EL501 Lab Activity

RISC V 32bit 5 Stage pipelined processor

|  |  |
| --- | --- |
| **Name** | **Student ID** |
| Parmar Rajdeep | 202011042 |
| Pandya Abhishek | 202011037 |

**Stage 1**

**Fetch**

**Fetch Unit**comprises a byte (8 bits) addressable instruction memory. It takes PC as input and instructs an output. PC is also incremented by

PC=PC+4.

Instruction Memory holds different instructions to be performed by the processor.

Depending upon branch or jump calculated address will be given to the PC.

**Stage 2**

**Decode**

**Decode Unit**reads the fetched instruction and decodes the address of two

source operands and destination register for register-based arithmetic, logical and shift instructions, immediate data(data or effective address) for immediate data-based instructions like arithmetic, logical, and branch type instructions.

**Register Bank**reads the value of source operands (Rs1 and Rs2) at the negative edge of a clock and writes the data in the destination register (Rd) at the positive edge of the clock.

The forwarding unit forwards the data from the next stages(e.g. execute, Memory) and decides whether to take operand for ALU operation from ALU result or Memory Data to eliminate hazard like RAW(Read after Write), WAR (Write after Read), WAW(Write after Write).

**Stall Unit** will create a delay of 1 clock cycle whenever exceptional case occurs i.e. load instruction followed by arithmetic instruction.

**Stage 3**

**Execute**

**ALU Module**uses the aluop generated by ALU Control Unit to operate on source operands.

It generates the result and stores it in the ex\_alu\_result register.

Operand Selector unit decides which data to take in the operand to perform different operations.

PSR(Program status register) shows the status according to the operations performed on the operands.

The branch unit decides whether to take the branch according to the status of PSR.

**Stage 4**

**Memory**

**Data Memory**: It stores the data of the processor.

**Memory Unit** will either write to the data memory or will read from the data memory.

**Stage 5**

**Write Back**

Write back unit will select the data and write the data to register bank.

**Instruction Format and Description**

**OPCODE**

R-type = 0110011

I-type = 0010011

L-type = 0000011

S-type = 0100011

B-type = 1100011

J-type = 1101111

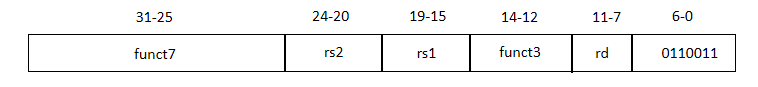
I-type(jalr) = 1100111

1. **R-type Instruction:**

**Assembly(e.g. register-register operation)**

Operation rd, rs1, rs2

**Machine Encoding**



**Semantics**

GPR[rd] = GPR[rs1] operation GPR[rs2]

PC = PC + 1

**Variations**

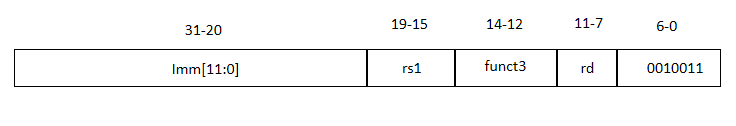
|  |  |  |  |
| --- | --- | --- | --- |
| **FUNC7** | **FUNC3** | **Instruction name** | **aluop** |
| 0000000 | 000 | add | 0000 |
| 0100000 | 000 | sub | 1000 |
| 0000000 | 001 | sll | 0001 |
| 0000000 | 010 | slt | 0010 |
| 0000000 | 011 | sltu | 0011 |
| 0000000 | 100 | xor | 0100 |
| 0000000 | 101 | srl | 0101 |
| 0100000 | 101 | sra | 1101 |
| 0000000 | 110 | or | 0110 |
| 0000000 | 111 | and | 0111 |

1. **I-type Instruction:**

**Assembly(e.g. register-Immediate operations)**

Operation rd, rs1, imm12

**Machine Encoding**



**Semantics**

GPR[rd] = GPR[rs1] operation Sign-extend(imm)

PC =PC + 1

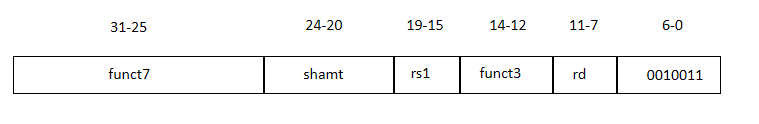
**Variations**

|  |  |  |
| --- | --- | --- |
| **FUNC3** | **Instruction name** | **aluop** |
| 000 | addi | x000 |
| 010 | slti | x010 |
| 011 | sltiu | x011 |
| 100 | xori | x100 |
| 110 | ori | x110 |
| 111 | andi | x111 |

**Assembly(e.g. register-Immediate operations)**

Operation rd, rs1, shamt

**Machine Encoding**



**Semantics**

GPR[rd] = GPR[rs1] operation Shamt

PC =PC + 1

**Shift instruction variations:**

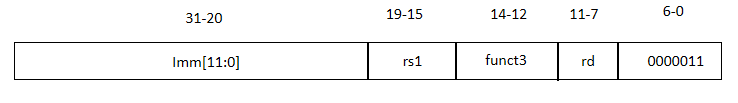
|  |  |  |  |
| --- | --- | --- | --- |
| **FUNC7** | **FUNC3** | **Instruction name** | **aluop** |
| 0000000 | 001 | slli | 0001 |
| 0000000 | 101 | srli | 0101 |
| 0100000 | 101 | srai | 1101 |

