

Project Report

on

5 Stage Pipelined RISC Processor

Prepared by

Name	Roll Number
Gajulapalli Venkata Naga Dinesh Kumar	193079022
Perumalla Koteswara Rao	19307R020
Poluru Sai Krishna Praneeth	19307R022
Devulapally Naveen	193079037

1 Introduction

The project implements a 5 stage Pipelined Processor, IITB - RISC. It follows the standard 5 stage pipelines (Instruction fetch, instruction decode register read, execute, memory access, and write back).

It has 8 general purpose registers (R0 to R7). Register R7 always stores the Program Counter. It has 19 instructions and the architecture uses condition code registers which has two flags, Carryflag and Zero flag.

The architecture is optimized for performance, it has **Forwarding Mechanism** to limit the stalls due to RAW Hazards, also a **1 bit Branch Predictor** is implemented to speculate the branches and jumps.

2 ISA

The ISA as per the problem statement is shown below. It has 3 instruction formats (R, I and J)and a total of 19 instructions.

R Type Instruction format

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

I Type Instruction format

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

J Type Instruction format

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

Figure 1: Instruction Set

The datapath is conceived using Hardware flowchart and evolves according to the instructionrequirement. The implementation/ HFC for the instructions are shown below.

Instructions Encoding:

ADD:	00_01	RA	RB	RC	0	00
ADC:	00_01	RA	RB	RC	0	10
ADZ:	00_01	RA	RB	RC	0	01
ADL:	00_01	RA	RB	RC	0	11
ADI:	00_00	RA	RB	6 bit Immediate		
NDU:	00_10	RA	RB	RC	0	00
NDC:	00_10	RA	RB	RC	0	10
NDZ:	00_10	RA	RB	RC	0	01
LHI:	00_11	RA	9 bit Immediate			
LW:	01_00	RA	RB	6 bit Immediate		
SW:	01_01	RA	RB	6 bit Immediate		
LM:	11_00	RA	0 + 8 bits corresponding to Reg R0 to R7 (left to right)			
SM:	11_01	RA	0 + 8 bits corresponding to Reg R0 to R7 (left to right)			
LA:	11_10	RA	0_0000_0000			
SA:	11_11	RA	0_0000_0000			
BEQ:	10_00	RA	RB	6 bit Immediate		
JAL:	10_01	RA	9 bit Immediate offset			
JLR:	10_10	RA	RB	000_000		
JRI	10_11	RA	9 bit Immediate offset			

RA: Register A

RB: Register B

RC: Register C

Instruction Description

Mnemonic	Name & Format	Assembly	Action
ADD	AD D (R)	<i>add rc, ra, rb</i>	Add content of regB to regA and store result in regC. <i>It modifies C and Z flags</i>
ADC	Add if carry set(R)	<i>adc rc, ra, rb</i>	Add content of regB to regA and store result in regC, if carry flaf is set. <i>It modifies C & Z flags</i>
ADZ	Add if zero set(R)	<i>adz rc, ra, rb</i>	Add content of regB to regA and store result in regC, if zero flag is set. <i>It modifies C & Z flags</i>
ADL	Add with one bit left shift of RB (R)	<i>Adl rc,ra,rb</i>	Add content of regB (after one bit left shift)to regA and store result in regC <i>It modifies C & Z flags</i>
ADI	Add immediate(I)	<i>adi rb, ra, imm6</i>	Add content of regA with Imm (sign extended) and store result in regB. <i>It modifies C and Z flags</i>
NDU	Nan d(R)	<i>ndu rc, ra, rb</i>	NAND the content of regB to regA and storeresult in regC. <i>It modifies Z flag</i>
NDC	Nand if carry set(R)	<i>ndc rc, ra, rb</i>	NAND the content of regB to regA and storeresult in regC if carry flag is set. <i>It modifies Z flag</i>

NDZ	Nand if zero set(R)	<i>ndc rc, ra, rb</i>	NAND the content of regB to regA and store result in regC if zero flag is set. <i>It modifies Z flag</i>
LHI	Load higher immediate (J)	<i>lhi ra, Imm</i>	Place 9 bits immediate into most significant 9 bits of register A (RA) and lower 7 bits are assigned to zero.
LW	Load (I)	<i>lw ra, rb, Imm</i>	Load value from memory into reg A. Memory address is formed by adding immediate 6 bits with content of reg B. <i>It modifies zero flag.</i>
SW	Store (I)	<i>sw ra, rb, Imm</i>	Store value from reg A into memory. Memory address is formed by adding immediate 6 bits with content of reg B.
LM	Load multiple(J)	<i>lw ra, Imm</i>	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)	<i>sm, ra, Imm</i>	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)	<i>la ra</i>	Load value from successive memory locations into registers R0 to R6. Starting memory address is given by RA.
SA	Store all(J)	<i>sa ra</i>	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

