Design of a Low Power Minimal Core Processor with Robust Pipeline stages for Error-Prone Application

Post Graduate Thesis

Submitted in partial fulfillment of the requirements of
BITS G629T Thesis

By

Aatib Mohammad
ID No. 2022H1230239P

Under the supervision of:

Dr. Nitin Chaturvedi
&
Dr. Chandra Sekhar

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI CAMPUS
May 2024

Declaration of Authorship

I, Aatib Mohammad, declare that this Post Graduate Thesis titled, 'Design of a Low Power Minimal Core Processor with Robust Pipeline stages for Error-Prone Application' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ______________________________

Date: ______________________________

Certificate

This is to certify that the thesis entitled, "*Design of a Low Power Minimal Core Processor with Robust Pipeline stages for Error-Prone Application*" and submitted by Aatib MOHAMMAD ID No. 2022H1230239P in partial fulfillment of the requirements of BITS G629T Thesis embodies the work done by him under my supervision.

Supervisor	*Co-Supervisor*
Dr. Nitin CHATURVEDI	Dr. Chandra SEKHAR
Associate Professor,	Senior Professor Emeritus,
BITS-Pilani Pilani Campus	BITS-Pilani Pilani Campus
Date:	Date:

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI CAMPUS

Abstract

Master of Engineering

Design of a Low Power Minimal Core Processor with Robust Pipeline stages for Error-Prone Application

by Aatib Mohammad

The processor is a crucial component of all computational applications. The RV32I base integer ISA is a popular choice due to its ease of use with compilers and support for modern OS environments. This thesis presents the design and implementation of RV32I for low-power, error-prone applications. The RV32I base 32-bit integer instruction set includes 47 instructions that are executed in 5 stages: fetch, decode, execute, memory, and writeback. This design addresses all types of hazards, including data hazards, structure hazards, and control hazards, using data forwarding and bubble insertion techniques.

Acknowledgements

I would like to express my gratitude to my mentor and guide, Dr Chandra Sekhar Sir, senior professor emeritus, and Dr. Nitin Chaturvedi, professor at the Department of Electrical and Electronics Engineering, for guiding me and answering all of my questions patiently.

His passion for teaching and dedication to the work will always inspire me. I would also like to thank respected Faculty Members and the institute Birla Institute of Technology and Science, Pilani for giving me support and motivation to finish the work on time

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As Moore's law is approaching its limit, which results in no more decrease in transistor size and threshold voltages, semiconductor designs have a higher risk of failure and increased error. Now, it's very difficult to maintain an error-free architecture. These physical property limitations have caused errors like depletion at the polysilicon gate, roll-off threshold voltage, DIBL (drain-induced barrier lowering), velocity saturation, rise in reverse leakage current, reduction in mobility, and hot carrier effects. This effect leads to process variation, the aging effect, noise margin reduction, and ease of proneness to soft errors.

Applications can tolerate some errors, but this is not true for each instruction within the application. Critical instructions can have a devastating effect on the execution of an application, even if the application is designed to tolerate errors. For example, if a memory access instruction points to a disallowed location due to a corrupted address, it may cause segmentation faults. Similarly, the program may hang or loop indefinitely if the control flow is corrupted. An instruction or location in an application is said to be error-intolerant if its corruption can result in a catastrophic failure, such as a crash or hang. All memory addressing and loop back-edges are considered error-intolerant to prevent a program from crashing due to segmentation faults or becoming unresponsive. Consequently, all control and data dependencies for intolerant instructions must also be considered error-intolerant.

Some techniques to improve reliability and reduce the likelihood of errors are redundancy, Error checking with recovery, and Voltage scaling. If a system has many hardware errors, which require many repetitive algorithms (loops), it can't be solved with software techniques. This system requires a reliable core that can easily partitioned to one control and many worker threads. Enejy and Flicker have proposed a distinct system to annotate the variable and divide the computational data into "Error-prone/Low Power" and "Error-free/High Power." Components that are prone to error will run error-tolerant instructions and error-tolerant data. Components that are not prone to error will run error-intolerant instructions and error-intolerant data. It increases the reliability of the system, but it reduces the efficiency because less than half of the instructions are immune to error, and less than half of the core is prone to error. Different techniques like redundancy of modules, error checking with recovery, and voltage scaling will reduce the frequency of error, but this technique is difficult to afford. Still, there are applications like media that can tolerate computation errors and provide accuracy. This method has forced us to explore error-tolerant architectures for both chips and peripherals having the same experimental results. These experimental-based results are used to apply specific hardware solutions. To ensure the smooth operation of an application that is prone to errors, it is crucial to have a solid understanding of its specific requirements. This involves identifying their functionalities, performance targets, and the errors which are very common. After a thorough understanding of the requirements of an application, it is possible to develop a minimalist Instruction Set Architecture (ISA) that is customized to meet those requirements. The focus of such a design should be on simplicity and efficiency to minimize the possibility of errors during instruction execution. To further improve the error detection and correction mechanisms, various techniques such as parity checks, checksums, or cyclic redundancy checks (CRC) can be incorporated into the processor's design. Error Correction Codes (ECC) can also be implemented to detect and correct errors in data transmission and storage, depending on the level of error correction required. Techniques such as Hamming codes, Reed-Solomon codes, or BCH codes can be used for this purpose. To reduce the propagation of errors, a pipeline with minimal stages should be designed. Hazard detection and handling mechanisms should also be implemented to ensure correct instruction execution despite pipeline hazards. The processor design should have a robust control logic that includes fault-tolerance features to mitigate the impact of hardware faults and transient errors. These features may include redundant components, error recovery circuits, and error masking techniques. To ensure the effectiveness of the processor design, thorough testing is essential. Simulation, emulation, and hardware testing platforms can be used for comprehensive testing. Additionally, fault injection techniques should be used to evaluate the effectiveness of error detection and correction mechanisms under various error scenarios. The processor design should be documented comprehensively, including error-handling strategies, fault tolerance mechanisms, and operational constraints. Monitoring features should also be implemented to detect and log errors during runtime for diagnostic analysis. To optimize the processor design for power efficiency and area utilization, low-power design techniques and trade-offs between hardware complexity and error resilience should be considered.

The design of the processor should be robust enough to endure environmental factors such as changes in temperature, voltage fluctuations, and radiation effects. This can be achieved by selecting components and materials that have high reliability and by incorporating design margins that account for variability in the operating conditions. Additionally, the processor design should be continuously improved by incorporating feedback from testing and real-world deployment. Any performance bottlenecks, reliability issues, or areas for improvement must be identified and addressed to enhance overall system reliability and effectiveness. With these steps in place, you can be confident that the error-prone application will run smoothly and reliably.

Chapter 2

RV32I Base Integer ISA

2.1 Introduction

The RV32I base integer ISA was developed with easier compiler targets and seamless support for modern OS environments in mind. Further, it drastically reduces hardware requirements for a minimal implementation. RISC-V has many extensions, and the RV32I can be used to emulate other extensions apart from the A-extension since it requires extra hardware to support atomicity.

The RISC-V instructions can be summed up into four core instruction formats: R, I, S, and U. However, there are six formats: R-type, I-type, S-type, B-type, U-type, and J-type. The four aforementioned fundamental instruction types constitute the basis for B-type and J-type. Each is explained in the later sections. All the instructions should be aligned in memory on a four-byte boundary in the little-endian format. If there are instructions that have a reduced length of 16 bits, the alignment constraint will then decrease to a two-byte boundary. The reason for the little-endian is that it is quite popular and common, which reduces the time needed to port low-level software.

An exception should be raised in the event of an address misalignment involving a branch address or a jump address. If a conditional branch is not taken, no exception is generated. Every instruction in our present implementation of the base RV32I implementation is 32 bits wide and aligned on a two-byte boundary. In the future, if necessary, this alignment will enable the insertion of 16-bit instructions. We have permitted the fetching of 32-bit in a single cycle for the time being because all of our instructions are 32-bit wide.

2.2 Instruction length encoding

The Programmer's model for this base subset contains 32 registers, each 32-bit wide, of which x1-x31 are general-purpose registers, and a register x0 is hardwired to constant 0. Registers x1-x31 hold integer values. The RV32I base 32-bit integer instruction set consists of 47 instructions.

Among the 47 instructions, eight are associated with the system, i.e., they perform system calls and play a decisive role in performance. The remaining 40 instructions play a crucial role in the calculation, flow of control between registers, and accesses pertaining to memory.

2.3 RV32I base instruction set

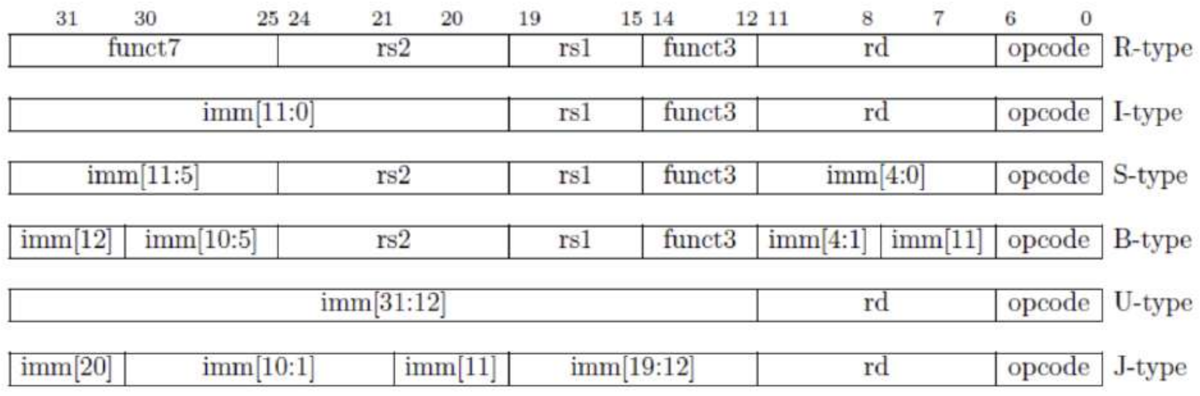

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type
imm[31:12] | rd | opcode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

FIGURE 2.1: Base Integer ISA Encoding

Every instruction in RISC-V has specific fixed fields encoded. The length of instruction, however, is the same. Some salient points regarding the ISA are stated below:

- 7 bits are allocated for the opcode.
- 10 bits for source registers in case of R-type instructions; otherwise, 5 bits for the remainder of the instruction are allocated to the immediate field.
- 3 bits additionally inform the functionality. Most instructions share opcodes, reducing the time to decode the instruction. However, the "funct3" field allows us to differentiate one instruction from the other.
- 5 bits represent the destination register in the case of R-type instruction. Otherwise, the concluding 12 bits are condensed to form the immediate field. This immediate field aids in the computation of memory address in case of unconditional jump or branch. However, these 12 bits are sometimes encoded in different fields, so it is necessary to arrange them.

2.3.1 RISC-V Immediate Encoding

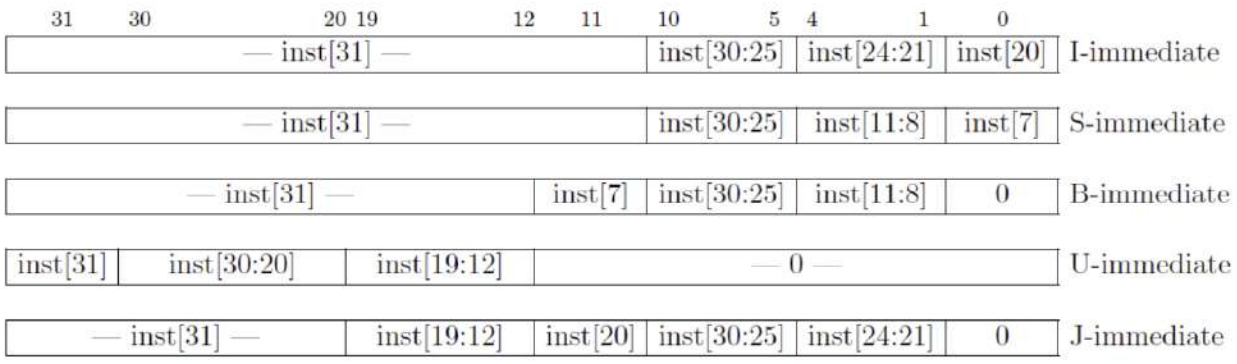


Figure 2.2: Immediate encoding

The picture above depicts which instruction bit of the 32-bit instruction is used to generate each bit of the immediate value. One of the most critical tasks in this architecture is sign extension. The bit number 31 (inst[31]) is utilised for sign extension in this design. In RISC-V, the instruction's sign bit is kept at bit 31. It allows sign-extension to proceed concurrently with instruction decoding. Even though the bits are scrambled, compilation takes no longer than usual.

Two more variations of the instruction format, known as B-type and J-type, have resulted from the immediate handling of this ISA. Twelve-bit immediate bits are used in the B-type instruction format to encode branch offsets in multiples of two. Specifically, we take the instruction's twelve immediate bits and insert a zero at the end. The remaining nineteen bits are then sign-extended with bit number 31. Similar to how U and J formats differ from one another, the immediate [31:12] in instruction in U-type is left-shifted by 12 bits, but in J-type, it is shifted by a bit, and the remaining trailing bits are sign-extended.

2.3.2 RISC-V Instruction Formats

R-type Format

R-type format or integer register-to-register operations consist of all arithmetic operations and all shift operations. Here, operations are performed on register contents(rs1 and rs2), and the result is stored in a destination register(rd).

The opcode for the R-type format is 7'b0110011. ADD performs addition, and SUB performs subtraction. The 32-bit result is stored from LSB onwards. Any carry from the MSB originating from the result is discarded.

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

Figure 2.3: R-type Instruction Format

I-type Format

I-type format or integer register - immediate instruction consists of all arithmetic operations except subtraction. Here, the operation is performed on a source register(rs1) and a 32-bit sign-extended immediate, and the result is stored in a destination register(rd). I-type format consists of variety of instructions which perform arithmetic, shift, logical, control transfer, loading of data from memory.

I-type Arithmetic Format

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

Figure 2.4: I type Arithmetic format

The opcode for the I-type arithmetic format is 7'b0010011. The instructions are similar to those of the R-type, the only difference being the immediate value instead of a register. The 32-bit result is stored from LSB onwards. Any carry from the MSB originating from the result is discarded.

I-type Shift Format

The opcode for the I-type shift format is 7'b0010011. The instructions are similar to those of the R-type, the only difference being the immediate value instead of a register.

I-type Load Format

The opcode for the I-type load format is 7'b0000011. The instructions facilitate to load memory content into a register to perform operations. As RISC-V is a load - store architecture, this

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

FIGURE 2.5: I type shift format

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

FIGURE 2.6: I type load format

is one of the instructions that provide access to the data Memory. Based on the I-type Load format, we have five instructions: LB, LH, LW, LBU, and LHU.

I-type JALR Format

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

FIGURE 2.7: I type JALR format

The opcode for the I-type JALR format is 7'b1100111. This format enables the unconditional control transfer of program execution. The jump address is calculated using the sign-extended 12-bit immediate, and it is then added to the contents of source register rs1. The result of the operation is then bitwise AND with 0xFFFFFFFE to align the address on a two-byte boundary. It is used generally for subroutine calls, and the return address (PC+4) is stored in the destination register rd.

S-TYPE

The opcode for the I-type load format is 7'b0100011. The instructions facilitate to store the results of a register in memory. As RISC-V is a load-store architecture, this is one of the

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

FIGURE 2.8: S type format

instructions that provide access to the data Memory. Based on the S-type format, we have three instructions: SB, SH, and SW.

B-TYPE FORMAT

The opcode for the B-type format is 7'b1100011. This format enables the conditional control transfer or conditional jump of program execution. The decision to jump is based on the equality check of the contents of registers rs1 and rs2. The jump address is calculated using the sign-extended 12-bit immediate, which encodes an offset of 2, and it is then added to the current PC value. It is also called PC-Relative Addressing. This results in the alignment of the jump address on the two-byte boundary. The conditional branch range comes out to be 4KiB. This instruction is used for conditional for, while, and other conditional loops in high-level languages. Based on the format, it has six instructions: BEQ, BNE, BLT, BGE, BLTU, and BGEU.

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

FIGURE 2.9: B type format

U-type Format

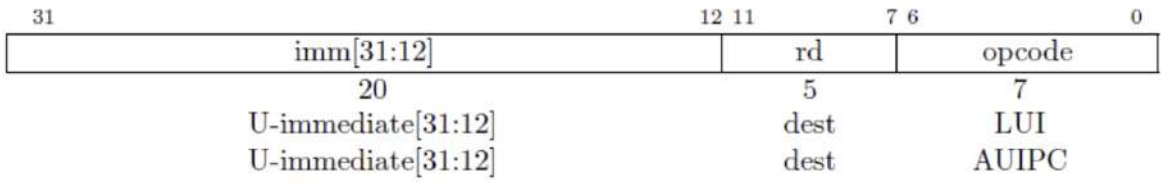
There are two instructions based on the U-type format: LUI and AUIPC.

U-type LUI format

LUI (Load Upper Immediate) is an instruction that enables user to form 32-bit constants. This instruction provides for a 20 bit immediate. The immediate is loaded into the upper 20 bits of a 32 bit destination register (rd), and the rest of the 12 bits are filled with zeros.

U-type AUIPC format

AUIPC (Add Upper Immediate to PC) allows the user to build a higher PC-Relative Addresses. In this the 20 bit immediate is sign-extended to 32 bits and added to PC And the corresponding value is saved in a destination register (rd).



31 — 12	11 — 7	6 — 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

Figure 2.10: U type format

Chapter 3

Design Methodology of RISC processor

3.1 Single Cycle Implementation:

Single cycle implementation involves a processor design strategy wherein every instruction necessitates just one clock cycle for processing. This methodology simplifies the control logic of the processor to ensure uniform execution time for each instruction. Typically, the duration of this clock cycle is dictated by the slowest instruction within the instruction set. The clock per instruction for single cycle implementation is one.

$$CPI = \frac{Clock}{Instruction} = 1 \tag{3.1}$$

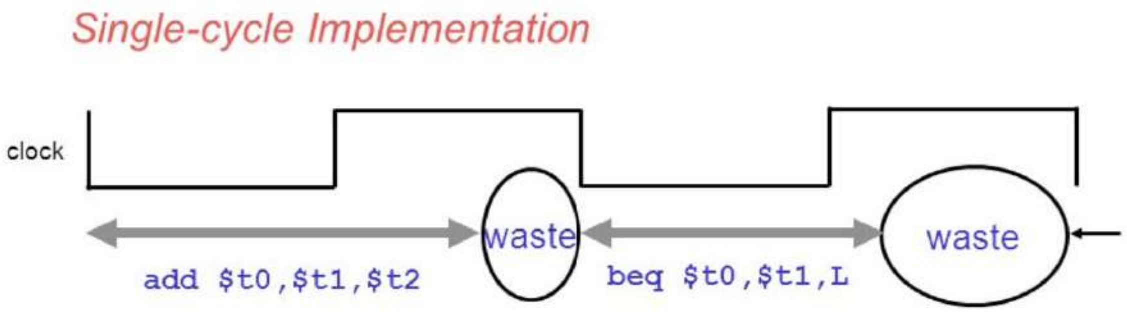


FIGURE 3.1: Single cycle approach

3.1.1 Features of Single Cycle Implementation:

- Each instruction takes exactly one clock cycle to be performed
- The control logic is designed to be straightforward in a simple decoded format. The generated control signals direct the data flow within the datapath.
- The instruction clock period is determined by the time taken for the execution of the slowest instruction in the instruction set. This limits the flexibility that RISC processors have to make a standardized approach to the implementation.
- The instruction clock cycle will now increase as well keep on increasing the ISA complexity and adding more complex instructions.
- The clock frequency of a processor implementing single cycle design is determined by the longest path through the processor's logic. This means that the clock frequency may be limited by the critical path, potentially reducing overall performance compared to designs with shorter critical paths.

3.1.2 Drawbacks of single cycle approach:

- **Wasted clock cycles**: Its possible a instruction may not require some portion of the clock period for execution and is already done with its execution but the data must wait to be ready for the remaining part of the clock cycle even though its execution is done.
- **Complex control logic**: Depending on the instruction, they may require complex control logic so as to execute within the same clock cycle. This need for extra control logic disrupts the uniformity of designing the control logic for other instructions, thus making the control logic dependent on the instruction rather than the hardware implementation.
- **Resource underutilization**: Certain hardware may not be utilised depending the type of instruction. For example, a memory stage is not required for the execution of register writeback instructions like R-type or I-type, therefore they are not utilised in a single cycle implementation and being a waste of resources.
- **Clock frequency constraints**: The critical path of the largest delay path determines the clock cycle of the processor thereby limiting the architecture to be constrained by the hardware implementation.

3.2 Multicycle Approach

In the multicycle design approach we break down the instruction into multiple stages where each stage takes one clock cycle for execution. It takes multiple cycles for the execution of a single instruction therefore the CPI for multicycle processors is greater than one. For a five stage multicycle design there are 5 stages each requiring one clock cycle for execution, the stage being Fetch, Decode, Execution, Memory access and writeback. Assuming each stage requires 1 clock cycle the CPI for a five stage multicycle processor would be 5.

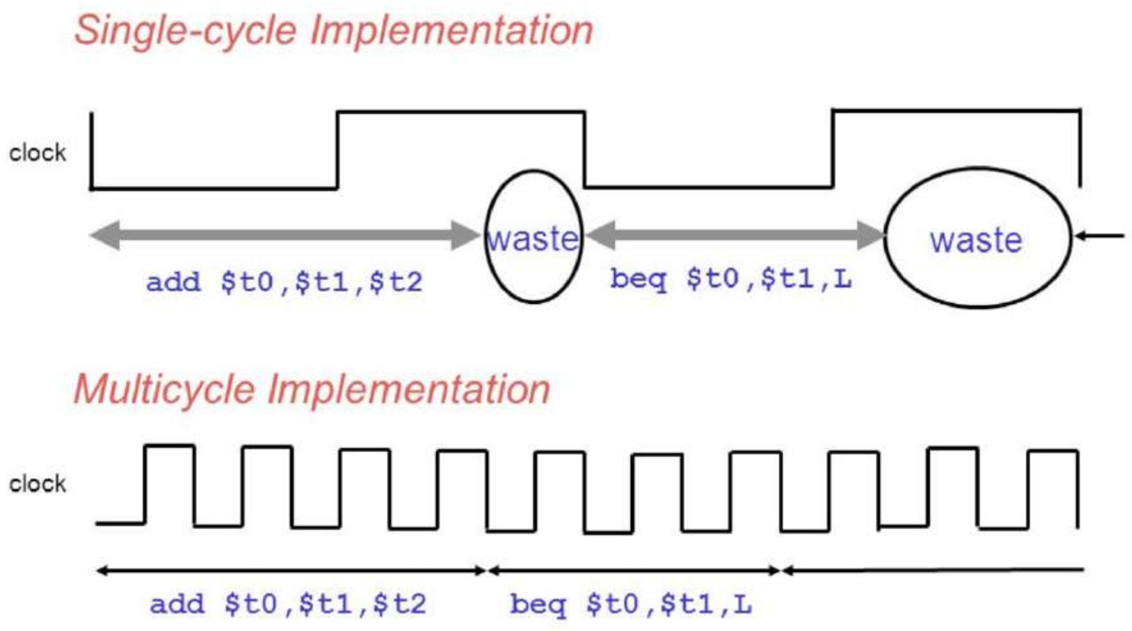


FIGURE 3.2: Multi cycle approach

$$CPI = \frac{Clock}{Instruction} > 1 \tag{3.2}$$

3.2.1 Features of Multicycle:

- **Different execution time**: Depending on the type of instruction, they will have different execution times since some instructions may require fewer clock cycles than others. For eg: a register writeback instruction will go through 4 stages - Fetch, decode, execution, writeback. Whereas a load instruction will go through 5 stages - fetch, decode, execution, memory access and register writeback.
- **Complex Control Logic**: To handle data dependencies and allow for proper flow of data and instruction, a more complex control logic is required compared to that of a single cycle design approach.
- **Higher Clock Frequency**: By dividing the critical path into multiple smaller stages the clock frequency can be improved since now the clock period is determined by the stage which requires the most time. This significantly improves the performance of of processor.

- **Improved Efficiency**: Multi-cycle implementation are much efficient compared to single-cycle implementation in terms of performance and speed, especially in cases with processors with complex instruction sets or instructions which have changing execution times. The hardware utilization is also much more efficient in multicycle approach compared to a single design approach.

3.2.2 Drawback of Multicycle approach:

- **Complex control logic** - This complexity arises from the need to manage the timing of each instruction phase and ensure proper sequencing of operations.
- **Hardware resource underutilisation** - In a multi-cycle implementation, hardware resources such as functional units, registers, and buses may not be fully utilized in each clock cycle. This underutilization can result in inefficient use of hardware resources and may limit the overall performance gains that can be achieved with a multi-cycle design. Different instructions require different stages of execution. Eg: - A memory load/store instruction goes through fetch, decode, execute and memory R/W stages and does not require the register writeback stage leaving the register file not utilized for the clock cycle. Similarly a register writeback instruction may not require a memory read/write and therefore leaving the memory stage as being not utilized.
- Longer execution time for some instructions

3.3 Pipelined Design approach

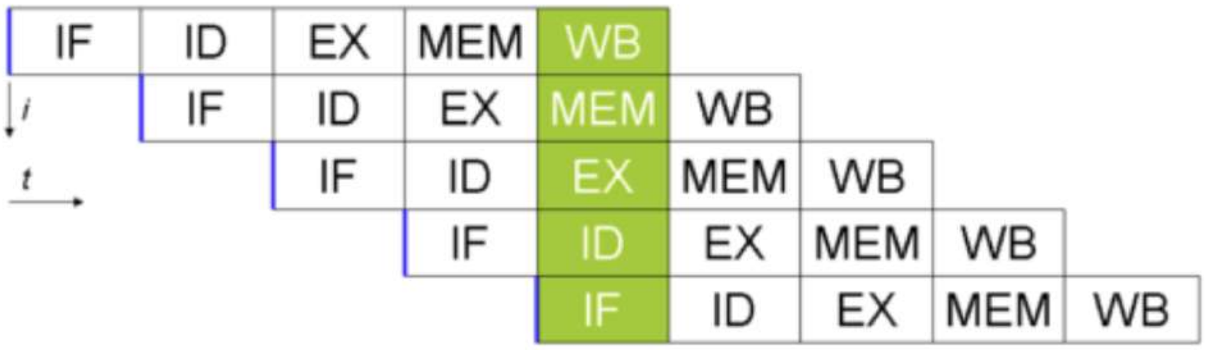


Figure 3.3: Pipelined Design approach

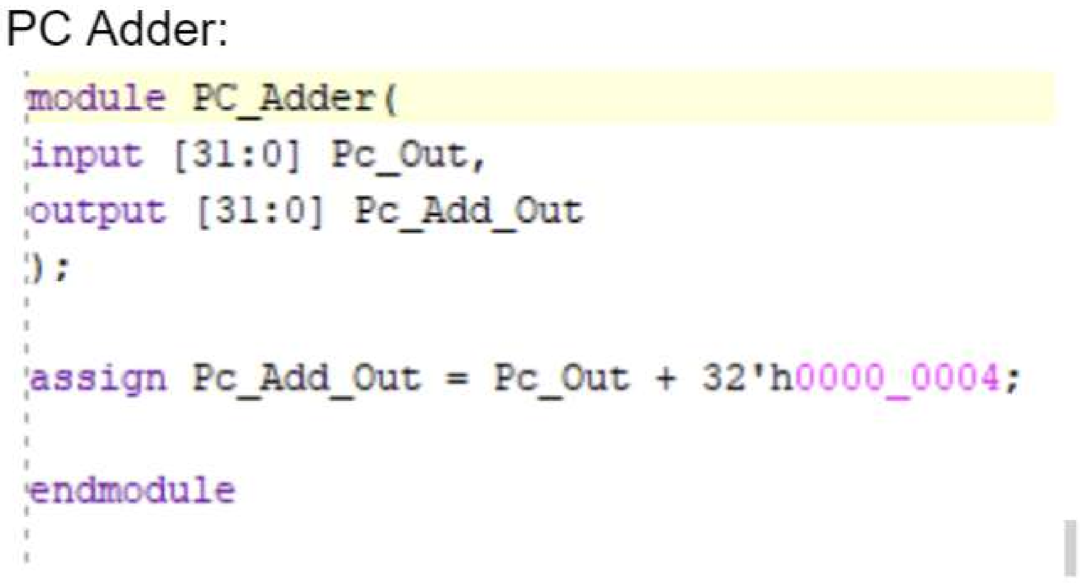
Pipelining improves the performance by increasing instruction throughput as opposed to decreasing the execution time of an individuIn pipeline design approach, each instruction is broken down into multiple stages and these instructions are overlapped in execution. Each stage in the pipeline completes a portion of the instruction and and can be operated in parallel with other stages.

Instructions will enter the pipeline at and one end and leave at the other end. Throughput will be determined as seen at the end of the pipeline. In an ideal pipelined implementation without any hazards or stalls, where each stage of the pipeline takes one clock cycle to complete, the CPI can be close to 1. This means that, on average, each instruction takes one clock cycle to execute. Pipelining doesn't reduce latency of a single task, it increases the throughput of the entire workload.

Chapter 4

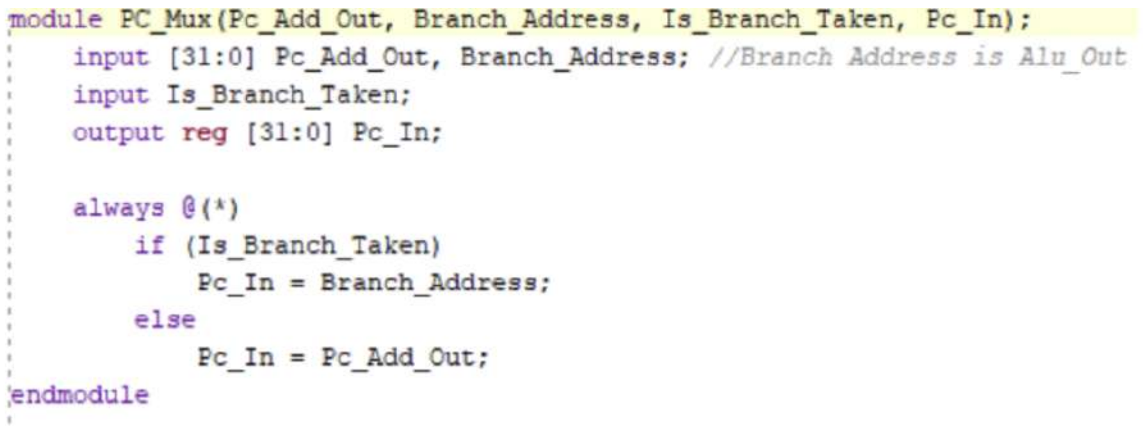
Instruction Fetch Stage

4.1 Verilog Code:


PC Adder:

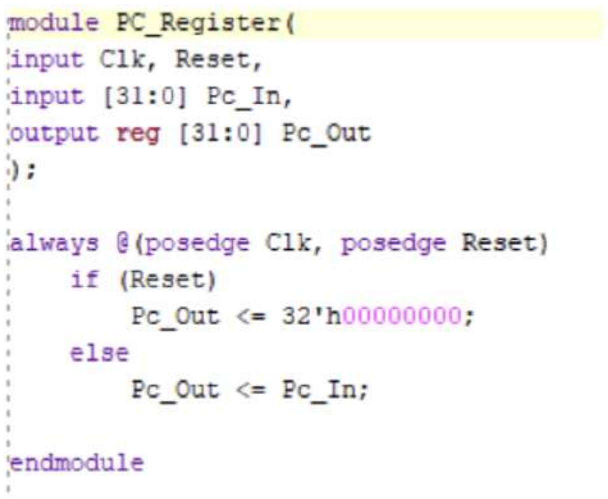
```
module PC_Adder(
input [31:0] Pc_Out,
output [31:0] Pc_Add_Out
);

assign Pc_Add_Out = Pc_Out + 32'h0000_0004;

endmodule
```

4.2 Pc mux:



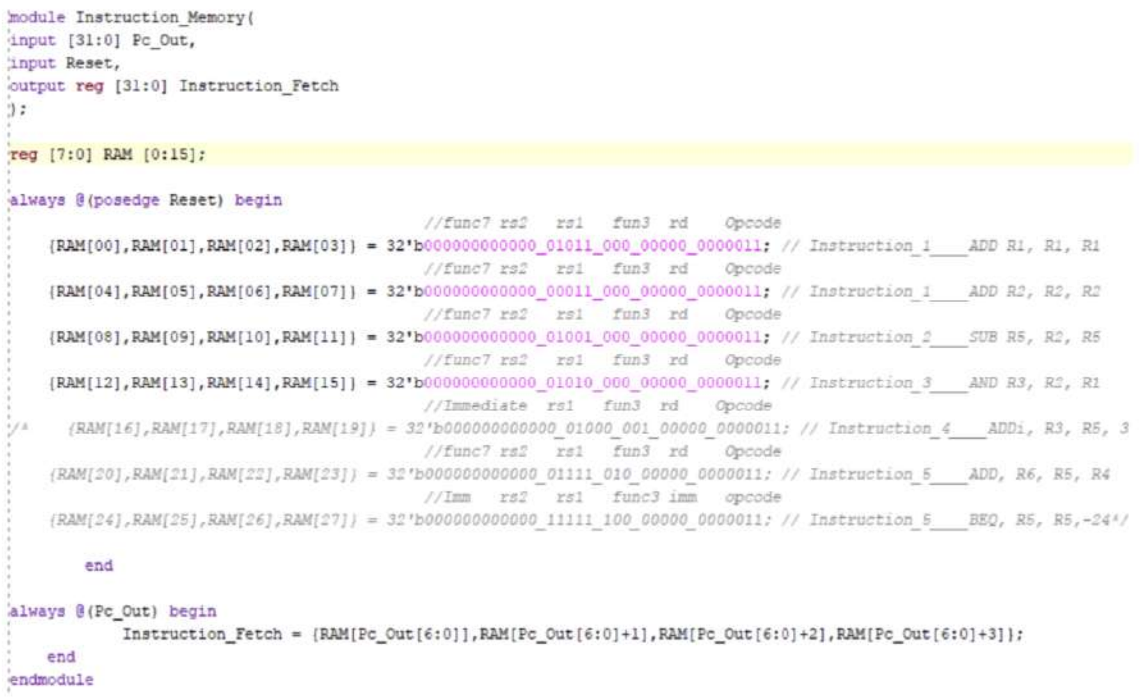
```
module PC_Mux(Pc_Add_Out, Branch_Address, Is_Branch_Taken, Pc_In);
    input [31:0] Pc_Add_Out, Branch_Address; //Branch Address is Alu_Out
    input Is_Branch_Taken;
    output reg [31:0] Pc_In;

    always @(*)
        if (Is_Branch_Taken)
            Pc_In = Branch_Address;
        else
            Pc_In = Pc_Add_Out;
endmodule
```

4.3 Pc register:



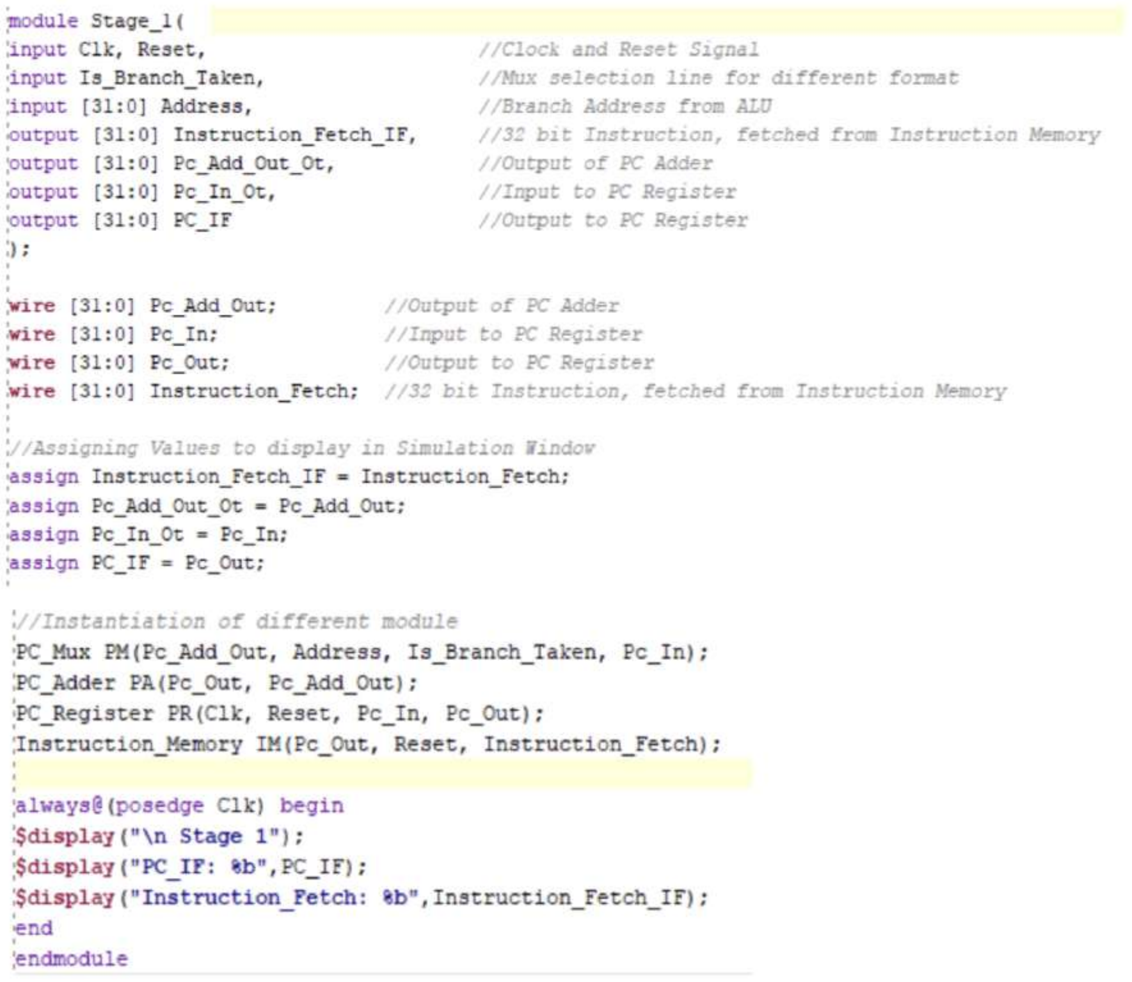
```
module PC_Register(
input Clk, Reset,
input [31:0] Pc_In,
output reg [31:0] Pc_Out
);

always @(posedge Clk, posedge Reset)
    if (Reset)
        Pc_Out <= 32'h00000000;
    else
        Pc_Out <= Pc_In;

endmodule
```

4.4 Instruction memory



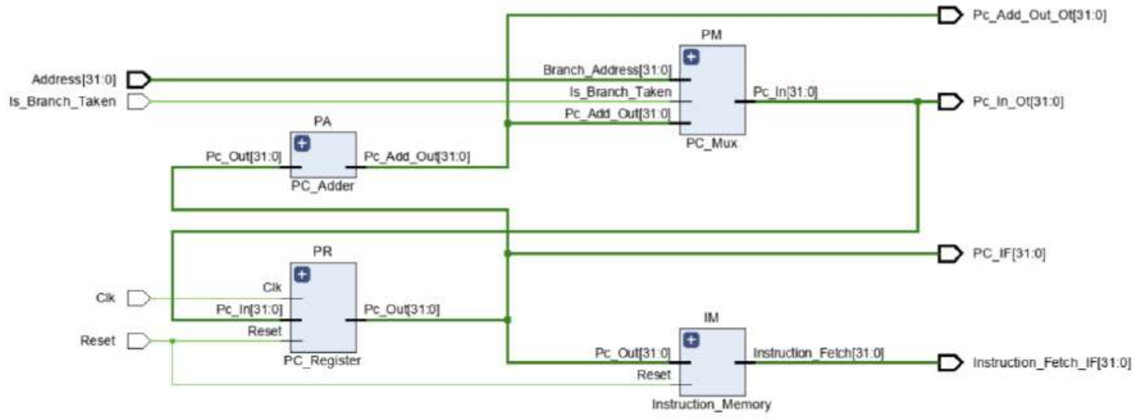
```
module Instruction_Memory(
input [31:0] Pc_Out,
input Reset,
output reg [31:0] Instruction_Fetch
);

reg [7:0] RAM [0:15];

always @(posedge Reset) begin
                                             //func7 rs2    rs1    fun3   rd     Opcode
    {RAM[00],RAM[01],RAM[02],RAM[03]} = 32'b000000000000_01011_000_00000_0000011; // Instruction_1____ADD R1, R1, R1
                                             //func7 rs2    rs1    fun3   rd     Opcode
    {RAM[04],RAM[05],RAM[06],RAM[07]} = 32'b000000000000_00011_000_00000_0000011; // Instruction_1____ADD R2, R2, R2
                                             //func7 rs2    rs1    fun3   rd     Opcode
    {RAM[08],RAM[09],RAM[10],RAM[11]} = 32'b000000000000_01001_000_00000_0000011; // Instruction_2____SUB R5, R2, R5
                                             //func7 rs2    rs1    fun3   rd     Opcode
    {RAM[12],RAM[13],RAM[14],RAM[15]} = 32'b000000000000_01010_000_00000_0000011; // Instruction_3____AND R3, R2, R1
                                             //Immediate  rs1    fun3   rd     Opcode
/*    {RAM[16],RAM[17],RAM[18],RAM[19]} = 32'b000000000000_01000_001_00000_0000011; // Instruction_4____ADDi, R3, R5, 3
                                             //func7 rs2    rs1    fun3   rd     Opcode
    {RAM[20],RAM[21],RAM[22],RAM[23]} = 32'b000000000000_01111_010_00000_0000011; // Instruction_5____ADD, R6, R5, R4
                                             //Imm   rs2    rs1    func3 imm     opcode
    {RAM[24],RAM[25],RAM[26],RAM[27]} = 32'b000000000000_11111_100_00000_0000011; // Instruction_5____BEQ, R5, R5,-24*/

        end

always @(Pc_Out) begin
            Instruction_Fetch = {RAM[Pc_Out[6:0]],RAM[Pc_Out[6:0]+1],RAM[Pc_Out[6:0]+2],RAM[Pc_Out[6:0]+3]};
    end
endmodule
```

4.5 Stage 1(all instatiation):

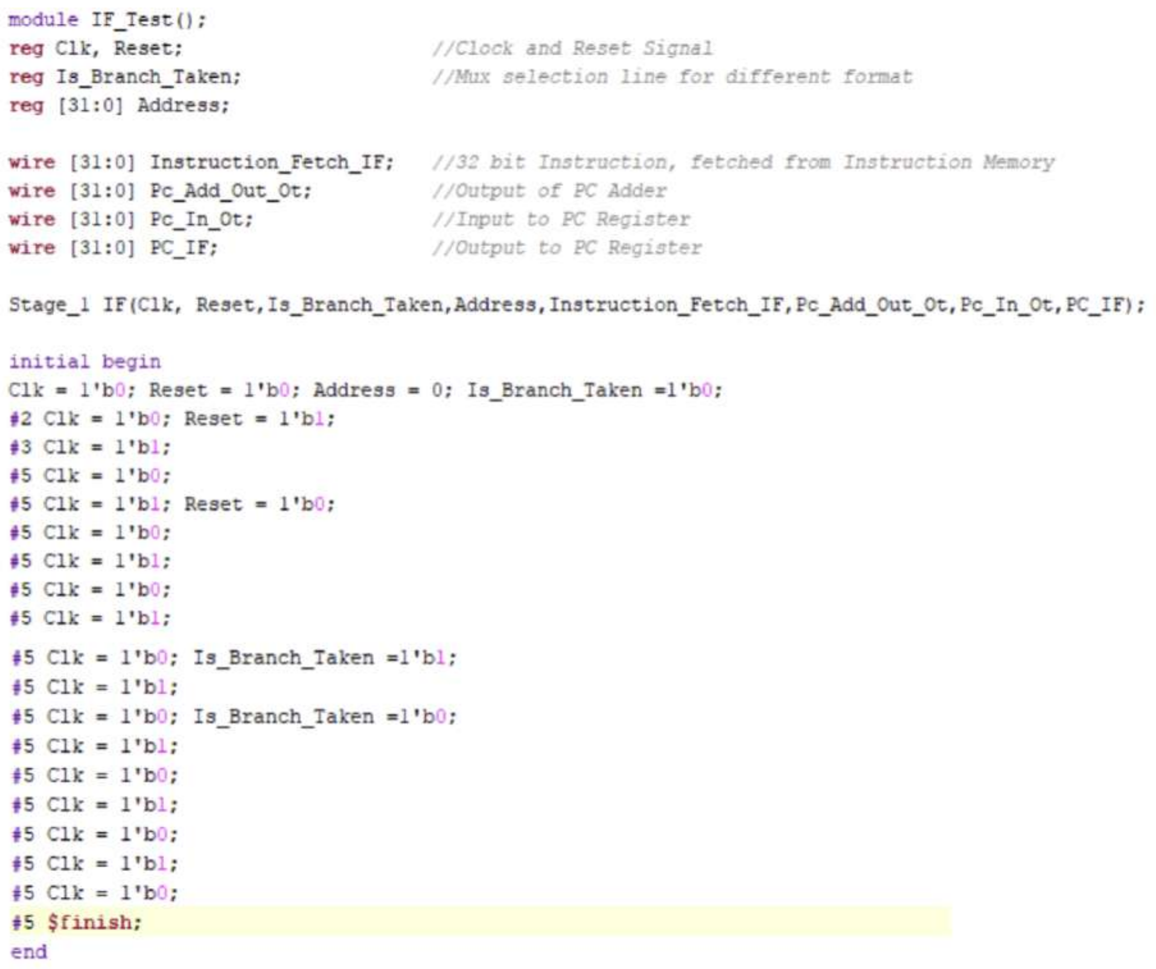


```
module Stage_1(
input Clk, Reset,                         //Clock and Reset Signal
input Is_Branch_Taken,                    //Mux selection line for different format
input [31:0] Address,                     //Branch Address from ALU
output [31:0] Instruction_Fetch_IF,       //32 bit Instruction, fetched from Instruction Memory
output [31:0] Pc_Add_Out_Ot,              //Output of PC Adder
output [31:0] Pc_In_Ot,                   //Input to PC Register
output [31:0] PC_IF                       //Output to PC Register
);

wire [31:0] Pc_Add_Out;          //Output of PC Adder
wire [31:0] Pc_In;               //Input to PC Register
wire [31:0] Pc_Out;              //Output to PC Register
wire [31:0] Instruction_Fetch;   //32 bit Instruction, fetched from Instruction Memory

//Assigning Values to display in Simulation Window
assign Instruction_Fetch_IF = Instruction_Fetch;
assign Pc_Add_Out_Ot = Pc_Add_Out;
assign Pc_In_Ot = Pc_In;
assign PC_IF = Pc_Out;

//Instantiation of different module
PC_Mux PM(Pc_Add_Out, Address, Is_Branch_Taken, Pc_In);
PC_Adder PA(Pc_Out, Pc_Add_Out);
PC_Register PR(Clk, Reset, Pc_In, Pc_Out);
Instruction_Memory IM(Pc_Out, Reset, Instruction_Fetch);

always@(posedge Clk) begin
$display("\n Stage 1");
$display("PC_IF: %b",PC_IF);
$display("Instruction_Fetch: %b",Instruction_Fetch_IF);
end
endmodule
```

4.6 Block level schematic:



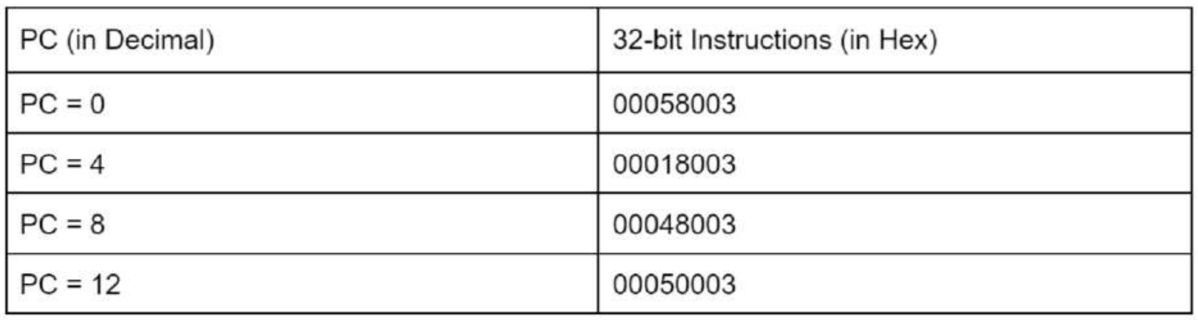
4.7 Testbench:



```
module IF_Test();
reg Clk, Reset;                     //Clock and Reset Signal
reg Is_Branch_Taken;                //Mux selection line for different format
reg [31:0] Address;

wire [31:0] Instruction_Fetch_IF;   //32 bit Instruction, fetched from Instruction Memory
wire [31:0] Pc_Add_Out_Ot;          //Output of PC Adder
wire [31:0] Pc_In_Ot;               //Input to PC Register
wire [31:0] PC_IF;                  //Output to PC Register

Stage_1 IF(Clk, Reset,Is_Branch_Taken,Address,Instruction_Fetch_IF,Pc_Add_Out_Ot,Pc_In_Ot,PC_IF);

initial begin
Clk = 1'b0; Reset = 1'b0; Address = 0; Is_Branch_Taken =1'b0;
#2 Clk = 1'b0; Reset = 1'b1;
#3 Clk = 1'b1;
#5 Clk = 1'b0;
#5 Clk = 1'b1; Reset = 1'b0;
#5 Clk = 1'b0;
#5 Clk = 1'b1;
#5 Clk = 1'b0;
#5 Clk = 1'b1;
#5 Clk = 1'b0; Is_Branch_Taken =1'b1;
#5 Clk = 1'b1;
#5 Clk = 1'b0; Is_Branch_Taken =1'b0;
#5 Clk = 1'b1;
#5 Clk = 1'b0;
#5 Clk = 1'b1;
#5 Clk = 1'b0;
#5 Clk = 1'b1;
#5 Clk = 1'b0;
#5 $finish;
end
```

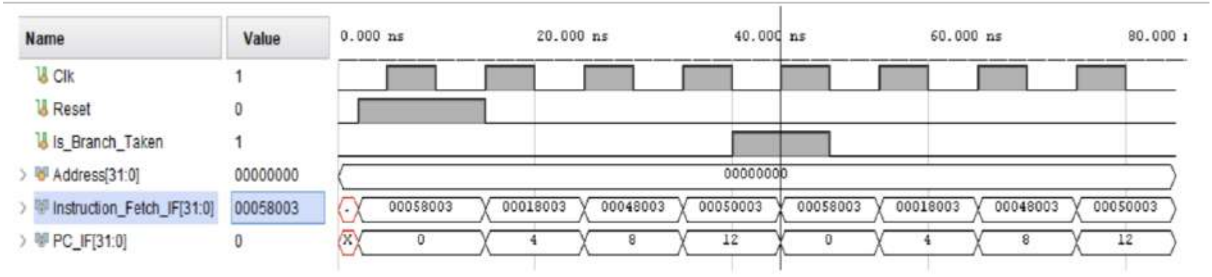
4.8 Result:

Instruction Memory is initialized by 4 instructions in hexadecimal format:



PC (in Decimal)	32-bit Instructions (in Hex)
PC = 0	00058003
PC = 4	00018003
PC = 8	00048003
PC = 12	00050003

These 4 instructions are executed in 4 clock pulse, then "Is_Branch_Taken" is switched to 1 and Branch Address is 0. So the PC value jumps to 0 and these 4 instructions are executed again.



