National Institute of Technology, Karnataka

June 23, 2020

Master Of Technology

Design of a Multi Cycle Processor

Students:
C Akshay Kumar (192SP006)
Katragadda Kiran Kumar (192SP012)

Guide:
Dr. M. S. Bhat

Contents

List of Figures

ABSTRACT

In this project we are focused on improving the performance of a Single Cycle MIPS processor by adding a **5 - stage Pipe-lining**. But the performance of the processor won't hike as expected theoretically because of the **Hazards** like Structural, Data and Control. We also worked to resolve these hazards by techniques like Data Forwarding, Branch Target Buffer. So at the end we are left with a processor who's performance is much better than a single cycle processor.

Keywords: *MIPS Multi-cycle, Pipe-lining*

ACKNOWLEDGEMENT

We would like to express our sincere thanks to our Faculty, Dr. M.S. Bhat for giving us the opportunity to take on this project and providing motivation to keep us engaged in our work.

We really liked the way Bhat Sir started with small Verilog assignments and progressed slowly onto the more complex things and finally the "Multi-Cycle Processor" design. It as very helpful as the class portions and the work of Processor design went side by side. Moreover, his motivation helped us to get better day by day with more challenging things. Finally we would like to again thank Bhat sir for his support and motivation.

1 Introduction

1.1 Overview

In this project we are going to design a multi cycle MIPS processor. The name MIPS stands for **Microprocessor without Interlocked Pipelined Stages**, which is a reduced instruction set computer (RISC) and it is developed by MIPS Technologies, US. There are 5 stages in this processor named: **Instruction Fetch**, **Instruction Decode**, **Execute**, **Memory** and **Write Back**.

Ideally the **throughput** of this Multi Cycle Processor will be number of stages (5 in this case) when compared to the a Single Cycle Processor. But practically it is much lesser due to **Pipeline Hazards**. These hazards are downsides of pipelining. In this project we are also going to resolve these hazards and making the performance much more closer to the Ideal one.

1.2 Outline of this Thesis

The chapter 2 of this thesis discuss about Single Cycle Processor and how to make it Multi cycle, Chapter 3 will give brief insights about Pipelining Hazards and the steps we have taken to resolve them, followed by Chapter 4 which discusses in detail about the Instruction Set Architecture of this Processor. Chapter 5 holds the result & simulation section comparing the performances of Multi Cycle Processor with and without hazard resolving techniques and Finally Chapter 6, 7 & 8 is all about Program Code, Conclusions and Future Work.

2 Single Cycle MIPS Processor

2.1 Introduction

For a processor to work, it has to fetch the instruction, understand (decode) the instruction, execute the command and store the results if needed. The single cycle MIPS processor means that an entire Instruction is executed completely in one cycle.

2.2 Datapath

The datapath of the single cycle MIPS processor is shown below.

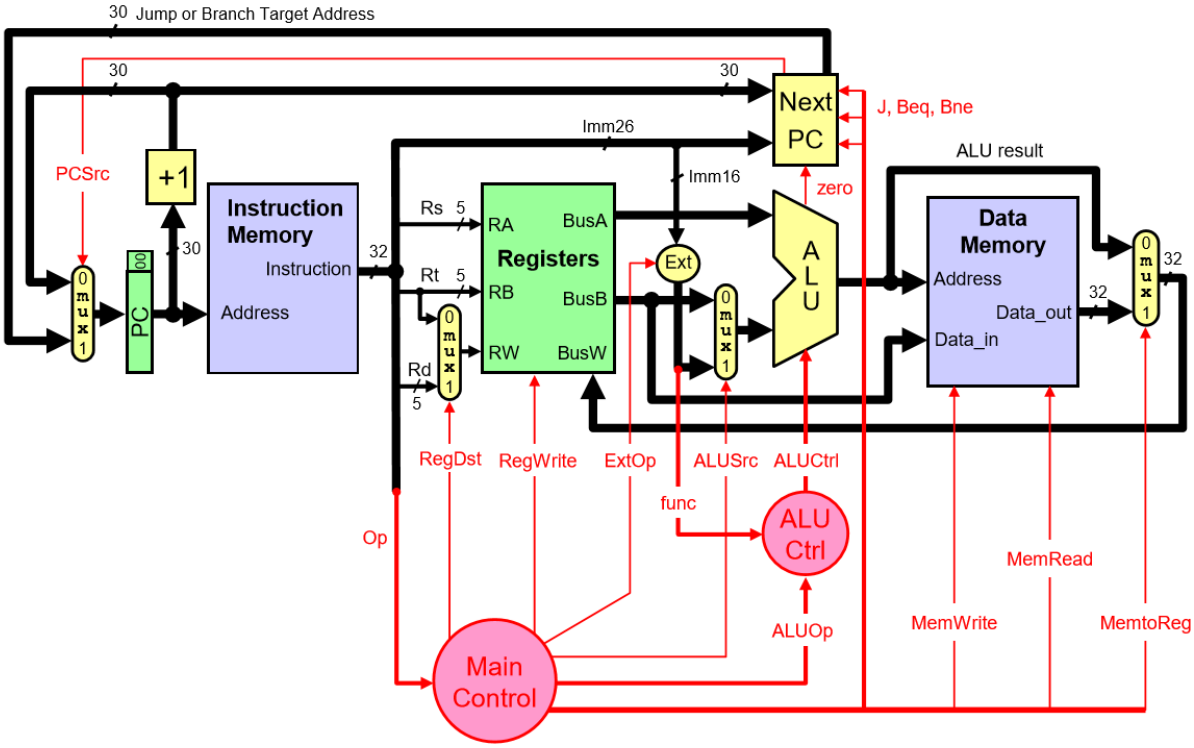

Figure 1: Single Cycle Processor Datapath

PC (Program Counter) is used to store the address of Instruction, By using this PC we will load an 32' Instruction from Instruction Memory, from which we take out Opcode, Register values, function and Intermediate data. Opcode and Function are used to specify which instruction to perform, so they are fed to the **Main Control** which will generate all the Control Signals. Register Values are used to load or store the data from Register Bank and the Intermediate value is used in Execution. Finally ALU result is generated after seeing ALU Control to decide which operation to perform on the Data coming from BUS A and BUS B. The Result is either stored in Register or Used to load or store from the data memory. This is just a brief explanation about the Single Cycle Processor Datapath.

2.3 Problem & Improvements

The problem with this single cycle is that since everything has to be completed in 1 cycle, the clock cycle time will be large. Due to this most of the elements will be **idle** in this large clock cycle, this is because the **Fetch Stage** has to wait until the **Write back** completes the first instruction and then only Fetch stage will get a new instruction. That means each stage is dependent on others.

The solution for this problem is make them Independent of each other, then the Fetch stage is free to execute a new instruction as soon as it finishes fetching the first instruction. That concept is making these stages work independently is called as **Pipelining**.

To make this possible we need to store the result generated by each stage until the next stage takes it and starts it's execution. That's why we are going to add 4 pipeline registers in between these 5 stages to store the intermediate result making them free to execute a new instruction independent of each other.
The Pipelined Datapath will look like this:

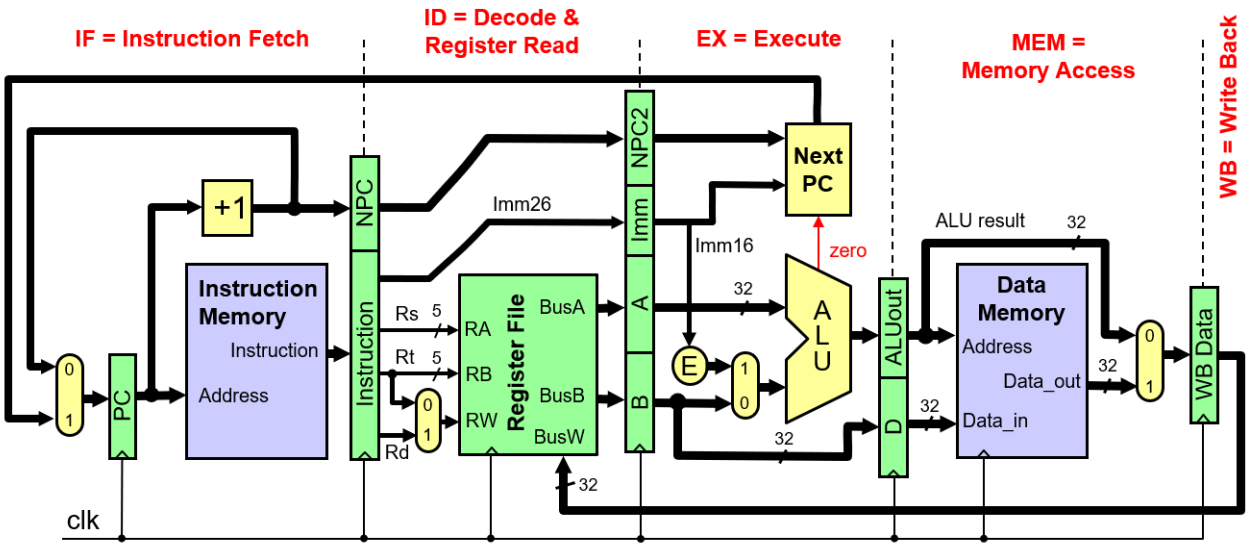

Figure 2: Multi Cycle Processor Datapath

The data flow between the stages isn't the only thing, the control signals are equally important to get expected output. So they also need to be pipelined because every instruction has separate set of control signals which are generated by Main Control and some of them are needed until EX stage while some of them are needed until the WB Stage. So we are going to use Registers to store the control signals too and the final datapath with pipelined control signals will look like:

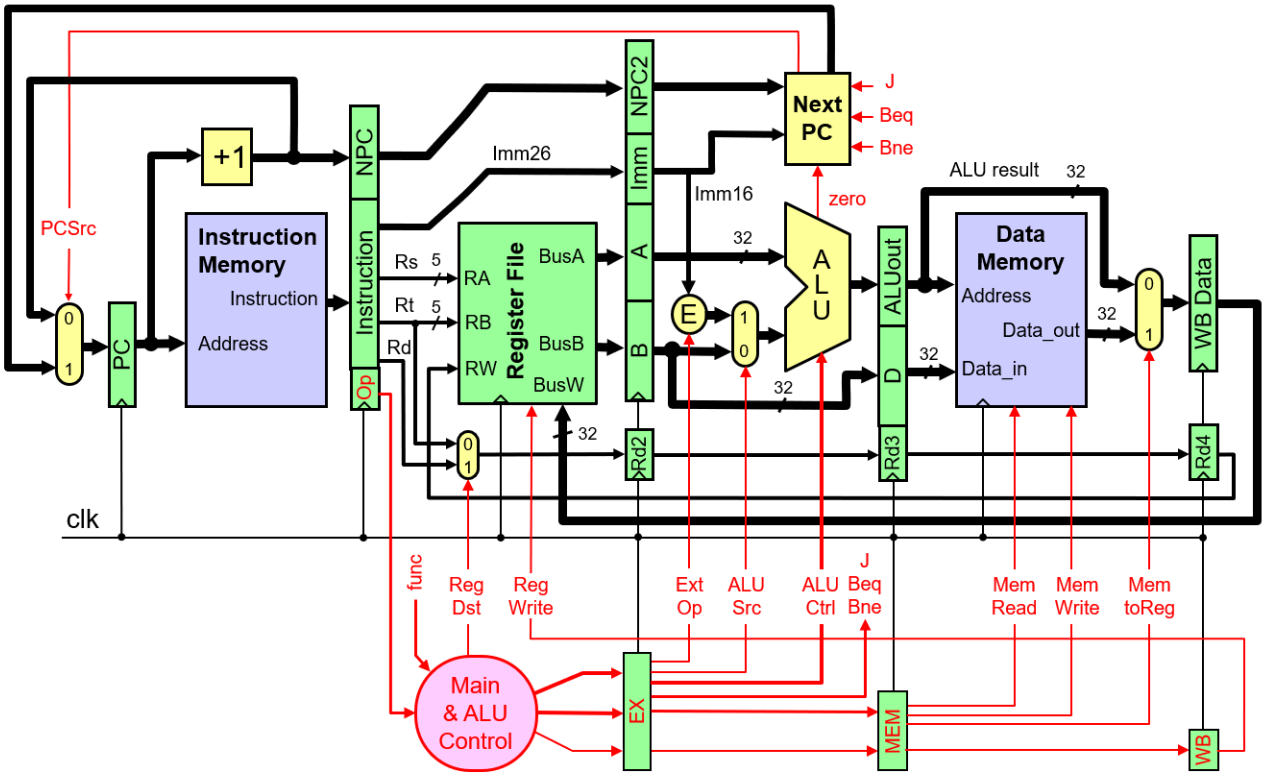

Figure 3: Complete Multi Cycle Processor Datapath

This is the final Multi Cycle processor which we are going to implement. The traits of this processor are: Now the stages are independent of each other and could execute in parallel. The downsides are the processor is now complex than the single cycle processor.

3 Resolving Pipelining Hazards

3.1 Introduction

Pipelining makes the stages independent, so because of this the Fetch stage don't the decision taken by the Execution stage for a branch, similarly the decode stage don't know whether the data in the register is the correct data or not. These are called as **Pipeline Hazards**. There are majorly 3 kinds of pipeline hazards, **Structural Hazards**, **Data Hazards** and **Control Hazards**.

3.2 Structural Hazards

The Structural Hazards are caused by **Resource Contention** which means using the same resource at the same time. This will happen if we allow **out of order completion** and if we use same memories for stroing Instructions and Data.

There is no possibility of Structural hazards in our code, because we are using **Two Different Memories to store Instructions and Data**. Also out of order completion is not allowed, by making it strict to pass through the 5 stages even if it don't need a stage in between.

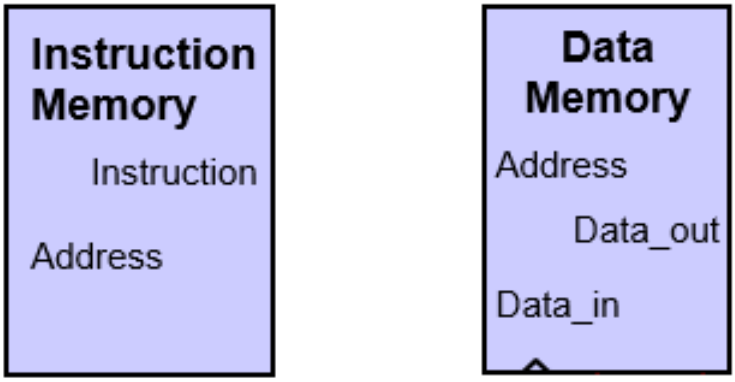

Figure 4: Separate Instruction and Data Memories

3.3 Data Hazards

The Data Hazards are caused **when the instruction compute a result which is needed by the next instruction**, when this happens the next instruction is going to use the old data (stored in Memory / Register Bank) because the Instruction decode stage is already happened.

To resolve this, we are using the concept of **Data Forwarding** in our code, so that the results which aren't stored in the memory / register bank can be used from the successive stages to calculate the correct result. But for Load there will be a stall which can't be eliminated as the data will come from **Memory Stage**, when this happens we are just stalling the pipeline.

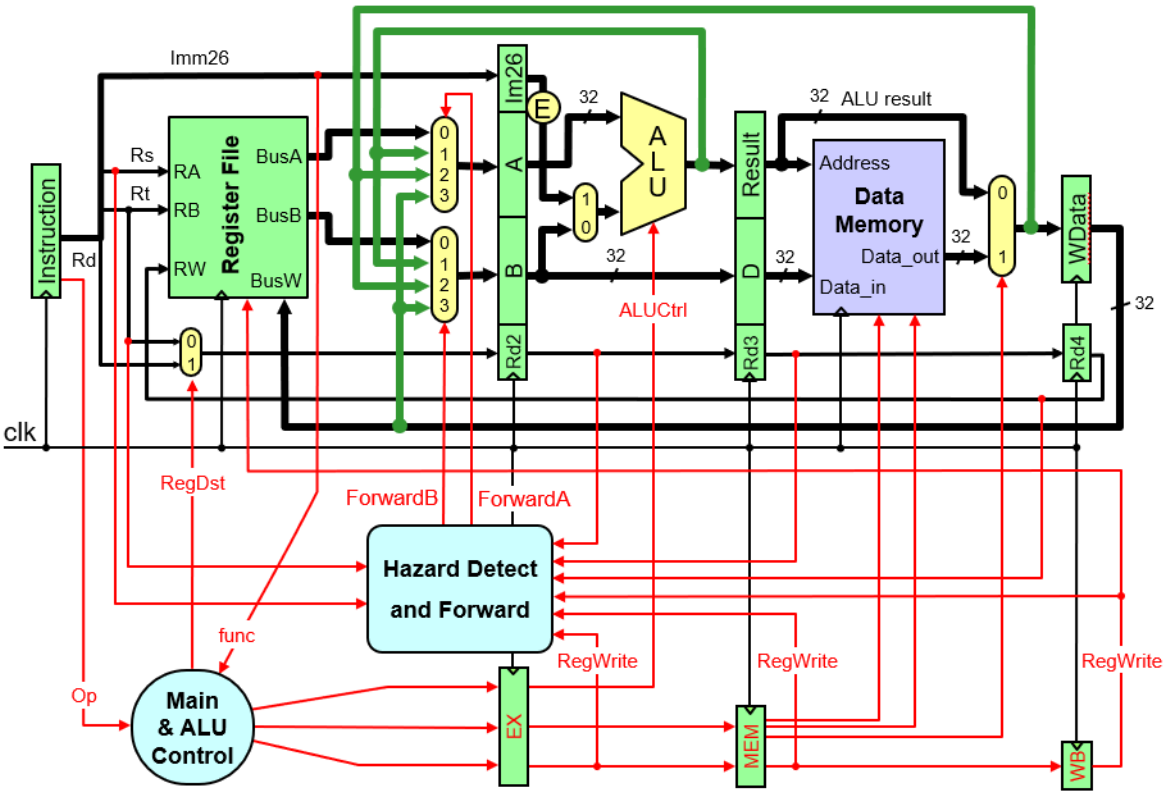

Figure 5: Data Path with Hazard Detect and Stall

3.4 Control Hazards

The control Hazards are caused because the Fetch stage is independent of Execution stage due to pipelining and due to which it don't know the decision of a **branch whether to take or not to take**. When a control hazard occurs, there will be a delay of 2 cycles if we take the wrong path.

To resolve this we are using a **Branch Target Buffer** which uses lower bits of PC (**10** bits in our Case) to know where this Instruction will probably jump next. A **2 bit prediction scheme with Alternative State Machine** is used to effectively predict the branch flow. When the decision is known to be wrong, we update the predict bits such that for the next time we take a correct decision. Also we need to flush the loaded instructions and go to new PC and start the process again.

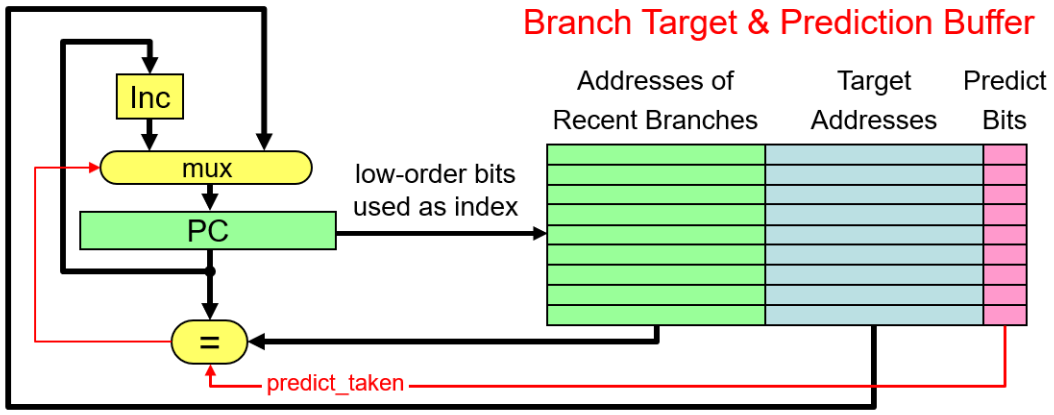

Figure 6: Branch Target Buffer with 2 Predict Bits

This is the Update rule:

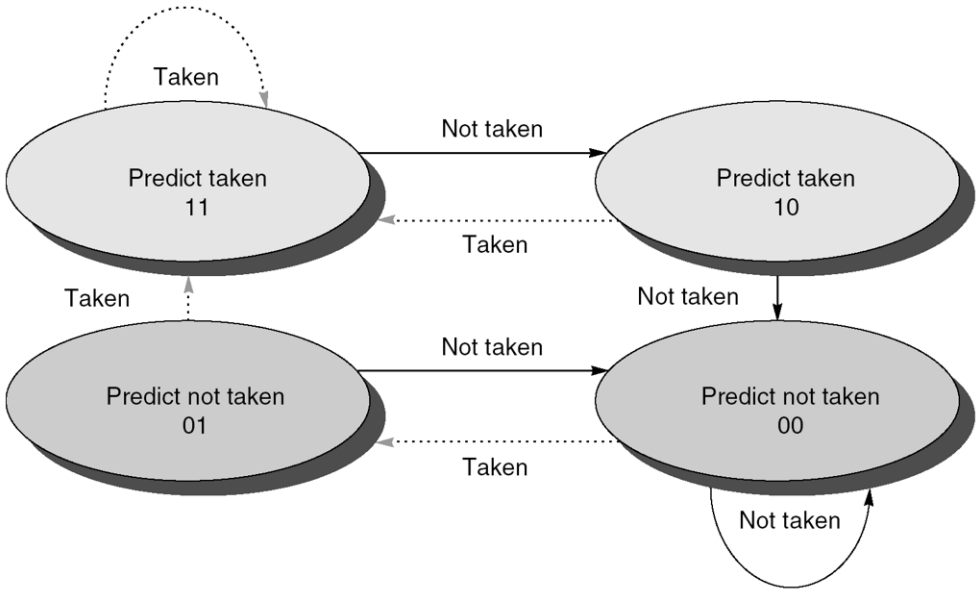

Figure 7: 2 bit Alternative State Machine

4 Instruction Set Architecture

4.1 Introduction

The Instruction Set Architecture specifies us about the Instruction Encoding, Location of the Operands and the Results, also the data types, operations etc. This ISA uses Registers and Memory Locations to load and store data. There are 3 types of Instructions used in this ISA, R-type, I-type & J-type. The following section gives the complete ISA used for this Processor and it contains 26 operations.