3 Implementation

3.1 Hardware Flow Chart

ADD, ADC, ADZ, NDU, NDC, NDZ

```
PC -> IM , ALU1_A
+1 -> ALU1_B
ALU1_OUT -> PC
IM_D [11:9] -> RF_A1
IM_D [8:6] -> RF_A2
RF_D1 -> ALU2_A
RF_D2 -> ALU2_B
ALU2_OUT -> RF_D3
IM_D [5:3] -> RF_A3
```

ADL

```
PC -> IM , ALU1_A
+1 -> ALU1_B
ALU1_OUT -> PC
IM_D [11:9] -> RF_A1
IM_D [8:6] -> RF_A2
RF_D1 -> ALU2_A
RF_D2 -> LS1 -> ALU2_B
ALU2_OUT -> RF_D3
IM_D [5:3] -> RF_A3
```

ADI

```
PC -> IM , ALU1_A
+1 -> ALU1_B
ALU1_OUT -> PC
IM_D [11:9] -> RF_A1
RF_D1 -> ALU2_A
IM_D[5:0] -> SE10 -> ALU2_B
ALU2_OUT -> RF_D3
IM_D [8:6] -> RF_A3
```

LW

```
PC -> IM , ALU1_A
+1 -> ALU1_B
ALU1_OUT -> PC
IM_D [8:6] -> RF_A1
RF_D1 -> ALU2_A
IM_D[5:0] -> SE10 -> ALU2_B
ALU2_OUT ->
DATA_MEM_ADD
IM_D [11:9] -> RF_A3
DATA_MEM_DATA -> RF_D3
```

LHI

```
PC -> IM , ALU1_A
+1 -> ALU1_B
ALU1_OUT -> PC
IM_D [8:0] -> LS7 -> RF_D3
IM_D [11:9] -> RF_A3
```

SW

```
PC -> IM , ALU1_A
+1 -> ALU1_B
ALU1_OUT -> PC
IM_D [8:6] -> RF_A1
RF_D1 -> ALU2_A
IM_D[5:0] -> SE10 -> ALU2_B
ALU2_OUT ->
DATA_MEM_ADD
IM_D [11:9] -> RF_A2
RF_D2 -> DATA_MEM_IN
```

JAL (ALU3 - TO DO PC + 9 BIT IMM)

```
PC -> IM , ALU1_A, ALU3_A
+1 -> ALU1_B
ALU1_OUT -> RF_D3
IM_D [11:9] -> RF_A3
IM_D[8:0] -> SE7 -> ALU3_B
ALU3_OUT -> PC
```

JLR

```
PC -> IM , ALU1_A
+1 -> ALU1_B
IM_D [8:6] -> RF_A2
ALU1_OUT -> RF_D3
IM_D [11:9] -> RF_A3
RF_D2 -> PC
```

JRI

```
PC -> IM , ALU1_A
+1 -> ALU1_B
IM_D [11:9] -> RF_A1
RF_D1-> ALU3_A
IM_D[8:0] -> SE7 -> ALU3_B
ALU3_OUT -> PC
```

BEQ

```
PC -> IM , ALU1_A, ALU3_A
+1 -> ALU1_B
IM_D [11:9] -> RF_A1
IM_D [8:6] -> RF_A2
RF_D1-> COMPARATOR_A
RF_D2 -> COMPARATOR_B
IM_D[5:0] -> SE10 -> ALU3_B
if (zero)
    ALU3_OUT -> PC
else
    ALU1_OUT -> PC
```


LA

```

PC -> IM , ALU1_A
+1 -> ALU1_B
ALU1_OUT -> PC
IM_D [11:9] -> RF_A1

for(K=0; K <=6; K++) {
    RF_D1 -> ALU4_A
    K -> ALU4_B
    ALU4_OUT -> DATA_MEM_ADD
    K -> RF_A3
    DATA_MEM_DATA -> RF_D3
}