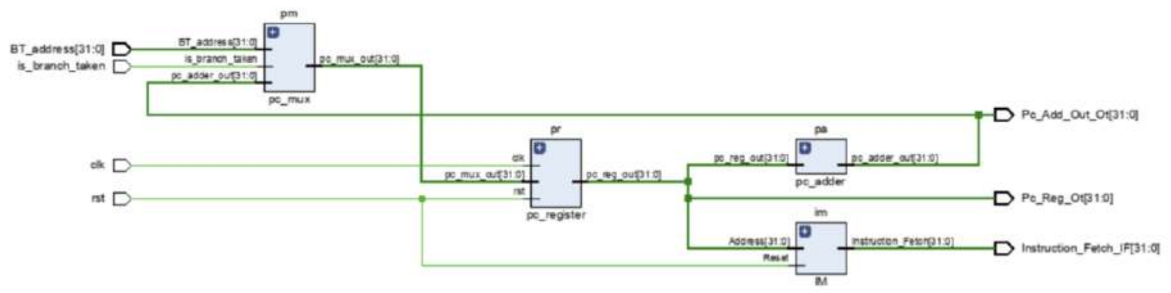
Name	Value
Clk	1
Reset	0
Is_Branch_Taken	1
Address[31:0]	00000000
Instruction_Fetch_IF[31:0]	00058003
PC_IF[31:0]	0

In this way we verified stage 1

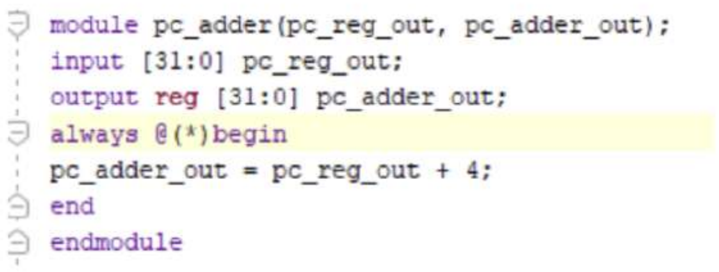
4.9 Stage 1: instruction fetch verilog code:

PC (in Decimal)	32-bit Instructions (in Hex)
PC = 0	00058003
PC = 4	00018003
PC = 8	00048003
PC = 12	00050003

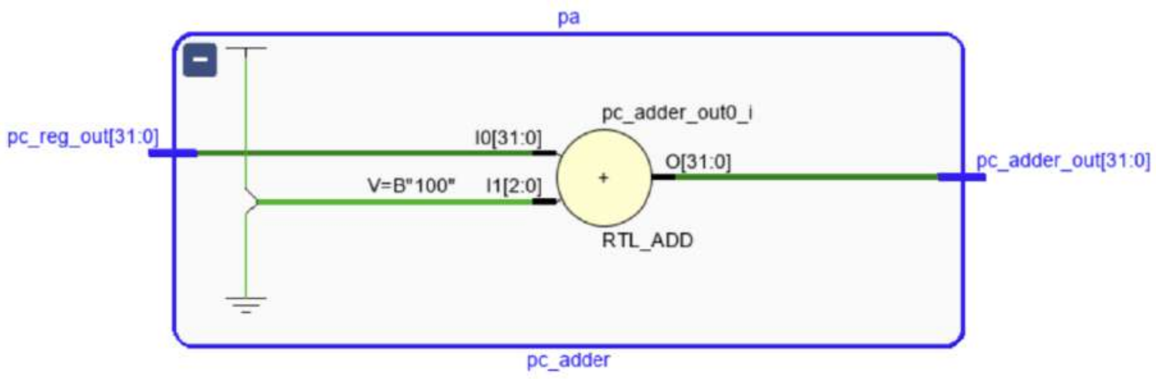
4.10 Block level schematic:



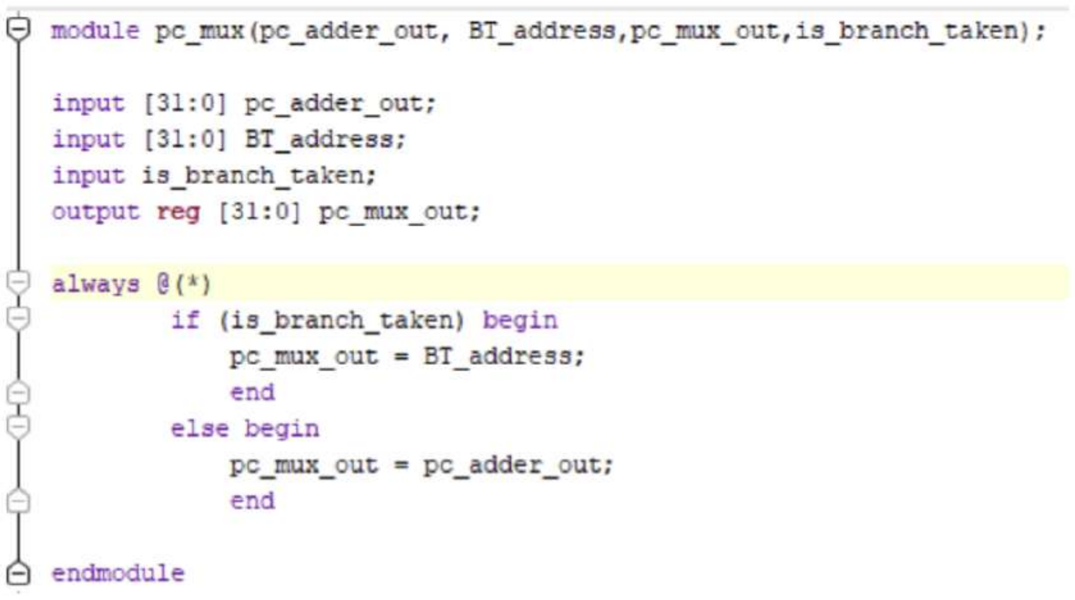
4.11 PC Adder:



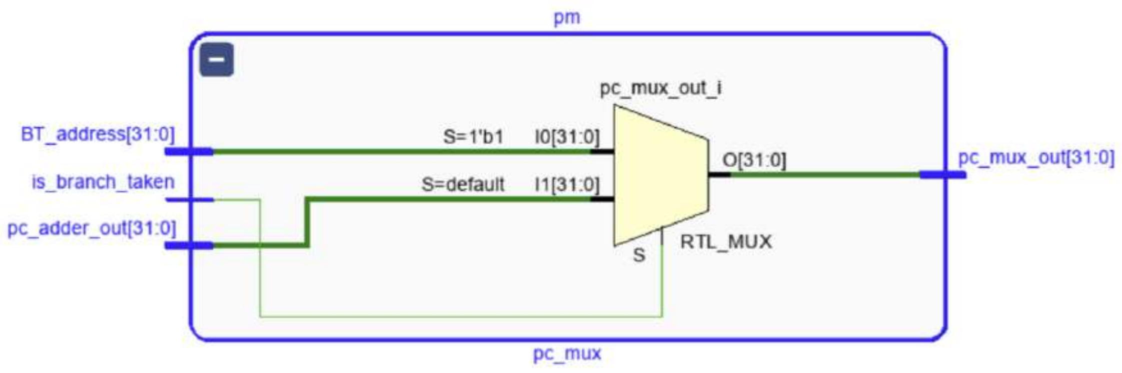
```
module pc_adder(pc_reg_out, pc_adder_out);
input [31:0] pc_reg_out;
output reg [31:0] pc_adder_out;
always @(*)begin
pc_adder_out = pc_reg_out + 4;
end
endmodule
```



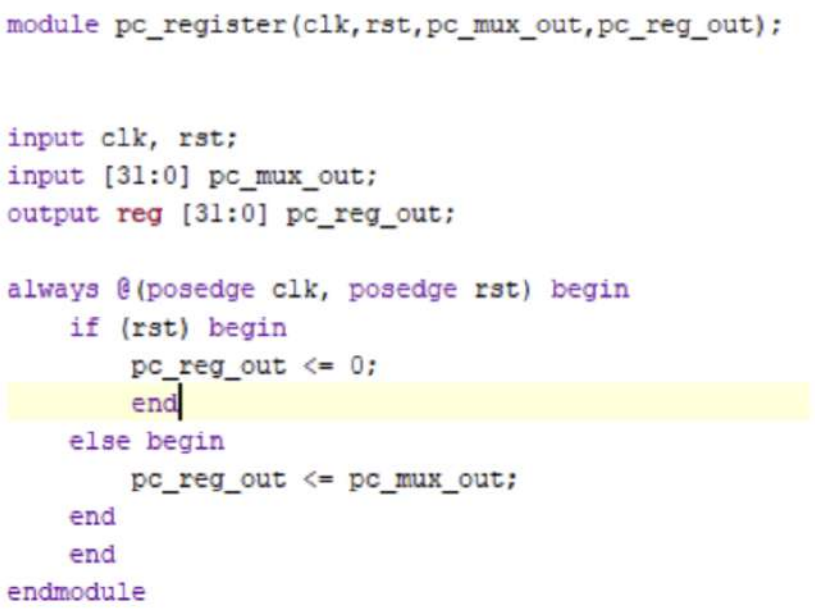
4.12 PC mux:

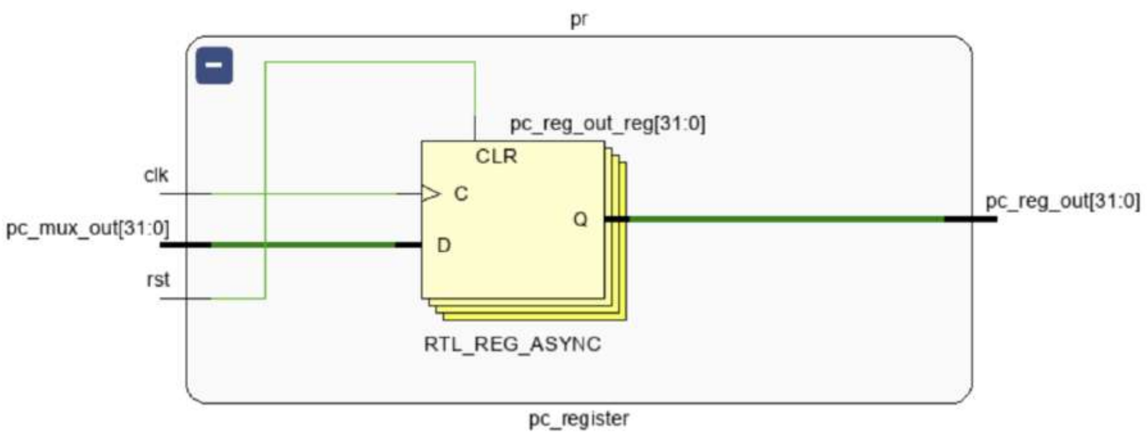


```
module pc_mux(pc_adder_out, BT_address,pc_mux_out,is_branch_taken);

input [31:0] pc_adder_out;
input [31:0] BT_address;
input is_branch_taken;
output reg [31:0] pc_mux_out;

always @(*)
        if (is_branch_taken) begin
            pc_mux_out = BT_address;
            end
        else begin
            pc_mux_out = pc_adder_out;
            end

endmodule
```

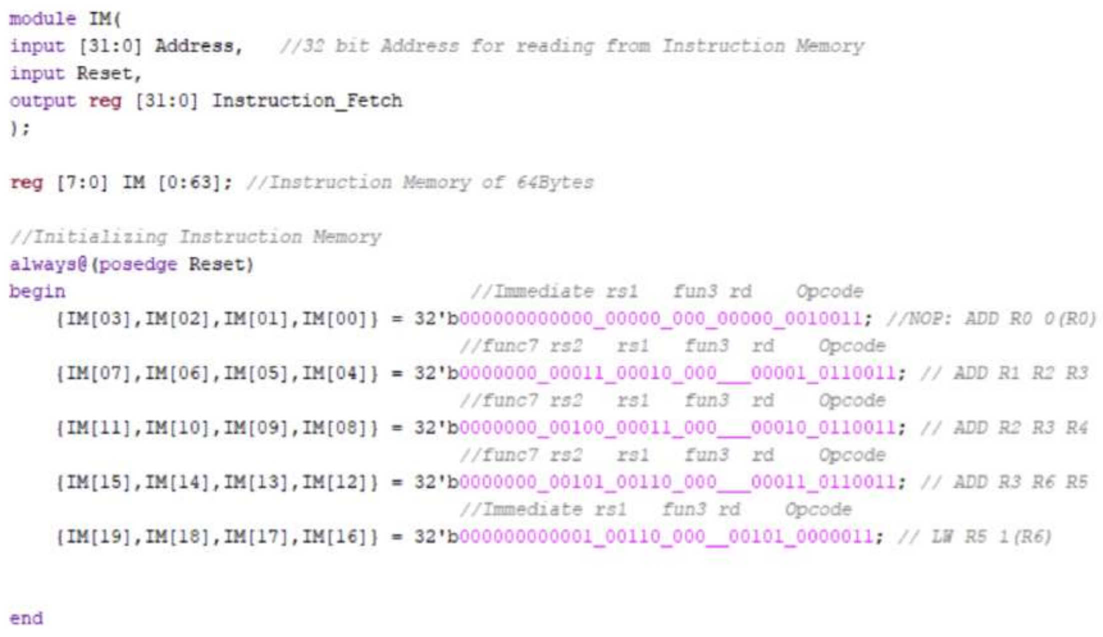


4.13 PC register:

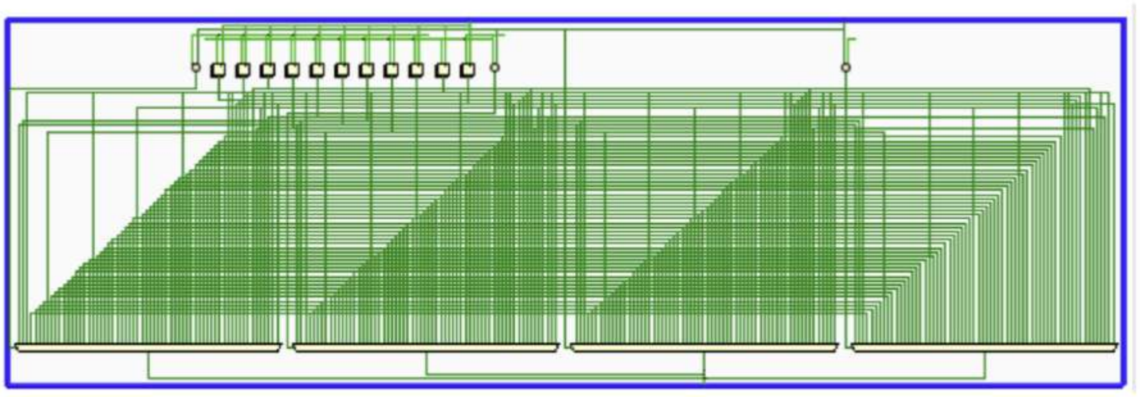

```
module pc_register(clk,rst,pc_mux_out,pc_reg_out);


input clk, rst;
input [31:0] pc_mux_out;
output reg [31:0] pc_reg_out;

always @(posedge clk, posedge rst) begin
    if (rst) begin
        pc_reg_out <= 0;
        end
    else begin
        pc_reg_out <= pc_mux_out;
    end
    end
endmodule
```



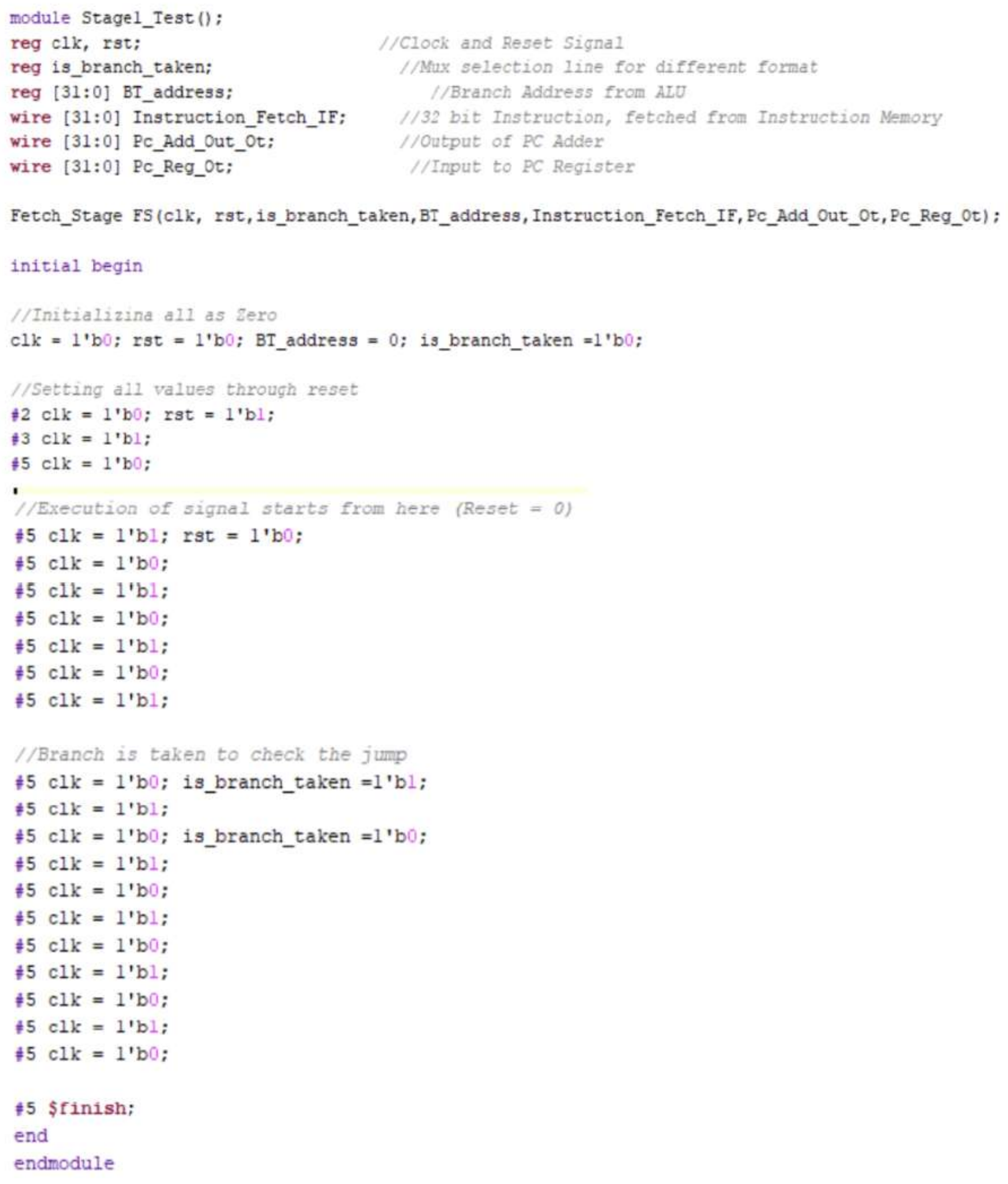
4.14 Instruction memory:



```
module IM(
input [31:0] Address,   //32 bit Address for reading from Instruction Memory
input Reset,
output reg [31:0] Instruction_Fetch
);

reg [7:0] IM [0:63]; //Instruction Memory of 64Bytes

//Initializing Instruction Memory
always@(posedge Reset)
begin                                  //Immediate rs1   fun3 rd    Opcode
    {IM[03],IM[02],IM[01],IM[00]} = 32'b000000000000_00000_000_00000_0010011; //NOP: ADD R0 0(R0)
                                      //func7 rs2   rs1   fun3  rd    Opcode
    {IM[07],IM[06],IM[05],IM[04]} = 32'b0000000_00011_00010_000___00001_0110011; // ADD R1 R2 R3
                                      //func7 rs2   rs1   fun3  rd    Opcode
    {IM[11],IM[10],IM[09],IM[08]} = 32'b0000000_00100_00011_000___00010_0110011; // ADD R2 R3 R4
                                      //func7 rs2   rs1   fun3  rd    Opcode
    {IM[15],IM[14],IM[13],IM[12]} = 32'b0000000_00101_00110_000___00011_0110011; // ADD R3 R6 R5
                                      //Immediate rs1   fun3 rd    Opcode
    {IM[19],IM[18],IM[17],IM[16]} = 32'b000000000001_00110_000__00101_0000011; // LW R5 1(R6)


end
```



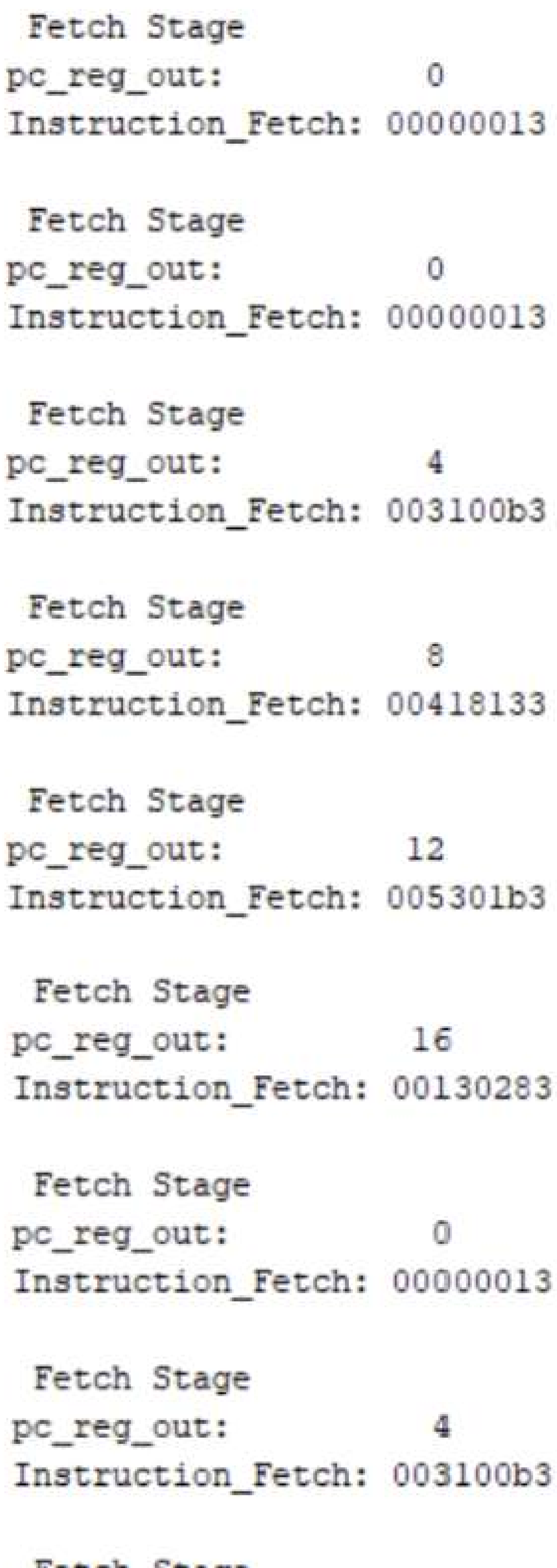
4.15 Testbench:



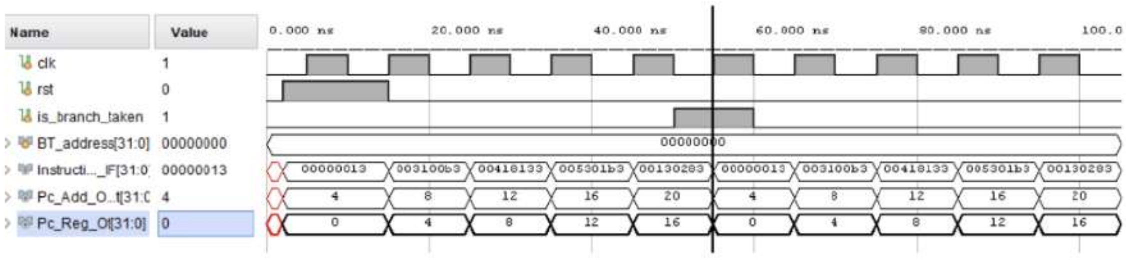
```
module Stage1_Test();
reg clk, rst;                           //Clock and Reset Signal
reg is_branch_taken;                      //Mux selection line for different format
reg [31:0] BT_address;                       //Branch Address from ALU
wire [31:0] Instruction_Fetch_IF;         //32 bit Instruction, fetched from Instruction Memory
wire [31:0] Pc_Add_Out_Ot;                //Output of PC Adder
wire [31:0] Pc_Reg_Ot;                     //Input to PC Register

Fetch_Stage FS(clk, rst,is_branch_taken,BT_address,Instruction_Fetch_IF,Pc_Add_Out_Ot,Pc_Reg_Ot);

initial begin

//Initializing all as Zero
clk = 1'b0; rst = 1'b0; BT_address = 0; is_branch_taken =1'b0;

//Setting all values through reset
#2 clk = 1'b0; rst = 1'b1;
#3 clk = 1'b1;
#5 clk = 1'b0;

//Execution of signal starts from here (Reset = 0)
#5 clk = 1'b1; rst = 1'b0;
#5 clk = 1'b0;
#5 clk = 1'b1;
#5 clk = 1'b0;
#5 clk = 1'b1;
#5 clk = 1'b0;
#5 clk = 1'b1;

//Branch is taken to check the jump
#5 clk = 1'b0; is_branch_taken =1'b1;
#5 clk = 1'b1;
#5 clk = 1'b0; is_branch_taken =1'b0;
#5 clk = 1'b1;
#5 clk = 1'b0;
#5 clk = 1'b1;
#5 clk = 1'b0;
#5 clk = 1'b1;
#5 clk = 1'b0;
#5 clk = 1'b1;
#5 clk = 1'b0;

#5 $finish;
end
endmodule
```

4.16 Result:

Reset is On for first 2 cycles so No Operation code (addi r0, 0(r0)) is executed. After that all instruction in "Instruction Memory" are processed till Branch instruction. In Branch instruction, the branch address is "0", so from start all instruction are executed again.



```
 Fetch Stage
pc_reg_out:          0
Instruction_Fetch: 00000013

 Fetch Stage
pc_reg_out:          0
Instruction_Fetch: 00000013

 Fetch Stage
pc_reg_out:          4
Instruction_Fetch: 003100b3

 Fetch Stage
pc_reg_out:          8
Instruction_Fetch: 00418133

 Fetch Stage
pc_reg_out:         12
Instruction_Fetch: 005301b3

 Fetch Stage
pc_reg_out:         16
Instruction_Fetch: 00130283

 Fetch Stage
pc_reg_out:          0
Instruction_Fetch: 00000013

 Fetch Stage
pc_reg_out:          4
Instruction_Fetch: 003100b3
```


Name
Value
0.000 ns
20.000 ns
40.000 ns
60.000 ns
80.000 ns
100.0
clk 1
rst 0
is_branch_taken 1
BT_address[31:0] 00000000
Instructi..._IF[31:0 00000013
Pc_Add_O...t[31:0 4
Pc_Reg_O[31:0] 0
00000000
00000013 003100b3 00418133 005301b3 00130283 00000013 003100b3 00418133 005301b3 00130283
4 8 12 16 20 4 8 12 16 20
0 4 8 12 16 0 4 8 12 16

Chapter 5

Instruction Decode Stage

5.1 Introduction

The main tasks of the instruction decode stage is:

- Format identification
- Operand fetch
- Operand select
- Immediate generation
- Execute control generation
- Memory control generation
- Writeback control generation

It is also the job of the instruction decode stage to raise a invalid interrupt if a invalid instruction format gets detected. The decode stage comprises of the most bulky modules and is very important from the design point of view. It also generates the control signals which is required for proper functioning of the later stages.

5.2 Register File

RV32I ISA supports a register bank consisting of 32 separate registers each of 32-bit width size out of which x0-x31 are general purpose registers with x0 being hardwire to ground at all times.

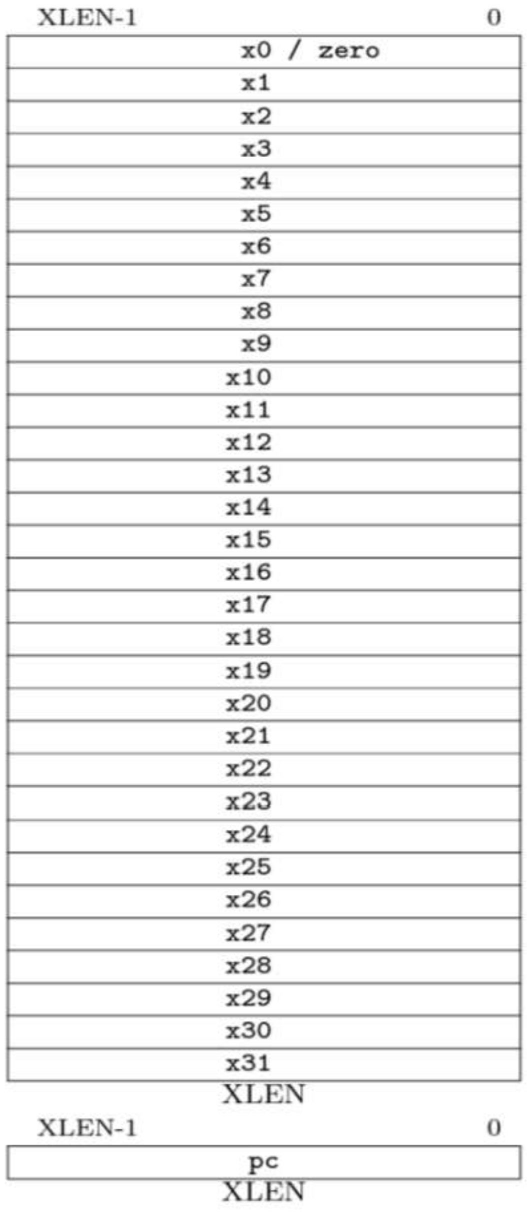
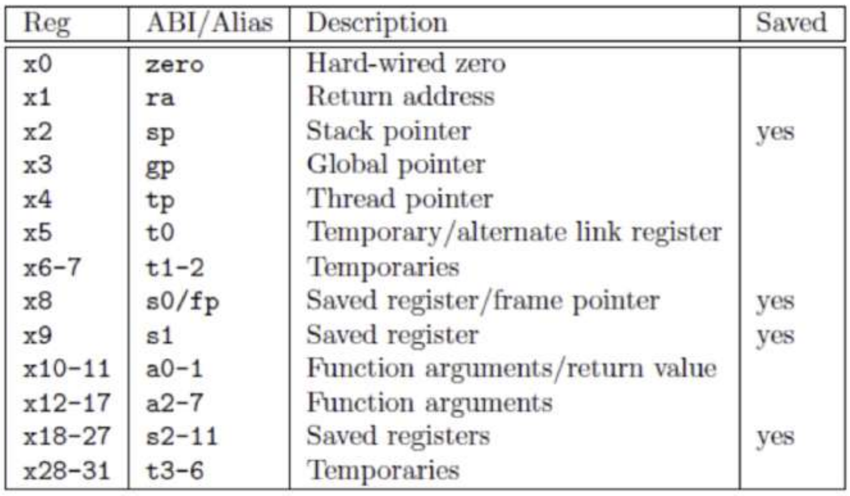


FIGURE 5.1: Register file

Nomenclature and description of each register is given as such in the above table.

The register file has two source operand addresses each of 5-bit width (Src reg adrs 1 and Src reg adrs 2) fields as decoded from the instruction fetched from the first stage. The two source operands are provided with their respective enable signals (Readenable 1 and Read enable 2) each of 1-bit width. The register file also has two register out operands (Rout1 and Rout2) each of 32-bit width as outputs whenever the respective register addresses are decoded.

Read Operation: If Read enable 1 is true then register value at Src reg adrs 1 is provided to Rout1. Similarly for Rout2 as well.

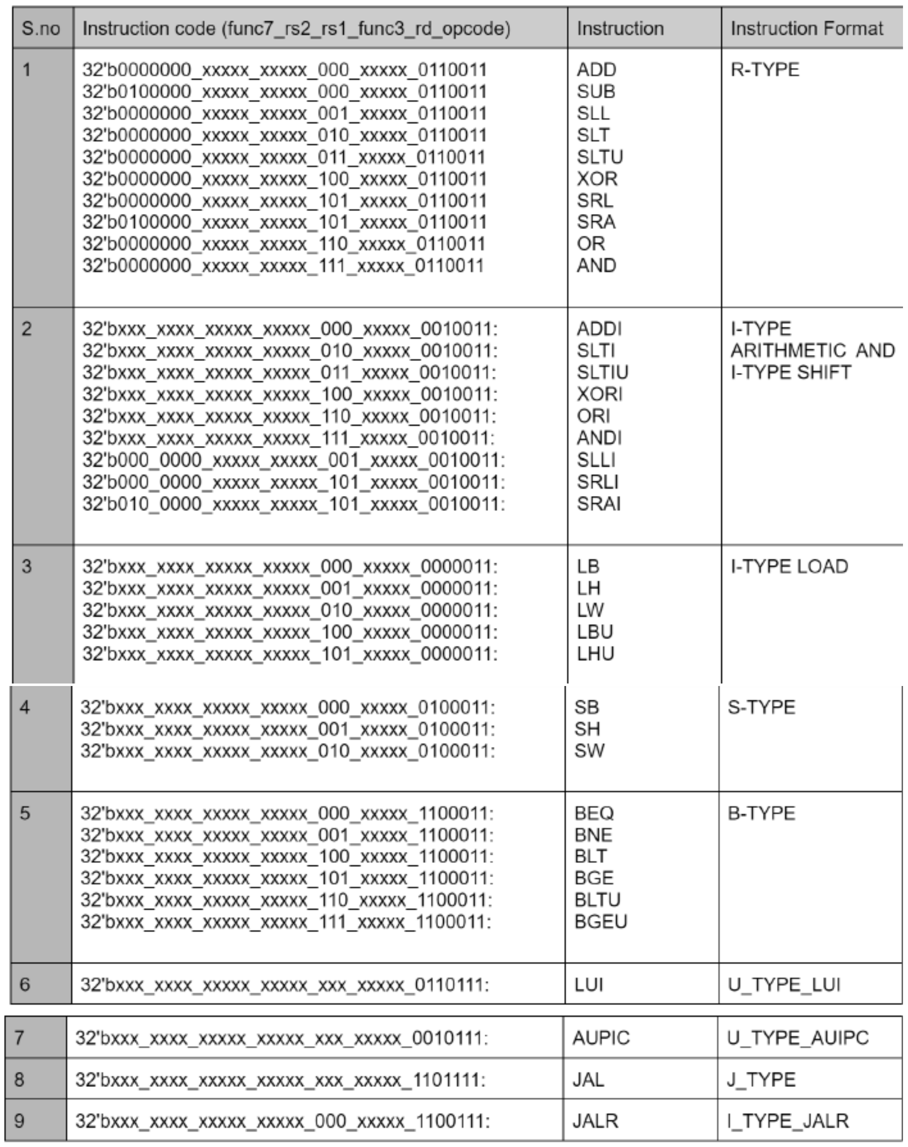


Reg	ABI/Alias	Description	Saved
x0	zero	Hard-wired zero	
x1	ra	Return address	
x2	sp	Stack pointer	yes
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary/alternate link register	
x6-7	t1-2	Temporaries	
x8	s0/fp	Saved register/frame pointer	yes
x9	s1	Saved register	yes
x10-11	a0-1	Function arguments/return value	
x12-17	a2-7	Function arguments	
x18-27	s2-11	Saved registers	yes
x28-31	t3-6	Temporaries	

FIGURE 5.2: Register file Nomenclature

Write Operation: Writeback operation is such that it happens at the negative edge of clock. Data in is another input required in the register file to store the data produced from the writeback stage in the register file. Data in is 32-bit width in size. Write operation will only happen at the negative edge of the clock as well if the write enable signal is high.

X0 is the register which will by default always be hardwired to ground.



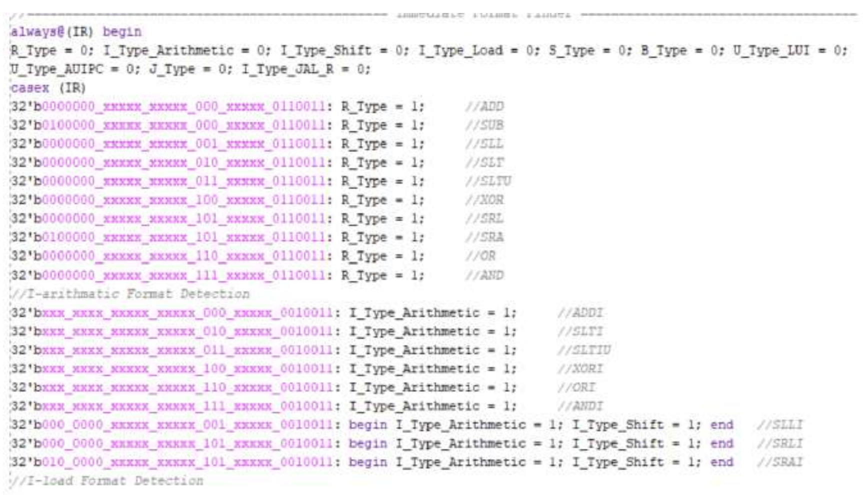
S.no	Instruction code (func7_rs2_rs1_func3_rd_opcode)	Instruction	Instruction Format
1	32'b0000000_xxxxx_xxxxx_000_xxxxx_0110011 32'b0100000_xxxxx_xxxxx_000_xxxxx_0110011 32'b0000000_xxxxx_xxxxx_001_xxxxx_0110011 32'b0000000_xxxxx_xxxxx_010_xxxxx_0110011 32'b0000000_xxxxx_xxxxx_011_xxxxx_0110011 32'b0000000_xxxxx_xxxxx_100_xxxxx_0110011 32'b0000000_xxxxx_xxxxx_101_xxxxx_0110011 32'b0100000_xxxxx_xxxxx_101_xxxxx_0110011 32'b0000000_xxxxx_xxxxx_110_xxxxx_0110011 32'b0000000_xxxxx_xxxxx_111_xxxxx_0110011	ADD SUB SLL SLT SLTU XOR SRL SRA OR AND	R-TYPE
2	32'bxxx_xxxx_xxxxx_xxxxx_000_xxxxx_0010011: 32'bxxx_xxxx_xxxxx_xxxxx_010_xxxxx_0010011: 32'bxxx_xxxx_xxxxx_xxxxx_011_xxxxx_0010011: 32'bxxx_xxxx_xxxxx_xxxxx_100_xxxxx_0010011: 32'bxxx_xxxx_xxxxx_xxxxx_110_xxxxx_0010011: 32'bxxx_xxxx_xxxxx_xxxxx_111_xxxxx_0010011: 32'b000_0000_xxxxx_xxxxx_001_xxxxx_0010011: 32'b000_0000_xxxxx_xxxxx_101_xxxxx_0010011: 32'b010_0000_xxxxx_xxxxx_101_xxxxx_0010011:	ADDI SLTI SLTIU XORI ORI ANDI SLLI SRLI SRAI	I-TYPE ARITHMETIC AND I-TYPE SHIFT
3	32'bxxx_xxxx_xxxxx_xxxxx_000_xxxxx_0000011: 32'bxxx_xxxx_xxxxx_xxxxx_001_xxxxx_0000011: 32'bxxx_xxxx_xxxxx_xxxxx_010_xxxxx_0000011: 32'bxxx_xxxx_xxxxx_xxxxx_100_xxxxx_0000011: 32'bxxx_xxxx_xxxxx_xxxxx_101_xxxxx_0000011:	LB LH LW LBU LHU	I-TYPE LOAD
4	32'bxxx_xxxx_xxxxx_xxxxx_000_xxxxx_0100011: 32'bxxx_xxxx_xxxxx_xxxxx_001_xxxxx_0100011: 32'bxxx_xxxx_xxxxx_xxxxx_010_xxxxx_0100011:	SB SH SW	S-TYPE
5	32'bxxx_xxxx_xxxxx_xxxxx_000_xxxxx_1100011: 32'bxxx_xxxx_xxxxx_xxxxx_001_xxxxx_1100011: 32'bxxx_xxxx_xxxxx_xxxxx_100_xxxxx_1100011: 32'bxxx_xxxx_xxxxx_xxxxx_101_xxxxx_1100011: 32'bxxx_xxxx_xxxxx_xxxxx_110_xxxxx_1100011: 32'bxxx_xxxx_xxxxx_xxxxx_111_xxxxx_1100011:	BEQ BNE BLT BGE BLTU BGEU	B-TYPE
6	32'bxxx_xxxx_xxxxx_xxxxx_xxx_xxxxx_0110111:	LUI	U_TYPE_LUI
7	32'bxxx_xxxx_xxxxx_xxxxx_xxx_xxxxx_0010111:	AUPIC	U_TYPE_AUIPC
8	32'bxxx_xxxx_xxxxx_xxxxx_xxx_xxxxx_1101111:	JAL	J_TYPE
9	32'bxxx_xxxx_xxxxx_xxxxx_000_xxxxx_1100111:	JALR	I_TYPE_JALR

5.3 Format Finder

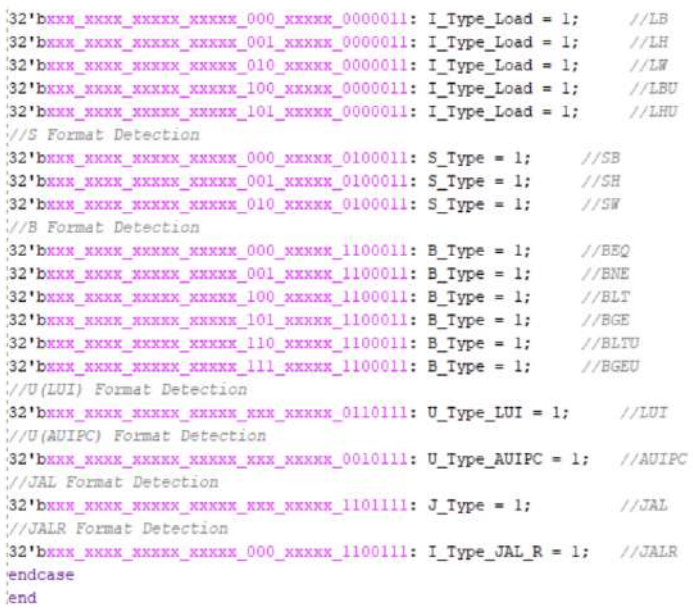
Depending on the IR the format finder will tell format of the instruction. There are total 9 different types of formats - R_TYPE, I_TYPE_ARITHMETIC/I_TYPE_SHIFT, I_TYPE_LOAD, S_TYPE, B_TYPE,U_LUI, U_AUPIC J_TYPE, I_TYPE_JALR. These signals will be forwarded to the execution, memory and writeback stages for their own execution.

