/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\*\*\* WRITE THE CODE OF 1-BIT ALU HERE \*\*\*/

/\* 1- BIT ALU CODE BEGINS HERE \*/

module oneBitALU (Result, Co, a, b, Cin, b\_invert, operation);

input a, b, Cin, b\_invert;

input [1:0] operation;

output Result, Co;

wire xorBBinv, aB, a\_b, sum;

xor (xorBBinv, b, b\_invert);

and (aB, a, xorBBinv);

or (a\_b, a, xorBBinv);

fullAdder FA1 (a, xorBBinv, Cin, Co, sum);

fourXOneMux ALUMux (Result, aB, a\_b, sum, 1'b0, operation[0], operation[1]);

endmodule

/\* 1- BIT ALU CODE ENDS HERE \*/

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

module halfAdder (S,C,x,y);

input x,y;

output S,C;

//Instantiate primitive gates

xor (S,x,y);

and (C,x,y);

endmodule

module fullAdder(x,y,z,C,S); //full adder with cascading halfadders

input x,y,z;

output S,C;

wire S1,D1,D2; //Outputs of first XOR and two AND gates

//Instantiate the halfadders

halfAdder HA1 (S1,D1,x,y),

HA2 (S,D2,S1,z);

or g1(C,D2,D1);

endmodule

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\*\*\* WRITE THE CODE OF 4x1 MUX HERE (FROM LAB 7 PART 1a. \*\*\*/

/\* 4x1 MUX CODE BEGINS HERE \*/

module fourXOneMux (Y, I0, I1, I2, I3, S0, S1);

output Y;

input I0, I1, I2, I3, S0, S1;

wire T0, T1, T2, T3, S0bar, S1bar;

not (S0bar, S0);

not (S1bar, S1);

and (T0, I0, S1bar, S0bar);

and (T1, I1, S1bar, S0);

and (T2, I2, S1, S0bar);

and (T3, I3, S1, S0);

or (Y, T0, T1, T2, T3);

endmodule

/\* 4x1 MUX CODE ENDS HERE \*/

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\*\*\* WRITE THE CODE OF 4-BIT ALU HERE \*\*\*/

/\* 4- BIT ALU CODE BEGINS HERE \*/

module fourBitALU (Result, overflow, A, B, Operation);

input [2:0] Operation;

input [3:0] A, B;

output [3:0] Result;

output overflow;

wire C0, C1, C2;

oneBitALU ALU0 (Result[0], C0, A[0], B[0], Operation[2], Operation[2], Operation[1:0]);

oneBitALU ALU1 (Result[1], C1, A[1], B[1], C0, Operation[2], Operation[1:0]);

oneBitALU ALU2 (Result[2], C2, A[2], B[2], C1, Operation[2], Operation[1:0]);

oneBitALU ALU3 (Result[3], overflow, A[3], B[3], C2, Operation[2], Operation[1:0]);

endmodule

/\* 4- BIT ALU CODE ENDS HERE \*/

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\* WRITE THE BEHAVIORAL CODE OF LI DECODER HERE \*/

// LI decoder. 0 if 100, 1 otherwise

module liDecoder (dectomux, opcode8, opcode7, opcode6);

input opcode8, opcode7, opcode6;

output dectomux;

assign dectomux = (opcode8 == 1'b1 && opcode7 == 1'b0 && opcode6 == 1'b0)? 1'b0 : 1'b1;

endmodule

/\* LI DECODER CODE ENDS HERE \*/

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\* WRITE THE BEHAVIORAL CODE OF QUADRUPLE 2x1 MUX HERE \*/

// QUADRUPLE 2x1 MUX CODE STARTS HERE

module quadTwoXOneMux (Y, I0, I1, S);

input S;

input [3:0] I0, I1;

output [3:0] Y;

//reg [3:0] Y;

assign Y = (S)? I1 : I0;

// always @ (S)

// case (S)

// 1'b0:

// Y = I0;

// 1'b1:

// Y = I1;

// endcase

endmodule

/\* QUADRUPLE 2x1 MUX CODE ENDS HERE \*/

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\* 1-BIT D-FF STARTS HERE: BEHAVIORAL MODELING (NEG EDGE TRIGGER)\*/

//D flip-flop

module DLatch(D, En, Q);

input D, En;

output Q;

wire Dbar, T1, T2, Qbar;

not (Dbar, D);

nand (T1, D, En);

nand (T2, Dbar, En);

nand (Q, T1, Qbar);

nand (Qbar, T2, Q);

endmodule

module DFlipFlop(D, En, Q);

input D, En;

output Q;

wire EnBar, Y;

not (EnBar, En);

