**CODE**

**IF.v**

`timescale 1ns/1ns

//`include "Instruction\_mem.v"

module IF #(parameter BOOT\_ADDRESS=32'b0)

(clk,rst,instr,pc,target\_pc,Branch\_taken);

input clk,rst,Branch\_taken;

input [31:0]target\_pc;

output [31:0]instr;//this is stored in a register , to help pipelining

output reg [31:0]pc;//always starts at BOOT\_ADDRESS

always@(posedge clk)

begin

if(rst)

pc=BOOT\_ADDRESS;

else

if(Branch\_taken)

pc = target\_pc;

else pc = pc+4;

end

Instruction\_mem inst\_mem(.clk(clk),.address(pc[12:2]),.instr(instr));

endmodule

**Instrn memory.v**

`timescale 1ns/1ns

module Instruction\_mem(clk,address,instr);

input clk;

input [10:0]address;

output reg[31:0]instr;

reg [31:0] instruction\_memory [0:39]; //capable of storing 40 instructions, each 32 bits wide

initial $readmemb("ISA1.txt",instruction\_memory);//instructions are stored in 'ISA1.txt', loading instrns into the instruction\_memory

always@(posedge clk)

begin

instr <= instruction\_memory[address];// value .. Mapping from +4 to +1

end

endmodule

**ID.v**

`timescale 1ns/1ns

module ID(clk,Ra,Rb,Rd,reg\_write\_data,reg\_wr,reg\_read\_data1,reg\_read\_data2,instr,imm\_ext);

input clk;

input [4:0]Ra;

input [4:0]Rb;

input [4:0]Rd; //right now this Rd is directly being sent to the register file but it should actually be the output of a mux

input [31:0]reg\_write\_data; //even this

input reg\_wr;

input [31:0] instr;//Design philosophy: all the input control signals to a stage are cosidered to be wired inputs

output [31:0]reg\_read\_data1;

output [31:0]reg\_read\_data2;

output [31:0] imm\_ext;

reg\_file reg\_file(.clk(clk),.reg\_out\_1(reg\_read\_data1),.reg\_out\_2(reg\_read\_data2),.reg\_addr1(Ra),.reg\_addr2(Rb),.reg\_write\_addr(Rd),.reg\_wr(reg\_wr),.reg\_din(reg\_write\_data));

sign\_ext sign\_ext(instr,imm\_ext);

endmodule

**Reg\_file.v**

`timescale 1ns / 1ns

module reg\_file(clk,reg\_out\_1,reg\_out\_2,reg\_addr1,reg\_addr2,reg\_write\_addr,reg\_wr,reg\_din);

input clk;

input [4:0] reg\_addr1,reg\_addr2,reg\_write\_addr;

input [31:0] reg\_din;

input reg\_wr;

output reg [31:0] reg\_out\_1,reg\_out\_2;

reg [31:0] registers [31:0];

//reg [31:0] temp\_din;

wire clk\_bar;

// integer i; //comment while synthesising

assign clk\_bar=~clk;

always @(reg\_addr1,reg\_addr2)

begin

reg\_out\_1 <= registers[reg\_addr1];

reg\_out\_2 <= registers[reg\_addr2];

end

always@(posedge clk\_bar)

begin

if(reg\_wr)registers[reg\_write\_addr]=reg\_din;

//if(reg\_wr)temp\_din=reg\_din;

end

//comment while synthesising

//always@(\*)

//begin

// for(i=0;i<32;i=i+1)

// begin

// $display("%d",registers[i]);

// end

// $display();

//end

//Need to remove this part from here and should find a different means of gettting these initial values in

initial begin

{reg\_out\_1,reg\_out\_2} = {32'h00000001,32'h00000002};

registers[5'b00000]<=32'h00000003; //according to RISC-V ISA address 0 must be bound to zero value

registers[5'b00001]<=32'h00000005;

registers[5'b00011]<=32'hfffffffC;

registers[5'b00100]<=32'h0000000A;

registers[5'b11111]<=32'h0000000C;

end

endmodule

**Sign\_ext.v**

`timescale 1ns/1ns

module sign\_ext(instr,imm\_ext);

input [31:0] instr;

output reg [31:0] imm\_ext;

reg [11:0]imm;

always@(instr[6:0]) begin

case(instr[6:0])

7'b0100011: //sw

imm={instr[31:25],instr[11:7]};

