i])
        "x":    continue;
        " ":    continue;
        ",":    max_range = i - 1;
        default:begin
                        if(min_range < 0)
                                min_range = i;
                end
        endcase
end
operand = operand.substr(min_range,max_range);
if(dv_instr_type  == "S")
        rs2 = operand.atoi();
else if(dv_instr_type == "B")
        rs1 = operand.atoi();
else
        rd = operand.atoi();
code = $fscanf(fh,"%s",operand);
if(dv_instr_type != "J" && dv_instr_type != "U")        begin
        min_range = -1;
        for(int i = 0; i < operand.len(); i++)  begin
                case(operand[i])
                "x":    continue;
                " ":    continue;
                "(":    begin
                                max_range = i - 1;
                                hold = operand.substr(min_range,max_range);
                                immediate_value [11:0] = hold.atoi();
                                min_range = -1;
                        end
                ")",
                ",":    max_range = i - 1;
                default:begin
                                if(min_range < 0)
                                        min_range = i;
                        end
                endcase
        end
        operand = operand.substr(min_range,max_range);
        if(dv_instr_type == "B")
                rs2 = operand.atoi();
        else
                rs1 = operand.atoi();
end
else
        immediate_value [20:1] = operand.atoi();
if(load_type == 0 && store_type == 0 && dv_instr_type != "J" && dv_instr_type != "U")   begin
        code = $fscanf(fh,"%s",operand);
        min_range = -1;
        max_range = operand.len() - 1;
        for(int i = 0; i < operand.len(); i++)  begin
                case(operand[i])
                "x",
                " ":    continue;
                default:begin
                                if(min_range < 0)
                                        min_range = i;
                        end
                endcase
        end
        operand = operand.substr(min_range,max_range);
        case(dv_instr_type)
        "R":    rs2 = operand.atoi();
        "I":    begin
                        if(shift_type == 1)
                                immediate_value [4:0] = operand.atoi();
                        else
                                immediate_value [11:0] = operand.atoi();
                end
        "B":    immediate_value [13:1] = operand.atoi();
        endcase
end
case(dv_instr_type)
"R":    instr_code = {funct7,rs2,rs1,funct3,rd,opcode};
"I":    instr_code = {immediate_value[11:0],rs1,funct3,rd,opcode};
"S":    instr_code = {immediate_value[11:5],rs2,rs1,funct3,
                      immediate_value[4:0],opcode};
```

```
                        "B":    instr_code = {immediate_value[12],immediate_value[10:5],
                                              rs2,rs1,funct3,immediate_value[4:1],immediate_value[11],opcode};
                        "J":    instr_code = {immediate_value[20],immediate_value[10:1],immediate_value[11],
                                               immediate_value[19:12],rd,opcode};
                        "U":    instr_code = {immediate_value[20:1],rd,opcode};
                        endcase
                        f_out = $fopen("ASM.txt", "a+");
                        $fdisplay(f_out, "%8h %8h", instr_addr, instr_code);
                        $fclose(f_out);
                        f_out = $fopen("PROM.txt","a+");
                        $fdisplay(f_out, "%2h %2h %2h %2h", instr_code[31:24], instr_code[23:16], instr_code[15:8], instr_code[7:0]);
                        $fclose(f_out);
                        instr_addr = instr_addr + 4;
                end
        endfunction
endclass
`endif
```

APPENDIX Z: Source Code of Verification Environment – UVM Monitor

```
`include        "uvm_pkg.sv"
`include        "uvm_macros.svh"
`include        "seq_item.sv"
import          uvm_pkg::*;
`ifndef monitor
        `define monitor
class   monitor         extends uvm_monitor;
        `uvm_component_utils(monitor)

        bit     run_all;

        function new(string name="monitor", uvm_component parent=null);
                super.new(name, parent);
        endfunction

        uvm_analysis_port #(seq_item)   analysis_port;
        virtual virtual_interface       interface_instance;

        virtual function void           build_phase(uvm_phase phase);
                super.build_phase(phase);
                if(!uvm_config_db#(virtual virtual_interface)::get(this,"","virtual_interface",interface_instance))
                        `uvm_fatal("MON", "Unable to access virtual interface on verification environment")
                analysis_port = new ("analysis_port",this);
                if(!uvm_config_db#(bit)::get(null,"uvm_test_top","run_all",run_all))
                        run_all = 0;
        endfunction

        virtual task run_phase(uvm_phase phase);
                super.run_phase(phase);
                forever begin
                @(interface_instance.cb);
                        if (interface_instance.monitor_start) begin
                                seq_item transaction = seq_item::type_id::create("transaction");
                                // DUT Transactions
                                transaction.dut_instr           = interface_instance.dut_instr;
                                transaction.dut_pc              = interface_instance.dut_pc;
                                transaction.dut_reg_read_addr_1 = interface_instance.dut_reg_read_addr_1;
                                transaction.dut_reg_read_addr_2 = interface_instance.dut_reg_read_addr_2;
                                transaction.dut_reg_read_data_1 = interface_instance.dut_reg_read_data_1;
                                transaction.dut_reg_read_data_2 = interface_instance.dut_reg_read_data_2;
                                transaction.dut_imm_val         = interface_instance.dut_imm_val;
                                transaction.dut_alu_output      = interface_instance.dut_alu_output;
                                transaction.dut_alu_zero        = interface_instance.dut_alu_zero;
                                transaction.dut_ctl_op          = interface_instance.dut_ctl_op;
                                transaction.dut_reg_write_addr  = interface_instance.dut_reg_write_addr;
                                transaction.dut_reg_write_data  = interface_instance.dut_reg_write_data;
                                transaction.dut_mem_write_data  = interface_instance.dut_mem_write_data;
                                transaction.dut_mem_addr        = interface_instance.dut_mem_addr;
                                // REF Transactions
                                transaction.instr_addr          = interface_instance.instr_addr;
                                transaction.ref_instr           = interface_instance.ref_instr;
                                transaction.ref_pc              = interface_instance.ref_pc;
                                transaction.ref_reg_read_addr_1 = interface_instance.ref_reg_read_addr_1;
                                transaction.ref_reg_read_addr_2 = interface_instance.ref_reg_read_addr_2;
                                transaction.ref_reg_read_data_1 = interface_instance.ref_reg_read_data_1;
                                transaction.ref_reg_read_data_2 = interface_instance.ref_reg_read_data_2;
                                transaction.ref_imm_val         = interface_instance.ref_imm_val;
                                transaction.ref_alu_output      = interface_instance.ref_alu_output;
                                transaction.ref_alu_zero        = interface_instance.ref_alu_zero;
                                transaction.ref_ctl_op          = interface_instance.ref_ctl_op;
                                transaction.ref_reg_write_addr  = interface_instance.ref_reg_write_addr;
                                transaction.ref_reg_write_data  = interface_instance.ref_reg_write_data;
                                transaction.ref_reg_write       = interface_instance.ref_reg_write;
                                transaction.ref_mem_write_data  = interface_instance.ref_mem_write_data;
                                transaction.ref_mem_addr        = interface_instance.ref_mem_addr;
                                transaction.ref_mem_read        = interface_instance.ref_mem_read;
                                transaction.ref_mem_write       = interface_instance.ref_mem_write;
                                transaction.ref_stall           = interface_instance.ref_stall;
                                transaction.ref_flush           = interface_instance.ref_flush;
                                transaction.ref_jump_link       = interface_instance.ref_jump_link;
                                transaction.ref_ID_instr        = interface_instance.ref_ID_instr;
                                transaction.ref_EX_instr        = interface_instance.ref_EX_instr;
                                transaction.ref_EX_pc           = interface_instance.ref_EX_pc;
                                transaction.end_of_test         = interface_instance.end_of_test;

                                // Write to analysis port (scoreboard and coverage checker)
                                analysis_port.write(transaction);

                                if(interface_instance.ref_instr === 32'bx)
                                        interface_instance.test_in_progress = 0;

                                if(interface_instance.end_of_test) begin
                                        `uvm_info("PASS","Test passed without errors",UVM_LOW)
                                        $display("------------------------------------------------------------------");
                                        $display("P A S S     P A S S     P A S S     P A S S     P A S S     P A S S");
                                        $display("------------------------------------------------------------------");
                                end

                                // Check for test log completion on continuous batch test run
                                if(transaction.mismatch_log_complete && run_all) begin
                                        interface_instance.test_in_progress = 0;
                                        interface_instance.mismatch_detected = 1;
```

```
                                $display("Bypassing Failed Test");
                            end
                        end
                    end
                endtask
endclass
`endif
```

APPENDIX AA: Source Code of Verification Environment – UVM Coverage

```
`include "uvm_pkg.sv"
`include "uvm_macros.svh"
`include "seq_item.sv"
`include "parameter_list.sv"
import    uvm_pkg::*;
`ifndef coverage
        `define coverage
class   coverage        extends uvm_subscriber #(seq_item);
        `uvm_component_utils(coverage)

        seq_item                        transaction;

        covergroup                      functional_cover;
                option.per_instance = 1;
                option.get_inst_coverage = 1;
                stall:          coverpoint      transaction.ref_stall           {
                                        option.weight = 0;
                                        bins    no_stall=       {0};
                                        bins    stalled =       {1};
                                }
                flush:          coverpoint      transaction.ref_flush           {
                                        option.weight = 0;
                                        bins    no_flush=       {0};
                                        bins    flushed =       {1};
                                }
                uncond_jump:    coverpoint      transaction.ref_ID_instr[`OPCODE_HI:`OPCODE_LO] {
                                        bins    jal     =       {`J_OPCODE};
                                        bins    jalr    =       {`JALR_OPCODE};
                                }
                cond_jumps:     coverpoint      {transaction.ref_ID_instr[`FUNCT3_HI:`FUNCT3_LO],
                                                 transaction.ref_ID_instr[`OPCODE_HI:`OPCODE_LO]} {
                                        bins    beq_    =       {`BEQ_CVR};
                                        bins    bne_    =       {`BNE_CVR};
                                        bins    blt_    =       {`BLT_CVR};
                                        bins    bge_    =       {`BGE_CVR};
                                        bins    bltu_   =       {`BLTU_CVR};
                                        bins    bgeu_   =       {`BGEU_CVR};
                                }
                loads:          coverpoint      {transaction.ref_EX_instr[`FUNCT3_HI:`FUNCT3_LO],
                                                 transaction.ref_EX_instr[`OPCODE_HI:`OPCODE_LO]} {
                                        bins    lb_     =       {`LB_CVR};
                                        bins    lh_     =       {`LH_CVR};
                                        bins    lw_     =       {`LW_CVR};
                                        bins    lbu_    =       {`LBU_CVR};
                                        bins    lhu_    =       {`LHU_CVR};
                                }
                instructions_A: coverpoint      {transaction.ref_EX_instr[`FUNCT3_HI:`FUNCT3_LO],
                                                 transaction.ref_EX_instr[`OPCODE_HI:`OPCODE_LO]} {
                                        bins    sb_     =       {`SB_CVR};
                                        bins    sh_     =       {`SH_CVR};
                                        bins    sw_     =       {`SW_CVR};
                                        bins    addi_   =       {`ADDI_CVR};
                                        bins    slli_   =       {`SLLI_CVR};
                                        bins    xori_   =       {`XORI_CVR};
                                        bins    ori_    =       {`ORI_CVR};
                                        bins    andi_   =       {`ANDI_CVR};
                                        bins    slti_   =       {`SLTI_CVR};
                                        bins    sltiu_  =       {`SLTIU_CVR};
                                }
                instructions_B: coverpoint      {transaction.ref_EX_instr[`FUNCT7_HI:`FUNCT7_LO],
                                                 transaction.ref_EX_instr[`FUNCT3_HI:`FUNCT3_LO],
                                                 transaction.ref_EX_instr[`OPCODE_HI:`OPCODE_LO]} {
                                        bins    srli_   =       {`SRLI_CVR};
                                        bins    srai_   =       {`SRAI_CVR};
                                        bins    add_    =       {`ADD_CVR};
                                        bins    sub     =       {`SUB_CVR};
                                        bins    sll_    =       {`SLL_CVR};
                                        bins    xor_    =       {`XOR_CVR};
                                        bins    srl_    =       {`SRL_CVR};
                                        bins    sra_    =       {`SRA_CVR};
                                        bins    or_     =       {`OR_CVR};
                                        bins    and_    =       {`AND_CVR};
                                        bins    slt_    =       {`SLT_CVR};
                                        bins    sltu_   =       {`SLTU_CVR};
                                }
                instruction_C:  coverpoint      {transaction.ref_EX_instr[`OPCODE_HI:`OPCODE_LO]}       {
                                        bins    lui     =       {`U_OPCODE};
                                }
                load_use_stalls:        cross   loads, stall;
                cond_jump_flushes:      cross   flush, cond_jumps;
                endgroup

        function new(string name="coverage", uvm_component parent=null);
```

```
                super.new(name, parent);
                functional_cover = new;
        endfunction

        virtual function void   write(seq_item t);
                transaction = t;
                functional_cover.sample();
                if(transaction.end_of_test)
                        $display("Functional Coverage Progress: %0.2f %%", functional_cover.get_inst_coverage());
        endfunction
endclass
`endif
```

APPENDIX BB: Source Code of Verification Environment – UVM Scoreboard