DLatch DL1 (D, En, Y);

DLatch DL2 (Y, EnBar, Q);

endmodule

//1- BIT DFF ENDS HERE

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\* 9-bit INSTRUCTION REGISTRER STARTS HERE: USING 1-bit DFF\*/

// Instruction register. Must be completed.

module NineBitRegister (In, CLK, Out);

input[8:0] In;

input CLK;

output[8:0] Out;

DFlipFlop DFF0 (In[0], CLK, Out[0]);

DFlipFlop DFF1 (In[1], CLK, Out[1]);

DFlipFlop DFF2 (In[2], CLK, Out[2]);

DFlipFlop DFF3 (In[3], CLK, Out[3]);

DFlipFlop DFF4 (In[4], CLK, Out[4]);

DFlipFlop DFF5 (In[5], CLK, Out[5]);

DFlipFlop DFF6 (In[6], CLK, Out[6]);

DFlipFlop DFF7 (In[7], CLK, Out[7]);

DFlipFlop DFF8 (In[8], CLK, Out[8]);

endmodule

/\* INSTRUCTION REGISTER ENDS HERE \*/

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\* 4-bit register file implemented in behavioral modeling (don't change it).\*/

module regfile (ReadReg1,ReadReg2,WriteReg,WriteData,ReadData1,ReadData2,CLK);

input [1:0] ReadReg1,ReadReg2,WriteReg;

input [3:0] WriteData;

input CLK;

output [3:0] ReadData1,ReadData2;

reg [3:0] Regs[0:3];

assign ReadData1 = Regs[ReadReg1];

assign ReadData2 = Regs[ReadReg2];

initial Regs[0] = 0;

always @(negedge CLK)

Regs[WriteReg] <= WriteData;

endmodule

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

// FINAL CPU MODULE STARTS HERE \*//

// FINISH THE CODE //

module cpu (Instruction, WriteData, CLK);

input [8:0] Instruction; // 9bits for mips instructions

input CLK; // clock

output [3:0] WriteData; // 4bit output of cp

wire [8:0] IR; // instruction register

// Declare more wires as needed

wire dectomux, oF;

wire [3:0] result;

wire [3:0] A, B; // ReadData Wires

/\* INSTANTIATE ALL THE MODULES AS PER FIGURE 1. The OP code and other fiels of this register that have to be

passed to other modules must be represented by their respective

indices (see the register file instance below).

\*/

NineBitRegister register (Instruction, CLK, IR);

// Define the module for the quadruple 2x1 mux and instatiate it here.

quadTwoXOneMux mux (WriteData, IR[5:2], result, dectomux);

// Define the module for the LI decoder and instantiate it here.

liDecoder li (dectomux, IR[8], IR[7], IR[6]);

// register file (the module definition is incuded in this file)

regfile regs(IR[5:4],IR[3:2],IR[1:0], WriteData, A, B, CLK);

// Define a module for the ALU and instantiate it here.

fourBitALU fBALU (result, oF, A, B, IR[8:6]);

endmodule

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\* THIS IS THE TESTBENCH. DO NOT CHANGE ANYTHING HERE. USE IT TO TEST YOUR CIRCUIT\*/

// Test module. Add more instructions as provided in the test program.

module test\_cpu;

reg [8:0] Instruction;

reg CLK;

wire [3:0] WriteData;

cpu cpu1 (Instruction, WriteData, CLK);

initial

begin

// LI $2, 15 # load decimal 15 in $2; $2=1111, which is -1 in 2's comp

#0 Instruction = 9'b100111110;

#0 CLK=1;

#1 CLK=0; // negative edge - execute instruction

/\* Machine code for the test program instructions are input here.

Use the format shown above. Pay attention to the register

order - the destination register is first in the assembly code

and last (LSB) in the machine code.

After each instruction a negative edge must be generated.