7'b0000011: //lw

imm=instr[31:20];

endcase

end

always@(imm) begin

if(imm[11]==1)

imm\_ext={20'b1,imm};

else

imm\_ext={20'b0,imm};

end

endmodule

**Control\_unit.v**

`timescale 1ns/1ns

module control\_unit(Op,RegWrite,ALUOp,MemRead,MemWrite,MemtoReg,is\_Branch,ALUSrc,ALURes\_ctrl);

input [6:0]Op; // instruction[6:0]

output reg RegWrite;

output reg [1:0] ALUOp;

output reg MemRead;

output reg MemWrite;

output reg MemtoReg;

output reg is\_Branch; //later used to compute PCSrc

output reg ALUSrc;

output reg ALURes\_ctrl; //modification for U format

always @(Op)

begin

case(Op)

7'b0110011: //R format

begin

RegWrite<=1;

ALUOp<=2'b10;

MemRead<=0;

MemWrite<=0;

MemtoReg<=0;

is\_Branch<=0;

ALUSrc<=0;

ALURes\_ctrl<=0;

end

7'b0000011: //lw format

begin

RegWrite<=1;

ALUOp<=2'b00;

MemRead<=1;

MemWrite<=0;

MemtoReg<=1;

is\_Branch<=0;

ALUSrc<=1;

ALURes\_ctrl<=0;

end

7'b0100011: //S type

begin

RegWrite<=0;

ALUOp<=2'b00;

MemRead<=0;

MemWrite<=1;

MemtoReg<=1'bx;

is\_Branch<=0;

ALUSrc<=1;

ALURes\_ctrl<=0;

end

7'b0110111: //LUI

begin

RegWrite<=1;

ALUOp<=2'bxx;

MemRead<=0;

MemWrite<=0;

MemtoReg<=0;

is\_Branch<=0;

ALUSrc<=1'bx;

ALURes\_ctrl<=1;

end

7'b0010111: //AUIPC

begin

RegWrite<=1;

ALUOp<=2'bxx;

MemRead<=0;

MemWrite<=0;

MemtoReg<=0;

is\_Branch<=0;

ALUSrc<=1'bx;

ALURes\_ctrl<=1;

end

7'b1100011: //SB type

begin

RegWrite<=0;

ALUOp<=2'b01;

MemRead<=0;

MemWrite<=0;

MemtoReg<=1'bx;

is\_Branch<=1;

ALUSrc<=0;

ALURes\_ctrl<=0;

end

7'b0010011: //I-format (verify the values of the signals )

begin

RegWrite<=1;

ALUOp<=2'b11; //(verify bit value)

MemRead<=0;

MemWrite<=0;

MemtoReg<=0;

is\_Branch<=0;

ALUSrc<=1;

ALURes\_ctrl<=0;

end

default:

begin

RegWrite<=1'bx;

ALUOp<=2'bxx;

MemRead<=1'bx;

MemWrite<=1'bx;

MemtoReg<=1'bx;

is\_Branch<=1'bx;

ALUSrc<=1'bx;

ALURes\_ctrl<=1'bx;

end

endcase

end

endmodule

**EX.v**

`timescale 1ns/1ns

//`include "alu.v"

//`include "alu\_control.v"

module EX(A,B,imm\_ext,Instr,ALUSrc,is\_Branch,ALUOp,Result,ALURes\_ctrl,Branch\_taken,pc,target\_pc);

input ALUSrc,is\_Branch;

input [31:0]A,B,Instr;

input [1:0]ALUOp;

input [31:0]imm\_ext,pc;

input ALURes\_ctrl; //modification for U format...Control signal to choose between the 2 outputs viz. result and U\_intermediate

output [31:0]Result;

output [31:0]target\_pc;

output reg Branch\_taken;

wire [3:0]ALU\_SEL;

wire [31:0]operand2;

wire zero;

wire [31:0]alu\_result;

reg [31:0]U\_intermediate;

alu\_control alu\_control(.func\_field({Instr[31:25],Instr[14:12]}),.ALUOp(ALUOp),.ALU\_SEL(ALU\_SEL));

ALU\_32bit alu(.ALU\_SEL(ALU\_SEL),.A(A),.B(operand2),.ALU\_OUT(alu\_result),.carry(),.zero(zero),.negative(),.overflow(),.underflow());

//mux to choose between register read value and immediate

assign operand2=ALUSrc?imm\_ext:B;

//Adder for branch target address calculation

assign target\_pc= pc + (imm\_ext<<1);//imm is twice the number of instructions we want to jump