1. **L-type instruction:**

**Assembly (e.g. Load 4/2/1-byte)**

Operation rd, offset12(base)

**Machine Encoding**



**Semantics**

Byte\_address32 = sign-extend(offset12) + GPR[base]

GPR[rd] = MEM32[byte\_address]

PC = PC + 1

**Variations**

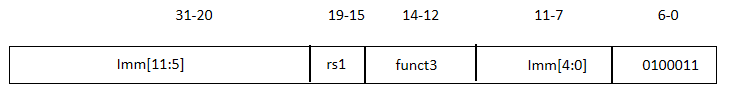
|  |  |  |
| --- | --- | --- |
| **FUNC3** | **Instruction name** | **aluop** |
| 000 | lb | x000 |
| 001 | lh | x001 |
| 010 | lw | x010 |
| 100 | Lbu | X100 |
| 101 | Lhu | X101 |

1. **S-type instruction:**

**Assembly(e.g. Store 4/2/1-byte)**

Operation rs2,offset12(base)

**Machine Encoding**



**Semantics**

byte\_address32 = sign-extend(offset12) + GPR[base]

MEM32[byte\_address] = GPR[rs2]

PC = PC + 1

**Variations**

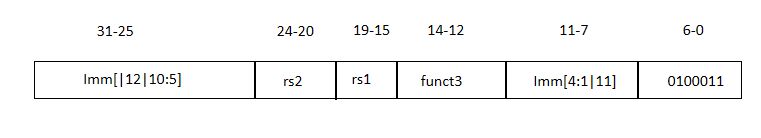
|  |  |  |
| --- | --- | --- |
| **FUNC3** | **Instruction name** | **aluop** |
| 000 | sb | x000 |
| 001 | sh | x001 |
| 010 | sw | x010 |

1. **B-Type instruction:**

**Assembly(e.g. branch if equal)**

Branch\_operation rs1, rs2, imm13

**Machine Encoding**



**Semantics**

Target = PC + sign-extend(imm13)

If GPR[rs1] operation GPR[rs2] then PC = target

Else PC = PC + 1

**Variation:**

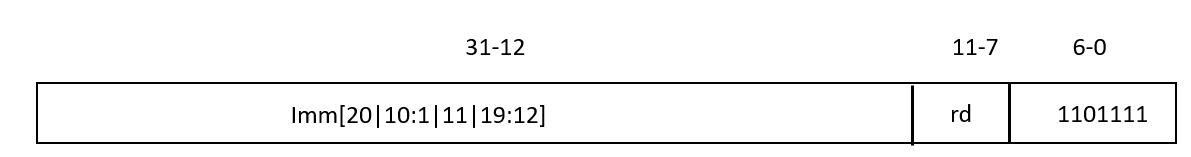
|  |  |  |
| --- | --- | --- |
| **FUNC3** | **Instruction name** | **aluop** |
| 000 | beq | x000 |
| 001 | bne | x001 |
| 100 | blt | x100 |
| 101 | bge | X101 |

1. **Jump and Link Instruction:**

**Assembly**

JAL rd imm21

**Machine Encoding**



**Semantics**

Target = PC + sign-extended(imm21)

GPR[rd] = PC + 4

PC = target

**Exceptions**

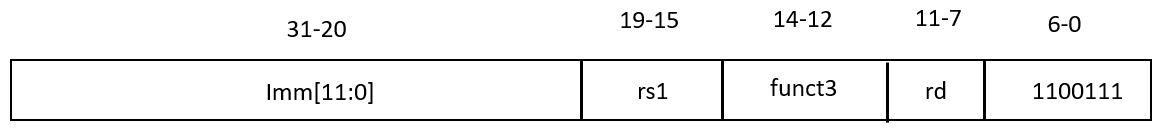
misaligned target (4‐byte)

1. **Jump Indirect and link Instruction:**

**Assembly**

JALR rd, rs1,imm12

**Machine Encoding**



**Semantics**

target = GPR[rs1] + sign‐extend(imm12)