4.2 Complete ISA

INSTRUCTION	OPCODE	FUNCTION	MEANING	CLASS	TYPE
ADD	6'b000000	6'b000001	Rx = Ry + Rz	BINARY	R
SUB	6'b000000	6'b000010	Rx = Ry - Rz	BINARY	R
AND	6'b000000	6'b000100	Rx = Ry & Rz	BINARY	R
OR	6'b000000	6'b001000	Rx = Ry | Rz	BINARY	R
XOR	6'b000000	6'b010000	Rx = Ry ^ Rz	BINARY	R
SLT	6'b000000	6'b100000	1 if Rx <Ry	BINARY	R
ADDI	6'b001000	-	Rx = Ry + Imm	BINARY	I
SLTI	6'b001010	-	Rx = 1 if Ry <Imm	BINARY	I
ANDI	6'b001100	-	Rx = Ry & Imm	BINARY	I
ORI	6'b001101	-	Rx = Ry | Imm	BINARY	I
XORI	6'b001110	-	Rx = Ry ^Imm	BINARY	I
LW	6'b100011	-	Rx← Mem	BINARY	I
SW	6'b101011	-	Rx → Mem	BINARY	I
BEQ	6'b000100	-	Branch if Equal	BINARY	I
BNE	6'b000101	-	Branch if not Equal	BINARY	I
BLE	6'b000110	-	Branch if Lesser	BINARY	I
BGE	6'b000111	-	Branch if Greater	BINARY	I
J	6'b001010	-	Jump	-	J
ABS	6'b000001	6'b000001	| Rx |	UNARY	R
INVS	6'b000001	6'b000010	Invert Sign	UNARY	R
MOV	6'b000001	6'b000100	Rx ← Ry	BINARY	R
LS	6'b000001	6'b001000	Rx← Ry − 1	UNARY	R
RS	6'b000001	6'b010000	Rx← Ry − 1	UNARY	R
COMP	6'b000001	6'b100000	˜Rx	UNARY	R
INC	6'b000001	6'b100001	Rx++	UNARY	R
DCR	6'b000001	6'b100010	Rx—	UNARY	R

5 Simulation and Results

5.1 Introduction

The processor is designed with the ISA given in the previous chapter and it is implemented on the Artix - 7 family **xc7a100tcsg324-1**. The block diagram, design parameters and the schematic are given below. The processor is tested with 5 programs of different levels and the performances of the processor with without DF and BTB are tabulated.

5.2 Features of the Processor

Name	**Representation**	**Size / Value**	**Number**
Program Counter	PC	32'b (unsigned)	1
Registers	Reg[i]	32'b (signed)	32
Instruction Memory	Ins[i]	32'b (unsigned)	1024
Data Memory	Dat[i]	32'b (signed)	1024
Pipeline Registers	-	(unsigned)	4
Control Registers	-	(unsigned)	3
Data Forwarding	-	Applicable	-
BTB	-	Applicable	-
ISA	-	-	26

- All Instruction and Data Memories are **WORD Addressable**.

5.3 Schematic

The Schematic of the Implementation is given below:

5.4 Results

These are the results for the programs tested on this processor, the programs tested along with the complete processor code is given in Appendix section of this report. Now coming to the results we got :

Level	Program Name	Stalling (cycles)	Data Forwarding (cycles)	DF and BTB (cycles)
1	SWAPPING	18	10	10
2	MULTIPLICATION (5*10)	123	55	43
2	FACTORIAL (5!)	189	71	65
3	POWER X,Y (5 ^10)	654	334	282
4	SORTING AN ARRAY (6 elmts)	882	456	378

When we observe the performances in terms of a bar graph:

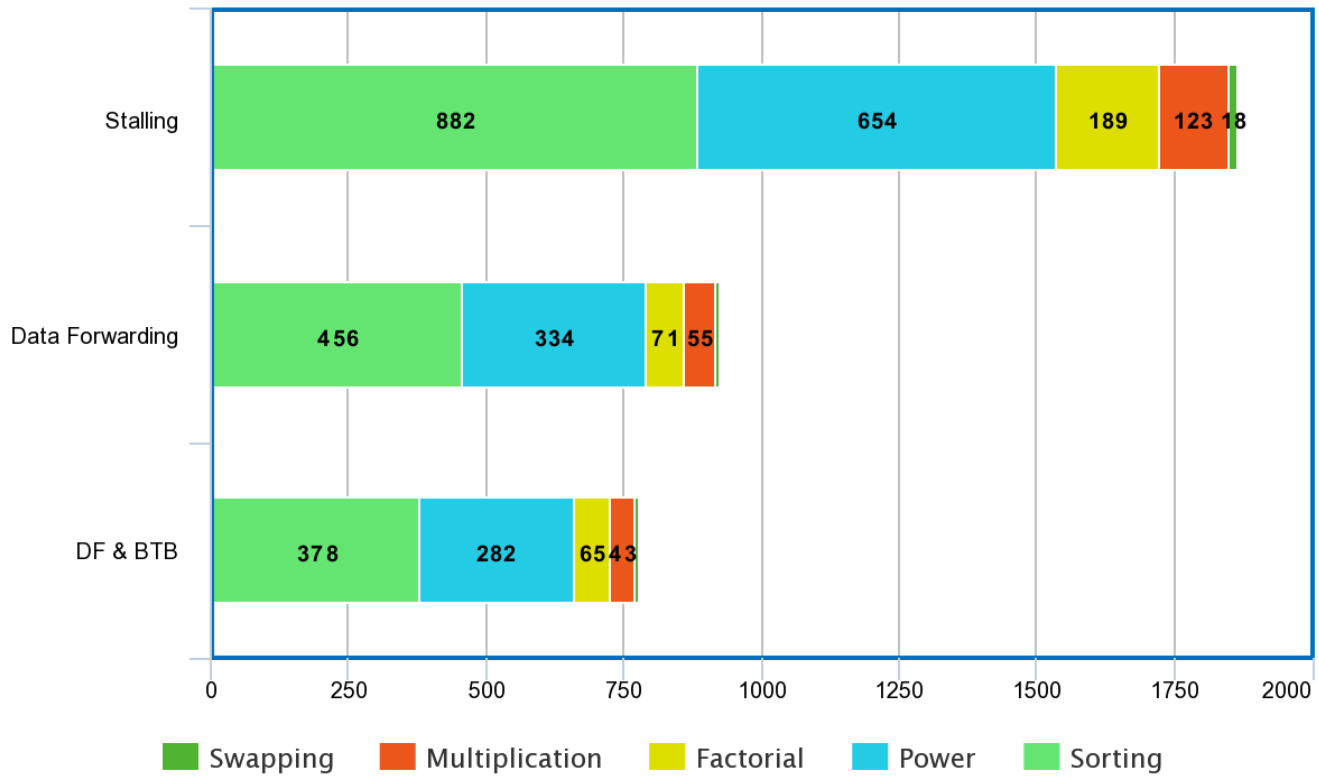

Figure 8: Comparison of Results

5.5 Conclusion From the Results

It is obvious from the results that **Data Forwarding** is must to a processor to save a lot of clock cycles. So with Data Forwarding all the data hazards are resolved but the load with suffer with 1 stall. That's why we can observe a huge difference between Stalling and Data Forwarding. But when it comes to the **DF** and **DF with BTB**, there is comparitively less improvement because the programs used has less number of loops, BTB is ideal for a programs containing loops that run for thousands of times.

6 Conclusion and Future Work

6.1 Conclusion

In this project we have created a **5- Stage Multi Cycle MIPS** processor by dividing the stages of a Single Cycle Processor by the concept of **Pipelining**. Later we started to resolve the Hazards like **Data Hazards and Control Hazards** with the help of **Data Forwarding** and a **Branch Target Buffer**. At the end we verified the processor with 6 programs of different difficulty level and observed the performance of Multi Cycle Processor before and after resolving these pipeline hazards.

In this long journey of making this processor, we have learnt a lot of insights about current day processors. Even though MIPS is a more like a Starting Point and there is a lot left to do, but this [U+FB01]rst step made us feel con[U+FB01]dent and ready for big things like Multi Core's, Graphics, Memories and a lot more.

6.2 Future Work

As a Future work, we would like to implement **Floating Point Arithmetic** and **Multi Cycle DLX** to our processor making it Super Scalar Architecture.

7 References

[1] Lecture Slides from Moodle.

[2] NPTEL Course on HPCA from IIT Kharagpur. Link

[3] Advanced Topic: An Introduction to Digital Design Using a Hardware Design Language to Describe and Model a Pipeline and More Pipelining Illustrations, Link

[4] Computer Organization and Design by David A. Patterson and John L.Hennessy. Link