```
`include        "uvm_pkg.sv"
`include        "uvm_macros.svh"
`include        "seq_item.sv"
`include        "parameter_list.sv"
import          uvm_pkg::*;
`ifndef scoreboard
        `define scoreboard
class   scoreboard      extends uvm_scoreboard;
        `uvm_component_utils(scoreboard)

        // Static variables for self checking mechanism
        logic   [`DATA_WIDTH-1:0]       data_1_buffer;
        logic   [`DATA_WIDTH-1:0]       data_2_buffer;
        logic   [`DATA_WIDTH-1:0]       data_1;
        logic   [`DATA_WIDTH-1:0]       data_2;
        logic                           load_flag;
        logic                           load_use_flag;
        logic                           stall_assertion;
        logic                           stall_buffer;
        logic   [`DATA_WIDTH-1:0]       write_data_buffer;
        logic   [`DATA_WIDTH-1:0]       write_data_check;
        logic   [`REG_ADDR_WIDTH-1:0]   write_address_buffer;
        logic   [`REG_ADDR_WIDTH-1:0]   write_address_check;
        logic                           reg_write_check_buffer;
        logic                           reg_write_check;
        logic                           branch_check_flag;
        logic                           jump_check_flag;
        logic                           flush_buffer;
        logic                           flush_assertion;

        // Static variables for macro mechanism
        int                             stall_count;
        int                             flush_count;
        int                             stall_check;
        int                             flush_check;

        // Verification Status Flags
        logic                           mismatch;
        logic                           mismatch_found;
        logic                           end_of_test;

        // Database Parameter Variables
        bit                             testlog;
        bit                             memlog;
        bit                             directed_test;
        bit                             run_all;
        bit                             bypass_macro;
        bit                             macro_overwrite;
        bit                             force_gen;
        int                             test_seed;

        function new(string name="scoreboard", uvm_component parent=null);
                super.new(name, parent);
        endfunction

        seq_item                transaction;
        uvm_analysis_imp        #(seq_item, scoreboard) m_analysis_imp;

        virtual function void   build_phase(uvm_phase phase);
                super.build_phase(phase);
                m_analysis_imp = new("m_analysis_imp",this);
        endfunction

        // Perform functionality check
        virtual function void   write(seq_item transaction);
                int     fh;

                mismatch = 0;

                // Obtain config parameters from database
                if(!uvm_config_db#(bit)::get(null,"uvm_test_top","testlog",testlog))
                        testlog = 0;
                if(!uvm_config_db#(bit)::get(null,"uvm_test_top","memlog",memlog))
                        memlog = 0;
                if(!uvm_config_db#(bit)::get(null,"uvm_test_top","directed_test",directed_test))
                        directed_test = 0;
                if(!uvm_config_db#(bit)::get(null,"uvm_test_top","run_all",run_all))
                        run_all = 0;
                if(!uvm_config_db#(bit)::get(null,"uvm_test_top","bypass_macro",bypass_macro))
                        bypass_macro = 0;
                if(!uvm_config_db#(bit)::get(null,"uvm_test_top","macro_overwrite",macro_overwrite))
                        macro_overwrite = 0;
                if(!uvm_config_db#(bit)::get(null,"uvm_test_top","force_gen",force_gen))
                        force_gen = 0;
                if(!uvm_config_db#(int)::get(null,"uvm_test_top","test_seed",test_seed))
                        test_seed = 0;

                self_check(transaction);        // Self check mechanism
                decode(transaction);            // Decode and store information
                mismatch_check(transaction);    // Check for mismatch

                if(mismatch || (testlog && transaction.end_of_test)) begin
                        test_logging(transaction);      // Produce Test Log
                        if(!mismatch && !bypass_macro)
                                macro_check();          // Perform Macro Check
                        // Move file to directory
                        if(directed_test) begin
                                move_file("ASM.txt","test_results/directed_test");
                                move_file("LOG.txt","test_results/directed_test");
```

```
                    end
                    else begin
                            move_file("ASM.txt",{"test_results/", test_seed});
                            move_file("LOG.txt",{"test_results/", test_seed});
                    end
                    clear_current_directory();
                    clear_static_variable();
                    if(mismatch) begin
                            $display("Test Failed");
                            $display("XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX");
                            $display("F A I L     F A I L     F A I L     F A I L     F A I L     F A I L");
                            $display("XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX");
                            transaction.mismatch_log_complete = 1;
                            if(!run_all)
                                    `uvm_fatal("MISMATCH","Successful Logging of Information")
                            else begin
                                    fh = $fopen("FAILED.txt","a+");
                                    $fdisplay(fh, "%s - Register Mismatch", test_seed);
                                    $fclose(fh);
                            end
                    end
                    else
                            `uvm_info("LOGGING","Successful Logging of Information", UVM_LOW)
                    mismatch = 0;
            end
    endfunction

    // Simple function for obtaining current system time
    function string get_time();
            int file_pointer;
            int code;
            string date_s_1;
            string date_s_2;
            string time_s;

            $system("date /t >> sys_time");
            $system("time /t >> sys_time");
            file_pointer = $fopen("sys_time","r");
            code = $fscanf(file_pointer, "%s %s %s", date_s_1, date_s_2, time_s);
            get_time = {date_s_1, " ", date_s_2, " ", time_s};
            $fclose(file_pointer);
            $system("del sys_time");
    endfunction

    // Self checking for reference model correctness
    // In self checking, several characteristics of the reference model will be checked
    // 1. Correct Register Read
    // 2. Correct Instruction Operation ALU Output
    // 3. Successful Detection of Load-Use Case and Stalling
    // 4. Correct Info (Specific Instruction Type) and Register Write
    // 5. Successful Detection and Execution of Branch/Jump
    function void self_check(seq_item transaction);
            logic   [`DATA_WIDTH-1:0]       behaviour_result;

            if(transaction.ref_ID_instr[`RS1_ADDR_HI:`RS1_ADDR_LO] != transaction.ref_reg_read_addr_1) begin
                    `uvm_fatal("REF MODEL ERROR",$sformatf
                    ("Incorrect 1st register read: Behaviour: x%2d     Model: x%2d",
                    transaction.ref_ID_instr[`RS1_ADDR_HI:`RS1_ADDR_LO], transaction.ref_reg_read_addr_1))
            end
            if(transaction.ref_ID_instr[`RS2_ADDR_HI:`RS2_ADDR_LO] != transaction.ref_reg_read_addr_2) begin
                    `uvm_fatal("REF MODEL ERROR",$sformatf
                    ("Incorrect 2nd register read: Behaviour: x%2d     Model: x%2d",
                    transaction.ref_ID_instr[`RS2_ADDR_HI:`RS2_ADDR_LO], transaction.ref_reg_read_addr_2))
            end

            // Pass ID Stage Operands to EX Stage for Execution
            data_1 = data_1_buffer;
            data_2 = data_2_buffer;
            data_1_buffer = transaction.ref_reg_read_data_1;
            case(transaction.ref_ID_instr[`OPCODE_HI:`OPCODE_LO])
            `R_OPCODE,
            `SB_OPCODE      : data_2_buffer = transaction.ref_reg_read_data_2;
            default         : data_2_buffer = transaction.ref_imm_val;
            endcase

            // Check EX Stage Execution Correctness
            case(transaction.ref_EX_instr[`OPCODE_HI:`OPCODE_LO])
            `R_OPCODE: begin
                    case(transaction.ref_EX_instr[`FUNCT3_HI:`FUNCT3_LO])
                    `ADD_FUNCT3: begin
                            case(transaction.ref_EX_instr[`FUNCT7_HI:`FUNCT7_LO])
                            `DEFAULT_FUNCT7: begin
                                    behaviour_result = data_1 + data_2;
                                    if(transaction.ref_alu_output != behaviour_result)
                                    `uvm_fatal("REF MODEL ERROR",$sformatf
                                    ("Incorrect ADD Result: Behaviour: %8h     Model: %8h",
                                    behaviour_result, transaction.ref_alu_output))
                                            end
                            `ALT_FUNCT7: begin
                                    behaviour_result = data_1 - data_2;
                                    if(transaction.ref_alu_output != behaviour_result)
                                    `uvm_fatal("REF MODEL ERROR",$sformatf
                                    ("Incorrect SUB Result: Behaviour: %8h     Model: %8h",
                                    behaviour_result, transaction.ref_alu_output))
                                        end
                            endcase
                                end
                    `SLL_FUNCT3: begin
                            behaviour_result = data_1 << data_2[4:0];
                            if(transaction.ref_alu_output != behaviour_result)
                            `uvm_fatal("REF MODEL ERROR",$sformatf
                            ("Incorrect SLL Result: Behaviour: %8h     Model: %8h",
                            behaviour_result, transaction.ref_alu_output))
                                    end
                    `SLT_FUNCT3: begin
                            behaviour_result = (($signed(data_1) >= $signed(data_2)) ? 0 : 1);
                            if(transaction.ref_alu_output != behaviour_result)
                            `uvm_fatal("REF MODEL ERROR",$sformatf
                            ("Incorrect SLT Result: Behaviour: %8h     Model: %8h",
                            behaviour_result, transaction.ref_alu_output))
                                    end
                    `SLTU_FUNCT3: begin
                            behaviour_result = ((data_1 >= data_2) ? 0 : 1);
```

```
                        if(transaction.ref_alu_output != behaviour_result)
                        `uvm_fatal("REF MODEL ERROR",$sformatf
                        ("Incorrect SLTU Result: Behaviour: %8h     Model: %8h",
                        behaviour_result, transaction.ref_alu_output))
                            end
                `XOR_FUNCT3: begin
                        behaviour_result = data_1 ^ data_2;
                        if(transaction.ref_alu_output != behaviour_result)
                        `uvm_fatal("REF MODEL ERROR",$sformatf
                        ("Incorrect XOR Result: Behaviour: %8h     Model: %8h",
                        behaviour_result, transaction.ref_alu_output))
                            end
                `SR_FUNCT3: begin
                        case(transaction.ref_EX_instr[`FUNCT7_HI:`FUNCT7_LO])
                        `DEFAULT_FUNCT7: begin
                                behaviour_result = data_1 >> data_2[4:0];
                                if(transaction.ref_alu_output != behaviour_result)
                                `uvm_fatal("REF MODEL ERROR",$sformatf
                                ("Incorrect SRL Result: Behaviour: %8h     Model: %8h",
                                behaviour_result, transaction.ref_alu_output))
                                    end
                        `ALT_FUNCT7: begin
                                behaviour_result = $signed(data_1) >>> data_2[4:0];
                                if(transaction.ref_alu_output != behaviour_result)
                                `uvm_fatal("REF MODEL ERROR",$sformatf
                                ("Incorrect SRA Result: Behaviour: %8h     Model: %8h",
                                behaviour_result, transaction.ref_alu_output))
                                    end
                        endcase

