

Pipelined Processor Implementation based on RISC V ISA

Final Project Report

Prepared by:

Sparsh Wairya

2017A3PS0115P

Vibhu I Verma

2017A3PS0189P

Rithik Dilip Rathi

2017A3PS0266P

Submitted in fulfilment of the course Design Oriented Project (EEE F376/377)

Under the supervision of

Prof. S. Gurunarayanan

Professor, Department of Electrical and Electronics

AT



BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

PILANI CAMPUS

(SEM-I 2020-21)

ACKNOWLEDGEMENTS

We express our sincere thanks to **Dr. S Gurunarayanan**, Professor, BITS Pilani for guiding us right from the beginning. We sincerely acknowledge him for extending his valuable guidance, support for literature and providing valuable critique of our work and for providing us an opportunity to work on this project.

ABSTRACT

The project focuses on the design of a 5-stage pipelined processor based on RISC V ISA with the added functionalities of support for data and control hazards and Instruction Level Parallelism Using Dynamic Scheduling.

A pipeline processor consists of a sequential, linear list of segments, where each segment performs one computational task or group of tasks. For such a processor, the work (or the ISA) needs to be divided into multiple segments and must each take about the same time to complete. For a smooth operation, the processor must also take into consideration of branches and hazards. Implementation of flushing of the pipeline registers in case of branching was done and also a hazard detection unit was maintained along with a forwarding unit to take care of hazards. Finally, a branch prediction algorithm was implemented using a 2bit branch history table and the table history was maintained according to the behaviour of branching in the instructions. This helped in saving up clock cycles in case the branch occurs at the same address again, as we now already have the target address stored and hence we have a speed up of the whole branching operation.

The design is modularised in terms of different combinational and sequential units, which come together in a top-level module of the main processor. The design has been implemented using Verilog and compiled and tested using the Modelsim software.

TABLE OF CONTENTS

<i>Title Page</i>	<i>i</i>
<i>Acknowledgments</i>	<i>1</i>
<i>Abstract</i>	<i>2</i>
<i>Table of contents</i>	<i>3</i>
1.Problem Statement	4
2.Introduction	5
3.Instruction Set Architecture (ISA)	6
4.Pipeline Stages and Modules	9
5.Simulation without data hazards	13
6.Forward Unit and Hazard Detection unit	16
7. Simulation with Data Hazards	19
8.Branch Prediction	21
9.Simulation with 2-bit Branch predictor	23
<i>Conclusions</i>	<i>29</i>

PROBLEM STATEMENT

Design and implement a Pipelined Processor based on RISC V ISA in the following phases:

1. Compare and contrast the difference of RISC V with MIPS Processor
2. Data path Design of Pipelined RISC V Processor
3. Implement support for handling data hazards
4. Implement Support for handling control hazards (Use Branch Predictor local branch predictor)

INTRODUCTION

A reduced instruction set computer, or **RISC**, is a computer with a small, highly optimized set of instructions, rather than the more specialized set often found in other types of architecture, such as in a complex instruction set computer (CISC). **RISC-V** is an open standard instruction set architecture based on established reduced instruction set computer principles.

Pipeline in the processor refers to the set of registers that insert between the hardware to divide it into different stages. These stages are Instruction Fetch (**IF**), Instruction Decode (**ID**), Execute (**EX**), Memory access (**MEM**) and Write Back (**WB**) as shown in the diagram below

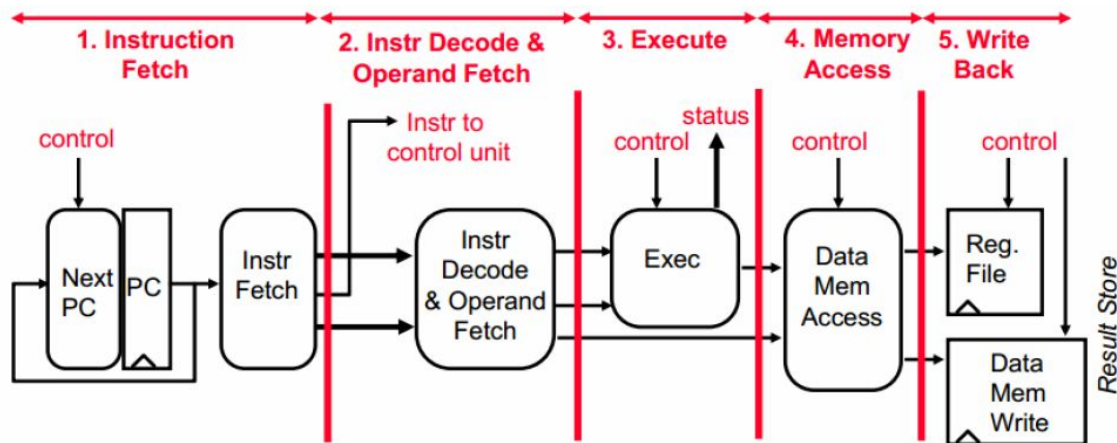


Figure 1

The input of the pipeline register in any particular stage is the output to the next stage also all the pipeline registers are synchronized by clock. This helps in passing of the information as it goes from one stage to another. Pipelining in a processor enables it to begin executing the next instruction before the current one is complete. This enhances the speed, performance and throughput of the processor as each stage of the processor processes different instructions simultaneously. This type of operation leads to different hazards like structural, data or control hazard. Mechanisms need to be developed to overcome them for smooth execution of the processor.

INSTRUCTION SET ARCHITECTURE

The ISA we followed for our implementation of the processor is the basic RV32I. In the base RV32I ISA, there are four core instruction formats R type, I type, S type and the U type. All the instructions are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory. For the simulation purpose we have made use of instructions from all formats and a register file having 32 numbers of 32bit wide registers.

Instruction specifications:

The following instructions are supported by the our current version of pipeline simulation

- **Arithmetic** - add,sub,addi
- **Logical** - and,or,xor,andi,ori,xori
- **Shift** - sll,srl,slli,srli,sra
- **Branch and Jump** - beq,bne,
- **Load/Store** - lw,sw

Instructions used in Simulation 1 (without hazards) are as follows:

Instructions	Hex Format
add x3,x1,x2	002081B3
addi x4,x1,3	00308213
sub x5,x2,x1	401102B3
ori x6,x2,4	00416313
sw x16,1(x1)	0100A0A3
xori x7,x1,8	0080C393
slli x8,x1,3	00309413

srli x9,x2,1	00115493
lw x10,5(x1)	0050A503
beq x5,x1,-11	FE128AE3

Instructions used in Simulation 2 (handling data hazards) are as follows:

Instructions	Hex Format
addi x2,x1,10	00A08113
add x3,x2,x1	001101B3
addi x4,x1,1	00108213
slli x5,x3,2	00219293
sw x16,1(x4)	010220A3
andi x6,x5,16	0102F313
addi x8,x31,1	001F8413
sub x7,x5,x6	406283B3
lw x10,5(x6)	00532503
beq x7,x8,-11	FE838AE3

PIPELINE STAGES AND MODULES

The datapath for our pipeline is shown in Figure 2 and modules defined for the implementation of the same are shown in Figure 3