CODE FOR FORMAT FINDER:

Verification:



```
//--------------------------------------------- Immediate Format Finder ------------------------------------
always@(IR) begin
R_Type = 0; I_Type_Arithmetic = 0; I_Type_Shift = 0; I_Type_Load = 0; S_Type = 0; B_Type = 0; U_Type_LUI = 0;
U_Type_AUIPC = 0; J_Type = 0; I_Type_JAL_R = 0;
casex (IR)
32'b0000000_xxxxx_xxxxx_000_xxxxx_0110011: R_Type = 1;     //ADD
32'b0100000_xxxxx_xxxxx_000_xxxxx_0110011: R_Type = 1;     //SUB
32'b0000000_xxxxx_xxxxx_001_xxxxx_0110011: R_Type = 1;     //SLL
32'b0000000_xxxxx_xxxxx_010_xxxxx_0110011: R_Type = 1;     //SLT
32'b0000000_xxxxx_xxxxx_011_xxxxx_0110011: R_Type = 1;     //SLTU
32'b0000000_xxxxx_xxxxx_100_xxxxx_0110011: R_Type = 1;     //XOR
32'b0000000_xxxxx_xxxxx_101_xxxxx_0110011: R_Type = 1;     //SRL
32'b0100000_xxxxx_xxxxx_101_xxxxx_0110011: R_Type = 1;     //SRA
32'b0000000_xxxxx_xxxxx_110_xxxxx_0110011: R_Type = 1;     //OR
32'b0000000_xxxxx_xxxxx_111_xxxxx_0110011: R_Type = 1;     //AND
//I-arithmatic Format Detection
32'bxxx_xxxx_xxxxx_xxxxx_000_xxxxx_0010011: I_Type_Arithmetic = 1;     //ADDI
32'bxxx_xxxx_xxxxx_xxxxx_010_xxxxx_0010011: I_Type_Arithmetic = 1;     //SLTI
32'bxxx_xxxx_xxxxx_xxxxx_011_xxxxx_0010011: I_Type_Arithmetic = 1;     //SLTIU
32'bxxx_xxxx_xxxxx_xxxxx_100_xxxxx_0010011: I_Type_Arithmetic = 1;     //XORI
32'bxxx_xxxx_xxxxx_xxxxx_110_xxxxx_0010011: I_Type_Arithmetic = 1;     //ORI
32'bxxx_xxxx_xxxxx_xxxxx_111_xxxxx_0010011: I_Type_Arithmetic = 1;     //ANDI
32'b000_0000_xxxxx_xxxxx_001_xxxxx_0010011: begin I_Type_Arithmetic = 1; I_Type_Shift = 1; end   //SLLI
32'b000_0000_xxxxx_xxxxx_101_xxxxx_0010011: begin I_Type_Arithmetic = 1; I_Type_Shift = 1; end   //SRLI
32'b010_0000_xxxxx_xxxxx_101_xxxxx_0010011: begin I_Type_Arithmetic = 1; I_Type_Shift = 1; end   //SRAI
//I-load Format Detection
```



```
32'bxxx_xxxx_xxxxx_xxxxx_000_xxxxx_0000011: I_Type_Load = 1;     //LB
32'bxxx_xxxx_xxxxx_xxxxx_001_xxxxx_0000011: I_Type_Load = 1;     //LH
32'bxxx_xxxx_xxxxx_xxxxx_010_xxxxx_0000011: I_Type_Load = 1;     //LW
32'bxxx_xxxx_xxxxx_xxxxx_100_xxxxx_0000011: I_Type_Load = 1;     //LBU
32'bxxx_xxxx_xxxxx_xxxxx_101_xxxxx_0000011: I_Type_Load = 1;     //LHU
//S Format Detection
32'bxxx_xxxx_xxxxx_xxxxx_000_xxxxx_0100011: S_Type = 1;     //SB
32'bxxx_xxxx_xxxxx_xxxxx_001_xxxxx_0100011: S_Type = 1;     //SH
32'bxxx_xxxx_xxxxx_xxxxx_010_xxxxx_0100011: S_Type = 1;     //SW
//B Format Detection
32'bxxx_xxxx_xxxxx_xxxxx_000_xxxxx_1100011: B_Type = 1;     //BEQ
32'bxxx_xxxx_xxxxx_xxxxx_001_xxxxx_1100011: B_Type = 1;     //BNE
32'bxxx_xxxx_xxxxx_xxxxx_100_xxxxx_1100011: B_Type = 1;     //BLT
32'bxxx_xxxx_xxxxx_xxxxx_101_xxxxx_1100011: B_Type = 1;     //BGE
32'bxxx_xxxx_xxxxx_xxxxx_110_xxxxx_1100011: B_Type = 1;     //BLTU
32'bxxx_xxxx_xxxxx_xxxxx_111_xxxxx_1100011: B_Type = 1;     //BGEU
//U(LUI) Format Detection
32'bxxx_xxxx_xxxxx_xxxxx_xxx_xxxxx_0110111: U_Type_LUI = 1;     //LUI
//U(AUIPC) Format Detection
32'bxxx_xxxx_xxxxx_xxxxx_xxx_xxxxx_0010111: U_Type_AUIPC = 1;   //AUIPC
//JAL Format Detection
32'bxxx_xxxx_xxxxx_xxxxx_xxx_xxxxx_1101111: J_Type = 1;         //JAL
//JALR Format Detection
32'bxxx_xxxx_xxxxx_xxxxx_000_xxxxx_1100111: I_Type_JAL_R = 1;   //JALR
endcase
end
```

5.4 Immediate Finder

The immediate generator will generate six different immediate values according to the instruction.

Imm_i_arithmetic and imm_i_shift :

The i_type instruction format has 12-bit signed immediate which gets encoded as shown to give a 32-bit sign extended immediate.

Imm_u:

The U-Type format is used for instructions that use a 20-bit immediate operand and an rd destination rd register. The 20-bit immediate from the instruction is manipulated as such to form 32-bit immediate value. The immediate is common for LUI and AUIPC instructions.

Imm_s:

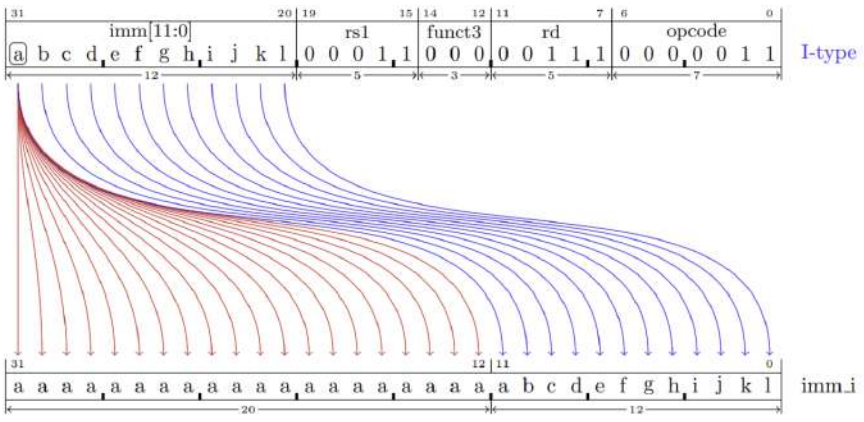


FIGURE 5.3: Imm_i

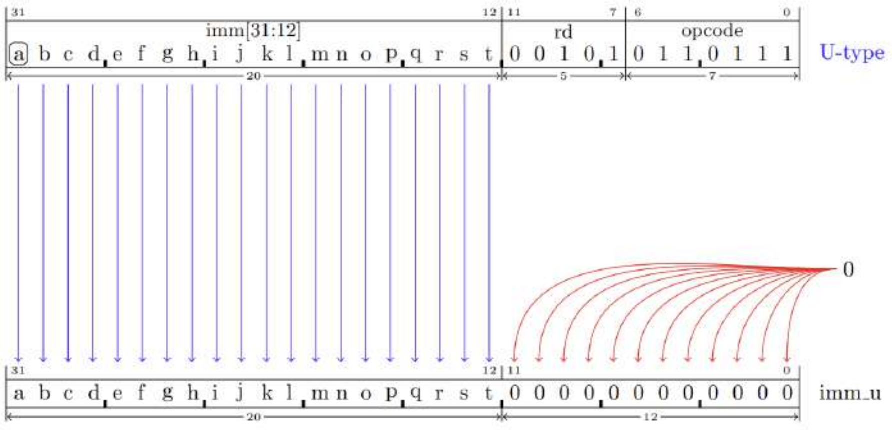


FIGURE 5.4: imm_u

The S-type instruction format comes with a signed 12-bit immediate operand with a range of 2046 to 2047, an rs1 register, and an rs2 register.

Imm_b:

The pc relative offset is expressed as 13-bit even value ranging from [-4096 to 4094] as shown below. The formed 32-bit imm_b offset will be relative to PC address.

Imm_jal:

The J-type instruction format is used to encode the JAL immediate value which is used as jump target address. Note that the imm_j value is encoded as 21-bit signed immediate value which has a range of [1048576..1048574].

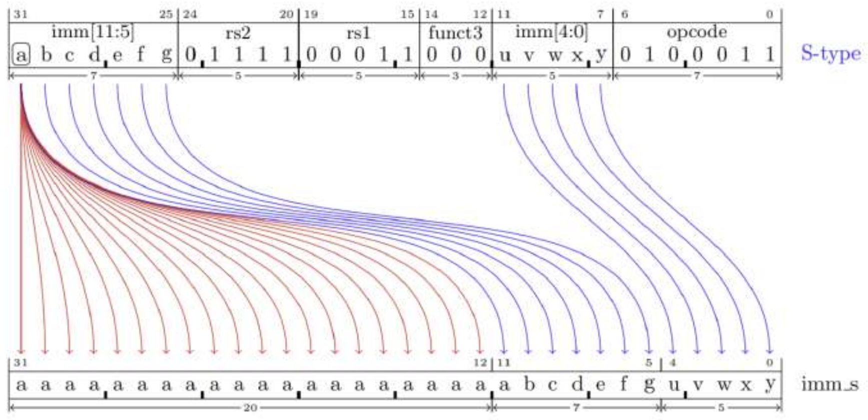


FIGURE 5.5: Imm_s

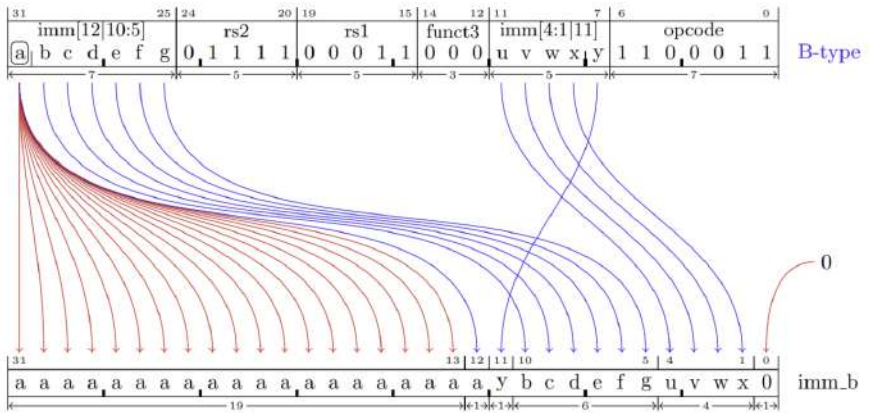


FIGURE 5.6: imm_b

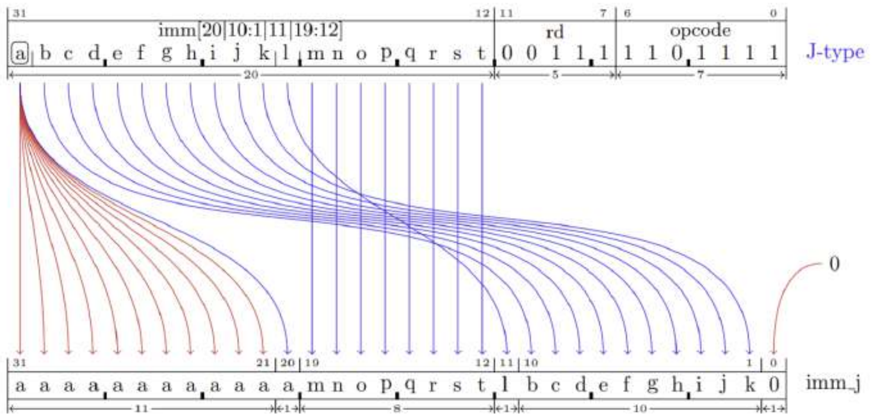
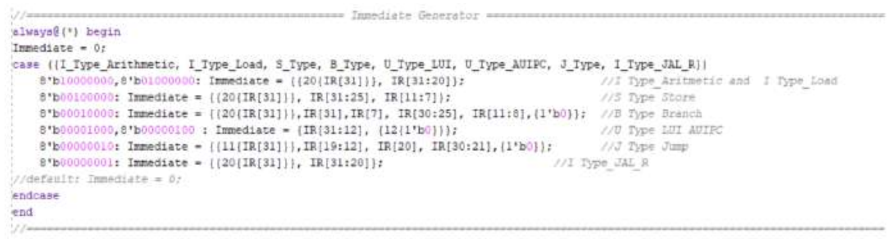
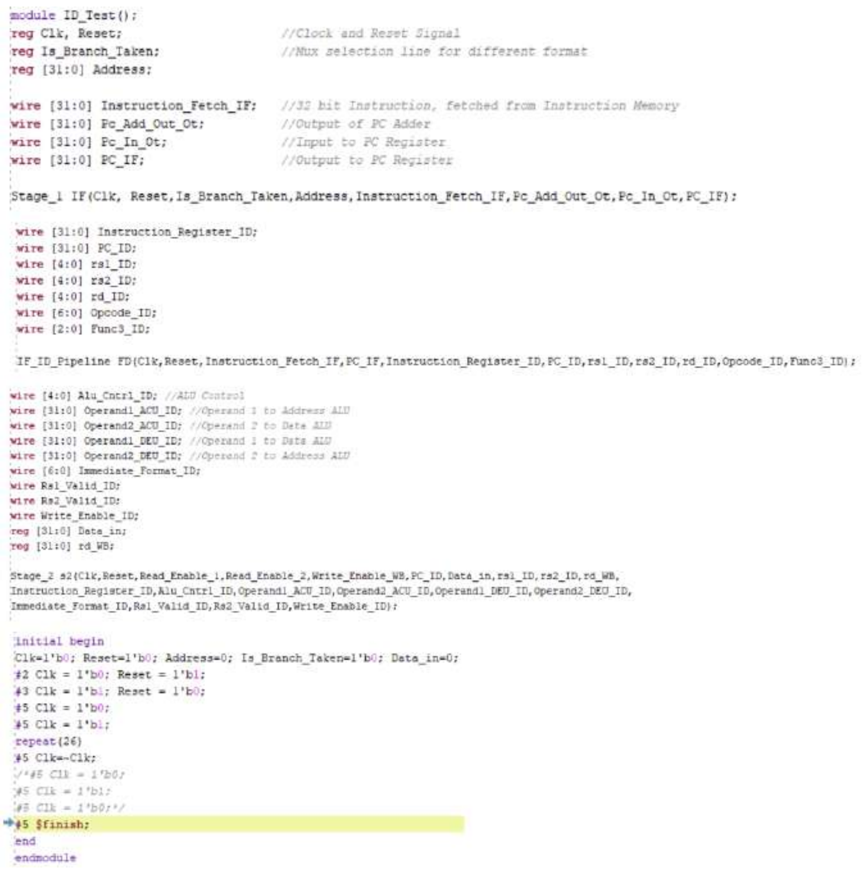


FIGURE 5.7: imm_j



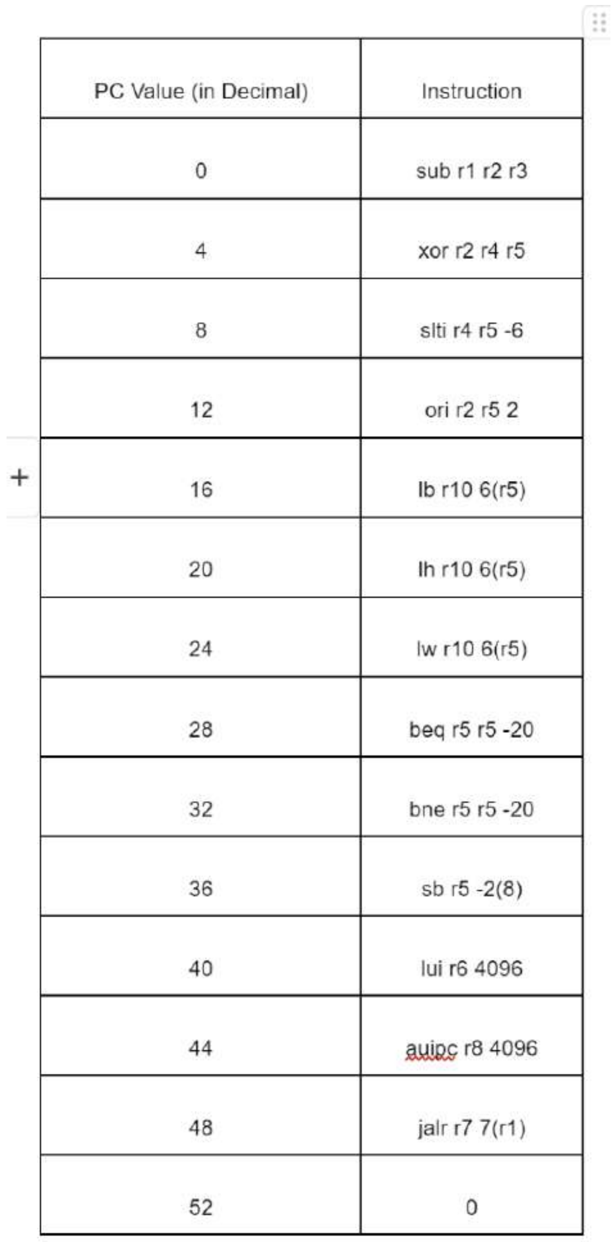
```
//================================================= Immediate Generator ==========================================================
always@(*) begin
Immediate = 0;
case ({I_Type_Arithmetic, I_Type_Load, S_Type, B_Type, U_Type_LUI, U_Type_AUIPC, J_Type, I_Type_JAL_R})
    8'b10000000,8'b01000000: Immediate = {{20{IR[31]}}, IR[31:20]};                    //I Type_Aritmetic and  I Type_Load
    8'b00100000: Immediate = {{20{IR[31]}}, IR[31:25], IR[11:7]};                        //S Type Store
    8'b00010000: Immediate = {{20{IR[31]}},IR[31],IR[7], IR[30:25], IR[11:8],{1'b0}};   //B Type Branch
    8'b00001000,8'b00000100 : Immediate = {IR[31:12], {12{1'b0}}};                      //U Type LUI AUIPC
    8'b00000010: Immediate = {{11{IR[31]}},IR[19:12], IR[20], IR[30:21],{1'b0}};         //J Type Jump
    8'b00000001: Immediate = {{20{IR[31]}}, IR[31:20]};                              //I Type_JAL_R
//default: Immediate = 0;
endcase
end
//==============================================================================================================================
```

5.4.1 Verification of immediate generator and format finder:

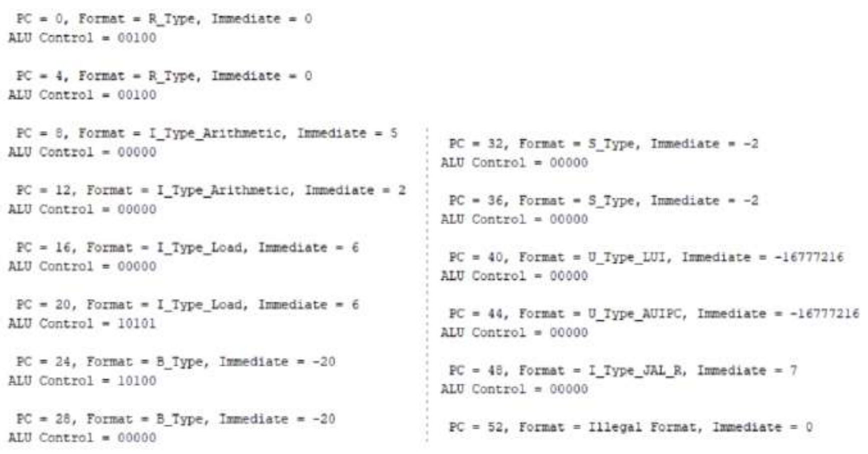


```
module ID_Test();
reg Clk, Reset;                     //Clock and Reset Signal
reg Is_Branch_Taken;                //Mux selection line for different format
reg [31:0] Address;

wire [31:0] Instruction_Fetch_IF;   //32 bit Instruction, fetched from Instruction Memory
wire [31:0] Pc_Add_Out_Ot;          //Output of PC Adder
wire [31:0] Pc_In_Ot;               //Input to PC Register
wire [31:0] PC_IF;                  //Output to PC Register

Stage_1 IF(Clk, Reset,Is_Branch_Taken,Address,Instruction_Fetch_IF,Pc_Add_Out_Ot,Pc_In_Ot,PC_IF);

 wire [31:0] Instruction_Register_ID;
 wire [31:0] PC_ID;
 wire [4:0] rs1_ID;
 wire [4:0] rs2_ID;
 wire [4:0] rd_ID;
 wire [6:0] Opcode_ID;
 wire [2:0] Func3_ID;

 IF_ID_Pipeline FD(Clk,Reset,Instruction_Fetch_IF,PC_IF,Instruction_Register_ID,PC_ID,rs1_ID,rs2_ID,rd_ID,Opcode_ID,Func3_ID);

wire [4:0] Alu_Cntrl_ID; //ALU Control
wire [31:0] Operand1_ACU_ID; //Operand 1 to Address ALU
wire [31:0] Operand2_ACU_ID; //Operand 2 to Data ALU
wire [31:0] Operand1_DEU_ID; //Operand 1 to Data ALU
wire [31:0] Operand2_DEU_ID; //Operand 2 to Address ALU
wire [6:0] Immediate_Format_ID;
wire Rs1_Valid_ID;
wire Rs2_Valid_ID;
wire Write_Enable_ID;
reg [31:0] Data_in;
reg [31:0] rd_WB;

Stage_2 s2(Clk,Reset,Read_Enable_1,Read_Enable_2,Write_Enable_WB,PC_ID,Data_in,rs1_ID,rs2_ID,rd_WB,
Instruction_Register_ID,Alu_Cntrl_ID,Operand1_ACU_ID,Operand2_ACU_ID,Operand1_DEU_ID,Operand2_DEU_ID,
Immediate_Format_ID,Rs1_Valid_ID,Rs2_Valid_ID,Write_Enable_ID);

 initial begin
 Clk=1'b0; Reset=1'b0; Address=0; Is_Branch_Taken=1'b0; Data_in=0;
 #2 Clk = 1'b0; Reset = 1'b1;
 #3 Clk = 1'b1; Reset = 1'b0;
 #5 Clk = 1'b0;
 #5 Clk = 1'b1;
 repeat(26)
 #5 Clk=~Clk;
 /*#5 Clk = 1'b0;
 #5 Clk = 1'b1;
 #5 Clk = 1'b0;*/
 #5 $finish;
 end
 endmodule
```

Simulation: These 14 arbitrary instructions are tested to verify the format type and immediate generator. Their corresponding simulations are also displayed in the command window. Both immediate and format type is verified. Last instruction is given as zero, to check whether it detects the illegal format or not.



PC Value (in Decimal)	Instruction
0	sub r1 r2 r3
4	xor r2 r4 r5
8	slti r4 r5 -6
12	ori r2 r5 2
16	lb r10 6(r5)
20	lh r10 6(r5)
24	lw r10 6(r5)
28	beq r5 r5 -20
32	bne r5 r5 -20
36	sb r5 -2(8)
40	lui r6 4096
44	auipc r8 4096
48	jalr r7 7(r1)
52	0



```
 PC = 0, Format = R_Type, Immediate = 0
ALU Control = 00100

 PC = 4, Format = R_Type, Immediate = 0
ALU Control = 00100

 PC = 8, Format = I_Type_Arithmetic, Immediate = 5
ALU Control = 00000

 PC = 12, Format = I_Type_Arithmetic, Immediate = 2
ALU Control = 00000

 PC = 16, Format = I_Type_Load, Immediate = 6
ALU Control = 00000

 PC = 20, Format = I_Type_Load, Immediate = 6
ALU Control = 10101

 PC = 24, Format = B_Type, Immediate = -20
ALU Control = 10100

 PC = 28, Format = B_Type, Immediate = -20
ALU Control = 00000

 PC = 32, Format = S_Type, Immediate = -2
ALU Control = 00000

 PC = 36, Format = S_Type, Immediate = -2
ALU Control = 00000

 PC = 40, Format = U_Type_LUI, Immediate = -16777216
ALU Control = 00000

 PC = 44, Format = U_Type_AUIPC, Immediate = -16777216
ALU Control = 00000

 PC = 48, Format = I_Type_JAL_R, Immediate = 7
ALU Control = 00000

 PC = 52, Format = Illegal Format, Immediate = 0
```

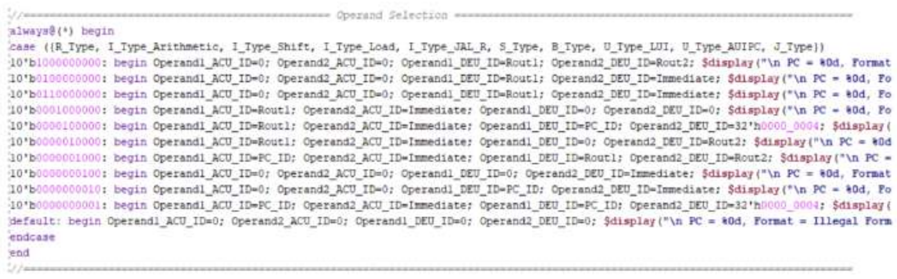
All the instruction format type and immediate generated are correct. This also ensure that Stage 1, If_ID_Pipeline and Stage 2 are verified.

5.5 Operand select module:

S.No	Operand_1_ACU	Operand_2_ACU	Operand_1_DEU	Operand_2_DEU	Instruction type
1	0	0	Rout1	Rout2	R-type
2	0	0	Rout1	immediate	I-type Arithmetic
3	0	0	Rout1	immediate	I-type Shift
4	Rout1	immediate	0	0	I-type Load
5	Rout1	immediate	PC	4	I-type JALR
6	Rout1	immediate	0	Rout2	S type
7	PC	immediate	Rout1	Rout2	B type
8	0	0	0	immediate	U type LUI
9	0	0	PC	immediate	U type AUIPC
10	PC	immediate	PC	4	J type JAL
11	0	0	0	0	Default

Depending on the type of instruction format which was found in the format finder the respective operand values (as shown in the above table) will be passed to the execution stage which houses two separate ALU units each one for address calculation and data calculation respectively. Any data which needs to written to register file in rd in the writeback stage will be calculated in Data Execution Unit while any address which need to be calculated to be sent to PC or the data memory will be calculated in the Address Calculation ALU unit. This was mainly done to avoid any structural hazards that may arise due to the pipelined approach that we have taken for our design. Structural hazards will be discussed in detail in a later chapter for pipeline hazards and mitigation techniques.

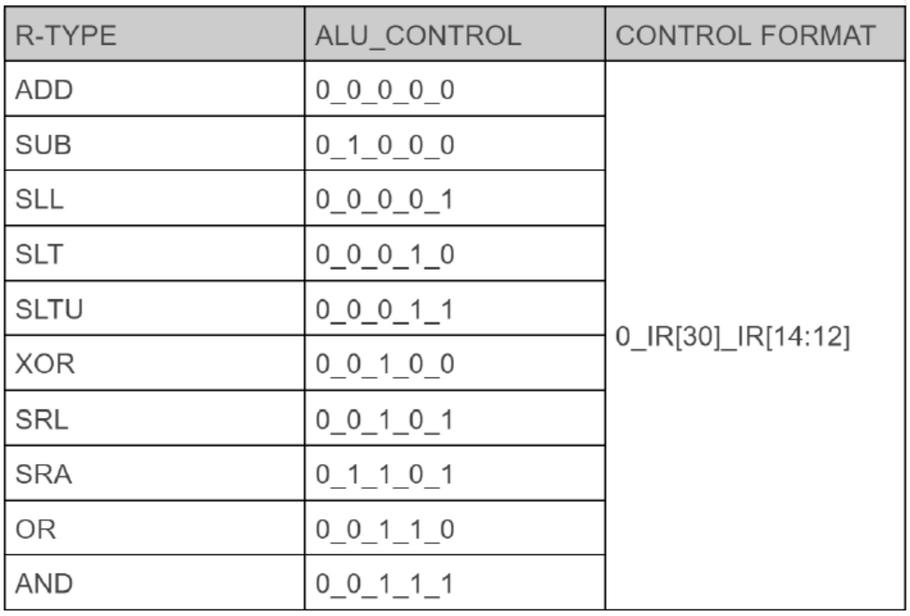
Code for Operation select module:



```
//================================================== Operand Selection ================================================================
always@(*) begin
case ({R_Type, I_Type_Arithmetic, I_Type_Shift, I_Type_Load, I_Type_JAL_R, S_Type, B_Type, U_Type_LUI, U_Type_AUIPC, J_Type})
10'b1000000000: begin Operand1_ACU_ID=0; Operand2_ACU_ID=0; Operand1_DEU_ID=Rout1; Operand2_DEU_ID=Rout2; $display("\n PC = %0d, Format
10'b0100000000: begin Operand1_ACU_ID=0; Operand2_ACU_ID=0; Operand1_DEU_ID=Rout1; Operand2_DEU_ID=Immediate; $display("\n PC = %0d, Fo
10'b0110000000: begin Operand1_ACU_ID=0; Operand2_ACU_ID=0; Operand1_DEU_ID=Rout1; Operand2_DEU_ID=Immediate; $display("\n PC = %0d, Fo
10'b0001000000: begin Operand1_ACU_ID=Rout1; Operand2_ACU_ID=Immediate; Operand1_DEU_ID=0; Operand2_DEU_ID=0; $display("\n PC = %0d, Fo
10'b0000100000: begin Operand1_ACU_ID=Rout1; Operand2_ACU_ID=Immediate; Operand1_DEU_ID=PC_ID; Operand2_DEU_ID=32'h0000_0004; $display(
10'b0000010000: begin Operand1_ACU_ID=Rout1; Operand2_ACU_ID=Immediate; Operand1_DEU_ID=0; Operand2_DEU_ID=Rout2; $display("\n PC = %0d
10'b0000001000: begin Operand1_ACU_ID=PC_ID; Operand2_ACU_ID=Immediate; Operand1_DEU_ID=Rout1; Operand2_DEU_ID=Rout2; $display("\n PC =
10'b0000000100: begin Operand1_ACU_ID=0; Operand2_ACU_ID=0; Operand1_DEU_ID=0; Operand2_DEU_ID=Immediate; $display("\n PC = %0d, Format
10'b0000000010: begin Operand1_ACU_ID=0; Operand2_ACU_ID=0; Operand1_DEU_ID=PC_ID; Operand2_DEU_ID=Immediate; $display("\n PC = %0d, Fo
10'b0000000001: begin Operand1_ACU_ID=PC_ID; Operand2_ACU_ID=Immediate; Operand1_DEU_ID=PC_ID; Operand2_DEU_ID=32'h0000_0004; $display(
default: begin Operand1_ACU_ID=0; Operand2_ACU_ID=0; Operand1_DEU_ID=0; Operand2_DEU_ID=0; $display("\n PC = %0d, Format = Illegal Form
endcase
end
//=====================================================================================================================================
```

5.6 Execution unit control

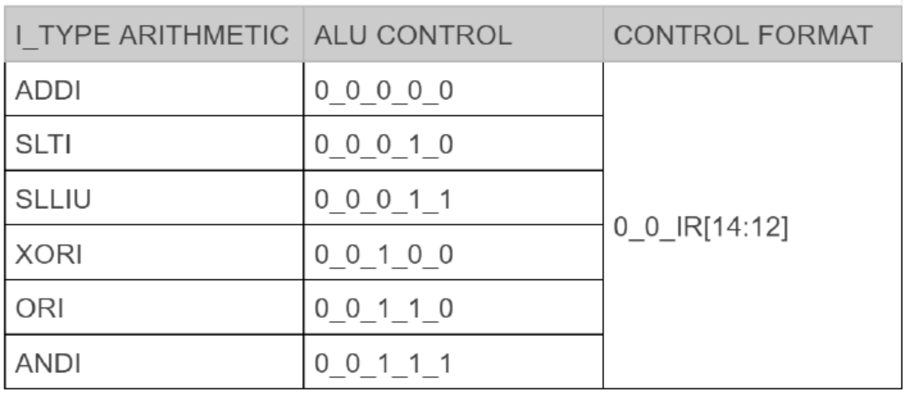
5.6.1 R type control



R-TYPE	ALU_CONTROL	CONTROL FORMAT
ADD	0_0_0_0_0	0_IR[30]_IR[14:12]
SUB	0_1_0_0_0	
SLL	0_0_0_0_1	
SLT	0_0_0_1_0	
SLTU	0_0_0_1_1	
XOR	0_0_1_0_0	
SRL	0_0_1_0_1	
SRA	0_1_1_0_1	
OR	0_0_1_1_0	
AND	0_0_1_1_1	

For R-type instructions, the main bit array to distinguish them from each other are the last four bits of the ALU control array. The MSB will be 0 and the remaining 4 bits are formed from IR[30] and IR[14:12] or func3 as it is called in the ISA.

5.6.2 I arithmetic control



I_TYPE ARITHMETIC	ALU CONTROL	CONTROL FORMAT
ADDI	0_0_0_0_0	0_0_IR[14:12]
SLTI	0_0_0_1_0	
SLLIU	0_0_0_1_1	
XORI	0_0_1_0_0	
ORI	0_0_1_1_0	
ANDI	0_0_1_1_1	

To distinguish between the I_type arithmetic instructions within themselves the last three bits of the ALU control can be used. The last three bits of the ALU control is formed from func3 or IR[14:12]. The MSB and the bit after will both be zero. Notice that ADD and ADDI might have same ALU control and they both differ in their operands so therefore the Operand_select module plays a crucial role here to pass the relevant operands according to the IR. The ALU merely operates on the operands received from the decode stage.

5.6.3 I type shift control

I_TYPE SHIFT	ALU CONTROL	CONTROL FORMAT
SLLI	0_0_0_0_1	0_IR[30]_IR[14:12]
SRLI	0_0_1_0_1	
SRAI	0_1_1_0_1	

For the I_type shift control they can be distinguished using the IR[30] bit and func3 array or IR[14:12].

5.6.4 B type control

B_TYPE	ALU CONTROL	CONTROL FORMAT
BEQ	1_0_0_0_0	1_0_IR[14:12]
BNE	1_0_0_0_1	
BLT	1_0_1_0_0	
BGE	1_0_1_0_1	
BLTU	1_0_1_1_0	
BGEU	1_0_1_1_1	

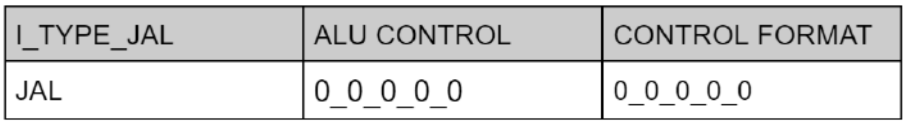
The branch instructions are distinguished within themselves using func3 array or IR[14:12]. The MSB and the next bit are kept at one and zero respectively. The MSB will be need to be forced to set high to distinguish the branch type instructions from other type instructions.

5.6.5 JALR type control

I_TYPE_JALR	ALU CONTROL	CONTROL FORMAT
JALR	0_0_0_0_0	0_0_0_0_0

JAL_R is a special case for ALU control as you dont just do ADD operation on the operands. After doing signed ADD operation on the operands to calculate the address you then need to bitwise AND the generated address with 32'bFFFFE. This is done so because as mentioned in the RV32I ISA, the address should be explicitly be forced to a even value.

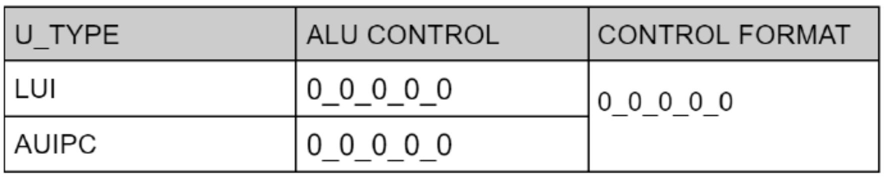
5.6.6 JAL type control



I_TYPE_JAL	ALU CONTROL	CONTROL FORMAT
JAL	0_0_0_0_0	0_0_0_0_0

Signed addition is the operation required for J-type instruction. PC gets added with the offset value to produce the jump target address.

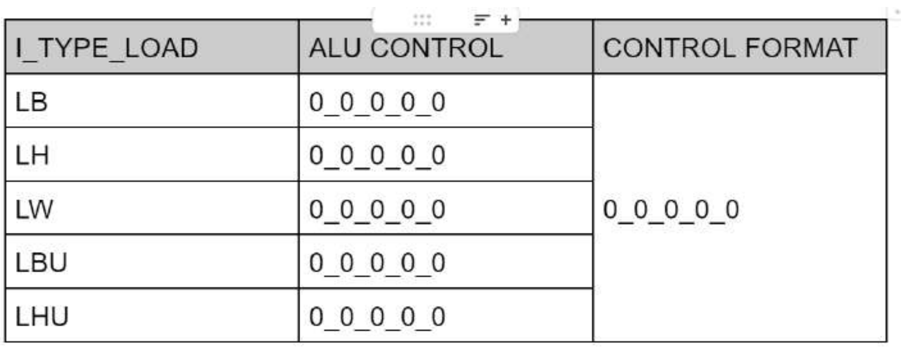
5.6.7 U type control



U_TYPE	ALU CONTROL	CONTROL FORMAT
LUI	0_0_0_0_0	0_0_0_0_0
AUIPC	0_0_0_0_0	

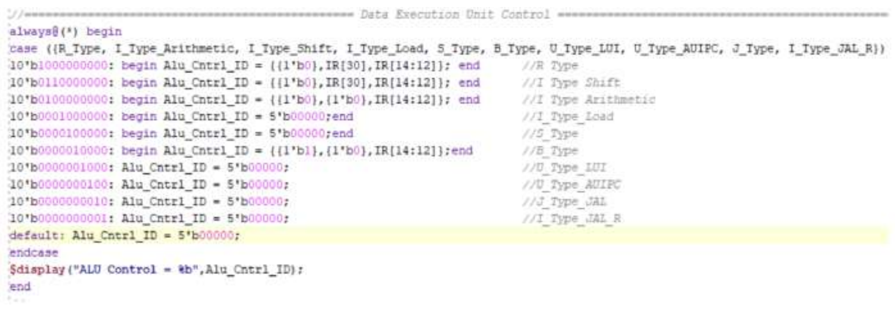
In AUIPC and LUI both the operation is ADD operation and therefore the ALU_control is 0_0_0_0_0.

5.6.8 I type Load control



I_TYPE_LOAD	ALU CONTROL	CONTROL FORMAT
LB	0_0_0_0_0	0_0_0_0_0
LH	0_0_0_0_0	
LW	0_0_0_0_0	
LBU	0_0_0_0_0	
LHU	0_0_0_0_0	

Code:

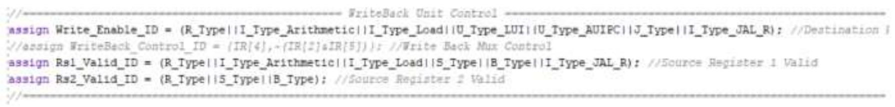


```
//================================================ Data Execution Unit Control ================================================
always@(*) begin
case ({R_Type, I_Type_Arithmetic, I_Type_Shift, I_Type_Load, S_Type, B_Type, U_Type_LUI, U_Type_AUIPC, J_Type, I_Type_JAL_R})
10'b1000000000: begin Alu_Cntrl_ID = {{1'b0},IR[30],IR[14:12]}; end      //R Type
10'b0110000000: begin Alu_Cntrl_ID = {{1'b0},IR[30],IR[14:12]}; end      //I Type Shift
10'b0100000000: begin Alu_Cntrl_ID = {{1'b0},{1'b0},IR[14:12]}; end      //I Type Arithmetic
10'b0001000000: begin Alu_Cntrl_ID = 5'b00000;end                         //I_Type_Load
10'b0000100000: begin Alu_Cntrl_ID = 5'b00000;end                         //S_Type
10'b0000010000: begin Alu_Cntrl_ID = {{1'b1},{1'b0},IR[14:12]};end       //B_Type
10'b0000001000: Alu_Cntrl_ID = 5'b00000;                                  //U_Type_LUI
10'b0000000100: Alu_Cntrl_ID = 5'b00000;                                  //U_Type_AUIPC
10'b0000000010: Alu_Cntrl_ID = 5'b00000;                                  //J_Type_JAL
10'b0000000001: Alu_Cntrl_ID = 5'b00000;                                  //I_Type_JAL_R
default: Alu_Cntrl_ID = 5'b00000;
endcase
$display("ALU Control = %b",Alu_Cntrl_ID);
end
```

5.7 Writeback Control

The writeback stage only needs one control i.e is_writeback signal. Based on the is_writeback signal either the Loaded_data from the memory stage is written back or the alu_out_data is written into the register file.

Code:



```
//------------------------------------------------ WriteBack Unit Control ------------------------------------------------------
assign Write_Enable_ID = (R_Type||I_Type_Arithmetic||I_Type_Load||U_Type_LUI||U_Type_AUIPC||J_Type||I_Type_JAL_R); //Destination R
//assign WriteBack_Control_ID = {IR[4],~(IR[2]&IR[5])}; //Write Back Mux Control
assign Rs1_Valid_ID = (R_Type||I_Type_Arithmetic||I_Type_Load||S_Type||B_Type||I_Type_JAL_R); //Source Register 1 Valid
assign Rs2_Valid_ID = (R_Type||S_Type||B_Type); //Source Register 2 Valid
//-------------------------------------------------------------------------------------------------------------------------------
```

5.8 Synthesis of stage 2

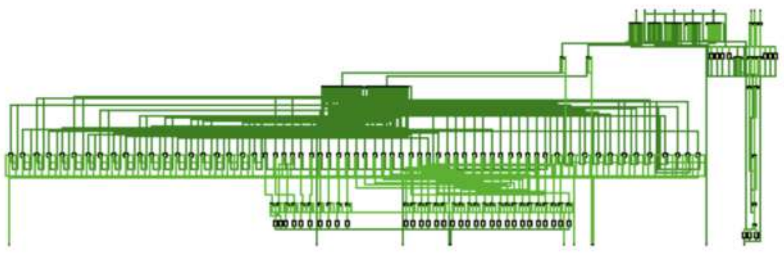


FIGURE 5.8: Stage 2 synthesis

5.9 Simulation of stage 2

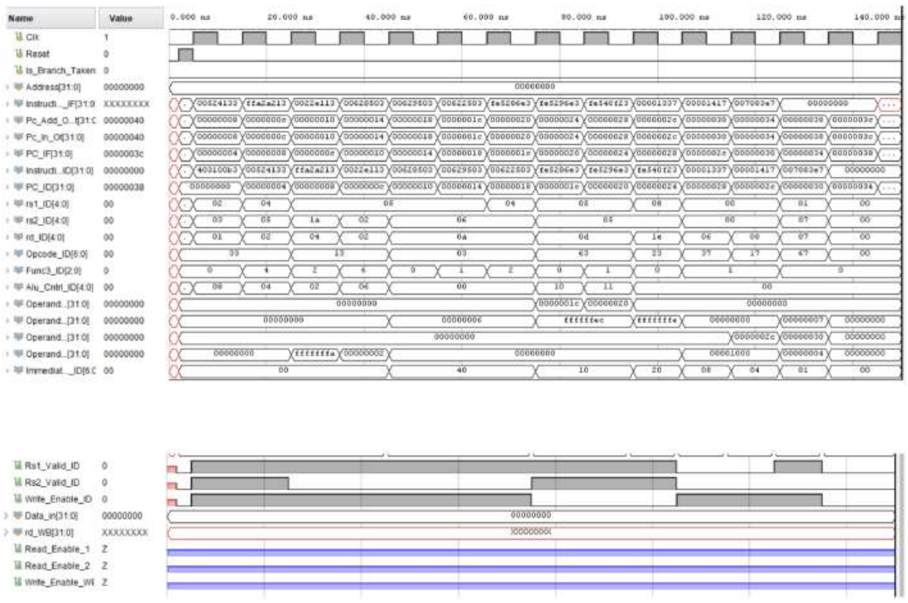
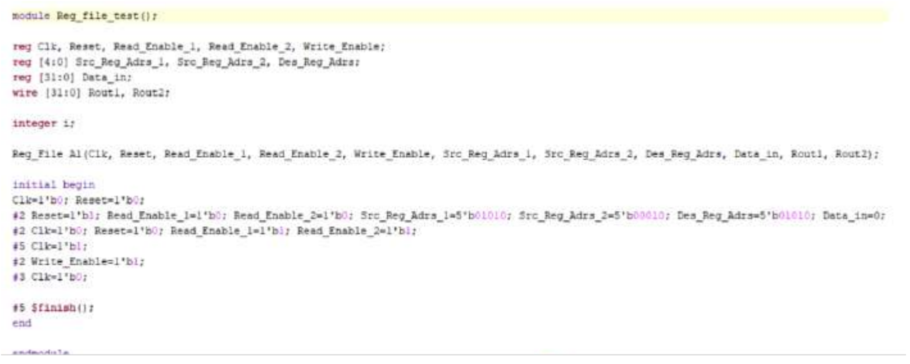


FIGURE 5.9: Stage 2 simulation

5.10 Simulation of register read/write

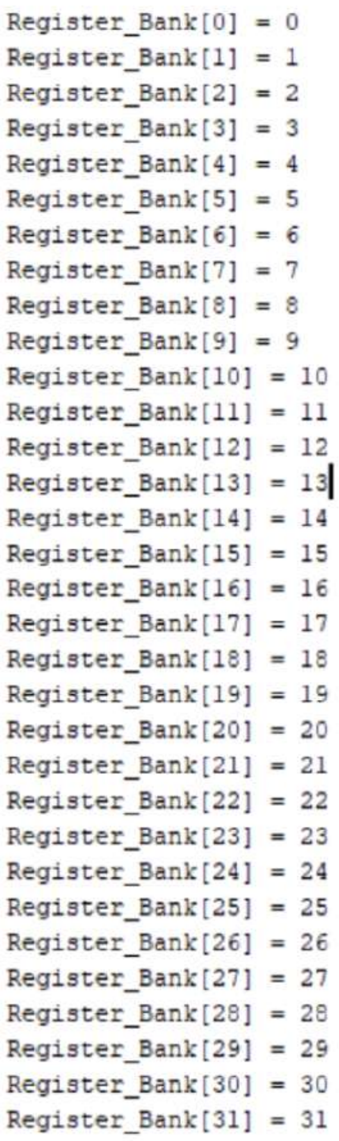


```
module Reg_file_test();

reg Clk, Reset, Read_Enable_1, Read_Enable_2, Write_Enable;
reg [4:0] Src_Reg_Adrs_1, Src_Reg_Adrs_2, Des_Reg_Adrs;
reg [31:0] Data_in;
wire [31:0] Rout1, Rout2;

integer i;

Reg_File A1(Clk, Reset, Read_Enable_1, Read_Enable_2, Write_Enable, Src_Reg_Adrs_1, Src_Reg_Adrs_2, Des_Reg_Adrs, Data_in, Rout1, Rout2);

initial begin
Clk=1'b0; Reset=1'b0;
#2 Reset=1'b1; Read_Enable_1=1'b0; Read_Enable_2=1'b0; Src_Reg_Adrs_1=5'b01010; Src_Reg_Adrs_2=5'b00010; Des_Reg_Adrs=5'b01010; Data_in=0;
#2 Clk=1'b0; Reset=1'b0; Read_Enable_1=1'b1; Read_Enable_2=1'b1;
#5 Clk=1'b1;
#2 Write_Enable=1'b1;
#3 Clk=1'b0;

#5 $finish();
end

endmodule
```

FIGURE 5.10: Register read/write testbench code

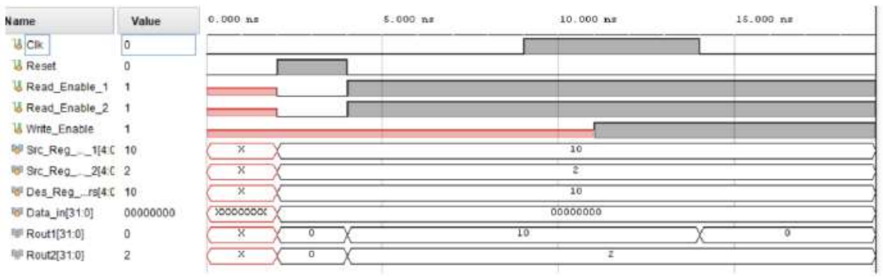
Then we read value from register address 10(rs1) and 2(rs1) which are shown in output Rout1 and Rout2. Then we write the 0 in address 10. All these results are shown in testbench and TCL Console.

