// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Module : controller

// Description : Controller for the MIPS processor

// Revision history :

// Date Programmer Desciption

// ---- ---------- -----------

// 11/08/98 Ngan Pham Module creation

//

// Note :

//

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

module controller ( Clock, // Master Clock

Resetn, // active low reset

Overflow, // overflow

Op, // Opcode of instruction

PCWriteCond,// used for checking BRANCH instruction

PCWrite, // Enable a write to PC

IorD, // Select addr source for Memory

MemRead, // Enable a read from Memory

MemWrite, // Enable a write to Memory

MemtoReg, // Select data source for RegFile

IRWrite, // Enable a write to Instruction Register

CauseWrite, // Enable a write to Cause register

IntCause, // Select the cause of interupt/exception

EPCWrite, // Enable a write to EPC register

PCSource, // Select source data for NextPC

ALUOp, // ALU operation mode

ALUSrcB, // select source for ALU input B

ALUSrcA, // select source for ALU input A

RegWrite, // Enable a write to register of RegFile

RegDst, // Select register to be written in RegFile

LoadByte // Logic for load byte enable CHANGE

);

// ~~~~~~~~~~~~~~~~~~~ Port declaration ~~~~~~~~~~~~~~~~~~~~~~~~~~~~

input Clock, Resetn, Overflow;

input [5:0] Op;

output PCWriteCond, PCWrite, IorD, MemRead, MemWrite, MemtoReg, IRWrite;

output CauseWrite, IntCause, EPCWrite, ALUSrcA, RegWrite, RegDst, LoadByte; // CHANGE

output [1:0] PCSource, ALUOp, ALUSrcB;

// ~~~~~~~~~~~~~~~~~~~ Nets and Regs ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

reg PCWriteCond, PCWrite, IorD, MemRead, MemWrite, MemtoReg, IRWrite;

reg CauseWrite, IntCause, EPCWrite, ALUSrcA, RegWrite, RegDst, LoadByte; // CHANGE

reg [1:0] PCSource, ALUOp, ALUSrcB;

reg [5:0] Current\_state, Next\_state;

// ~~~~~~~~~~~~~~~~~~~~ PARAMETER DEFINITIONS ~~~~~~~~~~~~~~~~~~~~~~~

parameter [4:0] // FSM state encoding

state0 = 5'h0,

state1 = 5'h1,

state2 = 5'h2,

state3 = 5'h3,

state4 = 5'h4,

state5 = 5'h5,

state6 = 5'h6,

state7 = 5'h7,

state8 = 5'h8,

state9 = 5'h9,

state10 = 5'ha,

state11 = 5'hb,

state12 = 5'hc,

state13 = 5'hd,

state14 = 5'he,

state15 = 5'hf,

reset = 5'h10;

parameter [5:0] // Instruction Op field encoding

Rtype = 6'b000000,

LW = 6'b100011,

SW = 6'b101011,

BEQ = 6'b000100,

J = 6'b000010,

LB = 6'b100000; // CHANGE

// ~~~~~~~~~~~~~~~~~~~~~~ NEXT STATE LOGIC ~~~~~~~~~~~~~~~~~~~~~~~~~

// Combinational logic to determine next state from current state.

// See figure 5.50 - "Computer Organization & Design" second edition

// for details.

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

always @(Current\_state or Op or Overflow)

begin

case (Current\_state)

reset : Next\_state = state0;

state0 : Next\_state = state1;

state1 : begin

case (Op)

LW : Next\_state = state2;

LB : Next\_state = state2; // CHANGE

SW : Next\_state = state2;

Rtype : Next\_state = state6;

BEQ : Next\_state = state8;

J : Next\_state = state9;

default : Next\_state = state10;

endcase

end

state2 : begin

case (Op)

LW : Next\_state = state3;

LB : Next\_state = state3; // CHANGE

SW : Next\_state = state5;

default : Next\_state = state0;

endcase

end

state3 : begin

if (Op == LB) // CHANGE

Next\_state = state12;

else

Next\_state = state4;

end

state4 : Next\_state = state0;

state5 : Next\_state = state0;

state6 : Next\_state = state7;

state7 : if (Overflow) Next\_state = state11;

else Next\_state = state0;

state8 : Next\_state = state0;

state9 : Next\_state = state0;

state10: Next\_state = state0;

state11: Next\_state = state0;

state12: Next\_state = state0; // CHANGE

default : Next\_state = state0;

endcase

end

// ~~~~~~~~~~~~~~~~~~~~~~~ LOGIC FOR OUTPUTS ~~~~~~~~~~~~~~~~~~~~~~~~~~

// This is combinational logic for outputs of each state in FSM

// See figure 5.50 for details

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

always @(Current\_state)

begin

PCWriteCond = 1'b0;