target &= 0xffff\_fffe

GPR[rd] = PC + 4

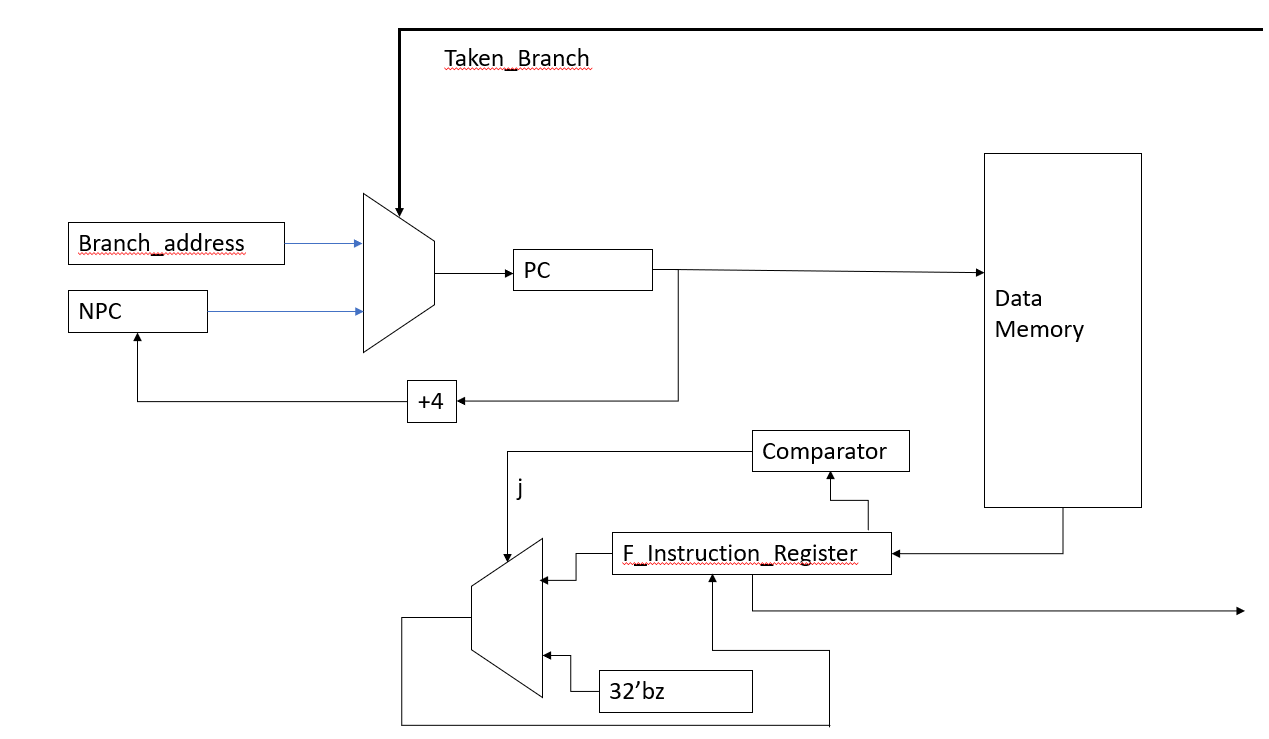
PC = target

**Exceptions**

misaligned target (4‐byte)

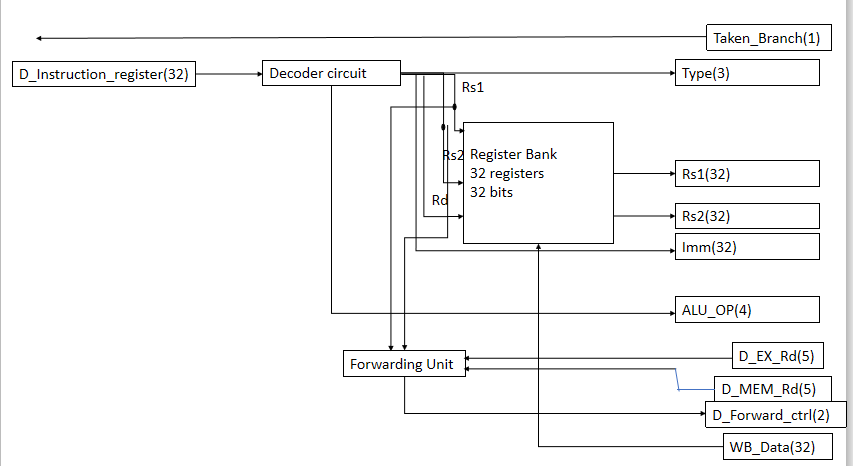
**Description**

**Fetch stage:**



This stage fetches the instruction from instruction memory and stores it in f\_instruction\_register. And after fetching the instruction, it increases the program counter by 4. In the next clock cycle, PC takes value according to the taken\_branch signal. If taken\_branch = 1, PC = branch\_address generated by the branch unit, otherwise PC = NPC. For the Jump instructions if j flag is set then we’ll not take any further two instructions it means that we’ll not fetch the next two instruction written just after the jump instruction because we have to jump to newer location. When fetch unit encounters the jump instructions it’ll set the flag.

