DESIGN AND SIMULATE RISC-V PROCESOR USING VERILOG

DAVID NGU TECK JOUNG

MASTER OF ENGINEERING IN ELECTRONIC SYSTEMS

FACULTY OF ENGINEERING AND GREEN TECHNOLOGY

UNIVERSITI TUNKU ABDUL RAHMAN

AUGUST 2023

DESIGN AND SIMULATE RISC-V PROCESOR USING VERILOG

By

DAVID NGU TECK JOUNG

A dissertation submitted to

Faculty of Engineering and Green Technology,

Universiti Tunku Abdul Rahman,

in partial fulfilment of the requirements for the degree of

Master of Engineering in Electronic Systems

AUGUST 2023

ABSTRACT

DESIGN AND SIMULATE RISC-V PROCESOR USING VERILOG

DAVID NGU TECK JOUNG

In this project, RISC-V processor is designed and simulated using Verilog. The design of RISC-V processor provides an alternative for software and hardware design to the computer designers as it provides free and open instruction set architecture (ISA). Besides, the designed RISC-V processor will be using 5-stage pipeline techniques to improve the overall performance of the processor. The project is started by implementing several main modules, such as *alu, aludec, maindec, imem, dmem, regfile, pc_mux, result_mux,* pipeline register (*IF/ID, ID/IEx, IEx/IMem,* and *IMem/IW*), *forwardMuxA* and *forwardMuxB*. Besides, hazard unit is implemented into the design to mitigate hazard conditions. The functionality of these modules were simulated and verified by using ModelSim software. Then, the modules were integrated into a main module to form a *riscv_pip_27* module. A simple testbench is written to verify the functionality of the RISC-V processor.

Keywords – RISC-V processor, Verilog, 5-stage pipeline, hazard

ACKNOWLEDGEMENTS

I would like to extend my sincere gratitude to my Master project supervisor, Dr. Loh Siu Hong, for his unwavering support, motivation, and vast knowledge throughout my research journey. In times of trouble or difficulty, he patiently listened to my concerns and guided me in the right direction. His invaluable guidance played a pivotal role in the entire research process and the completion of this dissertation.

I am also deeply thankful to the moderators of my Master project, Dr. Lee Han Kee and Ts Tan Yee Chyan, for their insightful comments, encouragement, and thought-provoking questions. Their guidance inspired me to explore my research from various perspectives and broaden my horizons.

Lastly, I would like to express my profound gratitude to my parents and friends for their unfailing support and continuous encouragement throughout the research journey and the process of writing this dissertation. Their presence and encouragement were instrumental in achieving this milestone. I am sincerely grateful for their contributions.

APPROVAL SHEET

This dissertation entitled "**<u>DESIGN AND SIMULATE RISC-V PROCESSOR USING VERILOG</u>**" was prepared by DAVID NGU TECK JOUNG and submitted as partial fulfilment of requirements for the degree of Master of Engineering in Electronic Systems at Universiti Tunku Abdul Rahman.

Approved by:

____

(Dr. Loh Siu Hong) Date:…………………

Supervisor

Department of Electronic Engineering

Faculty of Engineering and Green Technology

Universiti Tunku Abdul Rahman

FACULTY OF ENGINEERING AND GREEN TECHNOLOGY

UNIVERSITI TUNKU ABDUL RAHMAN

Date: ____________

SUBMISSION OF DISSERTATION

It is hereby certified that **David Ngu Teck Joung** (ID No: **21AGM06719**) has completed this dissertation entitled "DESIGN AND SIMULATE RISC-V PROCESSOR USING VERILOG" under the supervision of Dr. Loh Siu Hong (Supervisor) from the Department of Electronic Engineering, Faculty of Engineering and Green Technology.

I understand that the University will upload softcopy of my dissertation in pdf format into UTAR Institutional Repository, which may be made accessible to UTAR community and public.

Yours truly,

____

(David Ngu Teck Joung)

DECLARATION

I David Ngu Teck Joung hereby declare that the thesis/dissertation is based on my original work except for quotations and citations which have been duly acknowledged. I also declare that it has not been previously or concurrently submitted for any other degree at UTAR or other institutions.

____

(DAVID NGU TECK JOUNG)

Date: ________________

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOL AND ABBREVIATIONS

ISA	Instruction Set Architecture
HDL	Hardware Description Language
CISC	Complex Instruction Set Computing
RISC	Reduced Instruction Set Computing
ARM	Advanced RISC Machines
MIPS	Microprocessor without Interlocked Pipelined Stages
FPGA	Field Programmable Gate Arrays
VWF	Vector Waveform File
PC	Program Counter
CPU	Central Processor Unit
MUX	Multiplexer
PCF	Program Counter Fetch
alu	Arithmetic Logic Unit
pc_mux	Program Counter Multiplexer
imem	Instruction Memory
dmem	Data Memory
regfile	Register File
aludec	Arithmetic Logic Unit Decoder
maindec	Main Decoder
result_mux	Write Data Selection Multiplexer
hazard_unit	Hazard Unit
forwardMuxA	Forward Multiplexer A
forwardMuxB	Forward Multiplexer B

IF/ID	Fetch and Decoder pipeline register
ID/IEx	Decode and Execute pipeline register
IEx_IMem	Execute and Memory pipeline register
IMem/IW	Memory and Writeback pipeline register

Chapter 1: Introduction

1.1 Project Overview

Very Large-Scale Integration (VLSI) design began in 1970s when semiconductor and communication technologies were being developed. It driven the revolution of microprocessors and the innovation of electronic computing systems. This scenario was further extended that extensive research fund was invested to explore the capabilities of the applications of computer in various fields such as aviation, medical, cell phones, and automobile industries. By 2030, more than 22.7 billion devices will be connected by IoT networks (W. Wang et al., 2021). These IoT devices lay as a foundation towards the accomplishment of future concepts such as smart city, self-driving cars, and space exploration technologies.

VLSI is the process of creating an integrated circuit (IC) by combining thousands of transistors into a single chip. Before the introduction of VLSI technology, most ICs had limited set of functions that they could perform. VLSI allows IC designers to add all the components such as Central Processing Unit (CPU), Read-Only Memory (ROM), Random Access Memory (RAM), and other logics together into a chip. The current cutting-edge technologies such as high resolution and low-bit rate video and cellular communications provide the end-users a marvelous number of applications, processing power and portability. This trend is expected to grow rapidly with very important implications on VLSI design and system design.

The processor is the “brain” of the established electronic computer system today. It helps to communicate with other devices such as mouse, keyboard, speakers, and works on the information that is has acquired from them. Most of the processors nowadays are built in Complex Instruction Set Computing (CISC) or Reduced Instruction Set Computing (RISC) architecture. They proposed different design styles and circuit that differ in how the data flows or where the data and instructions were stored. Both CISC and RISC had their own advantages and disadvantages. Therefore, implementing and enhancing the advantages on existing ones is still impactful to both academic and industries.

RISC-V is a free and open Instruction Set Architecture (ISA) based on established RISC architecture. It was founded in 2010 at the University of California, Berkeley. RISC-V started to gain attention from the industries because it provided open-source licenses that do not require fees to use. With this, it breaks down barriers in the semiconductor industries. RISC-V is fundamentally designed for modular approach. It only has 47 based instructions, and it can be modularly adjusting those extension based on the design requirement. RISC-V ISA does not define how a design must be implemented or which subsets it must contain. Therefore, many RISC-V computers might implement the compact extension to reduce power consumption, code size, and memory use.

1.2 Problem Statement

Instruction Set Architecture (ISA) is part of the abstract model of a computer that defines how the CPU is controlled by the software. It acts as an interface between the hardware and the software. It also defined the supported data types, registers, memories, instruction to be executed, and features of a processor. Companies such as Intel, IBM and ARM uses their own ISAs in their products. Unfortunately, these ISAs were patterned to avert others from using them without a permit. Negotiations takes months and it tends to peak up the cost, which make it difficult for community apprehensive and small organizations. To overcome this issue, an open ISA should be found to attain sizable innovation. With shared open core designs, it helps some small organizations to compete in the market. This possesses a positive competitive atmosphere between them. Therefore, consumers can be benefit by purchasing affordable product with adequate performances.

As stated in RISC-V organization webpage, “The worldwide interest in RISC-V is not because it is a great new chip technology, the interest is because it is a common free and open standard to which software can be ported, and which allows anyone to freely develop their own hardware to run the software”. These properties of the RISC-V ISA make it ideal for our desire use.

Efficiency of the processor is strongly affected by instruction implementation. Processors with single cycle design will execute the next instruction only when the current instruction is completed. The efficiency of the single-cycle processor will be greatly reduced when the complexity of the instructions increases. Besides, the cycle time of the processor need to be designed to accommodate the slowest instruction. Therefore, the latency of the processor will be increased due to the length of the clock cycle being too long for the execution of each instruction. With pipeline design, this problem can be overcome by executing multiple instruction simultaneous in overlapped manner.

1.3 Objective

1. To understand the basic of RISC-V architecture
2. To implement 5-stage pipeline design of the RISC-V processor in Verilog
3. To verify the functionality of the design by performing testbench and simulation

Chapter 2: Literature Review

2.1 Overview

In this chapter, the implementation of difference architectures such as MIPS, RISC, and CISC by other authors will be discussed.

2.2 Previous Work

The design of a 32-bit RISC processor based on MIPS using VHDL coding was presented by (S. P. Pitpurkar et al., 2015). They argued that RISC CPU had more benefits than CISC such as higher speed, simpler structure, and easier implementation. They used pipeline design to describe the system and achieve fewer clock cycles per instruction. They verified the design through extensive simulations. They used Xilinx 13.li ISE Simulator to design, synthesize and simulate the RISC processor based on MIPS. Their results showed that the design had a combinational delay of 0.758 ns and a maximum operating frequency of 1.350 Ghz.

Using Cadence, a software tool for electronic design automation, (Mohit N. Topiwala et al., 2014) designed and implemented a 32-bit processor based on MIPS. MIPS is a RISC architecture, which stands for reduced instruction set computer. RISC architectures aim to increase the speed of the processor by using a small set of simple and fast instructions. The authors stated that power consumption is a critical factor for embedded and portable applications. However, there is a trade-off between power, area, and delay in integrated circuits. For some applications, low power circuits are required, and the design engineers have to sacrifice more area and delay. Therefore, they suggested a power reduction technique by skipping pipeline stages that cause unnecessary switching activities. They designed Hazard detection unit and Data forwarding unit for efficient implementation of the pipeline. They used Verilog-HDL to implement the design and Cadence RTL complier to synthesize it using typical libraries of TSMC 0.18 μm technology.

Using Verilog HDL, (Shofiqul Islam et al., 2006) developed a high speed-pipelined execution unit of 32-bit RISC processor. They arranged the block in different stages of pipeline so that the pipeline can operate at high frequency. The execution stage in a typical pipeline scheme consists of input data mux, operational block and output ALU mux. To increase the speed of the pipeline, they selected the data for the computational blocks in the execution stage one stage earlier in the data select stage. They also proposed a dependency resolver module to deal with a possible problem of consecutive data dependent instruction in the pipeline. This module handles both stalling and data forwarding. They synthesized the processor at 0.1 micron technology and achieved a working frequency of 714Mhz.

(Animesh Kulshreshtha et al., 2021) compared the behavioural models of 16-bit and 32-bit RISC processors and their different instruction sets. The 16-bit RISC processor was a non-pipeline CPU based on Harvard architecture, which had separate data memory and instruction memory. The 32-bit RISC processor was a pipelined CPU that followed the MIPS architecture. They aimed to study the differences between the models based on their instruction set and performance factors such as speed and power consumption. They used an optimized Multiplier algorithm to improve the data path. In general, the 32-bit processor consumed about 60% more power than the 16-bit processor because of its higher operating frequency. However, the 32-bit processor was 70% faster than the 16-bit processor. These results were expected because the 32-bit processor can store more computational values and the pipelined architecture of the processor reduces the length of each instruction cycle, which increases the operating frequency and decreases the combinational delay. Based on their results, the maximum operating frequency for the 16-bit processor and the 32-bit processor was 78.654 MHz and 139.438 MHz, respectively. The maximum combinational delay was 13.981 ns for the 16-bit processor and 7.028 ns for the 32-bit processor.

Using Verilog HDL coding, (Shawkat S. Khairullah, 2022) designed and implemented a 16-bit RISC processor with 5 pipeline stages that was simulated using Xilinx ISE Design Suite 14.7 tool. They synthesized the design on device Xilinx XC3S200FT256 FPGA chip. They showed the experimental and timing diagram results that indicated that the execution unit hardware design used 2% of Spartan – FPGA XC3S2000 area with a maximum speed of 56.8 MHz. They also showed that the data memory unit hardware design used 8% of the same FPGA area with a maximum speed of 67.32 MHz. Moreover, they showed that the instruction unit hardware design used 6% of the same FPGA area with a maximum speed of 106 MHz.

Using VHDL, (Sarah M. Al-sudany et al., 2021) designed and implemented a multicore RISC processor on FPGA. They used 32-bit MIPS processor with three main components: 32-bit data path, control unit, and hazard unit. They divided the single cycle MIPS system into five pipeline stages to create the pipeline MIPS processor. They also solved the data hazard, control hazard, and structural hazard in their design. They developed the MIPS using Xilinx ISE 14.7 design suite and successfully implemented it on Xilinx Virtex-6 XC6VLX240T-1FFG1156 FPGA. The multicore MIPS processor consumed 3.422 watt of power and had an operating frequency of 136.444 MHz.

Table 2.1: Implementation and design of various processor from previous works

Author	Description
S. P. Pitpurkar et al., 2015	• Implement a design of 32-bit RISC based MIPS processor using VHDL coding • RISC has more advantages, such as faster speed, simplified structure, and easier to be implemented as compared to CISC • Xilinx 13.li ISE Simulator was used to the design, synthesis and simulation • Achieved combinational delay of 0.758ns and maximum operating frequency of 1.350 GHz
Mohit N. Topiwala et al., 2014	• Implemented a 32-bit MIPS based processor using Cadence • Power is the most important parameter for embedded and portable applications • Proposed a power reduction technique through by-passing pipeline stages that cause unnecessary switching activities. • Hazard detection unit and Data forwarding unit were designed for efficient implementation of the pipeline

	• Implemented using Verilog-HDL and synthesized using Cadence RTL complier using typical libraries of TSMC 0.18 μm technology
Shawkat S. Khairullah, 2022	• Presented hardware realization of 5 pipeline stages of a 16-bit RISC processor • Processor is simulated using Xilinx ISE Design Suite 14.7 tool • Synthesis process of the proposed system is realized on device Xilinx XC3S200FT256 FPGA chip • Execution unit uses 2% of Spartan – FPGA XC3S2000 area with maximum allowable speed of 56.8 MHz. • Data memory unit uses 8% of Spartan – FPGA XC3S2000 area with maximum allowable speed of 67.32 MHz • Instruction unit uses 6% of the same FPGA ship area with maximum allowable speed of 106 MHz
Shofiqul Islam et al., 2006	• Designed a high speed-pipelined execution unit of 32-bit RISC processor • Data selection for the computational blocks in Execution stage is performed one stage ahead

	• Dependency Resolver module were proposed to solve consecutive data dependent instruction in the pipeline • Basically, the module handles both stalling as well as data forwarding
Animesh Kulshreshtha et al., 2021	• Analysed behavioural model of 16-bit and 32-bit RISC processor and their independent instruction sets • 16-bit RISC processor was a non-pipeline Harvard architecture-based CPU • 32-bit RISC was a pipelined processor from MIPS architecture • Total power consumption for 32-bit processor was about 60% more than 16-bit processor due to higher operating frequency • 32-bit processor was 70% faster than 16-bit processor • Maximum operating frequency for 16-bit processor and 32-bit processor is 78.654 MHz and 139.438 MHz • Maximum combinational delay for 16-bit processor and 32-bit processor is 13.981 ns and 7.028 ns

Sarah M. Al-sudany et al., 2021	• Studied for multicore RISC processor implemented on FPGA • MIPS was developed using Xilinx ISE 14.7 design suite and were implemented successfully on Xilinx Virtex-6 XC6VLX240T-1FFG1156 FPGA • 32-bit MIPS processor was designed using VHDL with 3 main structures: 32-bit data path, control unit and hazard unit • Total power analysis of multicore MIPS processor is 3.422 watt, and the operating frequency is 136.444 MHz

Chapter 3: Methodology

3.1 Background Theories

3.1.1 Instruction Set Architecture

Instruction Set Architecture (ISA) is a term that refers to the set of instructions that a computer processor can execute. It is a fundamental aspect of computer architecture that determines the capabilities and limitations of a processor.

An ISA is composed of a set of instructions that define the operations that a processor can perform, as well as the format and meaning of each instruction. Each instruction typically includes an opcode, which specifies the operation to be performed, and one of more operands, which specify the data on which the operation is to be performed. Figure xxx shows the example of opcode and operand for a MOV instruction.

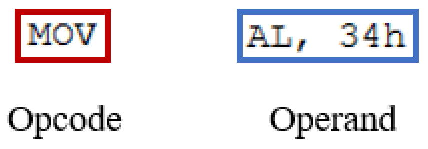


Figure 3.1 Opcode and Operand of a MOV instruction

From Figure 3.1, the opcode is the MOV instruction. The other parts are called the operands. Operands are manipulated by the opcode. In this example, the operands are the register named AL and the value 34 hex.

ISAs can be classified into 2 main categories: Reduces Instruction Set Computing (RISC) and Complex Instruction Set Computing (CISC). RISC ISAs have a smaller set of simple instructions that can be executed quickly, whereas CISC ISAs have a larger set of more complex instructions that can perform multiple operations in a single instruction. The RISC approach is generally favored in modern processors because it allows for faster execution times and simpler processor designs. However, CISC architectures are still used in certain specialized applications where the additional complexity is justified by the increased functionality.

One example of a CISC ISA is the x86 architecture, which is used in many personal computers and servers. The x86 ISA includes a large set of instructions which are quite complex and can perform multiple operations in a single instruction. For example, MOV instruction from CISC ISA can transfer data between two memory locations or between a register and a memory location in a single instruction. Besides, the x86 ISA also includes a variety of specialized instructions for performing common tasks such as string manipulation, input/output operations, and floating-point arithmetic. x86 ISA is known for its complexity and backward compatibility. However, this complexity can make it difficult to optimize for performance or energy efficiency. Hence, it is more challenging to write software that runs efficiently on different processor with different implementations of the ISA.

One example of a RISC ISA is the ARM architecture, which is used in a wide range of device including smartphones, tables, and embedded systems. The ARM ISA includes a relatively small set of simple instructions which can be executed faster. For example, the MOV instruction from RISC ISA only copies data from one register to another. Besides, ARM ISA also includes a variety of specialized instruction such as loading and storing multiple registers at once, performing conditional instructions, and performing arithmetic operations on multiple data types. The ARM ISA is known for its simplicity and energy efficiency. The simplicity of the ISA also makes it easier to optimize for performance and make it easier to write software on different processors with different implementations of the ISA.

3.1.2 5-Stage pipeline

With traditional single cycle data path, instructions are executed in a single clock cycle. The next instruction will need to wait for the previous instruction to be complete before it can be executed. Moreover, the execution time for each instruction is different. There are instructions that takes longer time to execute than the other. This might lead to wasted clock cycle and reduced performance of the processor. However, this issue can be solved by using pipelining.

Pipelining is a technique used in computer architecture to increase the overall performance of a processor. It involves breaking down the execution of a task into smaller stages and allowing these stages to overlap in time. With this, multiple instructions can be executed simultaneously and improve the throughput of the system.

The most common used pipeline technique in modern processor design is 5-Stage pipeline technique. As suggested by its name, the instruction execution cycle was divided into 5 different stages. Each stage performs a specific operation on the input data and passes the result to the next stage. The output of the first stage become the input of the second stage, and so on. Besides, every stage operates on a different part of the data, which allow multiple instructions to be in different stages of execution at the same time. The 5 stages of a typical pipeline are: fetch, decode, execute, memory, and writeback. The details for each stage were discussed below.

1. Fetch

   In this stage, the processor fetches the instruction from memory. Then, the instruction is loaded into the instruction cache, which is used to store the recently used instructions.

2. Decode

   In this stage, the processor decodes the fetched instruction to determine the operation it needs to perform. The instruction is analyzed to determine the type of operation, registers, and the locations of any operands.

3. Execute

   In this stage, the processor performs actual operation specified by the instruction. This involves arithmetic or logical operations, such as addition and subtraction. Besides, it also involves memory access or branching to a different part of the program.

4. Memory

   In this stage, the processor access memory to read or write the data obtained from previous stage. However, this stage is optional. Some instruction such as “add” does not involve memory access.

5. Writeback

In this stage, the results of the operation are written back to the appropriate register in the register file.

Figure 3.2 illustrates the overview of 5 stage pipeline.

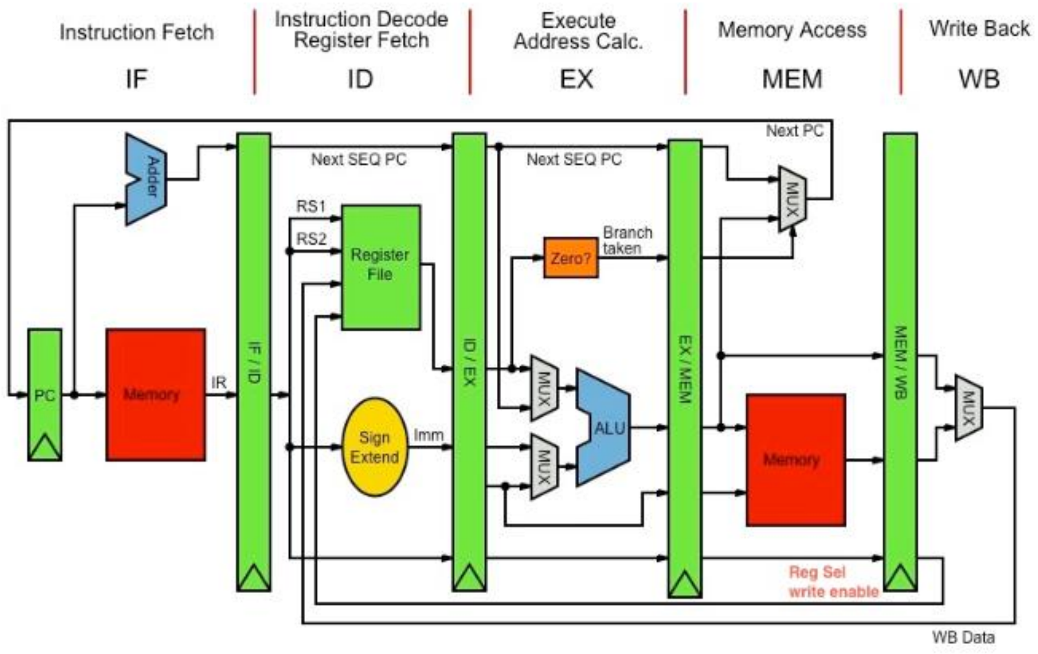


Figure 3.2: 5 Stage pipeline

3.1.3 ModelSim

ModelSim is a popular simulation and verification tool for digital circuits and system. It is widely used by engineers and designers in the electronics industry to validate and debug their designs before they are implemented in hardware.

ModelSim provides a powerful set of features for designing, simulating, and verifying digital circuits and system. For simulation, ModelSim supports both Verilog and VHDL languages. It can simulate all levels of abstraction, from the gate level to the behavioral level. For verification, ModelSim supports functional and timing simulation, as well as assertion-based verification. It can also be integrated with other verification tools like Questa and UVM for more comprehensive verification. For design, ModelSim supports the creation of design hierarchy, which allows designers to organize their designs into logical blocks. Besides, it also provides design checking features like linting and syntax checking.

There are several benefits of using ModelSim. First and foremost, ModelSim helps to improve design quality. It helps to identify and fix design errors early in the design cycle, which improves the quality of the final product. Next, ModelSim reduces the need for expensive hardware prototyping by providing a virtual environment for design validation and verification. This help to reduce the cost needed in design. Furthermore, ModelSim also supports industry-standard languages and interface, which promotes standardization and interoperability in the electronic industry.