```

SA

```

PC -> IM , ALU1_A
+1 -> ALU1_B
ALU1_OUT -> PC
IM_D [11:9] -> RF_A1

for(K=0; K <=6; K++) {
    RF_D1 -> ALU4_A
    K -> ALU4_B
    ALU4_OUT -> DATA_MEM_ADD
    RF_R6-R0[(K+1)*16 -1 : K*16] -> DATA_MEM_IN
}

```

LM

```

PC -> IM , ALU1_A
+1 -> ALU1_B
ALU1_OUT -> PC
IM_D [11:9] -> RF_A1
L = 0
for(K=0; K <=6; K++) {
    RF_D1 -> ALU4_A
    if( IM_D [6:0] [6 - K] == 1){
        L -> ALU4_B
        ALU4_OUT -> DATA_MEM_ADD
        K -> RF_A3
        DATA_MEM_DATA -> RF_D3
        L = L + 1;
    }
}
}

```

SM

```

PC -> IM , ALU1_A
+1 -> ALU1_B
ALU1_OUT -> PC
IM_D [11:9] -> RF_A1
L = 0
for(K=0; K <=6; K++) {
    RF_D1 -> ALU4_A
    if( IM_D [6:0] [6 - K] == 1){
        L -> ALU4_B
        ALU4_OUT -> DATA_MEM_ADD
        RF_R6-R0[(K+1)*16 -1 : K*16] -> DATA_MEM_IN
        L = L + 1;
    }
}
}

```

3.2 Summary of Components in Datapath

The implementation has been done in Verilog.

Pipelined and CCR Registers

There will be 4 pipelined registers namely IF/ID, ID/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. Also it has 3 additional registers for Program Counter, Carry and Zero Flags.

Multiplexers

As shown in the datapath, 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. 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.

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.

ALU and ALU Controller

The ALU performs ADD, NAND or NOP operations based on the control signals from the ALU Controller. Although the main decoder decides the operation the ALU should perform, it is re-evaluated at EX stage in case of ADC, ADZ and other conditional instructions to determine whether the operation should be performed based on C and Z flags. 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 will be redefined to logic low (0) so that it does not modify the system state.

Forwarding Unit

A separate control for Forwarding unit is present at the execute stage. Forwarding is implemented at the beginning of execute stage. It compares the source operand address with the destination register address in the stages ahead (if it is supposed to update the register, which means writeEnable of the register file should also be high). If a match is present it forwards the most recent data.

Control Decoder

The main control decoder at the Decode Stage, it provides the control signals for all the multiplexers for the steering logic, the write enable signals for the memory and register file based on the decoded instruction.

Register File

The register file consists of 8 registers (R0 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. The register R7 cannot be modified by any other instructions, but it can be read. A separate hardwired entry is made of register R7 from the PC, so that the latest value of PC is updated on to R7 at every clock.

Since the SA/ SM instruction needs access to all the register values, a separate output that gives out all the values of registers in a 112 bit vector format is given to LA LM SA SM controller

LA LM SA SM Controller

A separate controller for LA, LM , SA , SM at Memory stage. Since memory can only have one read/write port, the controller will stall the pipeline every time an LA or LM or SA or SM instruction comes at memory stage. This will stall the pipeline until one of these 4 instructions finish at MEM stage.

Branch and Jump Controller

A separate controller which selects the next instruction address calculated from the ID or EX stage. (jal, beq, jlr, jri).

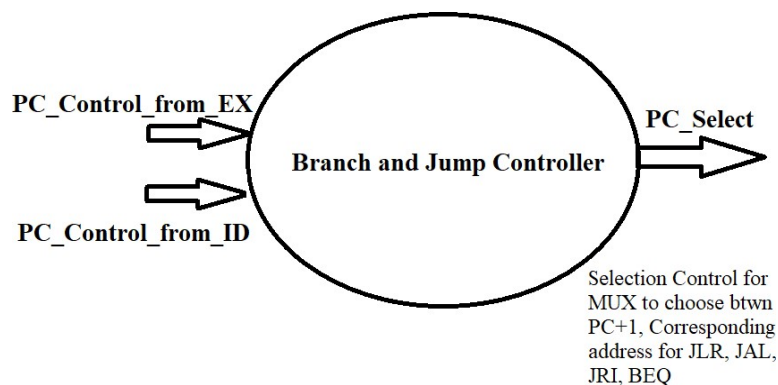


Figure 2: Branch Controller

Branch Predictor

A 1 bit predictor. The depth of the predictor is 8 entries. Each entry consists of 33 bits, a 16 bit PC, a 16 bit Branch Target address and a 1 bit History Bit. 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 bit, where match becomes high when an entry corresponding to the present PC is found in Look Up Table. Each time a new Jump or Branch instruction is seen, the History Table is updated. It also dynamically updates the History bit based on taken/not taken.

