Indian Institute of Technology Bombay

6 Stage Pipelined Processor
EE 739: Processor Design Course Project - 1

Anuranan Das - 18D070037
Aditya Rajeev Harakare - 18D100001
Ruchir Chheda - 18D100016

Department of Electrical Engineering

Contents

1 Problem Statement 3

2 Design Description 3

3 Datapath Implementation 5

3.1 Multiplexers . . . 5
3.2 ALU . . . 5
3.3 Memory . . . 6
3.4 Register File . . . 6
3.5 Intermediate Pipeline Registers . . . 6
3.6 CCR . . . 6
3.7 Forwarding Unit . . . 6
3.8 Control Decoder . . . 6
3.9 LA/LM/SA/SM Controller . . . 6
3.10 Branch and Jump Controller . . . 7
3.11 Branch Predictor . . . 7

4 Pipeline Stages 7

4.1 Instruction Fetch . . . 7
4.2 Instruction Decode . . . 8
4.3 Register Read . . . 8
4.4 Execute . . . 8
4.5 Memory Access . . . 8
4.6 Write Back . . . 8

5 Intermediate Pipeline Registers 8

5.1 IF/ID Register . . . 8
5.2 ID/RR Register . . . 9
5.3 RR/EX Register . . . 9
5.4 EX/MEM Register . . . 9
5.5 MEM/WB Register . . . 9

6 Hazards 10

6.1 Load Hazards . . . 10
6.2 LA LM Hazards . . . 10
6.3 Branch Hazards . . . 10

7 Simulations and Results 10

7.1 Running the Simualtions . . . 10

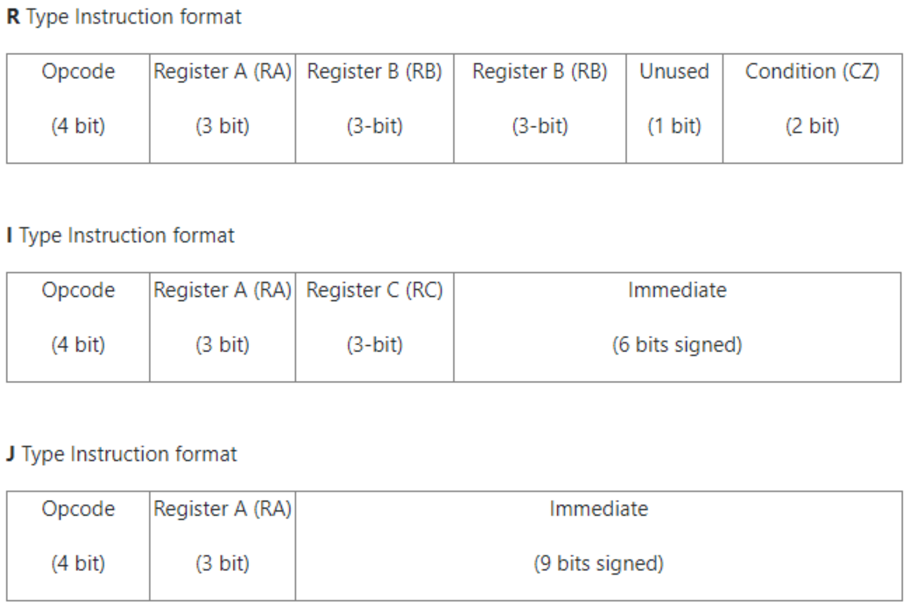
8 Conclusion 13

1 Problem Statement

To design a 6 stage pipelined processor, IITB-RISC-22, whose instruction set architecture is provided. IITB-RISCis a 16-bit very simple computer developed for the teaching that is based on the Little Computer Architecture. The IITB-RISC-22 is a 16-bit computer system with 8 registers. It should follow the standard 6 stage pipelines (Instruction fetch, instruction decode, register read, execute, memory access, and write back). The architecture should be optimized for performance, i.e., should include hazard mitigation techniques. Hence, it should have forwarding and branch prediction technique.

2 Design Description

IITB-RISC is a 16-bit processor based on the Little Computer Architecture. IITB RISC has 8 general-purpose registers (R0 to R7). Register R7 is always stores Program Counter. This architecture uses condition code register which has two flags Carry flag (C) and Zero flag (Z). The architecture is optimized for performance and thereby includes branch prediction and hazard mitigation techniques. There are three machine-code instruction formats (R, I, and J type) as illustrated in the figure below.