In the project, ModelSim will be the main software used to design and simulation the RISC-V processor. Various module such as instruction memory, adder, register, data memory, alu and alu control will be coded in ModelSim using Verilog. The functionality for each element will also be tested and verified by simulation using ModelSim. Finally, each element will be integrated in a main module to form the top design of a 32-bit 5-stage pipeline RISC-V processor.

Figure 3.3 shows the flow chart of the project.

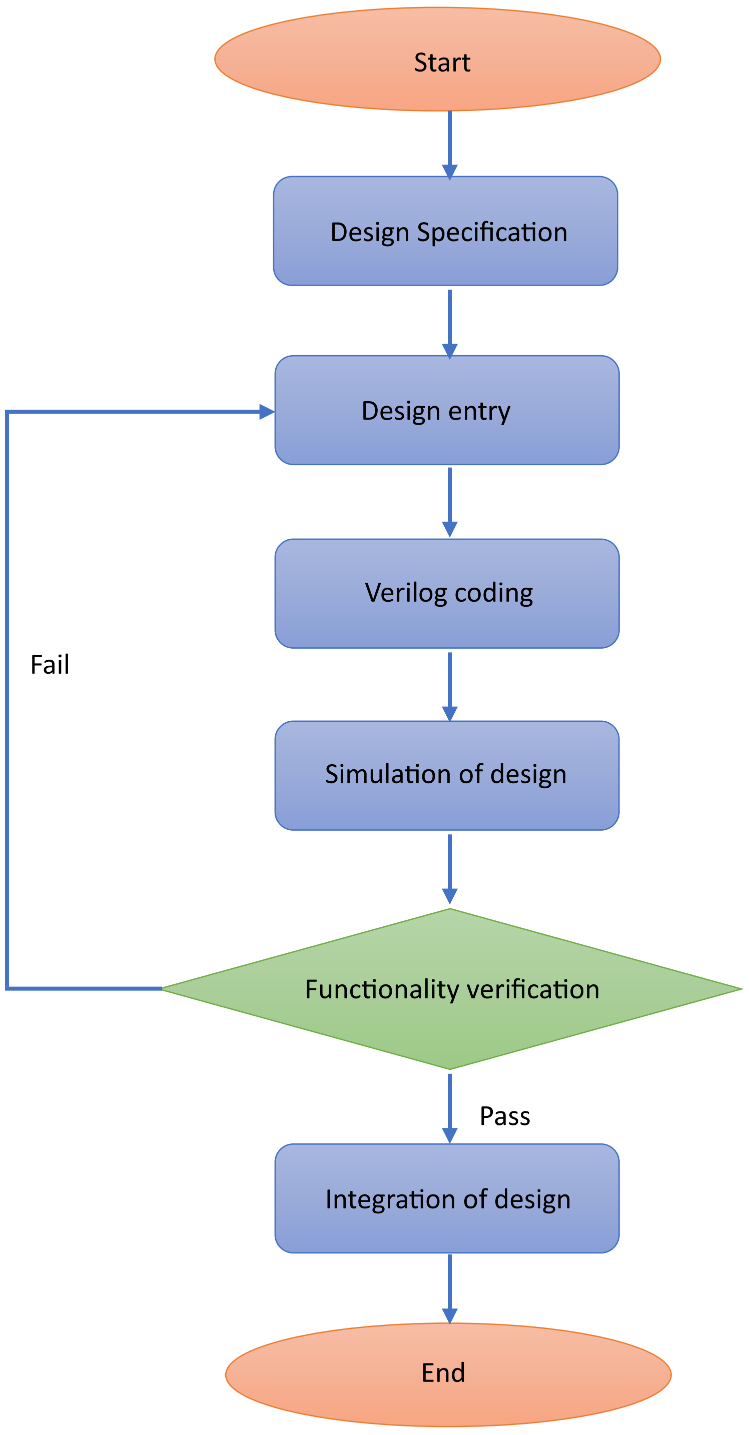


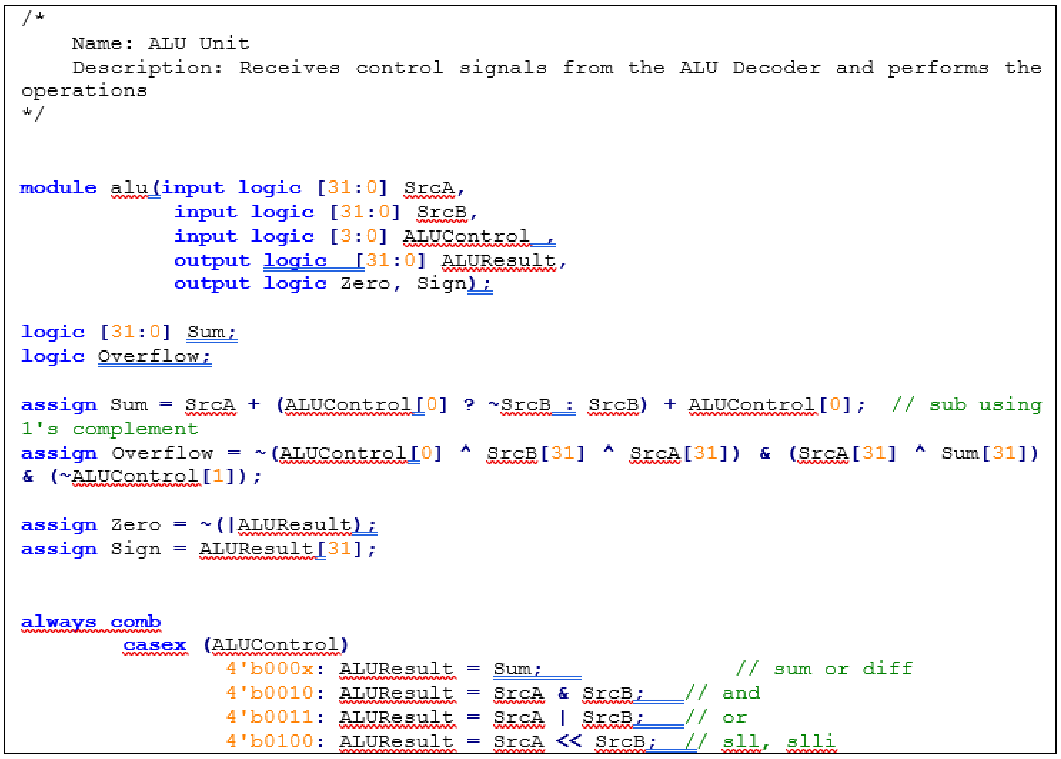
Figure 3.3: Flow chard of the project

3.2 Implementation of Design

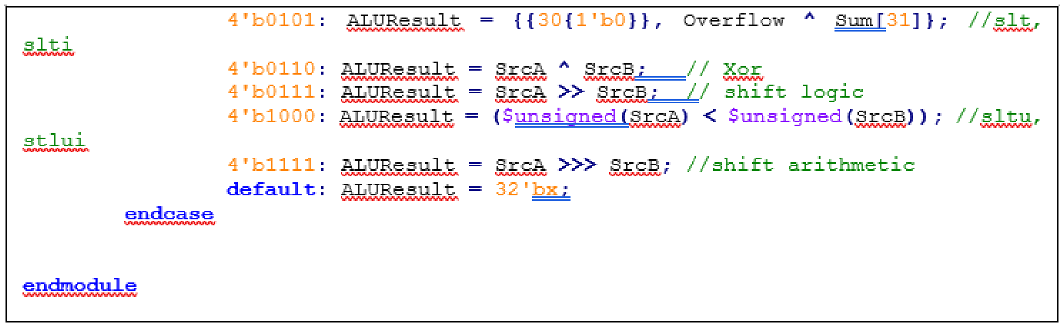
The processor consists of *alu, aludec, maindec, imem, dmem, IF_ID, ID_IEx, IEx_IMem, Imem_IW, pc_mux, reg_file, forwardMuxA, and forwardMuxB*. Each component of the datapath will be discussed in this section.

3.2.1 Arithmetic Logic Unit (*alu*)

Arithmetic Logic Unit (*alu*) is a fundamental component of a CPU. It is responsible for performing arithmetic and logical operations on binary data. It reads the data from pipeline register *ID_IEx* as an input and perform various arithmetic operation based on the signals from *aludec*. The output is stored as *ALUResults*. In this RISC-V processor design, 9 instructions are implemented in the ALU. The implementation of *alu* in Verilog is shown in Figure 3.4.



```
/*
    Name: ALU Unit
    Description: Receives control signals from the ALU Decoder and performs the
operations
*/


module alu(input logic [31:0] SrcA,
            input logic [31:0] SrcB,
            input logic [3:0] ALUControl ,
            output logic   [31:0] ALUResult,
            output logic Zero, Sign);

logic [31:0] Sum;
logic Overflow;

assign Sum = SrcA + (ALUControl[0] ? ~SrcB : SrcB) + ALUControl[0];  // sub using
1's complement
assign Overflow = ~(ALUControl[0] ^ SrcB[31] ^ SrcA[31]) & (SrcA[31] ^ Sum[31])
& (~ALUControl[1]);

assign Zero = ~(|ALUResult);
assign Sign = ALUResult[31];


always_comb
        casex (ALUControl)
                4'b000x: ALUResult = Sum;              // sum or diff
                4'b0010: ALUResult = SrcA & SrcB;    // and
                4'b0011: ALUResult = SrcA | SrcB;    // or
                4'b0100: ALUResult = SrcA << SrcB;   // sll, slli
```



```
                4'b0101:  ALUResult  =  {{30{1'b0}},  Overflow  ^  Sum[31]};  //slt,
slti
                4'b0110: ALUResult = SrcA ^ SrcB;    // Xor
                4'b0111: ALUResult = SrcA >> SrcB;  // shift logic
                4'b1000: ALUResult = ($unsigned(SrcA) < $unsigned(SrcB)); //sltu,
stlui
                4'b1111: ALUResult = SrcA >>> SrcB; //shift arithmetic
                default: ALUResult = 32'bx;
        endcase


endmodule
```

Figure 3.4: Implementation of *alu* in Verilog code

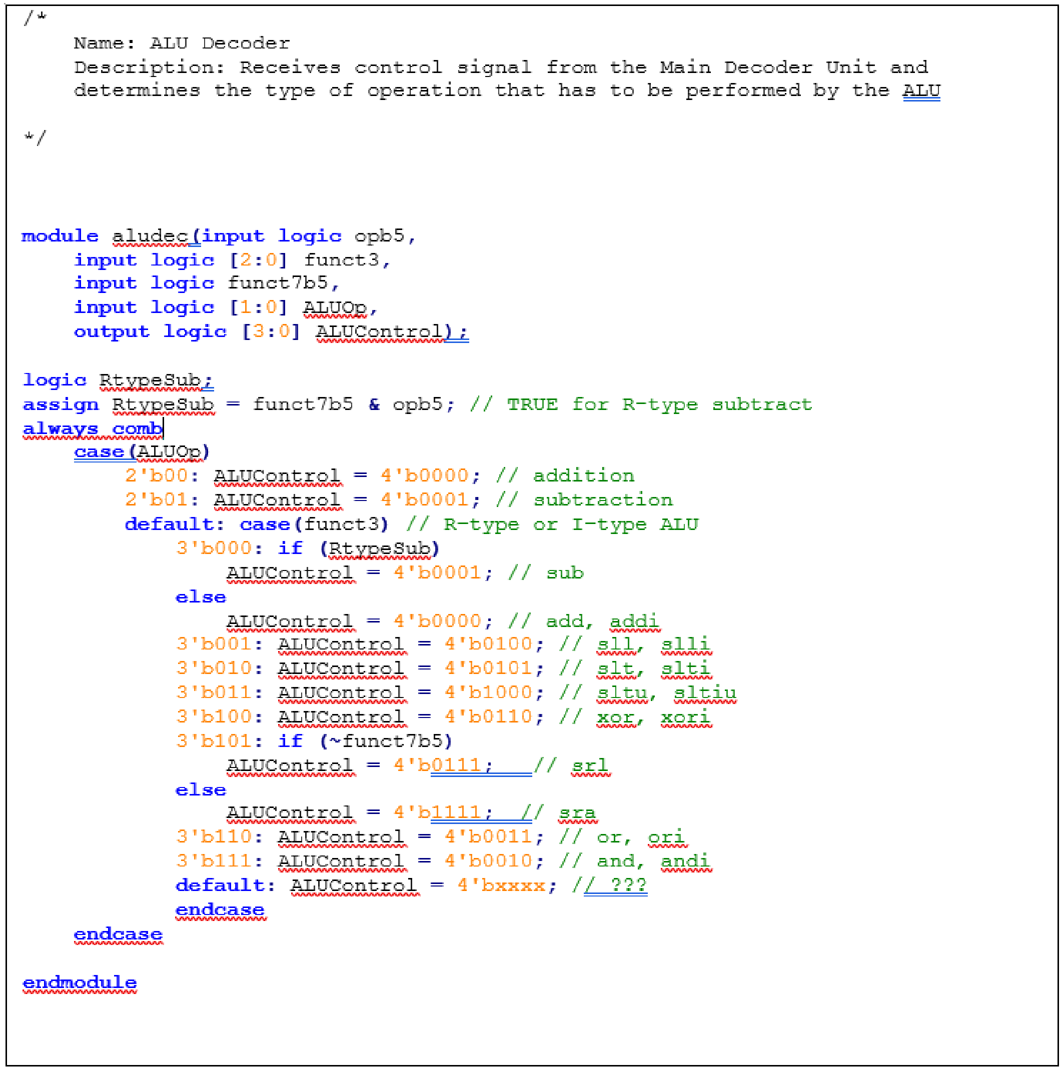
Table 3.1 shows the base integer instructions for RV321.

Table 3.1: Base Integer Instructions for RV321

Category	Mnemonic	Description
Arithmetic		
ADD	rd, rs1, rs2	rd 🡐 rs1 + rs2
SUB	rd, rs1, rs2	rd 🡐 rs1 - rs2
Logical		
XOR	rd, rs1, rs2	rd 🡐 rs1 ^ rs2
AND	rd, rs1, rs2	rd 🡐 rs1 & rs2
OR	rd, rs1, rs2	rd 🡐 rs1 | rs2
Shifts		
SHL	rd, rs1, rs2	rd 🡐 rs1 << rs2
SHR	rd, rs1, rs2	rd 🡐 rs1 >> rs2
Compare		
SLT	rd, rs1, rs2	rd 🡐 rs1 < rs2 ? 1 : 0
SLTU	rd, rs1, rs2	rd 🡐 rs1 < rs2 ? 1 : 0

3.2.2 ALU Decoder (*aludec*)

Arithmetic Logic Unit Decoder (*aludec*) is used to decode the instructions. It received the signal from the Main Decoder Unit (*maindec*) and determine the type of operation that had to be performed by the *alu*. It combined all 4 inputs from *ALUOp, funct3, funct7b5* and *opb5* to decode the instruction. *funct7b5* is referring to *instruction[31:25]*, *funct3* is referring *instruction[14:12]*, whereas *opb5* is referring to *instruction[0:6]* in RISC-V instruction format. *RTypeSub* is obtained by performing *AND* operation on *funct7b5* and *opb5*. This is used to differentiate the R-type *ADD* and *SUB* instruction. The implementation of *aludec* is shown in Figure 4.12.



```
/*
    Name: ALU Decoder
    Description: Receives control signal from the Main Decoder Unit and
    determines the type of operation that has to be performed by the ALU

*/



module aludec(input logic opb5,
    input logic [2:0] funct3,
    input logic funct7b5,
    input logic [1:0] ALUOp,
    output logic [3:0] ALUControl);

logic RtypeSub;
assign RtypeSub = funct7b5 & opb5; // TRUE for R-type subtract
always_comb
    case(ALUOp)
        2'b00: ALUControl = 4'b0000; // addition
        2'b01: ALUControl = 4'b0001; // subtraction
        default: case(funct3) // R-type or I-type ALU
            3'b000: if (RtypeSub)
                ALUControl = 4'b0001; // sub
            else
                ALUControl = 4'b0000; // add, addi
            3'b001: ALUControl = 4'b0100; // sll, slli
            3'b010: ALUControl = 4'b0101; // slt, slti
            3'b011: ALUControl = 4'b1000; // sltu, sltiu
            3'b100: ALUControl = 4'b0110; // xor, xori
            3'b101: if (~funct7b5)
                ALUControl = 4'b0111;    // srl
            else
                ALUControl = 4'b1111;  // sra
            3'b110: ALUControl = 4'b0011; // or, ori
            3'b111: ALUControl = 4'b0010; // and, andi
            default: ALUControl = 4'bxxxx; // ???
            endcase
    endcase

endmodule
```

Figure 3.5: Implementation of *aludec* in Verilog code

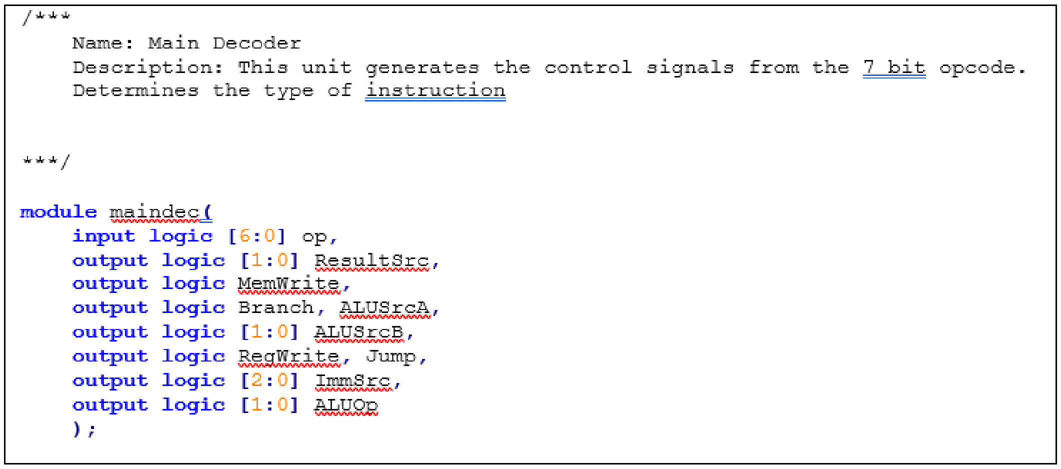
The *aludec* output corresponds to the case selected as shown in Table 3.2.

Table 3.2: ALU decoder output

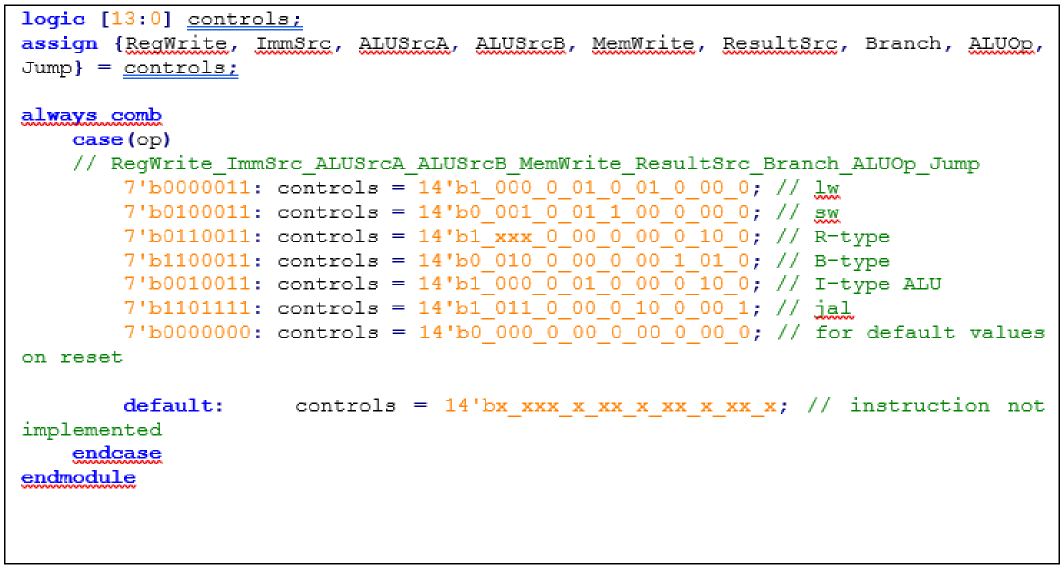
ALUOp	*RTypeSub*	*funct3*	Alu decode	Operation
2’b10	0	3’b000	4’b0000	AND
2’b10	1	3’b000	4’b0001	SUB
2’b10	0	3’b111	4’b0010	AND
2’b10	0	3’b110	4’b0011	OR
2’b10	0	3’b001	4’b0100	SLLI
2’b10	0	3’b011	4’b0101	SLTI
2’b10	0	3’b100	4’b0110	XOR
2’b10	0	3’b101	4’b0111	SHR
2’b10	0	3’b101	4’b1000	SLTU
2’b10	0	3’b101	4’b1111	SHL

3.2.3 Main Decoder (*maindec*)

Main Decoder (*maindec*) is used to generate the control signals from the 7 bits opcode (*instruction[6:0]*) to determine the types of instruction. The control signals are *RegWrite, ImmSrc, ALUSrcA, ALUSrcB, MemWrite, ResultSrc, Branch, ALUOp,* and *Jump.* Each of these control signals control the multiplexer for making decisions in the datapath to allow the data flow accordingly to the instructions. The implementation of *maindec* is shown in Figure 3.6.



```
/***
    Name: Main Decoder
    Description: This unit generates the control signals from the 7 bit opcode.
    Determines the type of instruction


***/

module maindec(
    input logic [6:0] op,
    output logic [1:0] ResultSrc,
    output logic MemWrite,
    output logic Branch, ALUSrcA,
    output logic [1:0] ALUSrcB,
    output logic RegWrite, Jump,
    output logic [2:0] ImmSrc,
    output logic [1:0] ALUOp
    );
```



```
logic [13:0] controls;
assign {RegWrite, ImmSrc, ALUSrcA, ALUSrcB, MemWrite, ResultSrc, Branch, ALUOp,
Jump} = controls;

always comb
    case(op)
    // RegWrite_ImmSrc_ALUSrcA_ALUSrcB_MemWrite_ResultSrc_Branch_ALUOp_Jump
        7'b0000011: controls = 14'b1_000_0_01_0_01_0_00_0; // lw
        7'b0100011: controls = 14'b0_001_0_01_1_00_0_00_0; // sw
        7'b0110011: controls = 14'b1_xxx_0_00_0_00_0_10_0; // R-type
        7'b1100011: controls = 14'b0_010_0_00_0_00_1_01_0; // B-type
        7'b0010011: controls = 14'b1_000_0_01_0_00_0_10_0; // I-type ALU
        7'b1101111: controls = 14'b1_011_0_00_0_10_0_00_1; // jal
        7'b0000000: controls = 14'b0_000_0_00_0_00_0_00_0; // for default values
on reset

        default:     controls = 14'bx_xxx_x_xx_x_xx_x_xx_x; // instruction not
implemented
    endcase
endmodule
```

Figure 3.6: Implementation of *maindec* in Verilog code

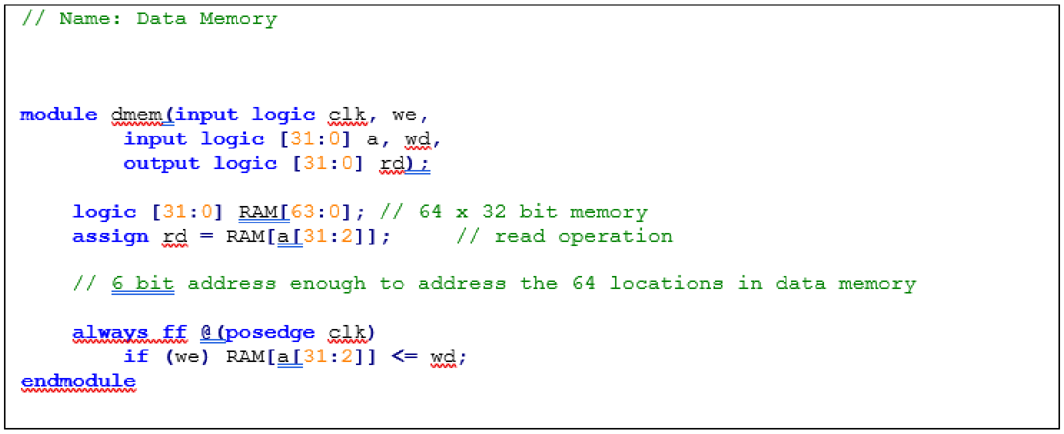
The on and off for these signals are based on the types of instructions tabulated in Table 3.3.