**Decode Stage:**



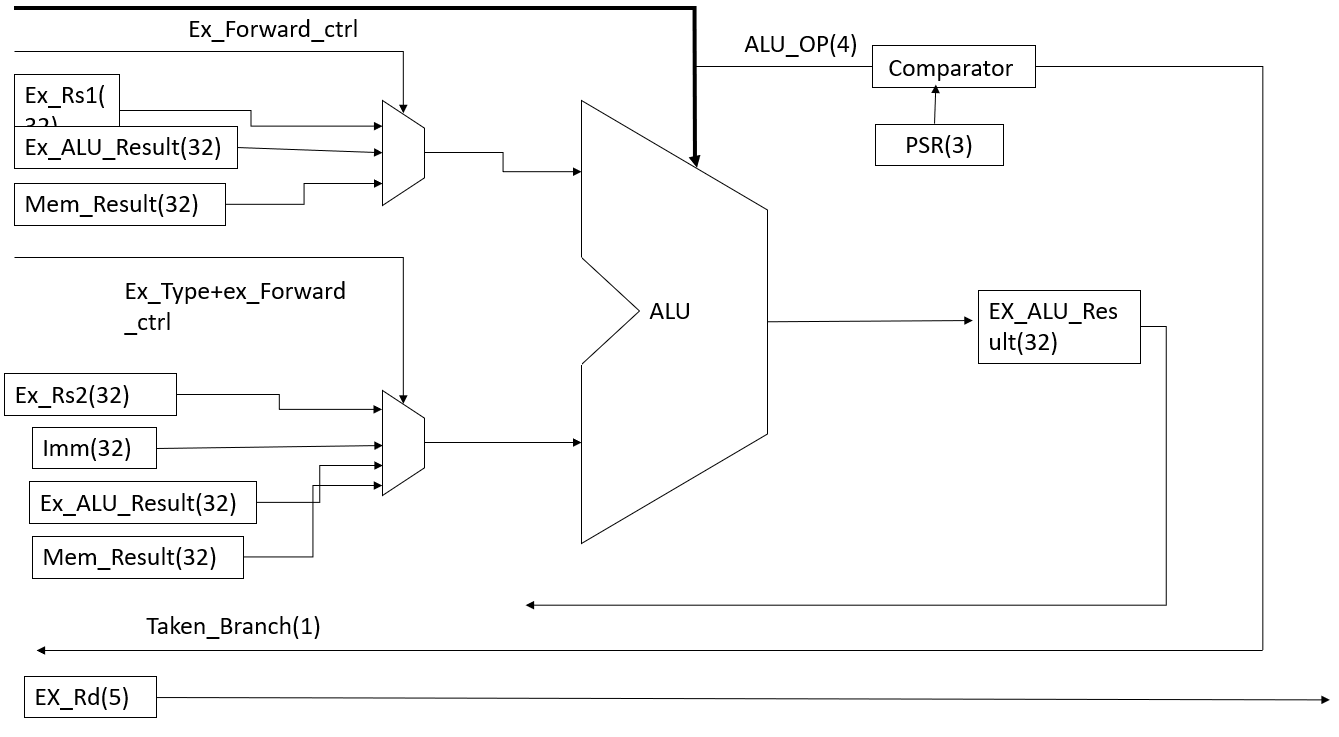
This stage decodes the instruction available in the instruction register and generates signals(i.e. alu\_op, type, etc.).

decoder circuit will give source register addresses (i.e. rs1\_address and rs2\_address), destination register address (i.e. rd\_address), and immediate data.

The forwarding unit will compare different destination register addresses from the next stages, with source operands address, and based on this it will generate d\_forward\_ctrl, which will be used in the operand selector.

alu\_op will decide the operation to be performed in ALU and the type signal will define the type of instruction.

**Execute stage:**



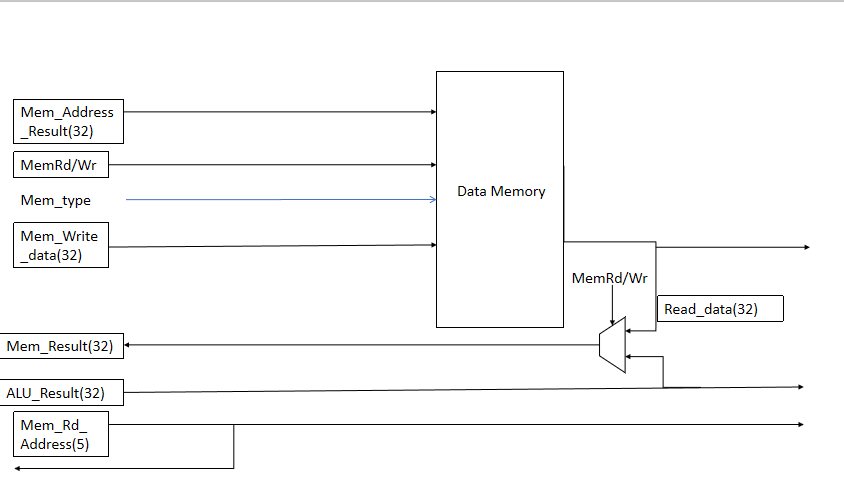
This stage executes different instructions according to the alu\_op signal in the ALU unit. Operand1 to the ALU will be selected according to the forward\_ctrl signal and Operand2 will be selected according to the type and forward\_ctrl signals.

ex\_rsk(k = 1,2 ) defines data of rsk register, ex\_alu\_result defines data of previously executed instruction, mem\_result defines data from the memory that is received in previous instruction and imm defines immediate data from the instruction.

PSR is a program status register. It contains 3 bits, which will be used to decide the taken\_branch signal in conditional branch instructions and jump type of instructions.

|  |  |
| --- | --- |
| **PSR bits(3)** | **Condition** |
| 000 | rs1 = rs2 |
| 001 | rs1 != rs2 |
| 011 | rs1 >= rs2 (signed) |
| 010 | rs1 < rs2 (signed) |
| 100 | Rs1 < Rs2 (unsigned) |
| 101 | Rs1 >= Rs2 (unsigned) |
| 110 | Jump and link |
| 111 | Jump and link with target address calculated using one of the register in reg\_bank |

**Memory stage :**



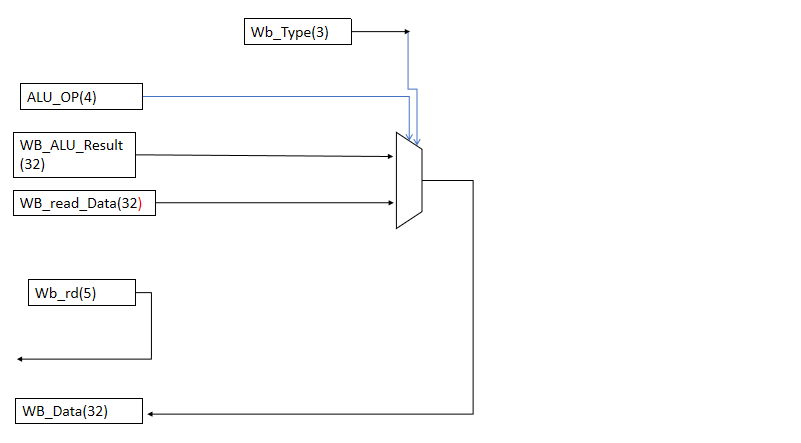
This stage is used to load and store data from register\_bank to data\_memory and vice-versa.

mem\_rd\_wr signal defines load or store operation(i.e. mem\_rd\_wr = 0 , load and mem\_rd\_wr = 1, store).

mem\_address\_result will contain either result or address. mem\_type defines the type of instruction. mem\_write\_data register defines the data that must be written in data\_memory in a store instruction.

mem\_result and mem\_rd\_adderess are used for data forwarding.