R Type Instruction format

Opcode	Register A (RA)	Register B (RB)	Register B (RB)	Unused	Condition (CZ)
(4 bit)	(3 bit)	(3-bit)	(3-bit)	(1 bit)	(2 bit)

I Type Instruction format

Opcode	Register A (RA)	Register C (RC)	Immediate
(4 bit)	(3 bit)	(3-bit)	(6 bits signed)

J Type Instruction format

Opcode	Register A (RA)	Immediate
(4 bit)	(3 bit)	(9 bits signed)

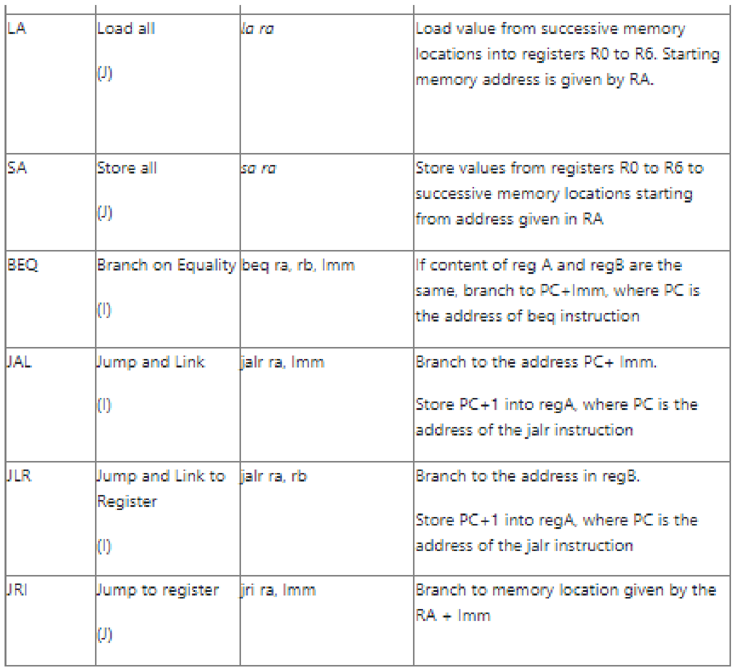
Figure 1: Instruction Formats

We have implemented a 6 stage pipelined design including the following stages:

1. Instruction Fetch Stage (IF Stage)
2. Instruction Decode Stage (ID Stage)
3. Register Read Stage (RR Stage)
4. Execution Stage (EX Stage)
5. Memory Stage (MEM Stage)
6. Write Back Stage (WB Stage)

The design supports the following instruction set:

Mnemonic	Name & Format	Assembly	Action
ADD	ADD (R)	*add rc, ra, rb*	Add content of regB to regA and store result in regC. *It modifies C and Z flags*
ADC	Add if carry set (R)	*adc rc, ra, rb*	Add content of regB to regA and store result in regC, if carry flaf is set. *It modifies C & Z flags*
ADZ	Add if zero set (R)	*adz rc, ra, rb*	Add content of regB to regA and store result in regC, if zero flag is set. *It modifies C & Z flags*
ADL	Add with one bit left shift of RB (R)	*Adl rc,ra,rb*	Add content of regB (after one bit left shift) to regA and store result in regC *It modifies C & Z flags*
ADI	Add immediate (I)	*adi rb, ra, imm6*	Add content of regA with Imm (sign extended) and store result in regB. *It modifies C and Z flags*
NDU	Nand (R)	*ndu rc, ra, rb*	NAND the content of regB to regA and store result in regC. *It modifies Z flag*
NDC	Nand if carry set (R)	*ndc rc, ra, rb*	NAND the content of regB to regA and store result in regC if carry flag is set. *It modifies Z flag*
NDZ	Nand if zero set (R)	*ndc rc, ra, rb*	NAND the content of regB to regA and store result in regC if zero flag is set. *It modifies Z flag*
LHI	Load higher immediate (J)	*lhi ra, Imm*	Place 9 bits immediate into most significant 9 bits of register A (RA) and lower 7 bits are assigned to zero.
LW	Load (I)	*lw ra, rb, Imm*	Load value from memory into reg A. Memory address is formed by adding immediate 6 bits with content of red B. *It modifies zero flag.*
SW	Store (I)	*sw ra, rb, Imm*	Store value from reg A into memory. Memory address is formed by adding immediate 6 bits with content of red B.
LM	Load multiple (J)	*lw ra, Imm*	Load multiple registers whose address is given in the immediate field (one bit per register, R0 to R7) in order from left to right, i.e, registers from R0 to R7 if corresponding bit is set. Memory address is given in reg A. Registers are loaded from consecutive addresses.
SM	Store multiple (J)	sm, ra, Imm	Store multiple registers whose address is given in the immediate field (one bit per register, R0 to R7) in order from left to right, i.e, registers from R0 to R7 if corresponding bit is set. Memory address is given in reg A. Registers are stored to consecutive addresses.