Table 3.3: Main Decoder control signal and types of instructions

Control Signal	Instruction					
	lw	sw	R-Type	B-Type	I-Type	jal
RegWrite	1	0	1	0	1	1
ImmSrc	000	001	xxx	010	000	011
ALUSrcA	0	0	0	0	0	0
ALUSrcB	01	01	00	00	01	00
MemWrite	0	1	0	0	00	0
ResultSrc	01	00	00	00	000	10
Branch	0	0	0	1	0	0
ALUOp	00	00	10	01	10	00
Jump	0	0	0	0	0	1

3.2.4 Data Memory (*dmem*)

In computer architecture, data memory is a component of a computer system that is responsible for storing and retrieving data. Data memory is typically used to store data that is actively being processed by *alu*. Usually there are 3 inputs in this module, which are *write_enable, data_address*, and *write_data.* The module takes the memory address from the results of *alu (data_address)* and data from register file (*write_data*). Write enable *(write_enable)* is used to control the write permission of the data to the data memory. The implementation of *dmem* in Verilog code is shown in Figure 3.7.

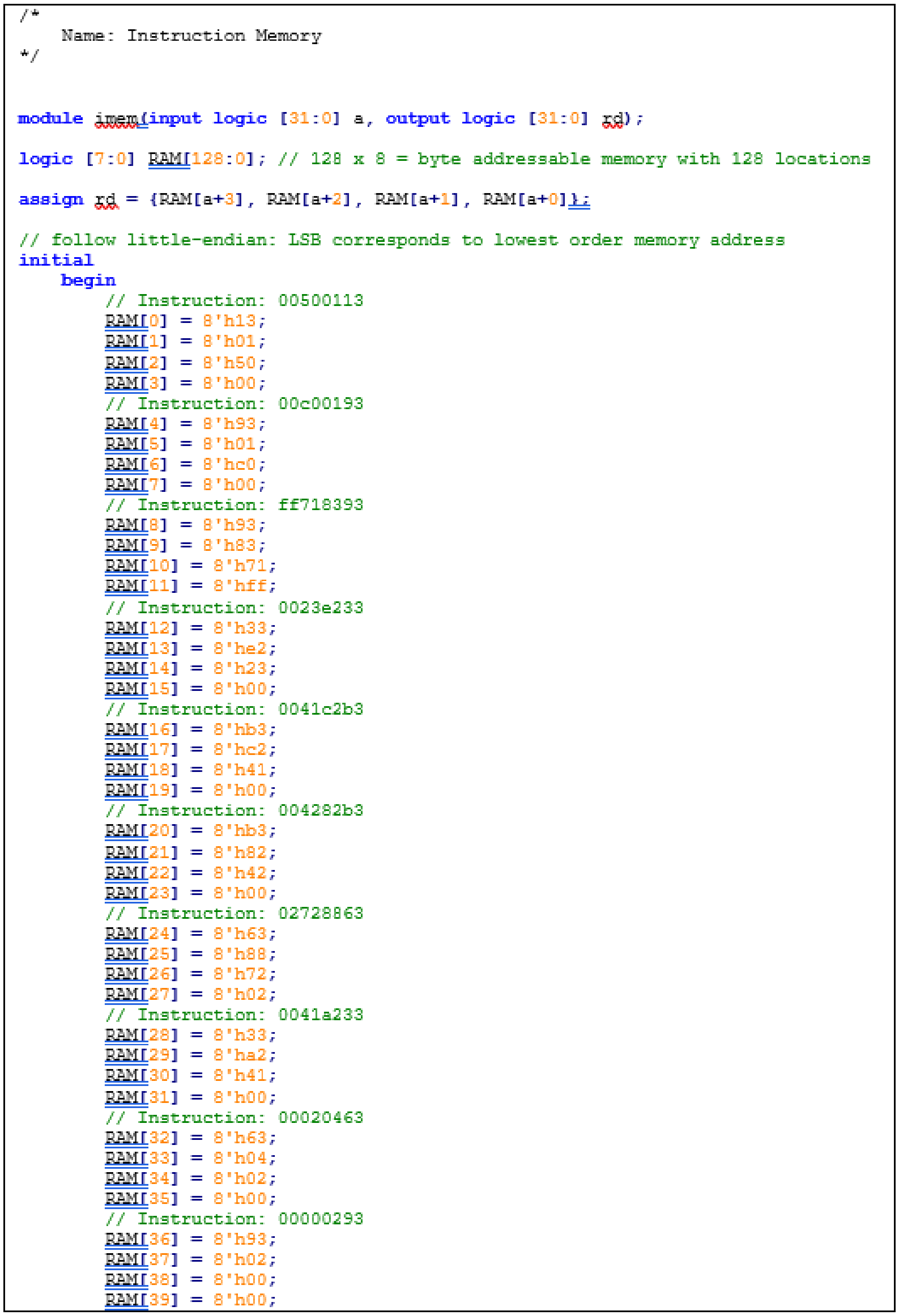


```
// Name: Data Memory


module dmem(input logic clk, we,
        input logic [31:0] a, wd,
        output logic [31:0] rd);

    logic [31:0] RAM[63:0]; // 64 x 32 bit memory
    assign rd = RAM[a[31:2]];     // read operation

    // 6 bit address enough to address the 64 locations in data memory

    always_ff @(posedge clk)
        if (we) RAM[a[31:2]] <= wd;
endmodule
```

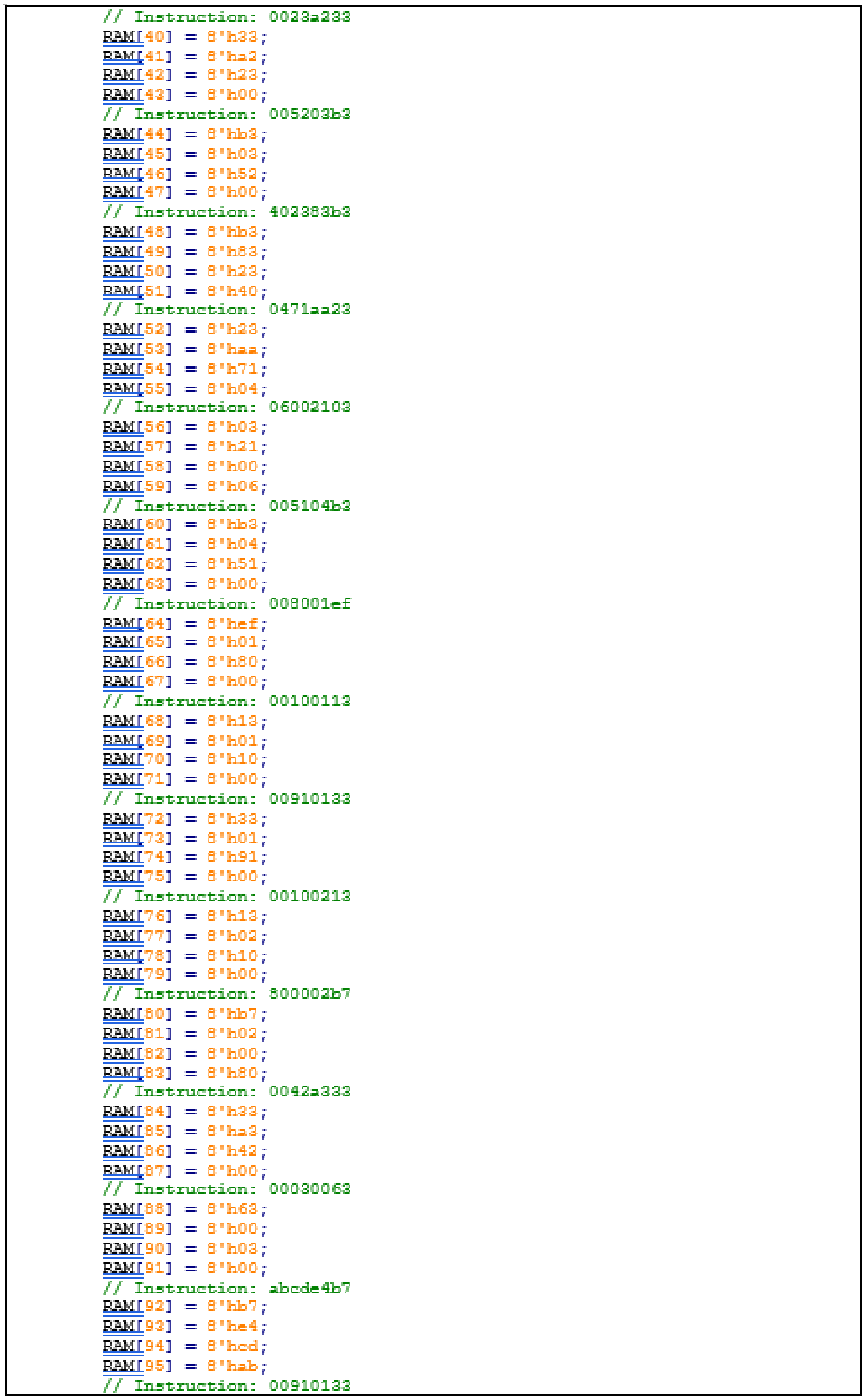
Figure 3.7: Implementation of *dmem* in Verilog code

3.2.5 Instruction Memory *(imem)*

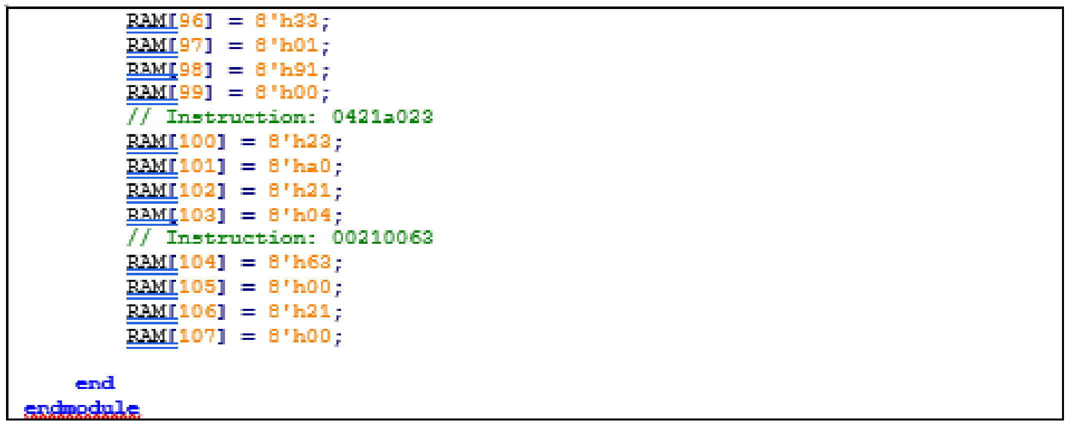
In computer architecture, instruction memory is a component of a computer system that stores the instruction of a program. It is responsible for holding the sequence of instruction that the CPU fetches, decodes, and executes during the program execution. In this module, 32-bit instruction set is generated and stored in the ram array. The instruction to be fetch is based on the program counter fetch (*PCF*) input. As each instruction is 4 bytes, the value of PCF will be incremented by 4 to fetch the next instruction. The implementation of *imem* in Verilog code is shown in Figure 3.8.



```
/*
    Name: Instruction Memory
*/


module imem(input logic [31:0] a, output logic [31:0] rd);

logic [7:0] RAM[128:0]; // 128 x 8 = byte addressable memory with 128 locations

assign rd = {RAM[a+3], RAM[a+2], RAM[a+1], RAM[a+0]};

// follow little-endian: LSB corresponds to lowest order memory address
initial
    begin
        // Instruction: 00500113
        RAM[0] = 8'h13;
        RAM[1] = 8'h01;
        RAM[2] = 8'h50;
        RAM[3] = 8'h00;
        // Instruction: 00c00193
        RAM[4] = 8'h93;
        RAM[5] = 8'h01;
        RAM[6] = 8'hc0;
        RAM[7] = 8'h00;
        // Instruction: ff718393
        RAM[8] = 8'h93;
        RAM[9] = 8'h83;
        RAM[10] = 8'h71;
        RAM[11] = 8'hff;
        // Instruction: 0023e233
        RAM[12] = 8'h33;
        RAM[13] = 8'he2;
        RAM[14] = 8'h23;
        RAM[15] = 8'h00;
        // Instruction: 0041c2b3
        RAM[16] = 8'hb3;
        RAM[17] = 8'hc2;
        RAM[18] = 8'h41;
        RAM[19] = 8'h00;
        // Instruction: 004282b3
        RAM[20] = 8'hb3;
        RAM[21] = 8'h82;
        RAM[22] = 8'h42;
        RAM[23] = 8'h00;
        // Instruction: 02728863
        RAM[24] = 8'h63;
        RAM[25] = 8'h88;
        RAM[26] = 8'h72;
        RAM[27] = 8'h02;
        // Instruction: 0041a233
        RAM[28] = 8'h33;
        RAM[29] = 8'ha2;
        RAM[30] = 8'h41;
        RAM[31] = 8'h00;
        // Instruction: 00020463
        RAM[32] = 8'h63;
        RAM[33] = 8'h04;
        RAM[34] = 8'h02;
        RAM[35] = 8'h00;
        // Instruction: 00000293
        RAM[36] = 8'h93;
        RAM[37] = 8'h02;
        RAM[38] = 8'h00;
        RAM[39] = 8'h00;
```



```
// Instruction: 0023a233
RAM[40] = 8'h33;
RAM[41] = 8'ha2;
RAM[42] = 8'h23;
RAM[43] = 8'h00;
// Instruction: 005203b3
RAM[44] = 8'hb3;
RAM[45] = 8'h03;
RAM[46] = 8'h52;
RAM[47] = 8'h00;
// Instruction: 402383b3
RAM[48] = 8'hb3;
RAM[49] = 8'h83;
RAM[50] = 8'h23;
RAM[51] = 8'h40;
// Instruction: 0471aa23
RAM[52] = 8'h23;
RAM[53] = 8'haa;
RAM[54] = 8'h71;
RAM[55] = 8'h04;
// Instruction: 06002103
RAM[56] = 8'h03;
RAM[57] = 8'h21;
RAM[58] = 8'h00;
RAM[59] = 8'h06;
// Instruction: 005104b3
RAM[60] = 8'hb3;
RAM[61] = 8'h04;
RAM[62] = 8'h51;
RAM[63] = 8'h00;
// Instruction: 008001ef
RAM[64] = 8'hef;
RAM[65] = 8'h01;
RAM[66] = 8'h80;
RAM[67] = 8'h00;
// Instruction: 00100113
RAM[68] = 8'h13;
RAM[69] = 8'h01;
RAM[70] = 8'h10;
RAM[71] = 8'h00;
// Instruction: 00910133
RAM[72] = 8'h33;
RAM[73] = 8'h01;
RAM[74] = 8'h91;
RAM[75] = 8'h00;
// Instruction: 00100213
RAM[76] = 8'h13;
RAM[77] = 8'h02;
RAM[78] = 8'h10;
RAM[79] = 8'h00;
// Instruction: 800002b7
RAM[80] = 8'hb7;
RAM[81] = 8'h02;
RAM[82] = 8'h00;
RAM[83] = 8'h80;
// Instruction: 0042a333
RAM[84] = 8'h33;
RAM[85] = 8'ha3;
RAM[86] = 8'h42;
RAM[87] = 8'h00;
// Instruction: 00030063
RAM[88] = 8'h63;
RAM[89] = 8'h00;
RAM[90] = 8'h03;
RAM[91] = 8'h00;
// Instruction: abcde4b7
RAM[92] = 8'hb7;
RAM[93] = 8'he4;
RAM[94] = 8'hcd;
RAM[95] = 8'hab;
// Instruction: 00910133
```



```
        RAM[96] = 8'h33;
        RAM[97] = 8'h01;
        RAM[98] = 8'h91;
        RAM[99] = 8'h00;
        // Instruction: 0421a023
        RAM[100] = 8'h23;
        RAM[101] = 8'ha0;
        RAM[102] = 8'h21;
        RAM[103] = 8'h04;
        // Instruction: 00210063
        RAM[104] = 8'h63;
        RAM[105] = 8'h00;
        RAM[106] = 8'h21;
        RAM[107] = 8'h00;

    end
endmodule
```

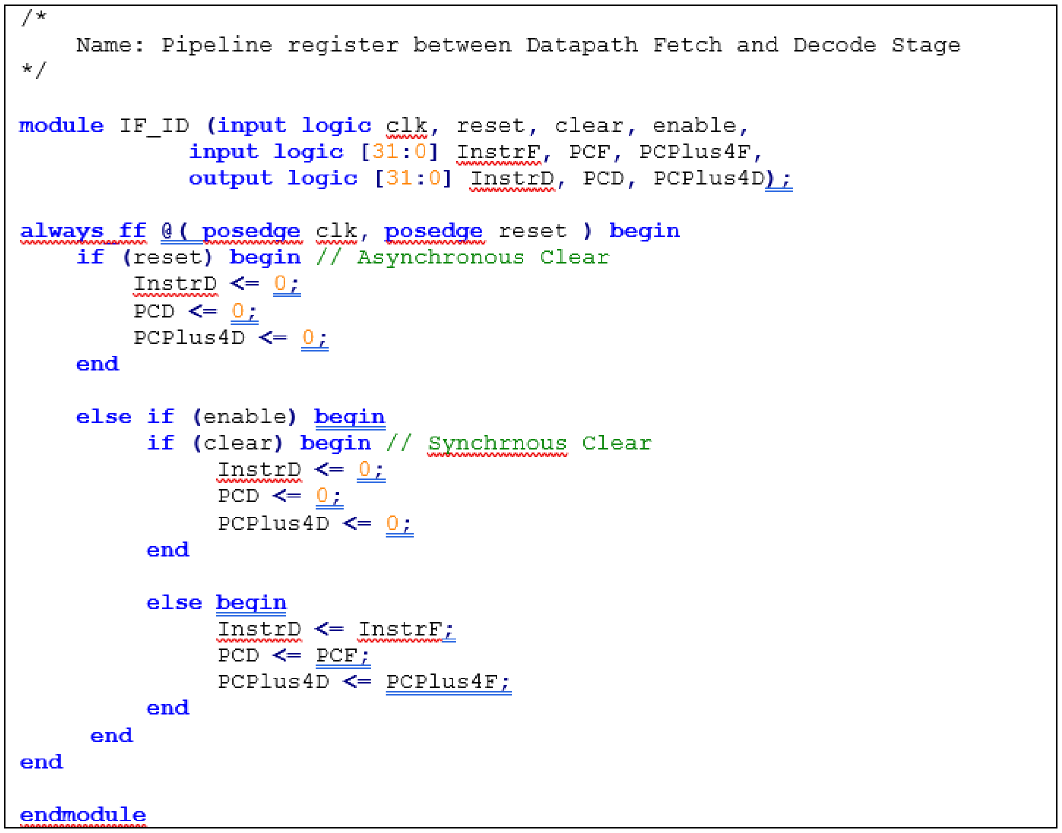
Figure 3.8: Implementation of *imem* in Verilog code

3.2.6 Pipeline Register

In computer architecture, pipeline register is a temporary storage element used un a processor's pipeline to hold data between different stages of instruction execution process. It serves as a synchronization point between adjacent stages, allowing instruction to flow through the pipeline in a controlled manner. In 5 stages pipeline, there are 4 pipeline registers namely *IF_ID, ID_IEx, IEx_IMem, and IMem_IW*. The registers are named for the two stages separated by that register. For example, the first pipeline register is *IF_ID* because it separates the instruction fetch and instruction decode stages.

3.2.6.1 *IF_ID*

IF_ID register as it names called, it separates the instruction fetch and instruction decode stages. It used to store data such as instruction fetch from instruction memory and ready to be released to decode stage on the next clock cycle. Besides, the current *PC* and next incremented *PC* address *(PCPlus4F*) is also saved in the *IF_ID* register in case it needed later for an instruction, such as *beq*. The implementation of *IF_ ID* register in Verilog code is shown in Figure 3.9.

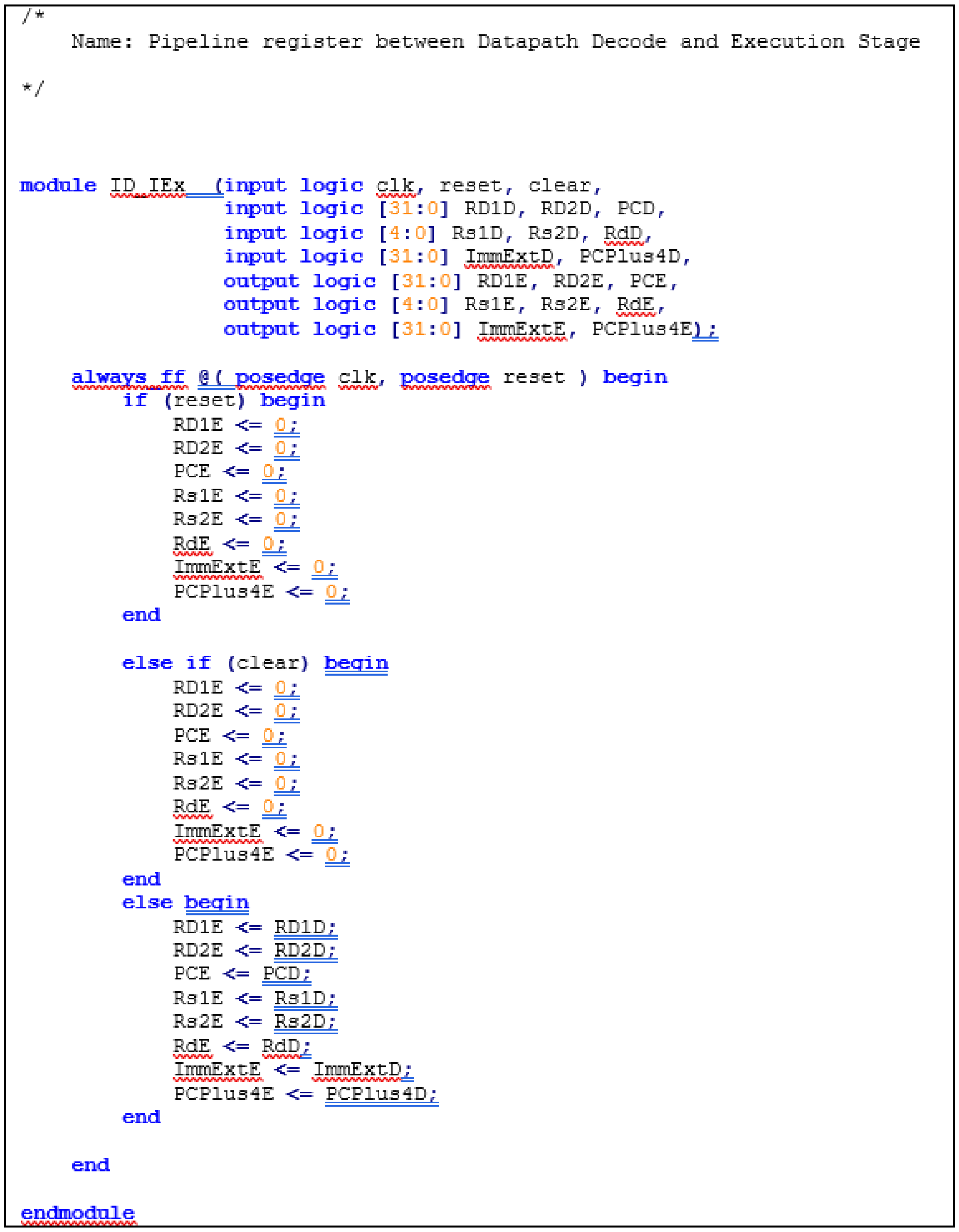


```
/*
    Name: Pipeline register between Datapath Fetch and Decode Stage
*/

module IF_ID (input logic clk, reset, clear, enable,
            input logic [31:0] InstrF, PCF, PCPlus4F,
            output logic [31:0] InstrD, PCD, PCPlus4D);

always_ff @( posedge clk, posedge reset ) begin
    if (reset) begin // Asynchronous Clear
        InstrD <= 0;
        PCD <= 0;
        PCPlus4D <= 0;
    end

    else if (enable) begin
         if (clear) begin // Synchrnous Clear
              InstrD <= 0;
              PCD <= 0;
              PCPlus4D <= 0;
         end

         else begin
              InstrD <= InstrF;
              PCD <= PCF;
              PCPlus4D <= PCPlus4F;
         end
     end
end

endmodule
```

Figure 3.9: Implementation of *IF/ID* register in Verilog code

3.2.6.2 *ID_IEx*

ID_IEx register as it names called, it separates the instruction decode and execute stages. It used to store information such as read data *(RD1, RD2)* from the register file and extended immediate value *(ImmExt)*. Besides, it carries forward the data of *PC* and *PCPlus4F* from *IF_ID* register. Instruction[11:7] *(rd)*, Instruction[19:15] *(rs1)*, and Instruction[24:20] *(rs2)* will also be stored to *ID_IEx* register and send to *Hazard Unit* in execute stage for hazard handling. The implementation if *ID_IEx* in Verilog code is shown in Figure 3.10.