**Write-Back stage**



This stage is used for writing the data in the register bank. alu\_op signal is used for defining the number of Bytes to be moved in the register bank. wb\_alu\_result is the data to be written in the register bank which is the result from the ALU and wb\_read\_data is the data to be written in the register bank which is from the memory stage (i.e. load). No. of bits stored in the register of register bank will depend upon the instruction (i.e. byte, half word, unsigned). It will store the pc content to the one of the register in the register bank for the jump instructions.

**Code:**

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company:

// Engineer: Parmar Rajdeep Pravinbhai and Pandya Abhishek Janakkumar

//

// Create Date: 21:31:32 03/29/2021

// Design Name: RISC V Processor

// Module Name: RISC\_processor\_5\_stage

// Project Name: RISC V Processor

// Target Devices:

// Tool versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module RISC\_processor\_5\_stage( rst, clk );

input clk,rst ;

reg [31:0]npc, pc, branch\_address, f\_instruction\_register;//fetch stage

reg [7:0]instruction\_memory[8191 - 1:0];//Fetch Stage

reg f\_taken\_branch;

reg [31:0]d\_instruction\_register, d\_rs1, d\_rs2, d\_imm;//decode stage

reg [1:0]d\_forward\_ctrl;//decode stage

reg [2:0]d\_type;//decode stage

reg [3:0]d\_alu\_op;//decode stage

reg [4:0]d\_rs1\_address, d\_rs2\_address, d\_rd\_address, d\_ex\_rd, d\_mem\_rd, d\_shamt;//decode stage

reg [31:0]reg\_bank[31:0];//decode stage

reg [31:0]ex\_rs1, ex\_mem\_result, ex\_rs2, ex\_imm, ex\_alu\_result;//execute stage

reg [4:0]ex\_rd\_address;//execute stage

reg [3:0]ex\_alu\_op;//execute stage

reg [1:0]ex\_forward\_ctrl;//execute stage

//reg [2:0]psr;//execute stage

reg [2:0]psr;//execute stage

reg [2:0]ex\_type;//execute stage

reg [4:0]ex\_shamt;//execute stage

reg [31:0]ex\_s\_rs2;//execute stage

reg [31:0]operand1, operand2;// operand for execute stage used in alu

reg [31:0]mem\_address\_result, mem\_write\_data, wb\_mem\_result, mem\_alu\_result, mem\_read\_data, mem\_result;//mem stage

reg mem\_rd\_wr;//memory unit

reg [4:0]mem\_rd\_address;//memory stage

reg [2:0]mem\_type;//memory stage

reg [2:0]mem\_alu\_op;//memory stage

reg [31:0]data\_memory[8191:0];//data memory

reg [2:0]wb\_type;//write back unit

reg [31:0]wb\_data; //write back unit

reg [4:0]wb\_rd\_address; //write back unit

reg [2:0]wb\_alu\_op;//write back unit

reg j;

initial

begin

reg\_bank[0] = 32'd0;//Initialize the register bank with some data

reg\_bank[1] = 32'd1;//Initialize the register bank with some data

reg\_bank[2] = 32'd2;//Initialize the register bank with some data

reg\_bank[3] = 32'd3;;//Initialize the register bank with some data

data\_memory[0] = 32'd00;//Initialize the data memory with some data

data\_memory[1] = 32'd10;//Initialize the data memory with some data

data\_memory[2] = 32'd20;//Initialize the data memory with some data

pc = 0;//Initialize pc to the 0

npc = 0;//Initialize npc to the 0

$readmemb("C://Users//rajde//Documents//daiict assignment//Semester 2//Digital system architecture//5 stage pipeline processor//instruction\_memory\_v2.txt", instruction\_memory);

end

always @(posedge clk)

begin

if(rst)//will reset processor

begin

pc = 0;

f\_taken\_branch = 0;

end

if(f\_taken\_branch) //will branch to the address based on condition of branch

begin

pc = branch\_address;

npc = pc + 4;

f\_instruction\_register[7:0] = instruction\_memory[pc+3];

f\_instruction\_register[15:8] = instruction\_memory[pc+2];

f\_instruction\_register[23:16] = instruction\_memory[pc+1];

f\_instruction\_register[31:24] = instruction\_memory[pc];

f\_taken\_branch = 1'b0;//reset f\_taken\_branch

j = 1'b0;

end

else if(j)

begin

f\_instruction\_register = 32'bz;

end

else //will fetch next instructions

begin

pc = npc;

f\_instruction\_register[7:0] = instruction\_memory[pc+3];

f\_instruction\_register[15:8] = instruction\_memory[pc+2];

f\_instruction\_register[23:16] = instruction\_memory[pc+1];

f\_instruction\_register[31:24] = instruction\_memory[pc];

npc = pc + 4;