LA	Load all (J)	*la ra*	Load value from successive memory locations into registers R0 to R6. Starting memory address is given by RA.
SA	Store all (J)	*sa ra*	Store values from registers R0 to R6 to successive memory locations starting from address given in RA
BEQ	Branch on Equality (I)	beq ra, rb, Imm	If content of reg A and regB are the same, branch to PC+Imm, where PC is the address of beq instruction
JAL	Jump and Link (I)	jalr ra, Imm	Branch to the address PC+ Imm. Store PC+1 into regA, where PC is the address of the jalr instruction
JLR	Jump and Link to Register (I)	jalr ra, rb	Branch to the address in regB. Store PC+1 into regA, where PC is the address of the jalr instruction
JRI	Jump to register (J)	jri ra, Imm	Branch to memory location given by the RA + Imm

Figure 2: Instruction Description

3 Datapath Implementation

The design has been implemented in Verilog. Following are the main components of the datapath:

3.1 Multiplexers

Multiplexers are required for the steering logic, whose control signals will be given by the main decoder (or from any other auxiliary decoders for branch / branch predictor / LA, LM Controller). Mainly there are 2 input and 4 input Multiplexers.
Example of a multiplexer: In the EX stage, forwarded operand1 from register file is directly connected to ALU, the second input to ALU comes from 3 possible combinations, either the forwarded operand2 from register file, or the forwarded operand2 from register file that is left shifted by 1 (for ADL instruction), or the 6 bit immediate sign extended to 16 bits. Similarly several multiplexers decides the steering logic of the processor.
The 2 to 1 mux and 4 to 1 mux is coded in the file 'mux.v'

3.2 ALU

The ALU performs ADD, NAND or NOP operations based on the control signals from the ALU Controller. The main decoder decides the operation the ALU should perform. In case of ADC, ADZ and other conditional instructions to determine which operation should be performed based on C and Z flags, reevaluation is done at EX stage. It also determines whether C and Z should be modified or not. In case of conditional instructions, if the ALU controller sees that the condition has not been met, then the ALU does not perform any operation, also the writeEnable of the Register File (RO to R7) and the data memory write enable would be defined LOW(0) to ensure that the system state is not modified.

3.3 Memory

The memory address points to two bytes in the memory and the size of the memory is 4096 bytes(4 KBytes). Both data and instruction memory are of size 4KB. All the unused part of the instruction memory are filled with NOP instructions.

3.4 Register File

The register file consists of 8 registers (RO to R7) that gets updated on the positive edge of the clock, if an active high write enable is asserted. Register R7 always stores the program counter. R7 cannot be modified by other instructions, but it can be read.
The SA/SM instruction needs access to all the register values. Hence, a seperate 112 bit vector output that gives out all the values of registers is given to LA/LM/SA/SM controller.

3.5 Intermediate Pipeline Registers

There will be 5 pipelined registers namely IF/ID, ID/RR, RR/EX, EX/MEM, and MEM/WB between each stage of the pipeline each having an active high write enable. Each corresponding pipeline registers stores the necessary data required for the upcoming stages.

3.6 CCR

We have 3 additional registers for storing the following:

1. Program Counter
2. Carry Flag
3. Zero Flag

3.7 Forwarding Unit

Data Forwarding is implemented at the beginning of execute stage. A separate control for Forwarding unit compares the source operand address with the destination register address in the stages ahead. If a match is present it forwards the most recent data.

3.8 Control Decoder

In the Instruction Decode stage, we have the main control decoder. It provides the control signals for all the multiplexers to drive the steering logic. These control signals drive the writeEnable lines in the register File and memory.

