CA Final Project

5 Stage Pipelined RISC V Processor

Team Members: Rida Khan(rk04364), Tasbiha Asim(05227), Maheen Khan(mk04389)

TASK 1

Changes made for Bubble Sort: Binary code for bubble sort was included in Instruction memory module and the relevant changes were made in the 64-bit ALU to implement bne and bgt commands.

ALU 64 bit:

```
module ALU_64_bit(,
input [63:0] a,b,
input [3:0] ALUop,
output [63:0] result,
output reg zero,
output reg bgt, bne
//give func3 as input from instruction parser
);

reg [63:0] temp_result;
assign result = temp_result;

always @(*)
begin
if (ALUop == 4'b0000) //and
temp_result = a & b;

else if (ALUop == 4'b0001)//or
temp_result = a | b;
```

```
else if (ALUop == 4'b0010)//add
temp_result = a + b;

else if (ALUop == 4'b0110)//sub
begin
temp_result = a - b;
end
else
temp_result = ~(a | b); //nor

if (temp_result == 64'b0)
begin

zero = 1'b1;
bne = 0;
end
else if (temp_result !=64'b0)
begin
zero = 1'b0;
bne = 1;
end

if (a>b)
bgt = 1'b1;
else if (a<b)
bgt = 1'b0;
end
Endmodule
```

Adder

```
module adder(,
input [63:0] A,
```

```
input [63:0] B,
output [63:0] C
);
assign C = A + B;
Endmodule
```

Control unit

```
module control_unit(,
input [6:0] opcode,
output reg ALUSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch,
output reg [1:0] ALUop
);
always @(*)
begin
case (opcode)
7'b0110011:
begin
ALUSrc = 1'b0;
MemToReg = 1'b0;
RegWrite = 1'b1;
MemRead = 1'b0;
MemWrite = 1'b0;
Branch = 1'b0;
ALUop = 2'b10;
end
7'b0000011:
begin
ALUSrc = 1'b1;
MemToReg = 1'b1;
```

```
RegWrite= 1'b1;
MemRead = 1'b1;
MemWrite = 1'b0;
Branch = 1'b0;
ALUop = 2'b00;
end
7'b0100011:
begin
ALUSrc = 1'b1;
MemToReg = 1'bx;
RegWrite= 1'b0;
MemRead = 1'b0;
MemWrite = 1'b1;
Branch = 1'b0;
ALUop = 2'b00;
end
7'b1100011:
begin
ALUSrc = 1'b0;
MemToReg = 1'bx;
RegWrite= 1'b0;
MemRead = 1'b0;
MemWrite = 1'b0;
Branch = 1'b1;
ALUop = 2'b01;
end
endcase
end
endmodule
```

DATA EXTRACTOR:

```
module data_extractor(,
input [31:0]instruction,
output reg [63:0] immdata);

wire [63:0]immdata1, immdata2, immdata3;

assign imm_data1 = {{52{instruction[31]}}, instruction[31:20]};
assign imm_data2 = {{52{instruction[31]}}, instruction[31:25], instruction[11:7]};
assign imm_data3 = {{52{instruction[31]}}, instruction[31], instruction[7], instruction[30:25], instruction[11:8]};

always@(*)
begin
if(instruction[5] == 1'b0 && instruction[6] == 1'b0)
immdata = imm_data1;
else if (instruction[5] == 1'b1 && instruction[6] == 1'b0)
immdata = imm_data2;
else if (instruction[6] == 1'b1)
immdata = imm_data3;
end

endmodule
```