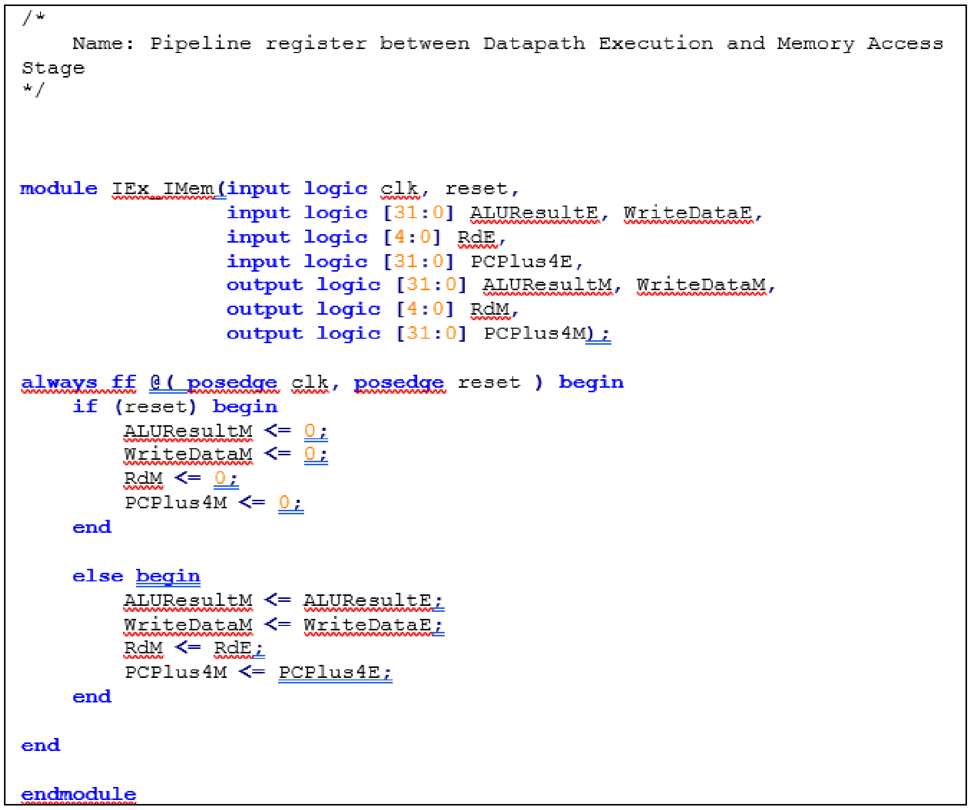


```
/*
    Name: Pipeline register between Datapath Decode and Execution Stage

*/



module ID_IEx  (input logic clk, reset, clear,
                input logic [31:0] RD1D, RD2D, PCD,
                input logic [4:0] Rs1D, Rs2D, RdD,
                input logic [31:0] ImmExtD, PCPlus4D,
                output logic [31:0] RD1E, RD2E, PCE,
                output logic [4:0] Rs1E, Rs2E, RdE,
                output logic [31:0] ImmExtE, PCPlus4E);

    always_ff @( posedge clk, posedge reset ) begin
        if (reset) begin
            RD1E <= 0;
            RD2E <= 0;
            PCE <= 0;
            Rs1E <= 0;
            Rs2E <= 0;
            RdE <= 0;
            ImmExtE <= 0;
            PCPlus4E <= 0;
        end

        else if (clear) begin
            RD1E <= 0;
            RD2E <= 0;
            PCE <= 0;
            Rs1E <= 0;
            Rs2E <= 0;
            RdE <= 0;
            ImmExtE <= 0;
            PCPlus4E <= 0;
        end
        else begin
            RD1E <= RD1D;
            RD2E <= RD2D;
            PCE <= PCD;
            Rs1E <= Rs1D;
            Rs2E <= Rs2D;
            RdE <= RdD;
            ImmExtE <= ImmExtD;
            PCPlus4E <= PCPlus4D;
        end

    end

endmodule
```

Figure 3.10: Implementation of *ID/IEx* in Verilog code

3.2.6.3 *IEx_IMem*

IEx_IMem register as it names called, it separates the execute and memory stages. It is used to store the ALU results *(ALUResult)* and write data *(Writedata)*. At the same time, Instruction[11:7] *(rd)* and *PCPlus4F* are also carried forward from previous pipeline registers and stored in *IEx_IMem* register. The implementation of *IEx_IMem* in Verilog code is shown in Figure 3.11.

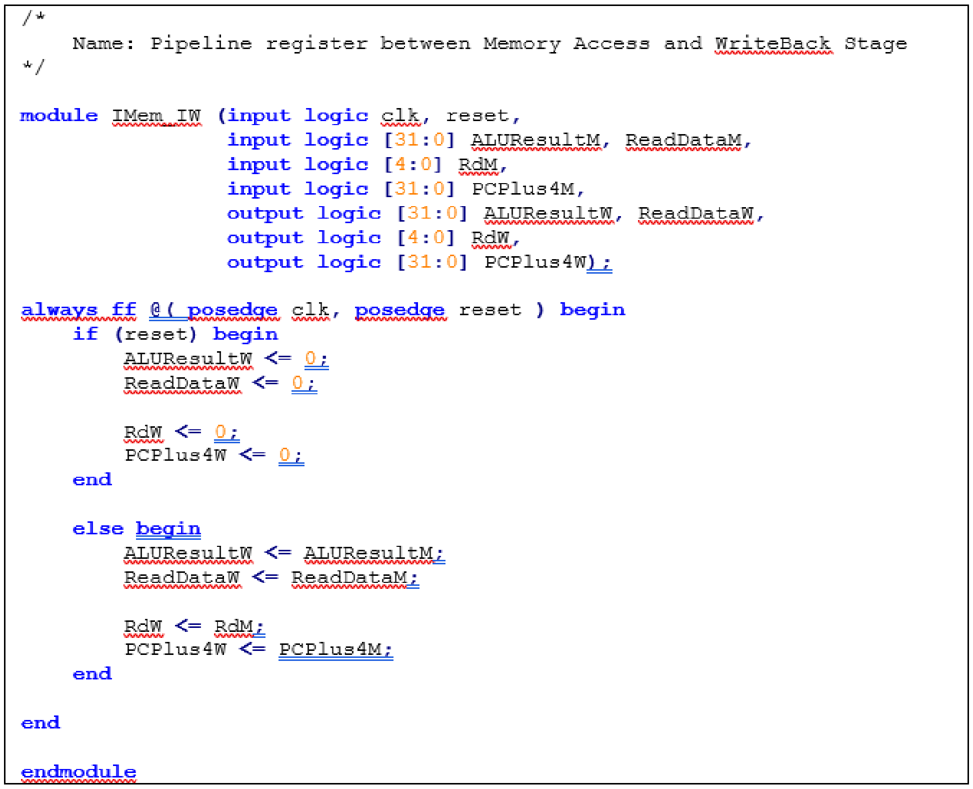


```
/*
    Name: Pipeline register between Datapath Execution and Memory Access
Stage
*/


module IEx_IMem(input logic clk, reset,
                input logic [31:0] ALUResultE, WriteDataE,
                input logic [4:0] RdE,
                input logic [31:0] PCPlus4E,
                output logic [31:0] ALUResultM, WriteDataM,
                output logic [4:0] RdM,
                output logic [31:0] PCPlus4M);

always_ff @( posedge clk, posedge reset ) begin
    if (reset) begin
        ALUResultM <= 0;
        WriteDataM <= 0;
        RdM <= 0;
        PCPlus4M <= 0;
    end

    else begin
        ALUResultM <= ALUResultE;
        WriteDataM <= WriteDataE;
        RdM <= RdE;
        PCPlus4M <= PCPlus4E;
    end

end

endmodule
```

Figure 3.11: Implementation of *IEx/IMem* register in Verilog code

3.2.6.4 *IMem_IW*

IMem_IW register as it names called, it separates the memory and writeback stages. It used to store ALU results *(ALUResult)* and read data (*ReadData)* from data memory. Instruction[11:7] *(rd)* and *PCPlus4F* are also carried forward from previous pipeline registers and store in *IMem_IW* register. The implementation of *IMem_IW* in Verilog code is shown in Figure 3.12.

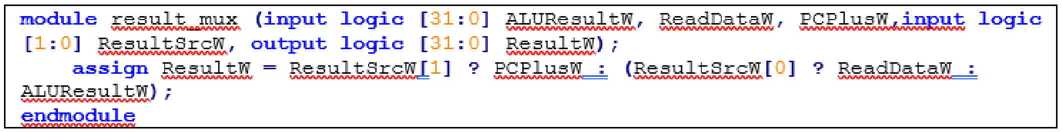


```
/*
    Name: Pipeline register between Memory Access and WriteBack Stage
*/

module IMem_IW (input logic clk, reset,
                input logic [31:0] ALUResultM, ReadDataM,
                input logic [4:0] RdM,
                input logic [31:0] PCPlus4M,
                output logic [31:0] ALUResultW, ReadDataW,
                output logic [4:0] RdW,
                output logic [31:0] PCPlus4W);

always_ff @( posedge clk, posedge reset ) begin
    if (reset) begin
        ALUResultW <= 0;
        ReadDataW <= 0;

        RdW <= 0;
        PCPlus4W <= 0;
    end

    else begin
        ALUResultW <= ALUResultM;
        ReadDataW <= ReadDataM;

        RdW <= RdM;
        PCPlus4W <= PCPlus4M;
    end

end

endmodule
```

Figure 3.12: Implementation of *IMem/IW* register in Verilog code

3.2.7 Write Data Selection MUX *(result_mux)*

The *ALU* has the capability to carry out arithmetic operations like addition (A+B) or logical operation like equality comparison (A=B). Depending on the specific instructions being executed, the output of the *ALU* could be either a memory address or the result obtained from the *ALU* operation. To handle this situation, a *MUX* is needed to make decision between selecting the data address or the *ALU* output value for writing back to the register file. The *MUX* acts as a switch that selects one of the two inputs based on *ResultSrc* control signal. This allows flexibility and efficient data handling in the processor's pipeline. The implementation of *result_mux* in Verilog code is shown in Figure 3.13.



```
module result_mux (input logic [31:0] ALUResultW, ReadDataW, PCPlusW,input logic
[1:0] ResultSrcW, output logic [31:0] ResultW);
    assign ResultW = ResultSrcW[1] ? PCPlusW : (ResultSrcW[0] ? ReadDataW :
ALUResultW);
endmodule
```

Figure 3.13: Implementation of *result_mux* in Verilog code

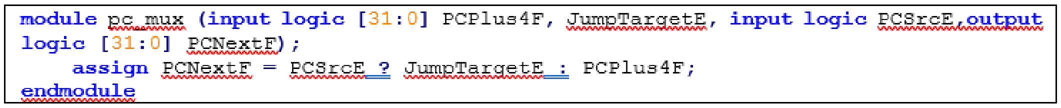
The corresponding output of *result_mux* is tabulated in Table 3.4 below.

Table 3.4: Write Data Selection MUX output table

ResultSrc	output
00	*ALUResultW*
01	*ReadDataW*
10	*PCPlusW*

3.2.8 Program Counter Selection MUX *(pc_mux)*

The Program Counter *(PC)* is a vital component used by the CPU to maintain the current instruction being executed. Under normal circumstances, the program counter increments by a fixed value, typically 4 (corresponding to a 32-bit instruction) for each clock cycle. This ensures that the program counter always points to the memory address of the next instruction to be executed. However, the program counter can be interrupted or modified by a jump signal *(jump)* from the control unit. When the predetermined conditions are met, the control unit instructs the program counter to deviate from its regular incrementation and instead updated its value to the jump address. Therefore, *pc_mux* is used to select the incremented instruction address *(PCPlus4F)* or the jump address *(JumpTargetE)*. The *pc_mux* is controlled by the *PCSrcE* signal. If *PCSrcE* signal is high, *pc_mux* will *JumpTargetE* on next clock cycle, else *PCPlus4F* will be selected. The implementation of *pc_mux* in Verilog code is shown in Figure 3.14.



```
module pc_mux (input logic [31:0] PCPlus4F, JumpTargetE, input logic PCSrcE,output
logic [31:0] PCNextF);
    assign PCNextF = PCSrcE ? JumpTargetE : PCPlus4F;
endmodule
```

Figure 3.14: Implementation of *pc_mux* in Verilog code

The corresponding output of *pc_mux* is tabulated in Table 3.5 below.

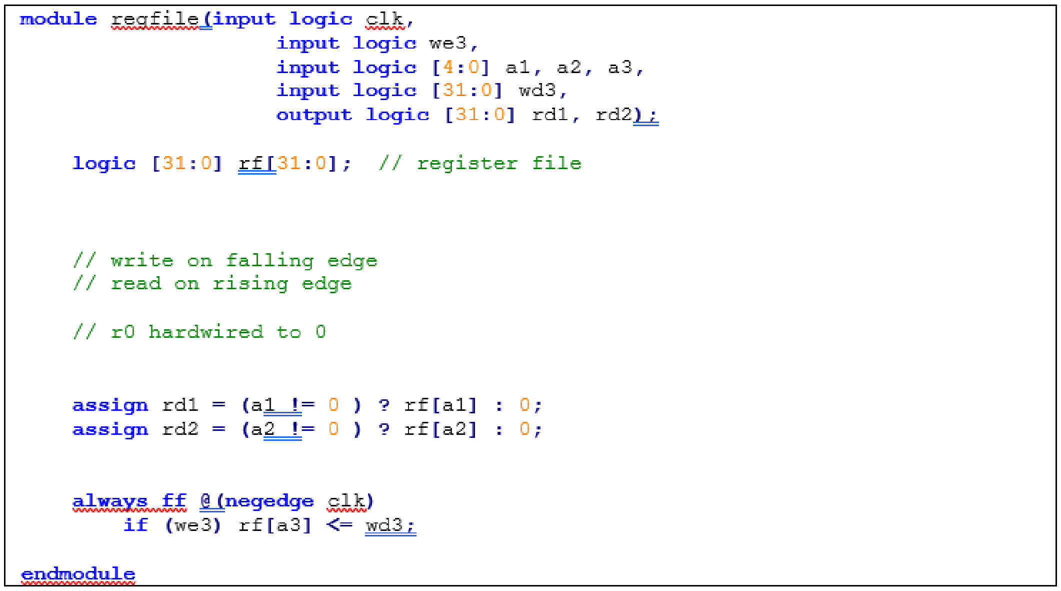
Table 3.5: Program Counter Selection MUX output table

PCSrcE	Output
0	*PCPlus4F*
1	*JumpTargetE*

3.2.9 Register File *(regfile)*

The register file *(regfile)* in a CPU plays a critical role in storing and manipulating data during program execution. It serves as a high-speed storage component that holds a set of registers, each capable of storing a fixed width if data. Usually it has

4 inputs namely, *RegWrite*, Instruction[19:15], Instruction[24:20], and *WriteData*. *RegWrite* control signal is used to control the write operation on the *regfile.* When *RegWrite* is high, the data from *WriteData* will be written into the *regfile.* Then, Instruction[19:15] and Instruction[24:20] served as an input from the pipeline register *(IF/ID)* and output to pipeline register *(ID/IEx)* for the ALU in execute stage. The implementation of *regfile* in Verilog code is shown in Figure 3.15.



```
module regfile(input logic clk,
                         input logic we3,
                         input logic [4:0] a1, a2, a3,
                         input logic [31:0] wd3,
                         output logic [31:0] rd1, rd2);

    logic [31:0] rf[31:0];  // register file


    // write on falling edge
    // read on rising edge

    // r0 hardwired to 0


    assign rd1 = (a1 != 0 ) ? rf[a1] : 0;
    assign rd2 = (a2 != 0 ) ? rf[a2] : 0;


    always_ff @(negedge clk)
        if (we3) rf[a3] <= wd3;

endmodule
```

Figure 3.15: Implementation of *regfile* in Verilog code

3.2.10 Hazard Unit *(hazardunit)*

The Hazard Unit is a component in a CPU's architecture that is responsible for detecting and handling hazards that can occur during the execution of instructions. Hazards refer to situations where the sequential execution of instructions may lead to incorrect or unintended behavior due to dependencies or conflicts between instructions. The hazard unit detects these hazards and takes appropriate actions to mitigate their effects. With this, the dependencies of write data begins from a pipeline register, rather than waiting for the WB stage to write the register file. Thus, the required data exists in time for later instructions, with the pipeline registers holding the data to be forwarded.

There are 2 solutions to handle hazards namely forwarding and stalling.

3.2.10.1 Forwarding

Forward is a technique used by hazard unit to handle data dependencies between instruction and mitigate hazards. It allows the CPU to forward data from the producer instruction to the consumer instruction, bypassing intermediate pipeline stages and avoid the need for stalls or bubbles.

There are 2 pairs of hazards conditions.

1 (a) EX/MEM.RegisterRd = ID/EX.RegisterRs

(b). EX/MEM.RegisterRd = ID/EX.RegisterRt

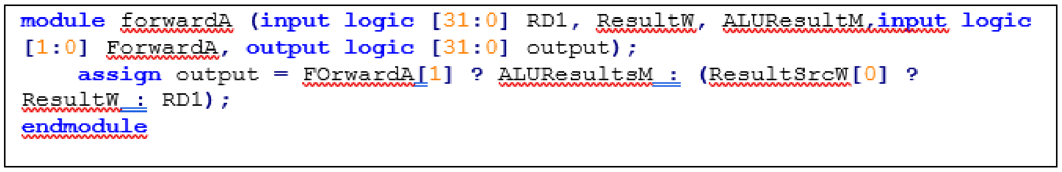
2 (a). MEM/WB.RegisterRd = ID/EX.RegisterRs

(b). MEM/WB.RegisterRd = ID/EX.RegisterRt

These hazard conditions can be handled by 2 forwarding control signals namely *ForwardA* and *ForwardB*. *ForwardA* is used to control *forwardMuxA* and *ForwardB* is used to control *forwardMuxB.*

3.2.10.1.1 Forward Multiplexer A *(forwardMuxA)*

Forward Multiplexer A takes the input of *RD1*, *ALUResultsM*, and *ResultW*. *RD1* is referred as register value. *ALUResultsM* is referred to the alu result in memory stage whereas *ResultW* is referred to the results in writeback stage. Both *ALUResultsM* and *ResultW* are forwarded values. *ForwardA* control signal is used to select either the register file value or the forwarded values. The implementation of *forwardMuxA* in Verilog code is shown in Figure 3.16.



```
module forwardA (input logic [31:0] RD1, ResultW, ALUResultM,input logic
[1:0] ForwardA, output logic [31:0] output);
    assign output = FOrwardA[1] ? ALUResultsM : (ResultSrcW[0] ?
ResultW : RD1);
endmodule
```

Figure 3.16: Implementation of *forwardMuxA* in Verilog code

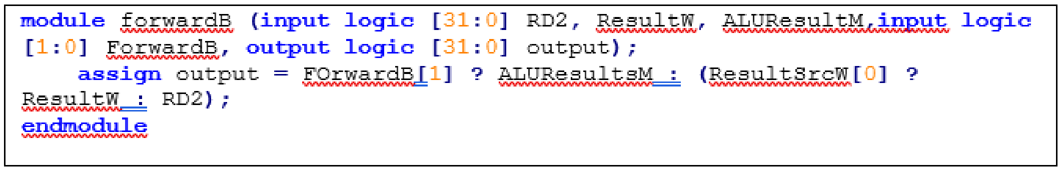
The corresponding output of *forwardMuxA* is tabulated in Table 3.6.

Table 3.6: Forward Multiplexer A output table

ForwardA	Output
00	*RD1*
01	*ResultW*
10	*ALUResultsM*

3.2.10.1.2 Forward Multiplexer B (forwardMuxB)

Forward Multiplexer B takes the input of *RD2*, *ALUResultsM*, and *ResultW*. *RD2* is referred as register value. *ALUResultsM* is referred to the alu result in memory stage whereas *ResultW* is referred to the results in writeback stage. Both *ALUResultsM* and *ResultW* are forwarded values. *ForwardB* control signal is used to select either the register file value or the forwarded values. The implementation of *forwardMuxB* in Verilog code is shown in Figure 3.17.



```
module forwardB (input logic [31:0] RD2, ResultW, ALUResultM,input logic
[1:0] ForwardB, output logic [31:0] output);
    assign output = FOrwardB[1] ? ALUResultsM : (ResultSrcW[0] ?
ResultW : RD2);
endmodule
```

Figure 3.17: Implementation of *forwardMuxB* in Verilog code

The corresponding output of *forwardMuxB* is tabulated in Table 3.7.

Table 3.7: Forward Multiplexer B output table

ForwardB	Output
00	*RD2*
01	*ResultW*
10	*ALUResultsM*

Below shows the conditions for detecting hazards and resolve them using forwarding control signals:

1. EX hazard
   a. if (EX/MEM.RegWrite and (EX/MEM.RegisterRd $\neq$ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

   b. if (EX/MEM.RegWrite and (EX/MEM.RegisterRd $\neq$ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10

2. EX hazard

   a. if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

   b. if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01

3.2.10.2 Stalling

Stalling is used when an instruction tries to read a register following a load instruction that writes the same register. The example of the instruction order is showed in Figure 3.18 below.

```
lw $2, 20($1)

and $4, $2, $5
```

Figure 3.18: Instruction sequence of Stalling condition

In this case, the *and* instruction will not get the updated value from *lw* because the data of the *lw* instruction is still being read from memory while the ALU is performing the operation for the *and* instruction. Therefore, the pipeline must be stalled for the combination of load following by an instruction that read its result.

Below shows the conditions for detecting hazards and resolve them using stalling:

```
if (ID/EX.MemRead and ((ID/EX.RegisterRt = IF/ID.RegisterRs)
or (ID/EX.RegisterRt = IF/ID.RegisterRt))) stall the pipeline
```

Chapter 4: Results and Discussion

4.1 Overview

In this chapter, each module that were discussed on previous chapter were simulated by using ModelSim software. Waveforms were generated from the simulations and the results were analyzed and discussed. Besides, the results were used to verify the functionality for each module of the design.