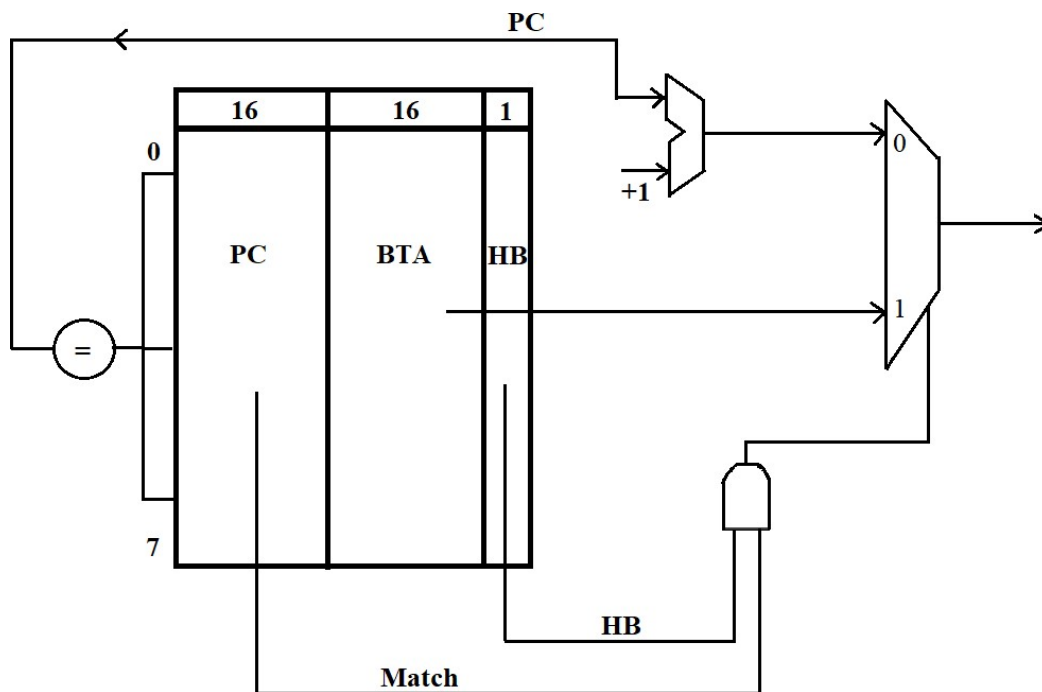


Figure 3: Branch Predictor Logic

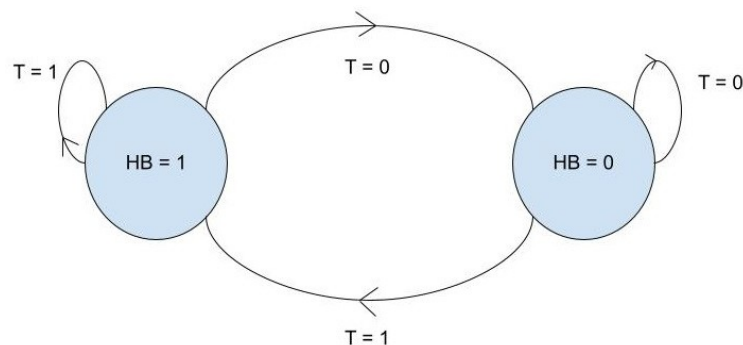
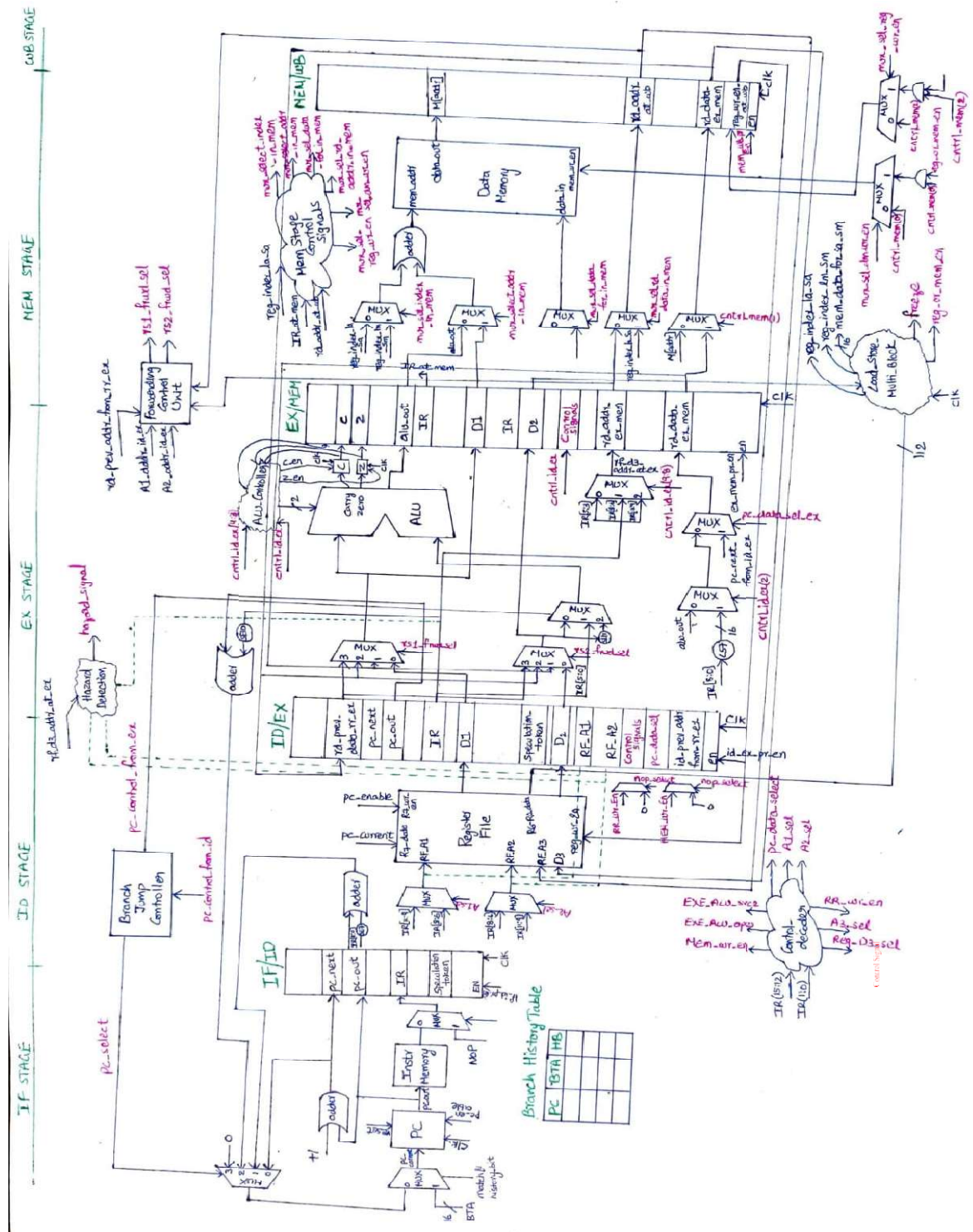


Figure 4: Branch State Diagram

3.3 Complete Datapath



3.4 Stages in the Pipeline

Instruction Fetch Stage

The first stage in the pipeline, that fetches the two byte instruction from the memory. Also consists of a PC Selector MUX, which selects between PC+1 or various other Target address based on the address resolved from ID and EX stages (jal, jlr, jri, beq). The output of PC Selector MUX goes to a branch predictor select MUX, where the other input to the MUX comes from the branch LUT. 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).

Instruction Decode + Operand Read Stage

The second stage in the pipeline, decides the operand address, the target destination address, whether the instruction modifies the memory or registers, and generates all the control signals for the steering logic, and controls to modify the system state. This stage also consists of the Register File, which was described in the previous section. It also has a NOP MUX, in case to flush the instructions when a branch occurs and the speculation of the instruction ahead was incorrect @ EX stage. JAL instruction is resolved at ID stage, if the same JAL occurs again at the same PC, there would not be any penalty since prediction takes care of it, however there would be a 1 cycle penalty when JAL at a particular PC occurs for the first time.

Execution Stage

The third stage in the pipeline, it consists of the forwarding controller, the ALU and ALU Controller which defines the enable for carry and zero, as well as redefines the register and memory write signals based on the conditional instructions. The instructions branch, jlr, jri is resolved @ EX stage, since the forwarded value of the registers is available @ EX stage. 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.