8 Appendix

8.1 Program Code

```
‘timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 30.05.2020 18:58:48
// Design Name:
// Module Name: main
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module processor(
    input clock,
    output reg [31:0] Result
    );
    // Defining all the Registers
    reg [31:0] PC, NewPC;
    // Pipeline Registers
    reg signed [63:0] IFID;
    reg signed [132:0] IDEX;
    reg signed [68:0] EXMEM;
    reg signed [36:0] MEMWB;
    // Registers to Store Control Signals
    reg [14:0]EX;
    reg [3:0]MEM;
    reg WB;
    // Branch Target Buffer
    reg [31:0] BTB [0:9];
    reg [1:0] pred [0:9];

    // Defining Data Memory
    reg [31:0]InsMem[0:1023], DatMem[0:1023];
    reg signed [31:0]Reg[0:31];
```

```
// Others like Hazard Detection
reg PCSrc, Hazard, StallID, StallEX, StallMem, StallWB;
integer clkcycle;
reg decision, decisionID;

reg UseTheResultA,UseTheResultB;
initial
    begin
        // Program Counter
        PC = 32'h00000000;

        // Instruction Memory
        InsMem[0] = 32'b00000000011000100000100000000000; //ADD R1 R2 R3
        InsMem[1] = 32'b00000000001000100001100000000000; //ADD R3 R1 R2


        // Data Memory

        // Initilization of Registers
        Reg[0] = 32'b00000000000000000000000000000001;
        Reg[1] = 32'b00000000000000000000000000000010;
        Reg[2] = 32'b00000000000000000000000000000011;
        Reg[3] = 32'b00000000000000000000000000000101;

        // Initialize Pipeline Registers
        IFID = 64'b0;
        IDEX = 133'b0;
        EXMEM = 69'b0;
        MEMWB = 37'b0;

        // Initialize Control Registers
        EX = 15'b0;
        MEM = 4'b0;
        WB = 1'b0;
        PCSrc = 1'b0;


        // Initialize Stall Signals
        StallID = 1'b1;
        StallEX = 1'b1;
        StallMem = 1'b1;
        StallWB = 1'b1;
        clkcycle = 0;
        UseTheResultA = 1'b0;
        UseTheResultB = 1'b0;


    end

always@(posedge clock)
```

```
 begin
     // Instruction Fetch and BTB
     IFID[63:32] <= InsMem[PC]; // Get the Instruction
     IFID[31:0] <= PC; // Copy the PC
     PC <= PCSrc?NewPC:PC+1;  // Update PC

     PCSrc <= 1'b0;           // Reset PCSrc Signal
     StallID <= 1'b0;         // Reset all Stall Signals
     StallEX <= 1'b0;
     StallMem <= 1'b0;
     StallWB <= 1'b0;
     UseTheResultA <= 1'b0;
     UseTheResultB <= 1'b0;
     clkcycle = clkcycle + 1; // Increment Clk Cycle Number
     decision <= 1'b1;

     $display("%d, %d", Reg[3],UseTheResultA);
 // Check the BTB
 if(BTB[PC[9:0]] != 32'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx) // check if it is
    dont care or not
begin

   if(pred[PC[9:0]] >2'b01)
      begin
         // predict Taken
         PCSrc <=1'b1;
         NewPC <= BTB[PC[9:0]];
         decision <= 1'b1; // Taken
      end
   else
      begin
         // predict not taken
         decision <= 1'b0; // Not Taken
      end
end

     // Instruction Decode and Jump

     if(StallID == 1'b0)
         begin
                IDEX[31:0] <= IFID[31:0];     // Copy the PC
                IDEX[63:32] <= IFID[63:32];  // Copy the Instruction
                IDEX[95:64] <= Reg[IFID[57:53]]; // Get the Rs Data
                decisionID <= decision;
                // R type Instructions
                if(IFID[63:58] == 6'b000000 | IFID[63:58] == 6'b000001)
                    begin
                        IDEX[127:96] <= Reg[IFID[52:48]]; // BUS B
                        IDEX[132:128]<= IFID[47:43];     // Write Back
                           Register
```

```
    end
else
    begin
        // J type Instructions
        if(IFID[63:58] == 6'b000010)
            begin
                // JUMP Instruction
                NewPC <= {{6{1'b0}},IFID[47:32]};
                PCSrc <= 1'b1;
                StallID <=1'b1;
            end
    end
// DATA FORWARDING in Registers
// BUS A
if(IFID[63:58]!= 6'b001010)
    begin
        if(IFID[57:53] == IDEX[132:128] & IFID[61] == 1'b0 &
            IFID[63:58] != 6'b010110)
            // Data Dependency with Immediate Instruction
            begin
                UseTheResultA <= 1'b1;
            end
        else if(IFID[57:53] == EXMEM[68:64])
            // Data Dependency with Second Immediate
                Instruction
            begin
                IDEX[95:64] <= EXMEM[31:0];
            end
        else if(IFID[57:53] == MEMWB[36:32])
            // Data Dependency with Third Immediate
                Instruction
            begin
                IDEX[95:64] <= MEMWB[31:0];
            end
    end
// BUS B
if(IFID[63:58] == 6'b000000)
    begin
        if(IFID[52:48] == IDEX[132:128] & IFID[61] == 1'b0 &
            IFID[63:58] != 6'b010110)
            // Data Dependency with Immediate Instruction
            begin
                UseTheResultB <= 1'b1;
            end
        else if(IFID[52:48] == EXMEM[68:64])
            // Data Dependency with Second Immediate
                Instruction
            begin
                IDEX[127:96] <= EXMEM[31:0];
            end
```

```
            else if(IFID[52:48] == MEMWB[36:32])
                // Data Dependency with Third Immediate
                    Instruction
                begin
                    IDEX[127:96] <= MEMWB[31:0];
                end
        end
    if(IFID[63:58] ==6'b000000)
        begin
            if(IFID[37:32] == 6'b000000)
                begin
                    EX[14:0] <=15'b001000000000000; // ADD
                end
            if(IFID[37:32] == 6'b000010)
                begin
                    EX[14:0] <=15'b001000000000100; // SUB
                end
            if(IFID[37:32] == 6'b000100)
                begin
                    EX[14:0] <= 15'b001000000001100; // AND
                end
            if(IFID[37:32] == 6'b001000)
                begin
                    EX[14:0] <= 15'b001000000010000; // OR
                end
            if(IFID[37:32] == 6'b010000)
                begin
                    EX[14:0] <= 15'b001000000010100; // XOR
                end
            if(IFID[37:32] == 6'b100000)
                begin
                    EX[14:0] <= 15'b001000000011000; // SLT
                end
        end

    else if(IFID[63:58] ==6'b000001)
            begin
                if(IFID[37:32] == 6'b000001)
                    begin
                        EX[14:0] <= 15'b001000000100000;
                            //ABSOLUTE
                    end
                if(IFID[37:32] == 6'b000010)
                    begin
                        EX[14:0] <= 15'b001000000100100; //INVERT
                            SIGN
                    end
                if(IFID[37:32] == 6'b000100)
                    begin
```

```
                    EX[14:0] <= 15'b001000000101100; //COPY
                        REG DATA
                end
            if(IFID[37:32] == 6'b001000)
                begin
                    EX[14:0] <= 15'b001000000110000; //LEFT
                        SHIFT
                end
            if(IFID[37:32] == 6'b010000)
                begin
                    EX[14:0] <= 15'b001000000110100; //RIGHT
                        SHIFT
                end
            if(IFID[37:32] == 6'b100001)
                begin
                    EX[14:0] <= 15'b001000000111000;
                        //INCREMENT
                end
            if(IFID[37:32] == 6'b100010)
                begin
                    EX[14:0] <= 15'b001000000111100;
                        //DECREMENT
                end
            if(IFID[37:32] == 6'b100011)
                begin
                    EX[14:0] <= 15'b001000000100100;
                        //DECREMENT
                end
        end
    else
        begin
            if(IFID[63:58] ==6'b010000) //ADD IMMEDIATE
                begin
                    EX[14:0] <= 15'b001000000000011;
                end

            if(IFID[63:58] ==6'b010001) //SUB IMMEDIATE
                begin
                    EX[14:0] <= 15'b001000000000111;
                end

            if(IFID[63:58] ==6'b010010) //SLT IMMEDIATE
                begin
                    EX[14:0] <= 15'b001000000011011;
                end
            if(IFID[63:58] ==6'b010011) //AND IMMEDIATE
                begin
                    EX[14:0] <= 15'b001000000001110;
                end
            if(IFID[63:58] ==6'b010100) //OR IMMEDIATE
```

```
                    begin
                        EX[14:0] <= 15'b001000000010010;
                    end
                if(IFID[63:58] ==6'b010101) //XOR IMMEDIATE
                    begin
                        EX[14:0] <= 15'b001000000010110;
                    end
                if(IFID[63:58] ==6'b010110) //LOAD WORD
                    begin
                        EX[14:0] <= 15'b001010000000010;
                    end
                if(IFID[63:58] ==6'b011111) //STORE WORD
                    begin
                        EX[14:0] <= 15'b000100000000010;
                    end
                if(IFID[63:58] ==6'b001000) //BRANCH IF EQUAL
                    begin
                        EX[14:0] <= 15'b000000010000000;
                    end
                if(IFID[63:58] ==6'b001001) //BRANCH NOT EQUAL
                    begin
                        EX[14:0] <= 15'b000000001000000;
                    end
                if(IFID[63:58] ==6'b001010) //JUMP
                    begin
                        EX[14:0] <= 15'b000000100000001;
                    end
                if(IFID[63:58] ==6'b001011) //BRANCH IF LESS
                    begin
                        EX[14:0] <= 15'b010000000000001;
                    end
                if(IFID[63:58] ==6'b001100) //BRANCH IF GREATER
                    begin
                        EX[14:0] <= 15'b100000000000001;
                    end
            end
    end
// Execute and Hazard Detect
if(StallEX == 1'b0)
    begin
    if(StallID == 1'b1)
        begin
            StallEX <= 1'b1;
        end
    // Copy the Control Signals


    MEM[2:0] <= EX[11:9];
    MEM[3] <= EX[12];
    // Copy the Pipeline Registers
```

```
EXMEM[68:64] <= IDEX[132:128];
EXMEM[63:32] <= IDEX[127:96];
// Start the Execution Unit
if(UseTheResultB ==1'b1 & UseTheResultA == 1'b1)
    begin
        case(EX[5:2])
          4'b0000: EXMEM[31:0] <= EXMEM[31:0] + EXMEM[31:0]; // ADD
          4'b0001: EXMEM[31:0] <= EXMEM[31:0] - EXMEM[31:0]; // SUB
          4'b0011: EXMEM[31:0] <= EXMEM[31:0] & EXMEM[31:0]; // AND
          4'b0100: EXMEM[31:0] <= EXMEM[31:0] | EXMEM[31:0]; // OR
          4'b0101: EXMEM[31:0] <= EXMEM[31:0] ^ EXMEM[31:0]; // XOR
          4'b0110: EXMEM[31:0] <= EXMEM[31:0] | EXMEM[31:0]; // SLT
          4'b1001: EXMEM[31:0] <= {{1{~EXMEM[31]}}, ~EXMEM[30:0]}
              + 1; // INVERT SIGN
          4'b1000: EXMEM[31:0]
              <=EXMEM[31]?{{1{1'b0}},~EXMEM[30:0]}+1:EXMEM[31:0];
          4'b1011: EXMEM[31:0] <= EXMEM[31:0]; // MOV
          4'b1100: EXMEM[31:0] <= EXMEM[31:0]<< 1; // LS
          4'b1101: EXMEM[31:0] <= EXMEM[31:0]>> 1; // RS
          4'b1110: EXMEM[31:0] <= EXMEM[31:0] + 1; // INC
          4'b1111: EXMEM[31:0] <= EXMEM[31:0] - 1; // DCR
          4'b1001: EXMEM[31:0] <= Reg[EXMEM[31:0]]; //ROR
        endcase
       // Branch If Equal
            if(EX[7])
                      begin
       if(decisionID == 1'b0) // That means we haven't taken the
           Branch
    begin
     // This is wrong decision and we update the predict bits
     if(pred[IDEX[9:0]] == 2'b10)
        begin
           pred[IDEX[41:32]] <=2'b00;
        end
     if(pred[IDEX[9:0]] == 2'b11)
        begin
           pred[IDEX[41:32]] <=2'b10;
        end
        // we have Fetched and Decoded wrong Instructions, we need
            to flush them
                            NewPC <= IDEX[31:0] + {{16{1'b0}},
                                IDEX[47:32]};
                            PCSrc <= 1'b1;
                            StallID <= 1'b1;
                            StallEX <= 1'b1;
  end
     else if (decisionID == 1'b1)
    begin
     // This is a correct decision and we update the predict bits
     if(pred[IDEX[9:0]] == 2'b00)
```

```
                begin
                   pred[IDEX[41:32]] <=2'b01;
                end
             if(pred[IDEX[9:0]] == 2'b01)
                begin
                   pred[IDEX[41:32]] <=2'b11;
                end
          end
             else
            begin
             // This is a new entry
             BTB[IDEX[9:0]] <= IDEX[31:0] + {{16{1'b0}}, IDEX[47:32]};
             pred[IDEX[9:0]] <= 2'b11;
                                  NewPC <= IDEX[31:0] + {{16{1'b0}},
                                      IDEX[47:32]};
                                  PCSrc <= 1'b1;
                                  StallID <= 1'b1;
                                  StallEX <= 1'b1;
          end
                           end
end
          end

      else if(UseTheResultA == 1'b1)
          begin

              case(EX[5:2])
                4'b0000: EXMEM[31:0] <= EXMEM[31:0] + IDEX[127:96]; //
                    ADD
                4'b0001: EXMEM[31:0] <= EXMEM[31:0] - IDEX[127:96]; //
                    SUB
                4'b0011: EXMEM[31:0] <= EXMEM[31:0] & IDEX[127:96]; //
                    AND
                4'b0100: EXMEM[31:0] <= EXMEM[31:0] | IDEX[127:96]; // OR
                4'b0101: EXMEM[31:0] <= EXMEM[31:0] ^ IDEX[127:96]; //
                    XOR
                4'b0110: EXMEM[31:0] <= EXMEM[31:0] | IDEX[127:96]; //
                    SLT
                4'b1001: EXMEM[31:0] <= {{1{~EXMEM[31]}}, ~EXMEM[30:0]}
                    + 1; // INVERT SIGN
                4'b1000: EXMEM[31:0] <= EXMEM[31]?{{1{1'b0}},
                    ~EXMEM[30:0]}+1:EXMEM[31:0]; // ABS