3.9 LA/LM/SA/SM Controller

The memory has only one read/write port. Hence. for the instructions LA, LM, SA, SM, a separate controller is designed which will stall the main pipeline every time one of these instructions comes at the memory stage. The pipeline will remain stalled until the complete memory operation is executed for these instructions.

3.10 Branch and Jump Controller

For the instructions JAL, BEQ, JLR, JRI, a separate controller is designed in which the next address is calculated. The output of this is the PC_Select line which drives the MUX to choose between PC+1 and corresponding address for JAL, BEQ, JLR and JRI.

3.11 Branch Predictor

We have designed a 1 bit predictor which has a depth of 8 entries. Each entry has 33 bits, 16 bit PC, 16 bit branch target address and 1 history bit.
Each time a new Jump or Branch instruction is encountered, the History Table is updated. It also dynamically updates the History bit based on branch taken/not taken.
If the present program counter matches with the entry in the Branch History Table, then the next Program counter is fetched from the target address from the LUT using a MUX. The control for the MUX will be logical AND operation of match and history bits. The match bit becomes 1 when an entry corresponding to the present PC is found in the LUT.

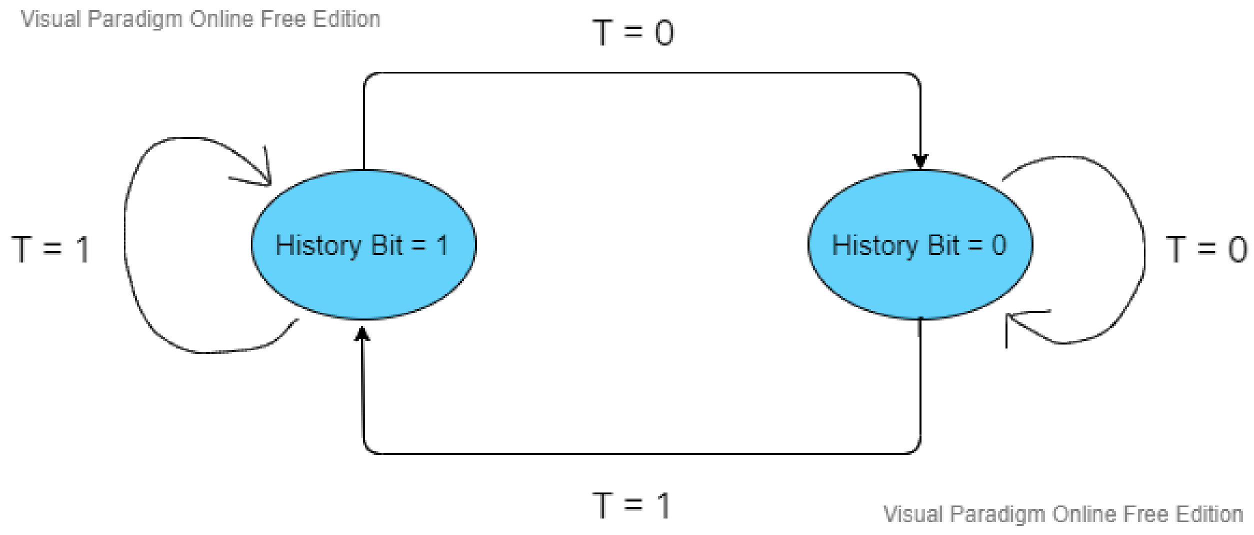

Visual Paradigm Online Free Edition
T = 0
T = 1
History Bit = 1
History Bit = 0
T = 0
T = 1
Visual Paradigm Online Free Edition

Figure 3: Branch Predictor State Diagram

4 Pipeline Stages

4.1 Instruction Fetch

This is the first stage in the pipeline, that fetches the two byte instruction from the memory. Instruction memory is a simple 16-bit addressable memory with 1 data read port. For fetching, we have the data for the instruction address in the register R7. The PC gets updated appropriately and fetches the instruction as the data output value from instruction memory. We have a PC selector MUX which selects between either PC+1 or the jump/branch target addresses calculated by the ID/EX stages of jal, jlr, jri and beq. The fetched instruction pass through a NOP MUX, whose control is determined by the branch controller (whether the current instruction is to be flushed or not, depending on the decision from branch/jump controller). If we need to stall, we don't forward the instruction fetched onto the next stage.

4.2 Instruction Decode

This is the second stage of the pipeline. It decides the operand address and the target destination address. This stage generates all the control signals based on the input instruction. In case of branch instruction and incorrect speculation, the pipeline needs to be flushed. This is done by the NOP MUX in this stage.