DATA MEMORY

```
module Data_memory(,
input [63:0] WriteData, Mem_Addr,
input MemWrite, clk, MemRead,
output reg[63:0]ReadData1
);
```

```
reg [7:0] memory [2:0];
integer seed, i,j;
initial begin
for (i=0; i<3; i=i+1)
begin
j = $urandom%50;
memory[i]= j;
$display("data %d, mem %d",j, i);
end
end


always @(MemRead or Mem_Addr)
begin
if (MemRead == 1)
begin
// ReadData1[7:0] = memory[Mem_Addr];
// ReadData1[15:8] = memory[Mem_Addr+1];
// ReadData1[23:16] = memory[Mem_Addr+2];
// ReadData1[31:24] = memory[Mem_Addr+3];
// ReadData1[39:32] = memory[Mem_Addr+4];
// ReadData1[47:40] = memory[Mem_Addr+5];
// ReadData1[55:48] = memory[Mem_Addr+6];
// ReadData1[63:56] = memory[Mem_Addr+7];
ReadData1 = {memory[Mem_Addr + 7], memory[Mem_Addr + 6],
memory[Mem_Addr + 5], memory[Mem_Addr + 4],
memory[Mem_Addr + 3], memory[Mem_Addr + 2],
memory[Mem_Addr + 1], memory[Mem_Addr]};
```

```
end
end


always @ (posedge clk)
begin
if (MemWrite == 1)
begin
memory[Mem_Addr] = WriteData[7:0];
memory[Mem_Addr+1] = WriteData[15:8];
memory[Mem_Addr+2] = WriteData[23:16];
memory[Mem_Addr+3] = WriteData[31:25];
memory[Mem_Addr+4] = WriteData[39:32];
memory[Mem_Addr+5] = WriteData[47:40];
memory[Mem_Addr+6] = WriteData[55:48];
memory[Mem_Addr+7] = WriteData[63:56];
end
ReadData1 = {memory[Mem_Addr + 7], memory[Mem_Addr + 6],
memory[Mem_Addr + 5], memory[Mem_Addr + 4],
memory[Mem_Addr + 3], memory[Mem_Addr + 2],
memory[Mem_Addr + 1], memory[Mem_Addr]};
end
endmodule
```

INSTRUCTION FETCH:

```
module instruction_fetch(,
input reset, clk,
output wire[31:0] instruction
);
```

```
wire [63:0] PC_out, out;
adder add(.A(PC_out), .B(64'd4), .C(out));
program_counter pc(.clk(clk), .reset(reset),.PC_in(out), .PC_out(PC_out));
Instruction_memory im(.Instruction_address(PC_out), .Instruction(instruction));
endmodule
```

INSTRUCTION MEMORY:

```
module Instruction_memory(,
input [63:0] Instruction_address,
output reg[31:0] Instruction
);
reg [7:0] memory [63:0];
initial begin
//0x01600bb3 add x23 x0 x22
memory[0] <= 8'b10110011;
memory[1] <= 8'b00001011;
memory[2] <= 8'b01100000;
memory[3] <= 8'b00000001;
//0x00a10633 add x12 x2 x10
memory[4] <= 8'b00110011;
memory[5] <= 8'b00000110;
memory[6] <= 8'b10100001;
memory[7] <= 8'b00000000;
//0x00062783 ld x15 0(x12)
memory[8] <= 8'b10000011;
memory[9] <= 8'b00110111;
memory[10] <= 8'b00000110;
memory[11] <= 8'b00000000;
//0x002b9193 addi x3, x23, 8
memory[12] <= 8'b10010011;
```

```
memory[13] <= 8'b10000001;
memory[14] <= 8'b10001011;
memory[15] <= 8'b00000000;
//0x00a186b3 add x13 x3 x10
memory[16] <= 8'b10110011;
memory[17] <= 8'b10000110;
memory[18] <= 8'b10100001;
memory[19] <= 8'b00000000;
//0x0006a803 ld x16 0(x13)
memory[20] <= 8'b00000011;
memory[21] <= 8'b10111000;
memory[22] <= 8'b00000110;
memory[23] <= 8'b00000000;
//0x0107c463 bgt x16, x15, 8
memory[24] <= 8'b01100011;
memory[25] <= 8'b01000100;
memory[26] <= 8'b11111000;
memory[27] <= 8'b00000001;
//0x00000863 beq x0 x0 16
memory[28] <= 8'b01100011;
memory[29] <= 8'b00001000;
memory[30] <= 8'b00000000;
memory[31] <= 8'b00000000;
//0x00f002b3 add x5 x0 x15
memory[32] <= 8'b10110011;
memory[33] <= 8'b00000010;
memory[34] <= 8'b11110000;
memory[35] <= 8'b00000000;
//0x01062023 sd x16 0(x12)
```

```
memory[36] <= 8'b00100011;
memory[37] <= 8'b00110000;
memory[38] <= 8'b00000110;
memory[39] <= 8'b00000001;
//0x0056a023 sd x5 0(x13)
memory[40] <= 8'b00100011;
memory[41] <= 8'b10110000;
memory[42] <= 8'b01010110;
memory[43] <= 8'b00000000;
//0x001b8b93 addi x23 x23 1
memory[44] <= 8'b10010011;
memory[45] <= 8'b10001011;
memory[46] <= 8'b00011011;
memory[47] <= 8'b00000000;
//0xfcbb9ae3 bne x23 x11 -44
memory[48] <= 8'b11100011;
memory[49] <= 8'b10011010;
memory[50] <= 8'b10111011;
memory[51] <= 8'b11111100;
//0x001b0b13 addi x22 x22 1
memory[52] <= 8'b00010011;
memory[53] <= 8'b00001011;
memory[54] <= 8'b00011011;
memory[55] <= 8'b00000000;
//0x00410113 addi x2 x2 4
memory[56] <= 8'b00010011;
memory[57] <= 8'b00000001;
memory[58] <= 8'b01000001;
memory[59] <= 8'b00000000;
```

```
//0xfcbb12e3 bne x22 x11 -60
memory[60] <= 8'b11100011;
memory[61] <= 8'b00010010;
memory[62] <= 8'b10111011;
memory[63] <= 8'b11111100;
end
always @(*)
begin
Instruction = {memory[Instruction_address[3:0] + 3], memory[Instruction_address[3:0] + 2],
memory[Instruction_address[3:0] + 1], memory[Instruction_address[3:0]]};
end
Endmodule
```