                4'b1011: EXMEM[31:0] <= EXMEM[31:0]; // MOV
                4'b1100: EXMEM[31:0] <= EXMEM[31:0]<< 1; // LS
                4'b1101: EXMEM[31:0] <= EXMEM[31:0]>> 1; // RS
                4'b1110: EXMEM[31:0] <= EXMEM[31:0] + 1; // INC
                4'b1111: EXMEM[31:0] <= EXMEM[31:0] - 1; // DCR
                4'b1001: EXMEM[31:0] <= Reg[EXMEM[31:0]]; //ROR
             endcase
```

```
    // Branch If Equal
    if(EX[7])
          begin
              if(EXMEM[31:0] == IDEX[127:96])
                  begin
    if(decisionID == 1'b0) // That means we haven't taken the
        Branch
  begin
   // This is wrong decision and we update the predict bits
   if(pred[IDEX[9:0]] == 2'b10)
      begin
         pred[IDEX[41:32]] <=2'b00;
      end
   if(pred[IDEX[9:0]] == 2'b11)
      begin
         pred[IDEX[41:32]] <=2'b10;
      end
      // we have Fetched and Decoded wrong Instructions, we need
          to flush them
                          NewPC <= IDEX[31:0] + {{16{1'b0}},
                              IDEX[47:32]};
                          PCSrc <= 1'b1;
                          StallID <= 1'b1;
                          StallEX <= 1'b1;

end
   else if (decisionID == 1'b1)
  begin
   // This is a correct decision and we update the predict bits:
   if(pred[IDEX[9:0]] == 2'b00)
      begin
         pred[IDEX[41:32]] <=2'b01;
      end
   if(pred[IDEX[9:0]] == 2'b01)
      begin
         pred[IDEX[41:32]] <=2'b11;
      end
end
   else
  begin
   // This is a new entry
   BTB[IDEX[9:0]] <= IDEX[31:0] + {{16{1'b0}}, IDEX[47:32]};
   pred[IDEX[9:0]] <= 2'b00;
                          NewPC <= IDEX[31:0] + {{16{1'b0}},
                              IDEX[47:32]};
                          PCSrc <= 1'b1;
                          StallID <= 1'b1;
                          StallEX <= 1'b1;
  end
                  end
```

```
else
    // Now that means taking this branch is a Mistake
                    begin

        if(decisionID == 1'b0) // That means we haven't taken the
            Branch
      begin
       // This is Correct decision and we update the predict bits
       if(pred[IDEX[9:0]] == 2'b00)
          begin
             pred[IDEX[41:32]] <=2'b01;
          end
       if(pred[IDEX[9:0]] == 2'b01)
          begin
             pred[IDEX[41:32]] <=2'b11;
          end
     end
        else if (decisionID == 1'b1)
      begin
       // This is a wrong decision and we update the predict bits
       // since we have Fetched and Decoded wrong Instructions, we
           need to flush them

       if(pred[IDEX[9:0]] == 2'b10)
          begin
             pred[IDEX[41:32]] <=2'b00;
          end
       if(pred[IDEX[9:0]] == 2'b11)
          begin
             pred[IDEX[41:32]] <=2'b10;
          end
                            NewPC <= IDEX[31:0] + 1;
                            PCSrc <= 1'b1;
                            StallID <= 1'b1;
                            StallEX <= 1'b1;
     end
                    end
            end

        // Branch If Not Equal

        if(EX[6])
              begin
                  if(EXMEM[31:0] != IDEX[127:96])
                      begin
        if(decisionID == 1'b0) // That means we haven't taken the
            Branch
      begin
       // This is wrong decision and we update the predict bits
       if(pred[IDEX[9:0]] == 2'b10)
```

```
                begin
                   pred[IDEX[41:32]] <=2'b00;
                end
             if(pred[IDEX[9:0]] == 2'b11)
                begin
                   pred[IDEX[41:32]] <=2'b10;
                end
                // we have Fetched and Decoded wrong Instructions, we need
                    to flush them
                                NewPC <= IDEX[31:0] + {{16{1'b0}},
                                    IDEX[47:32]};
                                PCSrc <= 1'b1;
                                StallID <= 1'b1;
                                StallEX <= 1'b1;

          end
             else if (decisionID == 1'b1)
            begin
             // This is a correct decision and we update the predict bits
             if(pred[IDEX[9:0]] == 2'b00)
                begin
                   pred[IDEX[41:32]] <=2'b01;
                end
             if(pred[IDEX[9:0]] == 2'b01)
                begin
                   pred[IDEX[41:32]] <=2'b11;
                end
          end
             else
            begin
             // This is a new entry
             BTB[IDEX[9:0]] <= IDEX[31:0] + {{16{1'b0}}, IDEX[47:32]};
             pred[IDEX[9:0]] <= 2'b00;
                                NewPC <= IDEX[31:0] + {{16{1'b0}},
                                    IDEX[47:32]};
                                PCSrc <= 1'b1;
                                StallID <= 1'b1;
                                StallEX <= 1'b1;
            end
                        end
    else
        // Now that means taking this branch is a Mistake
                        begin

             if(decisionID == 1'b0) // That means we haven't taken the
                 Branch
            begin
             // This is Correct decision and we update the predict bits
             if(pred[IDEX[9:0]] == 2'b00)
                begin
```

```
                pred[IDEX[41:32]] <=2'b01;
              end
            if(pred[IDEX[9:0]] == 2'b01)
              begin
                pred[IDEX[41:32]] <=2'b11;
              end
        end
           else if (decisionID == 1'b1)
          begin
           // This is a wrong decision and we update the predict bits
           // since we have Fetched and Decoded wrong Instructions, we
               need to flush them

           if(pred[IDEX[9:0]] == 2'b10)
              begin
                pred[IDEX[41:32]] <=2'b00;
              end
           if(pred[IDEX[9:0]] == 2'b11)
              begin
                pred[IDEX[41:32]] <=2'b10;
              end
                                NewPC <= IDEX[31:0] + 1;
                                PCSrc <= 1'b1;
                                StallID <= 1'b1;
                                StallEX <= 1'b1;
        end
                          end

end
            // Branch if Lesser
            if(EX[13])
                  begin
                      if(EXMEM[31:0] < IDEX[127:96])
                          begin
            if(decisionID == 1'b0) // That means we haven't taken the
               Branch
          begin
           // This is wrong decision and we update the predict bits
           if(pred[IDEX[9:0]] == 2'b10)
              begin
                pred[IDEX[41:32]] <=2'b00;
              end
           if(pred[IDEX[9:0]] == 2'b11)
              begin
                pred[IDEX[41:32]] <=2'b10;
              end
              // we have Fetched and Decoded wrong Instructions, we need
                  to flush them
                                NewPC <= IDEX[31:0] + {{16{1'b0}},
                                    IDEX[47:32]};
```

```
                                PCSrc <= 1'b1;
                                StallID <= 1'b1;
                                StallEX <= 1'b1;

        end
           else if (decisionID == 1'b1)
          begin
           // This is a correct decision and we update the predict bits
           if(pred[IDEX[9:0]] == 2'b00)
              begin
                 pred[IDEX[41:32]] <=2'b01;
              end
           if(pred[IDEX[9:0]] == 2'b01)
              begin
                 pred[IDEX[41:32]] <=2'b11;
              end
        end
           else
          begin
           // This is a new entry
           BTB[IDEX[9:0]] <= IDEX[31:0] + {{16{1'b0}}, IDEX[47:32]};
           pred[IDEX[9:0]] <= 2'b00;
                                NewPC <= IDEX[31:0] + {{16{1'b0}},
                                    IDEX[47:32]};
                                PCSrc <= 1'b1;
                                StallID <= 1'b1;
                                StallEX <= 1'b1;
          end
                           end
else
     // Now that means taking this branch is a Mistake
                           begin

            if(decisionID == 1'b0) // That means we haven't taken the
                Branch
          begin
           // This is Correct decision and we update the predict bits
           if(pred[IDEX[9:0]] == 2'b00)
              begin
                 pred[IDEX[41:32]] <=2'b01;
              end
           if(pred[IDEX[9:0]] == 2'b01)
              begin
                 pred[IDEX[41:32]] <=2'b11;
              end
        end
           else if (decisionID == 1'b1)
          begin
           // This is a wrong decision and we update the predict bits
```

```
                // since we have Fetched and Decoded wrong Instructions, we
                    need to flush them

                if(pred[IDEX[9:0]] == 2'b10)
                   begin
                      pred[IDEX[41:32]] <=2'b00;
                   end
                if(pred[IDEX[9:0]] == 2'b11)
                   begin
                      pred[IDEX[41:32]] <=2'b10;
                   end
                                        NewPC <= IDEX[31:0] + 1;
                                        PCSrc <= 1'b1;
                                        StallID <= 1'b1;
                                        StallEX <= 1'b1;
            end
                              end

    end
                // Branch if Greater
                if(EX[14])
                      begin
                          if(EXMEM[31:0] > IDEX[127:96])
                              begin
                if(decisionID == 1'b0) // That means we haven't taken the
                    Branch
              begin
                // This is wrong decision and we update the predict bits
                if(pred[IDEX[9:0]] == 2'b10)
                   begin
                      pred[IDEX[41:32]] <=2'b00;
                   end
                if(pred[IDEX[9:0]] == 2'b11)
                   begin
                      pred[IDEX[41:32]] <=2'b10;
                   end
                   // we have Fetched and Decoded wrong Instructions, we need
                       to flush them
                                        NewPC <= IDEX[31:0] + {{16{1'b0}},
                                            IDEX[47:32]};
                                        PCSrc <= 1'b1;
                                        StallID <= 1'b1;
                                        StallEX <= 1'b1;

            end
                else if (decisionID == 1'b1)
              begin
                // This is a correct decision and we update the predict bits
                if(pred[IDEX[9:0]] == 2'b00)
                   begin
```

```
                pred[IDEX[41:32]] <=2'b01;
              end
            if(pred[IDEX[9:0]] == 2'b01)
              begin
                pred[IDEX[41:32]] <=2'b11;
              end
          end
            else
           begin
            // This is a new entry
            BTB[IDEX[9:0]] <= IDEX[31:0] + {{16{1'b0}}, IDEX[47:32]};
            pred[IDEX[9:0]] <= 2'b00;
                                NewPC <= IDEX[31:0] + {{16{1'b0}},
                                    IDEX[47:32]};
                                PCSrc <= 1'b1;
                                StallID <= 1'b1;
                                StallEX <= 1'b1;
           end
                           end
    else
        // Now that means taking this branch is a Mistake
                           begin

            if(decisionID == 1'b0) // That means we haven't taken the
                Branch
           begin
            // This is Correct decision and we update the predict bits
            if(pred[IDEX[9:0]] == 2'b00)
              begin
                pred[IDEX[41:32]] <=2'b01;
              end
            if(pred[IDEX[9:0]] == 2'b01)
              begin
                pred[IDEX[41:32]] <=2'b11;
              end
          end
            else if (decisionID == 1'b1)
           begin
            // This is a wrong decision and we update the predict bits
            // since we have Fetched and Decoded wrong Instructions, we
                need to flush them

            if(pred[IDEX[9:0]] == 2'b10)
              begin
                pred[IDEX[41:32]] <=2'b00;
              end
            if(pred[IDEX[9:0]] == 2'b11)
              begin
                pred[IDEX[41:32]] <=2'b10;
              end
```

```
                                NewPC <= IDEX[31:0] + 1;
                                PCSrc <= 1'b1;
                                StallID <= 1'b1;
                                StallEX <= 1'b1;
    end
                        end
                end
        end
    else if(UseTheResultB == 1'b1)
        begin

            case(EX[5:2])
              4'b0000: EXMEM[31:0] <= IDEX[95:64] + EXMEM[31:0]; // ADD
              4'b0001: EXMEM[31:0] <= IDEX[95:64] - EXMEM[31:0]; // SUB
              4'b0011: EXMEM[31:0] <= IDEX[95:64] & EXMEM[31:0]; // AND
              4'b0100: EXMEM[31:0] <= IDEX[95:64] | EXMEM[31:0]; // OR
              4'b0101: EXMEM[31:0] <= IDEX[95:64] ^ EXMEM[31:0]; // XOR
              4'b0110: EXMEM[31:0] <= IDEX[95:64] | EXMEM[31:0]; // SLT
              4'b1001: EXMEM[31:0] <= {{1{~IDEX[95]}}, ~IDEX[94:64]} +
                  1; // INVERT SIGN
              4'b1000: EXMEM[31:0] <= IDEX[95]?{{1{1'b0}},
                  ~IDEX[94:64]}+1:IDEX[95:64]; // ABS
              4'b1011: EXMEM[31:0] <= IDEX[95:64]; // MOV
              4'b1100: EXMEM[31:0] <= IDEX[95:64]<< 1; // LS
              4'b1101: EXMEM[31:0] <= IDEX[95:64]>> 1; // RS
              4'b1110: EXMEM[31:0] <= IDEX[95:64] + 1; // INC
              4'b1111: EXMEM[31:0] <= IDEX[95:64] - 1; // DCR
              4'b1001: EXMEM[31:0] <= Reg[IDEX[95:64]]; //ROR
            endcase
           // Branch If Equal
           if(EX[7])
                 begin