//Is branch taken? {and gate}

always @(\*)

begin

case (Instr[14:12])//funct3

3'b000: Branch\_taken = is\_Branch & zero;//eq

3'b001: Branch\_taken = is\_Branch & ~zero;//ne

3'b100: Branch\_taken = is\_Branch & Result[0];//lt

3'b101: Branch\_taken = is\_Branch & ~Result[0];//ge

3'b110: Branch\_taken = is\_Branch & Result[0];//ltu

3'b111: Branch\_taken = is\_Branch & ~Result[0];//geu

default: Branch\_taken = 1'b0;

endcase

end

always @(Instr[6:0]) //Dealing with U format instrns

case (Instr[6:0])

7'b0110111: //lui

U\_intermediate<={Instr[31:12],12'b0};

7'b0010111: //auipc

U\_intermediate<={Instr[31:12],12'b0}+pc;

default: U\_intermediate<=32'bx;

endcase

assign Result= ALURes\_ctrl ? U\_intermediate : alu\_result;

endmodule

**alu.v**

`timescale 1ns / 1ns

module ALU\_32bit(

//port declarations

input [3:0]ALU\_SEL,

input [31:0]A,B,

output reg [31:0]ALU\_OUT,

output carry,

output zero,negative,overflow,underflow

);

wire [32:0] tmp;

assign tmp = {1'b0,A} + {1'b0,B};

assign carry = tmp[32]; // Carryout flag

assign zero = ~(|ALU\_OUT); //zero flag

assign negative = (ALU\_OUT[31] == 1 && (ALU\_SEL == 4'b0110));

assign overflow = ({carry,ALU\_OUT[31]} == 2'b01);

assign underflow = ({carry,ALU\_OUT[31]} == 2'b10);

always@(\*)

begin

case(ALU\_SEL)

0: ALU\_OUT<= A & B;

1: ALU\_OUT<= A | B;

2: ALU\_OUT<= A + B;

6: ALU\_OUT<= A - B;

7: ALU\_OUT<= A < B ? 32'b1 : 32'b0; //stlu

8: ALU\_OUT<= (A[31] ^ B[31])? {31'b0,A[31]}:{31'b0,(A<B)}; //slt

12: ALU\_OUT<= -(A|B); //nor

default: ALU\_OUT<=32'bx;

/\* 4'b0000: // AdditionOK

ALU\_OUT <= A + B ;

4'b0001: // SubtractionOK

ALU\_OUT <= B - A ;

4'b0010: // setlessthan

ALU\_OUT <= (A<B)?32'd1:32'd0;

4'b0011: // setlessthan unsigned

ALU\_OUT <= (A>B)?32'd1:32'd0;

4'b0100: // Logical shift left

ALU\_OUT <= A<<1;

4'b0101: // Logical shift right

ALU\_OUT <= A>>1;

4'b0110: // shiftrightarithmetic(if the bits are signed,otherwise it is same as right shift)

ALU\_OUT <= A>>>1;

4'b0111: // logicl and

ALU\_OUT <= A & B;

4'b1000: // Logical or

ALU\_OUT <= A | B;

4'b1001: // Logical xor

ALU\_OUT <= A ;

default: ALU\_OUT <= A + B;

\*/

endcase

end

endmodule

**alu\_control.v**

`timescale 1ns / 1ns

module alu\_control(func\_field,ALUOp,ALU\_SEL);

input [9:0] func\_field; // Instruction[31:25(func7),12-14(func3)]

input [1:0] ALUOp; //comes from the Main control Unit

output reg [3:0] ALU\_SEL; //O/p of the ALU control goes to the ALU to perform the specified operation

always @(ALUOp)

case(ALUOp)

2'b00: //lw,sw instn

//Irrespective of the func field, it generates an add control signal[to calculate the address]

ALU\_SEL<=4'b0010;

2'b01: //branch instn

case(func\_field[2:0])

3'b000: //beq

ALU\_SEL<=4'b0110;

3'b001: //bne

ALU\_SEL<=4'b0110;

3'b100: //blt

ALU\_SEL<=4'b1000;

3'b101: //bge

ALU\_SEL<=4'b1000;

3'b110: //bltu

ALU\_SEL<=4'b0111;

3'b111: //bgeu

ALU\_SEL<=4'b0111;

default:

ALU\_SEL<=4'bz;

endcase

2'b10: //R-type instruction

case(func\_field)

10'b0000000000: //add

ALU\_SEL<=4'b0010;

10'b0100000000: //sub

ALU\_SEL<=4'b0110;