4.3 Register Read

The IITB RISC design has 8 registers from R0 to R7 in the register file. For instructions involving register read, 2 data read ports are provided to obtain the values in corresponding registers. Register file also has a write port used during the writeback stage.

4.4 Execute

The third stage in the pipeline, it consists of the forwarding controller, the ALU and ALU Controller. The data of the operands are read from the intermediate pipeline register and fed in the ALU. Both the operands are 16 bit. All the instructions functionality can be broken into two operations namely AND and NAND. Hence, the ALU supports these two operations. The 'ALU.v' and 'aluCtrl.v' file have all these operations and control logic coded respectively. There are 2 flags present, namely zero flag and carry flag. he operands are extended by 1 bit before providing the the ALU for execution. Based on the conditional instructions, memory and register write signals are redefined. Branch and jump instructions are resolved in this stage as the forwarded values are available here. If the speculated BEQ instruction is correct then there would not be any penalty, however there would be a 2 cycle penalty in case the speculation is incorrect.

4.5 Memory Access

The fourth stage in the pipeline, it consists of a 4KB memory, with an active high write enable. It performs the task of reading or writing from data memory based upon the instruction provided. We have a 16-bit addressable memory with 1 port which is used for both read and write operations.
This stage also has a separate LA LM SA SM controller. The main functionalities of the controller is to stall the pipeline whenever an LA or LM or SA or SM instruction reaches the memory stage. Until all the memory accesses are complete, the pipeline is stalled. The controller also computes the consecutive memory address that needs to be accessed. This is necessary because the memory in practice cannot have multiple ports for read or write.

4.6 Write Back

The final stage in the pipeline, this updates the register file if the corresponding write enable is high.

5 Intermediate Pipeline Registers

Following signals are present in the respective pipeline registers:

5.1 IF/ID Register

1. Current PC

2. PC+1 (In case prediction fails)
3. Instruction (Current Instruction being executed)
4. Speculation (Whether the branch has been speculated)

5.2 ID/RR Register

1. Current PC
2. PC+1 (In case prediction fails)
3. Instruction (Current Instruction being executed)
4. Speculation (Whether the branch has been speculated)
5. Control Signals generated

5.3 RR/EX Register

1. Current PC
2. PC+1 (In case prediction fails)
3. Instruction (Current Instruction being executed)
4. Speculation (Whether the branch has been speculated)
5. Control Signals generated
6. Source Operand Address (for forwarding logic at EX)
7. Destination address and data from WB stage for forwarding 3-cycle apart dependency. WB updates register file at the positive edge of CLK, hence the data would only be available at the next clock edge . To accomodate the forwarding for data at WB, this entry is made.

5.4 EX/MEM Register

1. Control Signals
2. Destination Address of register and ALU result
3. Register Data to be written to memory in case of SW, SA, SM
4. Memory Access Address.

5.5 MEM/WB Register

1. Register File write enable
2. Destination address of the Register File.
3. Data to be updated at the Register File.

6 Hazards

To minimize stalling and improve the performance, we have taken care of different hazards using data Forwarding and Dynamic Branch Prediction using a 1 bit History. Using data forwarding, we take care of all the 3 cases namely: immediate dependencies, 2 cycles apart dependencies and 3 cycles apart dependencies.
If the branch is incorrectly predicted, the pipeline needs to be stalled. The pipeline is further flushed by inserting NOP instructions wherever needed. Following hazards are handled by our implementation.

6.1 Load Hazards

Example of a load hazard:
I1: lw ra, mem_addr
I2: add rc, ra, rb
Here data from memory is read in I1 and is stored in ra. And this same register is accessed in the immediate instruction. When I1 reaches the EX stage, I2 will reach the ID stage. ra will only be updated at the end of the MEM stage and hence 1 cycle stall is needed between these instructions.

6.2 LA LM Hazards

Example of a LA LM hazard:
I1: la ra, mem_addr
I2: add rc, ra, rb
When instruction I1 reaches MEM stage, the pipeline is stalled so that the memory can be accessed only through one read port in each cycle. LA requires to access memory for 7 cycles. I2 can't stay in the EX stage as it needs the updated values after la is executed completely. Hence I2 will be stalled in the ID stage and only after MEM stage of I1 is complete for all cycles, I2 can proceed.