INSTRUCTION PARSER

```
module instruction_parser(,
input [31:0] inst,
output [6:0] opcode,
output [4:0] rd,
output [2:0] funct3,
output [4:0] rs1,
output [4:0] rs2,
output [6:0] funct7
);

assign opcode = inst[6:0];
assign rd= inst[11:7];
assign funct3 =inst[14:12];
assign rs1 = inst[19:15];
assign rs2 = inst[24:20];
```

```
assign funct7 = inst[31:25];
endmodule
```

MUX:

```
module mux(,
input [63:0] a, b,
input sel,
output [63:0]dataout
);
assign dataout = sel ? a:b;

endmodule
```

Program Counter:

```
module program_counter(,
input [63:0] PC_in,
input clk, reset,
output reg[63:0] PC_out
);
always @(posedge clk or posedge reset)
begin
if (reset)
begin
PC_out = 0;
end
else
begin
PC_out =PC_in;
end
end
```

```
endmodule
```

Register file

```
module registerFile(,
input [63:0] WriteData,
input [4:0]RS1,RS2, RD,
input RegWrite, clk, reset,
output reg[63:0]ReadData1, ReadData2
);
reg [63:0] Registers [31:0];
integer seed, i,j;
initial begin
for (i=0; i<32; i=i+1)
begin
j=i;
Registers[i]<= j;
end
end
always @(*)
begin
if (reset)
begin
for (i=0; i<32; i=i+1) Registers[i] <= i;
ReadData1 <= Registers[RS1];
ReadData2 <= Registers[RS2];
end
else if (RegWrite == 0)
begin
ReadData1 <= Registers[RS1];
ReadData2 <= Registers[RS2];
```

```
end
else if (RegWrite == 1)
begin

ReadData1 <= Registers[RS1];
ReadData2 <= Registers[RS2];
Registers[RD] <= WriteData;

end
end
endmodule
```

Test bench:

```
module tb();,

reg clk, reset;

top ta(.clk(clk), .reset(reset));

initial begin

clk = 1'b0;
reset = 1'b1;
#7 reset = 1'b0;
#7 reset = 1'b0;
end
always
#5 clk = ~clk;
```