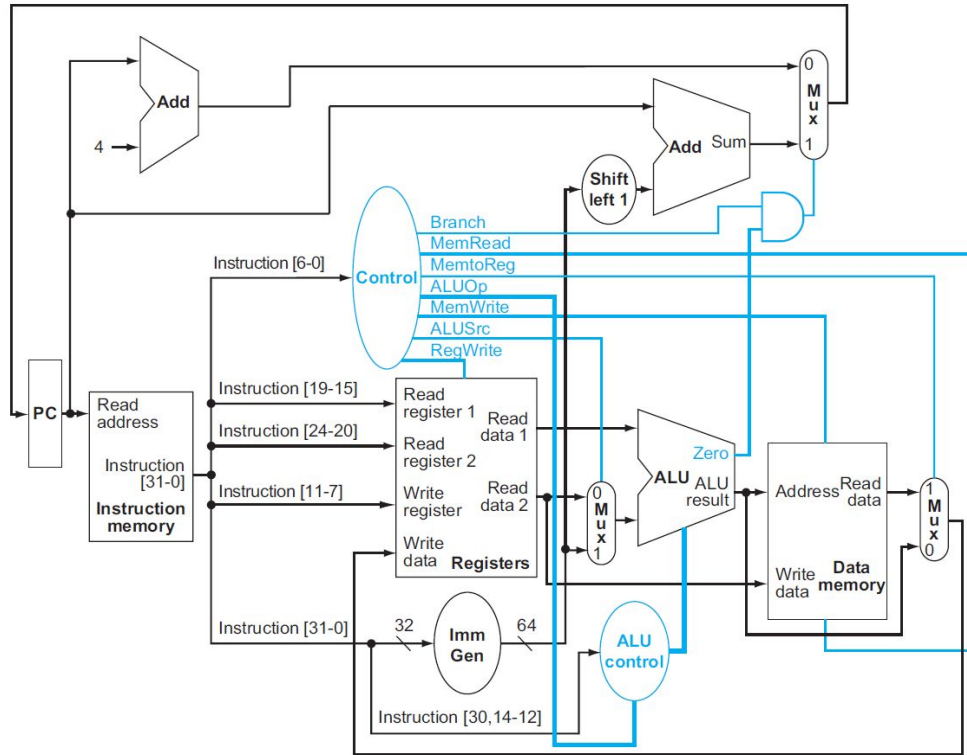


Figure 2















Name	Status	Type	Order	Modified
 mux32.v	✓	Verilog	4	10/26/2020 11:33:30 .
 data_mem.v	✓	Verilog	0	10/26/2020 11:33:30 .
 tbtop.v	✓	Verilog	7	10/26/2020 11:33:30 .
 mux.v	✓	Verilog	3	10/26/2020 11:33:30 .
 ALU.v	✓	Verilog	10	10/26/2020 11:33:29 .
 ALUcontroller.v	✓	Verilog	11	10/12/2020 12:06:22 .
 regfile.v	✓	Verilog	6	10/26/2020 11:33:30 .
 cu.v	✓	Verilog	12	10/26/2020 11:33:30 .
 top_module.v	✓	Verilog	8	10/26/2020 11:33:30 .
 adder.v	✓	Verilog	9	10/26/2020 11:33:30 .
 imm_generator.v	✓	Verilog	1	10/07/2020 10:14:25 .
 pipo_reg.v	✓	Verilog	5	10/26/2020 11:33:30 .
 data_hzd_unit.v	✓	Verilog	13	10/26/2020 11:33:30 .
 instr_mem.v	✓	Verilog	2	10/12/2020 10:05:33 .

Figure 3

Initialized Data Memory

00000000	00000000	00000001
00000002	00000002	00000003
00000004	00000004	00000005
00000006	00000006	00000007
00000008	00000008	00000009
0000000a	0000000a	0000000b
0000000c	0000000c	0000000d
0000000e	0000000e	0000000f
00000010	00000010	00000011
00000012	00000012	00000013
00000014	00000014	00000015
00000016	00000016	00000017
00000018	00000018	00000019
0000001a	0000001a	0000001b
0000001c	0000001c	0000001d
0000001e	0000001e	0000001f

Figure 4

Initialized Register Memory

00000000	00000000	00000001
00000002	00000002	00000003
00000004	00000004	00000005
00000006	00000006	00000007
00000008	00000008	00000009
0000000a	0000000a	0000000b
0000000c	0000000c	0000000d
0000000e	0000000e	0000000f
00000010	00000010	00000011
00000012	00000012	00000013
00000014	00000014	00000015
00000016	00000016	00000017
00000018	00000018	00000019
0000001a	0000001a	0000001b
0000001c	0000001c	0000001d
0000001e	0000001e	0000001f

Figure 5

ALU Control - The following table show the ALU control signals generated by the ALU control unit depending on the instruction

Instruction	ALUop[1]	ALUop[0]	funct_3	Funct_7	ALUcontrol
add	1	0	000	0000000	0010
addi	1	1	000	xxxxxxx	
lw	0	0	xxx	xxxxxxx	
sw	0	0	xxx	xxxxxxx	
sub	1	0	000	0100000	0110
beq	0	1	xxx	xxxxxxx	
bne	0	1	xxx	xxxxxxx	
xor	1	0	100	0000000	0011
xori	1	1	100	xxxxxxx	
and	1	0	111	0000000	0000
andi	1	1	111	xxxxxxx	
or	1	0	110	0000000	0001
ori	1	1	110	xxxxxxx	
sll	1	0	001	0000000	0100
slli	1	1	001	0000000	
srl	1	0	101	0000000	0101
srli	1	1	101	0000000	
sra	1	0	101	0100000	0111
srai	1	1	101	0100000	

Table 1

IF Stage

```
module top(rst,clk);
    input rst,clk;

    //Before the IF-ID Pipeline register
    wire [31:0]PCin,PCout; //Program Counter
    wire [31:0]instruction; //instruction from memory
    wire [31:0]newPC; //PC <- PC + 4
    wire [63:0]IFIDin = {newPC,instruction};

    wire PCwrite;
    wire ifidwrite;
    wire stall;

    adder #(.N(32)) add1(newPC,PCout,32'd1); //Full adder for incrementing PC
    instr_mem imem(instruction,PCout,clk,rst); //Instr Memory
```

ID Stage

```
//Between IF-ID and ID-EX Pipeline registers
wire [63:0]IFIDout;
wire [31:0]IFID_PC_out;
wire [31:0]IFID_instr_out;
wire [6:0]opcode;
wire [2:0]funct3;
wire [6:0]funct7;
wire [4:0]Rs1,Rs2,Rd;
wire [31:0]immgen_out,Rs1data,Rs2data;
wire [160:0]IDEXin;

wire [1:0]ALUop;
wire ALUsrc,MtoR,regwrite,memread,memwrite,branch;
wire [7:0]controlsig = {ALUop,ALUsrc,MtoR,regwrite,memread,memwrite,branch}; //control signals
wire [7:0]controlsig_f;

assign IFID_PC_out = IFIDout[63:32];
assign IFID_instr_out = IFIDout[31:0];
assign opcode = IFIDout[6:0];
assign funct3 = IFIDout[14:12];
assign funct7 = IFIDout[31:25];
assign Rs1 = IFIDout[19:15];
assign Rs2 = IFIDout[24:20];
assign Rd = IFIDout[11:7];
assign IDEXin = {Rs1,Rs2,IFID_PC_out,Rs1data,Rs2data,immgen_out,funct7,funct3,Rd,controlsig_f}; //Signal

pipo_reg #(.N(64)) IFID(IFIDout,IFIDin,clk,rst,ifidwrite); //Program Counter
controlunit CU(opcode,rst,ALUop,ALUsrc,MtoR,regwrite,memread,memwrite,branch); //Main Control Unit
imm_gen IG(immgen_out,IFID_instr_out,opcode); //Immediate generator depending on instruction type
```