PCWrite = 1'b0;

IorD = 1'b0;

MemRead = 1'b0;

MemWrite = 1'b0;

MemtoReg = 1'b0;

IRWrite = 1'b0;

CauseWrite = 1'b0;

IntCause = 1'b0;

EPCWrite = 1'b0;

PCSource = 2'b00;

ALUOp = 2'b00;

ALUSrcB = 2'b00;

ALUSrcA = 1'b0;

RegWrite = 1'b0;

RegDst = 1'b0;

LoadByte = 1'b0; // CHANGE

case (Current\_state)

reset : begin // PC = reset vector; IR = Memory[PC]

end

state0 : // IR = Mem[PC]

// PC = PC + 4

begin

IorD = 1'b0; // reset I or D type command flag

MemRead = 1'b1; // read memory

ALUSrcA = 1'b0; // set ALU sources to increment PC + 4

ALUSrcB = 2'b01;

ALUOp = 2'b00; // add

PCSource = 2'b00; // source 0 for ALU output

PCWrite = 1'b1; // enable PC write

IRWrite = 1'b1; // enable IR write

end

state1 : // ALUOut = PC + (sign\_extend(IR[15:0] << 2)

begin

ALUSrcA = 1'b0;

ALUSrcB = 2'b11;

ALUOp = 2'b00;

end

state2 : // ALUOut = A + sign\_extend(IR[15:0])

begin

ALUSrcA = 1'b1;

ALUSrcB = 2'b10; // book got error here

ALUOp = 2'b00;

end

state3 : // MDR = Memory[ALURegOut]

begin

IorD = 1'b1;

MemRead = 1'b1;

end

state4 : //Reg[ IR[20:16] ] = MDR

begin

MemtoReg = 1'b1; // write to reg enable

RegDst = 1'b0; // send through target register to RF

RegWrite = 1'b1; // enable register write

end

state5 : // Memory[ALURegOut] = B

begin

IorD = 1'b1;

MemWrite = 1'b1;

end

state6 : // ALUOut = A op B

begin

ALUSrcA = 1'b1;

ALUSrcB = 2'b00;

ALUOp = 2'b10;

end

state7 : // Reg[ IR[15:11] ] = ALURegOut

begin

RegDst = 1'b1;

MemtoReg = 1'b0;

RegWrite = 1'b1;

end

state8 : // if (A==B) then PC = ALUOut

begin

ALUSrcA = 1'b1;

ALUSrcB = 2'b00;

ALUOp = 2'b01;

PCWriteCond = 1'b1;

PCSource = 2'b01;

end

state9 : //PC = { PC[31:28], (IR[25:0]<<2) }

begin

PCSource = 2'b10;

PCWrite = 1'b1;

end

state10: // Illegal instruction

// Write to Cause Reg

// EPC = PC - 4

// PC = 0xc000\_0000;

begin

IntCause = 1'b0;

CauseWrite = 1'b1;

ALUSrcA = 1'b0;

ALUSrcB = 2'b01;

ALUOp = 2'b01;

EPCWrite = 1'b1;

PCSource = 2'b11;

PCWrite = 1'b1;

end

state11: // Overflow exception

// Write to cause reg

// EPC = PC - 4

// PC = 0xc000\_0000

begin

IntCause = 1'b1;

CauseWrite = 1'b1;

ALUSrcA = 1'b0;

ALUSrcB = 2'b01;

ALUOp = 2'b01;

EPCWrite = 1'b1;

PCWrite = 1'b1;

PCSource = 2'b11;

end

state12: begin // CHANGE

MemtoReg = 1'b1; // write to reg enable

RegDst = 1'b0; // send through target register to RF

RegWrite = 1'b1; // enable register write

LoadByte = 1'b1;

end

endcase

end

// ~~~~~~~~~~~~~~~~~~~~~~~ CURRENT\_STATE LOGIC ~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Sequencial logic for Current\_state from Next\_state at positive clock

// edge of each clock cycle. Enable synchronous reset to cause FSM to

// execute exception vector instruction

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

always @(posedge Clock)

begin

if (!Resetn) Current\_state <= reset;

else Current\_state <= Next\_state;

end

endmodule