Then we read value from register address 10(rs1) and 2(rs1) which are shown in output Rout1 and Rout2. Then we write the 0 in address 10. All these results are shown in testbench and TCL Console.



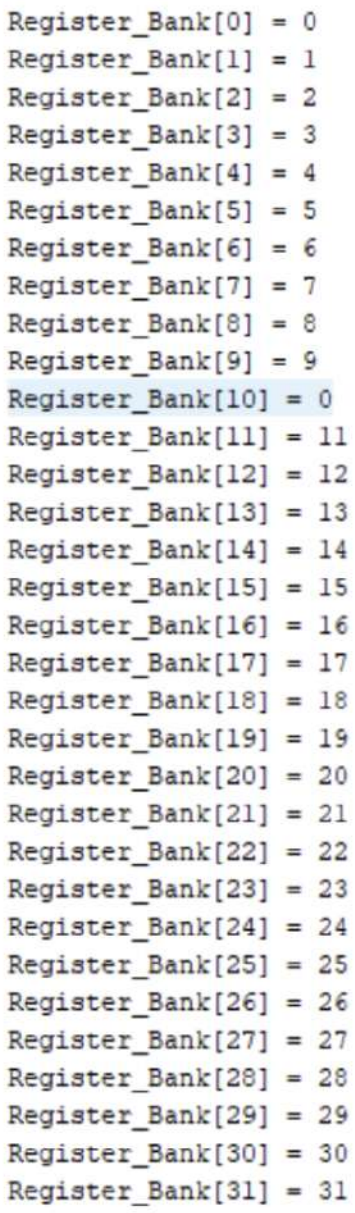
```
Register_Bank[0] = 0
Register_Bank[1] = 1
Register_Bank[2] = 2
Register_Bank[3] = 3
Register_Bank[4] = 4
Register_Bank[5] = 5
Register_Bank[6] = 6
Register_Bank[7] = 7
Register_Bank[8] = 8
Register_Bank[9] = 9
Register_Bank[10] = 10
Register_Bank[11] = 11
Register_Bank[12] = 12
Register_Bank[13] = 13
Register_Bank[14] = 14
Register_Bank[15] = 15
Register_Bank[16] = 16
Register_Bank[17] = 17
Register_Bank[18] = 18
Register_Bank[19] = 19
Register_Bank[20] = 20
Register_Bank[21] = 21
Register_Bank[22] = 22
Register_Bank[23] = 23
Register_Bank[24] = 24
Register_Bank[25] = 25
Register_Bank[26] = 26
Register_Bank[27] = 27
Register_Bank[28] = 28
Register_Bank[29] = 29
Register_Bank[30] = 30
Register_Bank[31] = 31
```

FIGURE 5.11: Register file after read operation



Name	Value
Clk	0
Reset	0
Read_Enable_1	1
Read_Enable_2	1
Write_Enable	1
Src_Reg_..._1[4:0	10
Src_Reg_..._2[4:0	2
Des_Reg_...rs[4:0	10
Data_in[31:0]	00000000
Rout1[31:0]	0
Rout2[31:0]	2

FIGURE 5.12: register read/write simulation



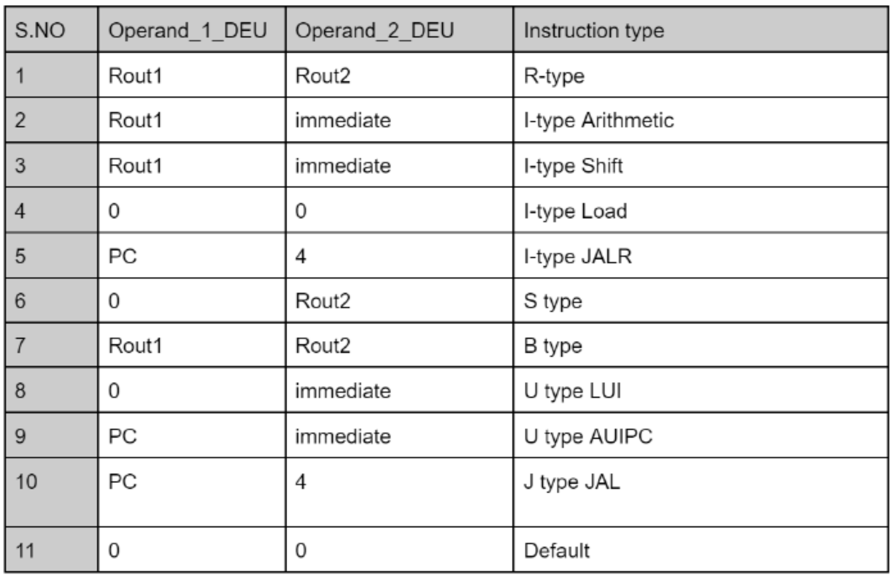
```
Register_Bank[0] = 0
Register_Bank[1] = 1
Register_Bank[2] = 2
Register_Bank[3] = 3
Register_Bank[4] = 4
Register_Bank[5] = 5
Register_Bank[6] = 6
Register_Bank[7] = 7
Register_Bank[8] = 8
Register_Bank[9] = 9
Register_Bank[10] = 0
Register_Bank[11] = 11
Register_Bank[12] = 12
Register_Bank[13] = 13
Register_Bank[14] = 14
Register_Bank[15] = 15
Register_Bank[16] = 16
Register_Bank[17] = 17
Register_Bank[18] = 18
Register_Bank[19] = 19
Register_Bank[20] = 20
Register_Bank[21] = 21
Register_Bank[22] = 22
Register_Bank[23] = 23
Register_Bank[24] = 24
Register_Bank[25] = 25
Register_Bank[26] = 26
Register_Bank[27] = 27
Register_Bank[28] = 28
Register_Bank[29] = 29
Register_Bank[30] = 30
Register_Bank[31] = 31
```

Figure 5.13: Register write operation

Chapter 6

Execute stage

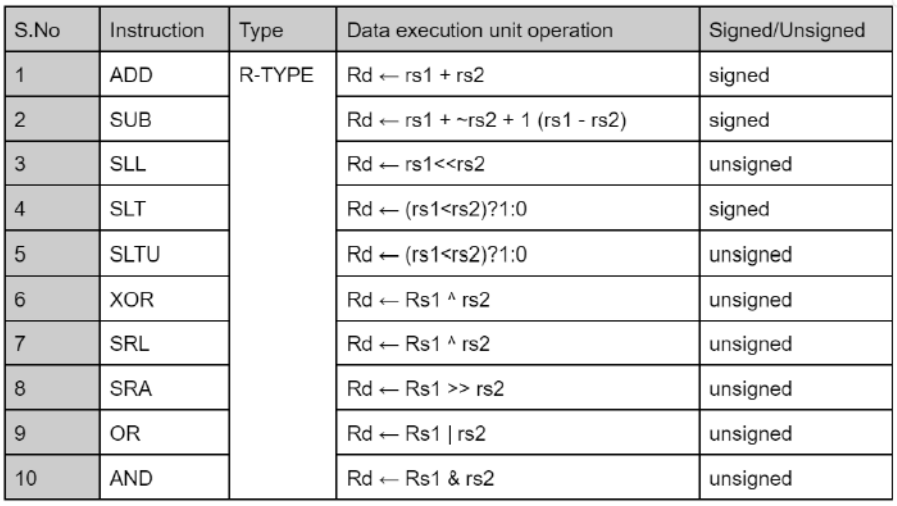
6.1 Data execution unit



S.NO	Operand_1_DEU	Operand_2_DEU	Instruction type
1	Rout1	Rout2	R-type
2	Rout1	immediate	I-type Arithmetic
3	Rout1	immediate	I-type Shift
4	0	0	I-type Load
5	PC	4	I-type JALR
6	0	Rout2	S type
7	Rout1	Rout2	B type
8	0	immediate	U type LUI
9	PC	immediate	U type AUIPC
10	PC	4	J type JAL
11	0	0	Default

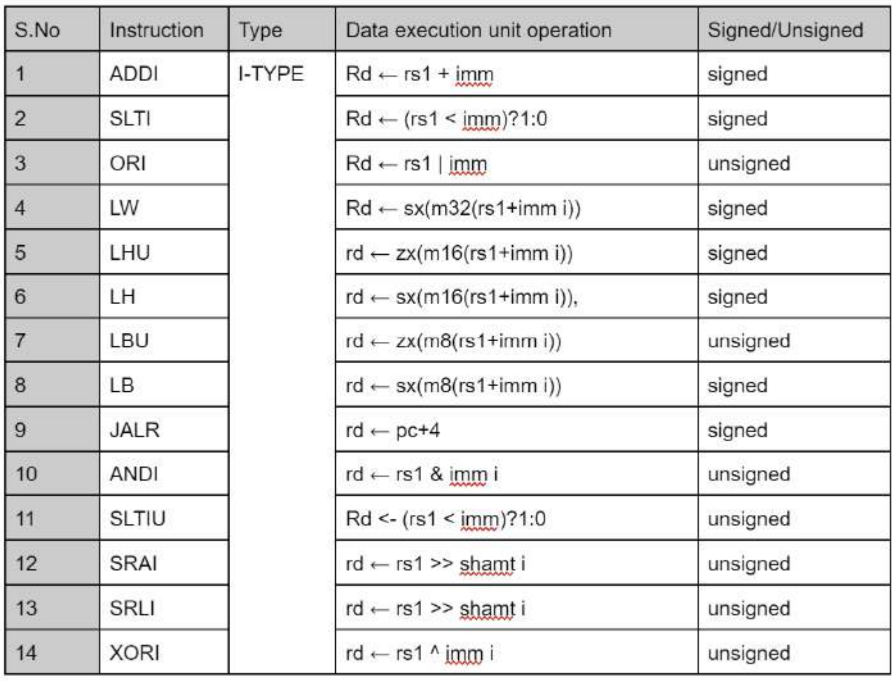
FIGURE 6.1: Data execution operand chart

The data execution unit operates on the data which needs to be written back to the register file in the writeback stage. As shown in the above table, the operands which were selected from the operand_select module from the decode stage would be passed to the data execution stage according to the instruction type which was decoded in the second stage. The table below lists the operation of the data execution unit for each of the instructions.



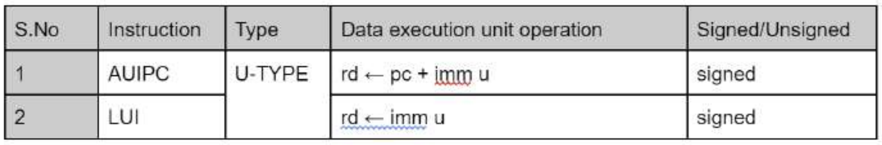
S.No	Instruction	Type	Data execution unit operation	Signed/Unsigned
1	ADD	R-TYPE	Rd ← rs1 + rs2	signed
2	SUB		Rd ← rs1 + ~rs2 + 1 (rs1 - rs2)	signed
3	SLL		Rd ← rs1<<rs2	unsigned
4	SLT		Rd ← (rs1<rs2)?1:0	signed
5	SLTU		Rd ← (rs1<rs2)?1:0	unsigned
6	XOR		Rd ← Rs1 ^ rs2	unsigned
7	SRL		Rd ← Rs1 ^ rs2	unsigned
8	SRA		Rd ← Rs1 >> rs2	unsigned
9	OR		Rd ← Rs1 | rs2	unsigned
10	AND		Rd ← Rs1 & rs2	unsigned

FIGURE 6.2: EU operation for R-type



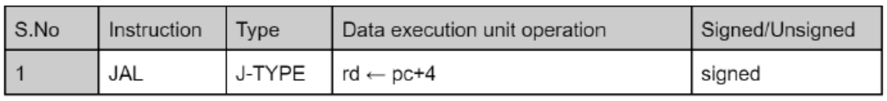
S.No	Instruction	Type	Data execution unit operation	Signed/Unsigned
1	ADDI	I-TYPE	Rd ← rs1 + imm	signed
2	SLTI		Rd ← (rs1 < imm)?1:0	signed
3	ORI		Rd ← rs1 | imm	unsigned
4	LW		Rd ← sx(m32(rs1+imm i))	signed
5	LHU		rd ← zx(m16(rs1+imm i))	signed
6	LH		rd ← sx(m16(rs1+imm i)),	signed
7	LBU		rd ← zx(m8(rs1+imm i))	unsigned
8	LB		rd ← sx(m8(rs1+imm i))	signed
9	JALR		rd ← pc+4	signed
10	ANDI		rd ← rs1 & imm i	unsigned
11	SLTIU		Rd <- (rs1 < imm)?1:0	unsigned
12	SRAI		rd ← rs1 >> shamt i	unsigned
13	SRLI		rd ← rs1 >> shamt i	unsigned
14	XORI		rd ← rs1 ^ imm i	unsigned

FIGURE 6.3: EU operation for I-type



S.No	Instruction	Type	Data execution unit operation	Signed/Unsigned
1	AUIPC	U-TYPE	rd ← pc + imm u	signed
2	LUI		rd ← imm u	signed

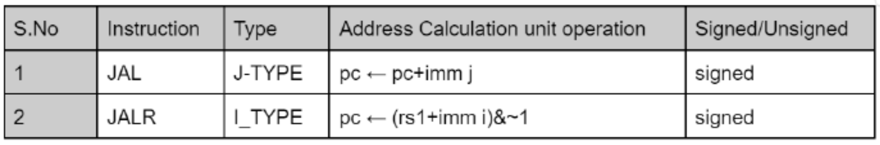
FIGURE 6.4: EU Operation for U-type



S.No	Instruction	Type	Data execution unit operation	Signed/Unsigned
1	JAL	J-TYPE	rd ← pc+4	signed

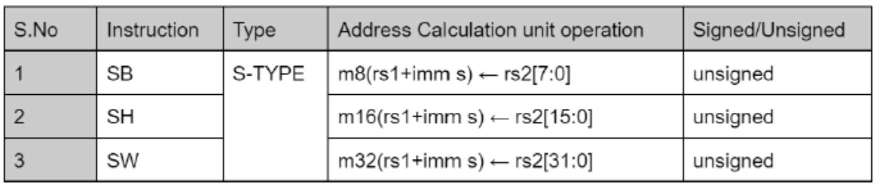
FIGURE 6.5: EU Operation for J-type

6.2 Address Calculation unit



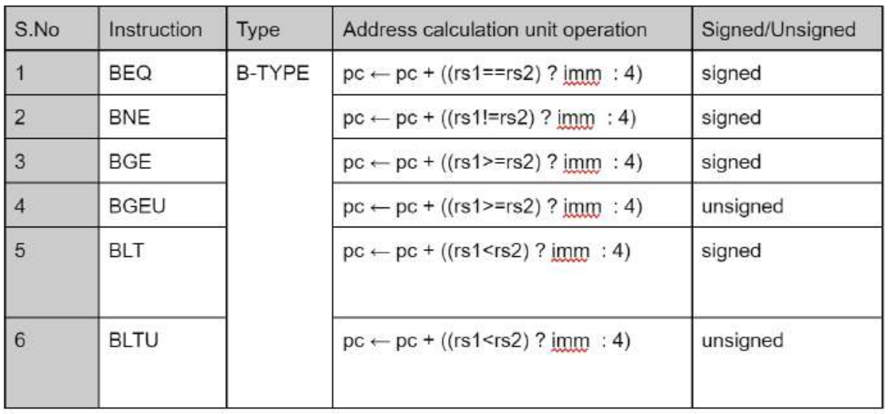
S.No	Instruction	Type	Address Calculation unit operation	Signed/Unsigned
1	JAL	J-TYPE	pc ← pc+imm j	signed
2	JALR	I_TYPE	pc ← (rs1+imm i)&~1	signed

Figure 6.6: ACU Operation for jalr and jal



S.No	Instruction	Type	Address Calculation unit operation	Signed/Unsigned
1	SB	S-TYPE	m8(rs1+imm s) ← rs2[7:0]	unsigned
2	SH		m16(rs1+imm s) ← rs2[15:0]	unsigned
3	SW		m32(rs1+imm s) ← rs2[31:0]	unsigned

Figure 6.7: ACU Operation for S-type



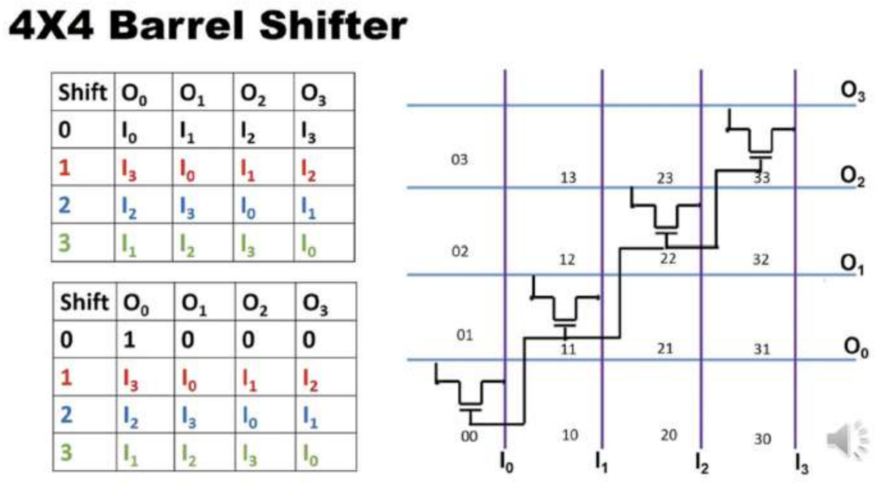
S.No	Instruction	Type	Address calculation unit operation	Signed/Unsigned
1	BEQ	B-TYPE	pc ← pc + ((rs1==rs2) ? imm : 4)	signed
2	BNE		pc ← pc + ((rs1!=rs2) ? imm : 4)	signed
3	BGE		pc ← pc + ((rs1>=rs2) ? imm : 4)	signed
4	BGEU		pc ← pc + ((rs1>=rs2) ? imm : 4)	unsigned
5	BLT		pc ← pc + ((rs1<rs2) ? imm : 4)	signed
6	BLTU		pc ← pc + ((rs1<rs2) ? imm : 4)	unsigned

Figure 6.8: ACU Operation for B-type

6.3 Barrel Shifter

Barrel Shifter shifts the binary data by a specific number of places in a single clock cycle. It uses a series of multiplexers controlled by a 2nd operand that defines the number of bits to be shifted to on the 1st operand. The output generated from this digital circuit is a shifted value. Normal Shift registers will shift data serially and use registers with more complex combinational units. However, the Barrel shifter is a combinational element with less chip area, low power, more accuracy, and less delay. It is mainly used in ALU for arithmetic and logical shifting.

Logical Shift Left: Shifting the first operand towards left and second operands defines the number of bits shifted. The result is trailed with zeros from the right side to make it 32-bit. Instruction – sll, slli Logical Shift Right: Shifting the first operand towards right and second operands defines



Shift	O_0	O_1	O_2	O_3
0	I_0	I_1	I_2	I_3
1	I_3	I_0	I_1	I_2
2	I_2	I_3	I_0	I_1
3	I_1	I_2	I_3	I_0

Shift	O_0	O_1	O_2	O_3
0	1	0	0	0
1	I_3	I_0	I_1	I_2
2	I_2	I_3	I_0	I_1
3	I_1	I_2	I_3	I_0

FIGURE 6.9: Barrel Shifter

the number of bits shifted. The result is filled with zeros from the left side to make it 32-bit. Instruction – srl, srli Arithmetic Shift Right: Shifting the first operand towards right and second operands defines the number of bits shifted. The result is sign extended to make it a 32-bit number. Instruction – sra, srai

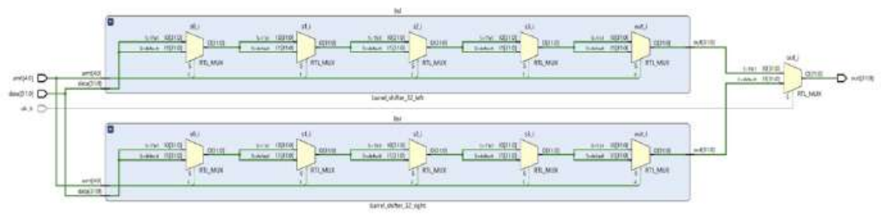
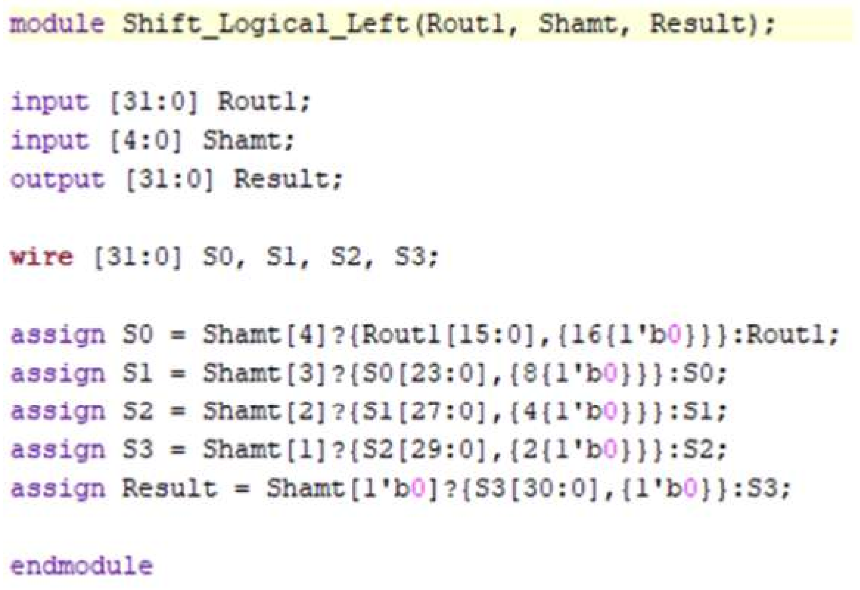


FIGURE 6.10: Barrel Shifter Synthesis

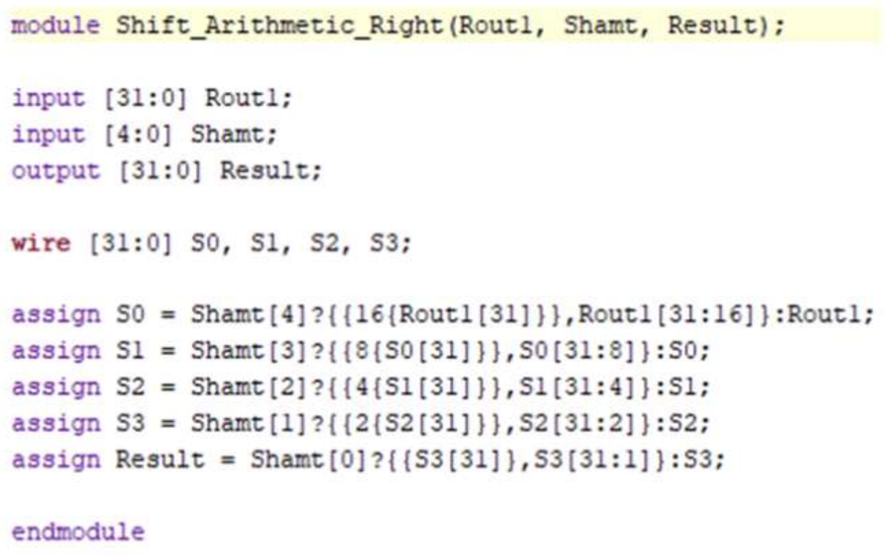
Both 1st Operand and 2nd operand are 32-bit unsigned numbers, but the shift amount is taken as the lowest 5 bits of the 2nd operand. ([4:0] Shamt = Operand_2 [4:0]) Algorithm for logical right shift: - · Operand1 is taken as input, If Shamt [4] is one then shift all 32-bits otherwise shift none. · The result from step 1 is taken as input, If Shamt [3] is one then shift all lower 16-bits or otherwise shift none. · The result from step 2 is taken as input, If Shamt [2] is one then shift all lower 8-bits or otherwise shift none. · The result from step 3 is taken as input, If Shamt [1] is one then shift all lower 4-bits or otherwise shift none. · The result from step 4 is taken as input, If Shamt [0] is one then shift all lower 2-bits or otherwise shift none. · The result from step 5 is the final result.

6.3.1 Code for SLL:



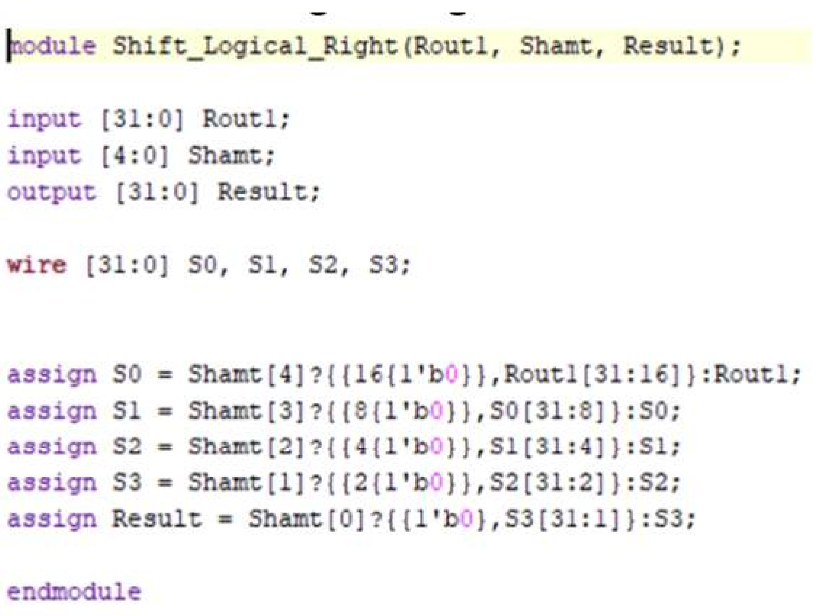
```verilog
module Shift_Logical_Left(Routl, Shamt, Result);

input [31:0] Routl;
input [4:0] Shamt;
output [31:0] Result;

wire [31:0] S0, S1, S2, S3;

assign S0 = Shamt[4]?{Routl[15:0],{16{1'b0}}}:Routl;
assign S1 = Shamt[3]?{S0[23:0],{8{1'b0}}}:S0;
assign S2 = Shamt[2]?{S1[27:0],{4{1'b0}}}:S1;
assign S3 = Shamt[1]?{S2[29:0],{2{1'b0}}}:S2;
assign Result = Shamt[1'b0]?{S3[30:0],{1'b0}}:S3;

endmodule
```

6.3.2 Code for SRA:



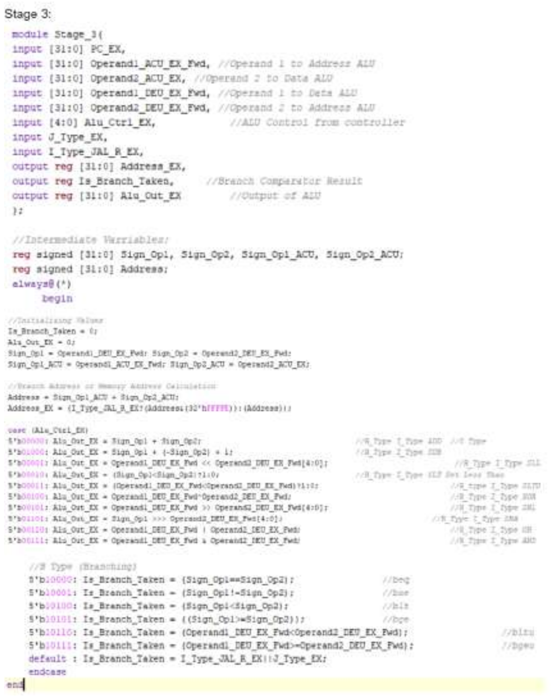
```verilog
module Shift_Arithmetic_Right(Routl, Shamt, Result);

input [31:0] Routl;
input [4:0] Shamt;
output [31:0] Result;

wire [31:0] S0, S1, S2, S3;

assign S0 = Shamt[4]?{{16{Routl[31]}},Routl[31:16]}:Routl;
assign S1 = Shamt[3]?{{8{S0[31]}},S0[31:8]}:S0;
assign S2 = Shamt[2]?{{4{S1[31]}},S1[31:4]}:S1;
assign S3 = Shamt[1]?{{2{S2[31]}},S2[31:2]}:S2;
assign Result = Shamt[0]?{{S3[31]},S3[31:1]}:S3;

endmodule
```

6.3.3 Code for SRL:



```
module Shift_Logical_Right(Rout1, Shamt, Result);

input [31:0] Rout1;
input [4:0] Shamt;
output [31:0] Result;

wire [31:0] S0, S1, S2, S3;


assign S0 = Shamt[4]?{{16{1'b0}},Rout1[31:16]}:Rout1;
assign S1 = Shamt[3]?{{8{1'b0}},S0[31:8]}:S0;
assign S2 = Shamt[2]?{{4{1'b0}},S1[31:4]}:S1;
assign S3 = Shamt[1]?{{2{1'b0}},S2[31:2]}:S2;
assign Result = Shamt[0]?{{1'b0},S3[31:1]}:S3;

endmodule
```

6.4 Verilog Code:



Stage 3:

```
module Stage_3(
input [31:0] PC_EX,
input [31:0] Operand1_ACU_EX_Fwd, //Operand 1 to Address ALU
input [31:0] Operand2_ACU_EX, //Operand 2 to Data ALU
input [31:0] Operand1_DEU_EX_Fwd, //Operand 1 to Data ALU
input [31:0] Operand2_DEU_EX_Fwd, //Operand 2 to Address ALU
input [4:0] Alu_Ctrl_EX,            //ALU Control from controller
input J_Type_EX,
input I_Type_JAL_R_EX,
output reg [31:0] Address_EX,
output reg Is_Branch_Taken,      //Branch Comparator Result
output reg [31:0] Alu_Out_EX        //Output of ALU
);

//Intermediate Varriables:
reg signed [31:0] Sign_Op1, Sign_Op2, Sign_Op1_ACU, Sign_Op2_ACU;
reg signed [31:0] Address;
always@(*)
     begin

//Initializing Values
Is_Branch_Taken = 0;
Alu_Out_EX = 0;
Sign_Op1 = Operand1_DEU_EX_Fwd; Sign_Op2 = Operand2_DEU_EX_Fwd;
Sign_Op1_ACU = Operand1_ACU_EX_Fwd; Sign_Op2_ACU = Operand2_ACU_EX;

//Branch Address or Memory Address Calculation
Address = Sign_Op1_ACU + Sign_Op2_ACU;
Address_EX = (I_Type_JAL_R_EX?(Address&(32'hFFFE)):(Address));

case (Alu_Ctrl_EX)
5'b00000: Alu_Out_EX = Sign_Op1 + Sign_Op2;                              //R_Type I_Type ADD //I Type
5'b01000: Alu_Out_EX = Sign_Op1 + (-Sign_Op2) + 1;                       //R_Type I_Type SUB
5'b00001: Alu_Out_EX = Operand1_DEU_EX_Fwd << Operand2_DEU_EX_Fwd[4:0];         //R_Type I_Type SLL
5'b00010: Alu_Out_EX = (Sign_Op1<Sign_Op2)?1:0;                          //R_Type I_Type SLT Set Less Than
5'b00011: Alu_Out_EX = (Operand1_DEU_EX_Fwd<Operand2_DEU_EX_Fwd)?1:0;            //R_type I_Type SLTU
5'b00100: Alu_Out_EX = Operand1_DEU_EX_Fwd^Operand2_DEU_EX_Fwd;                  //R_Type I_Type XOR
5'b00101: Alu_Out_EX = Operand1_DEU_EX_Fwd >> Operand2_DEU_EX_Fwd[4:0];          //R_Type I_Type SRL
5'b01101: Alu_Out_EX = Sign_Op1 >>> Operand2_DEU_EX_Fwd[4:0];                  //R_Type I_Type SRA
5'b00110: Alu_Out_EX = Operand1_DEU_EX_Fwd | Operand2_DEU_EX_Fwd;                //R_Type I_Type OR
5'b00111: Alu_Out_EX = Operand1_DEU_EX_Fwd & Operand2_DEU_EX_Fwd;                //R_Type I_Type AND

    //B Type (Branching)
    5'b10000: Is_Branch_Taken = (Sign_Op1==Sign_Op2);                    //beq
    5'b10001: Is_Branch_Taken = (Sign_Op1!=Sign_Op2);                    //bne
    5'b10100: Is_Branch_Taken = (Sign_Op1<Sign_Op2);                     //blt
    5'b10101: Is_Branch_Taken = ((Sign_Op1>=Sign_Op2));                  //bge
    5'b10110: Is_Branch_Taken = (Operand1_DEU_EX_Fwd<Operand2_DEU_EX_Fwd);          //bltu
    5'b10111: Is_Branch_Taken = (Operand1_DEU_EX_Fwd>=Operand2_DEU_EX_Fwd);         //bgeu
    default : Is_Branch_Taken = I_Type_JAL_R_EX||J_Type_EX;
    endcase
end
```

Testbech Code:

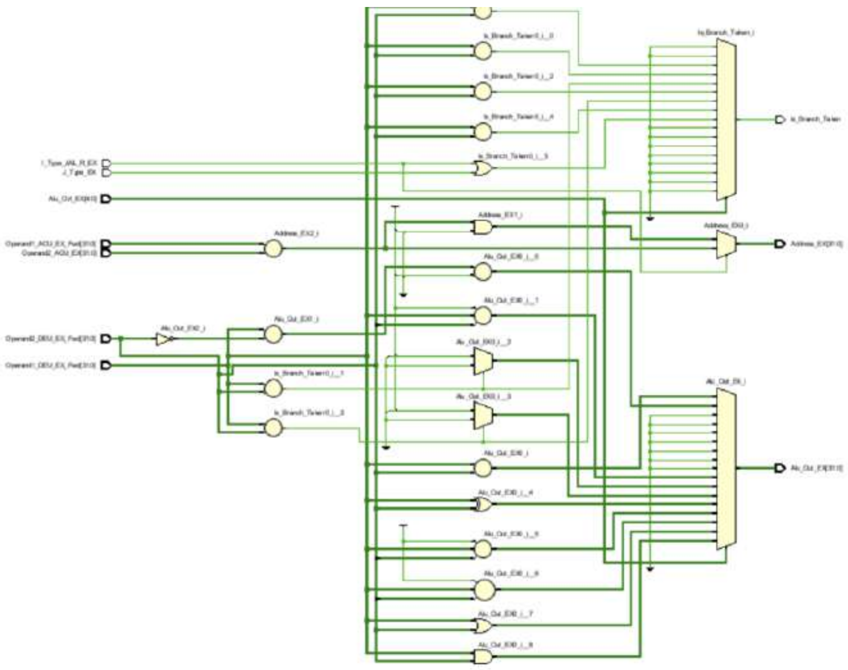
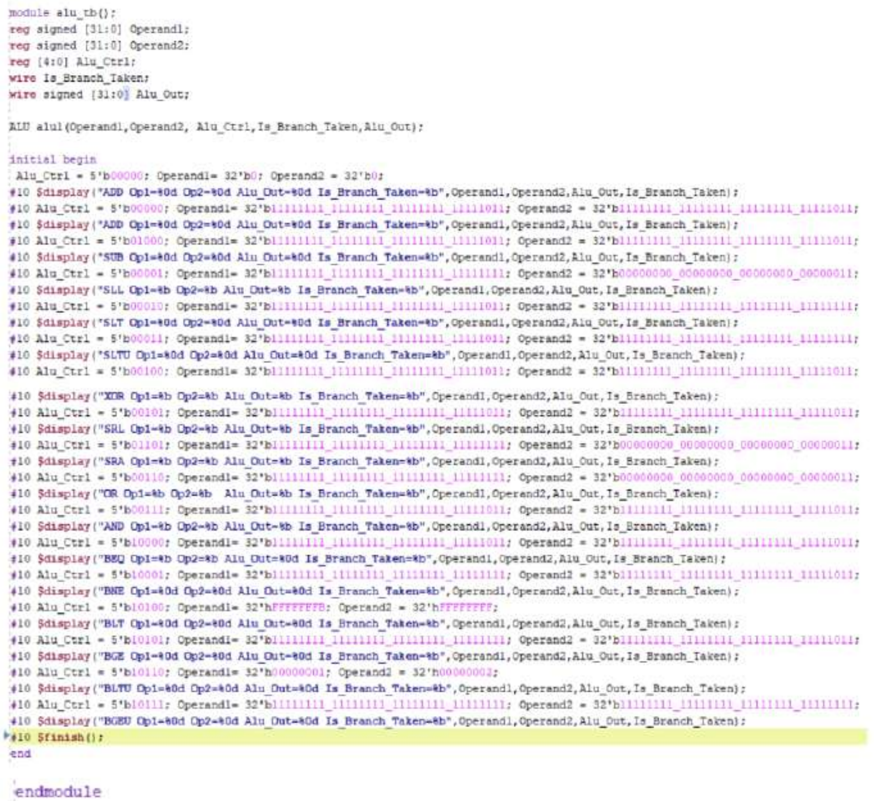
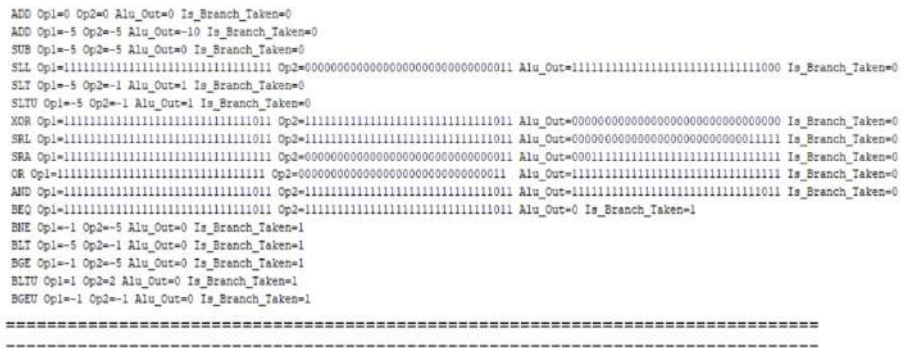


FIGURE 6.11: Stage 3 Schematic

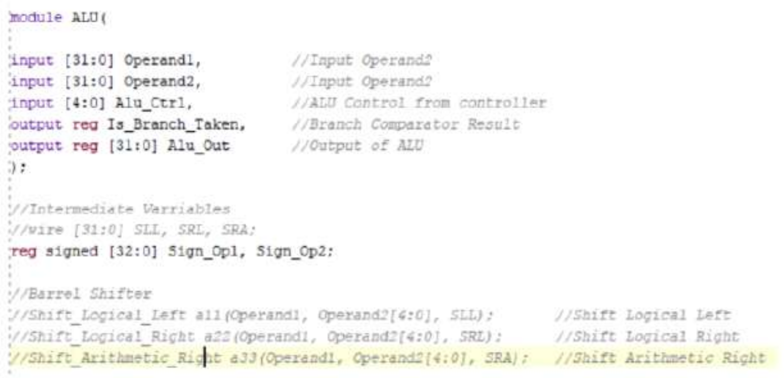


```
module alu_tb();
reg signed [31:0] Operand1;
reg signed [31:0] Operand2;
reg [4:0] Alu_Ctrl;
wire Is_Branch_Taken;
wire signed [31:0] Alu_Out;

ALU alu1(Operand1,Operand2, Alu_Ctrl,Is_Branch_Taken,Alu_Out);

initial begin
 Alu_Ctrl = 5'b00000; Operand1= 32'b0; Operand2 = 32'b0;
#10 $display("ADD Op1=%0d Op2=%0d Alu_Out=%0d Is_Branch_Taken=%b",Operand1,Operand2,Alu_Out,Is_Branch_Taken);
#10 Alu_Ctrl = 5'b00000; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 $display("ADD Op1=%0d Op2=%0d Alu_Out=%0d Is_Branch_Taken=%b",Operand1,Operand2,Alu_Out,Is_Branch_Taken);
#10 Alu_Ctrl = 5'b01000; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 $display("SUB Op1=%0d Op2=%0d Alu_Out=%0d Is_Branch_Taken=%b",Operand1,Operand2,Alu_Out,Is_Branch_Taken);
#10 Alu_Ctrl = 5'b00001; Operand1= 32'b11111111_11111111_11111111_11111111; Operand2 = 32'b00000000_00000000_00000000_00000011;
#10 $display("SLL Op1=%b Op2=%b Alu_Out=%b Is_Branch_Taken=%b",Operand1,Operand2,Alu_Out,Is_Branch_Taken);
#10 Alu_Ctrl = 5'b00010; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111111;
#10 $display("SLT Op1=%0d Op2=%0d Alu_Out=%0d Is_Branch_Taken=%b",Operand1,Operand2,Alu_Out,Is_Branch_Taken);
#10 Alu_Ctrl = 5'b00011; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111111;
#10 $display("SLTU Op1=%0d Op2=%0d Alu_Out=%0d Is_Branch_Taken=%b",Operand1,Operand2,Alu_Out,Is_Branch_Taken);
#10 Alu_Ctrl = 5'b00100; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;

#10 $display("XOR Op1=%b Op2=%b Alu_Out=%b Is_Branch_Taken=%b",Operand1,Operand2,Alu_Out,Is_Branch_Taken);
#10 Alu_Ctrl = 5'b00101; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 $display("SRL Op1=%b Op2=%b Alu_Out=%b Is_Branch_Taken=%b",Operand1,Operand2,Alu_Out,Is_Branch_Taken);
#10 Alu_Ctrl = 5'b01101; Operand1= 32'b11111111_11111111_11111111_11111111; Operand2 = 32'b00000000_00000000_00000000_00000011;
#10 $display("SRA Op1=%b Op2=%b Alu_Out=%b Is_Branch_Taken=%b",Operand1,Operand2,Alu_Out,Is_Branch_Taken);
#10 Alu_Ctrl = 5'b00110; Operand1= 32'b11111111_11111111_11111111_11111111; Operand2 = 32'b00000000_00000000_00000000_00000011;
#10 $display("OR Op1=%b Op2=%b  Alu_Out=%b Is_Branch_Taken=%b",Operand1,Operand2,Alu_Out,Is_Branch_Taken);
#10 Alu_Ctrl = 5'b00111; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 $display("AND Op1=%b Op2=%b Alu_Out=%b Is_Branch_Taken=%b",Operand1,Operand2,Alu_Out,Is_Branch_Taken);
#10 Alu_Ctrl = 5'b10000; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 $display("BEQ Op1=%b Op2=%b Alu_Out=%0d Is_Branch_Taken=%b",Operand1,Operand2,Alu_Out,Is_Branch_Taken);
#10 Alu_Ctrl = 5'b10001; Operand1= 32'b11111111_11111111_11111111_11111111; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 $display("BNE Op1=%0d Op2=%0d Alu_Out=%0d Is_Branch_Taken=%b",Operand1,Operand2,Alu_Out,Is_Branch_Taken);
#10 Alu_Ctrl = 5'b10100; Operand1= 32'hFFFFFFFB; Operand2 = 32'hFFFFFFFF;
#10 $display("BLT Op1=%0d Op2=%0d Alu_Out=%0d Is_Branch_Taken=%b",Operand1,Operand2,Alu_Out,Is_Branch_Taken);
#10 Alu_Ctrl = 5'b10101; Operand1= 32'b11111111_11111111_11111111_11111111; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 $display("BGE Op1=%0d Op2=%0d Alu_Out=%0d Is_Branch_Taken=%b",Operand1,Operand2,Alu_Out,Is_Branch_Taken);
#10 Alu_Ctrl = 5'b10110; Operand1= 32'h00000001; Operand2 = 32'h00000002;
#10 $display("BLTU Op1=%0d Op2=%0d Alu_Out=%0d Is_Branch_Taken=%b",Operand1,Operand2,Alu_Out,Is_Branch_Taken);
#10 Alu_Ctrl = 5'b10111; Operand1= 32'b11111111_11111111_11111111_11111111; Operand2 = 32'b11111111_11111111_11111111_11111111;
#10 $display("BGEU Op1=%0d Op2=%0d Alu_Out=%0d Is_Branch_Taken=%b",Operand1,Operand2,Alu_Out,Is_Branch_Taken);
#10 $finish();
end

endmodule
```

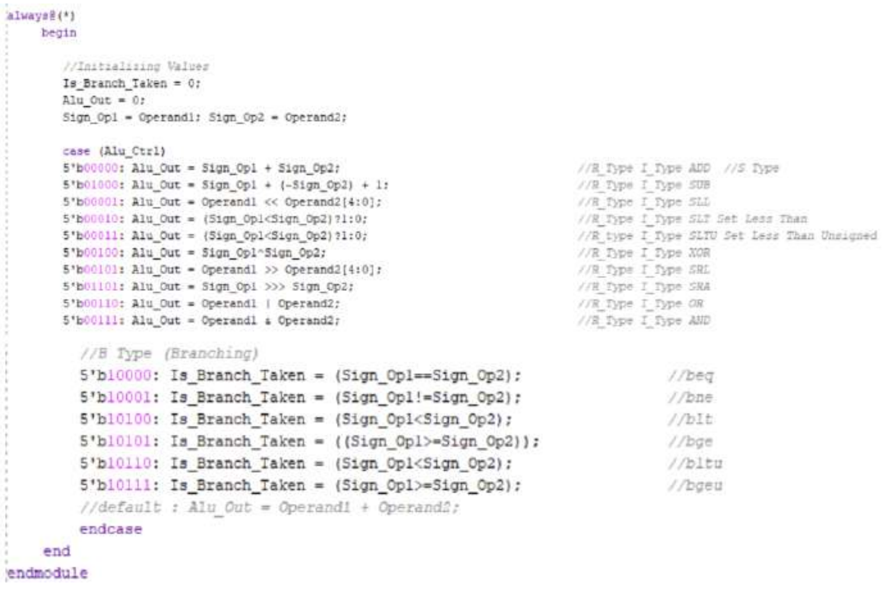


```
ADD Op1=0 Op2=0 Alu_Out=0 Is_Branch_Taken=0
ADD Op1=-5 Op2=-5 Alu_Out=-10 Is_Branch_Taken=0
SUB Op1=-5 Op2=-5 Alu_Out=0 Is_Branch_Taken=0
SLL Op1=11111111111111111111111111111111 Op2=00000000000000000000000000000011 Alu_Out=11111111111111111111111111111000 Is_Branch_Taken=0
SLT Op1=-5 Op2=-1 Alu_Out=1 Is_Branch_Taken=0
SLTU Op1=-5 Op2=-1 Alu_Out=1 Is_Branch_Taken=0
XOR Op1=11111111111111111111111111111011 Op2=11111111111111111111111111111011 Alu_Out=00000000000000000000000000000000 Is_Branch_Taken=0
SRL Op1=11111111111111111111111111111011 Op2=11111111111111111111111111111011 Alu_Out=00000000000000000000000000011111 Is_Branch_Taken=0
SRA Op1=11111111111111111111111111111111 Op2=00000000000000000000000000000011 Alu_Out=00011111111111111111111111111111 Is_Branch_Taken=0
OR Op1=11111111111111111111111111111111 Op2=00000000000000000000000000000011  Alu_Out=11111111111111111111111111111111 Is_Branch_Taken=0
AND Op1=11111111111111111111111111111011 Op2=11111111111111111111111111111011 Alu_Out=11111111111111111111111111111011 Is_Branch_Taken=0
BEQ Op1=11111111111111111111111111111011 Op2=11111111111111111111111111111011 Alu_Out=0 Is_Branch_Taken=1
BNE Op1=-1 Op2=-5 Alu_Out=0 Is_Branch_Taken=1
BLT Op1=-5 Op2=-1 Alu_Out=0 Is_Branch_Taken=1
BGE Op1=-1 Op2=-5 Alu_Out=0 Is_Branch_Taken=1
BLTU Op1=1 Op2=2 Alu_Out=0 Is_Branch_Taken=1
BGEU Op1=-1 Op2=-1 Alu_Out=0 Is_Branch_Taken=1

==============================================================================
------------------------------------------------------------------------------
```