                            end
                `OR_FUNCT3: begin
                        behaviour_result = data_1 | data_2;
                        if(transaction.ref_alu_output != behaviour_result)
                        `uvm_fatal("REF MODEL ERROR",$sformatf
                        ("Incorrect OR Result: Behaviour: %8h     Model: %8h",
                        behaviour_result, transaction.ref_alu_output))
                            end
                `AND_FUNCT3: begin
                        behaviour_result = data_1 & data_2;
                        if(transaction.ref_alu_output != behaviour_result)
                        `uvm_fatal("REF MODEL ERROR",$sformatf
                        ("Incorrect AND Result: Behaviour: %8h     Model: %8h",
                        behaviour_result, transaction.ref_alu_output))
                            end
                endcase
                    end
            `S_OPCODE,
            `LOAD_OPCODE: begin
                behaviour_result = data_1 + data_2;
                if(transaction.ref_alu_output != behaviour_result)
                `uvm_fatal("REF MODEL ERROR",$sformatf
                ("Incorrect Load/Store Address Calculated: Behaviour: %8h     Model: %8h",
                behaviour_result, transaction.ref_alu_output))
                    end
            `U_OPCODE: begin
                behaviour result = data 2;
                if(transaction.ref_alu_output != behaviour_result)
                `uvm_fatal("REF MODEL ERROR",$sformatf
                ("Incorrect Load Upper Immediate Result: Behaviour: %8h     Model: %8h",
                behaviour_result, transaction.ref_alu_output))
                    end
            `I_OPCODE: begin
                case(transaction.ref_EX_instr[`FUNCT3_HI:`FUNCT3_LO])
                `ADD_FUNCT3: begin
                        behaviour_result = data_1 + data_2;
                        if(transaction.ref_alu_output != behaviour_result)
                        `uvm_fatal("REF MODEL ERROR",$sformatf
                        ("Incorrect ADDI Result: Behaviour: %8h     Model: %8h",
                        behaviour_result, transaction.ref_alu_output))
                            end
                `SLL_FUNCT3: begin
                        behaviour_result = data_1 << data_2[4:0];
                        if(transaction.ref_alu_output != behaviour_result)
                        `uvm_fatal("REF MODEL ERROR",$sformatf
                        ("Incorrect SLLI Result: Behaviour: %8h     Model: %8h",
                        behaviour_result, transaction.ref_alu_output))
                            end
                `SLT_FUNCT3: begin
                        behaviour_result = (($signed(data_1) >= $signed(data_2)) ? 0 : 1);
                        if(transaction.ref_alu_output != behaviour_result)
                        `uvm_fatal("REF MODEL ERROR",$sformatf
                        ("Incorrect SLTI Result: Behaviour: %8h     Model: %8h",
                        behaviour_result, transaction.ref_alu_output))
                            end
                `SLTU_FUNCT3: begin
                        behaviour_result = ((data_1 >= data_2) ? 0 : 1);
                        if(transaction.ref_alu_output != behaviour_result)
                        `uvm_fatal("REF MODEL ERROR",$sformatf
                        ("Incorrect SLTIU Result: Behaviour: %8h     Model: %8h",
                        behaviour_result, transaction.ref_alu_output))
                            end
                `XOR_FUNCT3: begin
                        behaviour_result = data_1 ^ data_2;
                        if(transaction.ref_alu_output != behaviour_result)
                        `uvm_fatal("REF MODEL ERROR",$sformatf
                        ("Incorrect XORI Result: Behaviour: %8h     Model: %8h",
                        behaviour_result, transaction.ref_alu_output))
                            end
                `SR_FUNCT3: begin
                        case(transaction.ref_EX_instr[`FUNCT7_HI:`FUNCT7_LO])
                        `DEFAULT_FUNCT7: begin
                                behaviour_result = data_1 >> data_2[4:0];
                                if(transaction.ref_alu_output != behaviour_result)
                                `uvm_fatal("REF MODEL ERROR",$sformatf
                                ("Incorrect SRLI Result: Behaviour: %8h     Model: %8h",
                                behaviour_result, transaction.ref_alu_output))
                                    end
                        `ALT_FUNCT7: begin
                                behaviour_result = $signed(data_1) >>> data_2[4:0];
```

```
                        if(transaction.ref_alu_output != behaviour_result)
                        `uvm_fatal("REF MODEL ERROR",$sformatf
                        ("Incorrect SRLI Result: Behaviour: %8h     Model: %8h",
                        behaviour_result, transaction.ref_alu_output))
                            end
                endcase
                    end
        `OR_FUNCT3: begin
                behaviour_result = data_1 | data_2;
                if(transaction.ref_alu_output != behaviour_result)
                `uvm_fatal("REF MODEL ERROR",$sformatf
                ("Incorrect ORI Result: Behaviour: %8h     Model: %8h",
                behaviour_result, transaction.ref_alu_output))
                  end
        `AND_FUNCT3: begin
                behaviour_result = data_1 & data_2;
                if(transaction.ref_alu_output != behaviour_result)
                `uvm_fatal("REF MODEL ERROR",$sformatf
                ("Incorrect ANDI Result: Behaviour: %8h     Model: %8h",
                behaviour_result, transaction.ref_alu_output))
                    end
        endcase
           end
`SB_OPCODE: begin
        case(transaction.ref_EX_instr[`FUNCT3_HI:`FUNCT3_LO])
        `BEQ_FUNCT3: begin
                behaviour_result = !(data_1 == data_2);
                if(transaction.ref_alu_output != behaviour_result)
                `uvm_fatal("REF MODEL ERROR",$sformatf
                ("Incorrect BEQ Result: Behaviour: %8h     Model: %8h",
                behaviour_result, transaction.ref_alu_output))
                    end
        `BNE_FUNCT3: begin
                behaviour_result = (data_1 == data_2);
                if(transaction.ref_alu_output != behaviour_result)
                `uvm_fatal("REF MODEL ERROR",$sformatf
                ("Incorrect BNE Result: Behaviour: %8h     Model: %8h",
                behaviour_result, transaction.ref_alu_output))
                    end
        `BLT_FUNCT3: begin
                behaviour_result = (($signed(data_1) < $signed(data_2)) ? 0 : 1);
                if(transaction.ref_alu_output != behaviour_result)
                `uvm_fatal("REF MODEL ERROR",$sformatf
                ("Incorrect BLT Result: Behaviour: %8h     Model: %8h",
                behaviour_result, transaction.ref_alu_output))
                    end
        `BGE_FUNCT3: begin
                behaviour_result = (($signed(data_1) >= $signed(data_2)) ? 0 : 1);
                if(transaction.ref_alu_output != behaviour_result)
                `uvm_fatal("REF MODEL ERROR",$sformatf
                ("Incorrect BGE Result: Behaviour: %8h     Model: %8h",
                behaviour_result, transaction.ref_alu_output))
                  end
        `BLTU_FUNCT3: begin
                behaviour_result = ((data_1 < data_2) ? 0 : 1);
                if(transaction.ref_alu_output != behaviour_result)
                `uvm_fatal("REF MODEL ERROR",$sformatf
                ("Incorrect BLTU Result: Behaviour: %8h     Model: %8h",
                behaviour_result, transaction.ref_alu_output))
                    end
        `BGEU_FUNCT3: begin
                behaviour_result = ((data_1 >= data_2) ? 0 : 1);
                if(transaction.ref_alu_output != behaviour_result)
                `uvm_fatal("REF MODEL ERROR",$sformatf
                ("Incorrect BGEU Result: Behaviour: %8h     Model: %8h",
                behaviour_result, transaction.ref_alu_output))
                    end
          endcase
              end
endcase

// Check for Load-use case and Stall assertion
if(load_flag &&
   (transaction.ref_ID_instr[`RS2_ADDR_HI:`RS2_ADDR_LO] == transaction.ref_EX_instr[`RD_ADDR_HI:`RD_ADDR_LO] ||
    transaction.ref_ID_instr[`RS1_ADDR_HI:`RS1_ADDR_LO] == transaction.ref_EX_instr[`RD_ADDR_HI:`RD_ADDR_LO]) &&
   transaction.ref_EX_instr[`RD_ADDR_HI:`RD_ADDR_LO] != 0)
        load_use_flag = 1;
else
        load_use_flag = 0;
// ID Stage Load Detection
if(transaction.ref_ID_instr[`OPCODE_HI:`OPCODE_LO] == `LOAD_OPCODE)
        load_flag = 1;
else
        load_flag = 0;
// Check for Stall Combinational Output upon EX Stage Load ID Stage Use
if(load_use_flag) begin
        if(!transaction.ref_stall)
                `uvm_fatal("REF MODEL ERROR","Load-use Case Not Stalled")
        else
                load_use_flag = 0;
end

// EXMEM Stage to MEMWB Stage Pipeline
reg_write_check = reg_write_check_buffer;
// Check instruction type at EX Stage
case(transaction.ref_EX_instr[`OPCODE_HI:`OPCODE_LO])
`R_OPCODE,
`J_OPCODE,
`JALR_OPCODE,
`U_OPCODE,
`I_OPCODE: begin
                reg_write_check_buffer = 1;
            end
default: reg_write_check_buffer = 0;
endcase

// EXMEM Stage to MEMWB Stage Pipeline
write_address_check = write_address_buffer;
write_data_check = write_data_buffer;
write_address_buffer = transaction.ref_EX_instr[`RD_ADDR_HI:`RD_ADDR_LO];
// For JAL and JALR
if(transaction.ref_EX_instr[`OPCODE_HI:`OPCODE_LO] == `J_OPCODE ||
```

```
                transaction.ref_EX_instr[`OPCODE_HI:`OPCODE_LO] == `JALR_OPCODE)
                write_data_buffer = transaction.ref_EX_pc + `INST_ADDR_SUM;
        else
                write_data_buffer = transaction.ref_alu_output;

        if(reg_write_check) begin
                //Check Address
                if(write_address_check != transaction.ref_reg_write_addr)
                        `uvm_fatal("REF MODEL ERROR",$sformatf
                        ("Incorrect write register address: Behaviour: x%2d     Model: x%2d",
                        write_address_check, transaction.ref_reg_write_addr))
                //Check Data
                if(write_data_check != transaction.ref_reg_write_data)
                        `uvm_fatal("REF MODEL ERROR",$sformatf
                        ("Incorrect write register data: Behaviour: %8h    Model: %8h",
                        write_data_check, transaction.ref_reg_write_data))
        end

        // Check for Branch or Jump Execution
        flush_assertion = flush_buffer;
        flush_buffer = transaction.ref_flush;
        case(transaction.ref_EX_instr[`OPCODE_HI:`OPCODE_LO])
        `SB_OPCODE: branch_check_flag = 1;
        `J_OPCODE,
        `JALR_OPCODE: jump_check_flag = 1;
        endcase
        if((branch_check_flag && transaction.ref_alu_zero) || jump_check_flag) begin
                if(!flush_assertion)
                        `uvm_fatal("REF MODEL ERROR","Branch or Jump not Executed")
        end
        branch_check_flag = 0;
        jump_check_flag = 0;
endfunction

// Decode and validate instruction on pipeline
// Store instruction information for logging
function void decode(seq_item transaction);
        int                             fl;
        logic   [`OPCODE_WIDTH-1:0]     opcode;
        logic   [`REG_ADDR_WIDTH-1:0]   rs1;
        logic   [`REG_ADDR_WIDTH-1:0]   rs2;
        logic   [`REG_ADDR_WIDTH-1:0]   rd;
        logic   [`FUNCT3_WIDTH-1:0]     funct3;
        logic   [`FUNCT7_WIDTH-1:0]     funct7;
        logic   [`I_IMM_WIDTH-1:0]      imm;
        logic   [`SB_IMM_WIDTH-1:0]     branch;
        logic   [`J_IMM_WIDTH-1:0]      jal_offset;
        logic   [`U_IMM_WIDTH-1:0]      lui_constant;

        opcode = transaction.ref_instr[`OPCODE_HI:`OPCODE_LO];
        rs1 = transaction.ref_instr[`RS1_ADDR_HI:`RS1_ADDR_LO];
        rs2 = transaction.ref_instr[`RS2_ADDR_HI:`RS2_ADDR_LO];
        rd = transaction.ref_instr[`RD_ADDR_HI:`RD_ADDR_LO];
        funct3 = transaction.ref_instr[`FUNCT3_HI:`FUNCT3_LO];
        funct7 = transaction.ref_instr[`FUNCT7_HI:`FUNCT7_LO];
        imm = transaction.ref_instr[31:20];
        branch = ({transaction.ref_instr[31],transaction.ref_instr[7], transaction.ref_instr[30:25],
                  transaction.ref_instr[11:8], 1'b0});
        jal_offset = ({transaction.ref_instr[31],transaction.ref_instr[19:12], transaction.ref_instr[20],
                     transaction.ref_instr[30:21], 1'b0});
        lui_constant = transaction.ref_instr[31:12];


        fl = $fopen("INSTR.txt", "a+");
        case (opcode)
        `R_OPCODE: begin
                case(funct3)
                `ADD_FUNCT3: begin
                        case (funct7)
                        `DEFAULT_FUNCT7: begin
                                `uvm_info("SCBD",$sformatf("add     x%2d,   x%2d,   x%2d",
                                rd, rs1, rs2), UVM_MEDIUM)
                                $fdisplay(fl, "add     x%2d,   x%2d,   x%2d", rd, rs1, rs2);
                                    end
                        `ALT_FUNCT7: begin
                                `uvm_info("SCBD",$sformatf("sub     x%2d,   x%2d,   x%2d",
                                rd, rs1, rs2), UVM_MEDIUM)
                                $fdisplay(fl, "sub     x%2d,   x%2d,   x%2d", rd, rs1, rs2);
                                    end
                        default: `uvm_fatal("ERROR",$sformatf
                                 ("Unknown funct7 field Failing Field: %7b      Failing Instruction: %8h",
                                 funct7, transaction.ref_instr))
                        endcase
                             end
                `SLL_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("sll     x%2d,   x%2d,   x%2d",
                         rd, rs1, rs2), UVM_MEDIUM)
                        $fdisplay(fl, "sll     x%2d,   x%2d,   x%2d", rd, rs1, rs2);
                             end
                `SLT_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("slt     x%2d,   x%2d,   x%2d",
                         rd, rs1, rs2), UVM_MEDIUM)
                        $fdisplay(fl, "slt     x%2d,   x%2d,   x%2d", rd, rs1, rs2);
                             end
                `SLTU_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("sltu     x%2d,   x%2d,   x%2d",
                         rd, rs1, rs2), UVM_MEDIUM)
                        $fdisplay(fl, "sltu     x%2d,   x%2d,   x%2d", rd, rs1, rs2);
                              end
                `XOR_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("xor     x%2d,   x%2d,   x%2d",
                         rd, rs1, rs2), UVM_MEDIUM)
                        $fdisplay(fl, "xor     x%2d,   x%2d,   x%2d", rd, rs1, rs2);
                             end
                `SR_FUNCT3: begin
                        case (funct7)
                        `DEFAULT_FUNCT7: begin
                                `uvm_info("SCBD",$sformatf("srl     x%2d,   x%2d,   x%2d",
                                rd, rs1, rs2), UVM_MEDIUM)
                                $fdisplay(fl, "srl     x%2d,   x%2d,   x%2d", rd, rs1, rs2);
                                    end
                        `ALT_FUNCT7: begin
```