6.3 Branch Hazards

Branch instructions are speculated with predictions, in case if the prediction is incorrect, then the instructions in IF and ID are replaced with NOP. Prediction is only applicable for JAL and BEQ instructions. For JLR and JRI, the branch is based on the target address in the register. Hence these instructions need a 2 cycle penalty since forwarding is implemented at the EX stage.

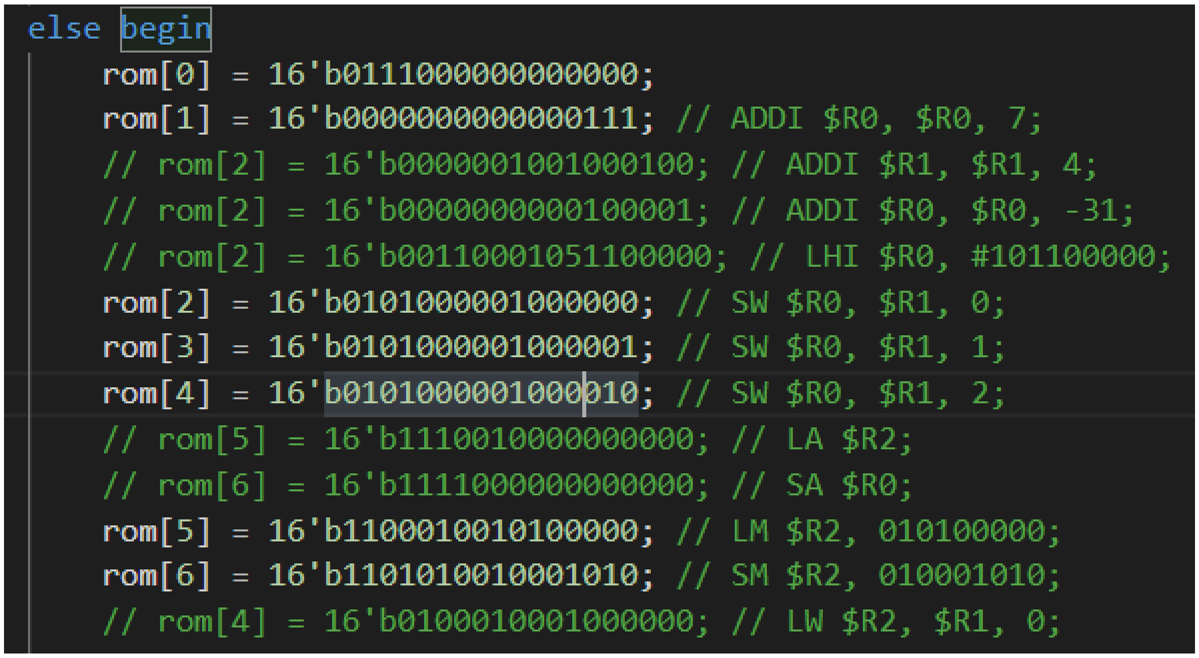
7 Simulations and Results

7.1 Running the Simualtions

Open a terminal in the Codes folder, use the following commands to start the processor:

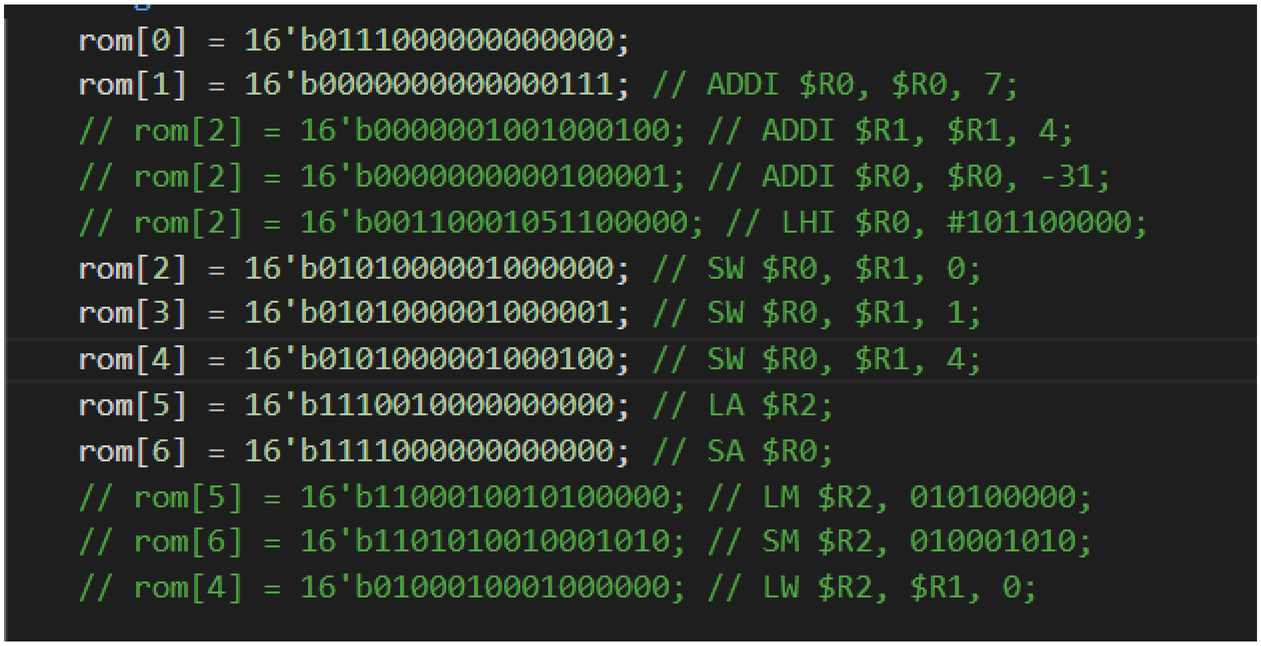
```
iverilog -o iitb_risc topModule.v alu.v aluCtrl.v branchHist.v branchJumpCtrl.v
ctrlUnit.v dataMem.v frwdingCntrl.v instrMem.v loadStoreMultiple.v mux.v
registerBank.v Stage12_IF2ID.v Stage23_ID2RR.v Stage34_RR2EX.v Stage45_EX2MM.v
Stage56_MM2WB.v topModuleTB.v
vvp iitb\_risc
tkwave dump.vcd
```

In this report we have presented the execution of the following two example codes as shown in Figure 4 and Figure 5 respectively.



```verilog
else begin
    rom[0] = 16'b0111000000000000;
    rom[1] = 16'b0000000000000111; // ADDI $R0, $R0, 7;
    // rom[2] = 16'b0000001001000100; // ADDI $R1, $R1, 4;
    // rom[2] = 16'b0000000000100001; // ADDI $R0, $R0, -31;
    // rom[2] = 16'b00110001051100000; // LHI $R0, #101100000;
    rom[2] = 16'b0101000001000000; // SW $R0, $R1, 0;
    rom[3] = 16'b0101000001000001; // SW $R0, $R1, 1;
    rom[4] = 16'b0101000001000010; // SW $R0, $R1, 2;
    // rom[5] = 16'b1110010000000000; // LA $R2;
    // rom[6] = 16'b1111000000000000; // SA $R0;
    rom[5] = 16'b1100010010100000; // LM $R2, 010100000;
    rom[6] = 16'b1101010010001010; // SM $R2, 010001010;
    // rom[4] = 16'b0100010001000000; // LW $R2, $R1, 0;
```

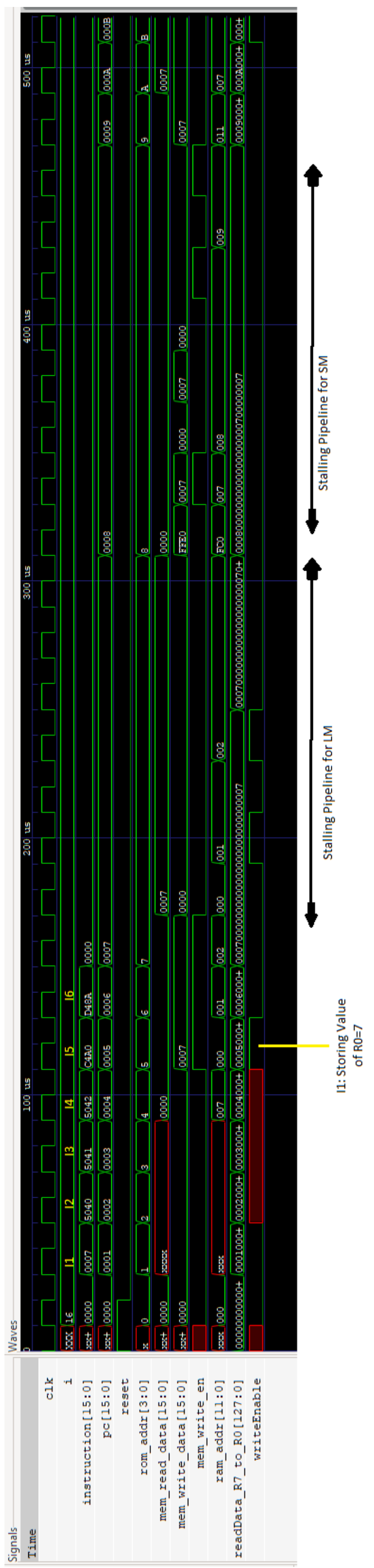
Figure 4: Instruction Memory - Example Code 1

