**ALU OPERATION**

|  |
| --- |
|  |
| **Module ALU\_Core(ALUSrc1 , ALUSrc2 , ALUCtrl , ALUResult , Zero);** |
|  | **input[31:0] ALUSrc1;** |
|  | **input[31:0] ALUSrc2;** |
|  | **input[2:0] ALUCtrl;** |
|  |  |
|  | **output Zero;** |
|  | **reg Zero;** |
|  |  |
|  | **output [31:0]ALUResult;** |
|  | **reg [31:0]ALUResult;** |
|  |  |
|  |  |
|  | **always @(ALUSrc1 or ALUSrc2 or ALUCtrl)** |
|  | **begin** |
|  |  |
|  | **if(ALUCtrl == 3'b010) //'add'** |
|  | **begin** |
|  | **ALUResult = ALUSrc1 + ALUSrc2;** |
|  | **if(ALUResult == 32'h0000)** |
|  | **begin** |
|  | **Zero = 1'b1;** |
|  | **end** |
|  | **else** |
|  | **begin** |
|  | **Zero = 1'b0;** |
|  | **end** |
|  | **end** |
|  |  |
|  | **if(ALUCtrl == 3'b110) // 'sub'** |
|  | **begin** |
|  | **ALUResult = ALUSrc1 - ALUSrc2;** |
|  | **if(ALUResult == 32'h0000)** |
|  | **begin** |
|  | **Zero = 1'b1;** |
|  | **end** |
|  | **else** |
|  | **begin** |
|  | **Zero = 1'b0;** |
|  | **end** |
|  | **end** |
|  |  |
|  | **if(ALUCtrl == 3'b000) // 'and'** |
|  | **begin** |
|  | **ALUResult = ALUSrc1 & ALUSrc2;** |
|  | **if(ALUResult == 32'h0000)** |
|  | **begin** |
|  | **Zero = 1'b1;** |
|  | **end** |
|  | **else** |
|  | **begin** |
|  | **Zero = 1'b0;** |
|  | **end** |
|  | **end** |
|  |  |
|  | **if(ALUCtrl == 3'b001) // 'or'** |
|  | **begin** |
|  | **ALUResult = ALUSrc1 | ALUSrc2;** |
|  | **if(ALUResult == 32'h0000)** |
|  | **begin** |
|  | **Zero = 1'b1;** |
|  | **end** |
|  | **else** |
|  | **begin** |
|  | **Zero = 1'b0;** |
|  | **end** |
|  | **end** |
|  |  |
|  | **if(ALUCtrl == 3'b111) // 'slt'** |
|  | **begin** |
|  | **ALUResult = ALUSrc1 - ALUSrc2;** |
|  | **if(ALUResult == 32'h0000)** |
|  | **begin** |
|  | **Zero = 1'b1;** |
|  | **end** |
|  | **else** |
|  | **begin** |
|  | **Zero = 1'b0;** |
|  | **end** |
|  | **end** |
|  |  |
|  | **end** |
|  |  |
|  | **endmodule** |
|  |  |
|  |  |
|  |  |
|  | **module ALU\_Control(FunctField, ALUOp, ALUCtrl);** |
|  | **input [5:0]FunctField;** |
|  | **input [1:0]ALUOp;** |
|  | **output [2:0]ALUCtrl;** |
|  | **reg [2:0]ALUCtrl;** |
|  |  |
|  | **always@(FunctField or ALUOp)** |
|  | **begin** |
|  | **if(ALUOp == 2'b10) //'Arithmetic' Type Instructions** |
|  | **begin** |
|  | **case(FunctField)** |
|  | **//begin** |
|  | **6'b100000: ALUCtrl = 3'b010; //ADDITION in 'R' Type** |
|  | **6'b100010: ALUCtrl = 3'b110; //SUBTRACTION in 'R' Type** |
|  | **6'b100100: ALUCtrl = 3'b000; //AND in 'R' Type** |
|  | **6'b100101: ALUCtrl = 3'b001; //OR in 'R' Type** |
|  | **6'b101010: ALUCtrl = 3'b111; //SLT in 'R' Type** |
|  | **// end** |
|  | **endcase** |
|  | **end** |
|  |  |
|  | **if(ALUOp == 2'b00) // 'LW/SW' Type Instructions** |
|  | **begin** |
|  | **ALUCtrl = 3'b010; //ADDITION irrespective of the FunctField.** |
|  | **end** |
|  |  |
|  | **if(ALUOp == 2'b01) // 'BEQ', 'BNE' Type Instructions** |
|  | **begin** |
|  | **ALUCtrl = 3'b110; //SUBTRACTION irrespective of the FunctField.** |
|  | **end** |
|  |  |
|  |  |
|  |  |
|  | **end //always block** |
|  |  |
|  | **endmodule //ALUOp module** |
|  |  |

**ADDER 32 BIT**

|  |
| --- |
| **module Adder32Bit(input1, input2, out, overflowBit);** |
|  |  |
|  | **input [31:0] input1, input2;** |
|  | **output [31:0] out;** |
|  | **reg [31:0]out;** |
|  | **output overflowBit;** |
|  | **reg overflowBit;** |
|  |  |
|  | **always@(input1 or input2)** |
|  | **begin** |
|  |  |
|  | **{overflowBit , out } = input1 + input2;** |
|  |  |
|  | **end** |
|  |  |
|  | **endmodule** |