Endmodule

Top for single cycle RISC processor:

```
module top(,
input clk,
input reset);
wire [63:0] PC_in, PC_out;
wire [31:0] Instruction;
wire [6:0] opcode;
wire [4:0] rd;
wire [4:0] rs1;
wire [4:0] rs2;
wire [2:0] funct3;
wire [6:0] funct7;
wire [63:0] immdata, out, WriteData;
wire [63:0]ReadData1, ReadData2;
wire ALUSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch;
wire [1:0] ALUop;
wire [3:0] operation;
wire [63:0]dataout;
wire [63:0] result1;
wire [63:0] result2;
wire [63:0] readdatadm;
wire [63:0] add_out;
wire [1:0] Branchcontrol;
wire final;
wire zero;
```

```
wire bgt;
wire bne;

wire [63:0] mx_Branch;
wire [63:0] mx_DM;
wire [63:0] mx_Offset;

wire [63:0] ALU_PC;
wire [63:0] ALU_shift;
wire [63:0] ALU_Main_result;



program_counter pc (.PC_in(mx_Branch), .clk(clk), .reset(reset), .PC_out(PC_out));

Instruction_memory im (.Instruction_address(PC_out), .Instruction(Instruction));

instruction_parser ip(.inst(Instruction),
.opcode(opcode),.rd(rd),.funct3(funct3),.rs1(rs1),.rs2(rs2),.funct7(funct7));

registerFile rf(.WriteData(mx_DM), .RS1(rs1), .RS2(rs2), .RD(rd),.RegWrite(RegWrite), .clk(clk),
.reset(reset), .ReadData1(ReadData1), .ReadData2(ReadData2));



data_extractor de(.instruction(Instruction), .immdata(immdata));

control_unit cu(.ALUop(ALUop), .opcode(opcode), .ALUSrc(ALUSrc), .MemToReg(MemToReg),
.RegWrite(RegWrite), .MemRead(MemRead), .MemWrite(MemWrite), .Branch(Branch) );

ALU_control alc(.ALUop(ALUop),.funct({Instruction[30], funct3}), .operation(operation),
.Branchcontrol(Branchcontrol));
```

```
Data_memory dm(.WriteData(ReadData2), .Mem_Addr(ALU_Main_result), .MemWrite(MemWrite),
.clk(clk), .MemRead(MemRead), .ReadData1(readdatadm));


ALU_64_bit Alu_PC (.a(PC_out), .b(64'b0100),.ALUop(4'b0010), .result(ALU_PC), .zero(zero), .bgt(bgt),
.bne(bne));


ALU_64_bit Alu_shift (.a(PC_out), .b(immdata), .ALUop(4'b0010), .result(ALU_shift), .zero(zero),
.bgt(bgt), .bne(bne) );


ALU_64_bit Alu (.a(ReadData1), .b(mx_Offset), .ALUop(operation), .result(ALU_Main_result),
.zero(zero), .bgt(bgt), .bne(bne) );


mux mxbranch(.a(ALU_PC), .b(ALU_shift), .sel(Branch & (zero | bgt | bne)),.dataout(mx_Branch));


mux mxdm (.a(ALU_Main_result), .b(readdatadm), .sel(MemToReg), .dataout(mx_DM));


mux mxoffset (.a(ReadData2), .b(immdata), .sel(ALUSrc), .dataout(mx_Offset));

endmodule
```

TASK 2

Top module for pipelined processor:

Here we have added four pipeline registers, IF_ID, ID_EX, EX_Mem, Mem_WB. We have made separate modules for each register and have instantiated them in our top module and made the necessary connections.

```
module top(,

input clk, reset

);
```

```
wire [63:0] PC_in, PC_out;
wire [31:0] Instruction;
wire [6:0] opcode;
wire [4:0] rd;
wire [2:0] funct3;
wire [4:0] rs1;
wire [4:0] rs2;
wire [6:0] funct7;
wire [63:0] immdata, out, WriteData;
wire RegWrite;
wire [63:0]ReadData1, ReadData2;
wire ALUSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch;
wire [1:0] ALUop;
wire [3:0] operation;
wire [63:0]dataout;
wire [63:0] result;
wire [63:0] readdatadm;
wire [63:0] add_out;
wire write_data;
wire zero;

program_counter pc(.PC_in(PC_in), .clk(clk), .reset(reset), .PC_out(PC_out));

Instruction_memory im(.Instruction_address(PC_out), .Instruction(Instruction));

data_extractor de(.instruction(Instruction), .immdata(immdata));

ID_IF idif(.reset(reset), .clk(clk),
.PC_out(PC_out), .Instruction(Instruction),
```

```
.instruction_address(PC_out), .Instruction_out(Instruction));


//module 1


instruction_parser ip(.inst(Instruction),
.opcode(opcode),.rd(write_data),.funct3(funct3),.rs1(rs1),.rs2(rs2),.funct7(funct7));


registerFile rf(.WriteData(WriteData), .RS1(rs1), .RS2(rs2), .RD(write_register),.RegWrite(RegWrite),
.clk(clk), .reset(reset), .ReadData1(ReadData1), .ReadData2(ReadData2));


control_unit cu(.opcode(opcode), .ALUSrc(ALUSrc), .MemToReg(MemToReg), .RegWrite(RegWrite),
.MemRead(MemRead), .MemWrite(MemWrite), .Branch(Branch));\


//module 2

wire ALU_c_input;


ID_EX idex(.reset(reset), .clk(clk), .Instruction_address(PC_out),

.ALUSrc(ALUsrc), .MemToReg(MemToReg), .RegWrite(RegWrite),

.MemRead(MemRead), .MemWrite(MemWrite), .Branch(Branch), .ALUop(ALUop),// from cu

.readdata1(ReadData1), .readdata2(ReadData2) // from reg file

.Immidiate(immdata), .Instruction(Instruction),

.write_register_in(write_register), //


.instruction_address_out(PC_out),

.ALU_input(ReadData1), .mux_input1(ReadData2), .mux_input2(immdata),
.ALU_control_input(ALU_c_input),

.write_register_out(write_register), .wb(RegWrite), .m(MemWrite), .ex(ALUop),
.Instruction(Instruction);
```

```
adder a(.A(PC_out), .B(64'd4), .C(PC_in)); //Add sum

ALU_control alc(.ALUop(ALUop),.funct(ALU_c_input), .operation(operation));

mux mx1(.a(ReadData2), .b(immdata), .sel(ALUSrc),.dataout(dataout));

ALU_64_bit Alu(.a(ReadData1),.b(dataout),.ALUop(ALUop), .result(result), .zero(zero));

adder b(.A(PC_out), .B(immdata<<1), .C(add_out));

//module3
EX_MEM exmem(

.reset(reset), .clk(clk),
.output_adder(add_out), //from adder
.output_ALU(result), //from 64 bit alu result
.instruction(Instruction), // from last module
.readdata2(ReadData2), // from register file
.WBin(RegWrite), .Min(MemWrite), // regwrite nd memwrite
.zero(zero), //from aLU
.write_register_in(write_register)

.input_mux(add_out), //to mux before PC
.WriteData(ReadData2), .Mem_Addr(result), //to data memory
.instruction_out(Instruction), // from last module
.write_register_out(write_register) //from last module
.WB(RegWrite), .M(MemWrite)
);
```

```
Data_memory dm(.WriteData(ReadData2), .Mem_Addr(result), .MemWrite(MemWrite), .clk(clk),
.MemRead(MemRead), .ReadData1(readdatadm));


//module 4
MEM_WEB memweb(
.readdata(readdatadm), //from data memory
.output_ALU(result), //from last module
.write_register_in(write_register),
.wbin(RegWrite),


.input_mux1(readdatadm),
.input_mux2(result),
.write_data_out(write_register),
.instruction_out(Instruction),
.wb(RegWrite)
);
mux mx2(.a(readdatadm), .b(result), .sel(MemToReg),.dataout(WriteData));
mux mx3(.a(out), .b(add_out), .sel(Branch && zero),.dataout(PC_in));
endmodule
```

ID_IF Register:

```
module ID_IF(,
input reset, clk,
input [63:0] PC_out, //from PC
input [31:0] Instruction, //from instruction memory


output reg[63:0] instruction_address, //to adder
output reg [31:0] Instruction_out //to other modules
);
```

```
always @(posedge clk)



if (reset==1)
begin
Instruction_out = 31b'0;
instruction_address = 64b'0;

end
else
begin
Instruction_out = Instruction;
instruction_address = PC_out;
end
end
endmodule
```