                    if(IDEX[95:64] == EXMEM[31:0])
                        begin
           if(decisionID == 1'b0) // That means we haven't taken the
               Branch
          begin
           // This is wrong decision and we update the predict bits
           if(pred[IDEX[9:0]] == 2'b10)
              begin
                 pred[IDEX[41:32]] <=2'b00;
              end
           if(pred[IDEX[9:0]] == 2'b11)
              begin
                 pred[IDEX[41:32]] <=2'b10;
              end
              // we have Fetched and Decoded wrong Instructions, we need
                  to flush them
                                NewPC <= IDEX[31:0] + {{16{1'b0}},
                                    IDEX[47:32]};
```

```
                                PCSrc <= 1'b1;
                                StallID <= 1'b1;
                                StallEX <= 1'b1;

        end
           else if (decisionID == 1'b1)
          begin
           // This is a correct decision and we update the predict bits
           if(pred[IDEX[9:0]] == 2'b00)
              begin
                 pred[IDEX[41:32]] <=2'b01;
              end
           if(pred[IDEX[9:0]] == 2'b01)
              begin
                 pred[IDEX[41:32]] <=2'b11;
              end
        end
           else
          begin
           // This is a new entry
           BTB[IDEX[9:0]] <= IDEX[31:0] + {{16{1'b0}}, IDEX[47:32]};
           pred[IDEX[9:0]] <= 2'b00;
                                NewPC <= IDEX[31:0] + {{16{1'b0}},
                                    IDEX[47:32]};
                                PCSrc <= 1'b1;
                                StallID <= 1'b1;
                                StallEX <= 1'b1;
          end
                           end
else
     // Now that means taking this branch is a Mistake
                           begin

            if(decisionID == 1'b0) // That means we haven't taken the
                Branch
          begin
           // This is Correct decision and we update the predict bits
           if(pred[IDEX[9:0]] == 2'b00)
              begin
                 pred[IDEX[41:32]] <=2'b01;
              end
           if(pred[IDEX[9:0]] == 2'b01)
              begin
                 pred[IDEX[41:32]] <=2'b11;
              end
        end
           else if (decisionID == 1'b1)
          begin
           // This is a wrong decision and we update the predict bits
```

```
   // since we have Fetched and Decoded wrong Instructions, we
       need to flush them

   if(pred[IDEX[9:0]] == 2'b10)
      begin
         pred[IDEX[41:32]] <=2'b00;
      end
   if(pred[IDEX[9:0]] == 2'b11)
      begin
         pred[IDEX[41:32]] <=2'b10;
      end
                           NewPC <= IDEX[31:0] + 1;
                           PCSrc <= 1'b1;
                           StallID <= 1'b1;
                           StallEX <= 1'b1;
end
                  end
          end

    // Branch If Not Equal

    if(EX[6])
          begin
              if(IDEX[95:64] != EXMEM[31:0])
                  begin
    if(decisionID == 1'b0) // That means we haven't taken the
       Branch
  begin
   // This is wrong decision and we update the predict bits
   if(pred[IDEX[9:0]] == 2'b10)
      begin
         pred[IDEX[41:32]] <=2'b00;
      end
   if(pred[IDEX[9:0]] == 2'b11)
      begin
         pred[IDEX[41:32]] <=2'b10;
      // we have Fetched and Decoded wrong Instructions, we need
          to flush them
                           NewPC <= IDEX[31:0] + {{16{1'b0}},
                              IDEX[47:32]};
                           PCSrc <= 1'b1;
                           StallID <= 1'b1;
                           StallEX <= 1'b1;

end
   else if (decisionID == 1'b1)
  begin
   // This is a correct decision and we update the predict bits
   if(pred[IDEX[9:0]] == 2'b00)
      begin
```

```
                pred[IDEX[41:32]] <=2'b01;
              end
            if(pred[IDEX[9:0]] == 2'b01)
              begin
                pred[IDEX[41:32]] <=2'b11;
              end
          end
            else
           begin
            // This is a new entry
            BTB[IDEX[9:0]] <= IDEX[31:0] + {{16{1'b0}}, IDEX[47:32]};
            pred[IDEX[9:0]] <= 2'b00;
                                NewPC <= IDEX[31:0] + {{16{1'b0}},
                                    IDEX[47:32]};
                                PCSrc <= 1'b1;
                                StallID <= 1'b1;
                                StallEX <= 1'b1;
           end
                           end
    else
        // Now that means taking this branch is a Mistake
                           begin

            if(decisionID == 1'b0) // That means we haven't taken the
                Branch
           begin
            // This is Correct decision and we update the predict bits
            if(pred[IDEX[9:0]] == 2'b00)
              begin
                pred[IDEX[41:32]] <=2'b01;
              end
            if(pred[IDEX[9:0]] == 2'b01)
              begin
                pred[IDEX[41:32]] <=2'b11;
              end
          end
            else if (decisionID == 1'b1)
           begin
            // This is a wrong decision and we update the predict bits
            // since we have Fetched and Decoded wrong Instructions, we
                need to flush them

            if(pred[IDEX[9:0]] == 2'b10)
              begin
                pred[IDEX[41:32]] <=2'b00;
              end
            if(pred[IDEX[9:0]] == 2'b11)
              begin
                pred[IDEX[41:32]] <=2'b10;
              end
```

```
                        NewPC <= IDEX[31:0] + 1;
                        PCSrc <= 1'b1;
                        StallID <= 1'b1;
                        StallEX <= 1'b1;
end
                end

          end
    // Branch if Lesser
    if(EX[13])
          begin
              if(IDEX[95:64] < EXMEM[31:0])
                  begin
    if(decisionID == 1'b0) // That means we haven't taken the
       Branch
  begin
   // This is wrong decision and we update the predict bits
   if(pred[IDEX[9:0]] == 2'b10)
      begin
         pred[IDEX[41:32]] <=2'b00;
      end
   if(pred[IDEX[9:0]] == 2'b11)
      begin
         pred[IDEX[41:32]] <=2'b10;
      end

      // we have Fetched and Decoded wrong Instructions, we need
          to flush them
                        NewPC <= IDEX[31:0] + {{16{1'b0}},
                            IDEX[47:32]};
                        PCSrc <= 1'b1;
                        StallID <= 1'b1;
                        StallEX <= 1'b1;

end
   else if (decisionID == 1'b1)
  begin
   // This is a correct decision and we update the predict bits
   if(pred[IDEX[9:0]] == 2'b00)
      begin
         pred[IDEX[41:32]] <=2'b01;
      end
   if(pred[IDEX[9:0]] == 2'b01)
      begin
         pred[IDEX[41:32]] <=2'b11;
      end
end
   else
  begin
   // This is a new entry
```

```
            BTB[IDEX[9:0]] <= IDEX[31:0] + {{16{1'b0}}, IDEX[47:32]};
            pred[IDEX[9:0]] <= 2'b00;
                                NewPC <= IDEX[31:0] + {{16{1'b0}},
                                    IDEX[47:32]};
                                PCSrc <= 1'b1;
                                StallID <= 1'b1;
                                StallEX <= 1'b1;
          end
                            end
    else
        // Now that means taking this branch is a Mistake
                            begin

           if(decisionID == 1'b0) // That means we haven't taken the
               Branch
          begin
           // This is Correct decision and we update the predict bits
           if(pred[IDEX[9:0]] == 2'b00)
              begin
                 pred[IDEX[41:32]] <=2'b01;
              end
           if(pred[IDEX[9:0]] == 2'b01)
              begin
                 pred[IDEX[41:32]] <=2'b11;
              end
        end
           else if (decisionID == 1'b1)
          begin
           // This is a wrong decision and we update the predict bits
           // since we have Fetched and Decoded wrong Instructions, we
               need to flush them

           if(pred[IDEX[9:0]] == 2'b10)
              begin
                 pred[IDEX[41:32]] <=2'b00;
              end
           if(pred[IDEX[9:0]] == 2'b11)
              begin
                 pred[IDEX[41:32]] <=2'b10;
              end
                                NewPC <= IDEX[31:0] + 1;
                                PCSrc <= 1'b1;
                                StallID <= 1'b1;
                                StallEX <= 1'b1;
        end
                            end
end
            // Branch if Greater
            if(EX[14])
                  begin
```

```
                    if(IDEX[95:64] > EXMEM[31:0])
                        begin
                 if(decisionID == 1'b0) // That means we haven't taken the
                     Branch
               begin
                // This is wrong decision and we update the predict bits
                if(pred[IDEX[9:0]] == 2'b10)
                   begin
                      pred[IDEX[41:32]] <=2'b00;
                   end
                if(pred[IDEX[9:0]] == 2'b11)
                   begin
                      pred[IDEX[41:32]] <=2'b10;
                   end

                   // we have Fetched and Decoded wrong Instructions, we need
                       to flush them
                                    NewPC <= IDEX[31:0] + {{16{1'b0}},
                                        IDEX[47:32]};
                                    PCSrc <= 1'b1;
                                    StallID <= 1'b1;
                                    StallEX <= 1'b1;

            end
               else if (decisionID == 1'b1)
               begin
                // This is a correct decision and we update the predict bits
                if(pred[IDEX[9:0]] == 2'b00)
                   begin
                      pred[IDEX[41:32]] <=2'b01;
                   end
                if(pred[IDEX[9:0]] == 2'b01)
                   begin
                      pred[IDEX[41:32]] <=2'b11;
                   end
            end
               else
               begin
                // This is a new entry
                BTB[IDEX[9:0]] <= IDEX[31:0] + {{16{1'b0}}, IDEX[47:32]};
                pred[IDEX[9:0]] <= 2'b00;
                                    NewPC <= IDEX[31:0] + {{16{1'b0}},
                                        IDEX[47:32]};
                                    PCSrc <= 1'b1;
                                    StallID <= 1'b1;
                                    StallEX <= 1'b1;
               end
                           end
    else
        // Now that means taking this branch is a Mistake
```

```
                    begin

     if(decisionID == 1'b0) // That means we haven't taken the
         Branch
   begin
    // This is Correct decision and we update the predict bits
    if(pred[IDEX[9:0]] == 2'b00)
       begin
          pred[IDEX[41:32]] <=2'b01;
       end
    if(pred[IDEX[9:0]] == 2'b01)
       begin
          pred[IDEX[41:32]] <=2'b11;
       end
 end
    else if (decisionID == 1'b1)
   begin
    // This is a wrong decision and we update the predict bits
    // since we have Fetched and Decoded wrong Instructions, we
        need to flush them

    if(pred[IDEX[9:0]] == 2'b10)
       begin
          pred[IDEX[41:32]] <=2'b00;
       end
    if(pred[IDEX[9:0]] == 2'b11)
       begin
          pred[IDEX[41:32]] <=2'b10;
       end
                          NewPC <= IDEX[31:0] + 1;
                          PCSrc <= 1'b1;
                          StallID <= 1'b1;
                          StallEX <= 1'b1;
 end
                    end
             end
   end

else
   begin
       case(EX[5:2])
             4'b0000: EXMEM[31:0] <= IDEX[95:64] + IDEX[127:96];
                // ADD
             4'b0001: EXMEM[31:0] <= IDEX[95:64] - IDEX[127:96];
                // SUB
             4'b0011: EXMEM[31:0] <= IDEX[95:64] & IDEX[127:96];
                // AND
             4'b0100: EXMEM[31:0] <= IDEX[95:64] | IDEX[127:96];
                // OR
```

```
                4'b0101: EXMEM[31:0] <= IDEX[95:64] ^ IDEX[127:96];
                    // XOR
                4'b0110: EXMEM[31:0] <= IDEX[95:64] | IDEX[127:96];
                    // SLT
                4'b1001: EXMEM[31:0] <= {{1{~IDEX[95]}},
                    ~IDEX[94:64]} + 1; // INVERT SIGN
                4'b1000: EXMEM[31:0] <= IDEX[95]?{{1{1'b0}},
                    ~IDEX[94:64]}+1:IDEX[95:64]; // ABS
                4'b1011: EXMEM[31:0] <= IDEX[95:64]; // MOV
                4'b1100: EXMEM[31:0] <= IDEX[95:64]<< 1; // LS
                4'b1101: EXMEM[31:0] <= IDEX[95:64]>> 1; // RS
                4'b1110: EXMEM[31:0] <= IDEX[95:64] + 1; // INC
                4'b1111: EXMEM[31:0] <= IDEX[95:64] - 1; // DCR
                4'b1001: EXMEM[31:0] <= Reg[IDEX[95:64]]; //ROR
            endcase
            // Branch If Equal //IDEX[95:64] == IDEX[127:96]
     // Branch If Equal
     if(EX[7])
           begin
               if(IDEX[95:64] == IDEX[127:96])
                   begin
     if(decisionID == 1'b0) // That means we haven't taken the
         Branch
   begin
     // This is wrong decision and we update the predict bits
     if(pred[IDEX[9:0]] == 2'b10)
        begin
           pred[IDEX[41:32]] <=2'b00;
        end
     if(pred[IDEX[9:0]] == 2'b11)
        begin
           pred[IDEX[41:32]] <=2'b10;
        end

        // we have Fetched and Decoded wrong Instructions, we need
            to flush them
                            NewPC <= IDEX[31:0] + {{16{1'b0}},
                                IDEX[47:32]};
                            PCSrc <= 1'b1;
                            StallID <= 1'b1;
                            StallEX <= 1'b1;

end
    else if (decisionID == 1'b1)
   begin
     // This is a correct decision and we update the predict bits