In the simulation, the functionality for each module were tested by running a set of test sequence with different instructions. The test sequence was shown in Figure 4.1 below. Furthermore, the clock cycle used throughout the simulation is set to 100ps.

main	
	addi x2 , x0, 5
	addi x3 , x0, 12
	addi x7 , x3, -9
	or x4 , x7 , x2
	xor x5 , x3 , x4
	add x5 , x5 , x4
	beq x5 , x7 , end
	slt x4 , x3 , x4
	beq x4 , x0 , around
	addi x5 , x0 , 0
around	
	slt x4 , x7 , x2
	add x7 , x4 , x5
	sub x7 , x7 , x2
	sw x7 , 84(x3)
	lw x2 , 96(x0)
	add x9 , x2 , x5
	jal x3 , end
	addi x2 , x0 , 1

end	
	add x2 , x2 , x9
	addi x4 , x0 , 1
	lui x5 , 0x80000
	slt x6 , x5 , x4
wrong	
	beq x6 , x0 , wrong
	lui x9 , 0xABCDE
	add x2 , x2 , x9
	sw x2 , 0x40(x3)
done	
	beq x2 , x2 , done

Figure 4.19: Simulation test sequence

From Figure 4.1, the test sequence consists of a main code with label *main,* 4 branches with label *around, end,* and *wrong*, and end with a *done* label. In the *main* label, it consists of some arithmetic and logical instruction such as *add*, *or*, and *xor*. With these instructions, *alu, data memory, instruction memory, register file,* and control unit module can be tested. Besides, the test was also designed to hit hazard conditions to test the hazard unit of the design. In the branches label, *sw* and *lw* instructions were executed to test the load/store operation of the design. Moreover, branching instruction such as *jal* and *beq* were also executed to test the jump or branch condition of the design. Other module such as *pipeline register, program counter* and *write data selection mux* were also tested and verified along with the simulation.

4.2 Simulation and Analysis of Waveforms

4.2.1 Program Counter Selection MUX (*pc_mux*) Waveform Analysis

The *pc_mux* are used to select *PCPlus4* or *JumpTarget* based on *PCSrcE* inputs. *PCPlus4* is the *PC* increment by 4 for each instruction cycle while *JumpTarget* is a specific point to call when conditions are met. When *PCSrcE* is high, *pc_mux* will select *JumpTarget* as an output, else *PCPlus4* is selected.

A part simulation waveform for *pc_mux* is captured and shown in Figure 4.2. The functionality of the module is verified by comparing the inputs and outputs waveforms as shown in Table 4.1 below.

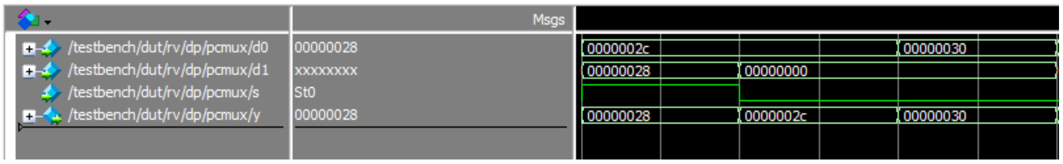

Msgs
/testbench/dut/rv/dp/pcmux/d0
/testbench/dut/rv/dp/pcmux/d1
/testbench/dut/rv/dp/pcmux/s
/testbench/dut/rv/dp/pcmux/y
00000028
xxxxxxxx
St0
00000028
0000002c
00000030
00000028
00000000
00000028
0000002c
00000030

Figure 4.2: Simulation waveform of *pc_mux*

From Figure 4.2, input *d0*, *d1*, and *s* are represented as *PCPlus4, JumpTarget*, and *PCSrcE* respectively and output y is represented as *PC.*

Table 4.1: Simulation output of *pc_mux*

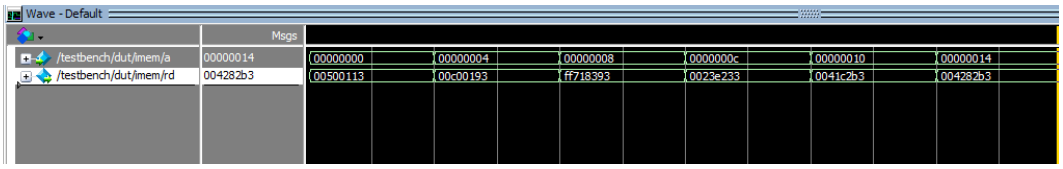
Inputs (hex)			Output (hex)
PCSrcE	*PCPlus4*	*JumpTarget*	*PC*
1	0000002c	00000028	00000028
0	0000002c	00000000	0000002c
0	00000030	00000000	00000030

From Table 4.1, the output is *JumpTarget* when *PCSrcE* is 1 whereas the output is *PCPlus4* when *PCSrcE* is 0. The outputs are expected. Therefore, the functionality of *PC* selection *MUX* is verified.

4.2.2 Instruction Memory (*imem*) Waveform Analysis

The instruction memory module is used to setup all the instruction flow of the test. From the simulation, there is a program counter fetch (*PCF*) input determines which instruction to fetch. Since each instruction had 4 bytes, the *PCF* value will increase by 4 to get the following instruction.

A part simulation waveform for *imem* is captured and shown in Figure 4.3. The functionality of the module is verified by viewing the changes of *PCF* value and the corresponding fetched instruction. The relationship of *PCF* and fetched instruction were tabulated in Table 4.2 below.



```
/testbench/dut/imem/a    00000014   00000000  00000004  00000008  0000000c  00000010  00000014
/testbench/dut/imem/rd   004282b3   00500113  00c00193  ff718393  0023e233  0041c2b3  004282b3
```

Figure 4.20: Simulation waveform of *imem*

From Figure 4.3, input *a* is represented as *PCF* and output *rd* is represented as fetched instruction. It is noticed that for every clock cycle, the value of *a* is added by 4. Besides, each value of *a* is tagged with a different instruction *rd*.

Table 4.2: Relationship of *PCF* and fetched instruction

PCF, a (hex)	*Fetched instruction, rd (hex)*
0	0x00500113
4	0x00C00193
8	0xFF718393
C	0x0023E233
10	0x0041C2B3
14	0x004282B3
18	0x02728863
1C	0x0041A233
20	0x00020463
24	0x00000293
28	0x0023A233
2C	0x005203B3
30	0x402383B3
34	0x0471AA23
38	0x06002103
3C	0x005104B3
40	0x008001EF
44	0x00100113
48	0x00910133
4C	0x00100213
50	0x800002B7
54	0x0042A333
58	0x00030063
5C	0xABCDE4B7
60	0x00910133
64	0x0421A023
68	0x00210063

4.2.3 Data Memory (*dmem*) Waveform Analysis

The data memory (*dmem*) is used to store data in the datapath. The data is written to the module if write enable bit (*MemWrite*) is set. Else, the data is read from the data address (*a*) from the *alu* output through *IEx/IMem* pipeline register.

A part simulation waveform for *dmem* is captured and shown in Figure 4.4.

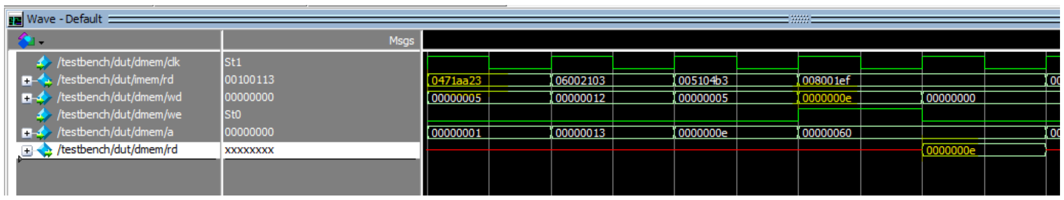

Wave - Default
Msgs
/testbench/dut/dmem/clk St1
/testbench/dut/imem/rd 00100113
/testbench/dut/dmem/wd 00000000
/testbench/dut/dmem/we St0
/testbench/dut/dmem/a 00000000
/testbench/dut/dmem/rd xxxxxxxx
0471aa23
06002103
005104b3
008001ef
00000005
00000012
00000005
0000000e
00000000
00000001
00000013
0000000e
00000060
0000000e

Figure 4.21: Simulation waveform of dmem

From Figure 4.4, *rd* represent the instruction. The focus of the simulation for *dmem* module is *sw* instruction because it is the only instruction that have *MemWrite* set to 1. Instruction 0x0471aa23 is an example of *sw* instruction. It is notice that after 3 cycles after *sw* instruction is fetched, the data reached memory stage. In memory stage, it read the *MemWrite* from the control unit. If the value is set, the writedata (*wd*) is written to data memory and output the value (*rd*) on the next cycle. Therefore, the functionality of *dmem* is verified.

4.2.4 Register File (*regfile*) Waveform Analysis

The register file (*regfile*) serves as a temporary memory to store data. From the simulation, the inputs for the *regfile* include read address (*a1* and *a2*), the write address of register (*a3*), the data to be written into the register (*wd3*), the data read from the register at the outputs (*rd1* and *rd2*), and the control signal (*we3*) to enable write data into the register.

A part simulation waveform for *regfile* is captured and shown in Figure 4.5.

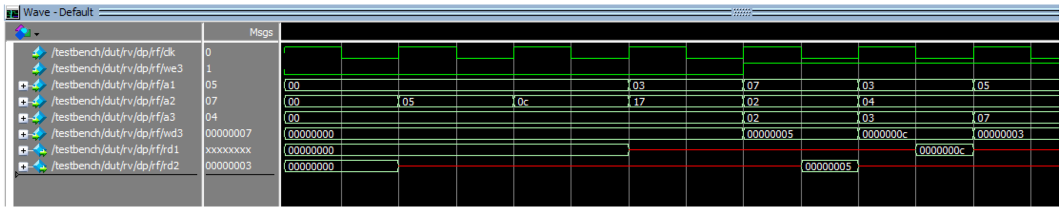

Wave - Default
Msgs
/testbench/dut/rv/dp/rf/clk
/testbench/dut/rv/dp/rf/we3
/testbench/dut/rv/dp/rf/a1
/testbench/dut/rv/dp/rf/a2
/testbench/dut/rv/dp/rf/a3
/testbench/dut/rv/dp/rf/wd3
/testbench/dut/rv/dp/rf/rd1
/testbench/dut/rv/dp/rf/rd2

Figure 4.22: Simulation waveform of *regfile*

From Figure 4.5, it is noticed that *a1* and *a2* holds the addresses of *rs1* and *rs2*. Besides, then *we3* is high, the data (*wd3*) is written into the addressed holds by *a3*. Furthermore, rd1 and rd2 output the value from the addresses holds by a1 and a2 respectively. Therefore, the functionality of *regfile* module is verified.

4.2.5 Arithmetic Logic Unit (*alu*) Waveform Analysis

The *alu* is an essential component of a CPU that handles arithmetic and logical operations on binary data. It receives input data from the *ID_IEx* stage (*SrcA and SrcB*) and perform various arithmetic operations based on signals from the ALU decoder (*ALUControl*). The output is stored as *ALUResults.*

A part simulation waveform for *alu* is captured and shown in Figure 4.6.

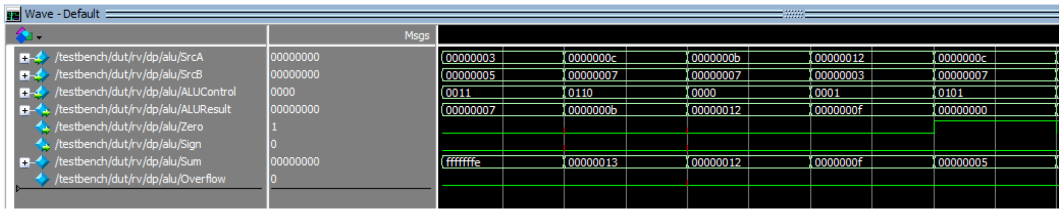

Wave - Default
Msgs
/testbench/dut/rv/dp/alu/SrcA
/testbench/dut/rv/dp/alu/SrcB
/testbench/dut/rv/dp/alu/ALUControl
/testbench/dut/rv/dp/alu/ALUResult
/testbench/dut/rv/dp/alu/Zero
/testbench/dut/rv/dp/alu/Sign
/testbench/dut/rv/dp/alu/Sum
/testbench/dut/rv/dp/alu/Overflow

Figure 4.23: Simulation waveform of *alu*

From Figure 4.6, instruction *OR, XOR, ADD, BEQ*, and *SLT* is simulated. With different *ALUControl,* the *alu* operates different operations. For example, with *ALUControl* = 0011, *alu* will perform OR operation. The inputs from SrcA and SrcB are 0x0001 and 0x0005 respectively. The result of the operation is 0x0007 and stored in *ALUResults.* Therefore, the functionality of *alu* module is verified. The analysis of the simulated *alu* waveform is tabulated in Table 4.3 below.

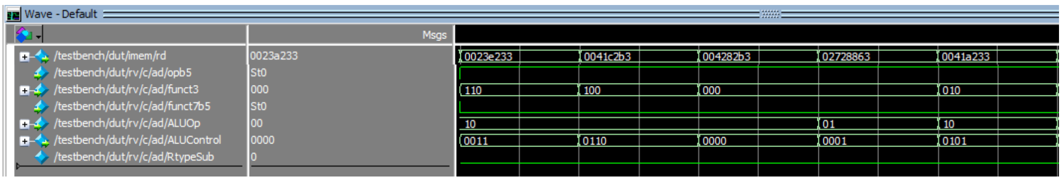
Table 4.3: Simulation output of *alu*

Operation	Input (hex)			Output (hex)
	SrcA	SrcB	ALUControl	ALUResults
OR	00000003	00000005	0011	00000007
XOR	0000000c	00000007	0010	0000000b
ADD	0000000b	00000007	0000	00000012
BEQ	00000012	00000003	0001	0000000f
SLT	0000000c	00000007	0101	00000000

4.2.6 Arithmetic Logic Unit Decoder (*aludec*) Waveform Analysis

The *aludec* decodes the instructions (*funct3, funct7b5* and *opb5*) and *ALUOp* from the control unit to determine the types of operation that has to be performed by the *alu.*

A part simulation waveform for *aludec* is captured and shown in Figure 4.7.



```
Wave - Default
                                    Msgs
/testbench/dut/imem/rd              0023a233   0023e233  0041c2b3  004282b3  02728863  0041a233
/testbench/dut/rv/c/ad/opb5         St0
/testbench/dut/rv/c/ad/funct3       000        110       100       000                 010
/testbench/dut/rv/c/ad/funct7b5     St0
/testbench/dut/rv/c/ad/ALUOp        00         10                            01        10
/testbench/dut/rv/c/ad/ALUControl   0000       0011      0110      0000      0001      0101
/testbench/dut/rv/c/ad/RtypeSub     0
```

Figure 4.24: Simulation waveform of *aludec*

From Figure 4.7, *rd* represent the instruction. *opb5, func3* and *func7b5* are extracted from *rd.* Besides, *ALUOp* is extracted from the control unit (*maindec*). With these extracted inputs, *aludec* returns the appropriate *ALUControl* as output to determine the operation. Therefore, the functionality of *aludec* is verified. The analysis of the simulated *aludec* waveform is tabulated in Table 4.4 below.

Table 4.4: Simulation output of *aludec*

Operation	Input					output
	rd (hex)	opb5	funct7b5	funct3	ALUOp	ALUControl
ADD	004282B3	0	1	000	10	0000
OR	0023E233	1	0	110	10	0011
XOR	0041C2B3	1	0	100	10	0110
BEQ	02728863	1	1	000	01	0001
SLT	0041A233	1	0	010	10	0101

4.2.7 Main decoder (*maindec*) Waveform Analysis

The *maindec* is used to generate control signals based on the 7 bits opcode (*instruction[6:0]*) of an instruction. Its purpose is to interpret the opcode and determine the type of instruction being executed.

A part simulation waveform for *maindec* is captured and shown in Figure 4.8.

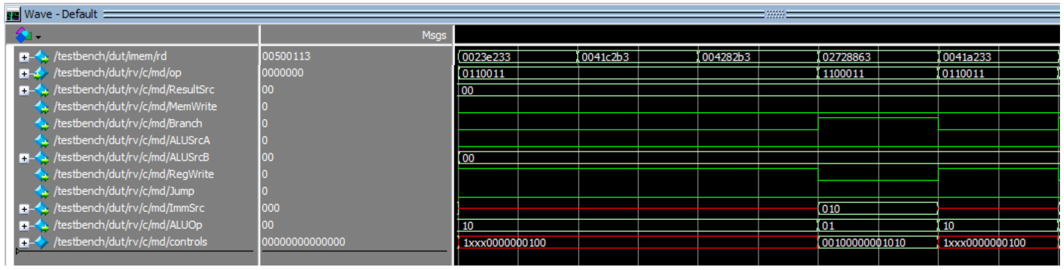

Wave - Default
Msgs
/testbench/dut/imem/rd 00500113
/testbench/dut/rv/c/md/op 0000000
/testbench/dut/rv/c/md/ResultSrc 00
/testbench/dut/rv/c/md/MemWrite 0
/testbench/dut/rv/c/md/Branch 0
/testbench/dut/rv/c/md/ALUSrcA 0
/testbench/dut/rv/c/md/ALUSrcB 00
/testbench/dut/rv/c/md/RegWrite 0
/testbench/dut/rv/c/md/Jump 0
/testbench/dut/rv/c/md/ImmSrc 000
/testbench/dut/rv/c/md/ALUOp 00
/testbench/dut/rv/c/md/controls 00000000000000
0023e233
0041c2b3
004282b3
02728863
0041a233
0110011
1100011
0110011
00
00
010
10
01
10
1xxx0000000100
00100000001010
1xxx0000000100

Figure 4.25: Simulation output of *maindec*

From Figure 4.8, 2 types of opcode simulation are shown – 0x0110011 and 0x1100011. 0x0110011 is categorized as R-type instruction. Therefore, the corresponding control signal are *RegWrite = 1, ImmSrc = xxx, ALUSrcA = 0, ALUSrcB = 00, MemWrite= 0, ResultSrc = 00, Branch = 0, ALUOp = 10,* and *Jump = 0.* Besides, 0x1100011 is categorized as B-type instruction. Therefore, the corresponding control signals are *RegWrite = 0, ImmSrc = 010, ALUSrcA = 0, ALUSrcB = 00, MemWrite= 0, ResultSrc = 00, Branch = 1, ALUOp = 01,* and *Jump = 0*. Other types of instructions such as *lw, sw, I-type,* and *jal* are verified through the simulation and tabulated in Table 4.5. Therefore, the functionality of *maindec* is verified.

Table 4.5: Simulation output of *maindec*

Control Signal	Instruction					
	lw	sw	R-Type	B-Type	I-Type	jal
Opcode[6:0]	0000011	0100011	0110011	1100011	0010011	1101111
RegWrite	1	0	1	0	1	1
ImmSrc	000	001	xxx	010	000	011
ALUSrcA	0	0	0	0	0	0
ALUSrcB	01	01	00	00	01	00
MemWrite	0	1	0	0	00	0
ResultSrc	01	00	00	00	000	10
Branch	0	0	0	1	0	0
ALUOp	00	00	10	01	10	00
Jump	0	0	0	0	0	1

4.2.8 Write Date Selection MUX (*result_mux*) Waveform Analysis

The *result_mux* is used to select *ALUResults* from ALU, *ReadData* from data memory, and *PCPlus* from program counter based on *ResultSrc* inputs. *ALUResults* is the address for the output from the *ALU* while *ReadData* is the data read from data memory. *PCPlus* is the *PC* increment by 4 for each instruction cycle. When bit 1 of *ResultSrc* is high, *result_mux* will select *PCPlus* as an output, else if bit 0 of *ResultSrc* is high, *result_mux* will select *ReadData*. Else, *ALUResult* will be selected.

A part simulation waveform for *result_mux* is captured and shown in Figure 4.9. The functionality of the module is verified by comparing the inputs and outputs waveforms as shown in Table 4.6 below.

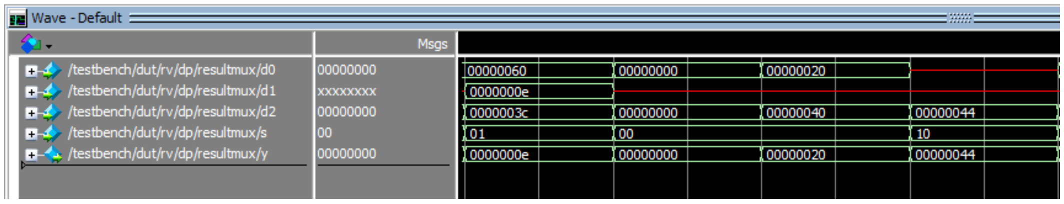

Wave - Default
Msgs
/testbench/dut/rv/dp/resultmux/d0 00000000
/testbench/dut/rv/dp/resultmux/d1 xxxxxxxx
/testbench/dut/rv/dp/resultmux/d2 00000000
/testbench/dut/rv/dp/resultmux/s 00
/testbench/dut/rv/dp/resultmux/y 00000000
00000060 00000000 00000020
0000000e
0000003c 00000000 00000040 00000044
01 00 10
0000000e 00000000 00000020 00000044

Figure 4.26: Simulation output of *result_mux*

From Figure 4.9, input *d0*, *d1*, *d2* and *s* are represented as *ALUResults, DataRead*, *PCPlus,* and *ResultSrc* respectively and output y is represented as *Result.*

Table 4.6: Simulation output of *result_mux*

Inputs				Output
ResultSrc (bin)	*ALUResults* (hex)	*DataRead* (hex)	*PCPlus* (hex)	*Result* (hex)
01	00000060	0000000e	0000003c	0000000e
00	00000000	x	00000000	00000000
00	00000020	x	00000040	00000020
10	x	x	00000044	00000044

From Table 4.6, the output is *PCPlus* if *ResultSrc* is 2b'10. Besides, the output is *DataRead* if *ResultSrc* is 2b'01. Moreover, the output is *PCPlus* if *ResultSrc* is

2b'00. The outputs are expected. Therefore, the functionality of *result_mux* is verified.

4.2.9 Pipeline Register

The pipeline register is used to store information from previous stage and load them to next stage. This ensures data can be carried forward correctly and allow instructions to be executed through the pipeline.

4.2.9.1 *IF_ID*

IF_ ID register located in between fetch stage and decode stage. It helps to store instructions (Instr), program counter (*PC*) and next cycle program counter (*PCPlus4*) from the fetch stage and load them to decode stage on the next cycle.

A part simulation waveform for *IF_ID* is captured and shown in Figure 4.10.

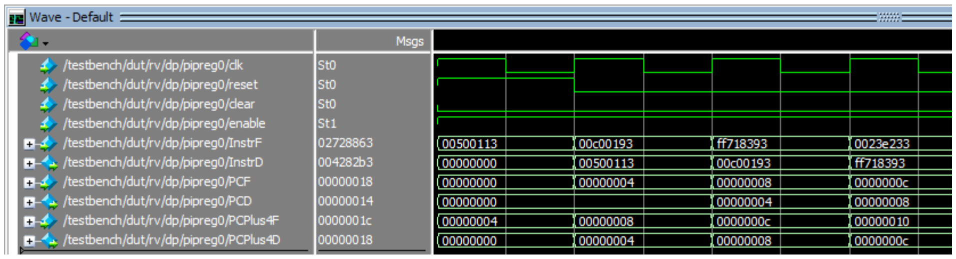

Wave - Default
Msgs
/testbench/dut/rv/dp/pipreg0/clk St0
/testbench/dut/rv/dp/pipreg0/reset St0
/testbench/dut/rv/dp/pipreg0/clear St0
/testbench/dut/rv/dp/pipreg0/enable St1
/testbench/dut/rv/dp/pipreg0/InstrF 02728863 00500113 00c00193 ff718393 0023e233
/testbench/dut/rv/dp/pipreg0/InstrD 004282b3 00000000 00500113 00c00193 ff718393
/testbench/dut/rv/dp/pipreg0/PCF 00000018 00000000 00000004 00000008 0000000c
/testbench/dut/rv/dp/pipreg0/PCD 00000014 00000000 00000004 00000008
/testbench/dut/rv/dp/pipreg0/PCPlus4F 0000001c 00000004 00000008 0000000c 00000010
/testbench/dut/rv/dp/pipreg0/PCPlus4D 00000018 00000000 00000004 00000008 0000000c

Figure 4.27: Simulation waveform of *IF/ID* register

From Figure 4.10, InstrF, PCF, and PCPlus4F represents the instruction, program counter and next cycle program counter in fetch stage respectively. Besides, InstrD, PCD, and PCPlus4D represents the instruction, program counter and next cycle program counter in decode stage respectively. It was noticed that the value of instruction, program counter, and next cycle program counter from fetch stage is loaded to decode stage in the next clock cycle. Therefore, the functionality of pipeline *IF_ID* is verified.