ID_EX register:

```
module ID_EX(,

input reset, clk,
input [63:0] Instruction_address; //from PC
input ALUSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch,
input [1:0] ALUop //control signals
input [63:0] readdata1, readdata2, //from register files
input [63:0] Immidiate, //from immidiate generator
input [31:0] Instruction, //instruction memory, last module
input [3:0] write_register_in
```

```
output [63:0]instruction_address_out, //to adder
output [31:0] Instruction_out,
output [63:0] ALU_input, //to 64 bit ALU
output [63:0] mux_input1, //to input
output [63:0] mux_input2, //shifting ke bad into Adders
output [4:0] ALU_control_input, //from instruction specific index to ALU control
output [3:0] write_register_out, //instruction ka aik part, will directly go to the next module
output wb, m, ex,
);

always @(posedge clk)

if (reset==1)
begin
Instruction_address_out = 63b'0;
ALU_input = 63b'0;
mux_input1 = 63b'0;
mux_input2 = 63b'0;
ALU_control_input = 4b'0;
write_register_out= 4b'0;
Instruction_out = 31b'0;
wb = 1b'0;
m = 1b'0;
ex = 2b'0;

end
else
begin
```

```
Instruction_address_out = Instruction_address;
ALU_input = readdata1;
mux_input1 = readdata2;
mux_input2 = Immidiate;
ALU_control_input = Instruction[30, 14, 13, 12];
write_register_out = write_register_in;
Instruction_out = Instruction;
wb = RegWrite;
m = MemWrite;
ex = ALUop;
end
end
endmodule
```

EX_MEM:

```
module EX_MEM(,
input reset, clk,
input [63:0] output_adder, //from adder
input [63:0] output_ALU, //from 64 bit alu result
input [63:0] instruction, // from last module
input [63:0] readdata2, // from register file
input WBin, Min, // regwrite nd memwrite
input zero, //from ALU
input [3:0] write_register_in

output [63:0] input_mux, //to mux before PC
output [63:0] WriteData, Mem_Addr, //to data memory
output [63:0] instruction_out, // from last module
```

```
output [3:0] write_register_out //from last module
output WB, M,
);

always @(posedge clk)

if (reset==1)
begin
input_mux = 63b'0;
WriteData = 63b'0;
Mem_Addr = 63b'0;
instruction_out = 63b'0;
write_register_out= 4b'0;
WB = 1b'0;
M = 1b'0;

end
else
begin
input_mux = output_adder;
WriteData = readdata2;
Mem_Addr = output_ALU;
instruction_out = instruction;
write_register_out = write_register_in;
WB = WBin;
M= Min;

end
end
```

```
endmodule
```

MEM_WEB:

```
module MEM_WEB(,
input [63:0] readdata, //from data memory
input [63:0] output_ALU, //from last module
input [3:0] write_register_in,
input wbin,

output [63:0] input_mux1,
output [63:0] input_mux2,
output [63:0] instruction_out,
output [3:0] write_register_out,
output wb
);

always @(posedge clk)

if (reset==0)
begin
input_mux1 = 63b'0;
input_mux2 = 63b'0;
write_register_out = 4b'0;
instruction_out = 63b'0;
wb = 1b'0
end
else
begin
```

```
input_mux1 = readdata;

input_mux2 = ouput_ALU;

write_register_out = write_register_in;

instruction_out = instruction;

wb = wbin;

end

end

endmodule
```

TASK 3

We have made the module for the forwarding unit and made the necessary changes to our top module to include it into our pipelined processor. We also modified our multiplexers to account for 3 inputs and 2 bit select input line.