////////////jal & jalr/////////

if( (f\_instruction\_register[6:0] == 7'b1101111) & (f\_instruction\_register[6:0] == 7'b1100111) )

begin

j = 1'b1;

end

end

end

///////////////////////////decode stage/////////////////////////////////////////

always @(negedge clk)

begin

if( ( (d\_instruction\_register[6:0] == 7'b0110011) | (d\_instruction\_register[6:0] == 7'b0010011) ) & (f\_instruction\_register[6:0] == 7'b0000011) ) //this will give 1 clock cycle delay stall for ldr followed by the r-type instruction

begin

#20;

end

d\_instruction\_register = f\_instruction\_register; //carry forward instruction

d\_rs1\_address = d\_instruction\_register[19:15]; //decoder ckt// will get the source operand1

d\_rs2\_address = d\_instruction\_register[24:20]; //decoder ckt// will get the source operand2

d\_rd\_address = d\_instruction\_register[11:7]; //decoder ckt// will get the destination operand

case(d\_instruction\_register[6:0]) //opcode define type of instructions

7'b0110011: d\_type = 3'b000;//define r-type instructions

7'b0010011: begin

d\_type = 3'b001;//define i-type instructions

d\_imm = { {21{d\_instruction\_register[31]}}, d\_instruction\_register[30:25], d\_instruction\_register[24:21], d\_instruction\_register[20] } ; //generate immediate value

end

7'b0000011: begin

d\_type = 3'b010;//define load type instructions

d\_imm = { {21{d\_instruction\_register[31]}}, d\_instruction\_register[30:25], d\_instruction\_register[24:21], d\_instruction\_register[20] } ; //generate immediate value

end

7'b0100011: begin

d\_type = 3'b011;//define store type instructions

d\_imm = { {21{d\_instruction\_register[31]}}, d\_instruction\_register[30:25], d\_instruction\_register[11:8], d\_instruction\_register[7] } ; //generate immediate value

end

7'b1100011: begin

d\_type = 3'b100;//define branch type instructions

d\_imm = { {20{d\_instruction\_register[31]}}, d\_instruction\_register[7], d\_instruction\_register[30:25], d\_instruction\_register[11:8], 1'b0}; //generate immediate value

end

7'b1101111: begin

d\_type = 3'b101;//define jump type instructions

d\_imm = { {12{d\_instruction\_register[31]}}, d\_instruction\_register[19:12], d\_instruction\_register[20], d\_instruction\_register[30:25], d\_instruction\_register[24:21], 1'b0};//generate immediate value

end

7'b1100111: begin

d\_type = 3'b110;//define jump type of jalr instructions

d\_imm = { {21{d\_instruction\_register[31]}}, d\_instruction\_register[30:25], d\_instruction\_register[24:21], d\_instruction\_register[20] } ; //generate immediate value

end

endcase

d\_alu\_op = {d\_instruction\_register[30],d\_instruction\_register[14:12]};//define operation to be performed in alu

///////////////////////////forwarding unit//////////////////////

if(d\_rs1\_address == d\_ex\_rd) //this is used to eliminate different hazards

d\_forward\_ctrl = 2'b00; //1st bit define ex or mem stage

else if(d\_rs2\_address == d\_ex\_rd)

d\_forward\_ctrl = 2'b01; //2nd bit define rs1 or rs2 of d\_forward\_ctrl

else if(d\_rs1\_address == d\_mem\_rd)

d\_forward\_ctrl = 2'b10;

else if(d\_rs2\_address == d\_mem\_rd)

d\_forward\_ctrl = 2'b11;

else

d\_forward\_ctrl = 2'bzz;

d\_rs1 = reg\_bank[d\_rs1\_address]; //get the data of source operand1

d\_rs2 = reg\_bank[d\_rs2\_address];//get the data of source operand2

d\_shamt = d\_rs2\_address; //define shift amount

ex\_s\_rs2 = reg\_bank[d\_rs2\_address]; //get the data of source operand2

end

////////////////////////////////execute stage/////////////////////////////////////////////////

always @(posedge clk)

begin

ex\_rs1 = d\_rs1; //define data of source operand1 which is forwarded from the previous stage

ex\_alu\_result = mem\_result; //define data of alu result which is forwarded from the next stage

ex\_mem\_result = mem\_result; //define data of memory which is forwarded from the next stage

ex\_rs2 = d\_rs2; //define data of source operand2 which is forwarded from the previous stage

ex\_imm = d\_imm; //define immediate data which is forwarded from the previous stage

ex\_rd\_address = d\_rd\_address; //define address of the destination register

ex\_forward\_ctrl = d\_forward\_ctrl; //define the bits for the operand selector

ex\_type = d\_type; //define type of instruction forwarded from the previous stage

ex\_alu\_op = d\_alu\_op; //define operation of instruction forwarded from the previous stage

ex\_shamt = d\_shamt; //define shift amount forwarded from the previous stage

d\_ex\_rd = ex\_rd\_address; //define the destination address forwarded from the previous stage

/////////////////////////////////operand selector/////////////////////////////

operand1 = ex\_rs1; //source operand1

operand2 = ex\_rs2; //source operand2

case(ex\_forward\_ctrl) //make decision based on the forwarding unit

2'b00: operand1 = ex\_alu\_result; //alu result

2'b10: operand1 = ex\_mem\_result; //memory data

endcase

case({ex\_type, ex\_forward\_ctrl })//selects appropriate operand based on the type of instruction and forward\_ctrl signal