4.2.9.2 *ID_IEx*

ID_IEx register located in between decode stage and execute stage. It helps to store read data *(RD1, RD2)* from the register file, extended immediate value *(ImmExt)*, Instruction[11:7] *(rd)*, Instruction[19:15] *(rs1)*, and Instruction[24:20] *(rs2)* from decode stage and load them to execute stage. Besides, *PC* and *PCPlus4F* from *IF_ID* register were also carried forward and stored to *ID_IEx.*

A part simulation waveform for *ID_IEx* is captured and shown in Figure 4.11.

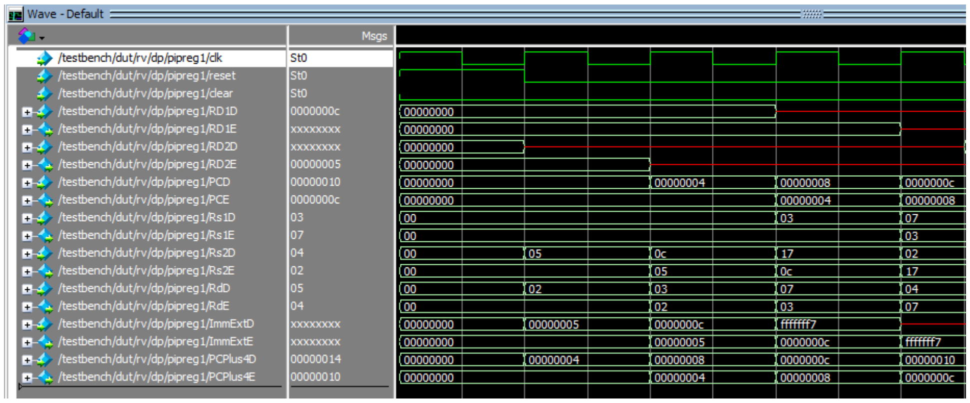

Wave - Default
Msgs
/testbench/dut/rv/dp/pipreg1/clk St0
/testbench/dut/rv/dp/pipreg1/reset St0
/testbench/dut/rv/dp/pipreg1/clear St0
/testbench/dut/rv/dp/pipreg1/RD1D 0000000c
/testbench/dut/rv/dp/pipreg1/RD1E xxxxxxxx
/testbench/dut/rv/dp/pipreg1/RD2D xxxxxxxx
/testbench/dut/rv/dp/pipreg1/RD2E 00000005
/testbench/dut/rv/dp/pipreg1/PCD 00000010
/testbench/dut/rv/dp/pipreg1/PCE 0000000c
/testbench/dut/rv/dp/pipreg1/Rs1D 03
/testbench/dut/rv/dp/pipreg1/Rs1E 07
/testbench/dut/rv/dp/pipreg1/Rs2D 04
/testbench/dut/rv/dp/pipreg1/Rs2E 02
/testbench/dut/rv/dp/pipreg1/RdD 05
/testbench/dut/rv/dp/pipreg1/RdE 04
/testbench/dut/rv/dp/pipreg1/ImmExtD xxxxxxxx
/testbench/dut/rv/dp/pipreg1/ImmExtE xxxxxxxx
/testbench/dut/rv/dp/pipreg1/PCPlus4D 00000014
/testbench/dut/rv/dp/pipreg1/PCPlus4E 00000010

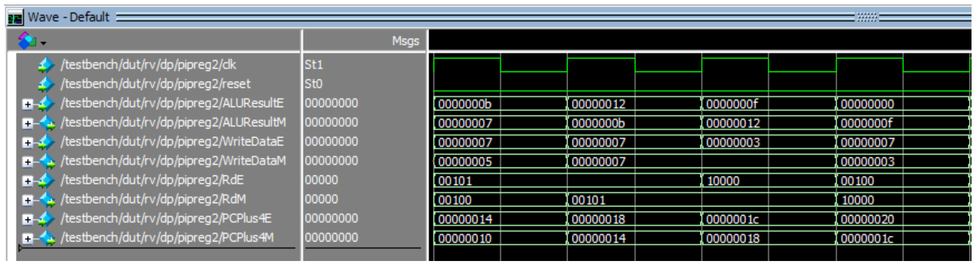
Figure 4.28: Simulation waveform of *ID/IEx* register

From Figure 4.11, *RD1D*, *RD2D*, *PCD*, *Rs1D*, *Rs2D*, *RdD*, *ImmExtD*, and *PCPlusD* are in decode stage whereas *RD1E*, *RD2E*, *PCE*, *Rs1E*, *Rs2E*, *RdE*, *ImmExtE*, and *PCPlusE* are in execute stage. It was noticed that the values from decode stage are loaded to the respective registers in execute stage in the next clock cycle. Therefore, the functionality of *ID_IEx* is verified.

4.2.9.3 *IEx_IMem*

IEx_IMem register located in between execute stage and memory stage. It helps to store the ALU results (*ALUResults*) and write data (*WriteData*) from execute stage and load them in memory stage. At the same time, Instruction[11:7] *(rd)* and *PCPlus4F* is carried forward and stored to *IEx_IMem*.

A part simulation waveform for *IEx_IMem* is captured and shown in Figure 4.12.



```
/testbench/dut/rv/dp/pipreg2/clk           St1
/testbench/dut/rv/dp/pipreg2/reset         St0
/testbench/dut/rv/dp/pipreg2/ALUResultE    00000000   0000000b   00000012   0000000f   00000000
/testbench/dut/rv/dp/pipreg2/ALUResultM    00000000   00000007   0000000b   00000012   0000000f
/testbench/dut/rv/dp/pipreg2/WriteDataE    00000000   00000007   00000007   00000003   00000007
/testbench/dut/rv/dp/pipreg2/WriteDataM    00000000   00000005   00000007              00000003
/testbench/dut/rv/dp/pipreg2/RdE           00000      00101                 10000      00100
/testbench/dut/rv/dp/pipreg2/RdM           00000      00100      00101                 10000
/testbench/dut/rv/dp/pipreg2/PCPlus4E      00000000   00000014   00000018   0000001c   00000020
/testbench/dut/rv/dp/pipreg2/PCPlus4M      00000000   00000010   00000014   00000018   0000001c
```

Figure 4.29: Simulation waveform of *IEx/IMem* register

From Figure 4.12, *ALUResultE*, *WriteDataE*, *RdE*, and *PCPlus4E* are in execute stage whereas *ALUResultM*, *WriteDataM*, *RdM*, and *PCPlus4M* are in execute stage. It was noticed that the values from execute stage are loaded to the respective registers in memory stage in the next clock cycle. Therefore, the functionality of *IEx_IMem* is verified.

4.2.9.4 *IMem_IW*

IMem_IW is located in between memory stage and writeback stage. It helps to store the ALU results (*ALUResults*) from the alu and read data (*ReadData)* from data memory. At the same time, Instruction[11:7] *(rd)* and *PCPlus4F* is carried forward and stored to *IMem_IW*.

A part simulation waveform for *IMem/IW* is captured and shown in Figure 4.13.

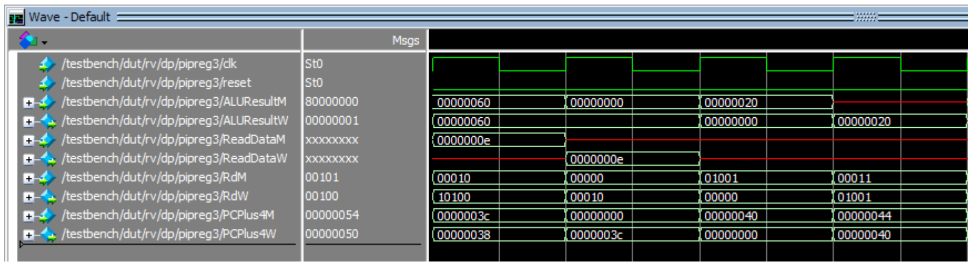

Wave - Default
Msgs
/testbench/dut/rv/dp/pipreg3/clk St0
/testbench/dut/rv/dp/pipreg3/reset St0
/testbench/dut/rv/dp/pipreg3/ALUResultM 80000000
/testbench/dut/rv/dp/pipreg3/ALUResultW 00000001
/testbench/dut/rv/dp/pipreg3/ReadDataM xxxxxxxx
/testbench/dut/rv/dp/pipreg3/ReadDataW xxxxxxxx
/testbench/dut/rv/dp/pipreg3/RdM 00101
/testbench/dut/rv/dp/pipreg3/RdW 00100
/testbench/dut/rv/dp/pipreg3/PCPlus4M 00000054
/testbench/dut/rv/dp/pipreg3/PCPlus4W 00000050
00000060 00000000 00000020
00000060 00000000 00000020
0000000e
0000000e
00010 00000 01001 00011
10100 00010 00000 01001
0000003c 00000000 00000040 00000044
00000038 0000003c 00000000 00000040

Figure 4.30: Simulation waveform of *IMem/IW* register

From Figure 4.13, *ALUResultM*, *ReadDataM*, *RdM*, and *PCPlus4M* are in memory stage whereas *ALUResultW*, *ReadDataW*, *RdW*, and *PCPlus4W* are in writeback stage. It was noticed that the values from memory stage are loaded to the respective registers in writeback stage in the next clock cycle. Therefore, the functionality of *IMem_IW* is verified.

4.2.10 Hazard Unit (*hazardunit*)

The hazard unit is used to detect situations where the sequential execution of instructions may lead to incorrect behaviour due to data dependencies from the previous instruction. For example, Figure 4.14 shows 2 instructions executed sequentially.

addi x7 , x3, -9
or x4 , x7 , x2

Figure 4.314: Instruction sequence with hazard condition

The first instruction was executed *addi* and store the output into *x7* register. Then, the second instruction was executed *or* with the dependencies on first instruction because *x7* register is an input for second instruction. In this case, the second instruction will get the wrong value of *x7* if hazard unit is not existed. This is because, the first instruction will only update the *x7* register in the writeback stage, which could not happen before the second instruction needs it. With hazard unit, the output of the first instruction can be forwarded from pipeline register. Therefore, the dependencies of write data can be mitigated rather than waiting for writeback stage to write the register.

4.2.10.1 Forward Multiplexer A (*forwardMuxA*)

Forward Multiplexer A (*forwardMuxA*) is used to selected between *RD1*, *ALUResultsM*, and *ResultW* based on *forwardA* control signal.

A part of simulation of *forwardMuxA* is shown in Figure 4.15. The functionality of the module is verified by comparing the inputs and outputs waveforms as shown in Table 4.7 below.

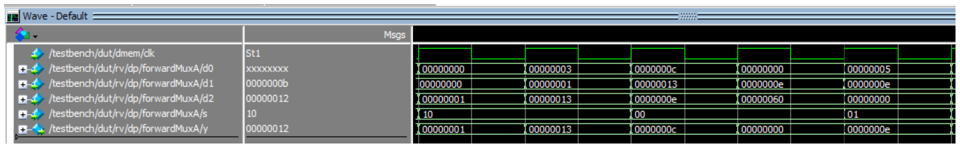

Wave - Default
Msgs
/testbench/dut/dmem/clk St1
/testbench/dut/rv/dp/forwardMuxA/d0 xxxxxxxx 00000000 00000003 0000000c 00000000 00000005
/testbench/dut/rv/dp/forwardMuxA/d1 0000000b 00000000 00000001 00000013 0000000e 0000000e
/testbench/dut/rv/dp/forwardMuxA/d2 00000012 00000001 00000013 0000000e 00000060 00000000
/testbench/dut/rv/dp/forwardMuxA/s 10 10 00 01
/testbench/dut/rv/dp/forwardMuxA/y 00000012 00000001 00000013 0000000c 00000000 0000000e

Figure 4.32: Simulation waveform of *forwardMuxA*

From Figure 4.15, input *d0*, *d1*, *d2* and *s* are represented as *RD1, ResultW*, *ALUResultsM,* and *forwardA* respectively and output y is represented as *Result.*

Table 4.7: Simulation output of *fowrardMuxA*

Inputs				Output
forwardA (bin)	*RD1* (hex)	*ResultW* (hex)	*ALUResultsM* (hex)	*Result* (hex)
00	0000000c	00000013	0000000e	0000000c
01	00000005	0000000e	00000000	0000000e
10	00000003	00000001	00000013	00000013

From Table 4.7, the output is *RD1* if *forwardA* is 2b'00. Besides, the output is *ResultW* if *forwardA* is 2b'01. Moreover, the output is *ALUResultsM* if *forwardA* is 2b'10. The outputs are expected. Therefore, the functionality of *forwardMuxA* is verified.

Forward Multiplexer A is used to handle the following 2 conditions.

1. EX/MEM.RegisterRd = ID/EX.RegisterRs
2. MEM/WB.RegisterRd = ID/EX.RegisterRs

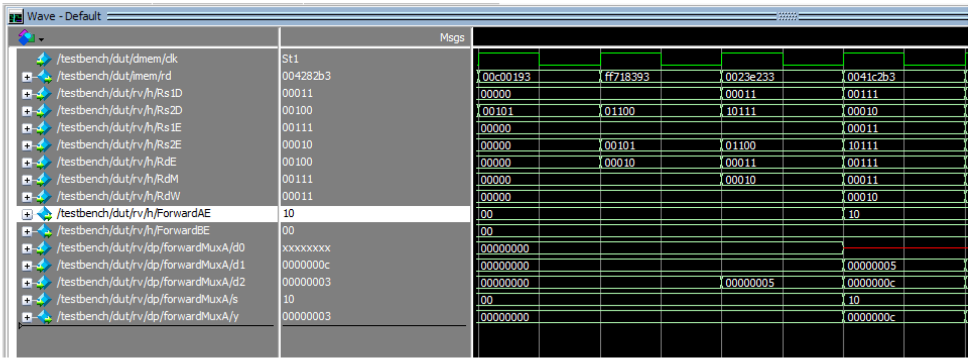
Conditions 1 happens when the destination register (*rd*) in *IEx/IMem* pipeline register is the same as the source register (*rs*) in *ID/IEx* pipeline register. The example of this condition is shown in Figure 4.16 below.

addi x3 , x0,   12
addi x7 , x3, -9

Figure 4.33: Instruction sequence with hazard condition 1

From Figure 4.16, the 2 instructions are executed sequentially. The first instruction *addi* has a destination register (*rd*) of register *x3*. Then, the second instruction used the register *x3* as source register (*rs*). As these instructions are executed sequentially, the first instruction is one stage ahead the second instruction. By the time the first instruction enters execute stage, the second instruction enter decode stage. This condition causes hazard because first instruction have not reach writeback stage to update the value in register *x3* before second instruction uses it.

Figure 4.17 shows the simulation of hazard condition 1.



```
/testbench/dut/dmem/clk                 St1
/testbench/dut/imem/rd                  004282b3
/testbench/dut/rv/h/Rs1D                00011
/testbench/dut/rv/h/Rs2D                00100
/testbench/dut/rv/h/Rs1E                00111
/testbench/dut/rv/h/Rs2E                00010
/testbench/dut/rv/h/RdE                 00100
/testbench/dut/rv/h/RdM                 00111
/testbench/dut/rv/h/RdW                 00011
/testbench/dut/rv/h/ForwardAE           10
/testbench/dut/rv/h/ForwardBE           00
/testbench/dut/rv/dp/forwardMuxA/d0     xxxxxxxx
/testbench/dut/rv/dp/forwardMuxA/d1     0000000c
/testbench/dut/rv/dp/forwardMuxA/d2     00000003
/testbench/dut/rv/dp/forwardMuxA/s      10
/testbench/dut/rv/dp/forwardMuxA/y      00000003
```

Figure 4.34: Simulation waveform of hazard condition 1

The simulation on Figure 4.17 is based on the instructions in Figure 4.16 above. The first instruction is fetched in the first clock cycle whereas the second instruction is fetched in the second clock cycle. At the fourth clock cycle, the first instruction reached memory stage whereas the second instruction reached execute stage. Then, it was noticed that the value of destination register in memory stage (*RdM*) is equal to the value of source register in execute stage (*Rs1E*). Therefore, hazard condition 1 is detected. With hazard condition 1 being detected, *forwardA* output a control signal of 2b'10 to forward the *ALUResultM* from the memory stage to be used by the second instruction.

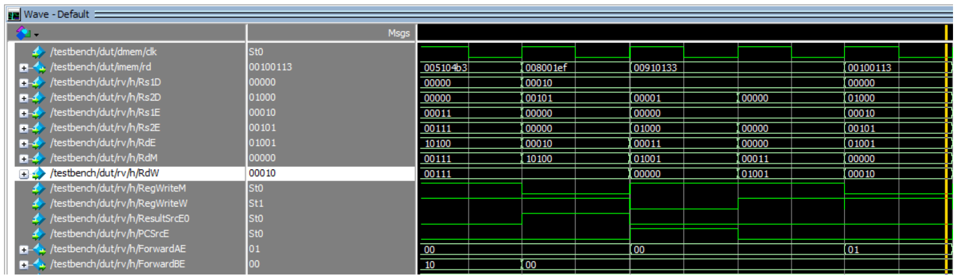
Condition 2 happens when destination register (*rd*) in *IMem/IW* pipeline register is the same as the source register (*rs*) in *ID/IEx* pipeline register. The example of this condition is shown in Figure 4.18 below.

addi x7 , x3, -9
sub x4 , x5 , x2
xor x1, x7, x6

Figure 4.35: Instruction sequence with hazard condition 2

From Figure 4.18, these 3 instructions are executed sequentially. The first instruction *addi* has a destination register (*rd*) of register *x7*, followed by the second instruction *sub* which had no dependencies on the first instruction. Then, the third instruction *xor* used register *x7* as source register which created dependencies on the first instruction. As these instructions are executed sequentially, the first instruction is 2 stages ahead on the third instruction. By the time the first instruction enters memory stage, the third instruction enters execute stage. This condition causes hazard because the first instruction have not reach writeback stage to update the value in register *x7* before the third instruction uses it.

Figure 4.19 shows the simulation of hazard condition 2.



```
Wave - Default
                                     Msgs
/testbench/dut/dmem/clk              St0
/testbench/dut/imem/rd               00100113
/testbench/dut/rv/h/Rs1D             00000
/testbench/dut/rv/h/Rs2D             01000
/testbench/dut/rv/h/Rs1E             00010
/testbench/dut/rv/h/Rs2E             00101
/testbench/dut/rv/h/RdE              01001
/testbench/dut/rv/h/RdM              00000
/testbench/dut/rv/h/RdW              00010
/testbench/dut/rv/h/RegWriteM        St0
/testbench/dut/rv/h/RegWriteW        St1
/testbench/dut/rv/h/ResultSrcE0      St0
/testbench/dut/rv/h/PCSrcE           St0
/testbench/dut/rv/h/ForwardAE        01
/testbench/dut/rv/h/ForwardBE        00
```

Figure 4.36: Waveform simulation of hazard condition 2

The simulation on Figure 4.19 is based on the instructions in Figure 4.18 above. The first instruction is fetched in the first clock cycle followed by the second instruction in the second clock cycle and the third instruction in the third clock cycle. At the fifth clock cycle, the first instruction reached writeback stage whereas the third instruction reached execute stage. Then, it was noticed that the value of destination register in writeback stage (*RdW*) is equal to the value of source register in execute stage (*Rs1E*). Therefore, hazard condition 2 is detected. With hazard condition 2 being detected, *forwardA* output a control signal of 2b'01 to forward the *ResultW* from the writeback stage to be used by the third instruction.

4.2.10.2 Forward Multiplexer B (*forwardMuxB*)

Forward Multiplexer B (*forwardMuxB*) is used to selected between *RD2*, *ALUResultsM*, and *ResultW* based on *forwardB* control signal.

A part of simulation of *forwardMuxB* is shown in Figure 4.20. The functionality of the module is verified by comparing the inputs and outputs waveforms as shown in Table 4.8 below.

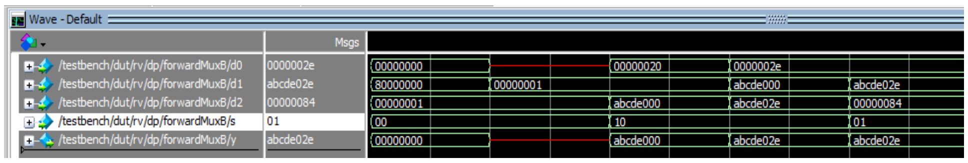


Figure 4.37: Simulation waveform of *forwardMuxB*

From Figure 4.20, input *d0*, *d1*, *d2* and *s* are represented as *RD2, ResultW*, *ALUResultsM,* and *forwardB* respectively and output y is represented as *Result.*

Table 4.8: Simulation output of *forwardMuxB*

Inputs				Output
forwardB (bin)	*RD2* (hex)	*ResultW* (hex)	*ALUResultsM* (hex)	*Result* (hex)
00	00000000	80000000	00000001	00000000
01	0000002e	abcde02e	00000084	abcde02e
10	00000020	00000001	abcde000	abcde000

From Table 4.8, the output is *RD2* if *forwardB* is 2b'00. Besides, the output is *ResultW* if *forwardB* is 2b'01. Moreover, the output is *ALUResultsM* if *forwardB* is 2b'10. The outputs are expected. Therefore, the functionality of *forwardMuxB* is verified.

Forward Multiplexer B is used to handle hazard condition 3 and 4 as shown below.

3. EX/MEM.RegisterRd = ID/EX.RegisterRt
4. MEM/WB.RegisterRd = ID/EX.RegisterRt

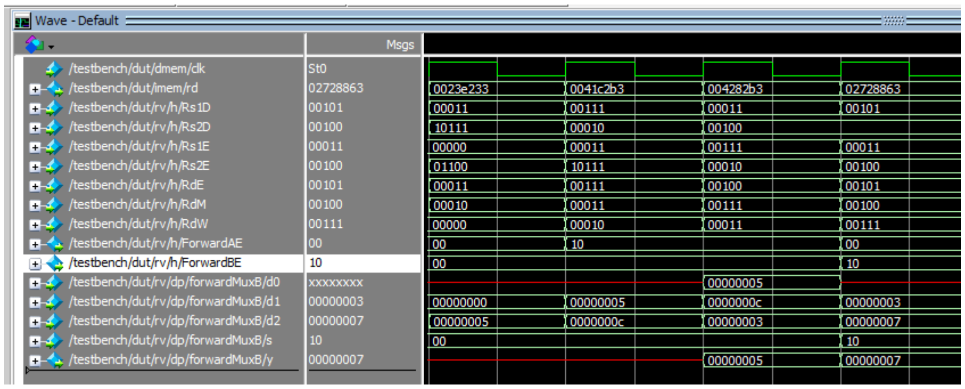
Conditions 3 happens when the destination register (*rd*) in *IEx/IMem* pipeline register is the same as the target register (*rt*) in *ID/IEx* pipeline register. The example of this condition is shown in Figure 4.21 below.

or x4 , x7 , x2
xor x5 , x3 , x4

Figure 4.38: Instruction sequence of hazard condition 3

From Figure 4.21, the 2 instructions are executed sequentially. The first instruction *or* has a destination register (*rd*) of register *x4*. Then, the second instruction used the register *x4* as target register (*rt*). As these instructions are executed sequentially, the first instruction is one stage ahead the second instruction. By the time the first instruction enters execute stage, the second instruction enter decode stage. This condition causes hazard because first instruction have not reach writeback stage to update the value in register *x4* before second instruction uses it.