6.5 Verilog Code:

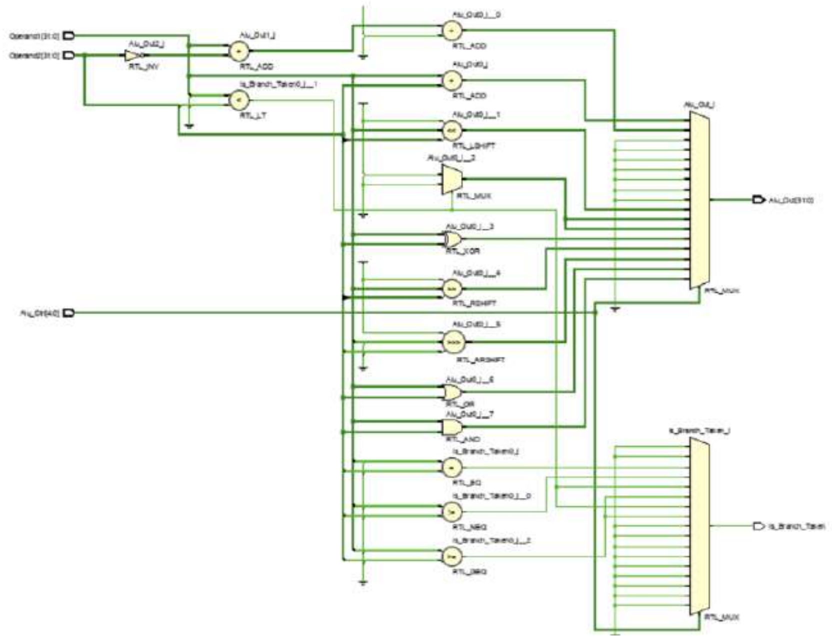


```
module ALU(

input [31:0] Operand1,          //Input Operand2
input [31:0] Operand2,          //Input Operand2
input [4:0] Alu_Ctrl,           //ALU Control from controller
output reg Is_Branch_Taken,     //Branch Comparator Result
output reg [31:0] Alu_Out       //Output of ALU
);

//Intermediate Varriables
//wire [31:0] SLL, SRL, SRA;
reg signed [32:0] Sign_Op1, Sign_Op2;

//Barrel Shifter
//Shift_Logical_Left a11(Operand1, Operand2[4:0], SLL);        //Shift Logical Left
//Shift_Logical_Right a22(Operand1, Operand2[4:0], SRL);       //Shift Logical Right
//Shift_Arithmetic_Right a33(Operand1, Operand2[4:0], SRA);    //Shift Arithmetic Right
```

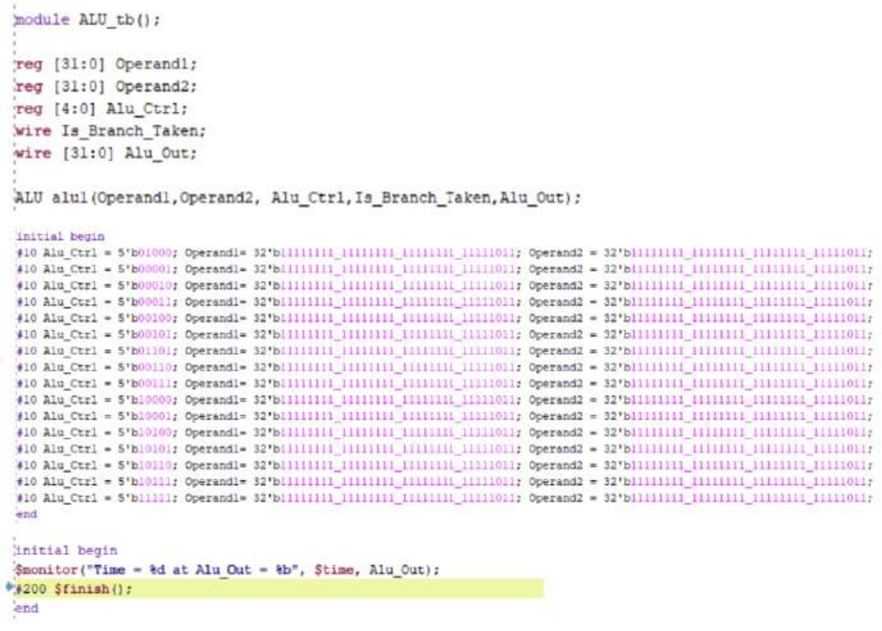


```
always@(*)
    begin

        //Initializing Values
        Is_Branch_Taken = 0;
        Alu_Out = 0;
        Sign_Op1 = Operand1; Sign_Op2 = Operand2;

        case (Alu_Ctrl)
        5'b00000: Alu_Out = Sign_Op1 + Sign_Op2;                  //R_Type I_Type ADD  //S Type
        5'b01000: Alu_Out = Sign_Op1 + (-Sign_Op2) + 1;           //R_Type I_Type SUB
        5'b00001: Alu_Out = Operand1 << Operand2[4:0];            //R_Type I_Type SLL
        5'b00010: Alu_Out = (Sign_Op1<Sign_Op2)?1:0;              //R_Type I_Type SLT Set Less Than
        5'b00011: Alu_Out = (Sign_Op1<Sign_Op2)?1:0;              //R_type I_Type SLTU Set Less Than Unsigned
        5'b00100: Alu_Out = Sign_Op1^Sign_Op2;                    //R_Type I_Type XOR
        5'b00101: Alu_Out = Operand1 >> Operand2[4:0];            //R_Type I_Type SRL
        5'b01101: Alu_Out = Sign_Op1 >>> Sign_Op2;                //R_Type I_Type SRA
        5'b00110: Alu_Out = Operand1 | Operand2;                  //R_Type I_Type OR
        5'b00111: Alu_Out = Operand1 & Operand2;                  //R_Type I_Type AND

         //B Type (Branching)
         5'b10000: Is_Branch_Taken = (Sign_Op1==Sign_Op2);        //beq
         5'b10001: Is_Branch_Taken = (Sign_Op1!=Sign_Op2);        //bne
         5'b10100: Is_Branch_Taken = (Sign_Op1<Sign_Op2);         //blt
         5'b10101: Is_Branch_Taken = ((Sign_Op1>=Sign_Op2));      //bge
         5'b10110: Is_Branch_Taken = (Sign_Op1<Sign_Op2);         //bltu
         5'b10111: Is_Branch_Taken = (Sign_Op1>=Sign_Op2);        //bgeu
         //default : Alu_Out = Operand1 + Operand2;
         endcase
     end
endmodule
```

6.6 RTL design:

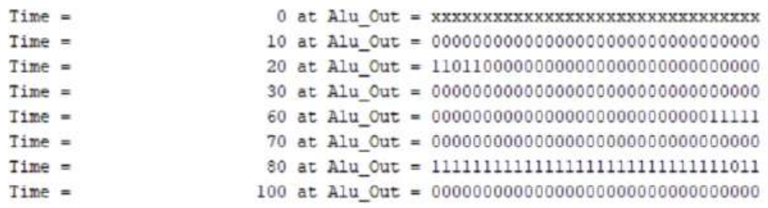


6.7 Testbench code:



```
module ALU_tb();

reg [31:0] Operand1;
reg [31:0] Operand2;
reg [4:0] Alu_Ctrl;
wire Is_Branch_Taken;
wire [31:0] Alu_Out;

ALU alu1(Operand1,Operand2, Alu_Ctrl,Is_Branch_Taken,Alu_Out);

initial begin
#10 Alu_Ctrl = 5'b01000; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 Alu_Ctrl = 5'b00001; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 Alu_Ctrl = 5'b00010; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 Alu_Ctrl = 5'b00011; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 Alu_Ctrl = 5'b00100; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 Alu_Ctrl = 5'b00101; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 Alu_Ctrl = 5'b01101; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 Alu_Ctrl = 5'b00110; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 Alu_Ctrl = 5'b00111; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 Alu_Ctrl = 5'b10000; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 Alu_Ctrl = 5'b10001; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 Alu_Ctrl = 5'b10100; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 Alu_Ctrl = 5'b10101; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 Alu_Ctrl = 5'b10110; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 Alu_Ctrl = 5'b10111; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;
#10 Alu_Ctrl = 5'b11111; Operand1= 32'b11111111_11111111_11111111_11111011; Operand2 = 32'b11111111_11111111_11111111_11111011;
end

initial begin
$monitor("Time = %d at Alu_Out = %b", $time, Alu_Out);
#200 $finish();
end
```

6.8 Testbench result:



```
Time =                    0 at Alu_Out = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Time =                   10 at Alu_Out = 00000000000000000000000000000000
Time =                   20 at Alu_Out = 11011000000000000000000000000000
Time =                   30 at Alu_Out = 00000000000000000000000000000000
Time =                   60 at Alu_Out = 00000000000000000000000000011111
Time =                   70 at Alu_Out = 00000000000000000000000000000000
Time =                   80 at Alu_Out = 11111111111111111111111111111011
Time =                  100 at Alu_Out = 00000000000000000000000000000000
```

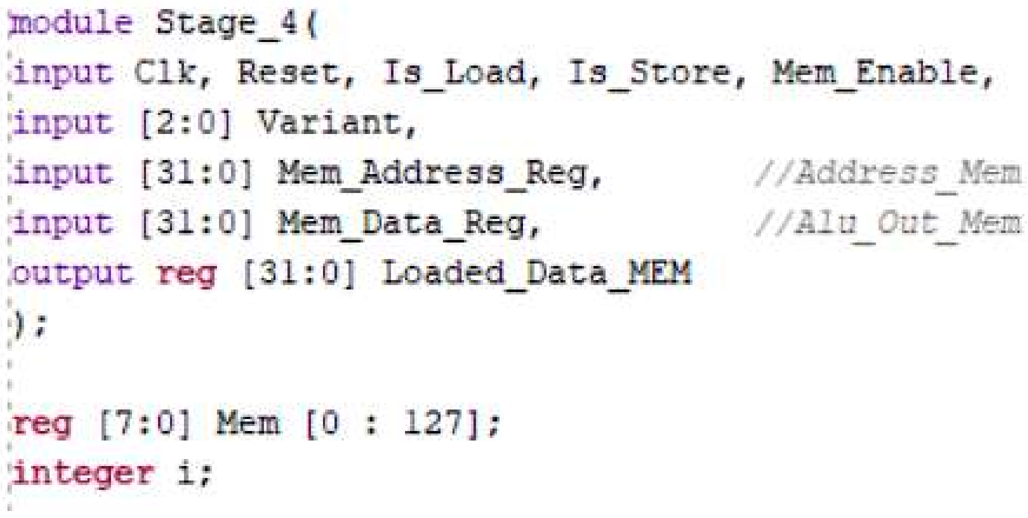
Chapter 7

Memory Access Stage

7.1 Verilog code

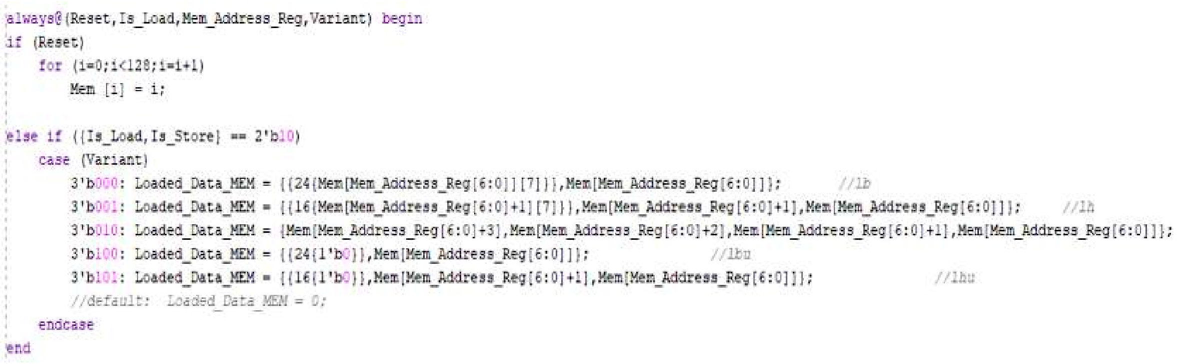
Verilog Code:

Stage 4:



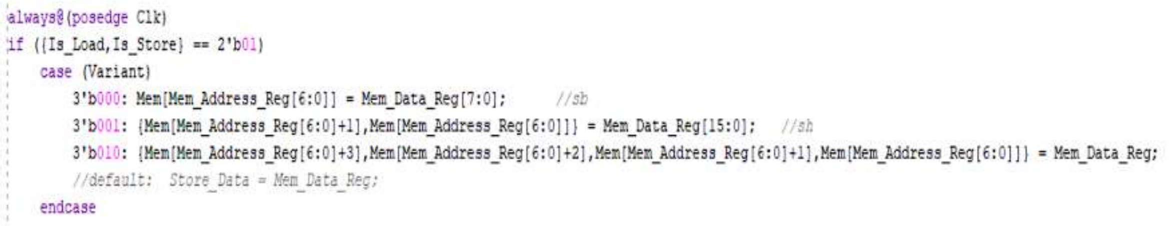
```
module Stage_4(
input Clk, Reset, Is_Load, Is_Store, Mem_Enable,
input [2:0] Variant,
input [31:0] Mem_Address_Reg,       //Address_Mem
input [31:0] Mem_Data_Reg,          //Alu_Out_Mem
output reg [31:0] Loaded_Data_MEM
);

reg [7:0] Mem [0 : 127];
integer i;
```

7.2 Load



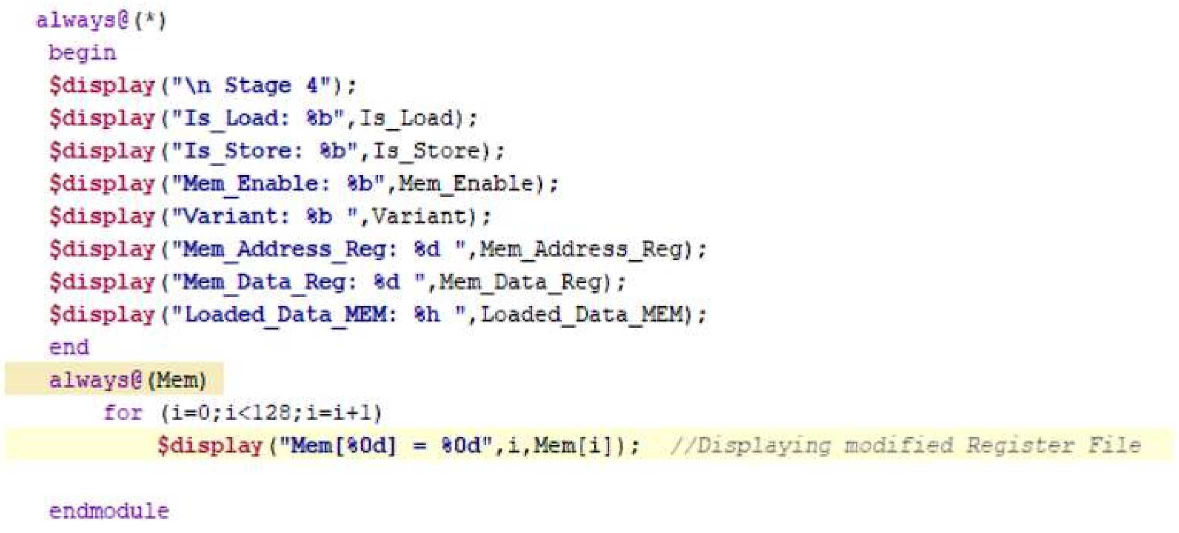
```
always@(Reset,Is_Load,Mem_Address_Reg,Variant) begin
if (Reset)
    for (i=0;i<128;i=i+1)
        Mem [i] = i;

else if ({Is_Load,Is_Store} == 2'b10)
    case (Variant)
        3'b000: Loaded_Data_MEM = {{24{Mem[Mem_Address_Reg[6:0]][7]}},Mem[Mem_Address_Reg[6:0]]};       //lb
        3'b001: Loaded_Data_MEM = {{16{Mem[Mem_Address_Reg[6:0]+1][7]}},Mem[Mem_Address_Reg[6:0]+1],Mem[Mem_Address_Reg[6:0]]};     //lh
        3'b010: Loaded_Data_MEM = {Mem[Mem_Address_Reg[6:0]+3],Mem[Mem_Address_Reg[6:0]+2],Mem[Mem_Address_Reg[6:0]+1],Mem[Mem_Address_Reg[6:0]]};
        3'b100: Loaded_Data_MEM = {{24{1'b0}},Mem[Mem_Address_Reg[6:0]]};                //lbu
        3'b101: Loaded_Data_MEM = {{16{1'b0}},Mem[Mem_Address_Reg[6:0]+1],Mem[Mem_Address_Reg[6:0]]};              //lhu
        //default:  Loaded_Data_MEM = 0;
    endcase
end
```

7.3 Store



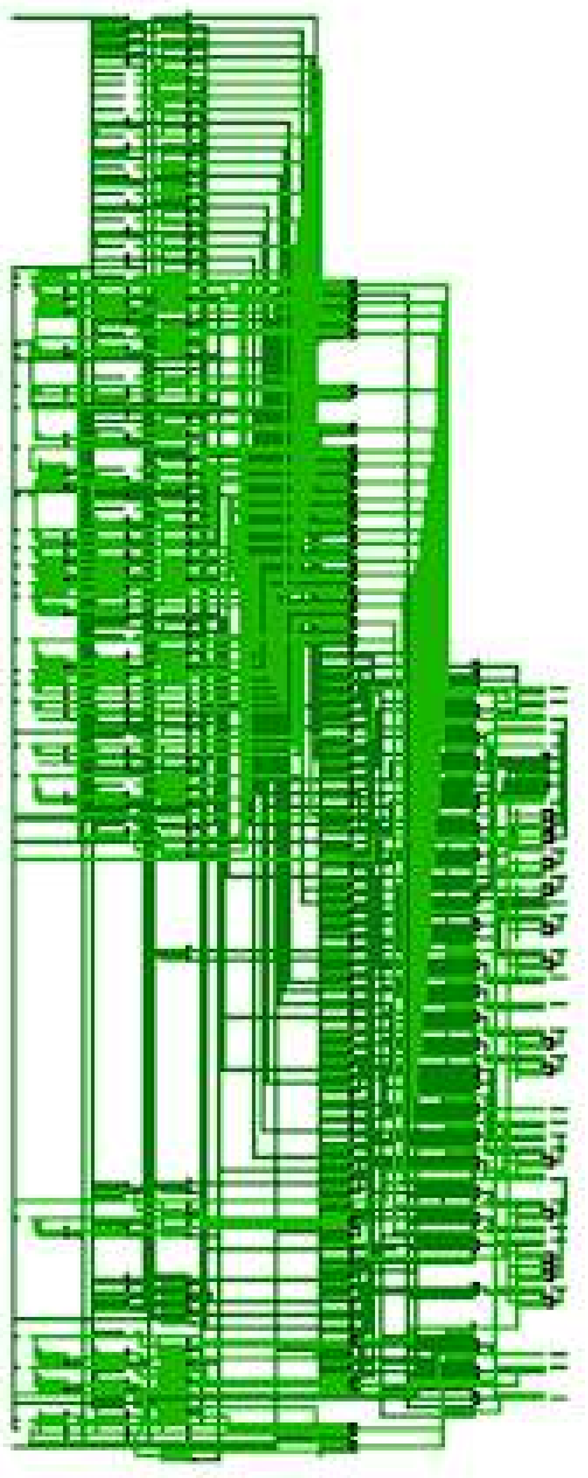
```
always@(posedge Clk)
if ({Is_Load,Is_Store} == 2'b01)
    case (Variant)
        3'b000: Mem[Mem_Address_Reg[6:0]] = Mem_Data_Reg[7:0];      //sb
        3'b001: {Mem[Mem_Address_Reg[6:0]+1],Mem[Mem_Address_Reg[6:0]]} = Mem_Data_Reg[15:0];   //sh
        3'b010: {Mem[Mem_Address_Reg[6:0]+3],Mem[Mem_Address_Reg[6:0]+2],Mem[Mem_Address_Reg[6:0]+1],Mem[Mem_Address_Reg[6:0]]} = Mem_Data_Reg;
        //default:  Store_Data = Mem_Data_Reg;
    endcase
```

7.4 Display Outputs for Comparison

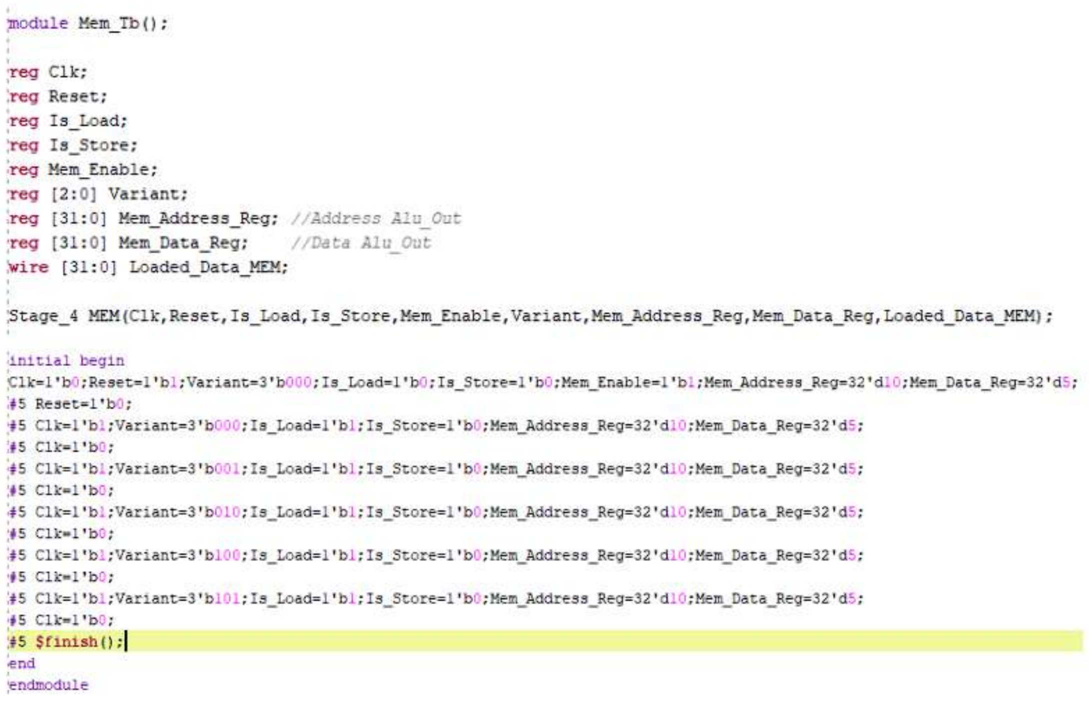


```
always@(*)
 begin
 $display("\n Stage 4");
 $display("Is_Load: %b",Is_Load);
 $display("Is_Store: %b",Is_Store);
 $display("Mem_Enable: %b",Mem_Enable);
 $display("Variant: %b ",Variant);
 $display("Mem_Address_Reg: %d ",Mem_Address_Reg);
 $display("Mem_Data_Reg: %d ",Mem_Data_Reg);
 $display("Loaded_Data_MEM: %h ",Loaded_Data_MEM);
 end
 always@(Mem)
     for (i=0;i<128;i=i+1)
         $display("Mem[%0d] = %0d",i,Mem[i]);  //Displaying modified Register File

 endmodule
```

7.5 Schematic



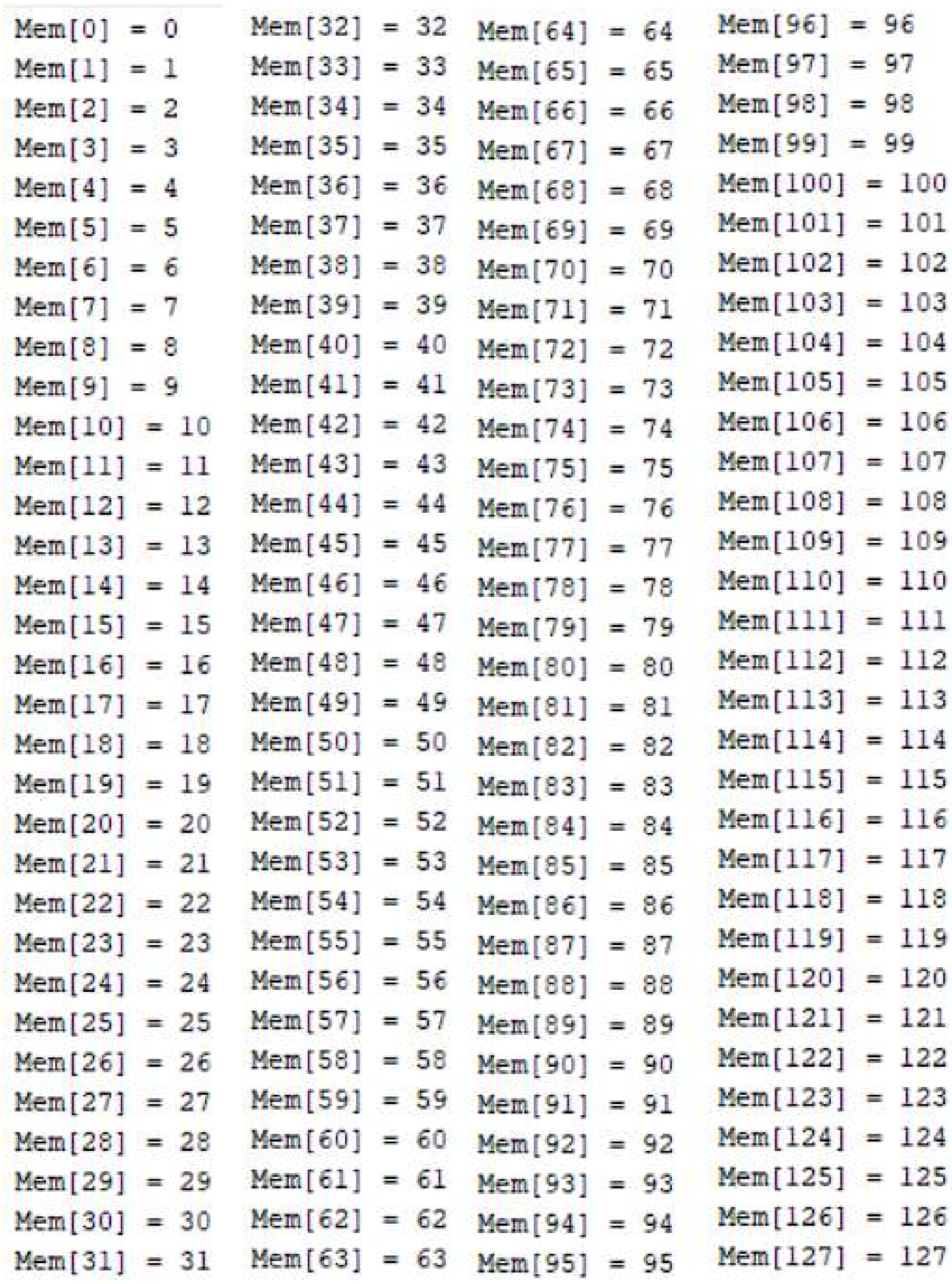
7.6 Testbench



```
module Mem_Tb();

reg Clk;
reg Reset;
reg Is_Load;
reg Is_Store;
reg Mem_Enable;
reg [2:0] Variant;
reg [31:0] Mem_Address_Reg; //Address Alu_Out
reg [31:0] Mem_Data_Reg;    //Data Alu_Out
wire [31:0] Loaded_Data_MEM;

Stage_4 MEM(Clk,Reset,Is_Load,Is_Store,Mem_Enable,Variant,Mem_Address_Reg,Mem_Data_Reg,Loaded_Data_MEM);

initial begin
Clk=1'b0;Reset=1'b1;Variant=3'b000;Is_Load=1'b0;Is_Store=1'b0;Mem_Enable=1'b1;Mem_Address_Reg=32'd10;Mem_Data_Reg=32'd5;
#5 Reset=1'b0;
#5 Clk=1'b1;Variant=3'b000;Is_Load=1'b1;Is_Store=1'b0;Mem_Address_Reg=32'd10;Mem_Data_Reg=32'd5;
#5 Clk=1'b0;
#5 Clk=1'b1;Variant=3'b001;Is_Load=1'b1;Is_Store=1'b0;Mem_Address_Reg=32'd10;Mem_Data_Reg=32'd5;
#5 Clk=1'b0;
#5 Clk=1'b1;Variant=3'b010;Is_Load=1'b1;Is_Store=1'b0;Mem_Address_Reg=32'd10;Mem_Data_Reg=32'd5;
#5 Clk=1'b0;
#5 Clk=1'b1;Variant=3'b100;Is_Load=1'b1;Is_Store=1'b0;Mem_Address_Reg=32'd10;Mem_Data_Reg=32'd5;
#5 Clk=1'b0;
#5 Clk=1'b1;Variant=3'b101;Is_Load=1'b1;Is_Store=1'b0;Mem_Address_Reg=32'd10;Mem_Data_Reg=32'd5;
#5 Clk=1'b0;
#5 $finish();
end
endmodule
```

7.7 Results and Discussionn

We have designed a 128-byte data memory. This memory is initialized as their index number as a value. It has control signals such as Memory enable, variant, Is-Store and Is-Load. Reading is asynchronous and writing is synchronous with the clock.



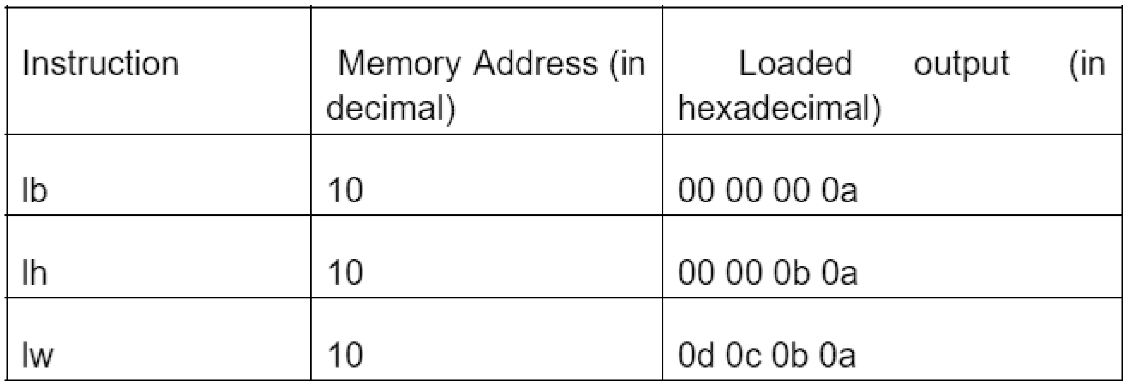
```
Mem[0] = 0      Mem[32] = 32   Mem[64] = 64   Mem[96] = 96
Mem[1] = 1      Mem[33] = 33   Mem[65] = 65   Mem[97] = 97
Mem[2] = 2      Mem[34] = 34   Mem[66] = 66   Mem[98] = 98
Mem[3] = 3      Mem[35] = 35   Mem[67] = 67   Mem[99] = 99
Mem[4] = 4      Mem[36] = 36   Mem[68] = 68   Mem[100] = 100
Mem[5] = 5      Mem[37] = 37   Mem[69] = 69   Mem[101] = 101
Mem[6] = 6      Mem[38] = 38   Mem[70] = 70   Mem[102] = 102
Mem[7] = 7      Mem[39] = 39   Mem[71] = 71   Mem[103] = 103
Mem[8] = 8      Mem[40] = 40   Mem[72] = 72   Mem[104] = 104
Mem[9] = 9      Mem[41] = 41   Mem[73] = 73   Mem[105] = 105
Mem[10] = 10    Mem[42] = 42   Mem[74] = 74   Mem[106] = 106
Mem[11] = 11    Mem[43] = 43   Mem[75] = 75   Mem[107] = 107
Mem[12] = 12    Mem[44] = 44   Mem[76] = 76   Mem[108] = 108
Mem[13] = 13    Mem[45] = 45   Mem[77] = 77   Mem[109] = 109
Mem[14] = 14    Mem[46] = 46   Mem[78] = 78   Mem[110] = 110
Mem[15] = 15    Mem[47] = 47   Mem[79] = 79   Mem[111] = 111
Mem[16] = 16    Mem[48] = 48   Mem[80] = 80   Mem[112] = 112
Mem[17] = 17    Mem[49] = 49   Mem[81] = 81   Mem[113] = 113
Mem[18] = 18    Mem[50] = 50   Mem[82] = 82   Mem[114] = 114
Mem[19] = 19    Mem[51] = 51   Mem[83] = 83   Mem[115] = 115
Mem[20] = 20    Mem[52] = 52   Mem[84] = 84   Mem[116] = 116
Mem[21] = 21    Mem[53] = 53   Mem[85] = 85   Mem[117] = 117
Mem[22] = 22    Mem[54] = 54   Mem[86] = 86   Mem[118] = 118
Mem[23] = 23    Mem[55] = 55   Mem[87] = 87   Mem[119] = 119
Mem[24] = 24    Mem[56] = 56   Mem[88] = 88   Mem[120] = 120
Mem[25] = 25    Mem[57] = 57   Mem[89] = 89   Mem[121] = 121
Mem[26] = 26    Mem[58] = 58   Mem[90] = 90   Mem[122] = 122
Mem[27] = 27    Mem[59] = 59   Mem[91] = 91   Mem[123] = 123
Mem[28] = 28    Mem[60] = 60   Mem[92] = 92   Mem[124] = 124
Mem[29] = 29    Mem[61] = 61   Mem[93] = 93   Mem[125] = 125
Mem[30] = 30    Mem[62] = 62   Mem[94] = 94   Mem[126] = 126
Mem[31] = 31    Mem[63] = 63   Mem[95] = 95   Mem[127] = 127
```

7.8 For Load

These three instruction are used to verify load instruction,

Load byte signed (lb)

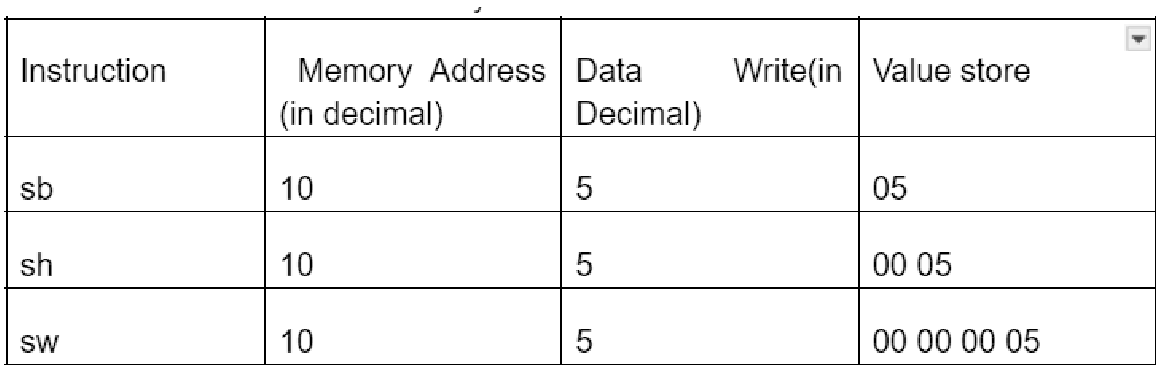
Load half word signed (lh)
Load word signed (lw) All three instruction should load value from memory address “10” and these outputs obtained.



Instruction	Memory Address (in decimal)	Loaded output (in hexadecimal)
lb	10	00 00 00 0a
lh	10	00 00 0b 0a
lw	10	0d 0c 0b 0a

7.9 For Store

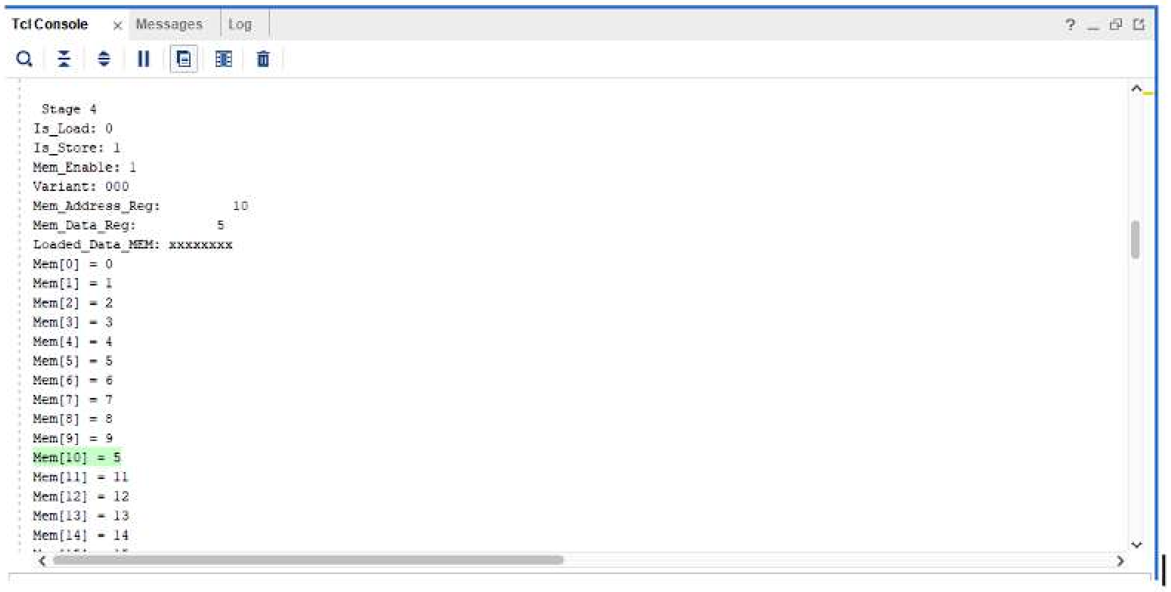
These three instruction are used to verify store instruction:



Instruction	Memory Address (in decimal)	Data Write(in Decimal)	Value store
sb	10	5	05
sh	10	5	00 05
sw	10	5	00 00 00 05

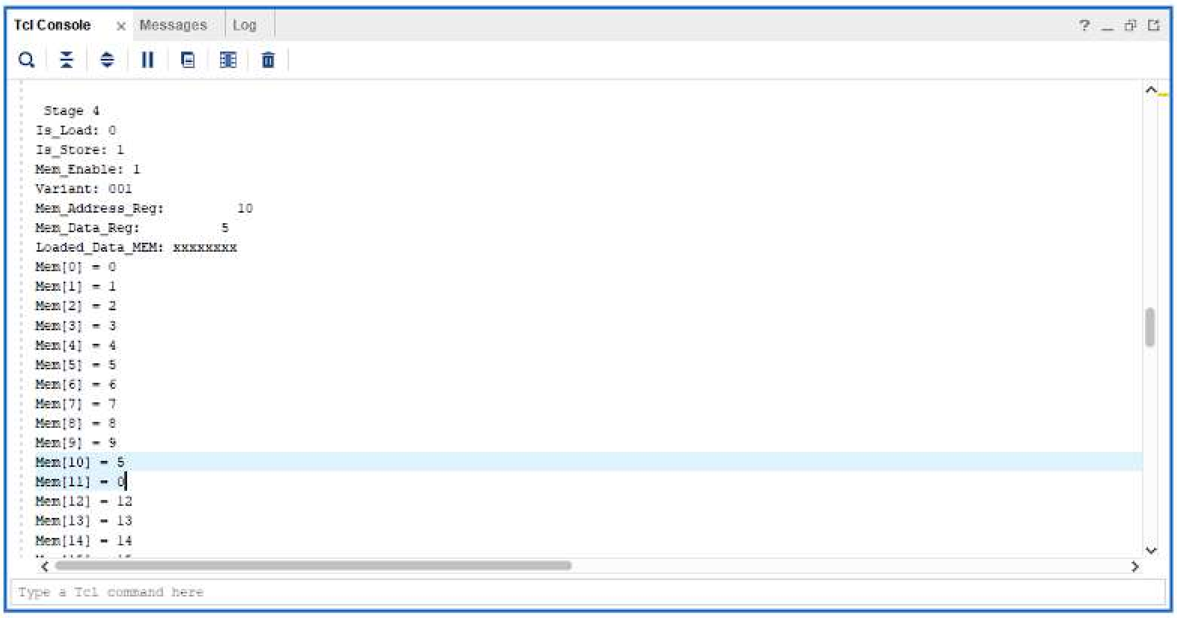
7.10 To Console Output

7.10.1 sb



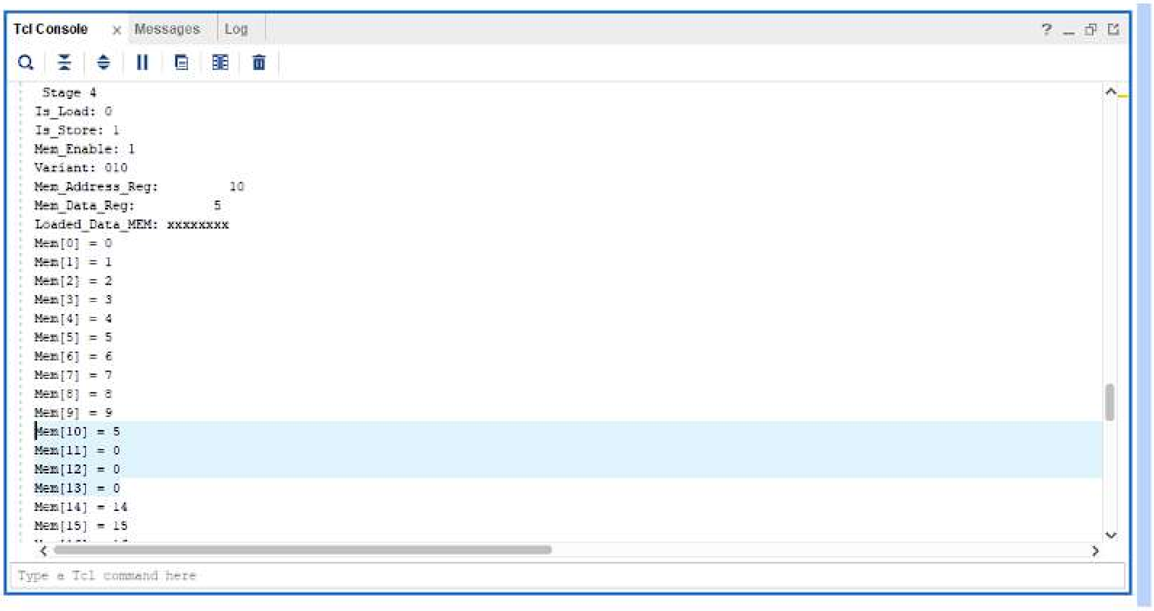
```
 Stage 4
Is_Load: 0
Is_Store: 1
Mem_Enable: 1
Variant: 000
Mem_Address_Reg:          10
Mem_Data_Reg:          5
Loaded_Data_MEM: xxxxxxxx
Mem[0] = 0
Mem[1] = 1
Mem[2] = 2
Mem[3] = 3
Mem[4] = 4
Mem[5] = 5
Mem[6] = 6
Mem[7] = 7
Mem[8] = 8
Mem[9] = 9
Mem[10] = 5
Mem[11] = 11
Mem[12] = 12
Mem[13] = 13
Mem[14] = 14
```

7.10.2 sh



```
 Stage 4
Is_Load: 0
Is_Store: 1
Mem_Enable: 1
Variant: 001
Mem_Address_Reg:          10
Mem_Data_Reg:          5
Loaded_Data_MEM: xxxxxxxx
Mem[0] = 0
Mem[1] = 1
Mem[2] = 2
Mem[3] = 3
Mem[4] = 4
Mem[5] = 5
Mem[6] = 6
Mem[7] = 7
Mem[8] = 8
Mem[9] = 9
Mem[10] = 5
Mem[11] = 0
Mem[12] = 12
Mem[13] = 13
Mem[14] = 14
```

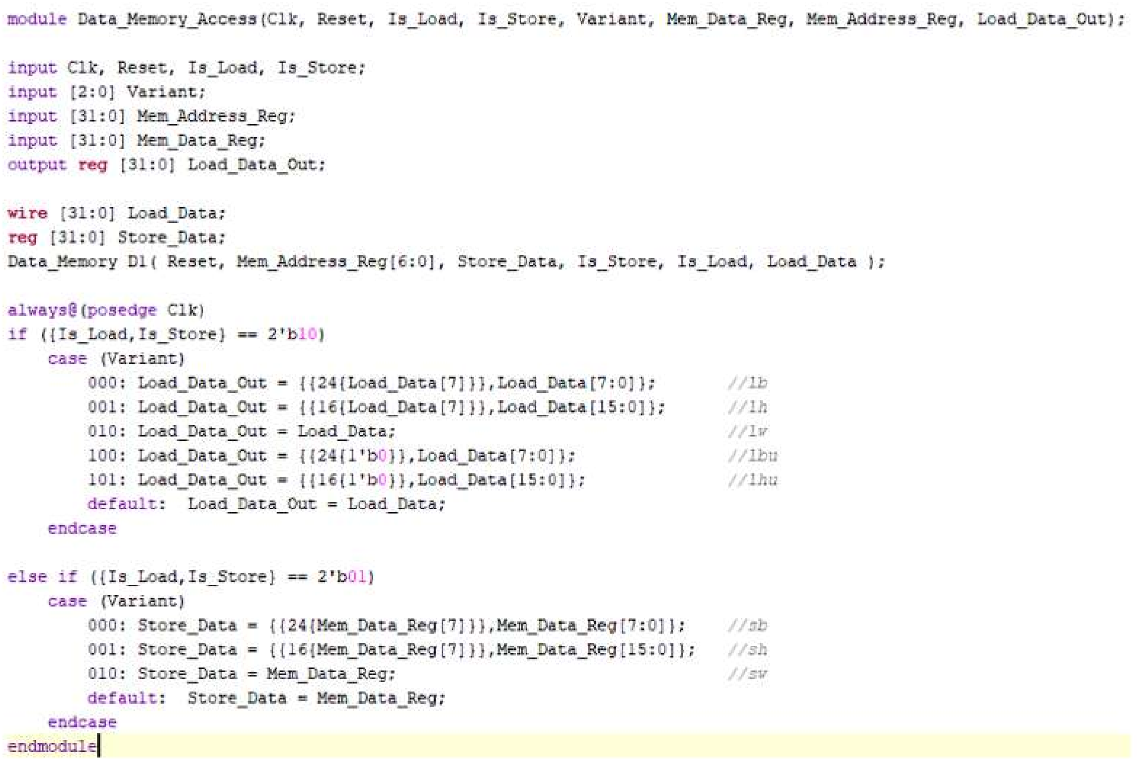
7.10.3 sw



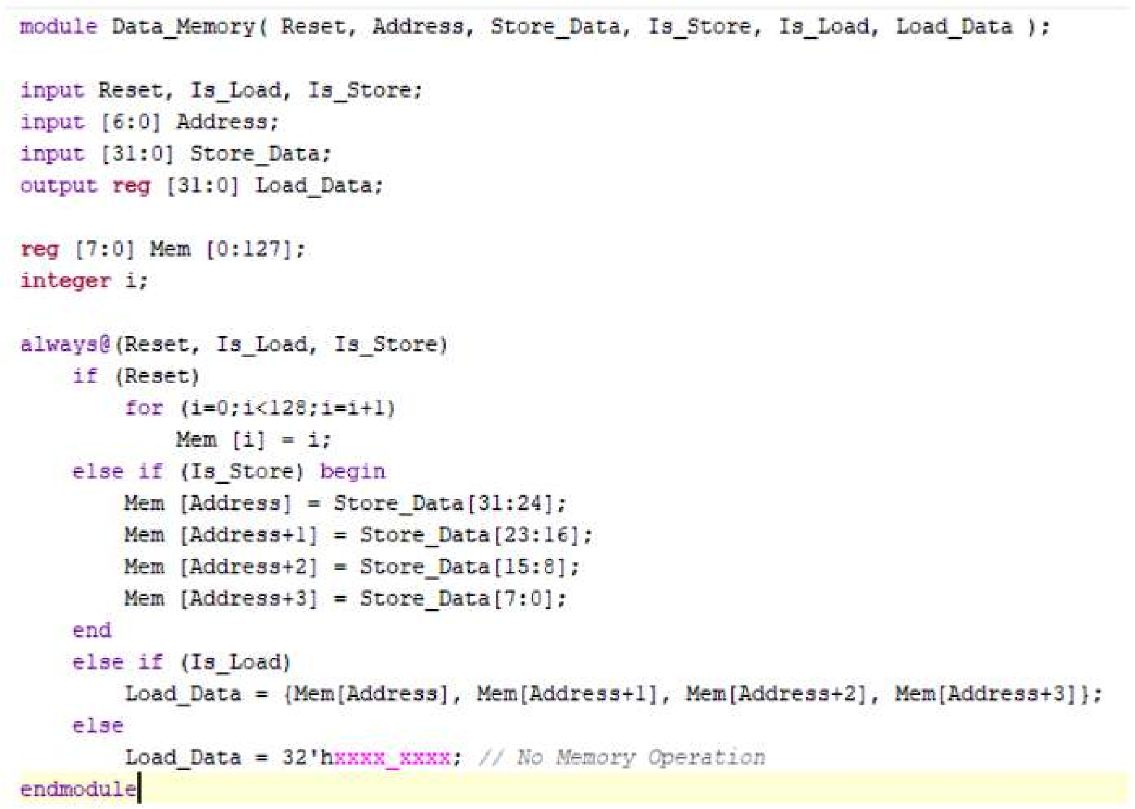
```
 Stage 4
Is_Load: 0
Is_Store: 1
Mem_Enable: 1
Variant: 010
Mem_Address_Reg:         10
Mem_Data_Reg:          5
Loaded_Data_MEM: xxxxxxxx
Mem[0] = 0
Mem[1] = 1
Mem[2] = 2
Mem[3] = 3
Mem[4] = 4
Mem[5] = 5
Mem[6] = 6
Mem[7] = 7
Mem[8] = 8
Mem[9] = 9
Mem[10] = 5
Mem[11] = 0
Mem[12] = 0
Mem[13] = 0
Mem[14] = 14
Mem[15] = 15
```

7.11 Verilog Code

Stage 4 consists of “Data Memory Access” and “Data Memory”.
Data Memory: This is the dual port, main memory. Memory size of 128 bytes. It is initialized as their index number as value. It has control signals as Is-Store and Is-Load. Both reading and writing are synchronous. This memory is synthesizable and can be implemented in the Xilinx Zynq 7000 family.
Data Memory Access: This module is mainly used for masking of word data to half word or byte, based on the instruction. It takes a variant (func3) as input to mask the data.