\*/

//LI $3, 8 # load decimal 8 (unsigned binary) into $3; $3= 1000, which is -8 in 2's comp

#1 Instruction = 9'b100100011;

#1 CLK=1;

#1 CLK=0; // negative edge - execute instruction

//AND $1, $2, $3 # $1 = $2 AND $3 ($1= 1000)

#1 Instruction = 9'b000101101;

#1 CLK=1;

#1 CLK=0; // negative edge - execute instruction

//SUB $3, $1, $2 # $3 = $1 - $2 = -8 - (-1) = -7 ($3=1001, which is -7 in 2's complement)

#1 Instruction = 9'b110011011;

#1 CLK=1;

#1 CLK=0; // negative edge - execute instruction

//ADD $1, $2, $3 # $1 = $2 + $3 = -1 + (-7) = -8 ($1=1000, which is -8 in 2's complement)

#1 Instruction = 9'b010101101;

#1 CLK=1;

#1 CLK=0; // negative edge - execute instruction

//SUB $2, $3, $1 # $2 = $3 - $1 = -7 - (-8) = 1 ($3= 0001, which is +1 in 2's comp)

#1 Instruction = 9'b110110110;

#1 CLK=1;

#1 CLK=0; // negative edge - execute instruction

//OR $3, $1, $2 # $3 = $1 OR $2 = 1001 ($3= 1000 | 0001= 1001)

#1 Instruction = 9'b001011011;

#1 CLK=1;

#1 CLK=0; // negative edge - execute instruction

end

endmodule

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

//Additional Testbenches:

//LI Testbench:

module test\_li();

reg opcode8, opcode7, opcode6;

wire dectomux;

liDecoder li (dectomux, opcode8, opcode7, opcode6);

initial

begin

opcode8 = 1'b0;

opcode7 = 1'b0;

opcode6 = 1'b0;

end

always #4 opcode8=~opcode8;

always #2 opcode7=~opcode7;

always #1 opcode6=~opcode6;

endmodule

//Mux Testbench:

module test\_mux();

reg [3:0] I0, I1;

reg S;

wire [3:0] Y;

quadTwoXOneMux mux (Y, I0, I1, S);

initial

begin

I0 = 4'b1010;

I1 = 4'b0101;

S = 1'b0;

end

always #1 S=~S;

endmodule

//4-Bit ALU Testbench:

module test\_quad\_ALU();

reg [3:0] A, B;

reg [2:0] Op;

wire [3:0] Result;

wire overflow;

fourBitALU ALU (Result[3:0], overflow, A[3:0], B[3:0], Op[2:0]);

initial

begin

A = 4'b0000;

B = 4'b0000;

Op = 3'b000;

end

always

begin

#1 Op = 3'b001;

#1 Op = 3'b010;

#1 Op = 3'b110;

#1 Op = 3'b000;

end

always #4 A[0] = ~A[0];

always #8 A[1] = ~A[1];

always #16 A[2] = ~A[2];

always #32 A[3] = ~A[3];

always #64 B[0] = ~B[0];

always #128 B[1] = ~B[1];

always #256 B[2] = ~B[2];

always #512 B[3] = ~B[3];

endmodule

//1-Bit ALU Testbench:

module test\_ALU();

reg a, b, Cin, b\_invert;

reg [1:0] operation;

wire Result, Co;

oneBitALU ALU (Result, Co, a, b, Cin, b\_invert, operation[1:0]);

initial

begin

a = 1'b0;

b = 1'b0;

Cin = 1'b0;

b\_invert = ~b;

operation[1] = 1'b0;

operation[0] = 1'b0;

end

always

begin

#1 operation[1] = 0;

#0 operation[0] = 1;

#1 operation[1] = 1;

#0 operation[0] = 0;

#1 operation[1] = 0;

#0 operation[0] = 0;

end

always #3 Cin = ~Cin;

always #6 b = ~b;

always #6 b\_invert = ~b\_invert;

always #12 a = ~a;

endmodule

//Half Adder Testbench:

module test\_half\_adder();

reg x, y;

wire S, C;

halfAdder HA (S, C, x, y);

initial

begin

x = 1'b0;

y = 1'b0;

end

always #2 x = ~x;

always #1 y = ~y;

endmodule

//Full Adder Testbench:

module test\_full\_adder();

reg x, y, z;

wire S, C;

fullAdder FA (x, y, z, C, S);

initial

begin

x = 1'b0;

y = 1'b0;

z = 1'b0;

end

always #4 x = ~x;

always #2 y = ~y;

always #1 z = ~z;

endmodule

//ALU Mux Testbench:

module test\_ALU\_mux();

reg I0, I1, I2, I3, S0, S1;

wire Y;

fourXOneMux MUX (Y, I0, I1, I2, I3, S0, S1);

initial

begin

I0 = 1'b1;

I1 = 1'b0;

I2 = 1'b1;

I3 = 1'b0;

S1 = 1'b0;

S0 = 1'b0;

end

always #2 S1 = ~S1;

always #1 S0 = ~S0;

endmodule