EX Stage

```
//Between ID-EX and EX-MEM Pipeline registers
wire [160:0]IDEXout;
wire [2:0]funct3_EX;
wire [6:0]funct7_EX;
wire [4:0]Rd_EX;
wire [4:0]Rs1_EX;
wire [4:0]Rs2_EX;
wire [31:0]Rs1data_EX,Rs2data_EX,immgen_out_EX,ALUsrca;
wire [31:0]IDEX_PC_out;
wire [7:0]controlsig_EX;
wire [31:0]BRadd;
wire [3:0]ALUoperation;
wire [31:0]result;
wire zeroflag;
wire [1:0]forwardA,forwardB;
wire [31:0]ALUsrc1,ALUsrc2;
wire [106:0]EXMEMin;

assign Rs1_EX = IDEXout[160:156];
assign Rs2_EX = IDEXout[155:151];
assign IDEX_PC_out = IDEXout[150:119];
assign Rs1data_EX = IDEXout[118:87];
assign Rs2data_EX = IDEXout[86:55];
assign immgen_out_EX = IDEXout[54:23];
assign funct7_EX = IDEXout[22:16];
assign funct3_EX = IDEXout[15:13];
assign Rd_EX = IDEXout[12:8];
assign controlsig_EX = IDEXout[7:0];
```

MEM Stage

```
//Between EX-MEM and MEM-WB Pipeline registers
wire [106:0]EXMEMout;
wire [4:0]controlsig_MEM;
wire [31:0]PCsrcB;
wire PCsrc;
wire [31:0]dmemaddr,dmemdata;
wire [31:0]re_data;
wire [4:0]Rd_MEM;
wire zero_br;

wire [70:0]MEMWBin;

assign PCsrc = zero_br & controlsig_MEM[0];
assign Rd_MEM = EXMEMout[4:0];
assign dmemdata = EXMEMout[36:5];
assign dmemaddr = EXMEMout[68:37];
assign zero_br = EXMEMout[69];
assign PCsrcB = EXMEMout[101:70];
assign controlsig_MEM = EXMEMout[106:102];

assign MEMWBin = {controlsig_MEM[4:3],re_data,dmemaddr,Rd_MEM};

pipo_reg #(.N(107)) EXMEM(EXMEMout,EXMEMin,clk,rst,1'b1);
data_mem dmem(clk,controlsig_MEM[2],controlsig_MEM[1],dmemaddr,dmemdata,re_data);
```

WB Stage

```
//After MEM-WB Pipeline Register
wire [70:0]MEMWBout;
wire [31:0]WBsrcA,WBsrcB;
wire [1:0]controlsig_WB;
wire [4:0]Rd_WB;
wire [31:0]writedata;

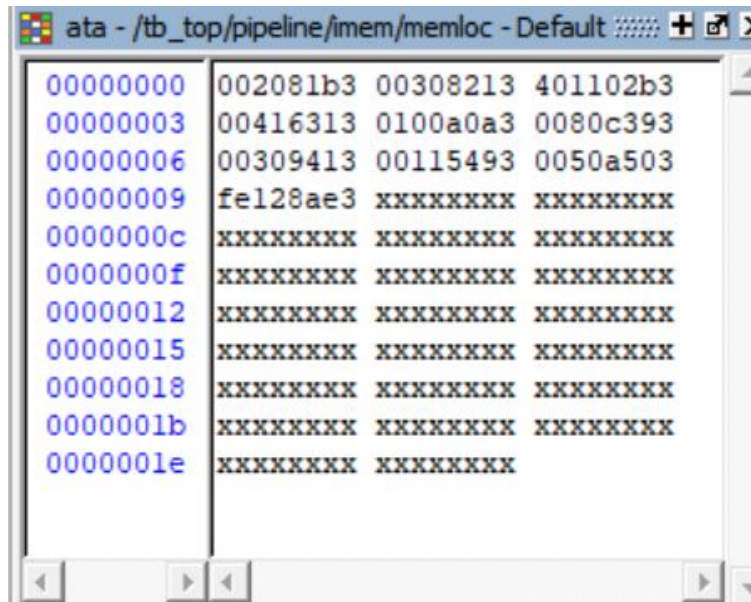
assign Rd_WB = MEMWBout[4:0];
assign WBsrcB = MEMWBout[36:5];
assign WBsrcA = MEMWBout[68:37];
assign controlsig_WB = MEMWBout[70:69];

pipo_reg #(.N(71)) MEMWB(MEMWBout,MEMWBin,clk,rst,1'b1);
mux32 #(.N(32)) M3(writedata,WBsrcB,WBsrcA,controlsig_WB[1]);
regfile Rfile(Rs1,Rs2,Rd_WB,writedata,controlsig_WB[0],clk,rst,Rs1data,Rs2data);//Register File
pipo_reg #(.N(32)) PC(PCout,PCin,clk,rst,PCwrite); //Program Counter
mux32 #(.N(32)) M1(PCin,newPC,PCsrcB,PCsrc);

fwdunit f1(Rs1_EX,Rs2_EX,controlsig_EX[3],Rd_MEM,controlsig_WB[0],Rd_WB,ALUop,forwardA,forwardB);
mux4_1 alux1(ALUsrc1,Rs1data_EX,writedata,dmemaddr,32'b0,forwardA);
mux4_1 alux2(ALUsrcb,Rs2data_EX,writedata,dmemaddr,32'b0,forwardB);
hzdunit h1(Rs1,Rs2,controlsig_EX[2],Rd_EX,PCwrite,ifidwrite,stall);
mux32 #(.N(8)) csmux(controlsig_f,controlsig,8'b0,stall);
```


SIMULATION FOR INSTRUCTIONS WITHOUT HAZARDS

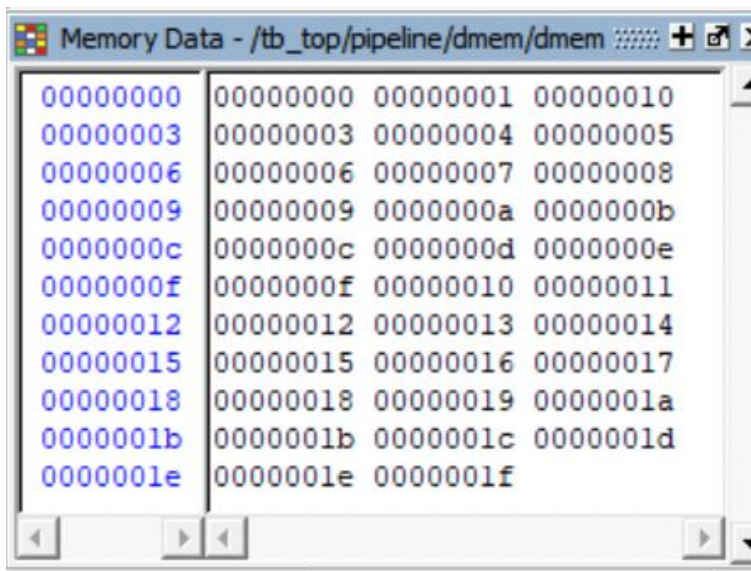
Instruction memory:



00000000	002081b3	00308213	401102b3
00000003	00416313	0100a0a3	0080c393
00000006	00309413	00115493	0050a503
00000009	fe128ae3	xxxxxxxx	xxxxxxxx
0000000c	xxxxxxxx	xxxxxxxx	xxxxxxxx
0000000f	xxxxxxxx	xxxxxxxx	xxxxxxxx
00000012	xxxxxxxx	xxxxxxxx	xxxxxxxx
00000015	xxxxxxxx	xxxxxxxx	xxxxxxxx
00000018	xxxxxxxx	xxxxxxxx	xxxxxxxx
0000001b	xxxxxxxx	xxxxxxxx	xxxxxxxx
0000001e	xxxxxxxx	xxxxxxxx	

Figure 6

Data memory:



00000000	00000000	00000001	00000010
00000003	00000003	00000004	00000005
00000006	00000006	00000007	00000008
00000009	00000009	0000000a	0000000b
0000000c	0000000c	0000000d	0000000e
0000000f	0000000f	00000010	00000011
00000012	00000012	00000013	00000014
00000015	00000015	00000016	00000017
00000018	00000018	00000019	0000001a
0000001b	0000001b	0000001c	0000001d
0000001e	0000001e	0000001f	

Figure 7

Waveform

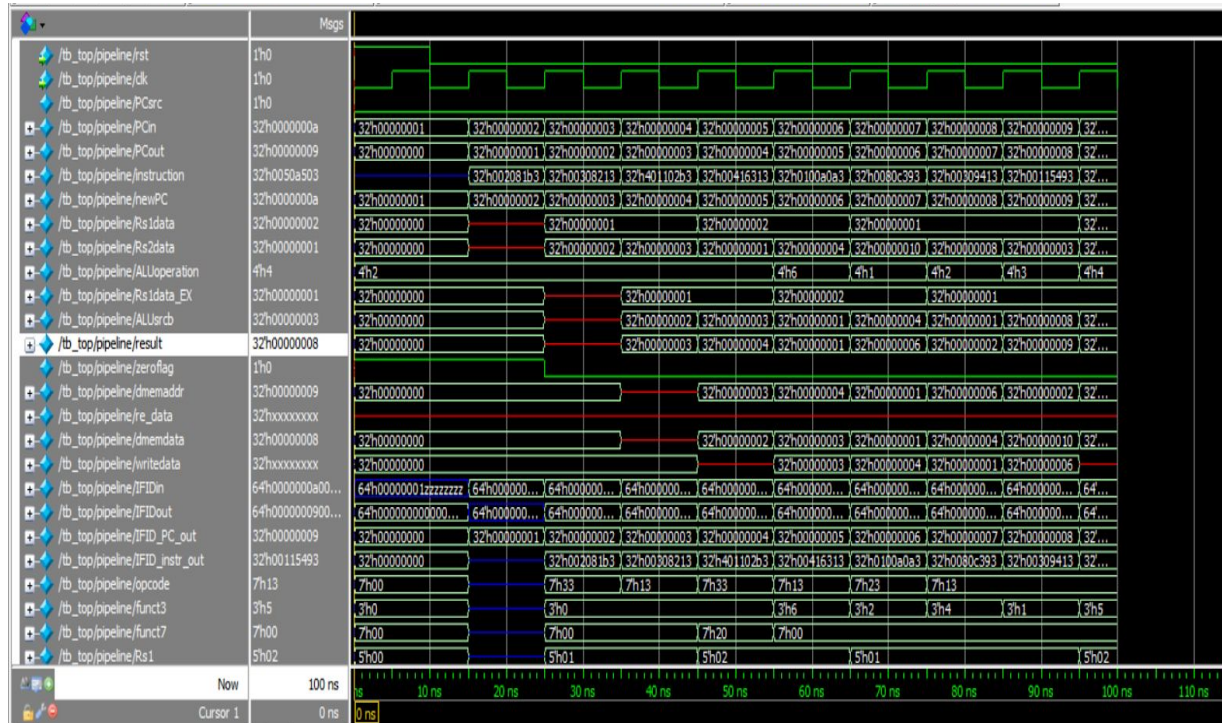


Figure 8

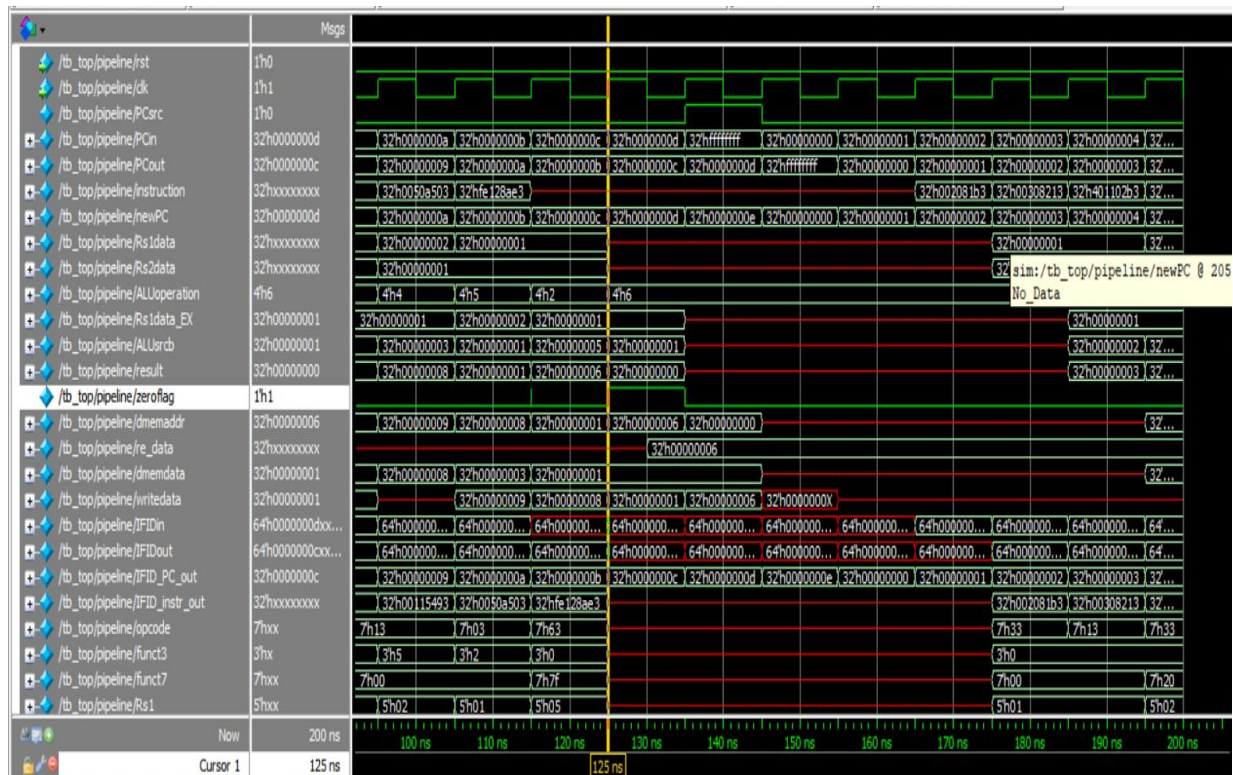


Figure 9

DATAPATH WITH FORWARDING UNIT AND HAZARD DETECTION UNIT

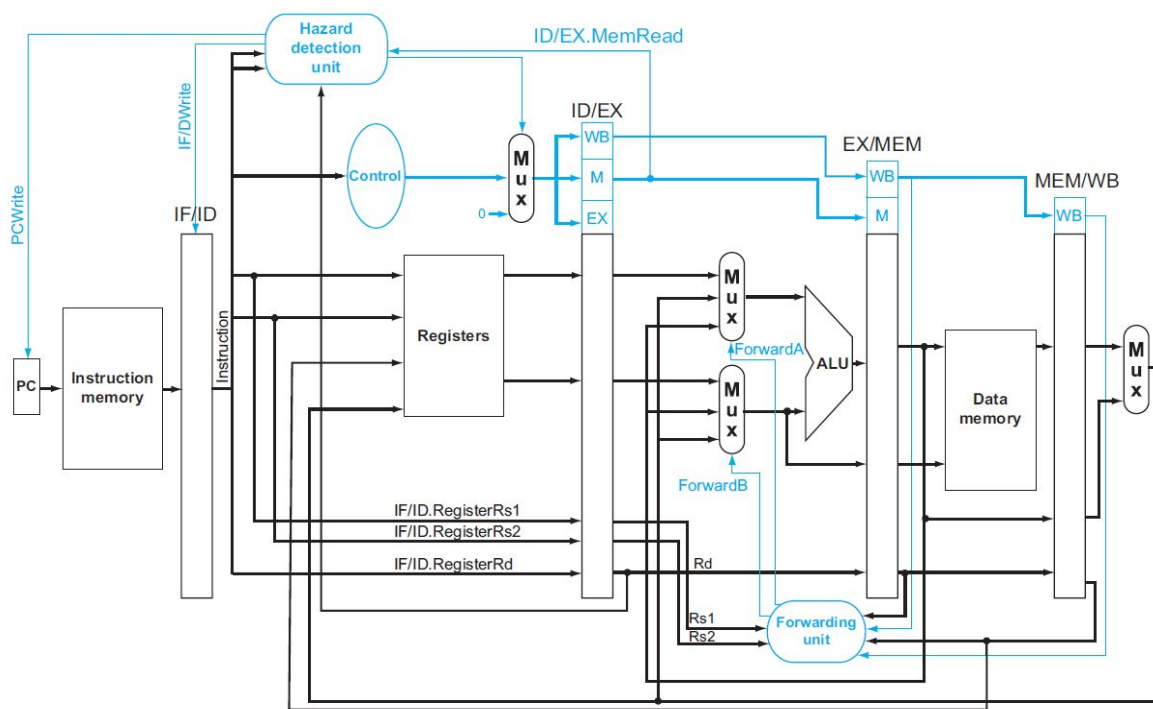


Figure 10

Including the Hazard unit and the Forwarding unit

```

1  // DATA FORWARD UNIT
2
3  module fwdunit(
4      input [4:0]idex_rs1,
5      input [4:0]idex_rs2,
6      input exmem_RegWrite,
7      input [4:0]exmem_rd,
8      input memwb_RegWrite,
9      input [4:0]memwb_rd,
10     input [1:0]ALUOp,
11     output reg[1:0] forwardA,
12     output reg[1:0] forwardB);
13
14 //Limitations of the module
15 // 1. It assumes all instruction R type, cases of hazard having other types not included yet.
16
17 //---- Function ----
18 //Two signals forwardA & forwardB decide the input to ALU.
19 //If EX hazard exist value from exmem is put in ALU
20 //If MEM hazard exist value from memwb is put in ALU
21 //If both exist EX given priority over MEM
22
23 always@(*)
24 begin
25     //if I type
26     if(ALUOp == 2'b11)
27     begin
28
29         //assign forwardA signal
30         if(exmem_RegWrite == 1'b1 && exmem_rd != 5'b0 && exmem_rd == idex_rs1)