```
                        `uvm_info("SCBD",$sformatf("sra     x%2d,   x%2d,   x%2d",
                         rd, rs1, rs2), UVM_MEDIUM)
                        $fdisplay(fl, "sra     x%2d,   x%2d,   x%2d", rd, rs1, rs2);
                            end
                    default: `uvm_fatal("ERROR",$sformatf
                             ("Unknown funct7 field Failing Field: %7b      Failing Instruction: %8h",
                             funct7, transaction.ref_instr))
                    endcase
                        end
                `OR_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("or      x%2d,   x%2d,   x%2d",
                         rd, rs1, rs2), UVM_MEDIUM)
                        $fdisplay(fl, "or      x%2d,   x%2d,   x%2d", rd, rs1, rs2);
                        end
                `AND_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("and     x%2d,   x%2d,   x%2d",
                         rd, rs1, rs2), UVM_MEDIUM)
                        $fdisplay(fl, "and     x%2d,   x%2d,   x%2d", rd, rs1, rs2);
                        end
                default: `uvm_fatal("ERROR",$sformatf
                         ("Unknown funct3 field Failing Field: %3b      Failing Instruction: %8h",
                         funct3, transaction.ref_instr))
                endcase
                  end
            `LOAD_OPCODE: begin
                case (funct3)
                `LB_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("lb      x%2d,   0x%3h(x%2d)",
                         rd, imm, rs1), UVM_MEDIUM)
                        $fdisplay(fl, "lb      x%2d,   0x%3h(x%2d)", rd, imm, rs1);
                        end
                `LH_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("lh      x%2d,   0x%3h(x%2d)",
                         rd, imm, rs1), UVM_MEDIUM)
                        $fdisplay(fl, "lh      x%2d,   0x%3h(x%2d)", rd, imm, rs1);
                        end
                `LW_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("lw      x%2d,   0x%3h(x%2d)",
                         rd, imm, rs1), UVM_MEDIUM)
                        $fdisplay(fl, "lw      x%2d,   0x%3h(x%2d)", rd, imm, rs1);
                        end
                `LBU_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("lbu     x%2d,   0x%3h(x%2d)",
                         rd, imm, rs1), UVM_MEDIUM)
                        $fdisplay(fl, "lbu     x%2d,   0x%3h(x%2d)", rd, imm, rs1);
                         end
                `LHU_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("lhu     x%2d,   0x%3h(x%2d)",
                         rd, imm, rs1), UVM_MEDIUM)
                        $fdisplay(fl, "lhu     x%2d,   0x%3h(x%2d)", rd, imm, rs1);
                         end
                default: `uvm_fatal("ERROR",$sformatf
                         ("Unknown funct3 field Failing Field: %3b      Failing Instruction: %8h",
                         funct3, transaction.ref_instr))
                endcase
                   end
            `I_OPCODE: begin
                case (funct3)
                `ADD_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("addi    x%2d,   x%2d,   0x%3h",
                         rd, rs1, imm), UVM_MEDIUM)
                        $fdisplay(fl, "addi    x%2d,   x%2d,   0x%3h", rd, rs1, imm);
                         end
                `SLL_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("slli    x%2d,   x%2d,   %d",
                         rd, rs1, imm[4:0]), UVM_MEDIUM)
                        $fdisplay(fl, "slli    x%2d,   x%2d,   %d", rd, rs1, imm[4:0]);
                         end
                `SLT_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("slti    x%2d,   x%2d,   0x%3h",
                         rd, rs1, imm), UVM_MEDIUM)
                        $fdisplay(fl, "slti    x%2d,   x%2d,   0x%3h", rd, rs1, imm);
                         end
                `SLTU_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("sltiu    x%2d,   x%2d,   0x%3h",
                         rd, rs1, imm), UVM_MEDIUM)
                        $fdisplay(fl, "sltiu    x%2d,   x%2d,   0x%3h", rd, rs1, imm);
                          end
                `XOR_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("xori    x%2d,   x%2d,   0x%3h",
                         rd, rs1, imm), UVM_MEDIUM)
                        $fdisplay(fl, "xori    x%2d,   x%2d,   0x%3h", rd, rs1, imm);
                         end
                `SR_FUNCT3: begin
                        case (funct7)
                        `DEFAULT_FUNCT7: begin
                                `uvm_info("SCBD",$sformatf("srli    x%2d,   x%2d,   %d",
                                 rd, rs1, imm[4:0]), UVM_MEDIUM)
                                $fdisplay(fl, "srli    x%2d,   x%2d,   %d", rd, rs1, imm[4:0]);
                                   end
                        `ALT_FUNCT7: begin
                                `uvm_info("SCBD",$sformatf("srai    x%2d,   x%2d,   %d",
                                 rd, rs1, imm[4:0]), UVM_MEDIUM)
                                $fdisplay(fl, "srai    x%2d,   x%2d,   %d", rd, rs1, imm[4:0]);
                                   end
                        default: `uvm_fatal("ERROR",$sformatf
                                 ("Unknown funct7 field Failing Field: %7b      Failing Instruction: %8h",
                                 funct7, transaction.ref_instr))
                        endcase
                            end
                `OR_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("ori     x%2d,   x%2d,   0x%3h",
                         rd, rs1, imm), UVM_MEDIUM)
                        $fdisplay(fl, "ori     x%2d,   x%2d,   0x%3h", rd, rs1, imm);
                            end
                `AND_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("andi    x%2d,   x%2d,   0x%3h",
                         rd, rs1, imm), UVM_MEDIUM)
                        $fdisplay(fl, "andi    x%2d,   x%2d,   0x%3h", rd, rs1, imm);
                         end
                default: `uvm_fatal("ERROR",$sformatf
```

```
                        ("Unknown funct3 field Failing Field: %3b      Failing Instruction: %8h",
                        funct3, transaction.ref_instr))
                endcase
                   end
        `S_OPCODE: begin
                case (funct3)
                `SB_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("sb      x%2d,   0x%3h(x%2d)",
                        rs2, {funct7,rd}, rs1), UVM_MEDIUM)
                       $fdisplay(fl, "sb      x%2d,   0x%3h(x%2d)", rs2, {funct7,rd}, rs1);
                           end
                `SH_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("sh      x%2d,   0x%3h(x%2d)",
                        rs2, {funct7,rd}, rs1), UVM_MEDIUM)
                       $fdisplay(fl, "sh      x%2d,   0x%3h(x%2d)", rs2, {funct7,rd}, rs1);
                           end
                `SW_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("sw      x%2d,   0x%3h(x%2d)",
                        rs2, {funct7,rd}, rs1), UVM_MEDIUM)
                       $fdisplay(fl, "sw      x%2d,   0x%3h(x%2d)", rs2, {funct7,rd}, rs1);
                           end
                default: `uvm_fatal("ERROR",$sformatf
                        ("Unknown funct3 field Failing Field: %3b      Failing Instruction: %8h",
                        funct3, transaction.ref_instr))
                endcase
                   end
        `SB_OPCODE: begin
                case (funct3)
                `BEQ_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("beq     x%2d,   x%2d,   0x%3h",
                        rs1, rs2, branch), UVM_MEDIUM)
                       $fdisplay(fl, "beq     x%2d,   x%2d,   0x%3h", rs1, rs2, branch);
                           end
                `BNE_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("bne     x%2d,   x%2d,   0x%3h",
                        rs1, rs2, branch), UVM_MEDIUM)
                       $fdisplay(fl, "bne     x%2d,   x%2d,   0x%3h", rs1, rs2, branch);
                           end
                `BLT_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("blt     x%2d,   x%2d,   0x%3h",
                        rs1, rs2, branch), UVM_MEDIUM)
                       $fdisplay(fl, "blt     x%2d,   x%2d,   0x%3h", rs1, rs2, branch);
                           end
                `BGE_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("bge     x%2d,   x%2d,   0x%3h",
                        rs1, rs2, branch), UVM_MEDIUM)
                       $fdisplay(fl, "bge     x%2d,   x%2d,   0x%3h", rs1, rs2, branch);
                           end
                `BLTU_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("bltu    x%2d,   x%2d,   0x%3h",
                        rs1, rs2, branch), UVM_MEDIUM)
                       $fdisplay(fl, "bltu    x%2d,   x%2d,   0x%3h", rs1, rs2, branch);
                           end
                `BGEU_FUNCT3: begin
                        `uvm_info("SCBD",$sformatf("bgeu    x%2d,   x%2d,   0x%3h",
                        rs1, rs2, branch), UVM_MEDIUM)
                       $fdisplay(fl, "bgeu    x%2d,   x%2d,   0x%3h", rs1, rs2, branch);
                           end
                default: `uvm_fatal("ERROR",$sformatf
                        ("Unknown functt3 field Failing Field: %3b     Failing Instruction: %8h",
                        funct3, transaction.ref_instr))
                endcase
                    end
        `U_OPCODE: begin
                `uvm_info("SCBD",$sformatf("lui     x%2d,   0x%5h",
                rd, lui_constant), UVM_MEDIUM)
               $fdisplay(fl, "lui     x%2d,   0x%5h", rd, lui_constant);
                   end
        `J_OPCODE: begin
                `uvm_info("SCBD",$sformatf("jal     x%2d,   0x%5h",
                rd, jal_offset), UVM_MEDIUM)
               $fdisplay(fl, "jal     x%2d,   0x%5h", rd, jal_offset);
                   end
        `JALR_OPCODE: begin
                `uvm_info("SCBD",$sformatf("jalr    x%2d,   0x%3h(x%2d)",
                rd, imm, rs1), UVM_MEDIUM)
               $fdisplay(fl, "jalr    x%2d,   0x%3h(x%2d)", rd, imm, rs1);
                    end
        `NOP_OPCODE: begin
                `uvm_info("SCBD",$sformatf("nop"), UVM_MEDIUM)
               $fdisplay(fl, "nop");
                    end
        endcase
        $fclose(fl);
endfunction

// Compare Reference Model Results and DUT Model Results
// Store corresponding information for logging
function void mismatch_check(seq_item transaction);
        int fh;
        fh = $fopen("PIPELINE.txt", "a+");
        $fdisplay(fh, "%1b", transaction.ref_stall);
        $fclose(fh);
        fh = $fopen("FLUSH.txt","a+");
        $fdisplay(fh, "%1b", transaction.ref_flush);
        $fclose(fh);
        fh = $fopen("JUMP.txt","a+");
        $fdisplay(fh, "%1b", transaction.ref_jump_link);
        $fclose(fh);


        fh = $fopen("IFID.txt", "a+");
        $fdisplay(fh, "%8h %8h", transaction.ref_pc, transaction.dut_pc);
        $fdisplay(fh, "%8h %8h", transaction.ref_instr, transaction.dut_instr);
        $fclose(fh);

        if(transaction.ref_pc != transaction.dut_pc) begin
                mismatch = 1;
                `uvm_info("MISMATCH","Mismatch Encountered at IF/ID Pipeline Register",UVM_LOW);
                `uvm_info("MISMATCH","Mismatching Program Counter",UVM_LOW);
                `uvm_info("MISMATCH",$sformatf("REF PC: %8h", transaction.ref_pc),UVM_LOW);
```