10'b0000000111: //AND

ALU\_SEL<=4'b0000;

10'b0000000110: //OR

ALU\_SEL<=4'b0001;

10'b0000000010: //slt

ALU\_SEL<=4'b1000;

10'b0000000011: //sltu

ALU\_SEL<=4'b0111;

default: ALU\_SEL<=4'bz;

endcase

2'b11: //I -type instruction ( verify the bit value)

case(func\_field)

10'bxxxxxxx000: //addimm

ALU\_SEL<=4'b0000;

10'bxxxxxxx111: //ANDI

ALU\_SEL<=4'b0111;

10'bxxxxxxx110: //ORI

ALU\_SEL<=4'b1000;

10'bxxxxxxx010: //SLTI\*

ALU\_SEL<=4'b0010;

10'bxxxxxxx011: //SLTIU\*

ALU\_SEL<=4'b0011;

10'bxxxxxxx100: //XORI

ALU\_SEL<=4'b1001;

10'b0000000001: //SLLI\*

ALU\_SEL<=4'b0100;

10'b0000000101: //SRLI\*

ALU\_SEL<=4'b0101;

10'b0100000101: //SRAI\*

ALU\_SEL<=4'b0110;

default:

ALU\_SEL<=4'bz;

endcase

default: ALU\_SEL<=4'bz;

endcase

endmodule

**MEM.v**

`timescale 1ns/1ns

module MEM(clk,mem\_read\_ctrl,mem\_write\_ctrl,mem\_address,mem\_data\_write,mem\_data\_read);

input mem\_read\_ctrl;

input mem\_write\_ctrl;

input clk;

wire clk\_bar;

//integer i;

//declaring these two as wires as they are being fed by the pipeline register EX\_MEM

input [31:0] mem\_address;

input [31:0] mem\_data\_write;

output reg [31:0]mem\_data\_read;

//clock bar

assign clk\_bar=~clk;

//defining ram within this module itself

reg [31:0] mem\_ram [0:39]; //Need to decide upon the number of words in the memory

always@(posedge clk\_bar)

begin

if(mem\_read\_ctrl && !mem\_write\_ctrl)

mem\_data\_read=mem\_ram[mem\_address];

if(!mem\_read\_ctrl && mem\_write\_ctrl)

mem\_ram[mem\_address]=mem\_data\_write;

end

//always@\*

//for(i=0;i<40;i=i+1)

//begin

// $display("%d",mem\_ram[i]);

//end

endmodule

**WB.v**

`timescale 1ns/1ns

module WB(mem\_to\_reg,Mem\_read\_out,ALUresult,out);

input mem\_to\_reg;

input [31:0]Mem\_read\_out,ALUresult;

output [31:0]out;

assign out = mem\_to\_reg ? Mem\_read\_out : ALUresult;

endmodule

**Processor.v**

`timescale 1ns/1ns

module processor #(parameter BOOT\_ADDRESS=32'b0)

(clk,rst,data\_in,mem\_read\_ctrl,mem\_write\_ctrl,address\_out,data\_out);

input clk,rst;

input [31:0]data\_in;

output wire mem\_read\_ctrl,mem\_write\_ctrl;

output wire [31:0]address\_out,data\_out;

//pipeline registers for storing data

wire [31:0]IF\_ID\_wire[0:1]; //index0=instruction ,index1=pc

reg [31:0]IF\_ID[0:1];

wire [31:0]ID\_EX\_wire[0:4]; //index0=instruction ,index1=Read\_out\_1 ,index2=Read\_out2 , index3=imm\_ext , index4=pc

reg [31:0]ID\_EX[0:4];

wire [31:0]EX\_MEM\_wire[0:2]; //index0=UNUSED ,index1=ALUresult ,index2=UNUSED

reg [31:0]EX\_MEM[0:2]; //index0=instruction ,index1=ALUresult ,index2=MEM\_data\_in\_for\_ld

wire [31:0]MEM\_WB\_wire[0:2]; //index0=UNUSED ,index1=UNUSED ,index2=MEM\_read\_data

reg [31:0]MEM\_WB[0:2]; //index0=instruction ,index1=ALUresult ,index2=MEM\_read\_data

//pipeline registers to store control signals

wire ID\_EX\_ctrl\_wire[0:8]; //index0=ALU\_src, index1 and 2=ALUOp, index3=mem\_write, index4=mem\_Read, index5=mem\_to\_reg, index6=Reg\_write, index7=ALURes\_ctrl ,index8=Branch: Ordering Philosophy is based on the ascending order of their entry point into a stage

reg ID\_EX\_ctrl[0:8];

reg EX\_MEM\_ctrl[0:3]; //index0=mem\_write, index1=mem\_Read, index2=mem\_to\_reg, index3=Reg\_write

wire Branch\_taken;

wire [31:0]target\_pc;

reg MEM\_WB\_ctrl[0:1]; //index0=mem\_to\_reg, index1=Reg\_write

//temporary

wire [31:0]temp1;

//Every stage gets a new output only the at posedge of the clk.