31             forwardA = 2'b10;
32         else if(memwb_RegWrite == 1'b1 && memwb_rd != 5'b0 && memwb_rd == idex_rs1)
33             forwardA = 2'b01;
34         else
35             forwardA = 2'b00;
36
37         //assign forwardB signal
38         forwardB = 2'b00;
39
40     end
41
42     //if other types dont skip rs2
43     else
44     begin
45
46         //assign forwardA signal
47         if(exmem_RegWrite == 1'b1 && exmem_rd != 5'b0 && exmem_rd == idex_rs1)
48             forwardA = 2'b10;
49         else if(memwb_RegWrite == 1'b1 && memwb_rd != 5'b0 && memwb_rd == idex_rs1)
50             forwardA = 2'b01;
51         else
52             forwardA = 2'b00;
53
54         //assign forwardB signal
55         if(exmem_RegWrite == 1'b1 && exmem_rd != 5'b0 && exmem_rd == idex_rs2)
56             forwardB = 2'b10;
57         else if(memwb_RegWrite == 1'b1 && memwb_rd != 5'b0 && memwb_rd == idex_rs2)
58             forwardB = 2'b01;
59         else
60             forwardB = 2'b00;
61
62     end
63
64 end
65 endmodule
66

```

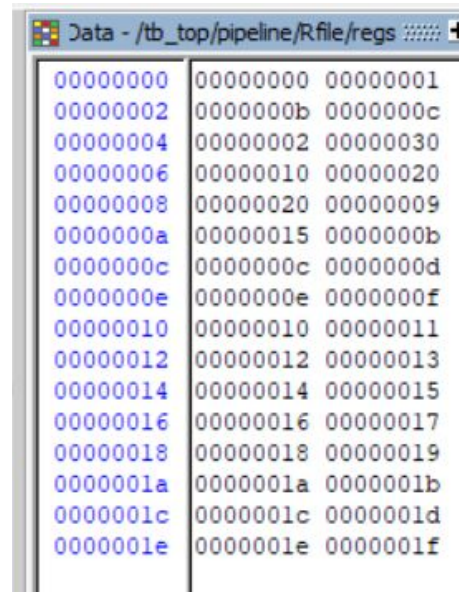
```

67 module hzdunit(
68     input [4:0]ifid_rs1,
69     input [4:0]ifid_rs2,
70     input idex_MemRead, |
71     input [4:0]idex_rd,
72     output reg PCWrite,ifidWrite,stall
73 );
74
75 //---- Function ----
76 //Two signals PCWrite & ifidWrite control if values will be written in PC and IFID resp.
77 //stall controls a mux to choose b/w hard ground or CU outputs
78 //Detects WB hazard i.e in case of load followed by R type
79
80 always@(*)
81 begin
82     if(idex_MemRead == 1'b1)
83     begin
84         if(idex_rd == ifid_rs1 || idex_rd == ifid_rs2)
85         begin
86             stall = 1'b1;
87             PCWrite = 1'b0;
88             ifidWrite = 1'b0;
89         end
90     end
91     else
92     begin
93         stall = 1'b0;
94         PCWrite = 1'b1;
95         ifidWrite = 1'b1;
96     end
97 end
98 endmodule