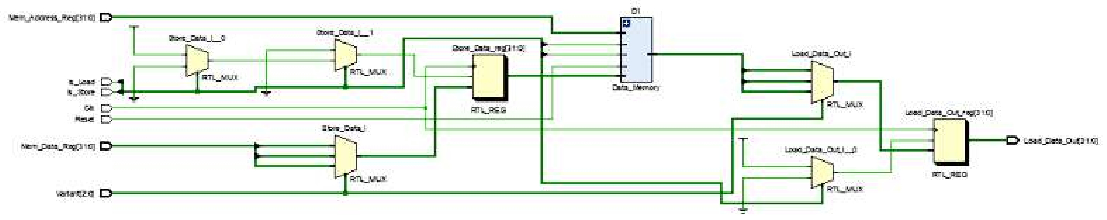


```
module Data_Memory_Access(Clk, Reset, Is_Load, Is_Store, Variant, Mem_Data_Reg, Mem_Address_Reg, Load_Data_Out);

input Clk, Reset, Is_Load, Is_Store;
input [2:0] Variant;
input [31:0] Mem_Address_Reg;
input [31:0] Mem_Data_Reg;
output reg [31:0] Load_Data_Out;

wire [31:0] Load_Data;
reg [31:0] Store_Data;
Data_Memory D1( Reset, Mem_Address_Reg[6:0], Store_Data, Is_Store, Is_Load, Load_Data );

always@(posedge Clk)
if ({Is_Load,Is_Store} == 2'b10)
    case (Variant)
        000: Load_Data_Out = {{24{Load_Data[7]}},Load_Data[7:0]};        //lb
        001: Load_Data_Out = {{16{Load_Data[7]}},Load_Data[15:0]};       //lh
        010: Load_Data_Out = Load_Data;                                  //lw
        100: Load_Data_Out = {{24{1'b0}},Load_Data[7:0]};                //lbu
        101: Load_Data_Out = {{16{1'b0}},Load_Data[15:0]};               //lhu
        default:  Load_Data_Out = Load_Data;
    endcase

else if ({Is_Load,Is_Store} == 2'b01)
    case (Variant)
        000: Store_Data = {{24{Mem_Data_Reg[7]}},Mem_Data_Reg[7:0]};     //sb
        001: Store_Data = {{16{Mem_Data_Reg[7]}},Mem_Data_Reg[15:0]};    //sh
        010: Store_Data = Mem_Data_Reg;                                  //sw
        default:  Store_Data = Mem_Data_Reg;
    endcase
endmodule
```

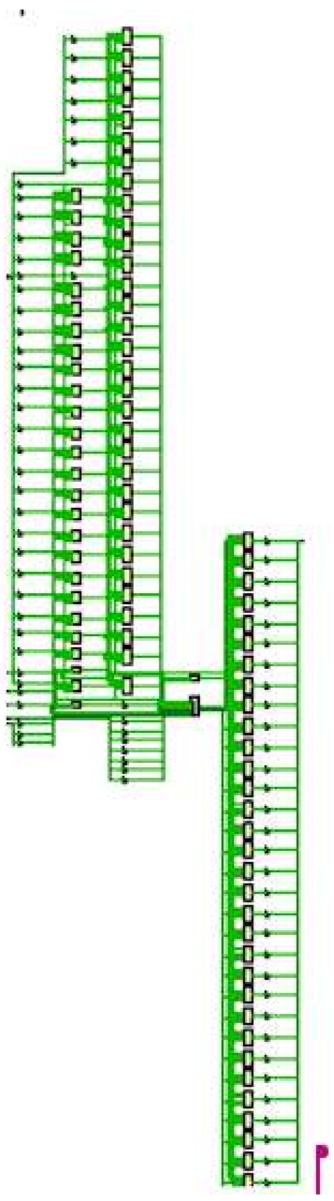


```
module Data_Memory( Reset, Address, Store_Data, Is_Store, Is_Load, Load_Data );

input Reset, Is_Load, Is_Store;
input [6:0] Address;
input [31:0] Store_Data;
output reg [31:0] Load_Data;

reg [7:0] Mem [0:127];
integer i;

always@(Reset, Is_Load, Is_Store)
    if (Reset)
        for (i=0;i<128;i=i+1)
            Mem [i] = i;
    else if (Is_Store) begin
        Mem [Address] = Store_Data[31:24];
        Mem [Address+1] = Store_Data[23:16];
        Mem [Address+2] = Store_Data[15:8];
        Mem [Address+3] = Store_Data[7:0];
    end
    else if (Is_Load)
        Load_Data = {Mem[Address], Mem[Address+1], Mem[Address+2], Mem[Address+3]};
    else
        Load_Data = 32'hxxxx_xxxx; // No Memory Operation
endmodule
```

7.12 Schematic

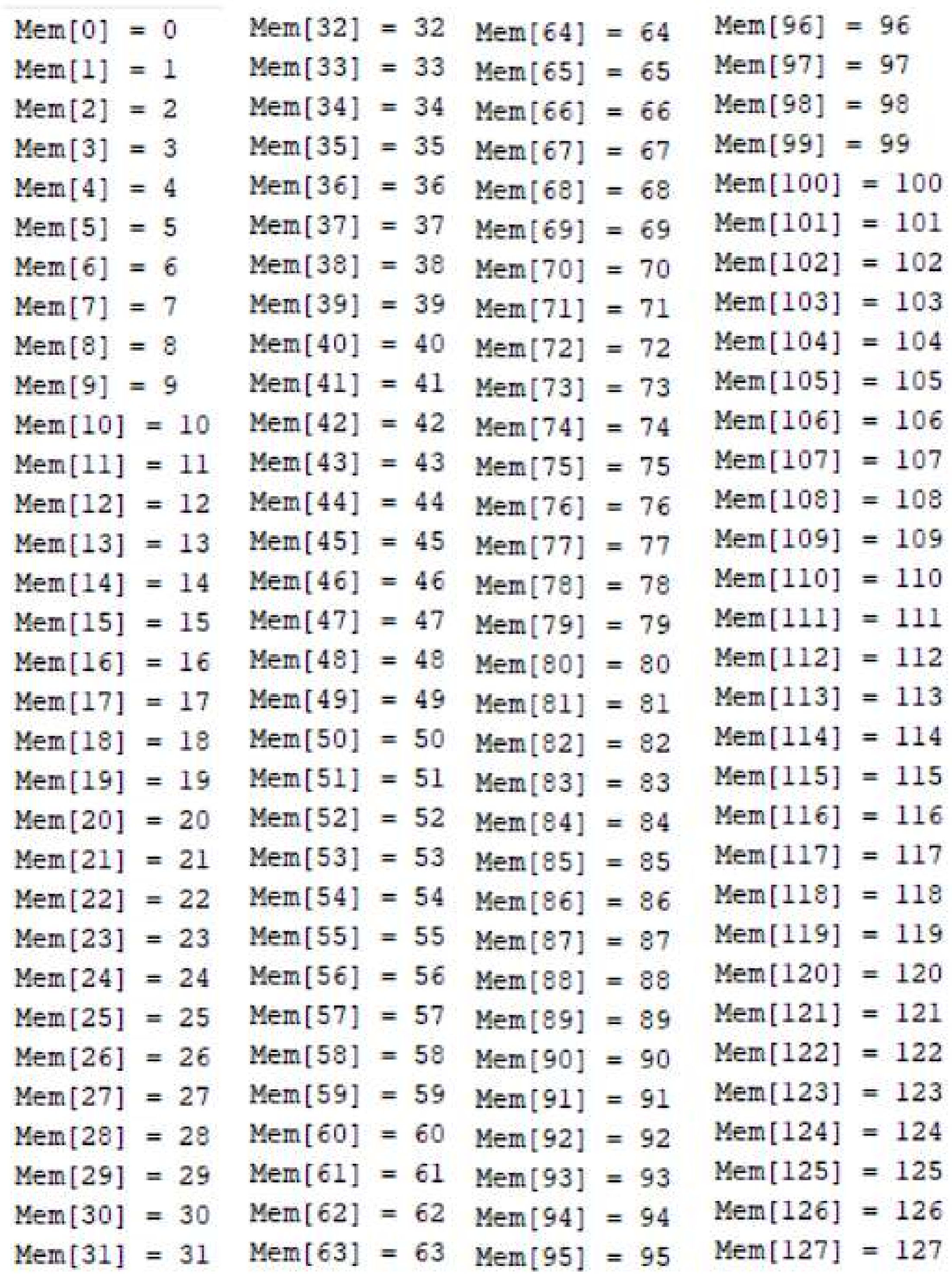


7.13 Implementation



7.14 Results and Discussion

We have designed a 128-byte data memory. This memory is initialized as their index number as value. It has control signals such as Memory enable, variant, Is-Store and Is-Load. Reading is synchronous and writing is also synchronous with the clock.



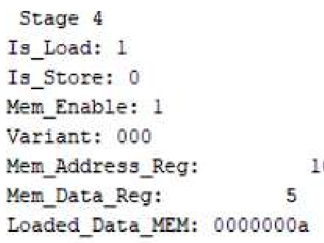
```
Mem[0] = 0      Mem[32] = 32    Mem[64] = 64    Mem[96] = 96
Mem[1] = 1      Mem[33] = 33    Mem[65] = 65    Mem[97] = 97
Mem[2] = 2      Mem[34] = 34    Mem[66] = 66    Mem[98] = 98
Mem[3] = 3      Mem[35] = 35    Mem[67] = 67    Mem[99] = 99
Mem[4] = 4      Mem[36] = 36    Mem[68] = 68    Mem[100] = 100
Mem[5] = 5      Mem[37] = 37    Mem[69] = 69    Mem[101] = 101
Mem[6] = 6      Mem[38] = 38    Mem[70] = 70    Mem[102] = 102
Mem[7] = 7      Mem[39] = 39    Mem[71] = 71    Mem[103] = 103
Mem[8] = 8      Mem[40] = 40    Mem[72] = 72    Mem[104] = 104
Mem[9] = 9      Mem[41] = 41    Mem[73] = 73    Mem[105] = 105
Mem[10] = 10    Mem[42] = 42    Mem[74] = 74    Mem[106] = 106
Mem[11] = 11    Mem[43] = 43    Mem[75] = 75    Mem[107] = 107
Mem[12] = 12    Mem[44] = 44    Mem[76] = 76    Mem[108] = 108
Mem[13] = 13    Mem[45] = 45    Mem[77] = 77    Mem[109] = 109
Mem[14] = 14    Mem[46] = 46    Mem[78] = 78    Mem[110] = 110
Mem[15] = 15    Mem[47] = 47    Mem[79] = 79    Mem[111] = 111
Mem[16] = 16    Mem[48] = 48    Mem[80] = 80    Mem[112] = 112
Mem[17] = 17    Mem[49] = 49    Mem[81] = 81    Mem[113] = 113
Mem[18] = 18    Mem[50] = 50    Mem[82] = 82    Mem[114] = 114
Mem[19] = 19    Mem[51] = 51    Mem[83] = 83    Mem[115] = 115
Mem[20] = 20    Mem[52] = 52    Mem[84] = 84    Mem[116] = 116
Mem[21] = 21    Mem[53] = 53    Mem[85] = 85    Mem[117] = 117
Mem[22] = 22    Mem[54] = 54    Mem[86] = 86    Mem[118] = 118
Mem[23] = 23    Mem[55] = 55    Mem[87] = 87    Mem[119] = 119
Mem[24] = 24    Mem[56] = 56    Mem[88] = 88    Mem[120] = 120
Mem[25] = 25    Mem[57] = 57    Mem[89] = 89    Mem[121] = 121
Mem[26] = 26    Mem[58] = 58    Mem[90] = 90    Mem[122] = 122
Mem[27] = 27    Mem[59] = 59    Mem[91] = 91    Mem[123] = 123
Mem[28] = 28    Mem[60] = 60    Mem[92] = 92    Mem[124] = 124
Mem[29] = 29    Mem[61] = 61    Mem[93] = 93    Mem[125] = 125
Mem[30] = 30    Mem[62] = 62    Mem[94] = 94    Mem[126] = 126
Mem[31] = 31    Mem[63] = 63    Mem[95] = 95    Mem[127] = 127
```

7.15 For Load

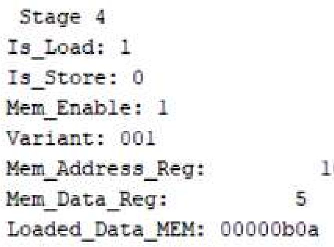
These three instruction are used to verify load instruction,
Load byte signed (lb) Load half word signed (lh) Load word signed (lw)
All three instruction should load value from memory address “10” and these outputs obtained.

Instruction	Memory Address (in decimal)	Loaded output (in hexadecimal)
lb	10	00 00 00 0a
lh	10	00 00 0b 0a
lw	10	0d 0c 0b 0a

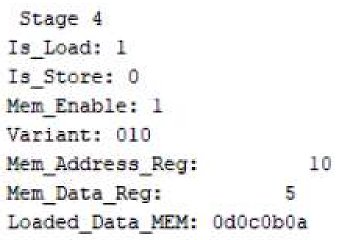
7.16 Output of Command Window



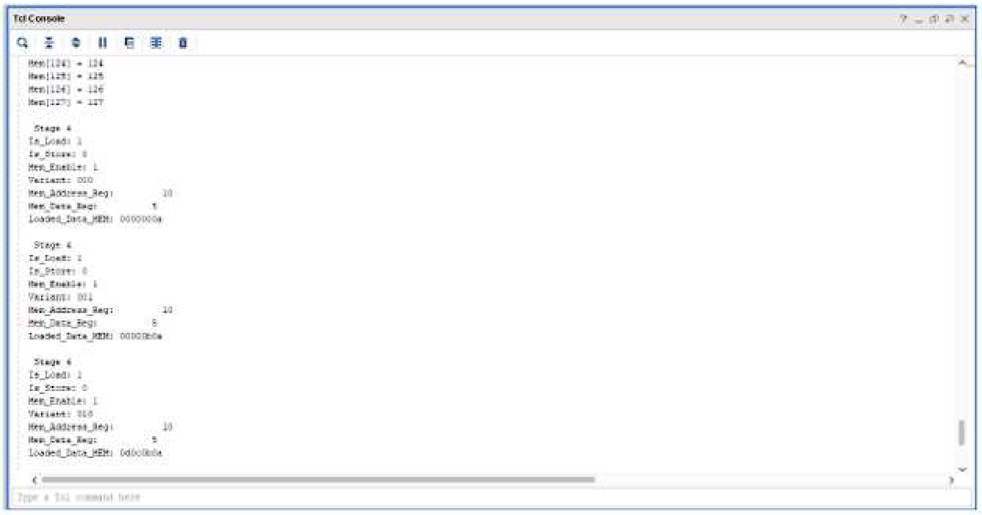
```
 Stage 4
Is_Load: 1
Is_Store: 0
Mem_Enable: 1
Variant: 000
Mem_Address_Reg:          10
Mem_Data_Reg:          5
Loaded_Data_MEM: 0000000a
```



```
 Stage 4
Is_Load: 1
Is_Store: 0
Mem_Enable: 1
Variant: 001
Mem_Address_Reg:          10
Mem_Data_Reg:          5
Loaded_Data_MEM: 00000b0a
```



```
 Stage 4
Is_Load: 1
Is_Store: 0
Mem_Enable: 1
Variant: 010
Mem_Address_Reg:          10
Mem_Data_Reg:          5
Loaded_Data_MEM: 0d0c0b0a
```



```
Mem[124] = 124
Mem[125] = 125
Mem[126] = 126
Mem[127] = 127

 Stage 4
Is_Load: 1
Is_Store: 0
Mem_Enable: 1
Variant: 000
Mem_Address_Reg:          10
Mem_Data_Reg:          5
Loaded_Data_MEM: 0000000a

 Stage 4
Is_Load: 1
Is_Store: 0
Mem_Enable: 1
Variant: 001
Mem_Address_Reg:          10
Mem_Data_Reg:          5
Loaded_Data_MEM: 00000b0a

 Stage 4
Is_Load: 1
Is_Store: 0
Mem_Enable: 1
Variant: 010
Mem_Address_Reg:          10
Mem_Data_Reg:          5
Loaded_Data_MEM: 0d0c0b0a
```

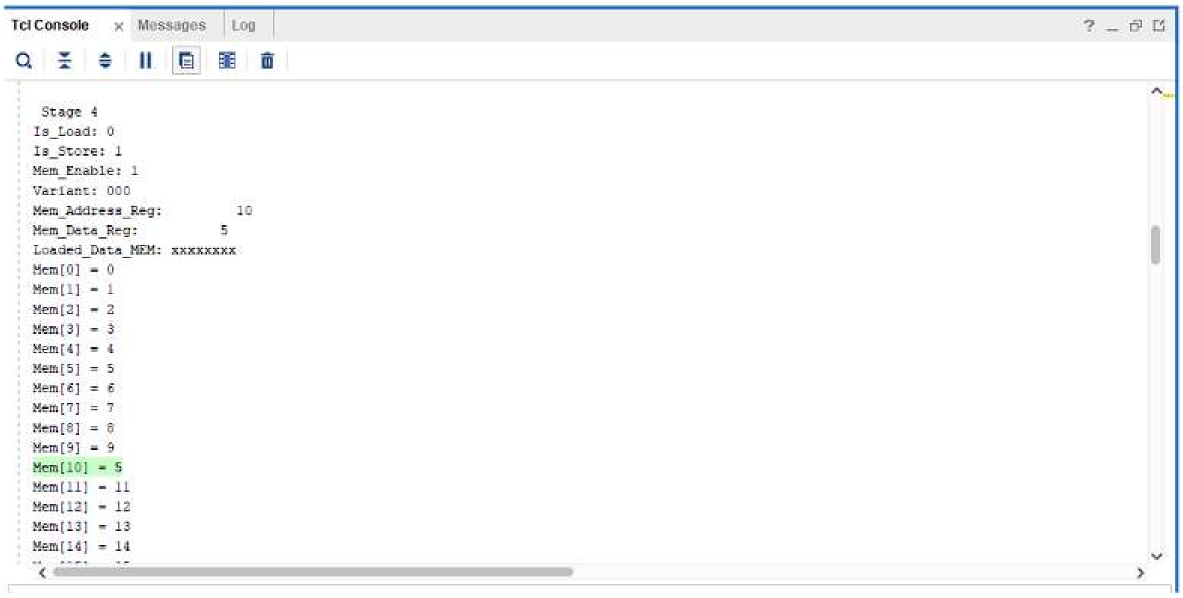
7.17 For store

These three instruction are used to verify store instruction:

Instruction	Memory Address (in decimal)	Data Write(in Decimal)	Value store
sb	10	5	05
sh	10	5	00 05
sw	10	5	00 00 00 05

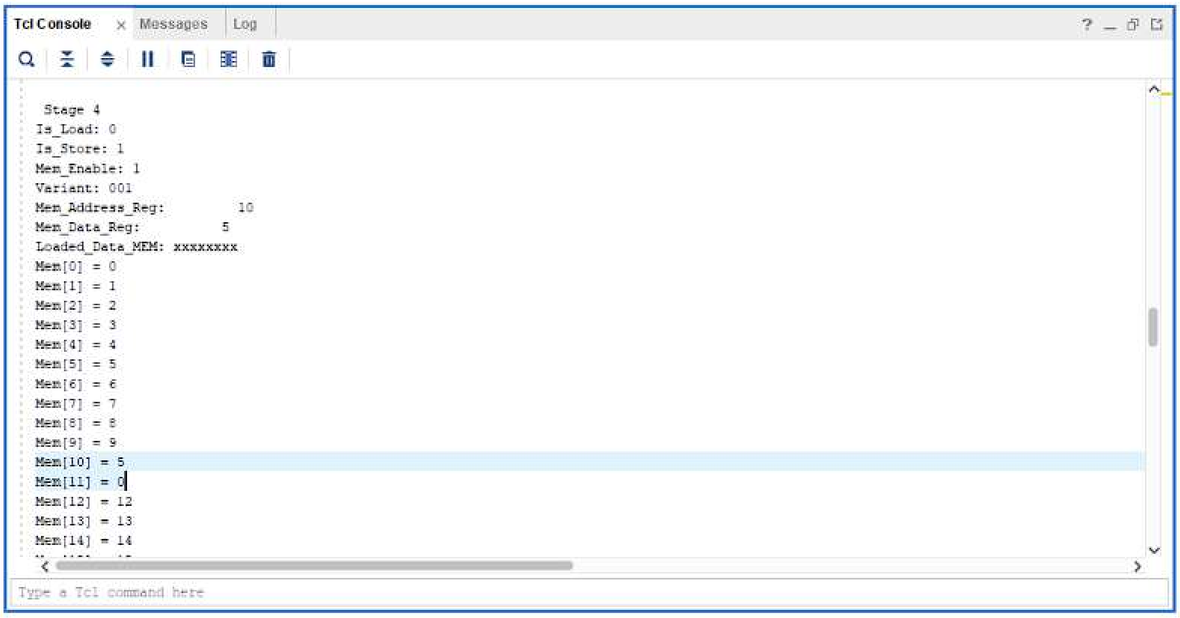
7.18 TCL console output

7.18.1 sb



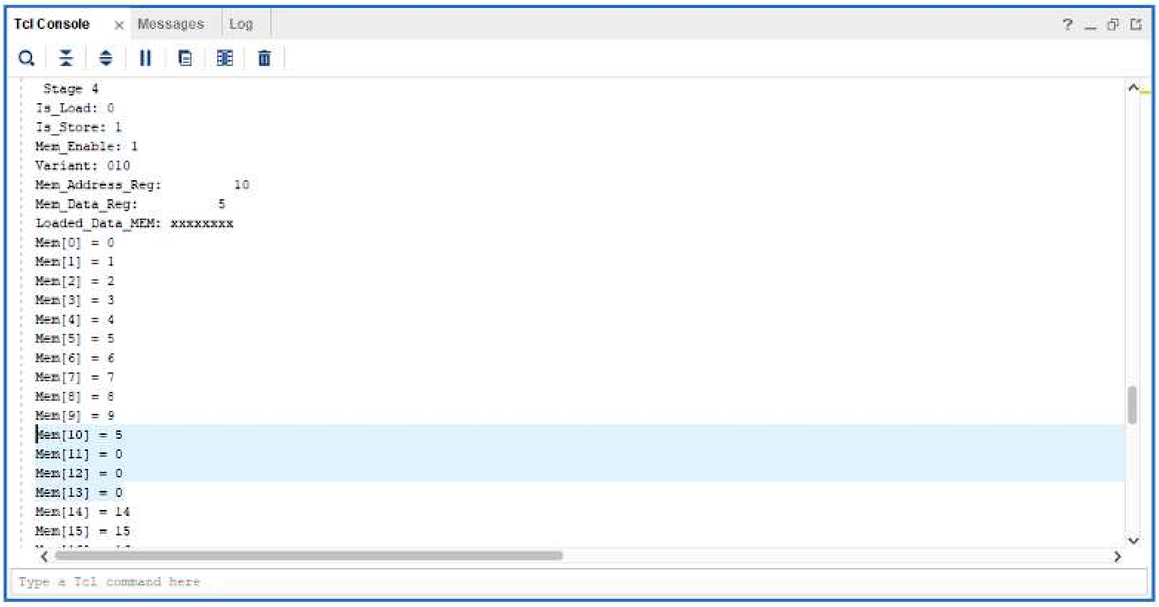
```
 Stage 4
Is_Load: 0
Is_Store: 1
Mem_Enable: 1
Variant: 000
Mem_Address_Reg:          10
Mem_Data_Reg:          5
Loaded_Data_MEM: xxxxxxxx
Mem[0] = 0
Mem[1] = 1
Mem[2] = 2
Mem[3] = 3
Mem[4] = 4
Mem[5] = 5
Mem[6] = 6
Mem[7] = 7
Mem[8] = 8
Mem[9] = 9
Mem[10] = 5
Mem[11] = 11
Mem[12] = 12
Mem[13] = 13
Mem[14] = 14
```

7.18.2 sh

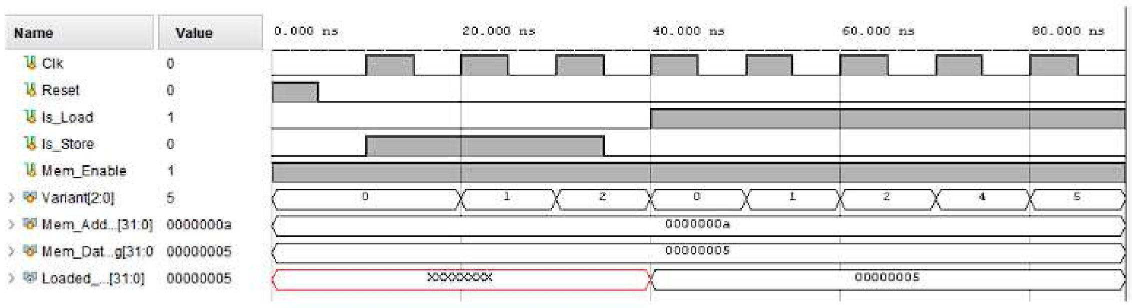


```
 Stage 4
Is_Load: 0
Is_Store: 1
Mem_Enable: 1
Variant: 001
Mem_Address_Reg:         10
Mem_Data_Reg:          5
Loaded_Data_MEM: xxxxxxxx
Mem[0]  = 0
Mem[1]  = 1
Mem[2]  = 2
Mem[3]  = 3
Mem[4]  = 4
Mem[5]  = 5
Mem[6]  = 6
Mem[7]  = 7
Mem[8]  = 8
Mem[9]  = 9
Mem[10] = 5
Mem[11] = 0
Mem[12] = 12
Mem[13] = 13
Mem[14] = 14
```

7.18.3 sw



```
 Stage 4
Is_Load: 0
Is_Store: 1
Mem_Enable: 1
Variant: 010
Mem_Address_Reg:         10
Mem_Data_Reg:          5
Loaded_Data_MEM: xxxxxxxx
Mem[0]  = 0
Mem[1]  = 1
Mem[2]  = 2
Mem[3]  = 3
Mem[4]  = 4
Mem[5]  = 5
Mem[6]  = 6
Mem[7]  = 7
Mem[8]  = 8
Mem[9]  = 9
Mem[10] = 5
Mem[11] = 0
Mem[12] = 0
Mem[13] = 0
Mem[14] = 14
Mem[15] = 15
```

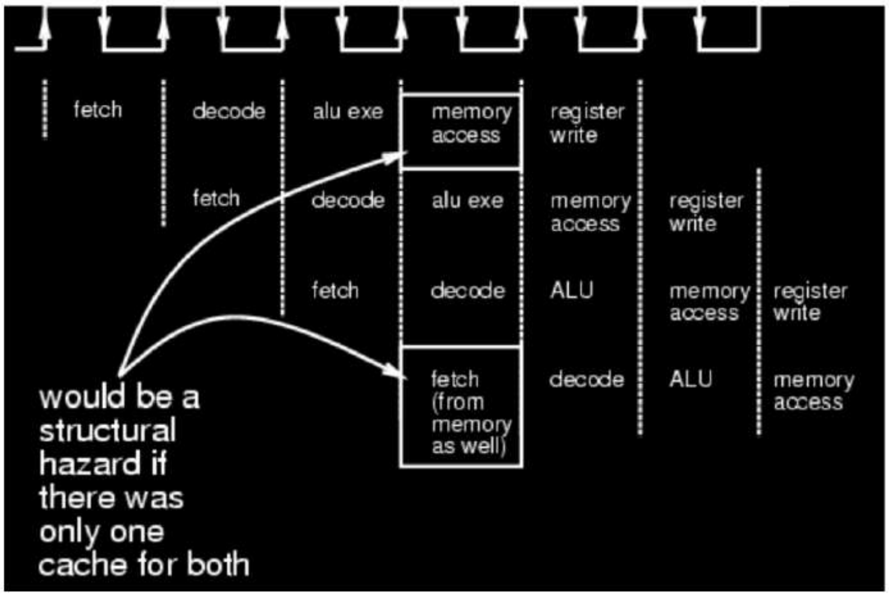

Name
Value
0.000 ns
20.000 ns
40.000 ns
60.000 ns
80.000 ns
Clk 0
Reset 0
Is_Load 1
Is_Store 0
Mem_Enable 1
Variant[2:0] 5
0 1 2 0 1 2 4 5
Mem_Add...[31:0] 0000000a
0000000a
Mem_Dat...g[31:0 00000005
00000005
Loaded_...[31:0] 00000005
XXXXXXXX
00000005

Chapter 8

Pipeline Hazards and Forwarding unit

8.1 I

n the above image you will see that in the same clock cycle two different instructions are trying to access the memory, one from the data cache and the other from the instruction cache. Had it been the case where both data cache and instruction cache were a single cache module, then a structural hazard would have occurred since two different instructions are trying to access the same memory module. You may encounter such a structural hazard when your processor is following Von-nuemann architecture, where there is only one set of data and an addresss bus. Since Harvard architecture provides different sets of data and address buses for data and instruction caches, such structural hazards will not occur. In our five stage pipelined design we have followed Harvard architecture where there is two different caches, one for instruction and other for data to avoid such hazards.



Structural hazards occur when different instructions require the same hardware at the same clock cycle. In RISC-V RV32I ISA certain instructions require address calculation as well data calculation. To keep the execution of data and address calculation in the same stage we have divided execution stage into two different ALUs, one for data execution and other for address calculation. Instructions like JAL_R and branch instructions require data and address calculation at the same time therefore separating the ALUs for the different functions help mitigate any kind of structural hazard.

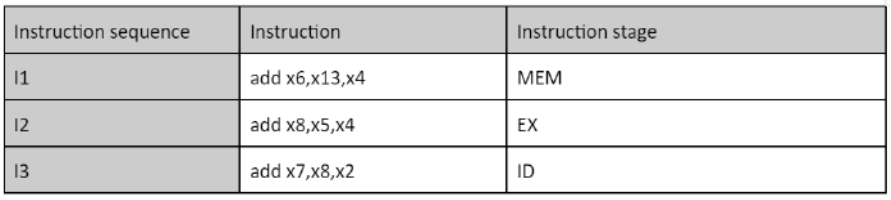
Another such hazard is when you try to perform read and write operation on the same register from the register file. Register file can be accessed in both decode stage as well writeback stage. The register file will get accessed in decode stage for read operation and in writeback stage the same register gets accessed for write operation. Such a hazard can be prevented by having seperate read and write ports. Another way is to design the register file such that it writes to the register bank in the positive half of the clock cycle and reads in the negative half of the clock cycle.

8.2 Data Hazards

Data hazards occur when there is a data dependency between instructions, such that an instruction is trying to request data assuming that data is ready, but in reality, that data might not have been written back to the register file or the data cache by the previous instruction. The data might still be in the intermediary stage, and to get the correct data for the most recent instruction, we would need to forward the data from the intermediary stage to the execution stage of the current instruction. There can be several cases of data hazards and some corner cases depending on the type of implementation you decide to go for your design. Data hazards are usually mitigated using forwarding, inserting NOPs (no operation) or stalling the pipeline.

8.3 Cases of dependencies

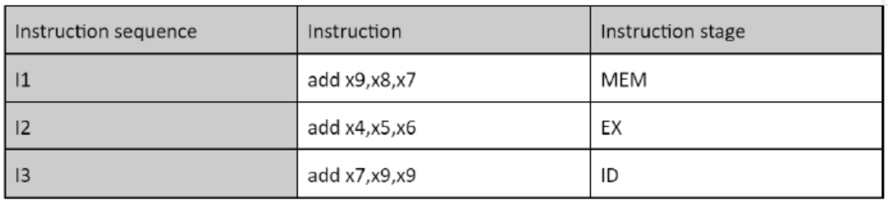
8.3.1 Case1



Instruction sequence	Instruction	Instruction stage
I1	add x6,x13,x4	MEM
I2	add x8,x5,x4	EX
I3	add x7,x8,x2	ID

In the above example there is a data dependency of r8 between I2 and I3. When I3 goes to execution stage I2 will go to memory stage and the r8 would not have been written back yet to the register, therefore the wrong value will be read by I3 while in execution stage. This is the case where there is a data dependency between I2 and I3.

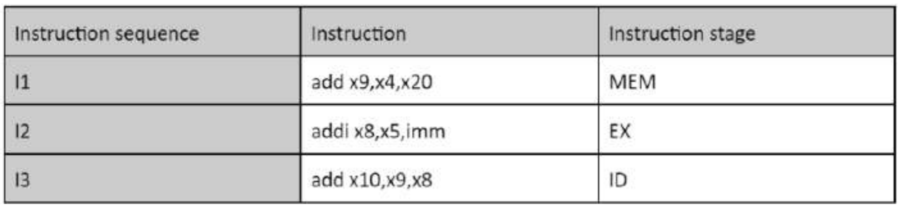
8.3.2 Case2



Instruction sequence	Instruction	Instruction stage
I1	add x9,x8,x7	MEM
I2	add x4,x5,x6	EX
I3	add x7,x9,x9	ID

In the above case the when I3 goes to execution stage after decode stage I1 will go to writeback stage and I2 will go memory stage. There is data dependency b/w I1 and I3 of r9. Value of r9 is required for both the operands of I3. Now because I1 will currently go to writeback stage the value has still not been updated in the register file and r9 value is still not ready. Therefore the wrong value will read by I3 and a data hazard will occur. This is the case where there is data dependency b/w I1 and I3.

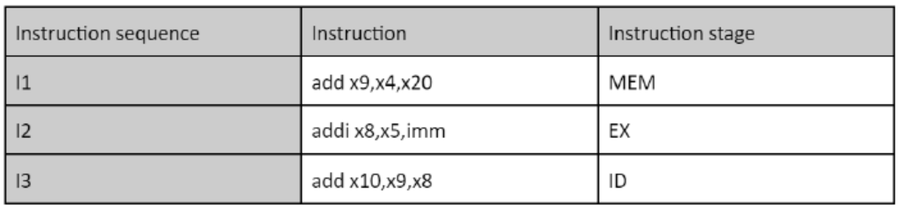
8.3.3 Case3



Instruction sequence	Instruction	Instruction stage
I1	add x9,x4,x20	MEM
I2	addi x8,x5,imm	EX
I3	add x10,x9,x8	ID

In the above case when I3 goes to exec stage I1 will go to Wb stage and I2 will go to Mem stage i.e both the values of r9 and r8 will not be ready for I3 to operate while it is in execution stage and therefore a double data dependency will occur between I1 and I3 as well I2 and I3.

8.3.4 Case4

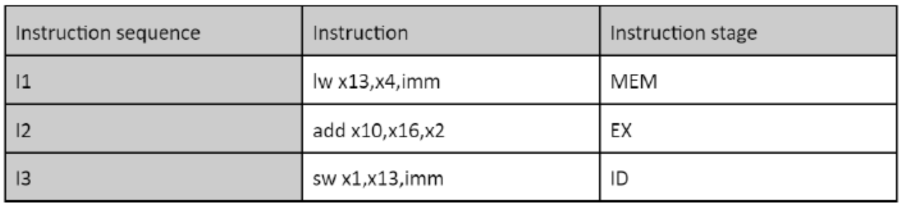


Instruction sequence	Instruction	Instruction stage
I1	add x9,x4,x20	MEM
I2	addi x8,x5,imm	EX
I3	add x10,x9,x8	ID

In the above case store instruction introduces a special kind of data dependency in our own implementation of the processor. From the operand select table the operands for store instruction in ACU are rs1 and imm and the operands for DEU were rs2 and zero. To avoid double forwarding in the case store we have introduced extra conditions to be checked before forwarding the correct data. This will be discussed in detail later in pipeline management chapter. In the case of load and JALR ,similar forwarding conditions are required but store cover almost all conditions and therefore it becomes a corner case to be checked in our design.

In the case above only there is data dependency between rs1 (x13) and from I3 and rd from I1.

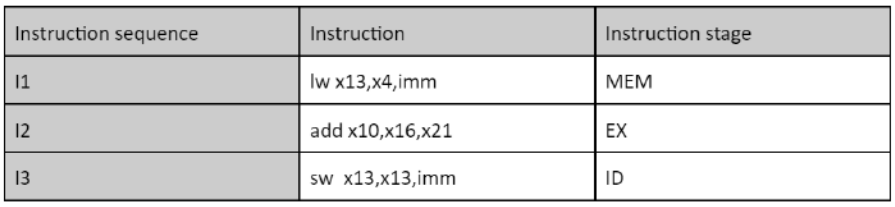
8.3.5 Case5



Instruction sequence	Instruction	Instruction stage
I1	lw x13,x4,imm	MEM
I2	add x10,x16,x2	EX
I3	sw x1,x13,imm	ID

In this case there is a data dependency between I3 and I1 as rs1 needs to be forwarded from I1 (rd).

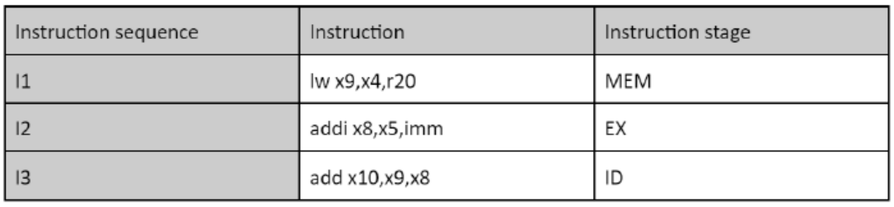
8.3.6 Case6



Instruction sequence	Instruction	Instruction stage
I1	lw x13,x4,imm	MEM
I2	add x10,x16,x21	EX
I3	sw x13,x13,imm	ID

In case 6 both the operands rs1 and rs2 (here x13) need to be forwarded from I1.

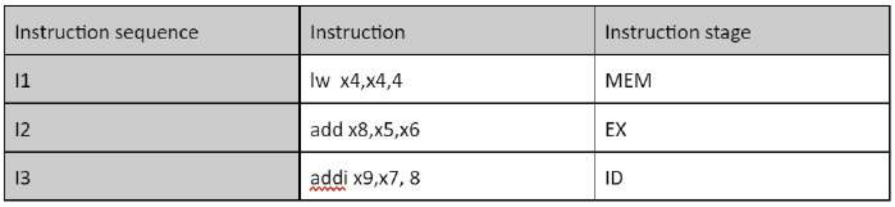
8.3.7 Case7



Instruction sequence	Instruction	Instruction stage
I1	lw x9,x4,r20	MEM
I2	addi x8,x5,imm	EX
I3	add x10,x9,x8	ID

In the case above when I3 goes to execution stage I1 will go to WB stage and I2 will go to MEM stage. Clearly there is data dependence between I2 and I3 as well a data dependence between I1 and I3. As we will see in the pipeline management chapter, there is not going to be any need to insert a stall after load in this case.

8.3.8 Case8



Instruction sequence	Instruction	Instruction stage
I1	lw x4,x4,4	MEM
I2	add x8,x5,x6	EX
I3	addi x9,x7, 8	ID

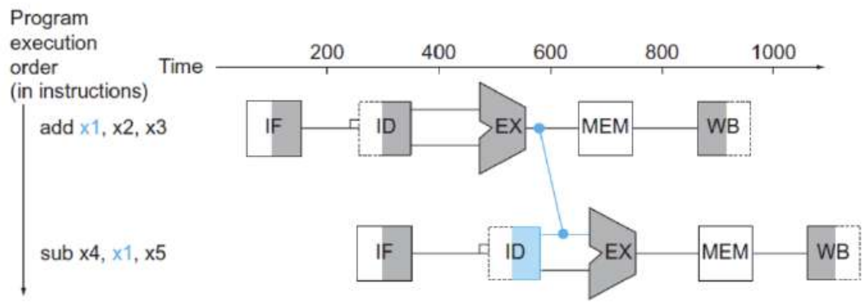
In the above case there should not be a data dependency between I2 and I3 as the value 8 should not be considered for forwarding. To do this we must introduce rs1_valid and rs2_valid, which tells us which instructions have the rs1 or rs2 fields. For example, addi r9,r7,8 doesn't have a valid rs2 field since that field is reserved for immediates in addi instruction.

8.4 Control Hazards:

Control Hazard arises when a control transfer instruction comes into picture which can be a B-type, JAL or JALR instruction. In our implementation the jump address for these instructions is calculated in the EX stage. For B-type instructions also known as conditional branch, the decision to branch is evaluated in the EX stage. In the mean time IF and ID stage get occupied with the instruction next in line to the B-type instruction. Instructions following the branch are fetched consequently because of the branch not taken policy. We assume that the branch condition will evaluate to false and keep on fetching consequent instructions. If branch condition evaluates to false we don't bother about it and the pipeline works seamlessly. On the other if branch condition is true then the instruction flow requires changing and the instructions following the branch in IF,ID stages are required to be flushed. Flushing here implies that the instructions in these stages should be discarded and the result of their operations should not change the register or the memory contents. A similar situation is encountered for JAL and JALR instructions. Although no condition checking is required for them but due to Jump address calculation in the EX stage they instigate flushing of IF and ID stages. Mitigation of control hazard implies minimum flushing. Minimum flushing can be achieved by either earlier detection of control transfer instructions and address calculation. It can also be taken care of via branch prediction.

8.5 Data Hazards: Forwarding Unit:

In the pipeline execution methodology, we feed an instruction per clock cycle under normal conditions. However,the semantics of instruction execution still remain sequential. There are conditions where feeding an instruction per clock cycle and sequential will not see each other eye to eye. Such situations manifest predominantly in the case of RAW (Read After Write), and to some extent on WAR (Write After Read), and WAW (Write After Write) hazards.



The above instruction sequence represents a data hazard. It is of RAW type. In RISC-V our only concern regarding hazard is for the RAW type. One way of dealing with RAW hazard is to stall the pipeline till the data for consumer instruction (sub) is made available by the producer instruction (add). It would require for sub instruction to be held back in ID stage and inserting NOP or a bubble in the further stages. Stalling severely reduces the performance of the pipeline and hence is avoided as much as possible. Another way of looking at this problem is that the operand value for the consumer instruction is available at the output of EX/MEM register which can be made available as shown below.

This would require additional hardware for detecting and forwarding the required operand value to the consumer instruction. The module that deals operand internal forwarding is called Forwarding Unit. The working of Forwarding unit is influenced heavily by the design of ALU stage in your design. Considering our implementation of ALU which has two units viz ACU (Address Calculation Unit) and DEU (Data Execution Unit) we have come up with guidelines for the implementation of internal forwarding which are stated below.