```
        `uvm_info("MISMATCH",$sformatf("DUT PC: %8h", transaction.dut_pc),UVM_LOW);
    end
    if(transaction.ref_instr != transaction.dut_instr) begin
        mismatch = 1;
        `uvm_info("MISMATCH","Mismatch Encountered at IF/ID Pipeline Register",UVM_LOW);
        `uvm_info("MISMATCH","Mismatching Instruction Code",UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("REF Instr Code: %8h", transaction.ref_instr),UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("DUT Instr Code: %8h", transaction.dut_instr),UVM_LOW);
    end

    fh = $fopen("IDEX.txt", "a+");
    $fdisplay(fh, "%d %8h", transaction.ref_reg_read_addr_1, transaction.ref_reg_read_data_1);
    $fdisplay(fh, "%d %8h", transaction.ref_reg_read_addr_2, transaction.ref_reg_read_data_2);
    $fdisplay(fh, "%d %8h", transaction.dut_reg_read_addr_1, transaction.dut_reg_read_data_1);
    $fdisplay(fh, "%d %8h", transaction.dut_reg_read_addr_2, transaction.dut_reg_read_data_2);
    $fdisplay(fh, "%8h %8h", transaction.ref_imm_val, transaction.dut_imm_val);
    $fclose(fh);

    if(transaction.ref_reg_read_addr_1 != transaction.dut_reg_read_addr_1)  begin
        mismatch = 1;
        `uvm_info("MISMATCH","Mismatch Encountered at ID/EX Pipeline Register",UVM_LOW);
        `uvm_info("MISMATCH","Mismatching Register 01 Address",UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("REF Reg01 Addr: %d", transaction.ref_reg_read_addr_1),UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("DUT Reg01 Addr: %d", transaction.dut_reg_read_addr_1),UVM_LOW);
    end
    if(transaction.ref_reg_read_addr_2 != transaction.dut_reg_read_addr_2)  begin
        mismatch = 1;
        `uvm_info("MISMATCH","Mismatch Encountered at ID/EX Pipeline Register",UVM_LOW);
        `uvm_info("MISMATCH","Mismatching Register 02 Address",UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("REF Reg02 Addr: %d", transaction.ref_reg_read_addr_2),UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("DUT Reg02 Addr: %d", transaction.dut_reg_read_addr_2),UVM_LOW);
    end
    if(transaction.ref_reg_read_data_1 != transaction.dut_reg_read_data_1)  begin
        mismatch = 1;
        `uvm_info("MISMATCH","Mismatch Encountered at ID/EX Pipeline Register",UVM_LOW);
        `uvm_info("MISMATCH","Mismatching Register 01 Data",UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("REF Reg01 Data: %8h", transaction.ref_reg_read_data_1),UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("DUT Reg01 Data: %8h", transaction.dut_reg_read_data_1),UVM_LOW);
    end
    if(transaction.ref_reg_read_data_2 != transaction.dut_reg_read_data_2)  begin
        mismatch = 1;
        `uvm_info("MISMATCH","Mismatch Encountered at ID/EX Pipeline Register",UVM_LOW);
        `uvm_info("MISMATCH","Mismatching Register 02 Data",UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("REF Reg02 Data: %8h", transaction.ref_reg_read_data_2),UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("DUT Reg02 Data: %8h", transaction.dut_reg_read_data_2),UVM_LOW);
    end
    if(transaction.ref_imm_val != transaction.dut_imm_val) begin
        mismatch = 1;
        `uvm_info("MISMATCH","Mismatch Encountered at ID/EX Pipeline Register",UVM_LOW);
        `uvm_info("MISMATCH","Mismatching Immediate Value",UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("REF Imm Val: %8h", transaction.ref_imm_val),UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("DUT Imm Val: %8h", transaction.dut_imm_val),UVM_LOW);
    end

    fh = $fopen("EXMEM.txt", "a+");
    $fdisplay(fh, "%8b %8b", transaction.ref_ctl_op, transaction.dut_ctl_op);
    $fdisplay(fh, "%8h %8h", transaction.ref_alu_output, transaction.dut_alu_output);
    $fdisplay(fh, "%1b %1b", transaction.ref_alu_zero, transaction.dut_alu_zero);
    $fclose(fh);

    if(transaction.ref_alu_output != transaction.dut_alu_output) begin
        mismatch = 1;
        `uvm_info("MISMATCH","Mismatch Encountered at EX/MEM Pipeline Register",UVM_LOW);
        `uvm_info("MISMATCH","Mismatching ALU Output",UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("REF ALU Out: %8h", transaction.ref_alu_output),UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("DUT ALU Out: %8h", transaction.dut_alu_output),UVM_LOW);
    end
    if(transaction.ref_alu_zero != transaction.dut_alu_zero) begin
        mismatch = 1;
        `uvm_info("MISMATCH","Mismatch Encountered at EX/MEM Pipeline Register",UVM_LOW);
        `uvm_info("MISMATCH","Mismatching ALU Zero",UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("REF ALU Zero: %1b", transaction.ref_alu_zero),UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("DUT ALU Zero: %1b", transaction.dut_alu_zero),UVM_LOW);
    end
    if(transaction.ref_ctl_op != transaction.dut_ctl_op) begin
        mismatch = 1;
        `uvm_info("MISMATCH","Mismatch Encountered at EX/MEM Pipeline Register",UVM_LOW);
        `uvm_info("MISMATCH","Mismatching CTL OP",UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("REF CTL OP: %8b", transaction.ref_ctl_op),UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("DUT CTL OP: %8b", transaction.dut_ctl_op),UVM_LOW);
    end

    fh = $fopen("MEMWB.txt", "a+");
    $fdisplay(fh, "%1b %1b %1b", transaction.ref_reg_write, transaction.ref_mem_write, transaction.ref_mem_read);
    $fdisplay(fh, "%8h %8h", transaction.ref_mem_addr, transaction.ref_mem_write_data);
    $fdisplay(fh, "%8h %8h", transaction.dut_mem_addr, transaction.dut_mem_write_data);
    $fdisplay(fh, "%d %8h", transaction.ref_reg_write_addr, transaction.ref_reg_write_data);
    $fdisplay(fh, "%d %8h", transaction.dut_reg_write_addr, transaction.dut_reg_write_data);
    $fclose(fh);

    if(transaction.ref_mem_addr != transaction.dut_mem_addr) begin
        mismatch = 1;
        `uvm_info("MISMATCH","Mismatch Encountered at MEM/WB Pipeline Register",UVM_LOW);
        `uvm_info("MISMATCH","Mismatching Memory Address",UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("REF Mem Addr: %8h", transaction.ref_mem_addr),UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("DUT Mem Addr: %8h", transaction.dut_mem_addr),UVM_LOW);
    end
    if(transaction.ref_mem_write_data != transaction.dut_mem_write_data) begin
        mismatch = 1;
        `uvm_info("MISMATCH","Mismatch Encountered at MEM/WB Pipeline Register",UVM_LOW);
        `uvm_info("MISMATCH","Mismatching Memory Write Data",UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("REF Mem Wr.Data: %8h", transaction.ref_mem_write_data),UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("DUT Mem Wr.Data: %8h", transaction.dut_mem_write_data),UVM_LOW);
    end
    if(transaction.ref_reg_write_addr != transaction.dut_reg_write_addr) begin
        mismatch = 1;
        `uvm_info("MISMATCH","Mismatch Encountered at MEM/WB Pipeline Register",UVM_LOW);
        `uvm_info("MISMATCH","Mismatching Register Write Address",UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("REF Reg Wr.Addr: %d", transaction.ref_reg_write_addr),UVM_LOW);
        `uvm_info("MISMATCH",$sformatf("DUT Reg Wr.Addr: %d", transaction.dut_reg_write_addr),UVM_LOW);
```

```
        end
        if(transaction.ref_reg_write_data != transaction.dut_reg_write_data) begin
                mismatch = 1;
                `uvm_info("MISMATCH","Mismatch Encountered at MEM/WB Pipeline Register",UVM_LOW);
                `uvm_info("MISMATCH","Mismatching Register Write Data",UVM_LOW);
                `uvm_info("MISMATCH",$sformatf("REF Reg Wr.Data: %8h", transaction.ref_reg_write_data),UVM_LOW);
                `uvm_info("MISMATCH",$sformatf("DUT Reg Wr.Data: %8h", transaction.dut_reg_write_data),UVM_LOW);
        end
endfunction

// Read information stored and perform test log formatting
// Produces the final test log for pipeline flow inspection
function void test_logging(seq_item transaction);
        int             code;
        int             fh_ifid;
        int             fh_idex;
        int             fh_exmem;
        int             fh_memwb;
        int             fh_instr;
        int             fh_stall;
        int             fh_flush;
        int             fh_jump;
        int             fh_log;
        int             instr_count;
        string          dump;
        string          instruction;
        string          stall;
        string          flush;
        string          jump;
        string          store;
        string          load;
        string          reg_write;
        string          file_transact_1;
        string          file_transact_2;
        string          file_transact_3;
        string          file_transact_4;
        string          file_transact_5;
        string          file_transact_6;
        string          file_transact_7;
        string          file_transact_8;
        string          file_transact_9;
        string          file_transact_10;

        mismatch_found = 0;
        stall_count = 0;
        flush_count = 0;
        instr_count = 1;

        fh_ifid = $fopen("IFID.txt", "r");
        fh_idex = $fopen("IDEX.txt", "r");
        fh_exmem = $fopen("EXMEM.txt", "r");
        fh_memwb = $fopen("MEMWB.txt", "r");
        fh_instr = $fopen("INSTR.txt", "r");
        fh_stall = $fopen("PIPELINE.txt", "r");
        fh_flush = $fopen("FLUSH.txt","r");
        fh_jump = $fopen("JUMP.txt","r");
        fh_log = $fopen("LOG.txt", "w");

        // Dump NULL information
        // Clear 1 cycle for ID/EX and macro status
        for(int i = 0;  i < 1;  i++)    begin
                for(int k = 0; k < 10; k++)
                        code = $fscanf(fh_idex,"%s",dump);
                code = $fscanf(fh_stall,"%s",dump);
                code = $fscanf(fh_flush,"%s",dump);
                code = $fscanf(fh_jump,"%s",dump);
        end
        // Clear 2 cycles for EX/MEM
        for(int i = 0; i < 2;   i++)    begin
                for(int k = 0; k < 6; k++)
                        code = $fscanf(fh_exmem,"%s",dump);
        end
        // Clear 3 cycles for MEM/WB
        for(int i = 0; i < 3;   i++)    begin
                for(int k = 0; k < 11; k++)
                        code = $fscanf(fh_memwb,"%s",dump);
        end

        // Start Logging
        while(!$feof(fh_ifid))  begin
                code = $fscanf(fh_ifid,"%s",file_transact_1);   //Instruction Address
                code = $fscanf(fh_ifid,"%s",file_transact_2);
                code = $fscanf(fh_ifid,"%s",file_transact_3);   //Instruction Code
                code = $fscanf(fh_ifid,"%s",file_transact_4);
                code = $fgets(instruction,fh_instr);            //Instruction Code (Assembly Language)
                if(file_transact_3 === "xxxxxxxx") begin
                        $fdisplay(fh_log, "=========================End of Testlog===========================");
                        $fdisplay(fh_log, "Test Completion Time:                          %s"     , get_time());
                        $fdisplay(fh_log, "Total Stall Encountered:                                       %3d", stall_count);
                        $fdisplay(fh_log, "Total Flush Encountered:                                       %3d", flush_count);
                        break;
                end
                $fdisplay(fh_log, "=========================Instruction %4d==========================", instr_count);
                $fdisplay(fh_log, "Instruction Address: REF: %8h\n                    DUT: %8h",
                          file_transact_1, file_transact_2);
                $fdisplay(fh_log, "Instruction Code    : REF: %8h\n                    DUT: %8h",
                          file_transact_3, file_transact_4);
                $fdisplay(fh_log, "Instruction Decode : %s", instruction);
                if((file_transact_1 != file_transact_2) || (file_transact_3 != file_transact_4)) begin
                        mismatch_found = 1;
                        break;
                end
                code = $fscanf(fh_stall,"%s",stall);    //Stall
                if(stall == "1")        begin
                        $fdisplay(fh_log, "***************Instruction Pipeline Stalling Occurred*************\n\n\n");
                        // Clear bubble information at each pipeline register
                        for(int k = 0; k < 10; k++)
                                code = $fscanf(fh_idex,"%s",dump);
                        for(int k = 0; k < 6; k++)
                                code = $fscanf(fh_exmem,"%s",dump);
                        for(int k = 0; k < 11; k++)
```

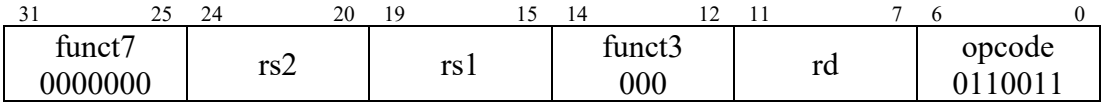
```
                code = $fscanf(fh_memwb,"%s",dump);
            // Clear macro of stalled cycle
            code = $fscanf(fh_flush,"%s",dump);
            code = $fscanf(fh_jump,"%s",dump);
            stall_count ++;
            continue;
    end
    code = $fscanf(fh_flush,"%s",flush);    //Flush
    code = $fscanf(fh_jump,"%s",jump);      //Jump
    if(flush == "1") begin
            if(jump == "0") begin //Conditional Branch Formatting
                    code = $fscanf(fh_idex,"%s",file_transact_1);   //Reg Addr 1
                    code = $fscanf(fh_idex,"%s",file_transact_2);   //Reg Data 1
                    code = $fscanf(fh_idex,"%s",file_transact_3);   //Reg Addr 2
                    code = $fscanf(fh_idex,"%s",file_transact_4);   //Reg Data 2
                    code = $fscanf(fh_idex,"%s",file_transact_5);
                    code = $fscanf(fh_idex,"%s",file_transact_6);
                    code = $fscanf(fh_idex,"%s",file_transact_7);
                    code = $fscanf(fh_idex,"%s",file_transact_8);
                    code = $fscanf(fh_idex,"%s",dump);              //Imm Value
                    code = $fscanf(fh_idex,"%s",dump);
                    $fdisplay(fh_log, "--------------------------Branch Compare--------------------------");
                    $fdisplay(fh_log, "Register Read      : REF: x%2d:%8h",
                              file_transact_1, file_transact_2);
                    $fdisplay(fh_log, "                            x%2d:%8h",
                              file_transact_3, file_transact_4);