```

SIMULATION FOR INSTRUCTIONS WITH HAZARDS

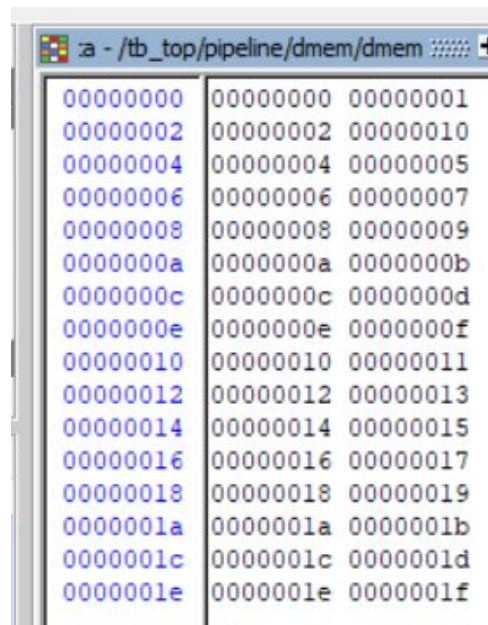
Register Memory:



00000000	00000000	00000001
00000002	0000000b	0000000c
00000004	00000002	00000030
00000006	00000010	00000020
00000008	00000020	00000009
0000000a	00000015	0000000b
0000000c	0000000c	0000000d
0000000e	0000000e	0000000f
00000010	00000010	00000011
00000012	00000012	00000013
00000014	00000014	00000015
00000016	00000016	00000017
00000018	00000018	00000019
0000001a	0000001a	0000001b
0000001c	0000001c	0000001d
0000001e	0000001e	0000001f

Figure 11

Data Memory:



00000000	00000000	00000001
00000002	00000002	00000010
00000004	00000004	00000005
00000006	00000006	00000007
00000008	00000008	00000009
0000000a	0000000a	0000000b
0000000c	0000000c	0000000d
0000000e	0000000e	0000000f
00000010	00000010	00000011
00000012	00000012	00000013
00000014	00000014	00000015
00000016	00000016	00000017
00000018	00000018	00000019
0000001a	0000001a	0000001b
0000001c	0000001c	0000001d
0000001e	0000001e	0000001f

Figure 12

Waveform:

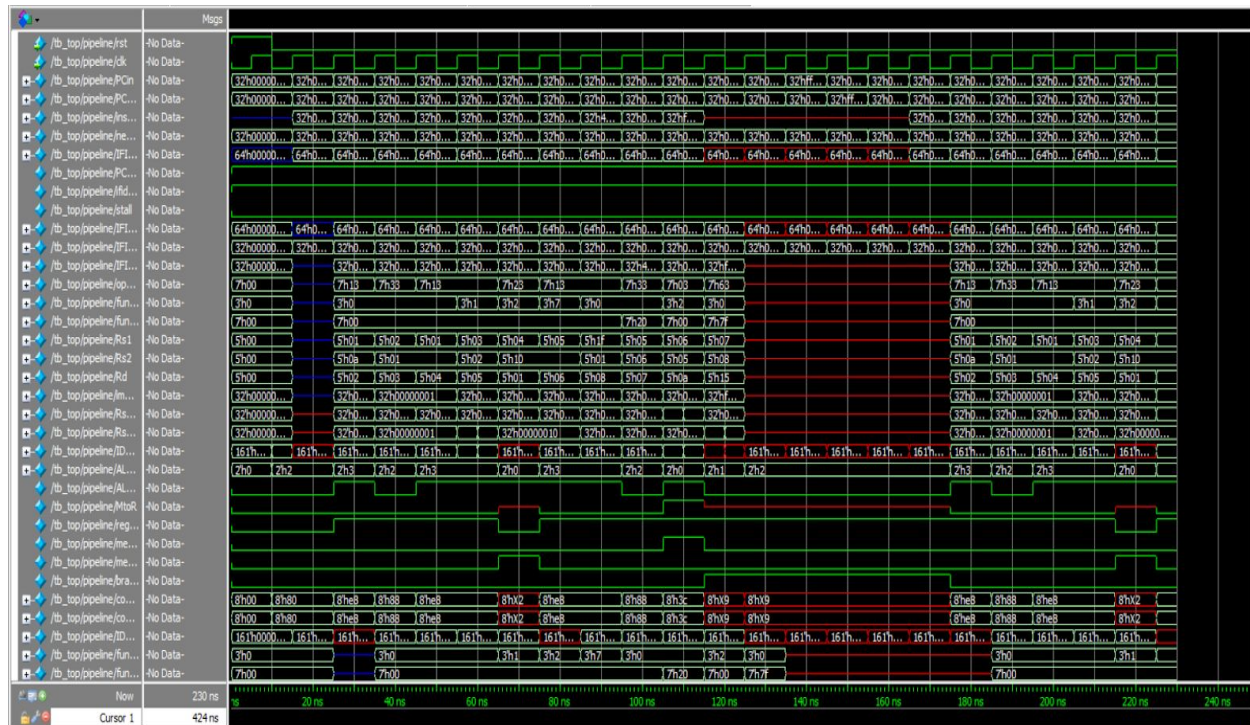


Figure 13

BRANCH PREDICTION

A normal pipelined processor without any prediction implementation would only know about the status of its branching instruction in the EX stage and till then, the next two instructions would already have been loaded in the previous pipeline registers. Now, if the branch is to be taken, we have two things to do, first, the instructions which are loaded in the previous pipeline registers are to be flushed out and a new instruction has to be fetched in the IF stage using the target address. This whole process makes our functioning of the pipeline slow, and decreases the CPI.

Branch Prediction is a common technique which is used to speed execution of branch instructions in pipelined processors. This is usually implemented using a branch predictor hardware and embedding it in the datapath of the processor. This usually includes a **FSM**, which tells us about the current state of the particular branch instruction and gives us the prediction and the states are updated accordingly to the branch result and it also has a piece of hardware which is commonly known as **Branch Target Buffer (BTB)**.