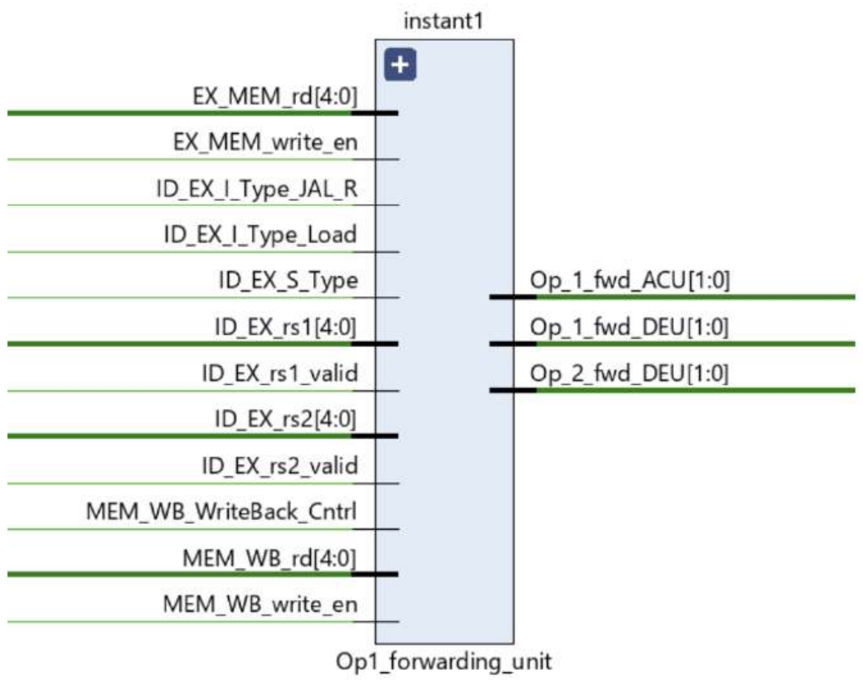
8.6 Guidelines for forwarding

- Internal forwarding of operands is restricted only to EX stage logic.
- Internal forwarding is done from only two pipeline registers: EX/MEM and/or MEM/WB.
- Separate internal forwarding logic blocks are used for:
- Internal forwarding of operand1 (in lieu of value read from rs1) from EX/MEM and/or MEM/WB.
- Internal forwarding of operand2 (in lieu of value read from rs2) from EX/MEM and/or MEM/WB.
- If internal forwarding condition(s) for operand 1 and/or operand 2 are simultaneously satisfied from both the forwarding sources e.g., EX/MEM and MEM/WB, then actual forwarding is done from EX/MEM (to supply latest value of the operand).
- If a consumer instruction happens to be one stage behind the producer instruction that produces the operand value in EX stage, then no stall injection is necessary.
- If a consumer instruction happens to be one stage behind the producer instruction, but the producing instruction produces the operand value not in EX stage but in MEM stage, then a stall is introduced to separate the two instructions by one instruction.
- A dummy add zero operation is performed in EX stage for both the source operands involved in a store instruction or a single source operand involved in a load instruction. Note: This helps in avoiding the need for forwarding from a pipeline stage beyond the WB

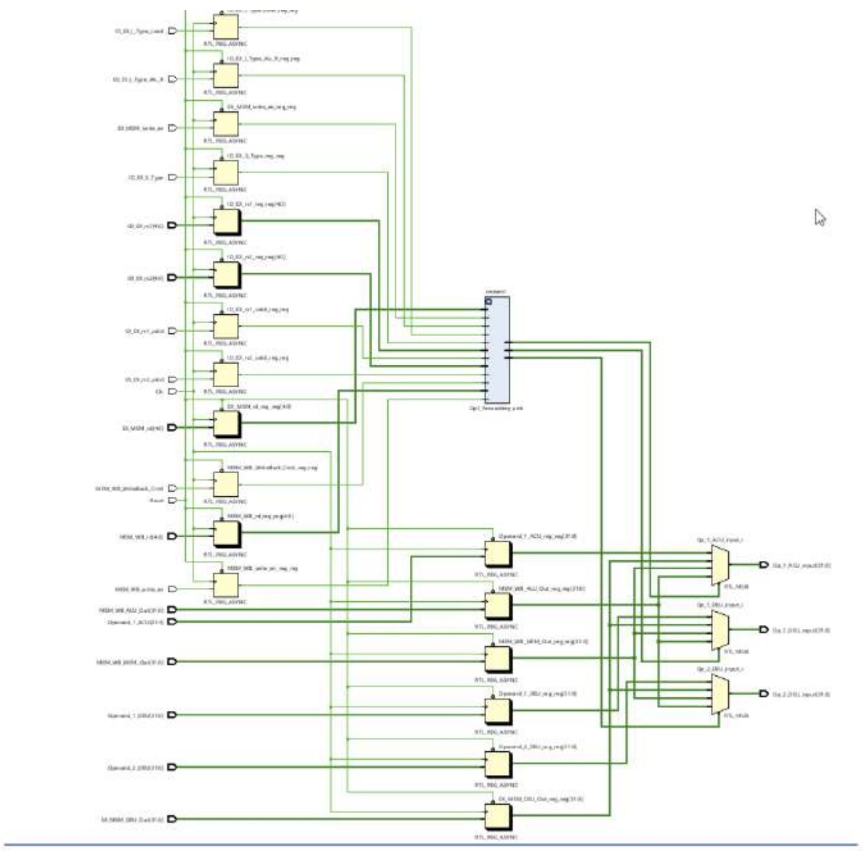
stage, to take care of a store instruction following a load instruction whose destination register is the source register of the store instruction.

8.7 Forwarding unit realization

The data hazard detection for forwarding unit takes place when the consumer instruction is in EX stage as just after the clock edge. One reason for this is that It is the minimum time needed for the producer instruction to produce the required operand output for the consumer instruction. The signals for hazard detection for consumer are taken from ID/EX stage and the producer can be in any of the succeding stages. Hence there signals are taken from EX/MEM and MEM/WB pipelined registers. Figure below shows these signals and corresponding outputs of module goes for controlling the multiplexers for switching inputs to each unit depending on the hazard condition.

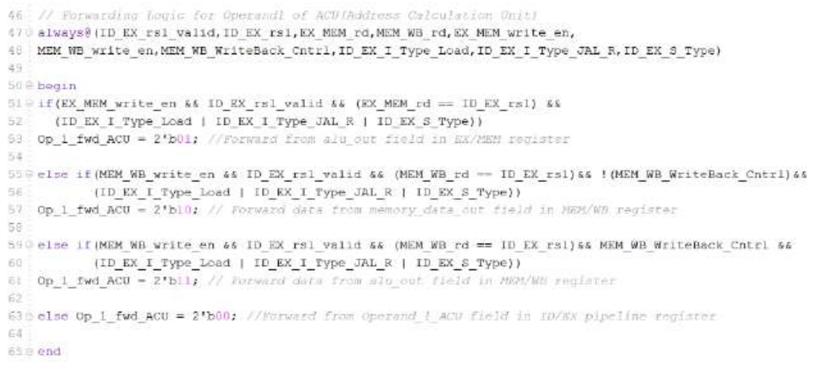


8.8 DUT



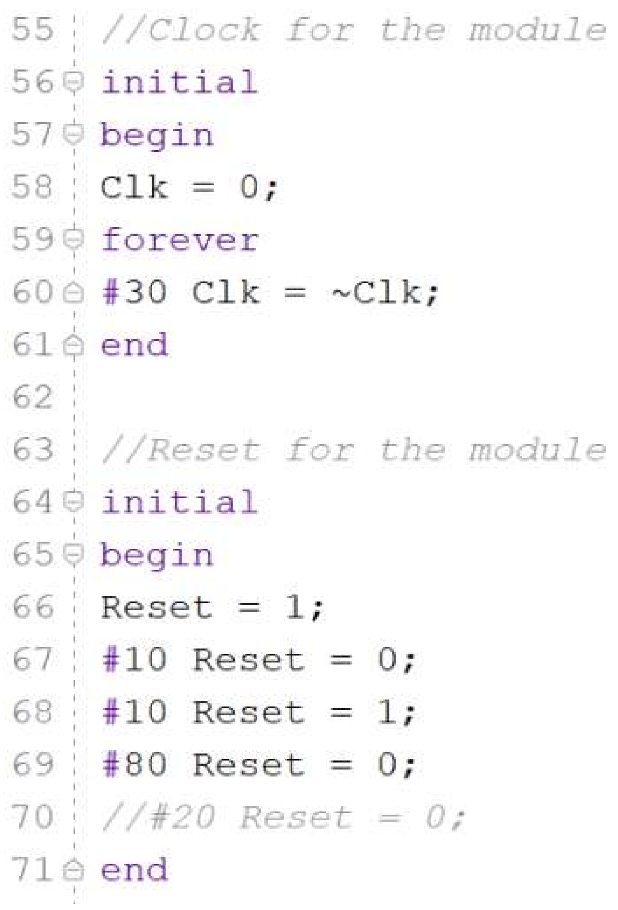
In the figure a DUT is shown for the forwarding unit simulating the conditions in which a forwarding unit works. The inputs to the forwarding unit are given by clocked pipeline registers and its output are control signals to the corresponding multiplexers.

8.9 Verilog code:

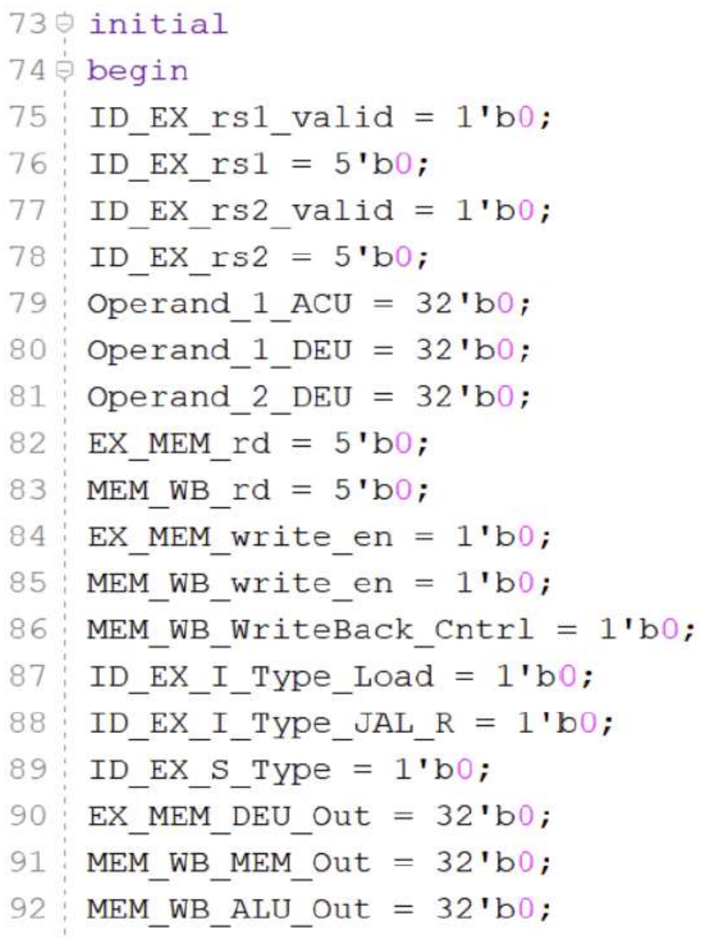


```verilog
// Forwarding Logic for Operand1 of ACU(Address Calculation Unit)
always@(ID_EX_rs1_valid,ID_EX_rs1,EX_MEM_rd,MEM_WB_rd,EX_MEM_write_en,
MEM_WB_write_en,MEM_WB_WriteBack_Cntrl,ID_EX_I_Type_Load,ID_EX_I_Type_JAL_R,ID_EX_S_Type)

begin
if(EX_MEM_write_en && ID_EX_rs1_valid && (EX_MEM_rd == ID_EX_rs1) &&
  (ID_EX_I_Type_Load | ID_EX_I_Type_JAL_R | ID_EX_S_Type))
Op_1_fwd_ACU = 2'b01; //Forward from alu_out field in EX/MEM register

else if(MEM_WB_write_en && ID_EX_rs1_valid && (MEM_WB_rd == ID_EX_rs1)&& !(MEM_WB_WriteBack_Cntrl)&&
        (ID_EX_I_Type_Load | ID_EX_I_Type_JAL_R | ID_EX_S_Type))
Op_1_fwd_ACU = 2'b10; // Forward data from memory_data_out field in MEM/WB register

else if(MEM_WB_write_en && ID_EX_rs1_valid && (MEM_WB_rd == ID_EX_rs1)&& MEM_WB_WriteBack_Cntrl &&
        (ID_EX_I_Type_Load | ID_EX_I_Type_JAL_R | ID_EX_S_Type))
Op_1_fwd_ACU = 2'b11; // Forward data from alu_out field in MEM/WB register

else Op_1_fwd_ACU = 2'b00; //Forward from Operand_1_ACU field in ID/EX pipeline register

end
```

8.10 Testbench:

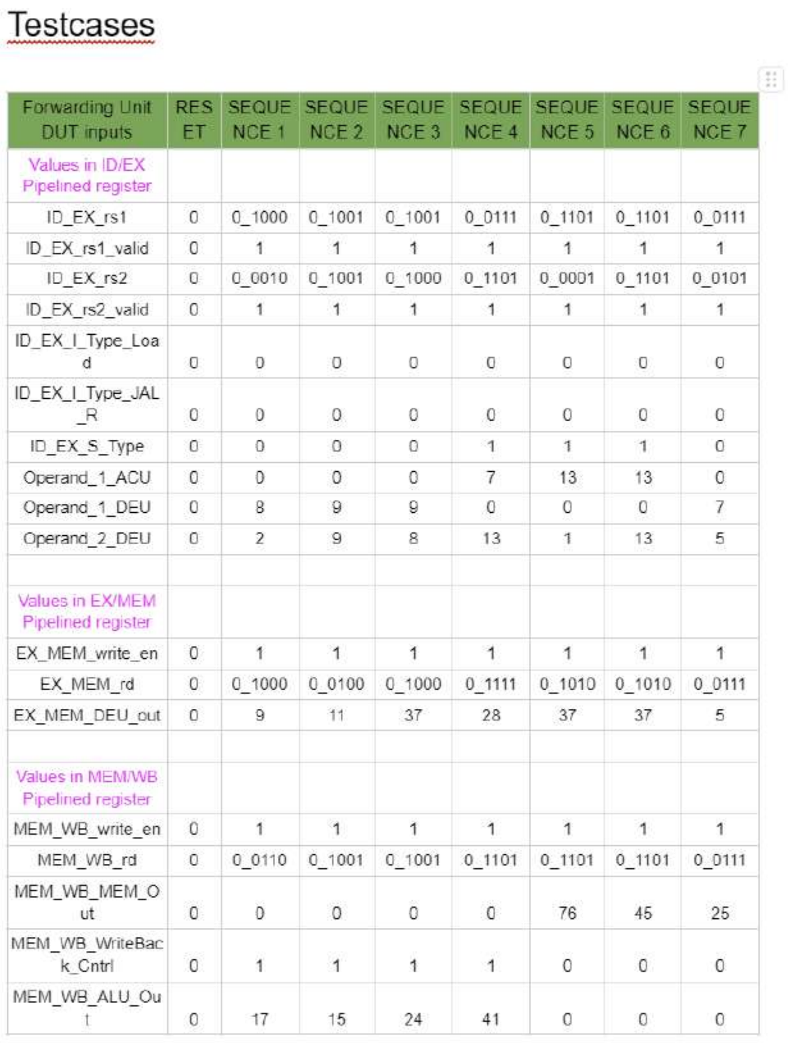


```
//Clock for the module
initial
begin
Clk = 0;
forever
#30 Clk = ~Clk;
end

//Reset for the module
initial
begin
Reset = 1;
#10 Reset = 0;
#10 Reset = 1;
#80 Reset = 0;
//#20 Reset = 0;
end
```



```
initial
begin
ID_EX_rs1_valid = 1'b0;
ID_EX_rs1 = 5'b0;
ID_EX_rs2_valid = 1'b0;
ID_EX_rs2 = 5'b0;
Operand_1_ACU = 32'b0;
Operand_1_DEU = 32'b0;
Operand_2_DEU = 32'b0;
EX_MEM_rd = 5'b0;
MEM_WB_rd = 5'b0;
EX_MEM_write_en = 1'b0;
MEM_WB_write_en = 1'b0;
MEM_WB_WriteBack_Cntrl = 1'b0;
ID_EX_I_Type_Load = 1'b0;
ID_EX_I_Type_JAL_R = 1'b0;
ID_EX_S_Type = 1'b0;
EX_MEM_DEU_Out = 32'b0;
MEM_WB_MEM_Out = 32'b0;
MEM_WB_ALU_Out = 32'b0;
```

8.11 Testcases

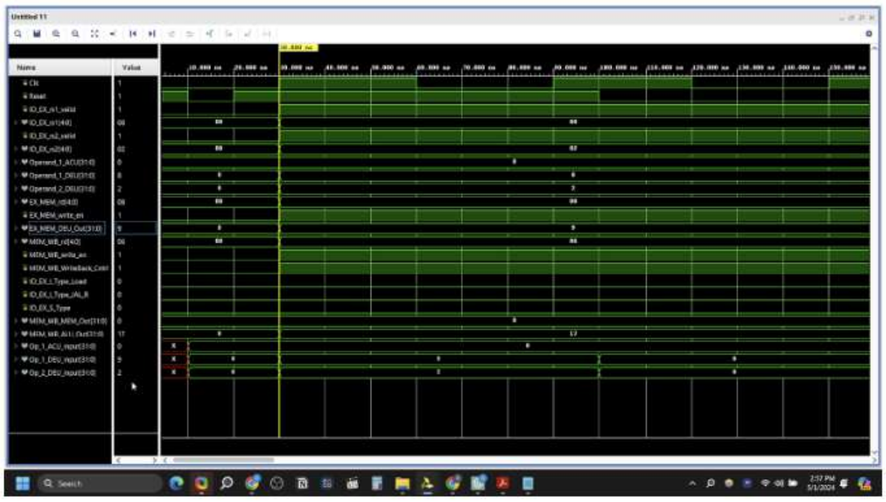


Testcases

Forwarding Unit DUT inputs	RESET	SEQUENCE 1	SEQUENCE 2	SEQUENCE 3	SEQUENCE 4	SEQUENCE 5	SEQUENCE 6	SEQUENCE 7
Values in ID/EX Pipelined register								
ID_EX_rs1	0	0_1000	0_1001	0_1001	0_0111	0_1101	0_1101	0_0111
ID_EX_rs1_valid	0	1	1	1	1	1	1	1
ID_EX_rs2	0	0_0010	0_1001	0_1000	0_1101	0_0001	0_1101	0_0101
ID_EX_rs2_valid	0	1	1	1	1	1	1	1
ID_EX_I_Type_Load	0	0	0	0	0	0	0	0
ID_EX_I_Type_JAL_R	0	0	0	0	0	0	0	0
ID_EX_S_Type	0	0	0	0	1	1	1	0
Operand_1_ACU	0	0	0	0	7	13	13	0
Operand_1_DEU	0	8	9	9	0	0	0	7
Operand_2_DEU	0	2	9	8	13	1	13	5
Values in EX/MEM Pipelined register								
EX_MEM_write_en	0	1	1	1	1	1	1	1
EX_MEM_rd	0	0_1000	0_0100	0_1000	0_1111	0_1010	0_1010	0_0111
EX_MEM_DEU_out	0	9	11	37	28	37	37	5
Values in MEM/WB Pipelined register								
MEM_WB_write_en	0	1	1	1	1	1	1	1
MEM_WB_rd	0	0_0110	0_1001	0_1001	0_1101	0_1101	0_1101	0_0111
MEM_WB_MEM_Out	0	0	0	0	0	76	45	25
MEM_WB_WriteBack_Cntrl	0	1	1	1	1	0	0	0
MEM_WB_ALU_Out	0	17	15	24	41	0	0	0

8.12 Simulation

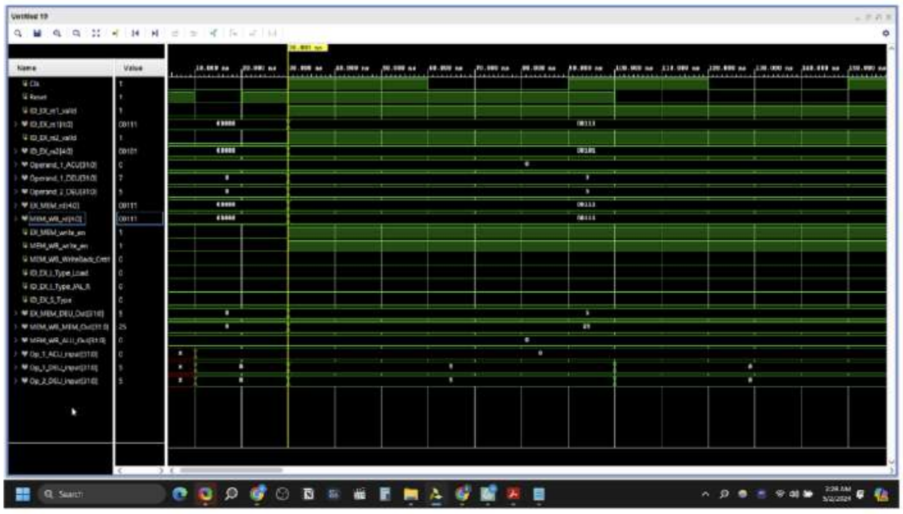
8.12.1 Sequence1



Untitled 11

Name	Value
Clk	1
Reset	1
ID_EX_rs1_valid	1
ID_EX_rs1[4:0]	08
ID_EX_rs2_valid	1
ID_EX_rs2[4:0]	02
Operand_1_ACU[31:0]	0
Operand_1_DEU[31:0]	8
Operand_2_DEU[31:0]	2
EX_MEM_rd[4:0]	08
EX_MEM_write_en	1
EX_MEM_DEU_Out[31:0]	9
MEM_WB_rd[4:0]	06
MEM_WB_write_en	1
MEM_WB_WriteBack_Cntrl	1
ID_EX_I_Type_Load	0
ID_EX_I_Type_ALU_R	0
ID_EX_S_Type	0
MEM_WB_MEM_Out[31:0]	0
MEM_WB_ALU_Out[31:0]	17
Op_1_ACU_input[31:0]	0
Op_1_DEU_input[31:0]	9
Op_2_DEU_input[31:0]	2

8.12.2 Sequence2



Untitled 19

Name	Value
Clk	1
Reset	1
ID_EX_rs1_valid	1
ID_EX_rs1[4:0]	00111
ID_EX_rs2_valid	1
ID_EX_rs2[4:0]	00101
Operand_1_ACU[31:0]	0
Operand_1_DEU[31:0]	7
Operand_2_DEU[31:0]	5
EX_MEM_rd[4:0]	00111
MEM_WB_rd[4:0]	00111
EX_MEM_write_en	1
MEM_WB_write_en	1
MEM_WB_WriteBack_Cntrl	0
ID_EX_I_Type_Load	0
ID_EX_I_Type_ALU_R	0
ID_EX_S_Type	0
EX_MEM_DEU_Out[31:0]	5
MEM_WB_MEM_Out[31:0]	25
MEM_WB_ALU_Out[31:0]	0
Op_1_ACU_input[31:0]	0
Op_1_DEU_input[31:0]	5
Op_2_DEU_input[31:0]	5

Chapter 9

Pipeline Hazard Management and Stalls

Our guidelines for data hazard management mentioned that if the consumer instruction happens to be one stage behind the producer instruction, but the producer instruction produces the operand value not in the EX stage but in the MEM stage. This situation cannot be overcome with internal forwarding. In such cases, a stall is introduced to separate the two instructions by one instruction. The instruction that separates the two instructions, in our case, happens to NOP (addi x0,x0,0). The introduction of NOP delays the further execution of the consumer instruction by one clock cycle. As the consumer instruction advances ahead, the required operand value is ready to be forwarded to the consumer via internal forwarding.

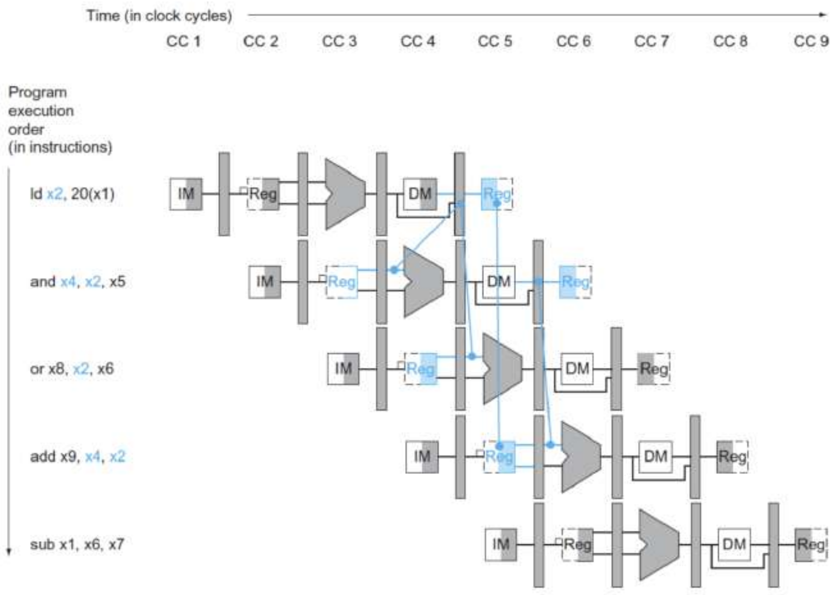


FIGURE 9.1: Input Image Read

9.1 Pipeline Dependencies and Stalls

Consider the sequence of instructions in the figure above, where instructions are represented temporally (with respect to time) in the five-stage pipeline. The dependencies are highlighted in blue. Here, `ld x2, 20(x1)` is the producer for subsequent `and x4, x2, x5`, `or x8, x2, x6`, and `add x9, x4, x2` instructions. `and x4, x2, x5` is also a producer for `add x9, x4, x2`.

If we go by the current internal forwarding system in place, we would have to go back in time to solve the dependency between instructions 1 and 2, which is impossible. In other words, the dependency cannot be solved if normal execution is allowed to proceed. The issue isn't that the forwarding doesn't work but that even the forwarding wouldn't solve the problem. If you refer to the forwarding condition laid down, you'll find that the `Operand1_DEU` for `and x4, x2, x5` is replaced by the output of `DEU` for `ld x2, 20(x1)`, which is `32'b0` and exactly the function of the forwarding unit.

To avoid this erroneous situation, we wait for the producer's instruction to provide the suitable operand in the `MEM` stage, and since that output would be required by the consumer, a stall for the consumer's instruction is necessary. The dependence between `ld x2, 20(x1)` and `or x8, x2, x6` is handled via normal forwarding logic, where the forwarding output comes from the `MEM/WB` pipeline register. A similar situation exists for `and x4, x2, x5` and `add x9, x4, x2`.

How a stall condition is realized is depicted in the following figure. The delay between instructions is realized, as discussed above, via a `NOP` instruction, which is also referred to as a "bubble," like a bubble in a physical pipe. It's quite evident that a stall impacts the performance of the pipeline, but in the grand scheme of things, when compared to sequential execution, the pipelined implementation emerges as a winner given the large number of instructions executed.

Here, the `EX` stage is bubbled by feeding the `ID/EX` register with a decoded version of the `NOP` instruction, whereas `and x4, x2, x5` is held back in the `ID` stage by freezing the `IF/ID` stage. This also implies that the corresponding program counter (PC) needs to be frozen as well.

9.2 Stall Detection

Regarding stall detection, one can question at which stage the consumer instruction should be stalled. A stall has to be succeeded with a forward from the `MEM/WB` stage of the producer to the `EX` stage of the consumer, which implies that under no circumstances should the consumer instruction proceed beyond the `EX` stage. This gives us two options to choose from. One option is to stall the consumer instruction in the `EX` stage or the `ID` stage. Again, this will require freeing either the `ID/EX` pipeline register or the `IF/ID` pipeline register. One would be inclined to do the detection when the consumer instruction reaches the `EX` stage, as was the case in internal

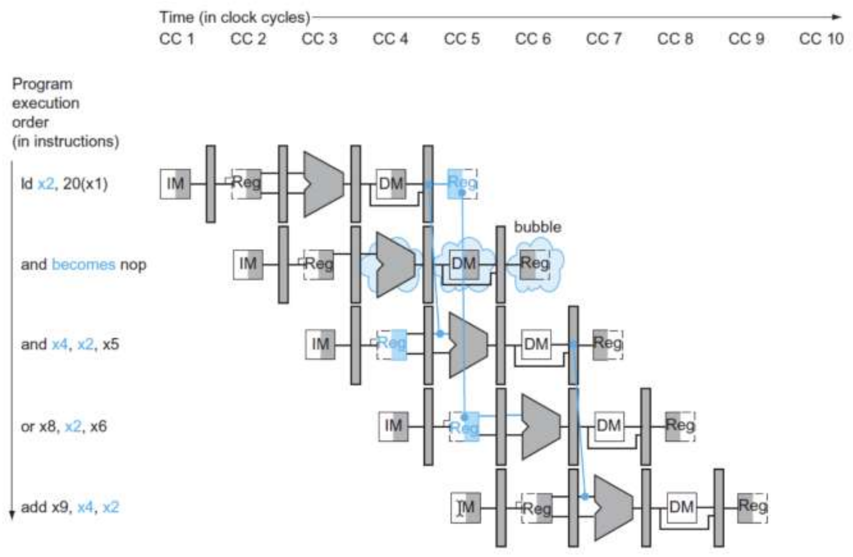


FIGURE 9.2: Input Image Read

forwarding. It also has the added benefit of getting decoded values of `rs1_valid` and `rs2_valid`, which are not available at the start of the decode stage but are made available during the `ID` stage. On the surface, detection when the consumer stage is in the `EX` stage seems like the right choice but should be avoided as is explained in the following sequence.

```
lw x7, x5, 0
lw x6, x4, 0
add x8, x7, x6
```

Since forwarding is required from two instructions ahead of `add`, both of which produce data values in the `MEM` stage, freezing the `add` instruction in the `EX` stage and injecting a bubble in the `EX/MEM` pipeline register will not provide the correct value of `x7`. Reading the correct value of `x7` by `add` will require repeating the `add` instruction in the `ID` stage by freezing the `IF/ID` pipeline register and injecting a bubble in the `ID/EX` pipeline register.

If one wants to proceed with the implementation scheme of detecting the stall condition when the consumer instruction enters the `EX` stage, they should also consider adding an extra pipelined register after the `WB` stage to facilitate the forwarding of the required operand for the consumer instruction. Stall detection in the `ID` stage has everything required in the `IF/ID` register except for the signals `rs1_valid` and `rs2_valid`, which are generated during the decode stage. Hence, these signals will be given through combinational logic to the pipeline management system, not through pipeline registers. This approach would result in successful execution but comes at the cost of increasing combinational logic between the `IF/ID` and `ID/EX`, thus increasing the minimum clock period required for stage completion.

9.3 Pipeline Management Block

Stalling an instruction is achieved by altering the normal operation of the pipeline system such that a bubble is required in the `EX` stage, and the `ID` stage has to be frozen. Another case where the normal operation of the pipeline system is altered is in the case of control hazards because of control transfer instructions such as `B-type`, `JAL`, and `JALR` instructions. In these instructions, both the `IF/ID` and `ID/EX` stages need to be flushed, or a bubble is inserted in both the `ID` and `EX` stages.

All these changes are done by altering the contents of pipeline registers, which represent the state of each stage. To facilitate changing these states, each field of the pipeline register is given via a multiplexer to both the `IF/ID` and `ID/EX` registers. One must also remember that altering these registers has a cascading effect on the program counter (PC). Hence, it's also controlled by a multiplexer to provide input addresses. To compare the states of both pipelined registers (`IF/ID` and `ID/EX`) and generate corresponding control signals for these multiplexers, a controller or module is required. We name this module as **Pipeline Management**.

9.3.1 Pipeline Management Block

Stalling an instruction is achieved by altering the normal pipeline system operations, inserting a bubble in the `EX` stage, and freezing the `ID` stage. Control hazards due to control transfer instructions like `B-type`, `JAL`, and `JALR` also require flushing the `IF/ID` and `ID/EX` stages or inserting bubbles.

The contents of pipeline registers represent the state of each stage. To facilitate these states, each field of pipeline registers is controlled by a multiplexer feeding both `IF/ID` and `ID/EX`. Altering these registers has cascading effects on the program counter (PC), which is also controlled by a multiplexer. A controller or module is required to compare the states of the `IF/ID` and `ID/EX` registers and generate corresponding control signals. This module is called **Pipeline Management**.

9.4 Pipeline Management Sequences and Control Signals

The Pipeline Management module was subjected to the following sequence of instructions. The corresponding control signals for these instructions are listed in the table shown above.

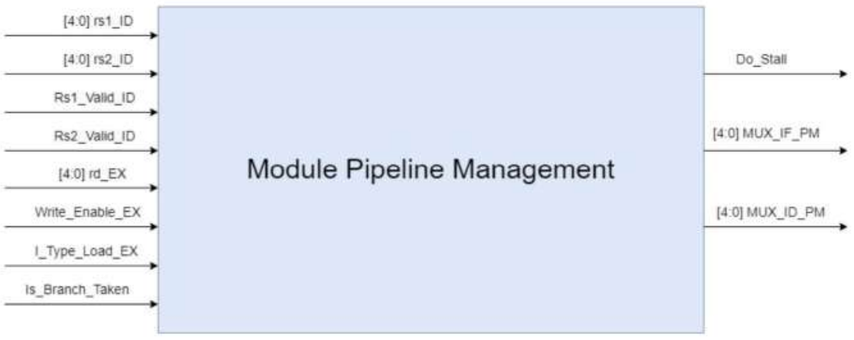
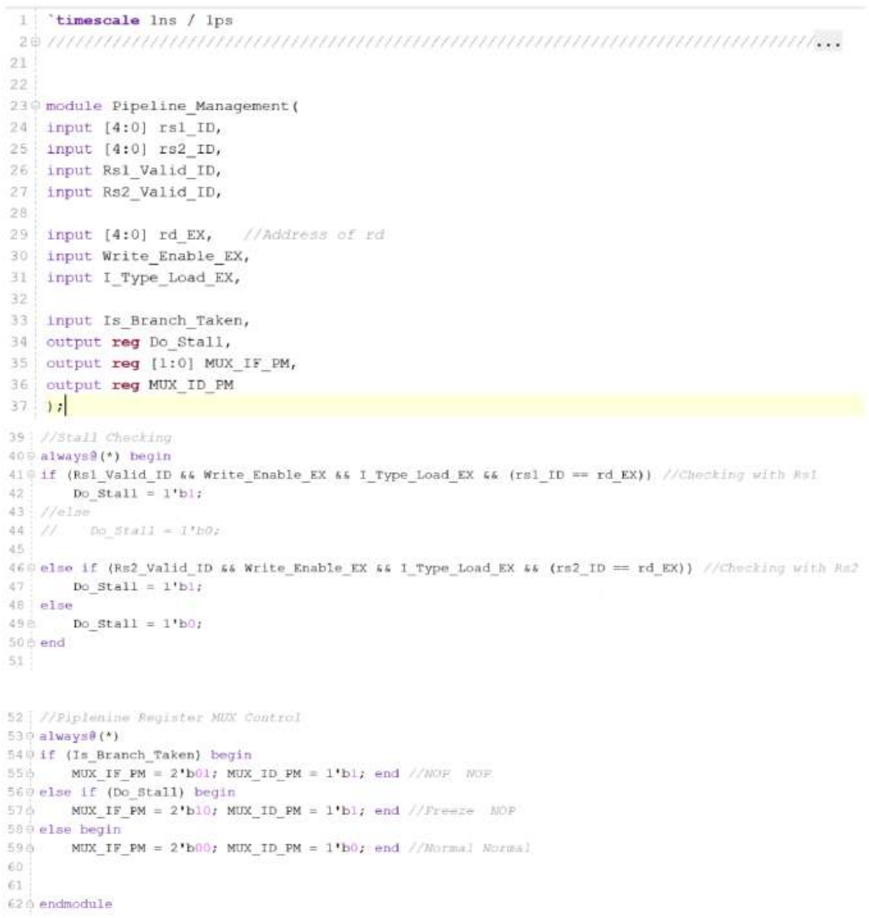


FIGURE 9.3: Pipeline Management Block Diagram

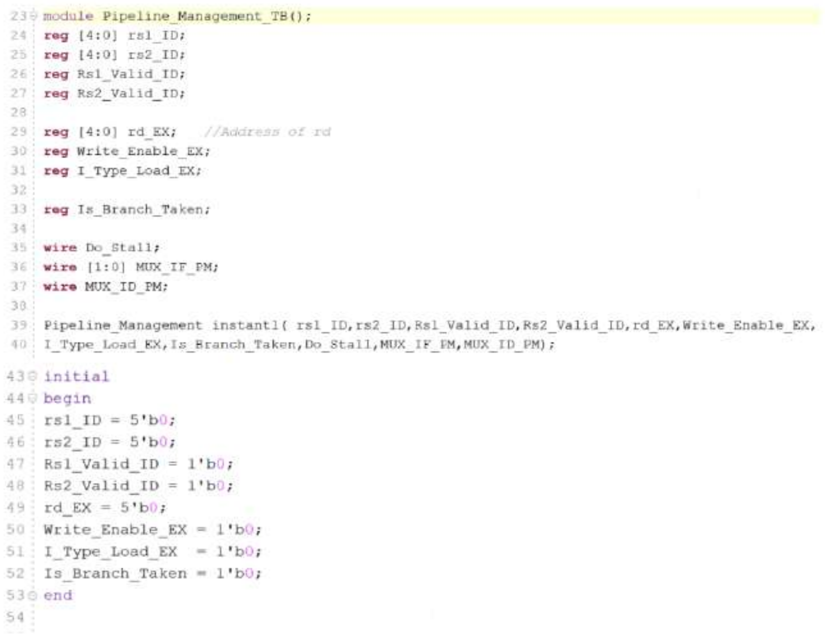


```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////...


module Pipeline_Management(
input [4:0] rs1_ID,
input [4:0] rs2_ID,
input Rs1_Valid_ID,
input Rs2_Valid_ID,

input [4:0] rd_EX,    //Address of rd
input Write_Enable_EX,
input I_Type_Load_EX,

input Is_Branch_Taken,
output reg Do_Stall,
output reg [1:0] MUX_IF_PM,
output reg MUX_ID_PM
);

//Stall Checking
always@(*) begin
if (Rs1_Valid_ID && Write_Enable_EX && I_Type_Load_EX && (rs1_ID == rd_EX)) //Checking with Rs1
    Do_Stall = 1'b1;
//else
//    Do_Stall = 1'b0;

else if (Rs2_Valid_ID && Write_Enable_EX && I_Type_Load_EX && (rs2_ID == rd_EX)) //Checking with Rs2
    Do_Stall = 1'b1;
else
    Do_Stall = 1'b0;
end


//Piplenine Register MUX Control
always@(*)
if (Is_Branch_Taken) begin
    MUX_IF_PM = 2'b01; MUX_ID_PM = 1'b1; end //NOP  NOP
else if (Do_Stall) begin
    MUX_IF_PM = 2'b10; MUX_ID_PM = 1'b1; end //Freeze  NOP
else begin
    MUX_IF_PM = 2'b00; MUX_ID_PM = 1'b0; end //Normal Normal


endmodule
```

FIGURE 9.4: Verilog code

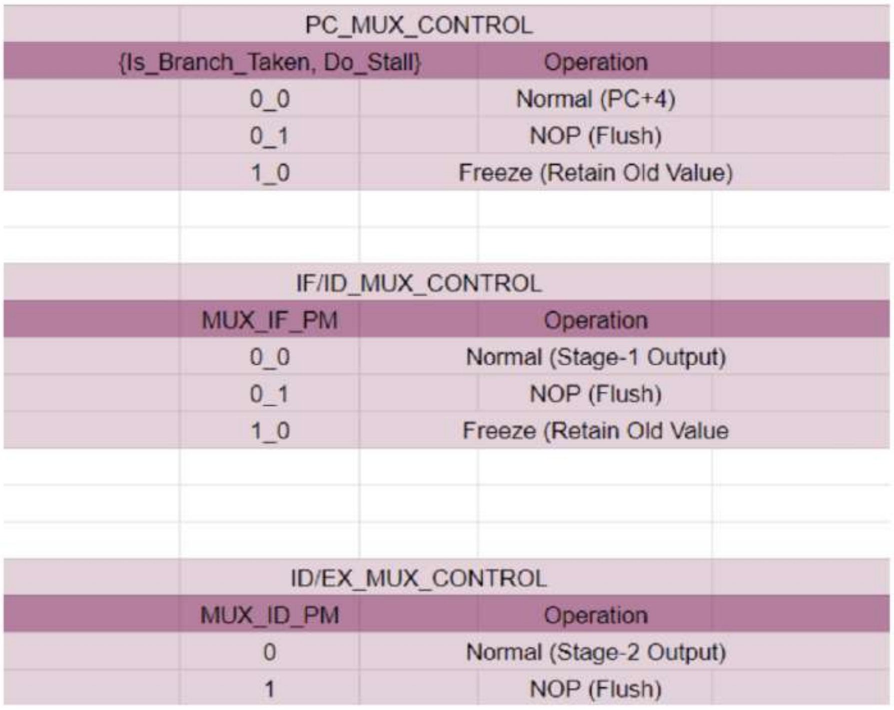
Sequence 1	Sequence 2	Sequence 3
lw x4, x3, 21	lw x3, x8, 7	beq x4, x4, 32
add x7, x6, x5	add x9, x3, x13	add x14, x13, x15
add x13, x9, x2	add x7, x7, x7	add x9, x3, x5

Mux Control via Pipeline Management



```verilog
module Pipeline_Management_TB();
reg [4:0] rs1_ID;
reg [4:0] rs2_ID;
reg Rs1_Valid_ID;
reg Rs2_Valid_ID;

reg [4:0] rd_EX;    //Address of rd
reg Write_Enable_EX;
reg I_Type_Load_EX;

reg Is_Branch_Taken;

wire Do_Stall;
wire [1:0] MUX_IF_PM;
wire MUX_ID_PM;

Pipeline_Management instant1( rs1_ID,rs2_ID,Rs1_Valid_ID,Rs2_Valid_ID,rd_EX,Write_Enable_EX,
I_Type_Load_EX,Is_Branch_Taken,Do_Stall,MUX_IF_PM,MUX_ID_PM);

initial
begin
rs1_ID = 5'b0;
rs2_ID = 5'b0;
Rs1_Valid_ID = 1'b0;
Rs2_Valid_ID = 1'b0;
rd_EX = 5'b0;
Write_Enable_EX = 1'b0;
I_Type_Load_EX  = 1'b0;
Is_Branch_Taken = 1'b0;
end
```

FIGURE 9.5: Testbench



PC_MUX_CONTROL	
{Is_Branch_Taken, Do_Stall}	Operation
0_0	Normal (PC+4)
0_1	NOP (Flush)
1_0	Freeze (Retain Old Value)

IF/ID_MUX_CONTROL	
MUX_IF_PM	Operation
0_0	Normal (Stage-1 Output)
0_1	NOP (Flush)
1_0	Freeze (Retain Old Value

ID/EX_MUX_CONTROL	
MUX_ID_PM	Operation
0	Normal (Stage-2 Output)
1	NOP (Flush)

FIGURE 9.6: Testbench

The above table shows the three multiplexers controlled by the Pipeline Management System and the corresponding operations for the given control.

Pipeline Management DUT Inputs	**SEQUENCE 1**	**SEQUENCE 2**	**SEQUENCE 3**
Values in IF/ID Pipelined Register			
rs1_ID	0_0110	0_0011	0_1101
rs2_ID	0_0010	0_1101	0_1111
Rs1_Valid_ID	1	1	1
Rs2_Valid_ID	1	1	1
Values in ID/EX Pipelined Register			
rd_EX	0_0100	0_0011	0_0000
Write_Enable_EX	1	1	0
I_Type_Load_EX	0	1	0
Is_Branch_Taken	0	0	1
MUX Controls for PC, IF/ID, ID/EX			
{Is_Branch_Taken, Do_Stall}	0_0	0_1	1_0
MUX_IF_PM	0_0	1_0	0_1
MUX_ID_PM	0_0	0_1	0_1

Table 9.1: Pipeline Management Data Sequences and Control Signals

9.5 Simulation Results

For the three sequences of instructions, the Pipeline Management module was simulated, and the following results were obtained.