Forwarding:

```
module forwarding(,

input [63:0] MEM_WB_RegWrite, MEM_WEB_RegisterRd, EX_MEM_RegWrite, EX_MEM_RegisterRd,

ID_EX_RegisterRs1, ID_EX_RegisterRs2,

output [1:0] Forward_A, Forward_B;

);


always @(*)

if (EX_MEM_RegisterRd == ID_EX_RegisterRs1 && EX_MEM_RegWrite == 1 && EX_MEM_RegisterRd != 0)

begin

Forward_A = 2'b10;

end
```

```
// MEM Hazard (a)
else if (MEM_WB_RegisterRd == ID_EX_RegisterRs1 && MEM_WB_RegWrite == 1 &&
MEM_WB_RegisterRd != 0
&&
!(EX_MEM_RegWrite == 1 && EX_MEM_RegisterRd != 0 && EX_MEM_RegisterRd ==
ID_EX_RegisterRs1)
)
begin
Forward_A = 2'b01;
end

// No Hazard at mux A
else
begin
Forward_A = 2'b00;
end


//FORWARD B LOGIC

// EX Hazard (b)
if (EX_MEM_RegisterRd == ID_EX_RegisterRs2 && EX_MEM_RegWrite == 1 && EX_MEM_RegisterRd !=
0)
begin
Forward_B = 2'b10;
end

// MEM Hazard (b)
```

```
else if (MEM_WB_RegisterRd == ID_EX_RegisterRs2 && MEM_WB_RegWrite == 1 &&
MEM_WB_RegisterRd != 0

&&

!(EX_MEM_RegWrite == 1 && EX_MEM_RegisterRd != 0 && EX_MEM_RegisterRd ==
ID_EX_RegisterRs2)

)

begin

Forward_B = 2'b01;

end


// No Hazard at mux B

else

begin

Forward_B = 2'b00;

end

end

endmodule
```

3x1 MUX

```
module mux3(,

input a,b,c,

input [1:0] sel,

output reg out1


);

always @(*)

if (sel==2’b00)

out1 = a;
```

```
else if (sel == 2'b01)
out1 = b;
else if (sel == 2'b10)
out1 = c;
endmodule
```

TOP module for pipeline with forwarding:

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

wire [63:0] PC_in, PC_out;
wire [31:0] Instruction;
wire [6:0] opcode;
wire [4:0] rd;
wire [2:0] funct3;
wire [4:0] rs1;
wire [4:0] rs2;
wire [6:0] funct7;
wire [63:0] immdata, out, WriteData;
wire RegWrite;
wire [63:0]ReadData1, ReadData2;
wire ALUSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch;
wire [1:0] ALUop;
wire [3:0] operation;
wire [63:0]dataout;
wire [63:0] result;
wire [63:0] readdatadm;
```

```
wire [63:0] add_out;

wire write_data;

wire zero;


program_counter pc(.PC_in(PC_in), .clk(clk), .reset(reset), .PC_out(PC_out));


Instruction_memory im(.Instruction_address(PC_out), .Instruction(Instruction));


data_extractor de(.instruction(Instruction), .immdata(immdata));


ID_IF idif(.reset(reset), .clk(clk),

.PC_out(PC_out), .Instruction(Instruction),

.instruction_address(PC_out), .Instruction_out(Instruction));


//module 1 ka output will go in instruction_parser


instruction_parser ip(.inst(Instruction),
.opcode(opcode),.rd(write_data),.funct3(funct3),.rs1(rs1),.rs2(rs2),.funct7(funct7));


registerFile rf(.WriteData(WriteData), .RS1(rs1), .RS2(rs2), .RD(write_register),.RegWrite(RegWrite),
.clk(clk), .reset(reset), .ReadData1(ReadData1), .ReadData2(ReadData2));


control_unit cu(.opcode(opcode), .ALUSrc(ALUSrc), .MemToReg(MemToReg), .RegWrite(RegWrite),
.MemRead(MemRead), .MemWrite(MemWrite), .Branch(Branch));\


//module 2

wire ALU_c_input;



wire ALU_c_input;
```

```
wire MEM_WB_RegWrite;
wire MEM_WEB_RegisterRd;
wire EX_MEM_RegWrite;
wire EX_MEM_RegisterRd;
wire ID_EX_RegisterRs1;
wire ID_EX_RegisterRs2;