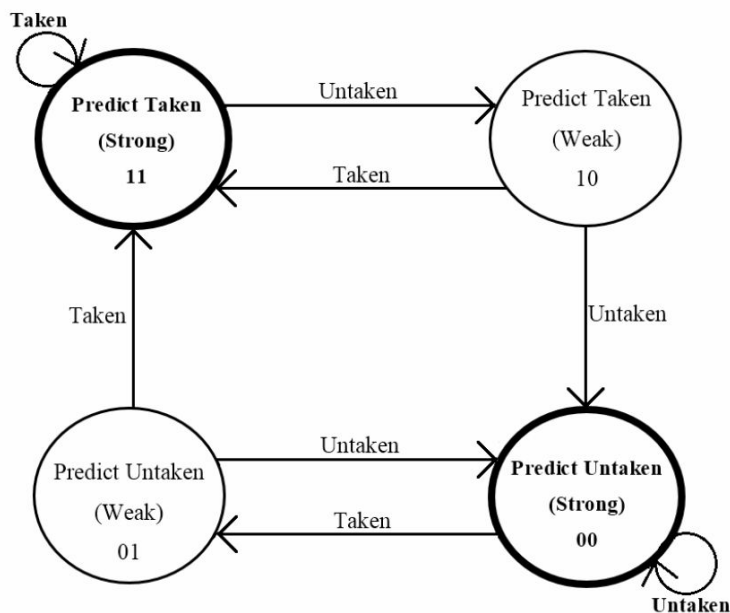


Figure 14 - The BTB FSM

The above figure shows the FSM which was implemented for the branch prediction operation. We have 4 states in the FSM as shown and it takes two rights to switch the state from strongly not taken to strongly taken and vice versa. This certifies that the chances of prediction being correct is high, as we observe the behaviour of the branches, update the states in the FSM and accordingly make a prediction. A new entry in the BTB starts with a strongly not taken state.

A snapshot of the FSM code:

```

1  //4 state Moore FSM for branch prediction
2  module bp_FSM(ip_state,b_res,op_state,pred);
3  input [1:0]ip_state;
4  input b_res; //Actual result of branch
5  output reg [1:0]op_state; //Output state after verifying actual result
6  output reg pred; //predicted operation for branch
7
8  localparam strong_nt = 2'b00,weak_nt = 2'b01, weak_t =2'b10,strong_t =2'b11;
9  reg [1:0]next;
10 //Output and Next state computation
11 //Same always block as we have considered Moore machine
12
13 always@(*)
14 begin
15     case(ip_state)
16     strong_nt: //00
17     begin
18         pred <= 1'b0;
19         if(b_res)
20             next <= weak_nt; //00-->01
21         else
22             next <= strong_nt; //00-->00
23     end
24     endcase
25 end

```

```

23     weak_nt: //01
24     begin
25         if(b_res) begin
26             next <= strong_t; //01-->11
27             pred <= 1'b1;
28         end
29         else begin
30             next <= strong_nt; //01-->00
31             pred <= 1'b0;
32         end
33     end
34     weak_t: //10
35     begin
36         if(b_res) begin
37             next <= strong_t; //10-->11
38             pred <= 1'b1;
39         end
40         else begin
41             next <= strong_nt; //10-->00
42             pred <= 1'b0;
43         end
44     end
45 end

```

```

46     strong_t: //11
47     begin
48         pred <= 1'b1;
49         if(b_res)
50             next <= strong_t; //11-->11
51         else
52             next <= weak_t; //11-->10
53         end
54     default:
55     begin
56         pred <= 1'b0;
57         next <= strong_nt;
58     end
59     endcase
60     op_state = next;
61 end
62 endmodule

```


Branch Target Buffer: BTB is a small piece of memory used to gather and store information about branch operations. The BTB behaves like a look-up table for the branch predictor to look for information of previous branch instruction that having same tag with the current address to perform branch prediction for the current instruction. We have implemented a 32 location BTB which is indexed by the last 5 bits of our instruction. The BTB in our case stores the target address, state of that branch, and the prediction bit.

A snapshot of the the BTB code:

```

1  module branch_table(rst,r_address,w_address,wr_data,re_data);
2      input rst;
3      input [4:0]r_address,w_address; //for now, a 5bit wide address line as only 32 word memory considered
4      input [34:0]wr_data;
5      output reg[34:0]re_data;
6
7      reg [[34:0]]btb[0:31]; //32 word data memory
8      integer i;
9
10     initial
11     begin
12         for(i = 0; i < 32; i = i+1)
13             btb[i] = 35'd0; //Loading memory with its corresponding address location
14     end
15
16     always@(rst)
17     begin
18         if (rst) begin
19             for(i = 0; i < 32; i = i+1)
20                 btb[i] = 35'd0;
21         end
22     end
23
24     always@(r_address)
25     begin
26         re_data <= btb[r_address];
27     end
28
29     always@(*)
30     begin
31
32         btb[w_address] <= wr_data;
33     end
34 end
35
36 endmodule

```

The Process of Branch Prediction:

- A 32 location BTB with each location of 35 bits (32+2+1) containing the predicted address, branch state and the prediction bit is initialized.
- The branch state here refers to the state fed by the FSM which is controlled by the actual result of the branch instruction.
- The BTB is indexed using the last 5 bits of the instruction address and so, whenever we encounter a branch for the first time, the target address, state and prediction of that particular branch is stored at the respective address in the BTB, to use that and reduce the number of cycles used in branching if the same branch is encountered again
- We check the predicted address with the actual branch target address for safety purposes, and if by chance the two have different values, the pipeline is flushed, and the predicted address is updated with the correct value.
- The overall process includes a number of modules, which are well incorporated in the top module.