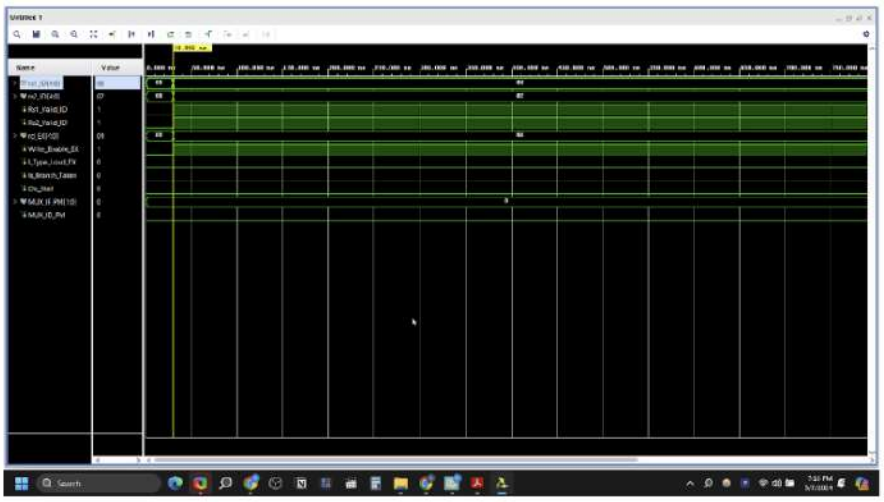


Figure 9.7: Normal Condition

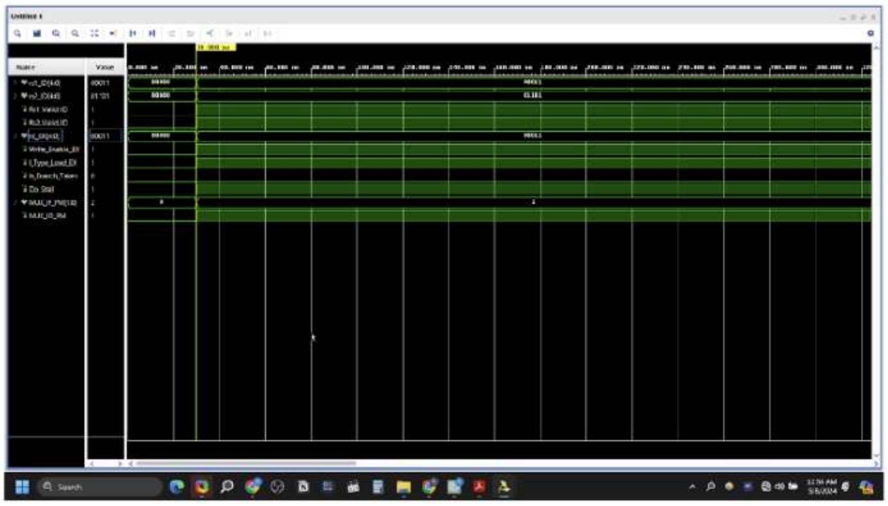


Figure 9.8: Stall Condition

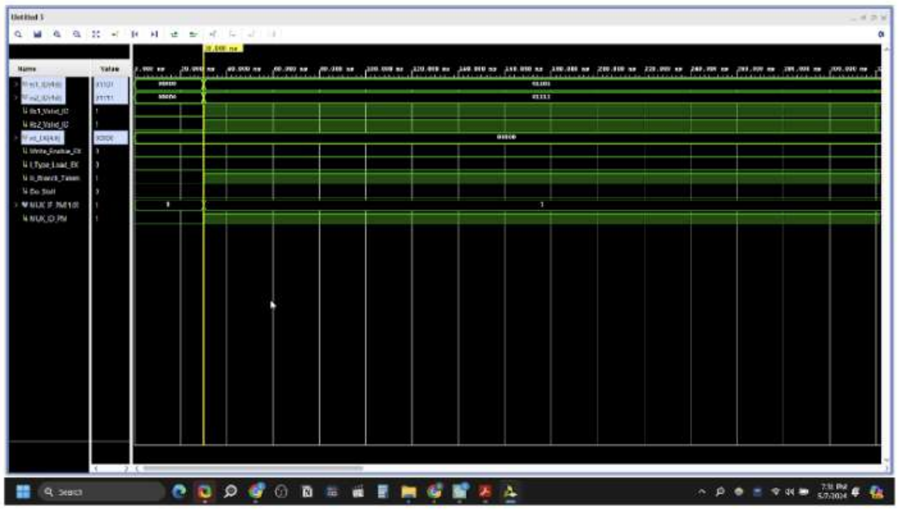
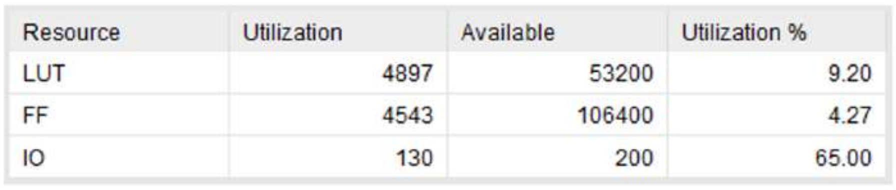


Figure 9.9: Jump Control

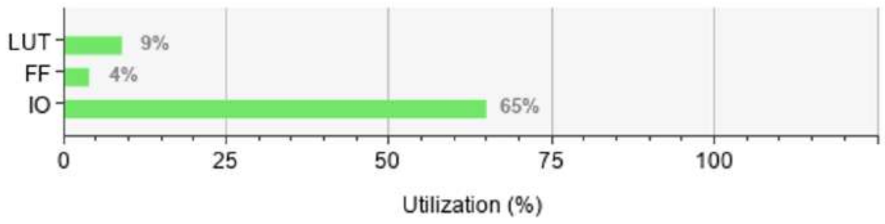
Chapter 10

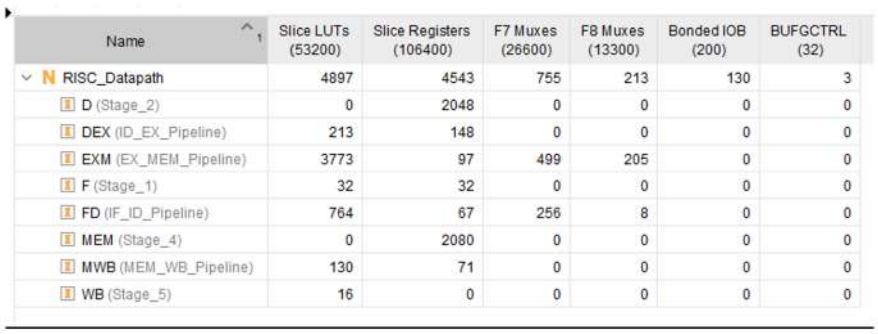
Final Verification of the Interlocked RISC-V core

10.1 Utilization



Resource	Utilization	Available	Utilization %
LUT	4897	53200	9.20
FF	4543	106400	4.27
IO	130	200	65.00

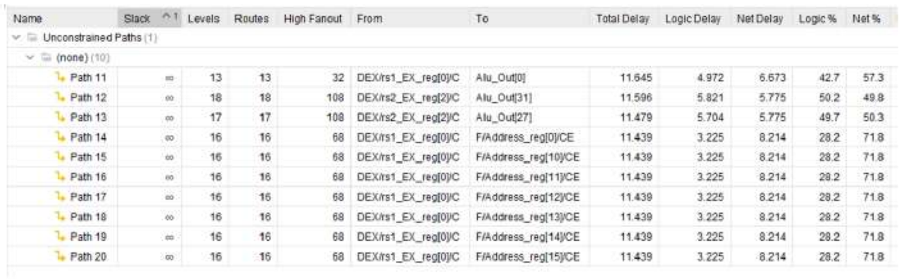




Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Bonded IOB (200)	BUFGCTRL (32)
RISC_Datapath	4897	4543	755	213	130	3
D (Stage_2)	0	2048	0	0	0	0
DEX (ID_EX_Pipeline)	213	148	0	0	0	0
EXM (EX_MEM_Pipeline)	3773	97	499	205	0	0
F (Stage_1)	32	32	0	0	0	0
FD (IF_ID_Pipeline)	764	67	256	8	0	0
MEM (Stage_4)	0	2080	0	0	0	0
MWB (MEM_WB_Pipeline)	130	71	0	0	0	0
WB (Stage_5)	16	0	0	0	0	0

10.2 Timings:

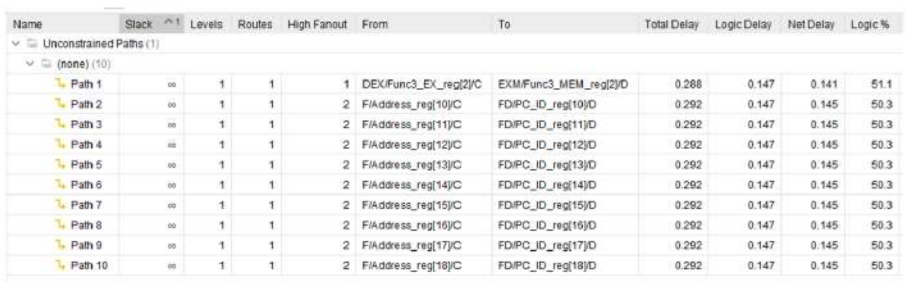
10.2.1 Setup:



Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Logic %	Net %
Unconstrained Paths (1)											
(none) (10)											
Path 11	∞	13	13	32	DEX/rs1_EX_reg[0]/C	Alu_Out[0]	11.645	4.972	6.673	42.7	57.3
Path 12	∞	18	18	108	DEX/rs2_EX_reg[2]/C	Alu_Out[31]	11.596	5.821	5.775	50.2	49.8
Path 13	∞	17	17	108	DEX/rs2_EX_reg[2]/C	Alu_Out[27]	11.479	5.704	5.775	49.7	50.3
Path 14	∞	16	16	68	DEX/rs1_EX_reg[0]/C	F/Address_reg[0]/CE	11.439	3.225	8.214	28.2	71.8
Path 15	∞	16	16	68	DEX/rs1_EX_reg[0]/C	F/Address_reg[10]/CE	11.439	3.225	8.214	28.2	71.8
Path 16	∞	16	16	68	DEX/rs1_EX_reg[0]/C	F/Address_reg[11]/CE	11.439	3.225	8.214	28.2	71.8
Path 17	∞	16	16	68	DEX/rs1_EX_reg[0]/C	F/Address_reg[12]/CE	11.439	3.225	8.214	28.2	71.8
Path 18	∞	16	16	68	DEX/rs1_EX_reg[0]/C	F/Address_reg[13]/CE	11.439	3.225	8.214	28.2	71.8
Path 19	∞	16	16	68	DEX/rs1_EX_reg[0]/C	F/Address_reg[14]/CE	11.439	3.225	8.214	28.2	71.8
Path 20	∞	16	16	68	DEX/rs1_EX_reg[0]/C	F/Address_reg[15]/CE	11.439	3.225	8.214	28.2	71.8

Max: 11.645ns

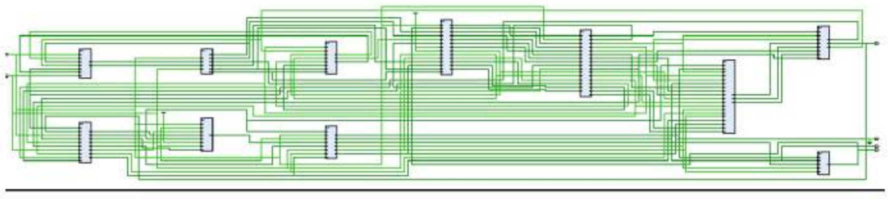
10.2.2 Hold:



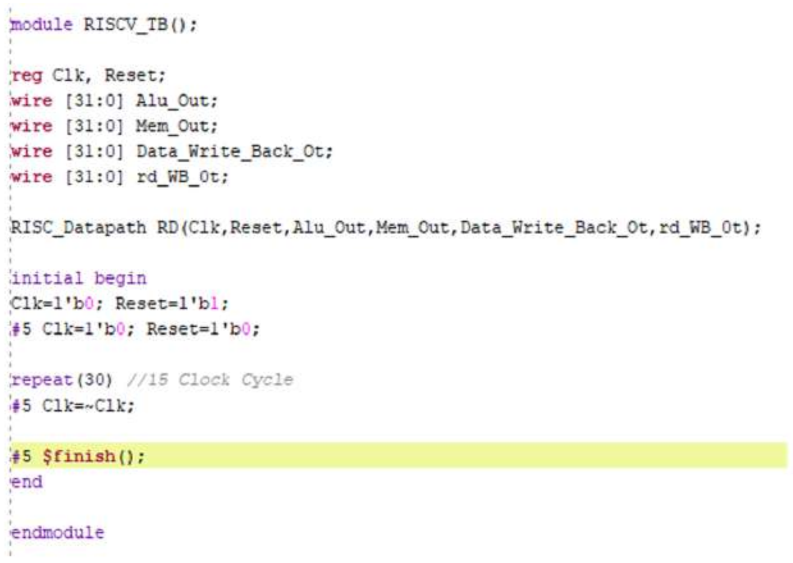
Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Logic %
Unconstrained Paths (1)										
(none) (10)										
Path 1	∞	1	1	1	DEX/Func3_EX_reg[2]/C	EXM/Func3_MEM_reg[2]/D	0.288	0.147	0.141	51.1
Path 2	∞	1	1	2	F/Address_reg[10]/C	FD/PC_ID_reg[10]/D	0.292	0.147	0.145	50.3
Path 3	∞	1	1	2	F/Address_reg[11]/C	FD/PC_ID_reg[11]/D	0.292	0.147	0.145	50.3
Path 4	∞	1	1	2	F/Address_reg[12]/C	FD/PC_ID_reg[12]/D	0.292	0.147	0.145	50.3
Path 5	∞	1	1	2	F/Address_reg[13]/C	FD/PC_ID_reg[13]/D	0.292	0.147	0.145	50.3
Path 6	∞	1	1	2	F/Address_reg[14]/C	FD/PC_ID_reg[14]/D	0.292	0.147	0.145	50.3
Path 7	∞	1	1	2	F/Address_reg[15]/C	FD/PC_ID_reg[15]/D	0.292	0.147	0.145	50.3
Path 8	∞	1	1	2	F/Address_reg[16]/C	FD/PC_ID_reg[16]/D	0.292	0.147	0.145	50.3
Path 9	∞	1	1	2	F/Address_reg[17]/C	FD/PC_ID_reg[17]/D	0.292	0.147	0.145	50.3
Path 10	∞	1	1	2	F/Address_reg[18]/C	FD/PC_ID_reg[18]/D	0.292	0.147	0.145	50.3

Min: 0.288 Frequency: 85.83 Mhz

10.3 Schematic of the whole processor:



10.4 Testbench:



```
module RISCV_TB();

reg Clk, Reset;
wire [31:0] Alu_Out;
wire [31:0] Mem_Out;
wire [31:0] Data_Write_Back_Ot;
wire [31:0] rd_WB_Ot;

RISC_Datapath RD(Clk,Reset,Alu_Out,Mem_Out,Data_Write_Back_Ot,rd_WB_Ot);

initial begin
Clk=1'b0; Reset=1'b1;
#5 Clk=1'b0; Reset=1'b0;

repeat(30) //15 Clock Cycle
#5 Clk=~Clk;

#5 $finish();
end

endmodule
```

10.5 Results and Discussion:

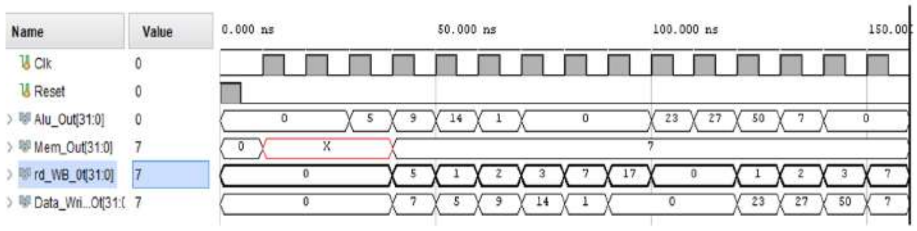
We have executed 6 instructions to verify the bubble insertion and data forwarding in full processor. It also ensures the normal operation of instruction.

PC Value	Instruction	Execution stage operation	Rd	Rd Value
0	lb $5 1($6)	6+1	$5	7
4	add $1 $2 $3	2+3	$1	5
8	add $2 $1 $4	5+4	$2	9
12	add $3 $2 $1	9+5	$3	14
16	sub $7 $8 $7	8-7	$7	1
20	beq $5 $5 -16	20-16		

As the branch condition is valid, the instruction in the fetch and decode stage will have the bubble, and the jump address is 20-16 = 4. Hence PC value will become 4. After all, 5 instructions from PC = 4 are executed.

PC Value	Instruction	Execution stage operation	Rd	Rd Value
4	add $1 $2 $3	9+14	$1	23
8	add $2 $1 $4	23+4	$2	27
12	add $3 $2 $1	27+23	$3	50
16	sub $7 $8 $7	8-1	$7	7
20	beq $5 $5 -16	20-16		

As the branch condition is valid, the instruction in the fetch and decode stage will have the bubble, and the jump address is 20-16 = 4. So, the PC value will again become 4, and all 5 instructions will execute again. But for our convenience, we stop the simulation after 15th clock.

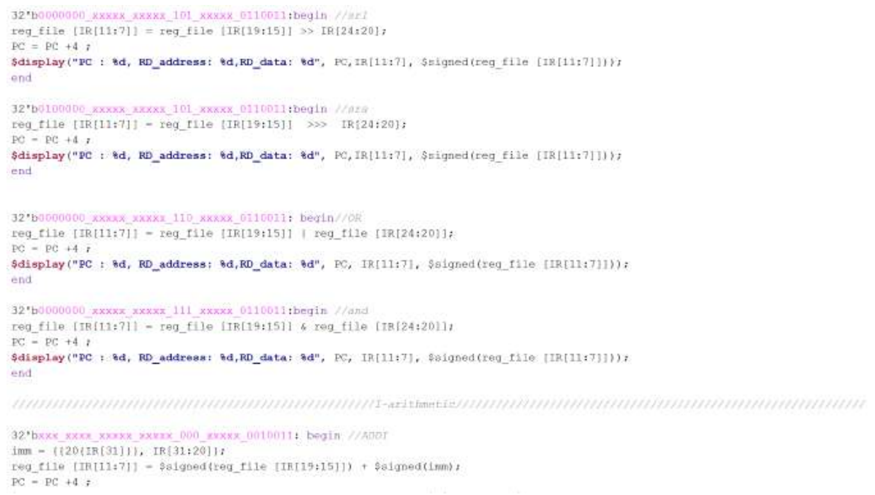

Name
Value
0.000 ns
50.000 ns
100.000 ns
150.00
Clk 0
Reset 0
Alu_Out[31:0] 0
Mem_Out[31:0] 7
rd_WB_0t[31:0] 7
Data_Wri...Ot[31:(7
0
5
9
14
1
0
23
27
50
7
0
0
X
7
0
5
1
2
3
7
17
0
1
2
3
7
0
7
5
9
14
1
0
23
27
50
7

Chapter 11

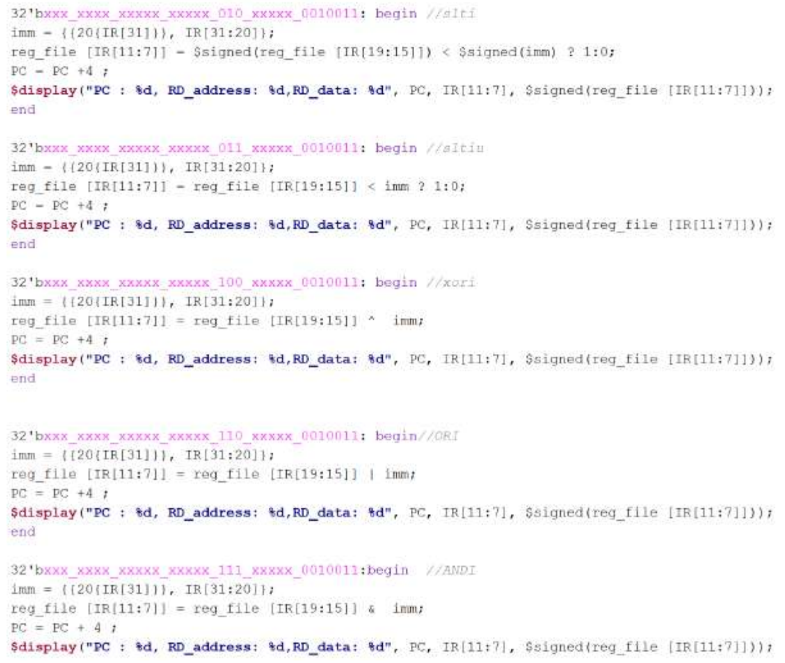
RISC-V SIMULATOR

11.1 Verilog Code:

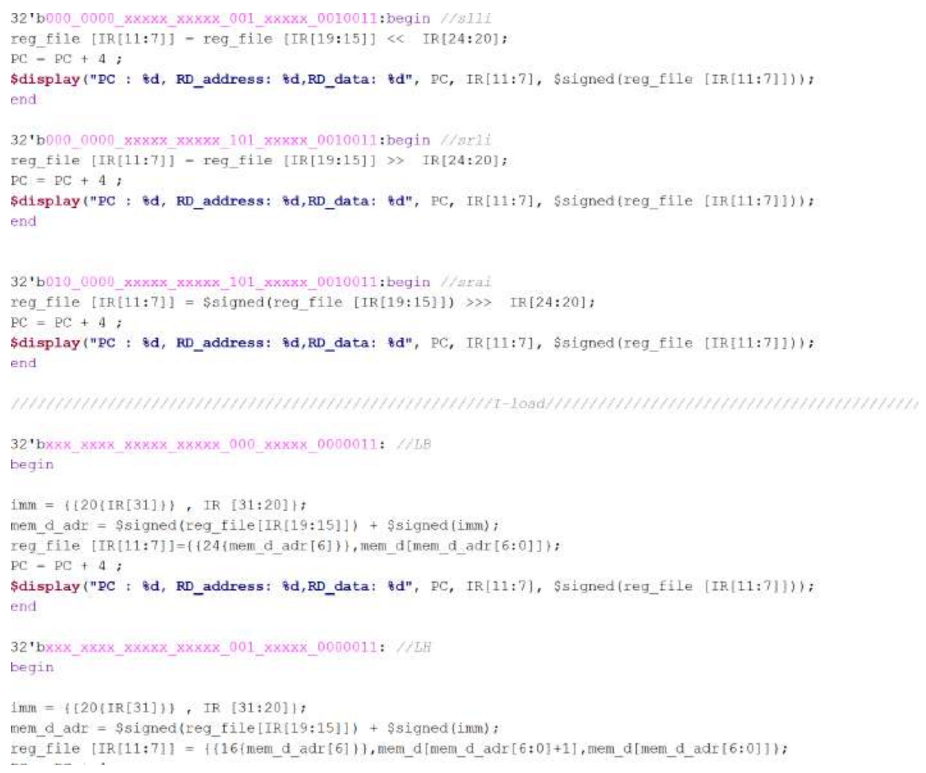
```
module Func_verify(

    input [31:0] IR

);

reg [31:0] reg_file [0:31];
reg [31:0] imm;
reg [31:0] mem_d_adr;
reg [31:0] mem_d [0:127];
reg [31:0] PC;
reg [31:0] rs2_value;
integer i = 0;

initial begin //////////initialize PC
PC = 0;
end

initial begin ///////////initialize register file
for(i = 0; i<31;i=i+1)
reg_file[i] = i;
end


initial begin //////////initialize data memory
for(i=0;i<127;i=i+1)
mem_d[i] = i;
end


always@(IR) begin

casex(IR)
32'b0000000_xxxxx_xxxxx_000_xxxxx_0110011:begin //add
reg_file [IR[11:7]] - $signed(reg_file[IR[19:15]]) + $signed(reg_file[IR[24:20]]);
PC = PC + 4;

$display("PC : %d, RD_address: %d,RD_data: %d", PC,IR[11:7], $signed(reg_file [IR[11:7]]));
```



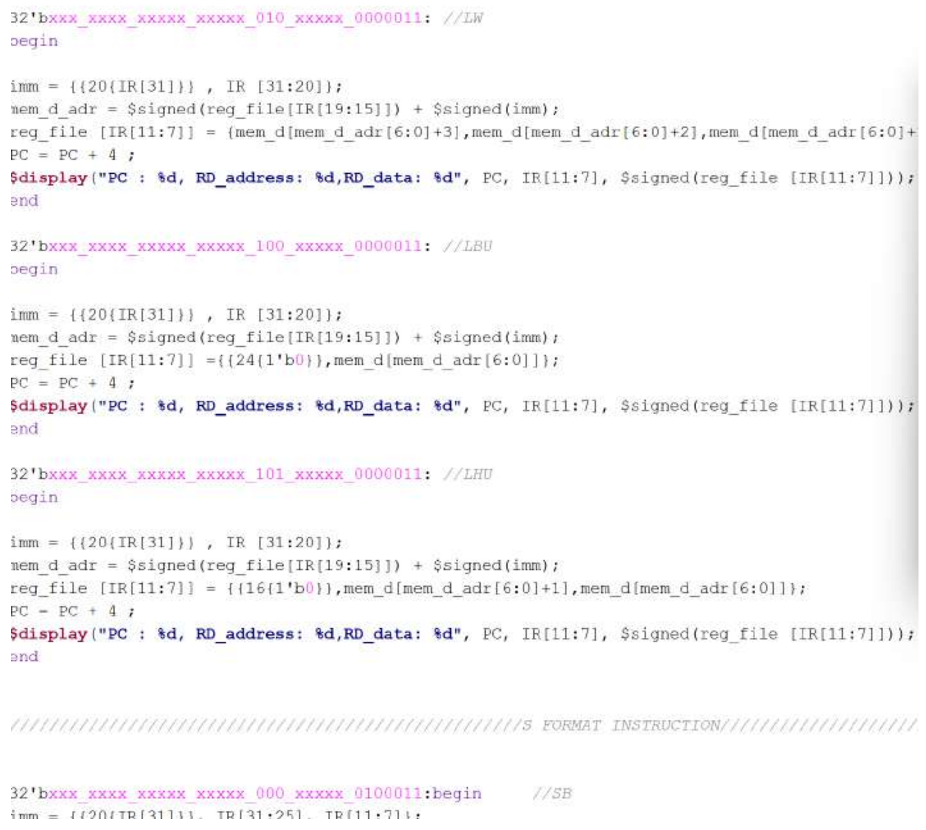
```
32'b0000000_xxxxx_xxxxx_101_xxxxx_0110011:begin //srl
reg_file [IR[11:7]] = reg_file [IR[19:15]] >> IR[24:20];
PC = PC +4 ;
$display("PC : %d, RD_address: %d,RD_data: %d", PC,IR[11:7], $signed(reg_file [IR[11:7]]));
end

32'b0100000_xxxxx_xxxxx_101_xxxxx_0110011:begin //sra
reg_file [IR[11:7]] = reg_file [IR[19:15]]  >>>  IR[24:20];
PC = PC +4 ;
$display("PC : %d, RD_address: %d,RD_data: %d", PC,IR[11:7], $signed(reg_file [IR[11:7]]));
end


32'b0000000_xxxxx_xxxxx_110_xxxxx_0110011: begin//OR
reg_file [IR[11:7]] = reg_file [IR[19:15]] | reg_file [IR[24:20]];
PC = PC +4 ;
$display("PC : %d, RD_address: %d,RD_data: %d", PC, IR[11:7], $signed(reg_file [IR[11:7]]));
end

32'b0000000_xxxxx_xxxxx_111_xxxxx_0110011:begin //and
reg_file [IR[11:7]] = reg_file [IR[19:15]] & reg_file [IR[24:20]];
PC = PC +4 ;
$display("PC : %d, RD_address: %d,RD_data: %d", PC, IR[11:7], $signed(reg_file [IR[11:7]]));
end

//////////////////////////////////////////////////////I-arithmetic/////////////////////////////////////////////////////////////

32'bxxx_xxxx_xxxxx_xxxxx_000_xxxxx_0010011: begin //ADDI
imm = {{20{IR[31]}}, IR[31:20]};
reg_file [IR[11:7]] = $signed(reg_file [IR[19:15]]) + $signed(imm);
PC = PC +4 ;
```



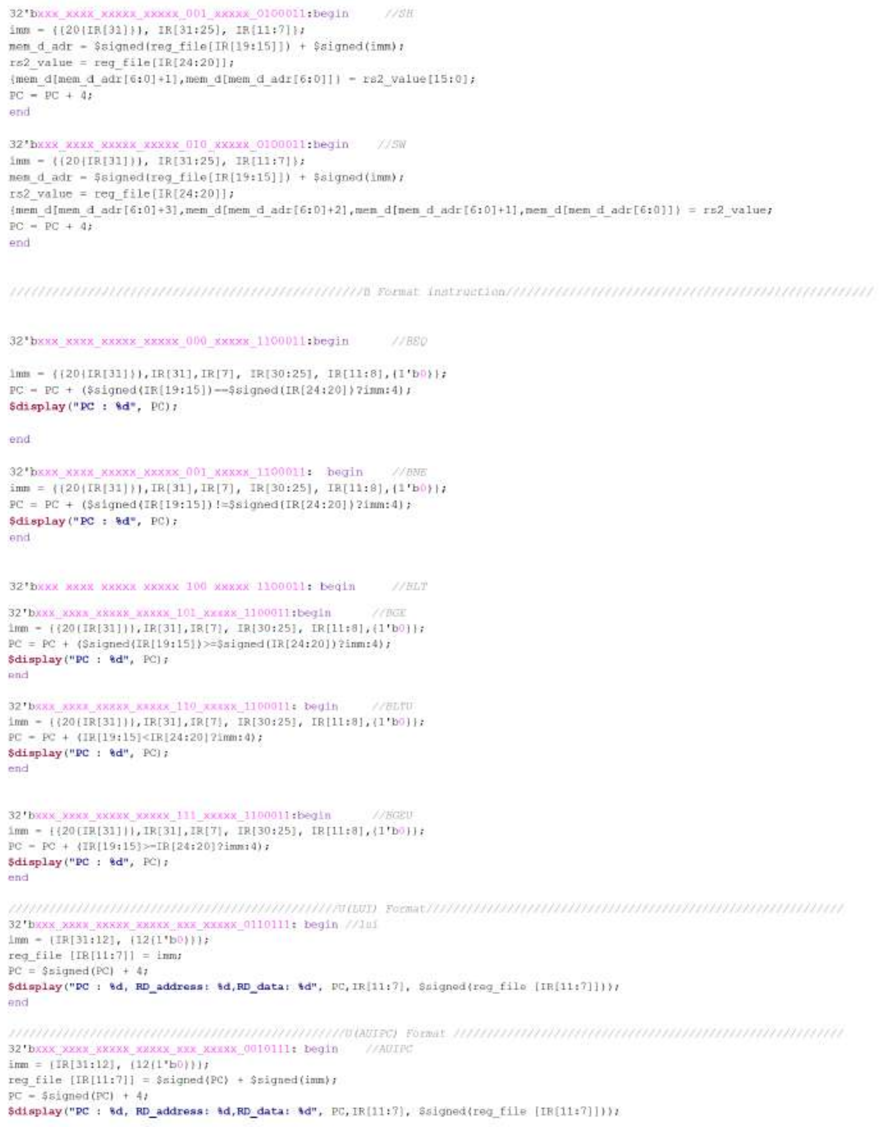
```
32'bxxx_xxxx_xxxxx_xxxxx_010_xxxxx_0010011: begin //slti
imm = {{20{IR[31]}}, IR[31:20]};
reg_file [IR[11:7]] = $signed(reg_file [IR[19:15]]) < $signed(imm) ? 1:0;
PC = PC +4 ;
$display("PC : %d, RD_address: %d,RD_data: %d", PC, IR[11:7], $signed(reg_file [IR[11:7]]));
end

32'bxxx_xxxx_xxxxx_xxxxx_011_xxxxx_0010011: begin //sltiu
imm = {{20{IR[31]}}, IR[31:20]};
reg_file [IR[11:7]] = reg_file [IR[19:15]] < imm ? 1:0;
PC = PC +4 ;
$display("PC : %d, RD_address: %d,RD_data: %d", PC, IR[11:7], $signed(reg_file [IR[11:7]]));
end

32'bxxx_xxxx_xxxxx_xxxxx_100_xxxxx_0010011: begin //xori
imm = {{20{IR[31]}}, IR[31:20]};
reg_file [IR[11:7]] = reg_file [IR[19:15]] ^  imm;
PC = PC +4 ;
$display("PC : %d, RD_address: %d,RD_data: %d", PC, IR[11:7], $signed(reg_file [IR[11:7]]));
end


32'bxxx_xxxx_xxxxx_xxxxx_110_xxxxx_0010011: begin//ORI
imm = {{20{IR[31]}}, IR[31:20]};
reg_file [IR[11:7]] = reg_file [IR[19:15]] | imm;
PC = PC +4 ;
$display("PC : %d, RD_address: %d,RD_data: %d", PC, IR[11:7], $signed(reg_file [IR[11:7]]));
end

32'bxxx_xxxx_xxxxx_xxxxx_111_xxxxx_0010011:begin  //ANDI
imm = {{20{IR[31]}}, IR[31:20]};
reg_file [IR[11:7]] = reg_file [IR[19:15]] &  imm;
PC = PC + 4 ;
$display("PC : %d, RD_address: %d,RD_data: %d", PC, IR[11:7], $signed(reg_file [IR[11:7]]));
```



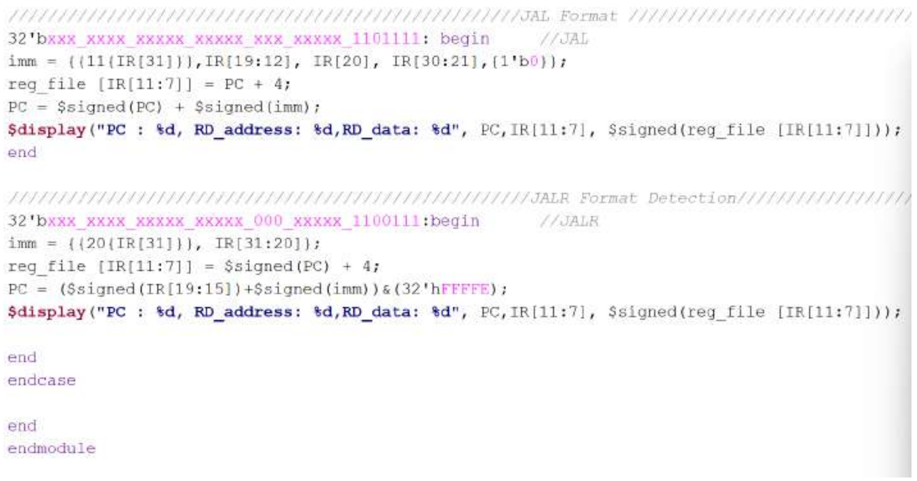
```
32'b000_0000_xxxxx_xxxxx_001_xxxxx_0010011:begin //slli
reg_file [IR[11:7]] = reg_file [IR[19:15]] <<  IR[24:20];
PC = PC + 4 ;
$display("PC : %d, RD_address: %d,RD_data: %d", PC, IR[11:7], $signed(reg_file [IR[11:7]]));
end

32'b000_0000_xxxxx_xxxxx_101_xxxxx_0010011:begin //srli
reg_file [IR[11:7]] = reg_file [IR[19:15]] >>  IR[24:20];
PC = PC + 4 ;
$display("PC : %d, RD_address: %d,RD_data: %d", PC, IR[11:7], $signed(reg_file [IR[11:7]]));
end


32'b010_0000_xxxxx_xxxxx_101_xxxxx_0010011:begin //srai
reg_file [IR[11:7]] = $signed(reg_file [IR[19:15]]) >>>  IR[24:20];
PC = PC + 4 ;
$display("PC : %d, RD_address: %d,RD_data: %d", PC, IR[11:7], $signed(reg_file [IR[11:7]]));
end

/////////////////////////////////////////////////////////I-load////////////////////////////////////////////

32'bxxx_xxxx_xxxxx_xxxxx_000_xxxxx_0000011: //LB
begin

imm = {{20{IR[31]}} , IR [31:20]};
mem_d_adr = $signed(reg_file[IR[19:15]]) + $signed(imm);
reg_file [IR[11:7]]={{24{mem_d_adr[6]}},mem_d[mem_d_adr[6:0]]};
PC = PC + 4 ;
$display("PC : %d, RD_address: %d,RD_data: %d", PC, IR[11:7], $signed(reg_file [IR[11:7]]));
end

32'bxxx_xxxx_xxxxx_xxxxx_001_xxxxx_0000011: //LH
begin

imm = {{20{IR[31]}} , IR [31:20]};
mem_d_adr = $signed(reg_file[IR[19:15]]) + $signed(imm);
reg_file [IR[11:7]] = {{16{mem_d_adr[6]}},mem_d[mem_d_adr[6:0]+1],mem_d[mem_d_adr[6:0]]};
```



```
32'bxxx_xxxx_xxxxx_xxxxx_010_xxxxx_0000011: //LW
begin

imm = {{20{IR[31]}} , IR [31:20]};
mem_d_adr = $signed(reg_file[IR[19:15]]) + $signed(imm);
reg_file [IR[11:7]] = {mem_d[mem_d_adr[6:0]+3],mem_d[mem_d_adr[6:0]+2],mem_d[mem_d_adr[6:0]+
PC = PC + 4 ;
$display("PC : %d, RD_address: %d,RD_data: %d", PC, IR[11:7], $signed(reg_file [IR[11:7]]));
end

32'bxxx_xxxx_xxxxx_xxxxx_100_xxxxx_0000011: //LBU
begin

imm = {{20{IR[31]}} , IR [31:20]};
mem_d_adr = $signed(reg_file[IR[19:15]]) + $signed(imm);
reg_file [IR[11:7]] ={{24{1'b0}},mem_d[mem_d_adr[6:0]]};
PC = PC + 4 ;
$display("PC : %d, RD_address: %d,RD_data: %d", PC, IR[11:7], $signed(reg_file [IR[11:7]]));
end

32'bxxx_xxxx_xxxxx_xxxxx_101_xxxxx_0000011: //LHU
begin

imm = {{20{IR[31]}} , IR [31:20]};
mem_d_adr = $signed(reg_file[IR[19:15]]) + $signed(imm);
reg_file [IR[11:7]] = {{16{1'b0}},mem_d[mem_d_adr[6:0]+1],mem_d[mem_d_adr[6:0]]};
PC = PC + 4 ;
$display("PC : %d, RD_address: %d,RD_data: %d", PC, IR[11:7], $signed(reg_file [IR[11:7]]));
end


/////////////////////////////////////////////////////////S FORMAT INSTRUCTION////////////////////


32'bxxx_xxxx_xxxxx_xxxxx_000_xxxxx_0100011:begin     //SB
imm = {{20{IR[31]}}, IR[31:25], IR[11:7]};
```



```
32'bxxx_xxxx_xxxxx_xxxxx_001_xxxxx_0100011:begin      //SH
imm = {{20{IR[31]}}, IR[31:25], IR[11:7]};
mem_d_adr = $signed(reg_file[IR[19:15]]) + $signed(imm);
rs2_value = reg_file[IR[24:20]];
{mem_d[mem_d_adr[6:0]+1],mem_d[mem_d_adr[6:0]]} = rs2_value[15:0];
PC = PC + 4;
end

32'bxxx_xxxx_xxxxx_xxxxx_010_xxxxx_0100011:begin     //SW
imm = {{20{IR[31]}}, IR[31:25], IR[11:7]};
mem_d_adr = $signed(reg_file[IR[19:15]]) + $signed(imm);
rs2_value = reg_file[IR[24:20]];
{mem_d[mem_d_adr[6:0]+3],mem_d[mem_d_adr[6:0]+2],mem_d[mem_d_adr[6:0]+1],mem_d[mem_d_adr[6:0]]} = rs2_value;
PC = PC + 4;
end


/////////////////////////////////////////////////B Format instruction/////////////////////////////////////////////////


32'bxxx_xxxx_xxxxx_xxxxx_000_xxxxx_1100011:begin      //BEQ

imm = {{20{IR[31]}},IR[31],IR[7], IR[30:25], IR[11:8],{1'b0}};
PC = PC + ($signed(IR[19:15])==$signed(IR[24:20])?imm:4);
$display("PC : %d", PC);

end

32'bxxx_xxxx_xxxxx_xxxxx_001_xxxxx_1100011:  begin    //BNE
imm = {{20{IR[31]}},IR[31],IR[7], IR[30:25], IR[11:8],{1'b0}};
PC = PC + ($signed(IR[19:15])!=$signed(IR[24:20])?imm:4);
$display("PC : %d", PC);
end


32'bxxx xxxx xxxxx xxxxx 100 xxxxx 1100011: begin     //BLT

32'bxxx_xxxx_xxxxx_xxxxx_101_xxxxx_1100011:begin      //BGE
imm = {{20{IR[31]}},IR[31],IR[7], IR[30:25], IR[11:8],{1'b0}};
PC = PC + ($signed(IR[19:15])>=$signed(IR[24:20])?imm:4);
$display("PC : %d", PC);
end

32'bxxx_xxxx_xxxxx_xxxxx_110_xxxxx_1100011: begin     //BLTU
imm = {{20{IR[31]}},IR[31],IR[7], IR[30:25], IR[11:8],{1'b0}};
PC = PC + (IR[19:15]<IR[24:20]?imm:4);
$display("PC : %d", PC);
end


32'bxxx_xxxx_xxxxx_xxxxx_111_xxxxx_1100011:begin      //BGEU
imm = {{20{IR[31]}},IR[31],IR[7], IR[30:25], IR[11:8],{1'b0}};
PC = PC + (IR[19:15]>=IR[24:20]?imm:4);
$display("PC : %d", PC);
end

/////////////////////////////////////////////////U(LUI) Format/////////////////////////////////////////////////
32'bxxx_xxxx_xxxxx_xxxxx_xxx_xxxxx_0110111: begin //lui
imm = {IR[31:12], {12{1'b0}}};
reg_file [IR[11:7]] = imm;
PC = $signed(PC) + 4;
$display("PC : %d, RD_address: %d,RD_data: %d", PC,IR[11:7], $signed(reg_file [IR[11:7]]));
end

/////////////////////////////////////////////////U(AUIPC) Format /////////////////////////////////////////////////
32'bxxx_xxxx_xxxxx_xxxxx_xxx_xxxxx_0010111: begin    //AUIPC
imm = {IR[31:12], {12{1'b0}}};
reg_file [IR[11:7]] = $signed(PC) + $signed(imm);
PC = $signed(PC) + 4;
$display("PC : %d, RD_address: %d,RD_data: %d", PC,IR[11:7], $signed(reg_file [IR[11:7]]));
```



```
//////////////////////////////////////////////////////JAL Format /////////////////////////////
32'bxxx_xxxx_xxxxx_xxxxx_xxx_xxxxx_1101111: begin     //JAL
imm = {{11{IR[31]}},IR[19:12], IR[20], IR[30:21],{1'b0}};
reg_file [IR[11:7]] = PC + 4;
PC = $signed(PC) + $signed(imm);
$display("PC : %d, RD_address: %d,RD_data: %d", PC,IR[11:7], $signed(reg_file [IR[11:7]]));
end

//////////////////////////////////////////////////////JALR Format Detection//////////////////
32'bxxx_xxxx_xxxxx_xxxxx_000_xxxxx_1100111:begin      //JALR
imm = {{20{IR[31]}}, IR[31:20]};
reg_file [IR[11:7]] = $signed(PC) + 4;
PC = ($signed(IR[19:15])+$signed(imm))&(32'hFFFFE);
$display("PC : %d, RD_address: %d,RD_data: %d", PC,IR[11:7], $signed(reg_file [IR[11:7]]));

end
endcase

end
endmodule
```

11.2 Testbench code:

```
module Func_ver_tb();

reg [31:0]IR ;

 Func_verify dut(IR);

    initial begin
    IR = 32'b000000000001_00001_000_00001_0110011; #5
    IR = 32'b010000000100_00011_000_00010_0110011; #5
    IR = 32'b000000000001_00010_000_00101_0110011;

    end

endmodule
```

11.3 Result:

In the testbench we ran three instructions namely - add x1,x2,x2 sub x2,x3,x4 add x5,x1,x2

The register file was initialized such that reg[i] = i. Data memory was initialized such that that data_mem[i] = i And pc was reset to 0.

The first instruction should give 2 The second should give -1 and the third should give 1

And we got the same result.

```
PC :           4, RD_address:  1,RD_data:          2
PC :           8, RD_address:  2,RD_data:         -1
PC :          12, RD_address:  5,RD_data:          1
```

Bibliography

[1] Abdelrahman Adel et al. "Implementation and Functional Verification of RISC-V Core for Secure IoT Applications". In: *2021 International Conference on Microelectronics (ICM)*. 2021, pp. 254–257. DOI: 10.1109/ICM52667.2021.9664926.

[2] Angel Barriga. "RISC-V processors design: a methodology for cores development". In: *2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS)*. 2020, pp. 1–6. DOI: 10.1109/DCIS51330.2020.9268639.

[3] Quanxiu Chen et al. "A Single Event Effect Simulation Method for RISC-V Processor". In: *2021 IEEE 15th International Conference on Anti-counterfeiting, Security, and Identification (ASID)*. 2021, pp. 106–110. DOI: 10.1109/ASID52932.2021.9651696.

[4] Sebastian Cieslak et al. "Retargeting the MIPS-II CPU Core to the RISC-V Architecture". In: *2019 MIXDES - 26th International Conference "Mixed Design of Integrated Circuits and Systems"*. 2019, pp. 261–264. DOI: 10.23919/MIXDES.2019.8787018.

[5] D. Collins. "Microprocessor Design and Application". In: *Computer* 8.10 (1975), pp. 20–21. ISSN: 1558-0814. DOI: 10.1109/C-M.1975.218775.

[6] Enfang Cui, Tianzheng Li, and Qian Wei. "RISC-V Instruction Set Architecture Extensions: A Survey". In: *IEEE Access* 11 (2023), pp. 24696–24711. DOI: 10.1109/ACCESS.2023.3246491.

[7] V. Rama Devi and Jadapalli Sreedhar. "Implementing Bit Manipulation Instructions on Architecture of RISC-V Processor". In: *2023 Global Conference on Information Technologies and Communications (GCITC)*. 2023, pp. 1–6. DOI: 10.1109/GCITC60406.2023.10426045.

[8] Wenqiang Gong, Fang Zhou, and Fen Ge. "A Multi-mode Convolution Coprocessor Based on RISC-V Instruction Set Architecture". In: *2023 IEEE 15th International Conference on ASIC (ASICON)*. 2023, pp. 1–5. DOI: 10.1109/ASICON58565.2023.10396531.

[9] Sukrat Gupta et al. "SHAKTI-F: A Fault Tolerant Microprocessor Architecture". In: *2015 IEEE 24th Asian Test Symposium (ATS)*. 2015, pp. 163–168. DOI: 10.1109/ATS.2015.35.

[10] Bowen Hu, Yun Chen, and Xiaoyang Zeng. "An Agile Instruction Set Extension Method Based on the RISC-V Processor". In: *2021 IEEE 4th International Conference on Electronics Technology (ICET)*. 2021, pp. 342–346. DOI: 10.1109/ICET51757.2021.9450911.

[11] Vineet Jain, Abhishek Sharma, and Eduardo Augusto Bezerra. "Implementation and Extension of Bit Manipulation Instruction on RISC-V Architecture using FPGA". In: *2020 IEEE 9th International Conference on Communication Systems and Network Technologies (CSNT)*. 2020, pp. 167–172. DOI: 10.1109/CSNT48778.2020.9115759.

[12] Hyeonguk Jang et al. "Developing a Multicore Platform Utilizing Open RISC-V Cores". In: *IEEE Access* 9 (2021), pp. 120010–120023. DOI: 10.1109/ACCESS.2021.3108475.

[13] Gopal Kanase and Nithin M. "ASIC Design of a 32-bit Low Power RISC-V based System Core for Medical Applications". In: *2021 6th International Conference on Communication and Electronics Systems (ICCES)*. 2021, pp. 1–5. DOI: 10.1109/ICCES51350.2021.9489067.

[14] Jin-Yang Lai et al. "Implement 32-bit RISC-V Architecture Processor using Verilog HDL". In: *2021 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*. 2021, pp. 1–2. DOI: 10.1109/ISPACS51563.2021.9651130.

[15] Zhenhao Li, Wei Hu, and Shuang Chen. "Design and Implementation of CNN Custom Processor Based on RISC-V Architecture". In: *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2019, pp. 1945–1950. DOI: 10.1109/HPCC/SmartCity/DSS.2019.00268.

[16] Binjie Mao et al. "A CLIC Extension Based Fast Interrupt System for Embedded RISC-V Processors". In: *2021 6th International Conference on Integrated Circuits and Microsystems (ICICM)*. 2021, pp. 109–113. DOI: 10.1109/ICICM54364.2021.9660345.

[17] Tyler McGrew, Eric Schonauer, and Peter Jamieson. "Framework and Tools for Undergraduates Designing RISC-V Processors on an FPGA in Computer Architecture Education". In: *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*. 2019, pp. 778–781. DOI: 10.1109/CSCI49370.2019.00148.

[18] "CHAPTER 5 - Error Coding Techniques". In: *Architecture Design for Soft Errors*. Ed. by Shubu Mukherjee. Burlington: Morgan Kaufmann, 2008, pp. 161–206. ISBN: 978-0-12-369529-1. DOI: https://doi.org/10.1016/B978-012369529-1.50007-0. URL: https://www.sciencedirect.com/science/article/pii/B9780123695291500070.

[19] Geraldine Shirley Nicholas, Yutian Gui, and Fareena Saqib. "A Survey and Analysis on SoC Platform Security in ARM, Intel and RISC-V Architecture". In: *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2020, pp. 718–721. DOI: 10.1109/MWSCAS48704.2020.9184573.

[20] Arul P et al. "Implementation of RISC-V Instruction Set Architecture for edge IoT computing platform". In: *2024 Fourth International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT)*. 2024, pp. 1–6. DOI: 10.1109/ICAECT60202.2024.10468869.

[21] Saman Payvar et al. “Instruction Extension of a RISC-V Processor Modeled with IP-XACT”. In: *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*. 2019, pp. 1–5. DOI: `10.1109/NORCHIP.2019.8906975`.

[22] Aneesh Raveendran et al. “A RISC-V instruction set processor-micro-architecture design and analysis”. In: *2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA)*. 2016, pp. 1–7. DOI: `10.1109/VLSI-SATA.2016.7593047`.

[23] Jonathan Saussereau et al. “AsteRISC: A Size-Optimized RISC-V Core for Design Space Exploration”. In: *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2023, pp. 1–5. DOI: `10.1109/ISCAS46773.2023.10181330`.

[24] Ronaldo Serrano et al. “A Low-Power Low-Area SoC based in RISC-V Processor for IoT Applications”. In: *2021 18th International SoC Design Conference (ISOCC)*. 2021, pp. 375–376. DOI: `10.1109/ISOCC53507.2021.9613880`.

[25] Riku Takayama and Jubee Tada. “An Implementation of a Pattern Matching Accelerator on a RISC-V Processor”. In: *2022 Tenth International Symposium on Computing and Networking Workshops (CANDARW)*. 2022, pp. 273–275. DOI: `10.1109/CANDARW57323.2022.00059`.

[26] Hui Wang, Yuyuan Du, and Xiangcheng Mu. “Research on In-System Programming IP of RISC-V Processor”. In: *2022 2nd International Conference on Algorithms, High Performance Computing and Artificial Intelligence (AHPCAI)*. 2022, pp. 264–269. DOI: `10.1109/AHPCAI57455.2022.10087608`.

[27] Wenzhu Wang et al. “The Design and Building of openKylin on RISC-V Architecture”. In: *2022 15th International Conference on Advanced Computer Theory and Engineering (ICACTE)*. 2022, pp. 88–91. DOI: `10.1109/ICACTE55855.2022.9943636`.

[28] Kangning Yue and Yubang Shen. “An overview of disruptive technologies for aquaculture”. In: *Aquaculture and Fisheries* 7.2 (2022). SI : Emerging and disruptive technologies for aquaculture, pp. 111–120. ISSN: 2468-550X. DOI: `https://doi.org/10.1016/j.aaf.2021.04.009`. URL: `https://www.sciencedirect.com/science/article/pii/S2468550X21000617`.

[29] Tian Zheng, Gang Cai, and Zhihong Huang. “A Soft RISC-V Processor IP with High-performance and Low-resource consumption for FPGA”. In: *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2022, pp. 2538–2541. DOI: `10.1109/ISCAS48785.2022.9937742`.