Memory Stage

The fourth stage in the pipeline, it consists of a 4KB memory, with an active high write enable. It also have a seperate 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. At this point of time the instruction coming after the LA or SA or LM or SM is halted at the ID stage by a seperate hazard logic block, and LA LM SA SM controller stalls one of LA or SA or LM or SM instruction in MEM stage until all the memory access are completed. 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.

Write Back Stage

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

3.5 Signals present in the Pipeline Registers

IF/ID Register

- (1)** Current Program Counter (PC) – Needed for calculating Branch/ Jump Address.
- (2)** PC + 1 – Needed in case if prediction fails and for updating register file for JAL, JLR.
- (3)** Instruction – Present Instruction being executed.
- (4)** Speculation – Whether the branch has been speculated.

ID/EX Register

- (1)** Current Program Counter (PC) – Needed for calculating Branch/ Jump Address.
- (2)** PC + 1 – Needed in case if prediction fails.
- (3)** Control Signals – All the control signals for the steering logic.
- (4)** Source Operand Address – needed for forwarding logic at EX
- (5)** One entry to hold the destination address and data from WB stage solely for the purpose of forwarding for 3 cycle apart data 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. WB stage simultaneously updates the register file, as well as the pipelined register).
- (6)** Instruction – Present Instruction being executed.
- (7)** Speculation – Whether the branch has been speculated.

EX/MEM Register

- (1)** Control signals – For the steering logic, register and memory write enable signals.
- (2)** Destination Address of register and ALU result – for the purpose of updating results, accessing or updating memory and for forwarding.
- (3)** Register Data to be written to memory in case of SW, SA, SM.

(4) Memory Access Address.

MEM/WB Register

(1) Register File write enable.

(2) Destination address of the Register File.

(3) Data to be updated at the Register File.

3.6 Hazards

To minimize stalling and improve the performance, we have implemented **Data Forwarding** and **Dynamic Branch Prediction using a 1 bit History**.

The Data Forwarding takes care of the immediate, 2 cycles apart and 3 cycles apart data dependency.

There are a couple more issues aside from the above two that needs to be handled. First is obviously misprediction of the branch - in that case the pipeline needs to be flushed. We have flushed the pipeline by inserting NOP instructions wherever needed since that was the most direct approach. The hazards that are addressed in this implementation are discussed below.

Branches

Branch instructions are speculated with predictions, in case if the prediction is incorrect, then the instructions in IF and ID are replaced with NOP. JLR, JRI has the target addressed based on the value of register. Hence prediction may not work most of the time. As a result prediction is only applicable for JAL and BEQ instructions. JLR, JRI will have two cycle penalty, since forwarding is implemented at EX and hence the latest value of registers are available at EX stage.

Load Hazard

Consider the below figure.

I1 : lw **ra**, MemoryAddr

I2 : op rd, **ra**, rb

Figure 5: Illustration of Immediate dependency on load

When I1 reaches EX stage, I2 will be in ID stage. However the value of ra would only be available from end of memory stage, therefore we need 1 cycle stall.

LA LM Hazards

Consider the below figure.

I1 : la/lm, MemoryAddr

I2 : op rd, ra, rb

Figure 6: Illustration of LA or LM hazards

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. If instruction I2 stays in EX stage, the main issue is that it won't get the updated register values that LA performs. Therefore the approach performed is that, I2 will be stalled in ID stage and while I1 has completed accessing the memory(MEM stage), I2 can proceed. Any pending update to register due to LA/LM instruction that is pending in WB will be handled through forwarding.

4 Results

Several Test Cases was performed to make sure that the accurate results are obtained. To our knowledge, the results are satisfactory. Here, we will demonstrate a couple of instructions being executed.

(1) The first program is to perform sum of natural numbers upto 20. The expected output should be $210 \left(\frac{n*(n+1)}{2} \right)$.

The pseudo code of the instruction is shown below.

PC	INSTRUCTION	
0x000	ADD R0,R0,R0	-- R0 = 0
0x001	ADI R1,R0,010100	-- R1 = 20 (SUM OF NATURAL NUMBERS TILL 20)
0x002	BEQ R0,R1,000100	
0x003	ADI R0,R0,000001	
0x004	ADD R2,R2,R0	-- R2 WILL HAVE FINAL RESULT (20*21/2)
0x005	JAL R4, 111111101	

Figure 7: Code to find sum of natural numbers upto 20

The values of all **registers at the beginning are 0**. The register R1 holds the value 20, and when register R0 becomes equal to R1 after repetitive iterations, the final sum will be held in R2. The code snippet from PC = 0x002 to 0x005 repeats 20 times.