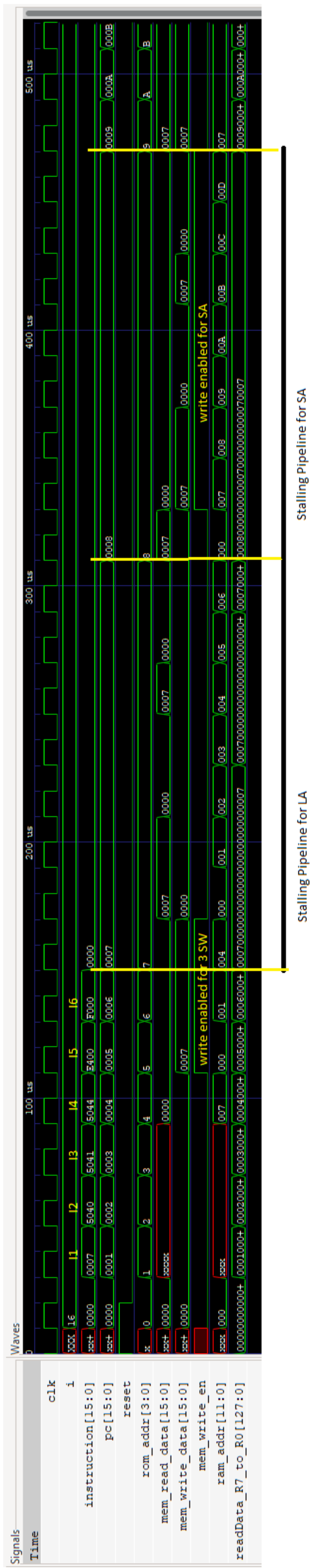


```verilog
rom[0] = 16'b0111000000000000;
rom[1] = 16'b0000000000000111; // ADDI $R0, $R0, 7;
// rom[2] = 16'b0000001001000100; // ADDI $R1, $R1, 4;
// rom[2] = 16'b0000000000100001; // ADDI $R0, $R0, -31;
// rom[2] = 16'b00110001051100000; // LHI $R0, #101100000;
rom[2] = 16'b0101000001000000; // SW $R0, $R1, 0;
rom[3] = 16'b0101000001000001; // SW $R0, $R1, 1;
rom[4] = 16'b0101000001000100; // SW $R0, $R1, 4;
rom[5] = 16'b1110010000000000; // LA $R2;
rom[6] = 16'b1111000000000000; // SA $R0;
// rom[5] = 16'b1100010010100000; // LM $R2, 010100000;
// rom[6] = 16'b1101010010001010; // SM $R2, 010001010;
// rom[4] = 16'b0100010001000000; // LW $R2, $R1, 0;
```

Figure 5: Instruction Memory - Example Code 2

The respective simulation results with all the important signals generated from GTKWave software are shown in the following Figures 6a and 6b.





(a) Simulation Results - Code 1

(b) Simulation Results - Code 2

Hence we can see from the simulation results that the processor is indeed giving the outputs as desired. Moreover, data forwarding, stalling of pipeline and other hazards are also verified. The processor was tested extensively with many corner cases.

8 Conclusion

We have implemented a 6 stage pipelined processor based on the RISC architecture. The IITB-RISC ISA was provided for this processor design. It supports 19 instructions and is optimized for performance using data forwarding and 1 bit branch prediction techniques. Other hazards are taken care of by stalling/inserting NOP instruction wherever required. We used Quartus for synthesizing the complete design and simulated all the instructions using GTKWave.