                    $fdisplay(fh_log, "Register Read      : DUT: x%2d:%8h",
                              file_transact_5, file_transact_6);
                    $fdisplay(fh_log, "                            x%2d:%8h",
                              file_transact_7, file_transact_8);
                    if((file_transact_1 != file_transact_5) || (file_transact_2 != file_transact_6) ||
                       (file_transact_3 != file_transact_7) || (file_transact_4 != file_transact_8)) begin
                            mismatch_found = 1;
                            break;
                    end
            end
            else begin //Unconditional Branch Formatting
                    for(int k = 0; k < 7; k++) //Remove Control Signal and Memory Access
                            code = $fscanf(fh_memwb,"%s",dump);
                    code = $fscanf(fh_memwb,"%s",file_transact_1);  //Wr.Reg Addr
                    code = $fscanf(fh_memwb,"%s",file_transact_2);  //Wr.Reg Data
                    code = $fscanf(fh_memwb,"%s",file_transact_3);
                    code = $fscanf(fh_memwb,"%s",file_transact_4);
                    $fdisplay(fh_log, "-------------------------Register Access-------------------------");
                    $fdisplay(fh_log, "Register Write     : REF: x%2d:%8h",
                              file_transact_1, file_transact_2);
                    $fdisplay(fh_log, "                            DUT: x%2d:%8h",
                              file_transact_3, file_transact_4);
                    if((file_transact_1 != file_transact_3) || (file_transact_2 != file_transact_4)) begin
                            mismatch_found = 1;
                            break;
                    end
            end
            $fdisplay(fh_log, "**************Instruction Pipeline Flushing Occurred**************\n\n\n");
            //Remove 2 cycles of flushed information
            for(int k = 0; k < 2; k++)      begin
                    for(int j = 0; j < 4; j++)
                            code = $fscanf(fh_ifid,"%s",dump);
                    for(int j = 0; j < 10; j++)
                            code = $fscanf(fh_idex,"%s",dump);
                    for(int j = 0; j < 11; j++)
                            code = $fscanf(fh_memwb,"%s",dump);
                    code = $fgets(dump,fh_instr);
                    code = $fscanf(fh_stall,"%s",dump);
                    code = $fscanf(fh_flush,"%s",dump);
                    code = $fscanf(fh_jump,"%s",dump);
            end
            //Remove an additional cycle on EX/MEM
            for(int k = 0;  k < 3;  k++)    begin
                    for(int j = 0; j < 6; j++)
                            code = $fscanf(fh_exmem,"%s",dump);
            end
            //Remove unused information for Conditional / Unconditional Branch
            if(jump == "0") begin
                    for(int j = 0; j < 11; j++)
                            code = $fscanf(fh_memwb,"%s",dump);
            end
            else begin
                    for(int j = 0; j < 10; j++)
                            code = $fscanf(fh_idex,"%s",dump);
            end
            flush_count ++;
            instr_count ++;
            continue;
    end
    code = $fscanf(fh_idex,"%s",file_transact_1);   //Reg Addr 1
    code = $fscanf(fh_idex,"%s",file_transact_2);   //Reg Data 1
    code = $fscanf(fh_idex,"%s",file_transact_3);   //Reg Addr 2
    code = $fscanf(fh_idex,"%s",file_transact_4);   //Reg Addr 2
    code = $fscanf(fh_idex,"%s",file_transact_5);
    code = $fscanf(fh_idex,"%s",file_transact_6);
    code = $fscanf(fh_idex,"%s",file_transact_7);
    code = $fscanf(fh_idex,"%s",file_transact_8);
    code = $fscanf(fh_idex,"%s",file_transact_9);   //Imm Value
    code = $fscanf(fh_idex,"%s",file_transact_10);
    $fdisplay(fh_log, "-------------------------Register Access-------------------------");
    $fdisplay(fh_log, "Register Read      : REF: x%2d:%8h",
              file_transact_1, file_transact_2);
    $fdisplay(fh_log, "                            x%2d:%8h",
              file_transact_3, file_transact_4);
    $fdisplay(fh_log, "Register Read      : DUT: x%2d:%8h",
              file_transact_5, file_transact_6);
    $fdisplay(fh_log, "                            x%2d:%8h",
              file_transact_7, file_transact_8);
    $fdisplay(fh_log, "Immediate  Value   : REF:     %8h\n                    DUT:     %8h",
              file_transact_1, file_transact_2);
    if((file_transact_1 != file_transact_5) || (file_transact_2 != file_transact_6) ||
       (file_transact_3 != file_transact_7) || (file_transact_4 != file_transact_8) ||
       (file_transact_9 != file_transact_10)) begin
            mismatch_found = 1;
```

```
                        break;
                    end
                    code = $fscanf(fh_exmem,"%s",file_transact_1);  //Control Signal
                    code = $fscanf(fh_exmem,"%s",file_transact_2);
                    code = $fscanf(fh_exmem,"%s",file_transact_3);  //ALU Output
                    code = $fscanf(fh_exmem,"%s",file_transact_4);
                    code = $fscanf(fh_exmem,"%s",file_transact_5);  //Zero
                    code = $fscanf(fh_exmem,"%s",file_transact_6);
                    $fdisplay(fh_log, "------------------------ALU Operation-----------------------------");

                    $fdisplay(fh_log,       "Control Signal     : REF: %8b\n                    DUT: %8b",
                                file_transact_1, file_transact_2);
                    $fdisplay(fh_log,       "ALU Output         : REF: %8h\n                    DUT: %8h",
                                file_transact_1, file_transact_2);
                    $fdisplay(fh_log,       "ALU Zero Signal    : REF: %1b\n                    DUT: %1b",
                                file_transact_1, file_transact_2);
                    if((file_transact_1 != file_transact_2) || (file_transact_3 != file_transact_4) ||
                       (file_transact_5 != file_transact_6)) begin
                            mismatch_found = 1;
                            break;
                    end
                    code = $fscanf(fh_memwb,"%s",reg_write);        //RegWrite
                    code = $fscanf(fh_memwb,"%s",store);            //MemWrite
                    code = $fscanf(fh_memwb,"%s",load);             //MemRead
                    code = $fscanf(fh_memwb,"%s",file_transact_1);  //Mem Address
                    code = $fscanf(fh_memwb,"%s",file_transact_2);  //Mem Wr.Data
                    code = $fscanf(fh_memwb,"%s",file_transact_3);
                    code = $fscanf(fh_memwb,"%s",file_transact_4);
                    code = $fscanf(fh_memwb,"%s",file_transact_5);  //Wr.Reg Addr
                    code = $fscanf(fh_memwb,"%s",file_transact_6);  //Wr.Reg Data
                    code = $fscanf(fh_memwb,"%s",file_transact_7);
                    code = $fscanf(fh_memwb,"%s",file_transact_8);
                    $fdisplay(fh_log, "-------------------------Writeback Access-------------------------");

                    if (store == "1")       begin
                            $fdisplay(fh_log, "Memory Write       : REF:     %4h:%8h",
                                        file_transact_1, file_transact_2);
                            $fdisplay(fh_log, "                     DUT:     %4h:%8h",
                                        file_transact_3, file_transact_4);
                            if((file_transact_1 != file_transact_3) || (file_transact_2 != file_transact_4)) begin
                                    mismatch_found = 1;
                                    break;
                            end
                    end
                    if(load == "1" || reg_write == "1")     begin
                            $fdisplay(fh_log, "Register Write     : REF: x%2d:%8h",
                                        file_transact_5, file_transact_6);
                            $fdisplay(fh_log, "                     DUT: x%2d:%8h",
                                        file_transact_7, file_transact_8);
                            if((file_transact_5 != file_transact_7) || (file_transact_6 != file_transact_8)) begin
                                    mismatch_found = 1;
                                    break;
                            end
                    end
                     if(store != "1" && load != "1" && reg_write != "1")
                            $fdisplay(fh_log, "\n");
                     $fdisplay(fh_log, "************************End of Instruction**********************\n\n\n");
                     instr_count ++;
            end
            $fclose(fh_ifid);
            $fclose(fh_idex);
            $fclose(fh_exmem);
            $fclose(fh_memwb);
            $fclose(fh_log);
            $fclose(fh_instr);
            $fclose(fh_stall);
            $fclose(fh_flush);
            $fclose(fh_jump);
    endfunction

    // Simple function for moving file to correct directory
    function void move_file(string file, string destination);
            string full_command;
            full_command = {"move ./", file, " ./", destination};
            $system(full_command);
    endfunction

    // Simple function to remove all temporary logging files
    function void clear_current_directory();
            $system("del PROM.txt");
            $system("del INSTR.txt");
            $system("del IFID.txt");
            $system("del IDEX.txt");
            $system("del EXMEM.txt");
            $system("del MEMWB.txt");
            $system("del PIPELINE.txt");
            $system("del FLUSH.txt");
            $system("del JUMP.txt");
    endfunction

    // Check macro status with history value or update macro history
    function void macro_check();
            int     fh;
            int     code;
            string  file_directory;
            string  file_address;

            stall_check = 0;
            flush_check = 0;

            $display("Begin Macro Checking");
            if(!directed_test)
                    file_directory = {"test_repo/", test_seed};
            else
                    file_directory = {"test_repo/directed_test"};
            file_address = {file_directory, "/MACRO.txt"};
            fh = $fopen(file_address, "r");
            if(fh && !force_gen && !macro_overwrite) begin
                    code = $fscanf(fh, "Stall: %d", stall_check);
                    code = $fscanf(fh, "Flush: %d", flush_check);
```

```
                        $fclose(fh);
                        if(stall_check == stall_count && flush_check == flush_count) begin
                                $display("Macro Check Passed");
                                $display("------------------------------------------------------------------");
                                $display("M A C R O     C H E C K     P A S S E D          M A C R O     C H ");
                                $display("------------------------------------------------------------------");
                                fh = $fopen("LOG.txt", "a+");
                                $fdisplay(fh, "Macro Check                                                  Passed");
                        end
                        else begin
                                $display("Macro Check Failed");
                                $display("XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX");
                                $display("M A C R O     C H E C K     F A I L E D          M A C R O     C H ");
                                $display("XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX");
                                fh = $fopen("LOG.txt", "a+");
                                $fdisplay(fh, "Macro Check                                                  Failed");
                                `uvm_info("MACRO MISMATCH","Macro Check Failed",UVM_LOW)
                                if(run_all) begin
                                        $fclose(fh);
                                        fh = $fopen("FAILED.txt","a+");
                                        $fdisplay(fh, "%s - Macro Mismatch", test_seed);
                                end
                        end
                end
                else begin
                        $fclose(fh);
                        fh = $fopen(file_address, "w");
                        $fdisplay(fh, "Stall: %d", stall_count);
                        $fdisplay(fh, "Flush: %d", flush_count);
                        $fdisplay(fh, "Last Updated on %s", get_time());
                        $fclose(fh);
                        $display("Macro History Updated");
                        $display("//////////////////////////////////////////////////////////////////////");
                        $display("M A C R O     U P D A T E D          M A C R O     U P D A T E D   ");
                        $display("//////////////////////////////////////////////////////////////////////");
                        fh = $fopen("LOG.txt", "a+");
                        $fdisplay(fh, "Macro Check                                                Updated");
                end
                $fclose(fh);
        endfunction