Figure 4.22 shows the simulation of hazard condition 3.



```
/testbench/dut/dmem/clk                St0
/testbench/dut/imem/rd                 02728863   0023e233  0041c2b3  004282b3  02728863
/testbench/dut/rv/h/Rs1D               00101      00011     00111     00011     00101
/testbench/dut/rv/h/Rs2D               00100      10111     00010     00100
/testbench/dut/rv/h/Rs1E               00011      00000     00011     00111     00011
/testbench/dut/rv/h/Rs2E               00100      01100     10111     00010     00100
/testbench/dut/rv/h/RdE                00101      00011     00111     00100     00101
/testbench/dut/rv/h/RdM                00100      00010     00011     00111     00100
/testbench/dut/rv/h/RdW                00111      00000     00010     00011     00111
/testbench/dut/rv/h/ForwardAE          00         00        10                  00
/testbench/dut/rv/h/ForwardBE          10         00                            10
/testbench/dut/rv/dp/forwardMuxB/d0    xxxxxxxx                       00000005
/testbench/dut/rv/dp/forwardMuxB/d1    00000003   00000000  00000005  0000000c  00000003
/testbench/dut/rv/dp/forwardMuxB/d2    00000007   00000005  0000000c  00000003  00000007
/testbench/dut/rv/dp/forwardMuxB/s     10         00                            10
/testbench/dut/rv/dp/forwardMuxB/y     00000007                       00000005  00000007
```

Figure 4.39: Simulation waveform of hazard condition 3

The simulation on Figure 4.22 is based on the instructions in Figure 4.21 above. The first instruction is fetched in the first clock cycle whereas the second instruction is fetched in the second clock cycle. At the fourth clock cycle, the first instruction reached memory stage whereas the second instruction reached execute stage. Then, it was noticed that the value of destination register in memory stage (*RdM*) is equal to the value of target register in execute stage (*Rs2E*). Therefore, hazard condition 3 is detected. With hazard condition 3 being detected, *forwardB* output a control signal of 2b'10 to forward the *ALUResultM* from the memory stage to be used by the second instruction.

Condition 4 happens when destination register (*rd*) in *IMem/IW* pipeline register is the same as the target register (*rt*) in *ID/IEx* pipeline register. The example of this condition is shown in Figure 4.23 below.

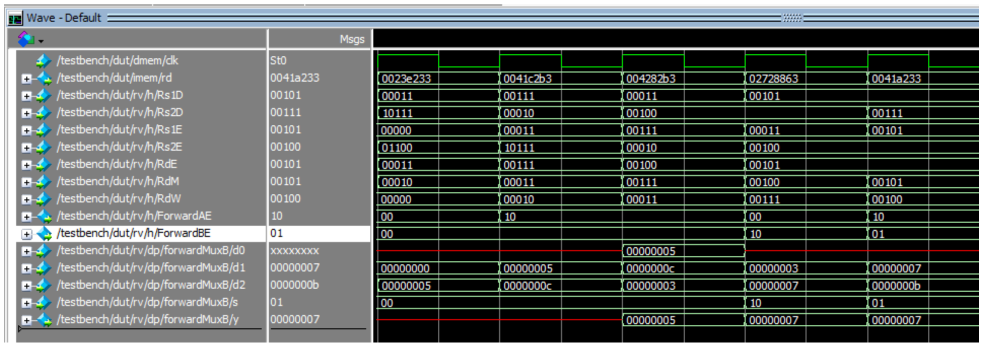
or x4 , x7 , x2
xor x6 , x3 , x5

add x5, x3, x4

Figure 4.40: Instruction sequence of hazard condition 4

From Figure 4.23, these 3 instructions are executed sequentially. The first instruction *or* has a destination register (*rd*) of register *x4*, followed by the second instruction *xor* which had no dependencies on the first instruction. Then, the third instruction *add* used register *x4* as target register which created dependencies on the first instruction. As these instructions are executed sequentially, the first instruction is 2 stages ahead on the third instruction. By the time the first instruction enters memory stage, the third instruction enters execute stage. This condition causes hazard because the first instruction have not reach writeback stage to update the value in register *x4* before the third instruction uses it.

Figure 4.24 shows the simulation of hazard condition 4.



Wave - Default

	Msgs					
/testbench/dut/dmem/clk	St0					
/testbench/dut/imem/rd	0041a233	0023e233	0041c2b3	004282b3	02728863	0041a233
/testbench/dut/rv/h/Rs1D	00101	00011	00111	00011	00101	
/testbench/dut/rv/h/Rs2D	00111	10111	00010	00100		00111
/testbench/dut/rv/h/Rs1E	00101	00000	00011	00111	00011	00101
/testbench/dut/rv/h/Rs2E	00100	01100	10111	00010	00100	
/testbench/dut/rv/h/RdE	00101	00011	00111	00100	00101	
/testbench/dut/rv/h/RdM	00101	00010	00011	00111	00100	00101
/testbench/dut/rv/h/RdW	00100	00000	00010	00011	00111	00100
/testbench/dut/rv/h/ForwardAE	10	00	10		00	10
/testbench/dut/rv/h/ForwardBE	01	00			10	01
/testbench/dut/rv/dp/forwardMuxB/d0	xxxxxxxx			00000005		
/testbench/dut/rv/dp/forwardMuxB/d1	00000007	00000000	00000005	0000000c	00000003	00000007
/testbench/dut/rv/dp/forwardMuxB/d2	0000000b	00000005	0000000c	00000003	00000007	0000000b
/testbench/dut/rv/dp/forwardMuxB/s	01	00			10	01
/testbench/dut/rv/dp/forwardMuxB/y	00000007			00000005	00000007	00000007

Figure 4.41: Simulation waveform of hazard condition 4

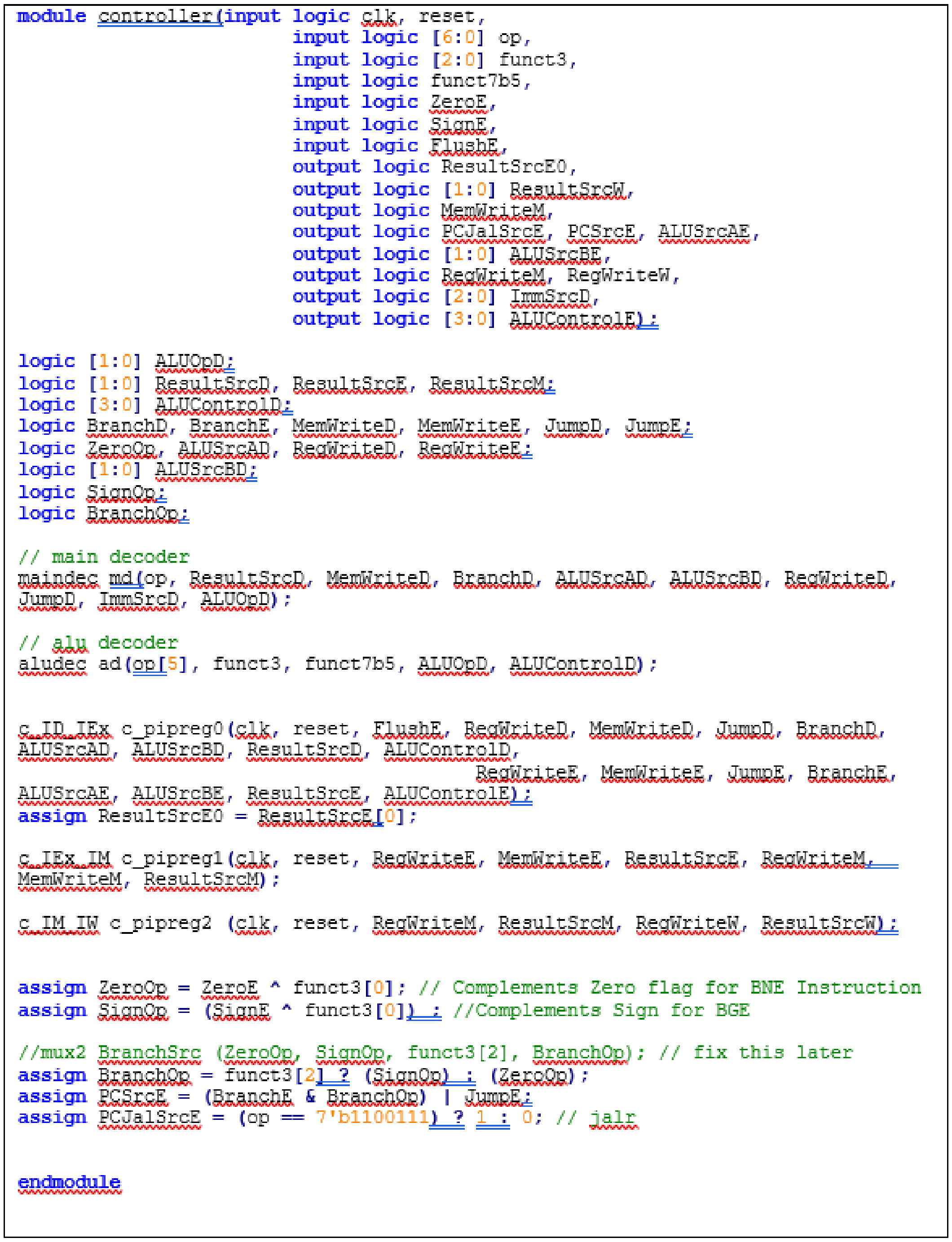
The simulation on Figure 4.24 is based on the instructions in Figure 4.23 above. The first instruction is fetched in the first clock cycle followed by the second instruction in the second clock cycle and the third instruction in the third clock cycle. At the fifth clock cycle, the first instruction reached writeback stage whereas the third instruction reached execute stage. Then, it was noticed that the value of destination register in writeback stage (*RdW*) is equal to the value of source register in execute stage (*Rs2E*). Therefore, hazard condition 4 is detected. With hazard condition 4 being detected, *forwardB* output a control signal of 2b'01 to forward the *ResultW* from the writeback stage to be used by the third instruction.

Based on the simulation above, it was noticed that the hazard unit module was able to identify all the hazard conditions. Besides, *forwardMuxA* was used in hazard unit to mitigate the hazard condition 1 and hazard condition 2 by forwarding the information from destination register in the memory or writeback stage to the source register in the execute stage based on *forwardA* control signal. Furthermore, *forwardMuxB* was used in hazard unit to mitigate the hazard condition 3 and hazard condition 4 by forwarding the information from destination register in the memory or writeback stage to the target register in the execute stage based on *forwardB* control signal. Therefore, the functionality of the hazard unit module is verified.

4.3 Integrating RISC-V processor

The RISC-V processor is developed by integrating all the verified modules together. The integration had 3 main units which namely controller unit, hazard unit, and datapath unit. The controller unit helped to send out control signals based on different instruction being executed. Besides, the hazard unit helped to detect hazard conditions during the execution and mitigated them. Moreover, the datapath unit helped to carry information throughout the 5 pipeline stages in the datapath.

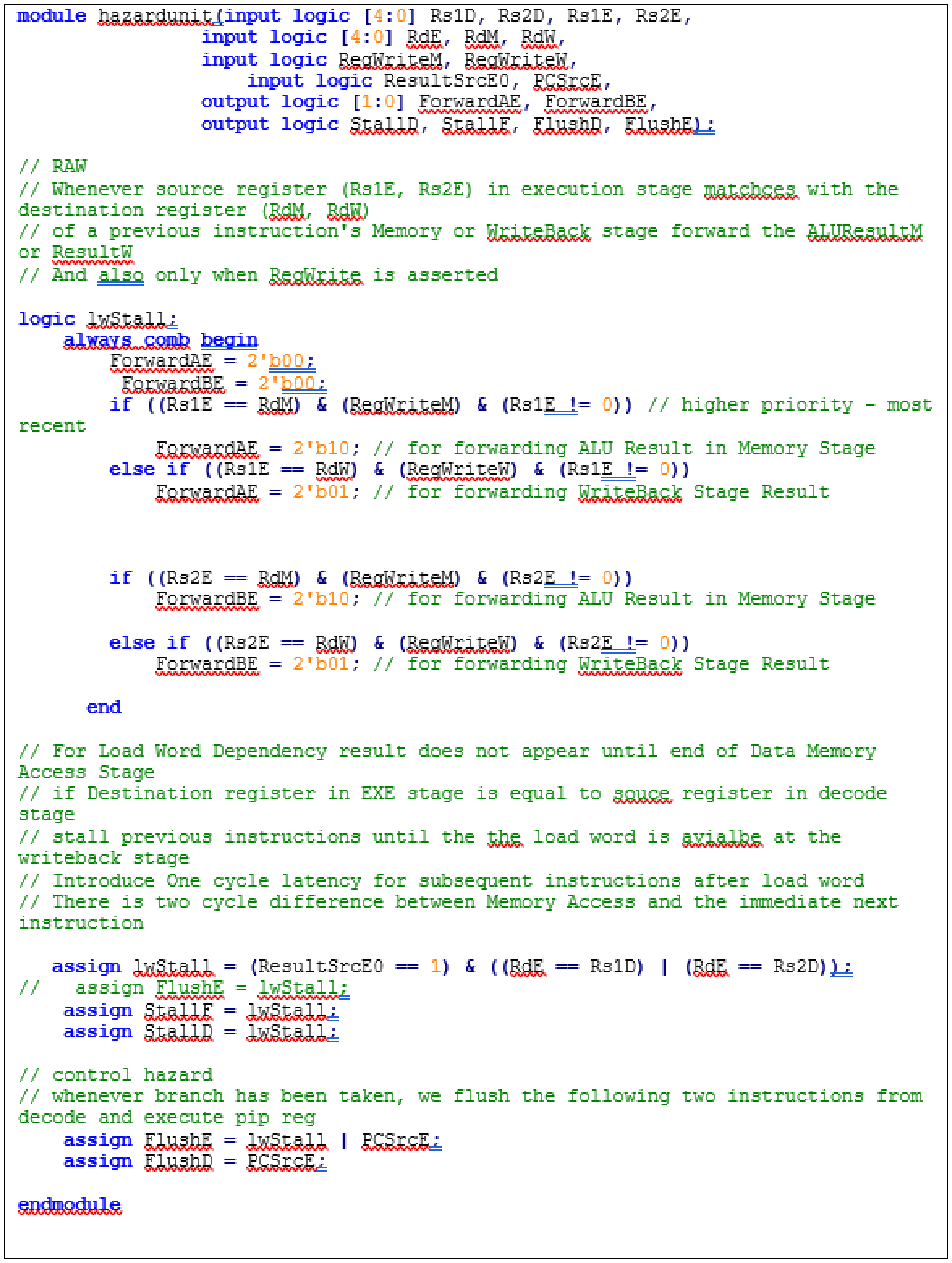
Figure 4.25 showed the Verilog code of the integrated control unit.



```
module controller(input logic clk, reset,
                        input logic [6:0] op,
                        input logic [2:0] funct3,
                        input logic funct7b5,
                        input logic ZeroE,
                        input logic SignE,
                        input logic FlushE,
                        output logic ResultSrcE0,
                        output logic [1:0] ResultSrcW,
                        output logic MemWriteM,
                        output logic PCJalSrcE, PCSrcE, ALUSrcAE,
                        output logic [1:0] ALUSrcBE,
                        output logic RegWriteM, RegWriteW,
                        output logic [2:0] ImmSrcD,
                        output logic [3:0] ALUControlE);

logic [1:0] ALUOpD;
logic [1:0] ResultSrcD, ResultSrcE, ResultSrcM;
logic [3:0] ALUControlD;
logic BranchD, BranchE, MemWriteD, MemWriteE, JumpD, JumpE;
logic ZeroOp, ALUSrcAD, RegWriteD, RegWriteE;
logic [1:0] ALUSrcBD;
logic SignOp;
logic BranchOp;

// main decoder
maindec md(op, ResultSrcD, MemWriteD, BranchD, ALUSrcAD, ALUSrcBD, RegWriteD,
JumpD, ImmSrcD, ALUOpD);

// alu decoder
aludec ad(op[5], funct3, funct7b5, ALUOpD, ALUControlD);


c_ID_IEx c_pipreg0(clk, reset, FlushE, RegWriteD, MemWriteD, JumpD, BranchD,
ALUSrcAD, ALUSrcBD, ResultSrcD, ALUControlD,
                                        RegWriteE, MemWriteE, JumpE, BranchE,
ALUSrcAE, ALUSrcBE, ResultSrcE, ALUControlE);
assign ResultSrcE0 = ResultSrcE[0];

c_IEx_IM c_pipreg1(clk, reset, RegWriteE, MemWriteE, ResultSrcE, RegWriteM,
MemWriteM, ResultSrcM);

c_IM_IW c_pipreg2 (clk, reset, RegWriteM, ResultSrcM, RegWriteW, ResultSrcW);


assign ZeroOp = ZeroE ^ funct3[0]; // Complements Zero flag for BNE Instruction
assign SignOp = (SignE ^ funct3[0]) ; //Complements Sign for BGE

//mux2 BranchSrc (ZeroOp, SignOp, funct3[2], BranchOp); // fix this later
assign BranchOp = funct3[2] ? (SignOp) : (ZeroOp);
assign PCSrcE = (BranchE & BranchOp) | JumpE;
assign PCJalSrcE = (op == 7'b1100111) ? 1 : 0; // jalr


endmodule
```

Figure 4.42: Implementation of *control_unit* in Verilog code

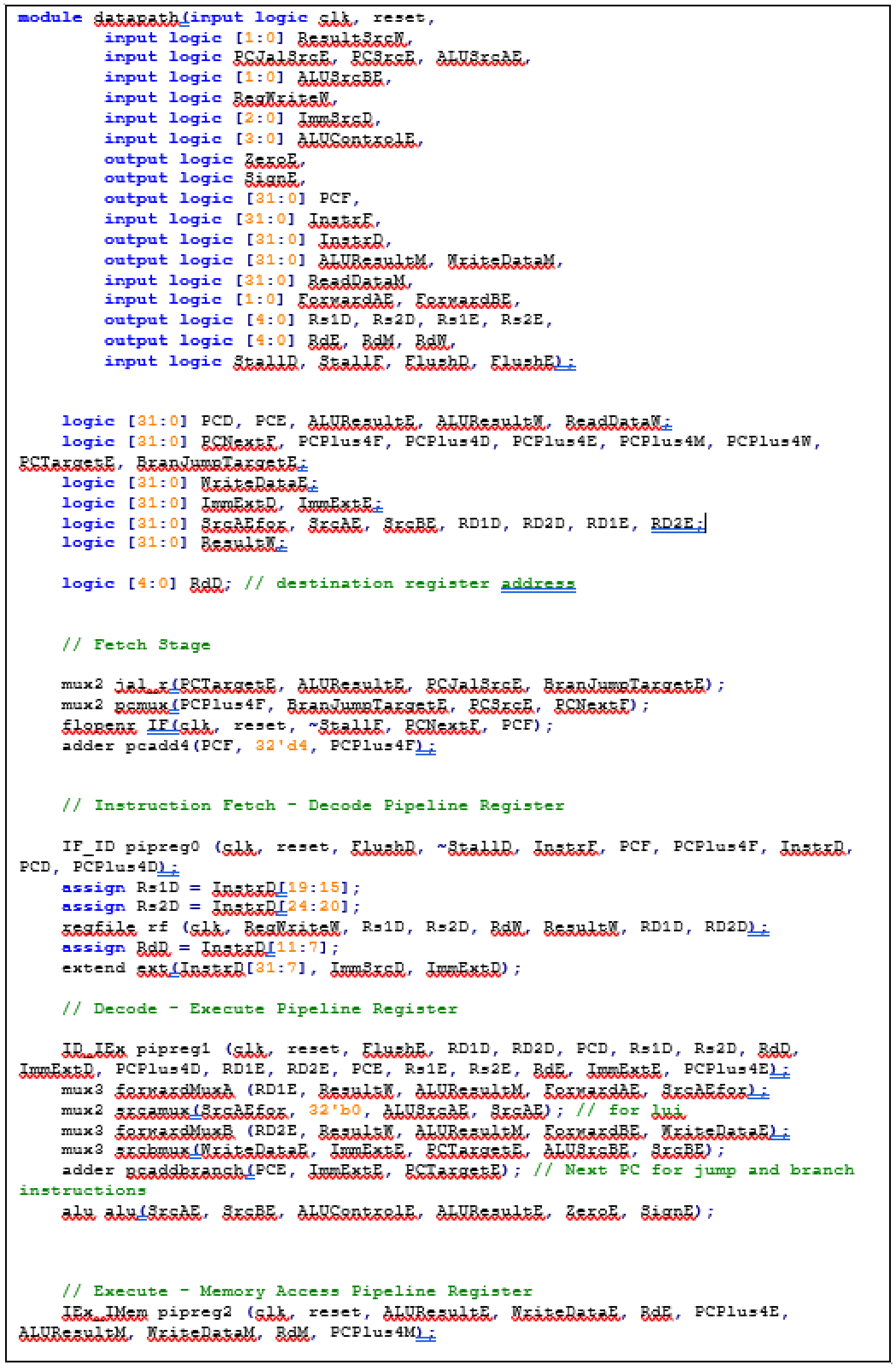
Figure 4.26 showed the Verilog code of the integrated hazard unit.



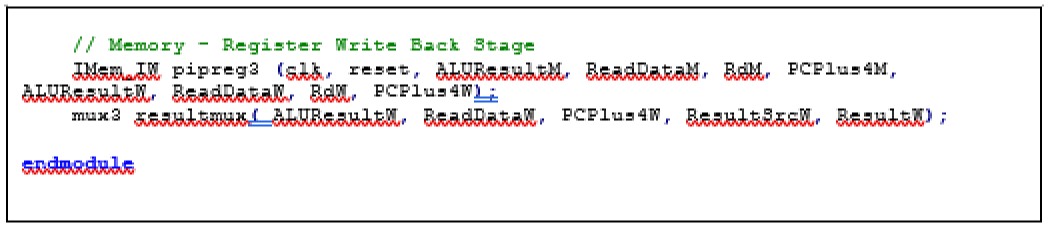
```
module hazardunit(input logic [4:0] Rs1D, Rs2D, Rs1E, Rs2E,
                 input logic [4:0] RdE, RdM, RdW,
                 input logic RegWriteM, RegWriteW,
                     input logic ResultSrcE0, PCSrcE,
                 output logic [1:0] ForwardAE, ForwardBE,
                 output logic StallD, StallF, FlushD, FlushE);

// RAW
// Whenever source register (Rs1E, Rs2E) in execution stage matchces with the
destination register (RdM, RdW)
// of a previous instruction's Memory or WriteBack stage forward the ALUResultM
or ResultW
// And also only when RegWrite is asserted

logic lwStall;
    always_comb begin
        ForwardAE = 2'b00;
         ForwardBE = 2'b00;
        if ((Rs1E == RdM) & (RegWriteM) & (Rs1E != 0)) // higher priority - most
recent
            ForwardAE = 2'b10; // for forwarding ALU Result in Memory Stage
        else if ((Rs1E == RdW) & (RegWriteW) & (Rs1E != 0))
            ForwardAE = 2'b01; // for forwarding WriteBack Stage Result



        if ((Rs2E == RdM) & (RegWriteM) & (Rs2E != 0))
            ForwardBE = 2'b10; // for forwarding ALU Result in Memory Stage

        else if ((Rs2E == RdW) & (RegWriteW) & (Rs2E != 0))
            ForwardBE = 2'b01; // for forwarding WriteBack Stage Result

      end

// For Load Word Dependency result does not appear until end of Data Memory
Access Stage
// if Destination register in EXE stage is equal to souce register in decode
stage
// stall previous instructions until the the load word is avialbe at the
writeback stage
// Introduce One cycle latency for subsequent instructions after load word
// There is two cycle difference between Memory Access and the immediate next
instruction

   assign lwStall = (ResultSrcE0 == 1) & ((RdE == Rs1D) | (RdE == Rs2D));
//    assign FlushE = lwStall;
    assign StallF = lwStall;
    assign StallD = lwStall;

// control hazard
// whenever branch has been taken, we flush the following two instructions from
decode and execute pip reg
    assign FlushE = lwStall | PCSrcE;
    assign FlushD = PCSrcE;

endmodule
```

Figure 4.43: Implementation of *hazard_unit* in Verilog code

Figure 4.27 showed the Verilog code for the integrated datapath unit.