IF #(.BOOT\_ADDRESS(BOOT\_ADDRESS))IF(.clk(clk),.rst(rst),.instr(IF\_ID\_wire[0]),.pc(IF\_ID\_wire[1]),.target\_pc(target\_pc),.Branch\_taken(Branch\_taken)); //outputs of a module cannot be driven by the 'reg' types

control\_unit control(.Op(IF\_ID[0][6:0]),.RegWrite(ID\_EX\_ctrl\_wire[6]),.ALUOp({ID\_EX\_ctrl\_wire[1],ID\_EX\_ctrl\_wire[2]}),.MemRead(ID\_EX\_ctrl\_wire[4]),.MemWrite(ID\_EX\_ctrl\_wire[3]),.MemtoReg(ID\_EX\_ctrl\_wire[5]),.is\_Branch(ID\_EX\_ctrl\_wire[8]),.ALUSrc(ID\_EX\_ctrl\_wire[0]),.ALURes\_ctrl(ID\_EX\_ctrl\_wire[7]));

ID ID(.clk(clk),.Ra(IF\_ID[0][19:15]),.Rb(IF\_ID[0][24:20]),.Rd(MEM\_WB[0][11:7]),.reg\_write\_data(temp1),.reg\_wr(MEM\_WB\_ctrl[1]),.reg\_read\_data1(ID\_EX\_wire[1]),.reg\_read\_data2(ID\_EX\_wire[2]),.instr(IF\_ID[0]),.imm\_ext(ID\_EX\_wire[3]));

EX EX(.A(ID\_EX[1]),.B(ID\_EX[2]),.imm\_ext(ID\_EX[3]),.Instr(ID\_EX[0]),.ALUSrc(ID\_EX\_ctrl[0]),.is\_Branch(ID\_EX\_ctrl[8]),.ALUOp({ID\_EX\_ctrl[1],ID\_EX\_ctrl[2]}),.Result(EX\_MEM\_wire[1]),.ALURes\_ctrl(ID\_EX\_ctrl[7]),.Branch\_taken(Branch\_taken),.pc(ID\_EX[4]),.target\_pc(target\_pc));

//MEM MEM(.mem\_read\_ctrl(EX\_MEM\_ctrl[1]),.mem\_write\_ctrl(EX\_MEM\_ctrl[0]),.mem\_address(EX\_MEM[1]),.mem\_data\_write(EX\_MEM[2]),.mem\_data\_read(MEM\_WB\_wire[2]));

WB WB(.mem\_to\_reg(MEM\_WB\_ctrl[0]),.Mem\_read\_out(MEM\_WB[2]),.ALUresult(MEM\_WB[1]),.out(temp1));

assign mem\_read\_ctrl=EX\_MEM\_ctrl[1];

assign mem\_write\_ctrl=EX\_MEM\_ctrl[0];

assign address\_out=EX\_MEM[1];

assign data\_out=EX\_MEM[2];

assign MEM\_WB\_wire[2]=data\_in;

//As soon as a particular stage produces result the output needs to be stored into a pipeline register.

//synchronous reset

always@(posedge clk) begin

if(rst)

begin

{IF\_ID[0],IF\_ID[1]}<={32'bz,32'bz};

{ID\_EX[0],ID\_EX[1],ID\_EX[2],ID\_EX[3],ID\_EX[4]}<={32'bz,32'bz,32'bz,32'bz,32'bz}; //not sure if pc=ID\_EX[4] shd be z or no!