The instruction memory being encoded is shown in the below figure.

```
// PROGRAM TO FIND SUM OF NUMBERS UPTO 20
instruction_mem[0] <= 16'b0001000000000000; // add r0 r0 r0
instruction_mem[1] <= 16'b0000000001010100; // ADI R1,R0,010100    r1 =20;
instruction_mem[2] <= 16'b1000000001000100; //BEQ R0,R1,000100
instruction_mem[3] <= 16'b0000000000000001; // ADI R0,R0,000001
instruction_mem[4] <= 16'b0001010000010000; // ADD R2,R2,R0
instruction_mem[5] <= 16'b1001100111111101; // JAL R4, 111111101
```

Figure 8: Screenshot of the program in Instruction Memory

All the other locations of Instruction memory are filled with NOP instructions.

The program took **85 cycles with prediction** to produce the result and the PC will point to location 0x006. Without prediction, theoretically it would have taken 104 cycles, hence there is performance improvement just by predicting jumps.

Instruction	Number of Cycles	Comments
I0: (ADD)	1	-
I1: (ADI)	1	-
I2: (BEQ)	20 (repeats 20 times) + 2	Prediction - by default NT for first 20 cycles. 2 cycle penalty for final iteration.
I3: (ADI)	20 (repeats 20 times)	-
I4: (ADD)	20 (repeats 20 times)	-
I5: (JAL)	20 (repeats 20 times) + 1	1 cycle penalty for first time Rest iterations, they are predicted taken correctly.

The final expected result is 210 in register R2, which is shown in the figure below.

00000000	00000000000010100
00000001	00000000000010100
00000002	0000000011010010
00000003	00000000000000000
00000004	00000000000000110
00000005	00000000000000000
00000006	00000000000000000
00000007	00000000000000110

210

Figure 9: Screenshot of Register File

The below figure shows the screenshot of Modelsim waveform with branch prediction. As shown in the highlighted area, when PC = 0x0005, the instruction is JAL. As a result the NEXT PC is predicted as 0x0002.

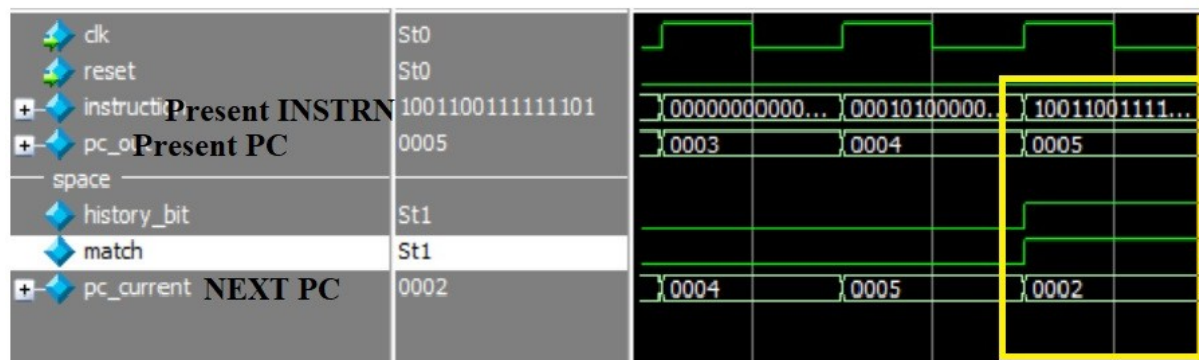


Figure 10: Demonstration of JAL prediction

The branch is predicted as NT for the first 20 iterations, as R0 and R1 are not equal, at the time it becomes equal, an entry is made into Branch History Table. However in the current program after BEQ at PC = 0x0002 becomes taken, it is not used anymore.

The below figure shows the entry in Branch History Table (BHT). The entries are made by pre - incrementing a pointer. When the pointer reaches 7, it goes back to 0 and overwrites the existing entries. In this scenario, the entry corresponding to 1 is for JAL, the most significant 16 bits represent PC (0x0005 in this case), the next significant 16 bits represents BTA(0x0002 in this case) and the LSB represents History Bit (always 1 for JAL). BEQ is in 2nd entry since it is TAKEN only in the final iteration. Here, for BEQ the entry corresponding to PC is 0x0002, the BTA is 0x0006 and History bit is 1 (since the last time while exiting the loop it was TAKEN). Successful prediction test case of branch is shown in the attached spreadsheet along with the archive.