     if(pred[IDEX[9:0]] == 2'b00)
        begin
           pred[IDEX[41:32]] <=2'b01;
        end
```

```
            if(pred[IDEX[9:0]] == 2'b01)
               begin
                  pred[IDEX[41:32]] <=2'b11;
               end
         end
            else
           begin
            // This is a new entry
            BTB[IDEX[9:0]] <= IDEX[31:0] + {{16{1'b0}}, IDEX[47:32]};
            pred[IDEX[9:0]] <= 2'b00;
                                 NewPC <= IDEX[31:0] + {{16{1'b0}},
                                     IDEX[47:32]};
                                 PCSrc <= 1'b1;
                                 StallID <= 1'b1;
                                 StallEX <= 1'b1;
           end
                           end
else
     // Now that means taking this branch is a Mistake
                           begin

            if(decisionID == 1'b0) // That means we haven't taken the
                Branch
           begin
            // This is Correct decision and we update the predict bits
            if(pred[IDEX[9:0]] == 2'b00)
               begin
                  pred[IDEX[41:32]] <=2'b01;
               end
            if(pred[IDEX[9:0]] == 2'b01)
               begin
                  pred[IDEX[41:32]] <=2'b11;
               end
         end
            else if (decisionID == 1'b1)
           begin
            // This is a wrong decision and we update the predict bits
            // since we have Fetched and Decoded wrong Instructions, we
                need to flush them

            if(pred[IDEX[9:0]] == 2'b10)
               begin
                  pred[IDEX[41:32]] <=2'b00;
               end
            if(pred[IDEX[9:0]] == 2'b11)
               begin
                  pred[IDEX[41:32]] <=2'b10;
               end
                                 NewPC <= IDEX[31:0] + 1;
                                 PCSrc <= 1'b1;
```

```
                                StallID <= 1'b1;
                                StallEX <= 1'b1;
            end
                            end

    end
            // Branch If Not Equal

            if(EX[6])
                  begin
                      if(IDEX[95:64] != IDEX[127:96])
                          begin
            if(decisionID == 1'b0) // That means we haven't taken the
                Branch
          begin
           // This is wrong decision and we update the predict bits
           if(pred[IDEX[9:0]] == 2'b10)
              begin
                 pred[IDEX[41:32]] <=2'b00;
              end
           if(pred[IDEX[9:0]] == 2'b11)
              begin
                 pred[IDEX[41:32]] <=2'b10;
              end

              // we have Fetched and Decoded wrong Instructions, we need
                  to flush them
                                NewPC <= IDEX[31:0] + {{16{1'b0}},
                                    IDEX[47:32]};
                                PCSrc <= 1'b1;
                                StallID <= 1'b1;
                                StallEX <= 1'b1;

            end
              else if (decisionID == 1'b1)
             begin
              // This is a correct decision and we update the predict bits
              if(pred[IDEX[9:0]] == 2'b00)
                 begin
                    pred[IDEX[41:32]] <=2'b01;
                 end
              if(pred[IDEX[9:0]] == 2'b01)
                 begin
                    pred[IDEX[41:32]] <=2'b11;
                 end
            end
              else
             begin
              // This is a new entry
              BTB[IDEX[9:0]] <= IDEX[31:0] + {{16{1'b0}}, IDEX[47:32]};
```

```
            pred[IDEX[9:0]] <= 2'b00;
                                NewPC <= IDEX[31:0] + {{16{1'b0}},
                                    IDEX[47:32]};
                                PCSrc <= 1'b1;
                                StallID <= 1'b1;
                                StallEX <= 1'b1;
          end
                            end
    else
        // Now that means taking this branch is a Mistake
                            begin

           if(decisionID == 1'b0) // That means we haven't taken the
               Branch
          begin
           // This is Correct decision and we update the predict bits
           if(pred[IDEX[9:0]] == 2'b00)
              begin
                 pred[IDEX[41:32]] <=2'b01;
              end
           if(pred[IDEX[9:0]] == 2'b01)
              begin
                 pred[IDEX[41:32]] <=2'b11;
              end
        end
           else if (decisionID == 1'b1)
          begin
           // This is a wrong decision and we update the predict bits
           // since we have Fetched and Decoded wrong Instructions, we
               need to flush them

           if(pred[IDEX[9:0]] == 2'b10)
              begin
                 pred[IDEX[41:32]] <=2'b00;
              end
           if(pred[IDEX[9:0]] == 2'b11)
              begin
                 pred[IDEX[41:32]] <=2'b10;
              end
                                NewPC <= IDEX[31:0] + 1;
                                PCSrc <= 1'b1;
                                StallID <= 1'b1;
                                StallEX <= 1'b1;
        end
                            end

end
            // Branch if Lesser
            if(EX[13])
                  begin
```

```
                    if(IDEX[95:64] < IDEX[127:96])
                        begin
                 if(decisionID == 1'b0) // That means we haven't taken the
                     Branch
               begin
                // This is wrong decision and we update the predict bits
                if(pred[IDEX[9:0]] == 2'b10)
                   begin
                      pred[IDEX[41:32]] <=2'b00;
                   end
                if(pred[IDEX[9:0]] == 2'b11)
                   begin
                      pred[IDEX[41:32]] <=2'b10;
                   end

                   // we have Fetched and Decoded wrong Instructions, we need
                       to flush them
                                    NewPC <= IDEX[31:0] + {{16{1'b0}},
                                        IDEX[47:32]};
                                    PCSrc <= 1'b1;
                                    StallID <= 1'b1;
                                    StallEX <= 1'b1;

            end
               else if (decisionID == 1'b1)
               begin
                // This is a correct decision and we update the predict bits
                if(pred[IDEX[9:0]] == 2'b00)
                   begin
                      pred[IDEX[41:32]] <=2'b01;
                   end
                if(pred[IDEX[9:0]] == 2'b01)
                   begin
                      pred[IDEX[41:32]] <=2'b11;
                   end
            end
               else
               begin
                // This is a new entry
                BTB[IDEX[9:0]] <= IDEX[31:0] + {{16{1'b0}}, IDEX[47:32]};
                pred[IDEX[9:0]] <= 2'b00;
                                    NewPC <= IDEX[31:0] + {{16{1'b0}},
                                        IDEX[47:32]};
                                    PCSrc <= 1'b1;
                                    StallID <= 1'b1;
                                    StallEX <= 1'b1;
               end
                           end
    else
        // Now that means taking this branch is a Mistake
```

```
                        begin

                 if(decisionID == 1'b0) // That means we haven't taken the
                     Branch
               begin
                // This is Correct decision and we update the predict bits
                if(pred[IDEX[9:0]] == 2'b00)
                   begin
                      pred[IDEX[41:32]] <=2'b01;
                   end
                if(pred[IDEX[9:0]] == 2'b01)
                   begin
                      pred[IDEX[41:32]] <=2'b11;
                   end
            end
               else if (decisionID == 1'b1)
              begin
               // This is a wrong decision and we update the predict bits
               // since we have Fetched and Decoded wrong Instructions, we
                   need to flush them

               if(pred[IDEX[9:0]] == 2'b10)
                   begin
                      pred[IDEX[41:32]] <=2'b00;
                   end
               if(pred[IDEX[9:0]] == 2'b11)
                   begin
                      pred[IDEX[41:32]] <=2'b10;
                   end
                                    NewPC <= IDEX[31:0] + 1;
                                    PCSrc <= 1'b1;
                                    StallID <= 1'b1;
                                    StallEX <= 1'b1;
            end
                        end

end
               // Branch if Greater
               if(EX[14])
                     begin
                         if(IDEX[95:64] > IDEX[127:96])
                             begin
               if(decisionID == 1'b0) // That means we haven't taken the
                   Branch
              begin
               // This is wrong decision and we update the predict bits
               if(pred[IDEX[9:0]] == 2'b10)
                   begin
                      pred[IDEX[41:32]] <=2'b00;
                   end
```

```
            if(pred[IDEX[9:0]] == 2'b11)
               begin
                  pred[IDEX[41:32]] <=2'b10;
               end

               // we have Fetched and Decoded wrong Instructions, we need
                   to flush them
                               NewPC <= IDEX[31:0] + {{16{1'b0}},
                                   IDEX[47:32]};
                              PCSrc <= 1'b1;
                              StallID <= 1'b1;
                              StallEX <= 1'b1;
                        end

            else if (decisionID == 1'b1)
                           begin
                               // This is a correct decision and we
                                   update the predict bits
                           if(pred[IDEX[9:0]] == 2'b00)
                                 begin
                                   pred[IDEX[41:32]] <=2'b01;
                                   end
                           if(pred[IDEX[9:0]] == 2'b01)
                                 begin
                                     pred[IDEX[41:32]] <=2'b11;
                                  end
            end
            else
                              begin
                           // This is a new entry
                           BTB[IDEX[9:0]] <= IDEX[31:0] + {{16{1'b0}},
                               IDEX[47:32]};
                           pred[IDEX[9:0]] <= 2'b00;
                           NewPC <= IDEX[31:0] + {{16{1'b0}},
                               IDEX[47:32]};
                           PCSrc <= 1'b1;
                           StallID <= 1'b1;
                           StallEX <= 1'b1;

                             end
      end
   else
       // Now that means taking this branch is a Mistake
             begin
                    if(decisionID == 1'b0) // That means we haven't
                        taken the Branch
                       begin
                         // This is Correct decision and we update the
                             predict bits
                       if(pred[IDEX[9:0]] == 2'b00)
```

```
                                    begin
                                        pred[IDEX[41:32]] <=2'b01;
                                    end
                                if(pred[IDEX[9:0]] == 2'b01)
                                    begin
                                        pred[IDEX[41:32]] <=2'b11;
                                    end
                                end
                        else if (decisionID == 1'b1)
                                begin
                                        // This is a wrong decision and we
                                            update the predict bits
                                    // since we have Fetched and Decoded
                                        wrong Instructions, we need to flush
                                        them

                                    if(pred[IDEX[9:0]] == 2'b10)
                                        begin
                                            pred[IDEX[41:32]] <=2'b00;
                                        end
                                    if(pred[IDEX[9:0]] == 2'b11)
                                        begin
                                            pred[IDEX[41:32]] <=2'b10;
                                        end
                                      NewPC <= IDEX[31:0] + 1;
                                      PCSrc <= 1'b1;
                                      StallID <= 1'b1;
                                      StallEX <= 1'b1;
                                    end
                                end
                            end

end
        end

    // Access Memory
    if(StallMem == 1'b0)
        begin
            if(StallEX == 1'b1)
                begin
                    StallMem <=1'b1;
                end
            WB <= MEM[3];
            MEMWB[31:0] <= MEM[1]?DatMem[EXMEM[31:0]]:EXMEM[31:0];
            if(MEM[2]) //Memory Write
                begin
                    DatMem[EXMEM[31:0]] <= Reg[EXMEM[68:64]];
                end
            MEMWB[36:32] <= EXMEM[68:64];
        end
```

```
            // Write Back
            if(StallWB == 1'b0)
                begin
                    if(StallMem == 1'b1)
                        begin
                            StallWB<=1'b1;
                        end
                    if(WB==1'b1)
                        begin
                           Reg[MEMWB[36:32]] <= MEMWB[31:0];
                           Result <= MEMWB[31:0];
                        end
                end

          end
endmodule
```