{EX\_MEM[0],EX\_MEM[1],EX\_MEM[2]}<={32'bz,32'bz,32'bz};

{MEM\_WB[0],MEM\_WB[1],MEM\_WB[2]}<={32'bz,32'bz,32'bz};

{ID\_EX\_ctrl[0],ID\_EX\_ctrl[1],ID\_EX\_ctrl[2],ID\_EX\_ctrl[3],ID\_EX\_ctrl[4],ID\_EX\_ctrl[5],ID\_EX\_ctrl[6],ID\_EX\_ctrl[7],ID\_EX\_ctrl[8]}<={1'bz,1'bz,1'bz,1'bz,1'bz,1'bz,1'bz,1'bz};

{EX\_MEM\_ctrl[0],EX\_MEM\_ctrl[1],EX\_MEM\_ctrl[2],EX\_MEM\_ctrl[3]}<={1'bz,1'bz,1'bz,1'bz};

{MEM\_WB\_ctrl[0],MEM\_WB\_ctrl[1]}<={1'bz,1'bz};

end

else begin

{IF\_ID[0],IF\_ID[1]}<={IF\_ID\_wire[0],IF\_ID\_wire[1]};

{ID\_EX[0],ID\_EX[1],ID\_EX[2],ID\_EX[3],ID\_EX[4]}<={IF\_ID[0],ID\_EX\_wire[1],ID\_EX\_wire[2],ID\_EX\_wire[3],IF\_ID[1]};

{EX\_MEM[0],EX\_MEM[1],EX\_MEM[2]}<={ID\_EX[0],EX\_MEM\_wire[1],ID\_EX[2]};

{MEM\_WB[0],MEM\_WB[1],MEM\_WB[2]}<={EX\_MEM[0],EX\_MEM[1],MEM\_WB\_wire[2]};

{ID\_EX\_ctrl[0],ID\_EX\_ctrl[1],ID\_EX\_ctrl[2],ID\_EX\_ctrl[3],ID\_EX\_ctrl[4],ID\_EX\_ctrl[5],ID\_EX\_ctrl[6],ID\_EX\_ctrl[7],ID\_EX\_ctrl[8]}<={ID\_EX\_ctrl\_wire[0],ID\_EX\_ctrl\_wire[1],ID\_EX\_ctrl\_wire[2],ID\_EX\_ctrl\_wire[3],ID\_EX\_ctrl\_wire[4],ID\_EX\_ctrl\_wire[5],ID\_EX\_ctrl\_wire[6],ID\_EX\_ctrl\_wire[7],ID\_EX\_ctrl\_wire[8]};

{EX\_MEM\_ctrl[0],EX\_MEM\_ctrl[1],EX\_MEM\_ctrl[2],EX\_MEM\_ctrl[3]}<={ID\_EX\_ctrl[3],ID\_EX\_ctrl[4],ID\_EX\_ctrl[5],ID\_EX\_ctrl[6]};

{MEM\_WB\_ctrl[0],MEM\_WB\_ctrl[1]}<={EX\_MEM\_ctrl[2],EX\_MEM\_ctrl[3]};

end

if(Branch\_taken) begin

{IF\_ID[0],IF\_ID[1]}<={32'bz,32'bz};

{ID\_EX[0],ID\_EX[1],ID\_EX[2],ID\_EX[3]}<={32'bz,32'bz,32'bz,32'bz};

{ID\_EX\_ctrl[0],ID\_EX\_ctrl[1],ID\_EX\_ctrl[2],ID\_EX\_ctrl[3],ID\_EX\_ctrl[4],ID\_EX\_ctrl[5],ID\_EX\_ctrl[6],ID\_EX\_ctrl[7],ID\_EX\_ctrl[8]}<={1'bz,1'bz,1'bz,1'bz,1'bz,1'bz,1'bz,1'bz};

end

end

endmodule

**Processor\_tb.v**

`timescale 1ns/1ns

//`include "processor.v"

module processor\_tb;

reg clk;

reg rst;

wire [31:0]data\_in,data\_out,address\_out;

wire mem\_read\_ctrl;

wire mem\_write\_ctrl;

parameter BOOT\_ADDRESS=32'b0;

processor #(.BOOT\_ADDRESS(BOOT\_ADDRESS)) processor(.clk(clk),.rst(rst),.data\_in(data\_in),.mem\_read\_ctrl(mem\_read\_ctrl),.mem\_write\_ctrl(mem\_write\_ctrl),.address\_out(address\_out),.data\_out(data\_out));

MEM MEM(.clk(clk),.mem\_read\_ctrl(mem\_read\_ctrl),.mem\_write\_ctrl(mem\_write\_ctrl),.mem\_address(address\_out),.mem\_data\_write(data\_out),.mem\_data\_read(data\_in));

always #20 clk=~clk; //Need to decide the time period

initial begin

rst=1;

clk=0;

#30 rst=0;

#400

$finish;

end

endmodule