Index	Value
00000000	xx
00000001	0000000000000001010000000000000101
00000002	00000000000000001000000000000001101
00000003	xx
00000004	xx
00000005	xx
00000006	xx
00000007	xx

Figure 11: Entries in Branch History Table

(2) For completion, the second program shows the illustration of load and store instructions mainly. The **initial conditions** are shown below.

Register	Value
R0	0
R1	1
R2	2
R3	3
R4	4
R5	5
R6	6
R7	0 (PC)

The data memory is initialized sequentially, that is, M[0] contains 0, M[1] contains 1, and so on. Hence M[4095] is 4095.

```
ADD R0, R0, R0
ADD R2, R1, R6
ADD R5, R2, R4
LA R5
ADD R1, R2, R3
LW R5, R5, 1
ADD R4, R5, R0
SA R4
SW R2, R6, 1
```

Figure 12: Test Program - 2

The encoded program in instruction memory is shown below.

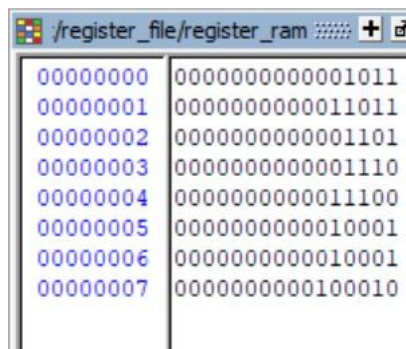
```
instruction_mem[0] <= 16'b0001000000000000;
instruction_mem[1] <= 16'b0001001110010000;
instruction_mem[2] <= 16'b0001010100101000;
instruction_mem[3] <= 16'b1110101000000000;
instruction_mem[4] <= 16'b0001010011001000;
instruction_mem[5] <= 16'b0100101101000001;
instruction_mem[6] <= 16'b0001101000100000;
instruction_mem[7] <= 16'b1111100000000000;
instruction_mem[8] <= 16'b0101010110000001;
```

Figure 13: Screenshot of Program in Instruction memory

The final register and memory being modified are

Register or Memory Locn.	Value
R0	11
R1	27
R2	13
R3	14
R4	28
R5	17
R6	17
R7	PC
M[18]	13
M[28]	11
M[29]	27
M[30]	13
M[31]	14
M[32]	28
M[33]	17
M[34]	17

The screenshot of result of register file is attached below, which matches with the expected results.



00000000	0000000000001011
00000001	0000000000001101
00000002	0000000000001101
00000003	0000000000001110
00000004	0000000000001110
00000005	0000000000001001
00000006	0000000000001001
00000007	00000000000010010

Figure 14: Screenshot of Register File for second Test Case

The memory being modified is highlighted and attached below, which also matches with the expected results.

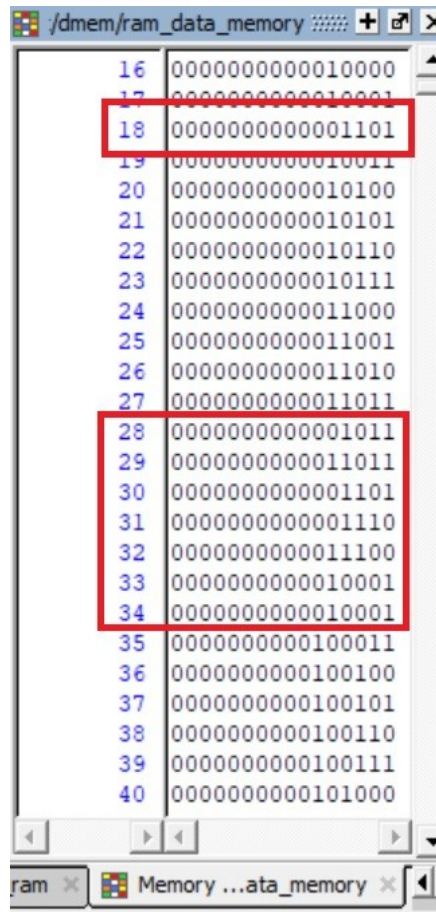


Figure 15: Screenshot of Data Memory for second Test Case

5 Conclusion

A 5 stage, 16 bit pipelined processor that supports 19 instructions and optimized for performance having data forwarding and a 1 bit branch prediction have been implemented. The design have been synthesized in Quartus 18.1 and RTL simulations have been carried out in ModelSim.