5'b001\_00 : operand2 = ex\_imm;

5'b001\_01 : operand2 = ex\_imm;

5'b001\_10 : operand2 = ex\_imm;

5'b001\_11 : operand2 = ex\_imm;

5'b001\_zz : operand2 = ex\_imm;

5'b010\_?? : operand2 = ex\_imm;

5'b011\_?? : operand2 = ex\_imm;

5'b000\_00 : operand2 = ex\_alu\_result;

5'b000\_01 : operand2 = ex\_alu\_result;

5'b000\_10 : operand2 = ex\_alu\_result;

5'b000\_11 : operand2 = ex\_alu\_result;

5'b???\_11 : operand2 = ex\_mem\_result;

endcase

casez( { ex\_type, ex\_alu\_op} )

////////////////////r-type//////////////////////////////

7'b000\_0\_000: ex\_alu\_result = operand1 + operand2;//add

7'b000\_1\_000: ex\_alu\_result = operand1 - operand2;//sub

7'b000\_0\_001: ex\_alu\_result = operand1 << operand2; //rd,rs1,rs2

7'b000\_0\_010: ex\_alu\_result = ($signed(operand1) < $signed(operand2))?1:0;// slt

7'b000\_0\_011: ex\_alu\_result = (operand1 < operand2)?1:0; //sltu

7'b000\_0\_100: ex\_alu\_result = operand1 ^ operand2; //xor

7'b000\_0\_101: ex\_alu\_result = operand1 >> operand2; //srl shift right logical

7'b000\_1\_101: ex\_alu\_result = operand1 >>> operand2; //sra 1010101 111\_101010

7'b000\_0\_110: ex\_alu\_result = operand1 | operand2;// or

7'b000\_0\_111: ex\_alu\_result = operand1 & operand2;//and

////////////////////i - type/////////////////////////////

7'b001\_?\_000: ex\_alu\_result = operand1 + operand2; //addi

7'b001\_?\_010: ex\_alu\_result = ( $signed(operand1) < $signed(operand2) )?1:0; //slti

7'b001\_?\_011: ex\_alu\_result = ( operand1 < operand2)?1:0; //sltiu

7'b001\_?\_100: ex\_alu\_result = operand1 ^ operand2;//xori

7'b001\_?\_110: ex\_alu\_result = operand1 | operand2; //ori

7'b001\_?\_111: ex\_alu\_result = operand1 & operand2; //andi

7'b001\_?\_001: ex\_alu\_result = operand1 << ex\_shamt; //slli

7'b001\_?\_101: ex\_alu\_result = operand1 >> ex\_shamt; //srli

7'b001\_1\_101: ex\_alu\_result = operand1 >>> operand2;//srai

//////////////////load type//////////////////////////////

7'b010\_?\_0??: begin

mem\_rd\_wr = 0; //if this is 0 that means read from the memory

ex\_alu\_result = operand1 + operand2;//load

end

7'b010\_?\_100: begin

mem\_rd\_wr = 0;

ex\_alu\_result = operand1 + operand2; //

end

7'b010\_?\_101: begin

mem\_rd\_wr = 0;

ex\_alu\_result = operand1 + operand2;

end

//////////////////store type////////////////////////////

7'b011\_?\_0??: begin

mem\_rd\_wr = 1; //if this is 1 that means write from the memory

ex\_alu\_result = operand1 + ex\_s\_rs2 ;//store

end

//////////////////Branch type////////////////////////////00 00 bne

7'b100\_?\_00?: begin

psr = (operand1 == operand2)?000:001; //beq = 000, bne = 001

branch\_address = npc + ex\_imm - 8;

end

7'b100\_?\_10?:begin

psr = ( $signed(operand1) < $signed(operand2) )?010:011; //blt = 010, bge = 011

branch\_address = npc + ex\_imm - 8;

end

7'b100\_?\_11?:begin

psr = ( operand1 < operand2)?100:101; //bltu = 100 , bgeu = 101

branch\_address = npc + ex\_imm - 8;

end

/////////////////jump type///////////////////////////

7'b101\_?\_???:begin //jal

psr = 3'b110;

branch\_address = npc + ex\_imm - 8;

ex\_alu\_result = npc-8 ;

end

/////////////////jalr type//////////////////

7'b110\_?\_???:begin

psr = 3'b111;

branch\_address = ( operand1 + ex\_imm) & (32'hffff\_fffe );

ex\_alu\_result = npc - 8;

end

endcase

casez({ex\_type, ex\_alu\_op})

7'b100\_?\_000: f\_taken\_branch = (psr == 3'b000)?1:0; //take branch if equal

7'b100\_?\_001: f\_taken\_branch = (psr == 3'b001)?1:0; //take branch if not equal

7'b100\_?\_100: f\_taken\_branch = (psr == 3'b010)?1:0; //take branch if less than

7'b101\_?\_101: f\_taken\_branch = (psr == 3'b011)?1:0; //take branch if greater or equal

7'b101\_?\_110: f\_taken\_branch = (psr == 3'b100)?1:0; //take branch if less than unsigned

7'b101\_?\_111: f\_taken\_branch = (psr == 3'b101)?1:0; //take branch if greater or equal unsigned