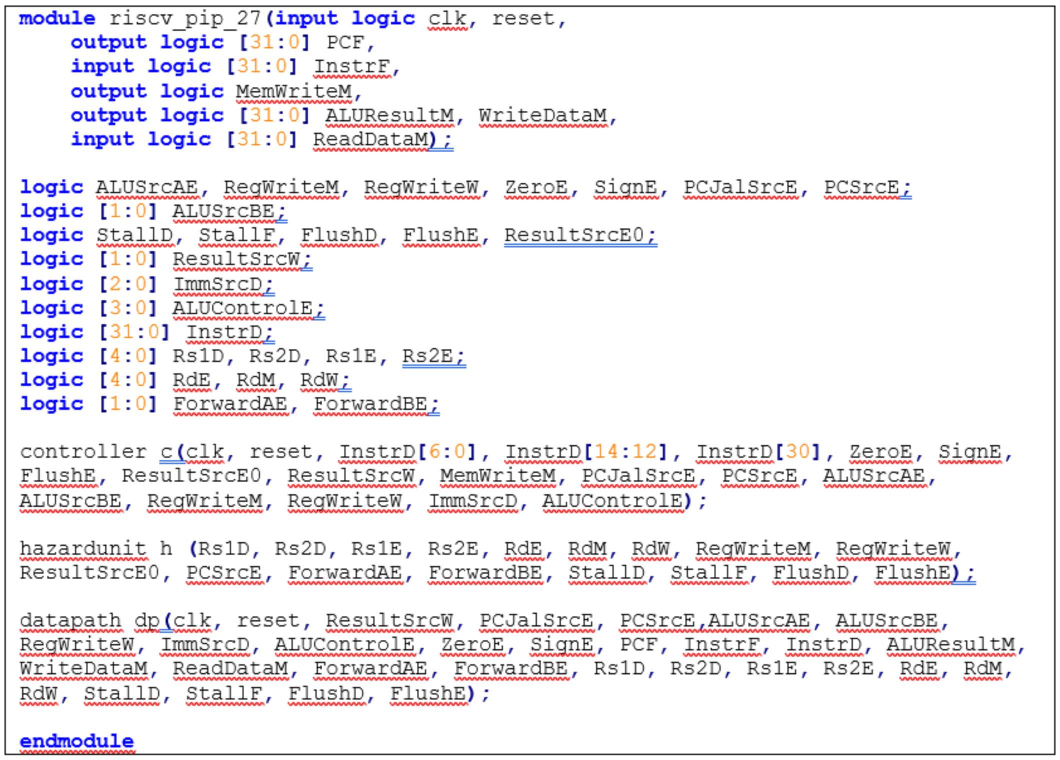
```
module datapath(input logic clk, reset,
        input logic [1:0] ResultSrcW,
        input logic PCJalSrcE, PCSrcE, ALUSrcAE,
        input logic [1:0] ALUSrcBE,
        input logic RegWriteW,
        input logic [2:0] ImmSrcD,
        input logic [3:0] ALUControlE,
        output logic ZeroE,
        output logic SignE,
        output logic [31:0] PCF,
        input logic [31:0] InstrF,
        output logic [31:0] InstrD,
        output logic [31:0] ALUResultM, WriteDataM,
        input logic [31:0] ReadDataM,
        input logic [1:0] ForwardAE, ForwardBE,
        output logic [4:0] Rs1D, Rs2D, Rs1E, Rs2E,
        output logic [4:0] RdE, RdM, RdW,
        input logic StallD, StallF, FlushD, FlushE);


    logic [31:0] PCD, PCE, ALUResultE, ALUResultW, ReadDataW;
    logic [31:0] PCNextF, PCPlus4F, PCPlus4D, PCPlus4E, PCPlus4M, PCPlus4W,
PCTargetE, BranJumpTargetE;
    logic [31:0] WriteDataE;
    logic [31:0] ImmExtD, ImmExtE;
    logic [31:0] SrcAEfor, SrcAE, SrcBE, RD1D, RD2D, RD1E, RD2E;
    logic [31:0] ResultW;

    logic [4:0] RdD; // destination register address


    // Fetch Stage

    mux2 jal_r(PCTargetE, ALUResultE, PCJalSrcE, BranJumpTargetE);
    mux2 pcmux(PCPlus4F, BranJumpTargetE, PCSrcE, PCNextF);
    flopenr IF(clk, reset, ~StallF, PCNextF, PCF);
    adder pcadd4(PCF, 32'd4, PCPlus4F);


    // Instruction Fetch - Decode Pipeline Register

    IF_ID pipreg0 (clk, reset, FlushD, ~StallD, InstrF, PCF, PCPlus4F, InstrD,
PCD, PCPlus4D);
    assign Rs1D = InstrD[19:15];
    assign Rs2D = InstrD[24:20];
    regfile rf (clk, RegWriteW, Rs1D, Rs2D, RdW, ResultW, RD1D, RD2D);
    assign RdD = InstrD[11:7];
    extend ext(InstrD[31:7], ImmSrcD, ImmExtD);

    // Decode - Execute Pipeline Register

    ID_IEx pipreg1 (clk, reset, FlushE, RD1D, RD2D, PCD, Rs1D, Rs2D, RdD,
ImmExtD, PCPlus4D, RD1E, RD2E, PCE, Rs1E, Rs2E, RdE, ImmExtE, PCPlus4E);
    mux3 forwardMuxA (RD1E, ResultW, ALUResultM, ForwardAE, SrcAEfor);
    mux2 srcamux(SrcAEfor, 32'b0, ALUSrcAE, SrcAE); // for lui
    mux3 forwardMuxB (RD2E, ResultW, ALUResultM, ForwardBE, WriteDataE);
    mux3 srcbmux(WriteDataE, ImmExtE, PCTargetE, ALUSrcBE, SrcBE);
    adder pcaddbranch(PCE, ImmExtE, PCTargetE); // Next PC for jump and branch
instructions
    alu alu(SrcAE, SrcBE, ALUControlE, ALUResultE, ZeroE, SignE);



    // Execute - Memory Access Pipeline Register
    IEx_IMem pipreg2 (clk, reset, ALUResultE, WriteDataE, RdE, PCPlus4E,
ALUResultM, WriteDataM, RdM, PCPlus4M);
```



```
    // Memory - Register Write Back Stage
    IMem_IW pipreg3 (clk, reset, ALUResultM, ReadDataM, RdM, PCPlus4M,
ALUResultW, ReadDataW, RdW, PCPlus4W);
    mux3 resultmux( ALUResultW, ReadDataW, PCPlus4W, ResultSrcW, ResultW);

endmodule
```

Figure 4.44: Implementation of *datapath_unit* in Verilog code

These 3 main units are integrated into a main module to form a *riscv_pip_27* module. Figure 4.28 shows the Verilog code of the integrated *riscv_pip_27* module.



```
module riscv_pip_27(input logic clk, reset,
    output logic [31:0] PCF,
    input logic [31:0] InstrF,
    output logic MemWriteM,
    output logic [31:0] ALUResultM, WriteDataM,
    input logic [31:0] ReadDataM);

logic ALUSrcAE, RegWriteM, RegWriteW, ZeroE, SignE, PCJalSrcE, PCSrcE;
logic [1:0] ALUSrcBE;
logic StallD, StallF, FlushD, FlushE, ResultSrcE0;
logic [1:0] ResultSrcW;
logic [2:0] ImmSrcD;
logic [3:0] ALUControlE;
logic [31:0] InstrD;
logic [4:0] Rs1D, Rs2D, Rs1E, Rs2E;
logic [4:0] RdE, RdM, RdW;
logic [1:0] ForwardAE, ForwardBE;

controller c(clk, reset, InstrD[6:0], InstrD[14:12], InstrD[30], ZeroE, SignE,
FlushE, ResultSrcE0, ResultSrcW, MemWriteM, PCJalSrcE, PCSrcE, ALUSrcAE,
ALUSrcBE, RegWriteM, RegWriteW, ImmSrcD, ALUControlE);

hazardunit h (Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW, RegWriteM, RegWriteW,
ResultSrcE0, PCSrcE, ForwardAE, ForwardBE, StallD, StallF, FlushD, FlushE);

datapath dp(clk, reset, ResultSrcW, PCJalSrcE, PCSrcE,ALUSrcAE, ALUSrcBE,
RegWriteW, ImmSrcD, ALUControlE, ZeroE, SignE, PCF, InstrF, InstrD, ALUResultM,
WriteDataM, ReadDataM, ForwardAE, ForwardBE, Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM,
RdW, StallD, StallF, FlushD, FlushE);

endmodule
```

Figure 4.45: Implementation of *riscv_pip_27* in Verilog code

Then, the *riscv_pip_27* module is integrated with the *imem* and *dmem* module to form the *top* design. The schematic of the *top* design is illustrated in Figure 4.29.

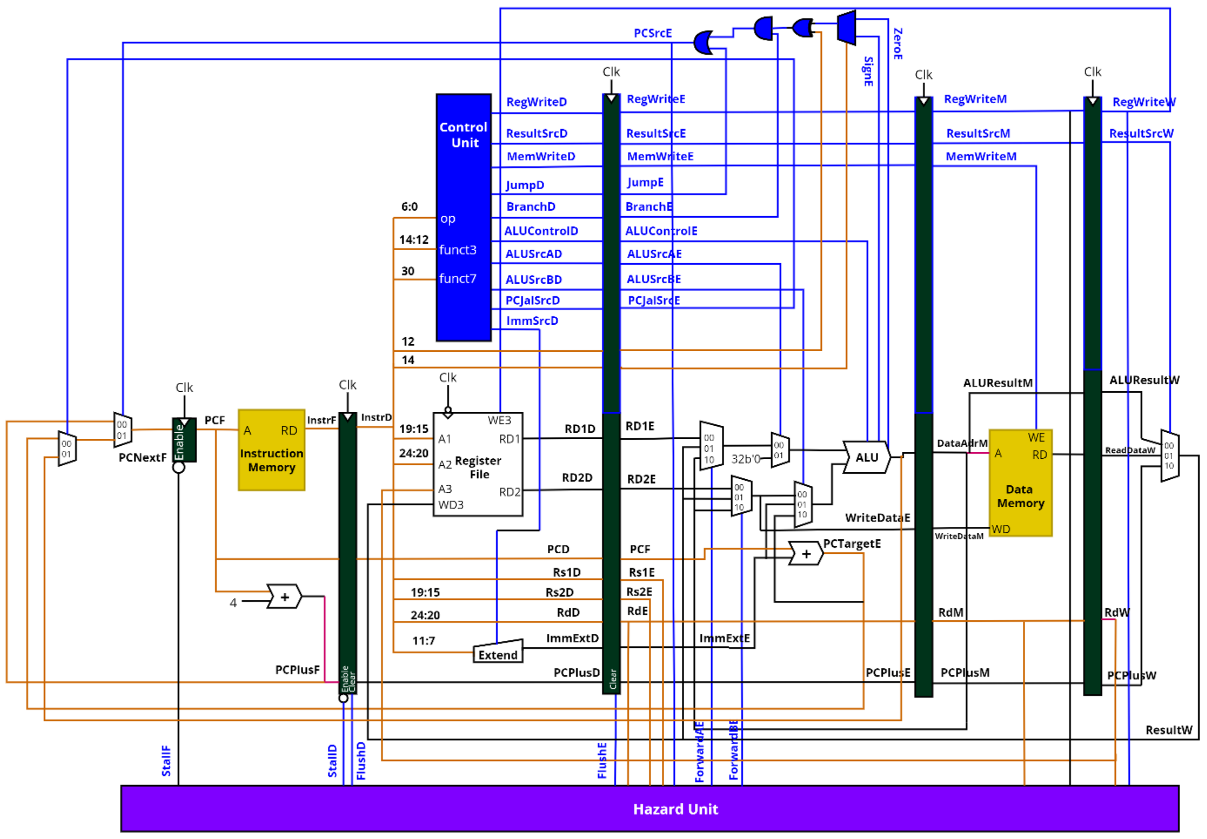
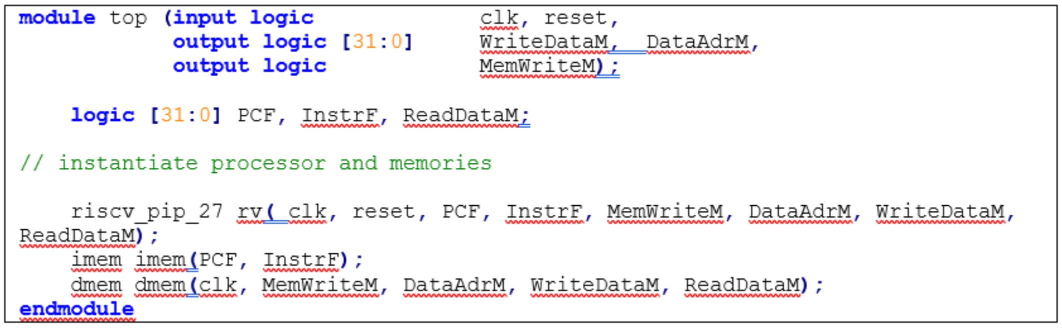


Figure 4.46: Schematic diagram of *top* design

Figure 4.30 showed the Verilog code for top design.

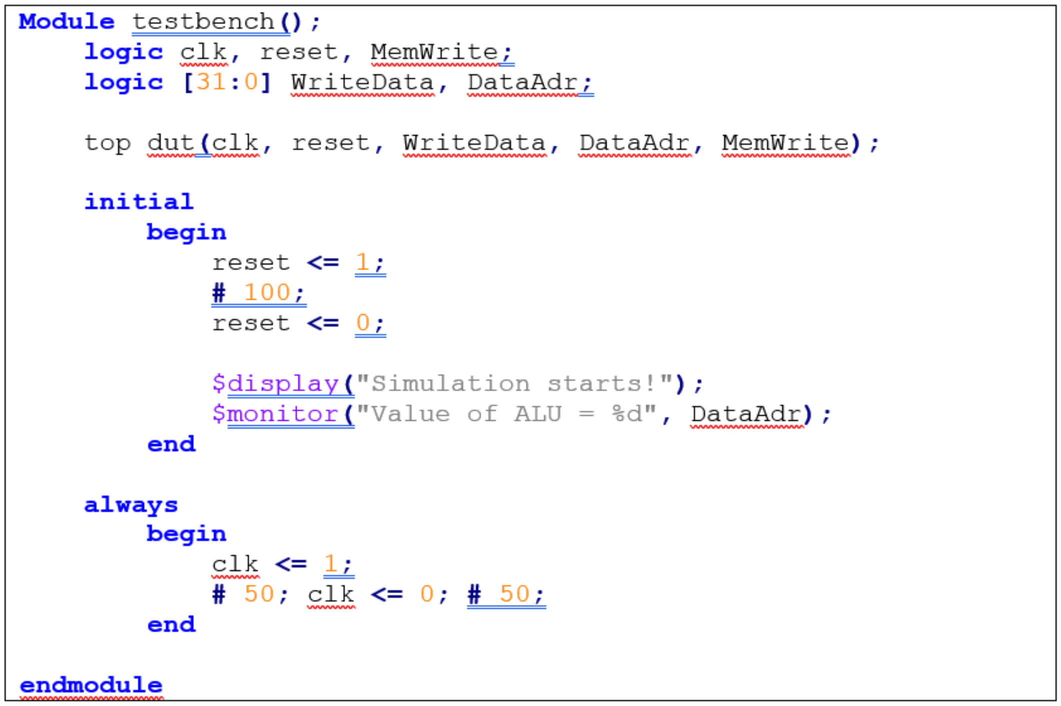


```
module top (input logic              clk, reset,
            output logic [31:0]      WriteDataM,  DataAdrM,
            output logic             MemWriteM);

    logic [31:0] PCF, InstrF, ReadDataM;

// instantiate processor and memories

    riscv_pip_27 rv( clk, reset, PCF, InstrF, MemWriteM, DataAdrM, WriteDataM,
ReadDataM);
    imem imem(PCF, InstrF);
    dmem dmem(clk, MemWriteM, DataAdrM, WriteDataM, ReadDataM);
endmodule
```

Figure 4.47: Implementation of *top* design in Verilog code

4.4 Testbench simulation of the top design

A simple testbench module is written to simulate the *top* design. The Verilog code for the testbench module is illustrated in Figure 4.31.

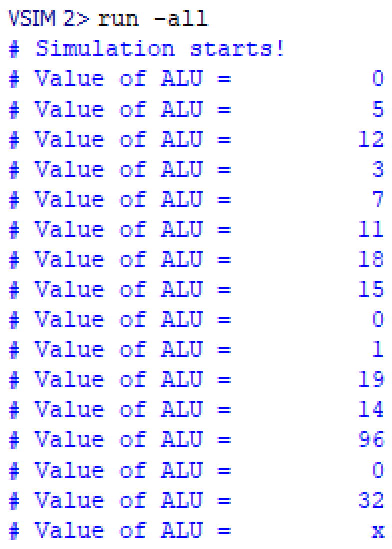


```
Module testbench();
     logic clk, reset, MemWrite;
     logic [31:0] WriteData, DataAdr;

     top dut(clk, reset, WriteData, DataAdr, MemWrite);

     initial
         begin
             reset <= 1;
             # 100;
             reset <= 0;

             $display("Simulation starts!");
             $monitor("Value of ALU = %d", DataAdr);
         end

     always
         begin
             clk <= 1;
             # 50; clk <= 0; # 50;
         end

endmodule
```

Figure 4.48: Testbench for *top* design

The focus of the testbench is to check the ALU value for each clock cycle. In other words, it is also checking the value of the destination register (*rd*) for each instruction after they were executed. Form Figure xxx, the clock cycle for the testbench is set to 100 ps. *$monitor* function is used in the testbench to print the value of *DataAdr* when the value is changed. The value of *DataAdr* is obtained from *IEx/IMem* pipeline register. Besides, it also acted as an input to the *dmem* module.

The simulation result of the testbench is shown in Figure 4.32.



```
VSIM 2> run -all
# Simulation starts!
# Value of ALU =          0
# Value of ALU =          5
# Value of ALU =         12
# Value of ALU =          3
# Value of ALU =          7
# Value of ALU =         11
# Value of ALU =         18
# Value of ALU =         15
# Value of ALU =          0
# Value of ALU =          1
# Value of ALU =         19
# Value of ALU =         14
# Value of ALU =         96
# Value of ALU =          0
# Value of ALU =         32
# Value of ALU =          x
```

Figure 4.49: Simulation results of the testbench

From Figure 4.32, the results shown is based on the test sequence in Figure 4.1. The first value shown is 0 because that is the initialized value for the ALU. Then, after the first instruction is executed (addi x2, x0, 5), the value of the ALU shown is 5. This is because, the output of this instruction is 5. Then, the second instruction (addi x3, x0, 12) has an output of value 12 and stored in the destination register (*x3*).

Therefore, the next value shown from the simulation is 12. Moreover, for third instruction (*addi x7, x3, -9*), there is a source register (*x3*) which is dependent on the destination register from the second register. The output for the third instruction is value 3. The output is expected because the value of register *x3* is forwarded and updated in second instruction before being used by the third instruction. This forwarding feature was done by the hazard unit module. The rest of the values shown are the output of the respective instructions. The functionality of the top design is verified. Hence, the design of the 32-bit 5 stage pipeline RISC-V processor is completed.

Chapter 5: Conclusion and Future work

In this project, a 32-bit 5 stage pipeline RISC-V processor is designed and implemented using Verilog coding. The top design consists of 3 main modules that are *riscv_pip_27*, *imem* and *dmem*. *riscv_pip_27* module is an integrated module forming by *control_unit*, *hazard_unit*, and *datapath_unit* module. The *control_unit* module is used to send out control signals based on different instruction being executed. Then, the *hazard_unit* module is used to detect different hazard conditions during execution and mitigate them. Next, the *datapath_unit* module is used to carry information in the datapath through pipeline registers. The *imem* module is used to store the instruction of a program whereas the *dmem* is responsible for storing and retrieving data.

Besides, several important modules such as *alu*, *aludec*, *maindec*, *regfile*, *result_mux*, *pc_mux*, pipeline register (*IF/ID, ID/IEx, IEx/IMem,* and *IMem/IW*), *forwardMuxA* and *forwardMuxB* were also designed and integrated in the processor. The functionality of these modules were tabulated in table 5.1.

Table 5.1: Table of modules and their functionality

Module	Functionality
alu	Performing arithmetic and logical operations
aludec	Decode the instructions and receive signal from *maindec* to determine the type of operations that had the be performed
maindec	Generate control signal from the *Opcode* to determine types of instruction

regfile	Storing and manipulating data during program execution
result_mux	Select *ALUResults, ReadData,* or *PCPlus* as output
pc_mux	Select *PCPlus4* or *JumpTarget* to be executed
IF/ID	Store data from fetch stage and load them in decode stage
ID/IEx	Store data from decode stage and load them in execute stage
IEx/IMem	Store data from execute stage and load them in memory stage
IMem/IW	Store data from memory stage and load them in writeback stage
forwardMuxA	Handle following 2 hazard conditions 1. EX/MEM.RegisterRd = ID/EX.RegisterRs 2. MEM/WB.RegisterRd = ID/EX.RegisterRs
forwardMuxB	Handle following 2 hazard conditions 1. EX/MEM.RegisterRd = ID/EX.RegisterRt 2. MEM/WB.RegisterRd = ID/EX.RegisterRt

The functionality of these modules were and verified analyzing the waveform generate by using ModelSim software. Besides, a testbench for *top* design were written to verify the overall functionality of the 32-bit 5 stage pipeline RISC-V processor.

However, there are some limitations in our RISC-V processor which can be further improved in the future.

1. Implementing different extensions to the base integer instruction set

   RISC-V had standardized a series of extensions that provide additional functionality beyond the base integer instructions, such as floating-point arithmetic, bit manipulation, vector operations and cryptography. These extensions can be implemented or omitted depending on the design goals and application requirements.

2. Improving the branch prediction accuracy and reducing the branch penalty

   Branch prediction is a technique to guess the outcome of a conditional branch instruction before it is executed. This allows the processor to fetch and execute instructions from the predicted branch without waiting for the actual branch instruction to be resolved. However, if the prediction is wrong, the processor had to flush the pipeline and fetch instructions from the correct branch, which causes a performance penalty. To improve the branch prediction accuracy and reduce branch penalty, techniques such as static branch prediction, dynamic branch prediction and branch history table can be used.

3. Exploring different cache architecture and memory hierarchies

   Cache is a small and fast memory that store frequently accessed data from the main memory. Memory hierarchy is a system of multiple level of memory with different sizes and speed. The goal of cache architecture and memory hierarchy design is to reduce the average memory access time and increase the memory bandwidth. To achieve this goal, different aspect such as cache size, cache organization and cache mapping can be explored.

Reference

1. Urquhart, Roddy (29 March 2021). "What Does RISC-V Stand For? A brief history of the open ISA". *Systems & Design: Opinion*. Semiconductor Engineering.
2. D. Bhandarkar and D.W. Clark, "Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization," Proceedings of the 4th Int'l. Conference on ASPLOS, Santa Clara, California, April 8-11, 1991.
3. M. N. Topiwala and N. Saraswathi, "Implementation of a 32-bit MIPS based RISC processor using Cadence," *2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies*, 2014, pp. 979-983, doi: 10.1109/ICACCCT.2014.7019240.
4. Kulshreshtha, A., Moudgil, A., Chaurasia, A. and Bhushan, B., 2021, March. Analysis of 16-Bit and 32-Bit RISC Processors. In *2021 7th International Conference on Advanced Computing and Communication Systems (ICACCS)* (Vol. 1, pp. 1318-1324). IEEE.
5. Islam, S., Chattopadhyay, D., Das, M.K., Neelima, V. and Sarkar, R., 2006, September. Design of High-Speed-Pipelined Execution Unit of 32-bit RISC Processor. In *2006 Annual IEEE India Conference* (pp. 1-5). IEEE.
6. Khairullah, S.S., 2022, June. Realization of a 16-bit MIPS RISC pipeline processor. In *2022 International Congress on Human-Computer*

Interaction, Optimization and Robotic Applications (HORA) (pp. 1-6). IEEE.

7. Al-sudany, S.M., Al-Araji, A.S. and Saeed, B.M., 2021. FPGA-Based Multi-Core MIPS Processor Design. *IRAQI JOURNAL OF COMPUTERS, COMMUNICATION, CONTROL & SYSTEMS ENGINEERING*, *21*(2).
8. Wang, W., Han, J., Cheng, X. and Zeng, X., 2021. An energy-efficient cryptoextension design for RISC-V. Microelectronics Journal, 115, p.105165