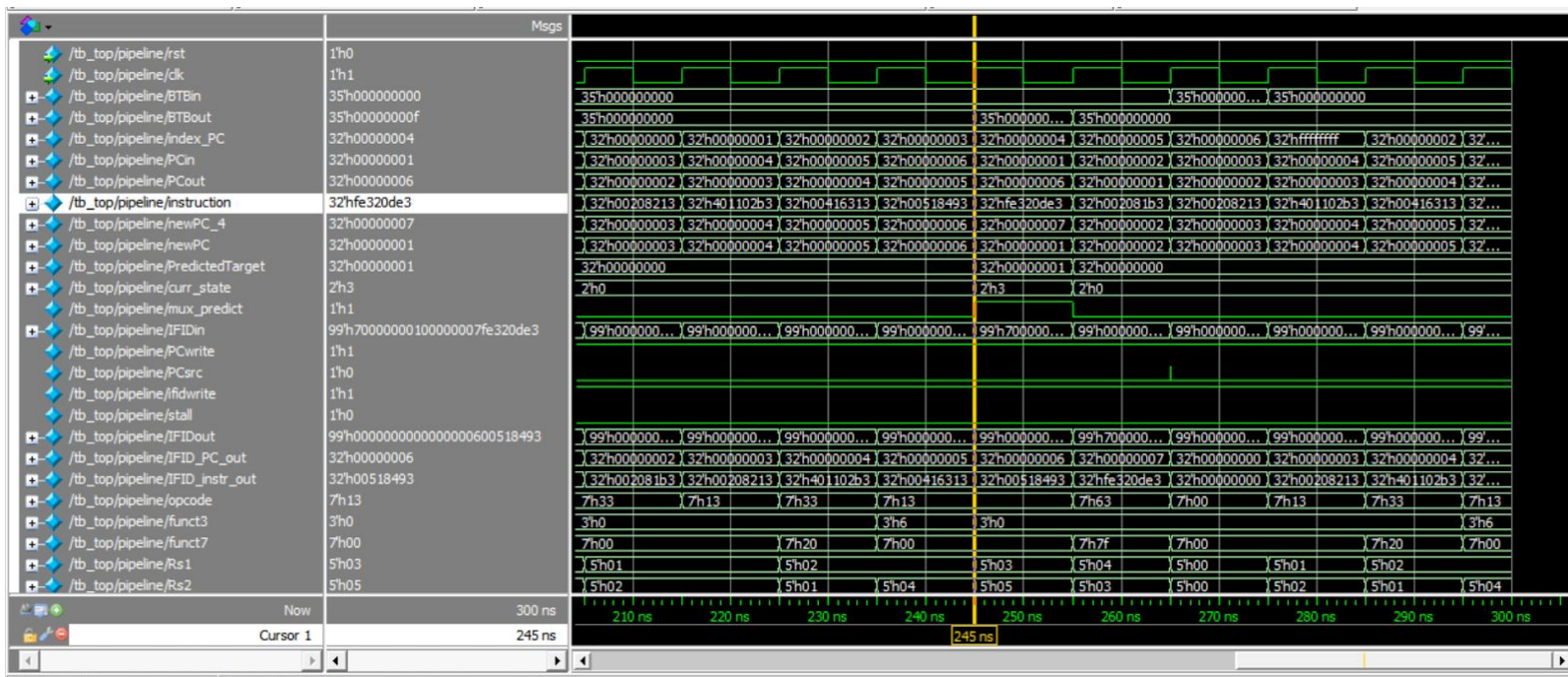
The instruction stimulus used for testing:

```
1  add x12,x11,x10
2  add x3,x1,x2
3  addi x4,x1,2
4  sub x5,x2,x1
5  ori x6,x2,4
6  addi x9,x3,5
7  beq x4,x3,-5
8  addi x8,x4,1
9  sub x9,x6,x5
10 addi x10,x11,1
```

Equivalent Hex code:

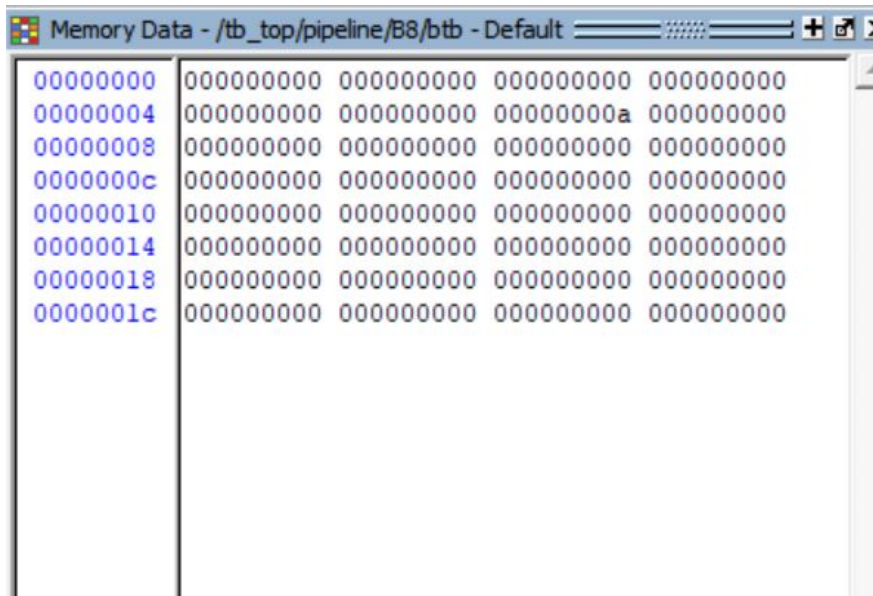
```
1  00A58633
2  002081B3
3  00208213
4  401102B3
5  00416313
6  00518493
7  FE320DE3
8  00120413
9  405304B3
10 00158513
```


Simulation with the branch prediction module:



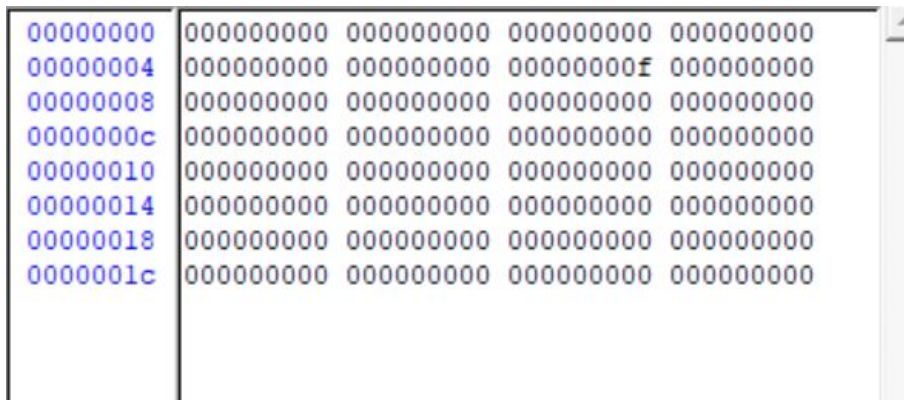
Now with the branch prediction feature included, it can be seen clearly that when the branch instruction (yellow marker) is appearing for the third time here (after two successful taken, which makes the FSM state as 11, strong taken), the branch directly takes the predicted address $PC = 01$ and we end up saving clock cycles in the whole operation. So, this is a strong advantage of including this feature which makes the performance of our pipelined processor even strong.

BTB during the whole operation



00000000	00000000	00000000	00000000	00000000
00000004	00000000	00000000	0000000a	00000000
00000008	00000000	00000000	00000000	00000000
0000000c	00000000	00000000	00000000	00000000
00000010	00000000	00000000	00000000	00000000
00000014	00000000	00000000	00000000	00000000
00000018	00000000	00000000	00000000	00000000
0000001c	00000000	00000000	00000000	00000000

Branch coming for the first time with target address 0x01, branch state = 01 and prediction = 0 making the entry at 0x06 as '0000000a'



00000000	00000000	00000000	00000000	00000000
00000004	00000000	00000000	0000000f	00000000
00000008	00000000	00000000	00000000	00000000
0000000c	00000000	00000000	00000000	00000000
00000010	00000000	00000000	00000000	00000000
00000014	00000000	00000000	00000000	00000000
00000018	00000000	00000000	00000000	00000000
0000001c	00000000	00000000	00000000	00000000

Branch coming for the second time with target address 0x01, branch state = 11 and prediction = 1 making the entry at 0x06 as '0000000f'

Now, this entry remains the same further as we are in the strongly taken state and the branch is being taken continuously with every 100ns in simulation time

CONCLUSION

The RISC-V 5 stage pipeline was designed and simulated standalone without any hazards. Functionality to handle data hazards was successfully added and tested via simulations. Further, a mechanism to handle control hazards was added too, one of which is having static prediction of “Not taken” always and another a 2-bit branch local predictor. Both these were verified using simulations in MODELSIM and tested.

For the future scope, the processor can be implemented using FPGAs and subjected to testing on standard benchmarks to gauge the performance.