7'b101\_?\_???: f\_taken\_branch = (psr == 3'b110)?1:0;// jal

7'b110\_?\_???: f\_taken\_branch = (psr == 3'b111)?1:0;//jalr

endcase

end

//////////////////////////memory stage /////////////////////////////////////

always @(negedge clk)

begin

mem\_address\_result = ex\_alu\_result; //forward alu result

mem\_write\_data = ex\_rs1; //forward data to be written in data memory

mem\_rd\_address = ex\_rd\_address; //forward the destination register address

mem\_type = ex\_type; //forward type of instruction

mem\_alu\_op = ex\_alu\_op; //forward aluop

if(!mem\_rd\_wr) //load instruction

begin

mem\_read\_data = data\_memory[mem\_address\_result];

mem\_result = mem\_read\_data;

mem\_rd\_wr = 1'bz; //reset condition

end

else if(mem\_rd\_wr) //store instruction

begin

case(mem\_alu\_op)

3'b000: data\_memory[mem\_address\_result] = { {24{mem\_write\_data[7]}}, mem\_write\_data[7:0]}; //sb

3'b001: data\_memory[mem\_address\_result] = { {16{mem\_write\_data[15]}}, mem\_write\_data[15:0]};//sh

3'b010: data\_memory[mem\_address\_result] = mem\_write\_data;//sw

endcase

mem\_rd\_wr = 1'bz; //reset condition

end

else

begin

mem\_result = mem\_address\_result; //forward the result to be written in wb unit

end

end

////////////////////write back stage////////////////////////////

always @(posedge clk)

begin

wb\_type = mem\_type; //forward the type of instruction

wb\_rd\_address = ex\_rd\_address; //forward the destination register

wb\_data = mem\_result; //data to be written

wb\_alu\_op = mem\_alu\_op; //define alu op and forwarded

casez(wb\_type)

3'b00?: reg\_bank[wb\_rd\_address] = wb\_data; //r-type or i-type

3'b010: begin

case(wb\_alu\_op)

3'b000: reg\_bank[wb\_rd\_address] = { {24{wb\_data[7]}} , wb\_data[7:0] }; //lb

3'b001: reg\_bank[wb\_rd\_address] = { {16{wb\_data[15]}} , wb\_data[15:0] }; //lh

3'b010: reg\_bank[wb\_rd\_address] = wb\_data; //lw

3'b100: reg\_bank[wb\_rd\_address] = { {24{1'b0}}, wb\_data[7:0]};//lbu

3'b101: reg\_bank[wb\_rd\_address] = { {16{1'b0}}, wb\_data[15:0] };//lhu

endcase

end

3'b101: begin

reg\_bank[wb\_rd\_address] = wb\_data;

end

3'b110: begin

reg\_bank[wb\_rd\_address] = wb\_data;

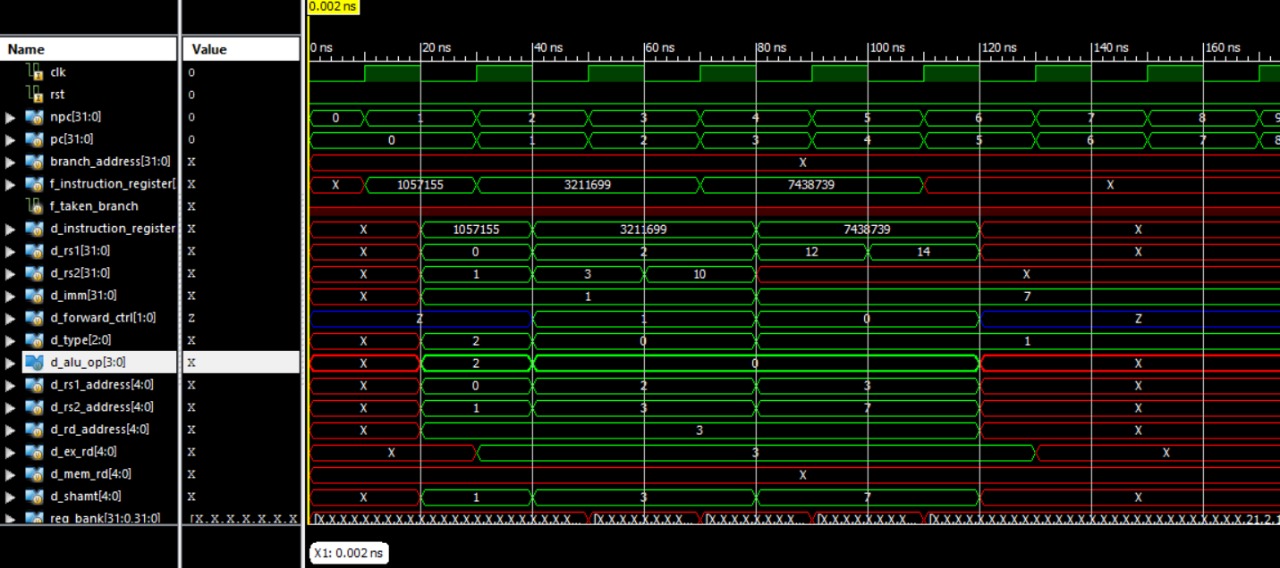
end

endcase

end

endmodule

**Simulation:**

****

**References –**

1. <http://users.ece.cmu.edu/~jhoe/course/ece447/S21handouts/L02.pdf>