        // Clear static variable for continuous batch test
        function void clear_static_variable();
                data_1_buffer = 0;
                data_2_buffer = 0;
                data_1 = 0;
                data_2 = 0;
                load_flag = 0;
                load_use_flag = 0;
                stall_assertion = 0;
                stall_buffer = 0;
                write_data_buffer = 0;
                write_data_check = 0;
                write_address_buffer = 0;
                write_address_check = 0;
                reg_write_check_buffer = 0;
                reg_write_check = 0;
                branch_check_flag = 0;
                jump_check_flag = 0;
                flush_buffer = 0;
                flush_assertion = 0;
        endfunction
endclass
`endif
```

APPENDIX CC: Instruction Set Manual

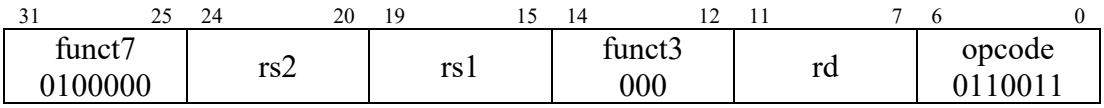
ADD **Addition**



Assembly Code Format: add rd, rs1, rs2

Description: Performs addition on the contents stored on source register *rs1* and *rs2* and stores the result onto destination register *rd*.

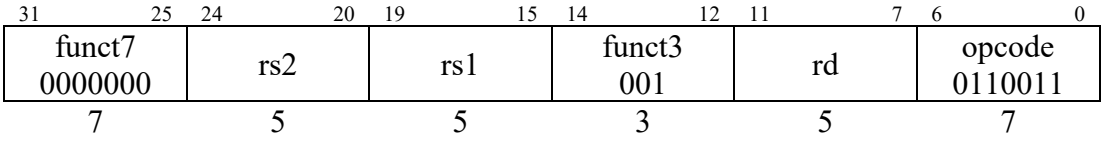
SUB **Subtraction**



Assembly Code Format: sub rd, rs1, rs2

Description: Performs subtraction on the contents stored on source register *rs1* and *rs2* and stores the result onto destination register *rd*.

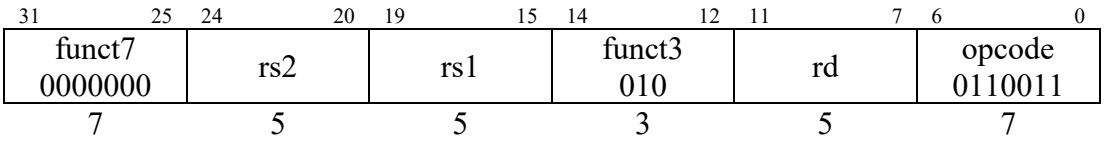
SLL **Shift Left Logical**



Assembly Code Format: sll rd, rs1, rs2

Description: Performs logical left shift on the register content stored on source register *rs1* by a shift amount specified by the lower 5 bits of register *rs2* and stores the result onto destination register *rd*. The shifted bits are replaced with 0s.

SLT **Set Less Than**



Assembly Code Format: slt rd, rs1, rs2

Description: Performs signed comparison between contents of source registers *rs1* and *rs2* and sets destination register *rd* to 1 if *rs1* is lesser than *rs2*, or 0 otherwise.

SLTU **Set Less Than Unsigned**

31 25	24 20	19 15	14 12	11 7	6 0
funct7 0000000	rs2	rs1	funct3 011	rd	opcode 0110011
7	5	5	3	5	7

Assembly Code Format: sltu rd, rs1, rs2

Description: Performs unsigned comparison between contents of source registers *rs1* and *rs2* and sets destination register *rd* to 1 if *rs1* is lesser than *rs2*, or 0 otherwise.

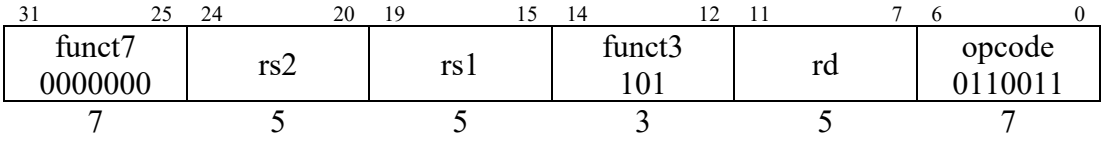
XOR **Bitwise Logical Exclusive OR**

31 25	24 20	19 15	14 12	11 7	6 0
funct7 0000000	rs2	rs1	funct3 100	rd	opcode 0110011
7	5	5	3	5	7

Assembly Code Format: xor rd, rs1, rs2

Description: Performs bitwise logical exclusive OR on the contents of source register *rs1* and *rs2* and writes the result to the destination register *rd*.

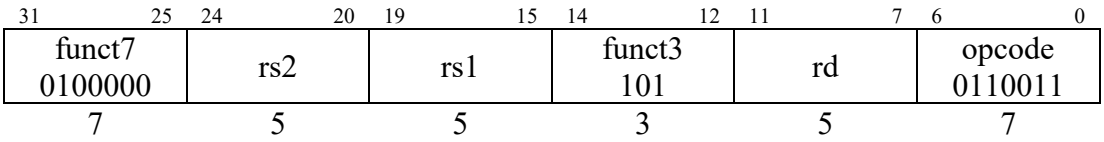
SRL **Shift Right Logical**



Assembly Code Format: srl rd, rs1, rs2

Description: Performs logical right shift on the register content stored on source register *rs1* by a shift amount specified by the lower 5 bits of register *rs2* and stores the result onto destination register *rd*. The shifted bits are replaced with 0s.

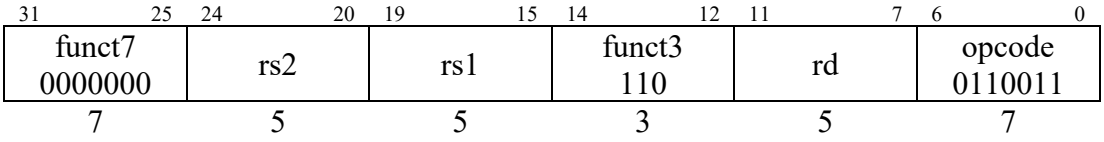
SRA **Shift Right Arithmetic**



Assembly Code Format: sra rd, rs1, rs2

Description: Performs arithmetic right shift on the register content stored on source register *rs1* by a shift amount specified by the lower 5 bits of register *rs2* and stores the result onto destination register *rd*. The shifted bits are replaced with the sign bit.

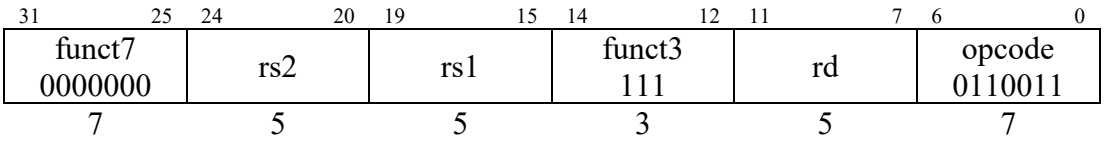
OR **Bitwise Logical OR**



Assembly Code Format: or rd, rs1, rs2

Description: Performs bitwise logical OR on the contents of source register *rs1* and *rs2* and writes the result to the destination register *rd*.

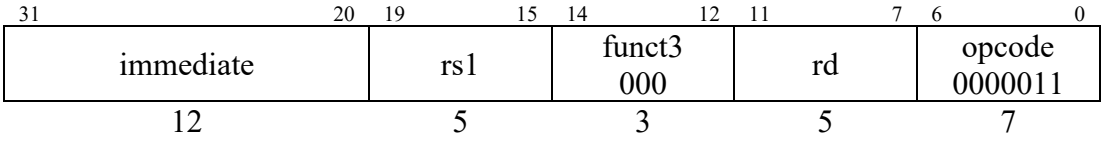
AND **Bitwise Logical AND**



Assembly Code Format: and rd, rs1, rs2

Description: Performs bitwise logical AND on the contents of source register *rs1* and *rs2* and writes the result to the destination register *rd*.

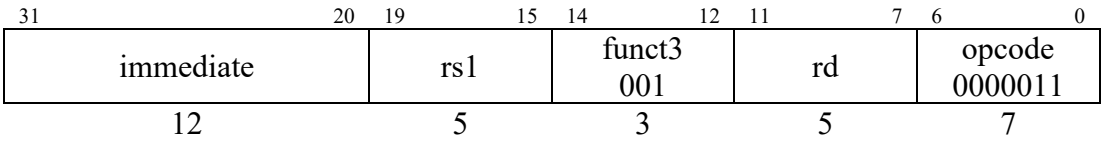
LB **Load Byte**



Assembly Code Format: lb rd, offset(rs1)

Description: Loads an 8-bit value from memory into destination register *rd*. The 8-bit value loaded is sign-extended to 32-bits before storing into *rd*.

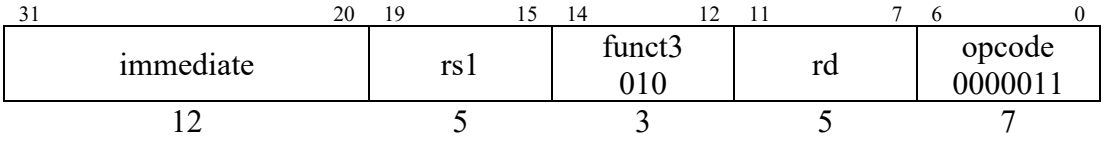
LH **Load Halfword**



Assembly Code Format: lh rd, offset(rs1)

Description: Loads a 16-bit value from memory into destination register *rd*. The 16-bit value loaded is sign-extended to 32-bits before storing into *rd*.

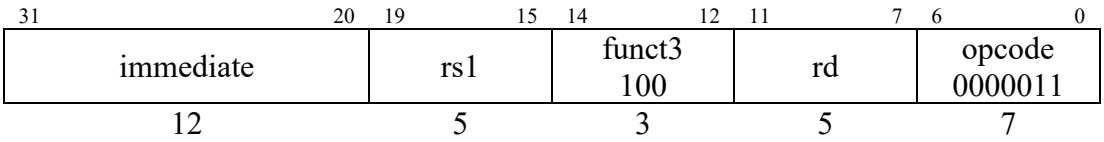
LW **Load Word**



Assembly Code Format: lw rd, offset(rs1)

Description: Loads a 32-bit value from memory into destination register *rd*.

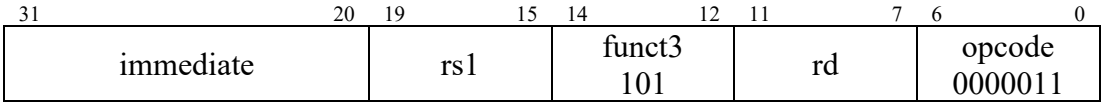
LBU **Load Byte Unsigned**



Assembly Code Format: lbu rd, offset(rs1)

Description: Loads an 8-bit value from memory into destination register *rd*. The 8-bit value loaded is zero-extended to 32-bits before storing into *rd*.

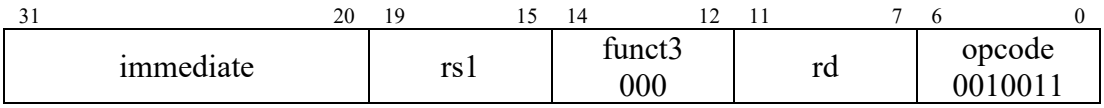
LHU **Load Halfword Unsigned**



Assembly Code Format: lhu rd, offset(rs1)

Description: Loads a 16-bit value from memory into destination register *rd*. The 16-bit value loaded is zero-extended to 32-bits before storing into *rd*.

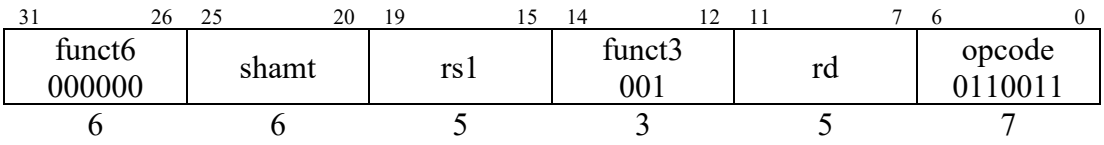
ADDI **Add Immediate**



Assembly Code Format: addi rd, rs1, immediate

Description: Performs addition on the content stored on source register *rs1* and a sign-extended 12-bit immediate and stores the result onto destination register *rd*.

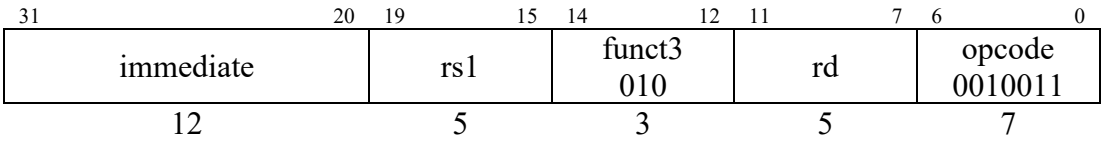
SLLI



Assembly Code Format: slli rd, rs1, shamt

Description: Performs logical left shift on the register content stored on source register *rs1* by a shift amount specified by *shamt* and stores the result onto destination register *rd*. The shifted bits are replaced with 0s.

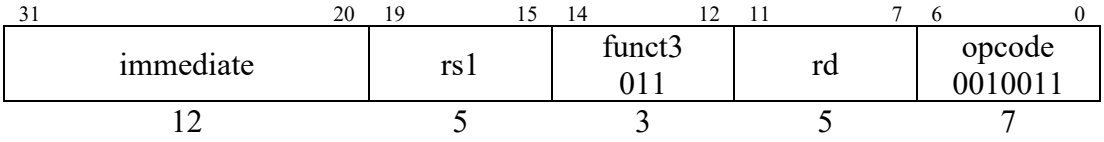
SLTI **Set Less Than Immediate**



Assembly Code Format: slti rd, rs1, immediate

Description: Performs signed comparison between contents of source registers *rs1* and a sign-extended 12-bit immediate and sets destination register *rd* to 1 if *rs1* is lesser than the sign-extended immediate value, or 0 otherwise.

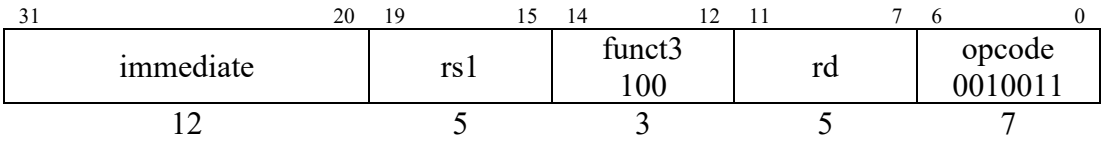
SLTIU **Set Less Than Immediate Unsigned**



Assembly Code Format: sltiu rd, rs1, immediate