//write_back_register of exmem will become exmemregrd: done
//write_back_register of memweb will become memwebregrd : done
//rs1 and rs2 of IDex : done
// wb of memweb will become mem_web_regwrite
//wb of exmem will become ex_mem_regwrite

//module2

ID_EX idex(.reset(reset), .clk(clk), .Instruction_address(PC_out),
.ALUSrc(ALUsrc), .MemToReg(MemToReg), .RegWrite(RegWrite),
.MemRead(MemRead), .MemWrite(MemWrite), .Branch(Branch), .ALUop(ALUop),// from cu
.readdata1(ReadData1), .readdata2(ReadData2) // from reg file
.Immidiate(immdata), .Instruction(Instruction),
.write_register_in(write_register),
.rsin(ID_EX_RegisterRs1), .rsout(ID_EX_RegisterRs2),//

.instruction_address_out(PC_out),
.ALU_input(ReadData1), .mux_input1(ReadData2), .mux_input2(immdata),
.ALU_control_input(ALU_c_input),
.write_register_out(write_register), .wb(RegWrite), .m(MemWrite), .ex(ALUop), .Instruction(Instruction)
.rs1(ID_EX_RegisterRs1), rs2(ID_EX_RegisterRs2)
```

```
);

wire select_a, select_b;
forwarding forw(
.MEM_WB_RegWrite(MEM_WEB_RegWrite), .MEM_WEB_RegisterRd(MEM_WEB_RegisterRd),
.EX_MEM_RegWrite(EX_MEM_RegWrite),
.EX_MEM_RegisterRd(EX_MEM_RegisterRd), .ID_EX_RegisterRs1(ID_EX_RegisterRs1),
.ID_EX_RegisterRs2(ID_EX_RegisterRs1),
forwardA(select_a), forwardB(select_b);
);

wire data_memory_input;
wire result1, result2;

mux3 a(
.a(ID_EX_RegisterRs1),.b(Writedata),.c(data_memory_input),.sel(select_a),
.out1(result1) //mux output
);

mux3 b(
.a(ID_EX_RegisterRs2),.b(Writedata),.c(data_memory_input),.sel(select_b),
.out1(result2)
);

adder a(.A(PC_out), .B(64'd4), .C(PC_in)); //Add sum

ALU_control alc(.ALUop(ALUop),.funct(ALU_c_input), .operation(operation));

mux mx1(.a(ReadData2), .b(immdata), .sel(ALUSrc),.dataout(dataout));
```

```
ALU_64_bit Alu(.a(result1),.b(result2),.ALUop(ALUop), .result(result), .zero(zero));

adder b(.A(PC_out), .B(immdata<<1), .C(add_out));

//module3
EX_MEM exmem(

.reset(reset), .clk(clk),
.output_adder(add_out), //from adder
.output_ALU(result), //from 64 bit alu result
.instruction(Instruction), // from last module
.readdata2(ReadData2), // from register file
.WBin(RegWrite), .Min(MemWrite), // regwrite nd memwrite
.zero(zero), //from aLU
.write_register_in(write_register)

.input_mux(add_out), //to mux before PC
.WriteData(ReadData2), .Mem_Addr(data_memory_input), //to data memory
.instruction_out(Instruction), // from last module
.write_register_out(EX_MEM_RegisterRd) //from last module
.WB(EX_MEM_RegWrite), .M(MemWrite)
);

Data_memory dm(.WriteData(ReadData2), .Mem_Addr(data_memory_input), .MemWrite(MemWrite),
.clk(clk), .MemRead(MemRead), .ReadData1(readdatadm));

//module 4
```

```
MEM_WEB memweb(
.readdata(readdatadm), //from data memory
.output_ALU(data_memory_input), //from last module
.write_register_in(EX_MEM_RegisterRd),
.wbin(EX_MEM_RegWrite),

.input_mux1(readdatadm),
.input_mux2(result),
.write_data_out(MEM_WEB_RegisterRd),
.instruction_out(Instruction),
.wb(MEM_WB_RegWrite)
);
mux mx2(.a(readdatadm), .b(result), .sel(MemToReg),.dataout(WriteData));
mux mx3(.a(out), .b(add_out), .sel(Branch && zero),.dataout(PC_in));
endmodule
```