8.2 Test Bench Code

```
module Test_Bench;
reg CLK;
wire [31:0] result;
processor uut(.clock(CLK), .Result(result));
initial
begin
   CLK = 1;
   #5 CLK = ~ CLK;
   #5 CLK = ~ CLK;
   #5 CLK = ~ CLK;
   #5 CLK = ~ CLK;
   #5 CLK = ~ CLK;
   #5 CLK = ~ CLK;
   #5 CLK = ~ CLK;
   #5 CLK = ~ CLK;
   #5 CLK = ~ CLK;
   #5 CLK = ~ CLK;
   #5 CLK = ~ CLK;
   #5 CLK = ~ CLK;
   #5 CLK = ~ CLK;
   #5 CLK = ~ CLK;
end
endmodule
```

8.3 Programs Used for Verification

SWAPPING:

```
LW R1 R0 00H        // load the first element say 2
LW R2 R0 01H        // load the second element say 3
ADD R1 R1 R2       // add both of them and store in R1 so 5,3
SUB R2 R1 R2       // store 5-3 in b, so 5,2
SUB R1 R1 R2       // store 5-2 in a, so 3,2
```

MULTIPLICATION:

```
LW R1 R0 00H     // load the first element
LW R2 R0 01H     // load the second element
XOR R3 R3 R3     // make a register value 0
ADD R3 R3 R1     // add R1 to R3, R2 number of times
DEC R2
BNE R2 R0 04H    // so at the end Product will be stored in R3
```

Note : Store 0 in R0 before to start the program.

FACTORIAL:

```
LW R1 R0 00H     //Load the first element
LW R6 R0 01H     //Backup of the second element
XOR R3 R3 R3     //Make a register zero to store the result
XOR R2 R2 R2
ADD R2 R2 R6     //Copy new value into R2
ADD R3 R3 R1     //Add R1 to R3 R2 Number of tines
DEC R2
BNE R2 R0 06H
DEC R6        //Backup is decremented
ADD R1 R3 R0
BNE R6 R0 03H
```

It is assumed that R0 is set to Zero and datamem [0] and datamem[1] contains the same number to which factorial is to be calculated.

POWER X,Y:

```
LW R1 R0 00H     //Load the first element
LW R7 R0 00H
LW R6 R0 01H     //Backup of the second element
XOR R3 R3 R3     //Make a register zero to store the result
XOR R2 R2 R2
ADD R2 R2 R7     //Copy new value into R2
ADD R3 R3 R1     //Add R1 to R3 R2 Number of tines
DEC R2
```

```
BNE R2 R0 06H
DEC R6        //Backup is decremented
ADD R1 R3 R0
BNE R6 R0 04H
```

It is assumed that R0 is set to Zero and datamem [0] and datamem[1] contains the X and Y respectively.

SORTING AN ARRAY:

```
LW R1 R0 00H     // load number of elements
LW R6 R0 00H     // copy of number of elements
ADDI R2 R0 1     // index of first element
ADDI R3 R0 2     // index of second element
LW R2 R4 00H     // Load the first element
LW R3 R5 00H     // Load the second element
BLE R4 R5     // Jump if lesser
SW R2 R5 00H     // Store in the other place
SW R3 R4 00H     // Store in the other place
INC R2        // increment the first index
INC R3        // increment the second index
BNE R1 R3 05H    // do this number of element times -1
DEC R6        // decrement the r6
BNE R6 R0 03H    // jump until we cover all elements
```

Note : Store 0 in R0 before starting this program