Description: Performs unsigned comparison between contents of source registers *rs1* and a sign-extended 12-bit immediate and sets destination register *rd* to 1 if *rs1* is lesser than the sign-extended immediate value, or 0 otherwise.

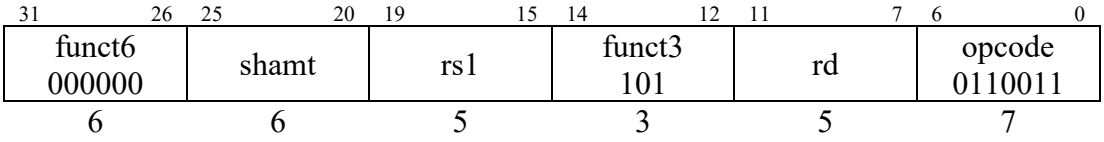
XORI **Bitwise Logical Exclusive OR Immediate**



Assembly Code Format: xori rd, rs1, immediate

Description: Performs bitwise logical exclusive or on the content of source register *rs1* and a sign-extended 12-bit immediate and writes the result to the destination register *rd*.

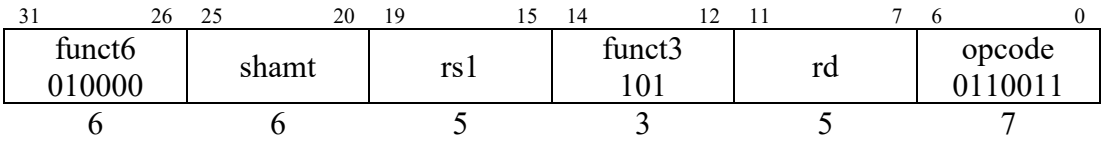
SRLI **Shift Right Logical Immediate**



Assembly Code Format: srli rd, rs1, shamt

Description: Performs logical right shift on the register content stored on source register *rs1* by a shift amount specified by *shamt* and stores the result onto destination register *rd*. The shifted bits are replaced with 0s.

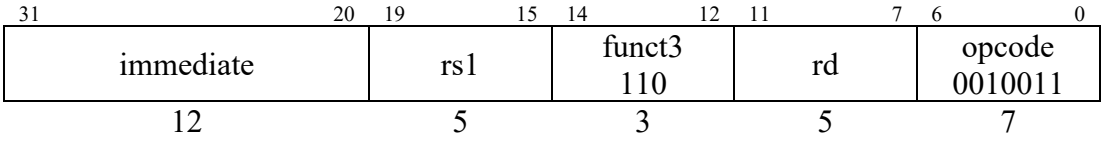
SRAI **Shift Right Arithmetic Immediate**



Assembly Code Format: srai rd, rs1, shamt

Description: Performs arithmetic right shift on the register content stored on source register *rs1* by a shift amount specified by *shamt* and stores the result onto destination register *rd*. The shifted bits are replaced with sign bit.

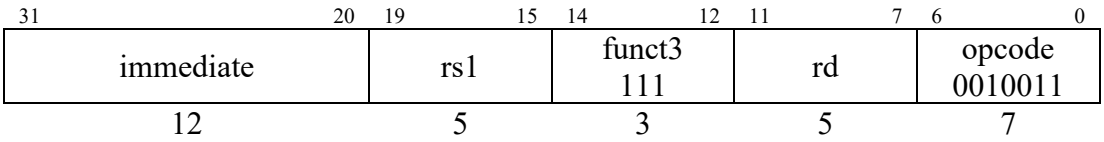
ORI **Bitwise Logical OR Immediate**



Assembly Code Format: ori rd, rs1, immediate

Description: Performs bitwise logical or on the content of source register *rs1* and a sign-extended 12-bit immediate and writes the result to the destination register *rd*.

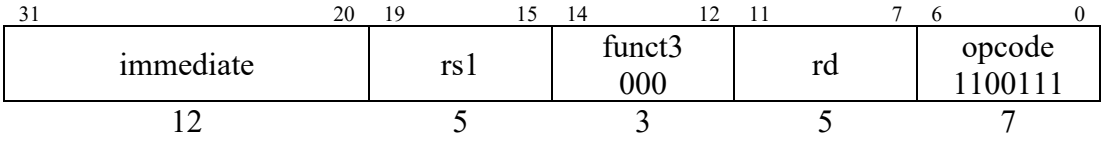
ANDI **Bitwise Logical AND Immediate**



Assembly Code Format: andi rd, rs1, immediate

Description: Performs bitwise logical and on the content of source register *rs1* and a sign-extended 12-bit immediate and writes the result to the destination register *rd*.

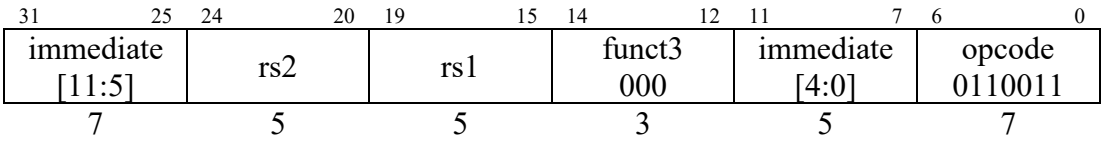
JALR **Jump and Link Register**



Assembly Code Format: jalr rd, offset(rs1)

Description: Performs indirect jump to a target address. The target address is obtained by summing the offset to the content of source register *rs1* and setting the two least significant bits of the result to zero. The address of the subsequent instruction (program counter + 4) is stored onto destination register *rd*.

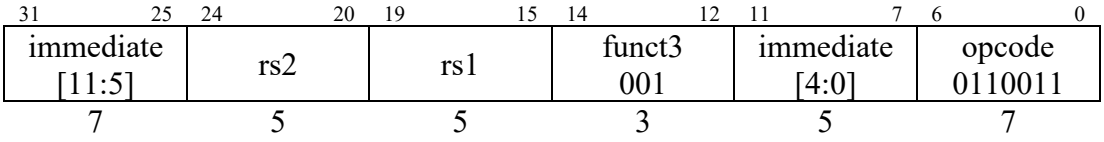
SB **Store Byte**



Assembly Code Format: sb rs2, offset(rs1)

Description: Stores 8-bit value from the low bits of the source register *rs2* onto target memory address. Target memory address is obtained by summing the offset to the content of source register *rs1*.

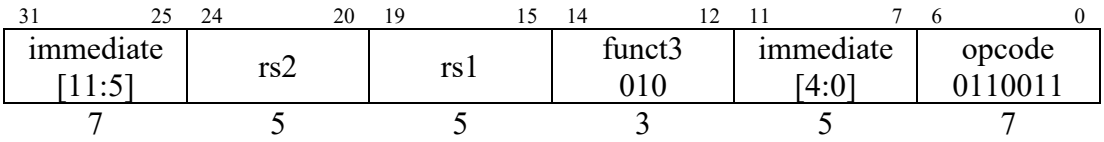
SH **Store Halfword**



Assembly Code Format: sh rs2, offset(rs1)

Description: Stores 16-bit value from the low bits of the source register *rs2* onto target memory address. Target memory address is obtained by summing the offset to the content of source register *rs1*.

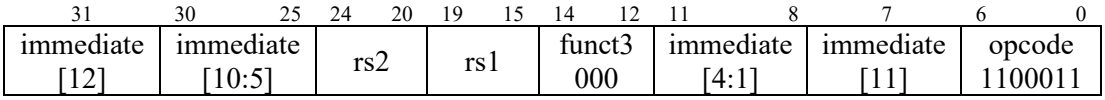
SW **Store Word**



Assembly Code Format: sw rs2, offset(rs1)

Description: Stores 32-bit value from the source register *rs2* onto target memory address. Target memory address is obtained by summing the offset to the content of source register *rs1*.

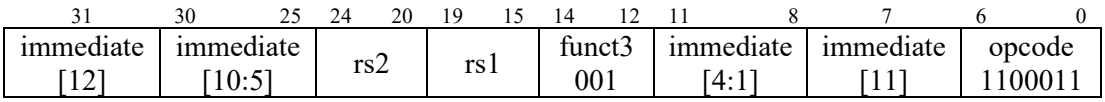
BEQ **Branch if Equal**



Assembly Code Format: beq rs1, rs2, immediate

Description: Compares the contents of source register *rs1* and *rs2*. If the contents of source registers are equal, branch is executed to a target address formed by adding the offset to the program counter.

BNE **Branch if Not Equal**



Assembly Code Format: bne rs1, rs2, immediate

Description: Compares the contents of source register *rs1* and *rs2*. If the contents of source registers are not equal, branch is executed to a target address formed by adding the offset to the program counter.

BLT **Branch if Less Than**

31	30 25	24 20	19 15	14 12	11 8	7	6 0
immediate [12]	immediate [10:5]	rs2	rs1	funct3 100	immediate [4:1]	immediate [11]	opcode 1100011
1	6	5	5	3	4	1	7

Assembly Code Format: blt rs1, rs2, immediate

Description: Compares the contents of source register *rs1* and *rs2*. If the content *rs1* is lesser than *rs2*, branch is executed to a target address formed by adding the offset to the program counter.

BGE **Branch if Greater or Equal**

31	30 25	24 20	19 15	14 12	11 8	7	6 0
immediate [12]	immediate [10:5]	rs2	rs1	funct3 101	immediate [4:1]	immediate [11]	opcode 1100011
1	6	5	5	3	4	1	7

Assembly Code Format: bge rs1, rs2, immediate

Description: Compares the contents of source register *rs1* and *rs2*. If the content *rs1* is greater than or equal to *rs2*, branch is executed to a target address formed by adding the offset to the program counter.

BLTU **Branch if Less Than Unsigned**

31	30 25	24 20	19 15	14 12	11 8	7	6 0
immediate [12]	immediate [10:5]	rs2	rs1	funct3 110	immediate [4:1]	immediate [11]	opcode 1100011
1	6	5	5	3	4	1	7

Assembly Code Format: bltu rs1, rs2, immediate

Description: Compares the unsigned contents of source register *rs1* and *rs2*. If the content *rs1* is lesser than *rs2*, branch is executed to a target address formed by adding the offset to the program counter.

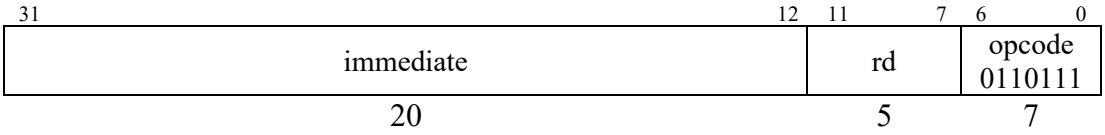
BGEU **Branch if Greater or Equal Unsigned**

31	30 25	24 20	19 15	14 12	11 8	7	6 0
immediate [12]	immediate [10:5]	rs2	rs1	funct3 111	immediate [4:1]	immediate [11]	opcode 1100011
1	6	5	5	3	4	1	7

Assembly Code Format: bgeu rs1, rs2, immediate

Description: Compares the unsigned contents of source register *rs1* and *rs2*. If the content *rs1* is greater than or equal to *rs2*, branch is executed to a target address formed by adding the offset to the program counter.

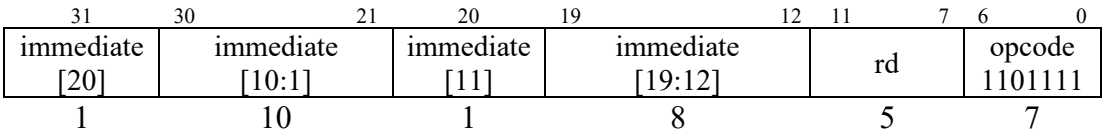
LUI **Load Upper Immediate**



Assembly Code Format: lui rd, immediate

Description: Places the 20-bit immediate in the high bits of destination register *rd* and fill the lower 12-bit with 0s.

JAL **Jump and Link**



Assembly Code Format: jal rd, immediate

Description: Performs indirect jump to a target address obtained by summing the shifted offset to the program counter. The address of the subsequent instruction (program counter + 4) is stored onto destination register *rd*.