**INSTRUCTION MEMORY**

|  |
| --- |
|  |
| **module InstructionMemory(readAddress, instruction);** | |
|  | | **input [31:0]readAddress;** |
|  | | **output [0:31]instruction;** |
|  | | **reg [0:31]instruction;** |
|  | |  |
|  | | **reg [0:7]InstructionMemory[0:31];** |
|  | |  |
|  | | **reg [4:0]internalAddress;** |
|  | |  |
|  | | **integer internalAddressINT, placeVal, i, j;** |
|  | |  |
|  | | **always@(readAddress)** |
|  | | **Begin** |
|  | | **//use this template to hardwire instructions.** |
|  | | **//Only 5 bit addresses supported.** |
|  | | **// InstructionMemory[0] = 32'b00000\_00000\_00000\_00000\_00000\_000000;** |
|  | |  |
|  | | **{InstructionMemory[0], InstructionMemory[1], InstructionMemory[2], InstructionMemory[3]} = 32'b001101\_10010\_10011\_0000000000000001; //ori $s2, $s1 , 1539;** |
|  | | **{InstructionMemory[4], InstructionMemory[5], InstructionMemory[6], InstructionMemory[7]} = 32'b000101\_10011\_00000\_0000000000000100; // bne $s2, reg1, 4h;** |
|  | |  |
|  | |  |
|  | | **{InstructionMemory[24], InstructionMemory[24+1], InstructionMemory[24+2], InstructionMemory[24+3]} = 32'b001000\_10011\_10010\_0000000000000100; //addi $s1, $s2 ,4;** |
|  | | **{InstructionMemory[28], InstructionMemory[28+1], InstructionMemory[28+2], InstructionMemory[28+3]} = 32'b000010\_00000\_00000\_0000000000000000; // j 0;** |
|  | | **// InstructionMemory[1] = 32'b001001\_10010\_10011\_00000\_00000\_000010; //addi $s2, $s3, 2;** |
|  | | **// InstructionMemory[3] = 32'b000000\_00010\_00001\_00000\_00000\_100000;** |
|  | |  |
|  | | **//truncating the address.** |
|  | | **internalAddress = readAddress[4:0];** |
|  | |  |
|  | | **//internalAddressINT = 0;** |
|  | | **//** |
|  | | **placeVal = 1;** |
|  | | **internalAddressINT = 0;** |
|  | | **for(i=0 ; i<5 ; i=i+1)** |
|  | | **Begin** |
|  | | **if(internalAddress[i] == 1)** |
|  | | **internalAddressINT = internalAddressINT + placeVal;** |
|  | |  |
|  | | **placeVal = placeVal \* 2;** |
|  | | **End** |
|  | |  |
|  | |  |
|  | | **for(i=0 ; i<32 ; i=i+1)** |
|  | | **Begin** |
|  | | **instruction[i] = InstructionMemory[internalAddressINT + i/8][i%8];** |
|  | | **End** |
|  | |  |
|  | |  |
|  | | **end** |
|  | |  |
|  | | **endmodule** |
|  | |  |

**LEFT SHIFTER**

|  |
| --- |
| **module LeftShifter\_2bit(inData,outData);** |
|  |  |
|  | **input [31:0]inData;** |
|  | **output [31:0]outData;** |
|  | **reg [31:0]outData;** |
|  |  |
|  | **always@(inData)** |
|  | **begin** |
|  |  |
|  | **outData=inData<<2;** |
|  |  |
|  | **end** |
|  |  |
|  | **endmodule** |

**MUX 2 TO 4**

|  |
| --- |
| **module MUX\_2to1( input1 , input2, select, out );** |
|  | **input [31:0] input1, input2;** |
|  | **input select;** |
|  | **output [31:0]out;** |
|  | **reg [31:0]out;** |
|  |  |
|  | **always @(input1 or input2 or select )** |
|  | **begin** |
|  | **case(select)** |
|  |  |
|  | **1'b0: out=input1;** |
|  | **1'b1: out=input2;** |
|  |  |
|  | **endcase** |
|  | **end** |
|  | **endmodule** |

**REGISTER FILE**

|  |
| --- |
| **module RegisterFile(readReg1, readReg2, writeReg, writeData, readData1, readData2, RegWrite);** |
|  | **input [4:0]readReg1, readReg2, writeReg;** |
|  | **input [31:0]writeData; //address of the register to be written on to.** |
|  | **input RegWrite; //RegWrite - register write signal; writeReg-the destination register.** |
|  |  |
|  | **output [31:0]readData1, readData2;** |
|  | **reg [31:0]readData1, readData2;** |
|  |  |
|  | **reg [31:0]RegMemory[0:31];** |
|  |  |
|  | **integer placeVal, i, j, writeRegINT=0, readReg1INT=0, readReg2INT=0;** |
|  |  |
|  | **initial** |
|  | **begin** |
|  | **for(i=0 ; i<32 ; i=i+1)** |
|  | **begin** |
|  | **for(j=0 ; j<32 ; j= j+1)** |
|  | **RegMemory[i][j] = 1'b0;** |
|  | **end** |
|  | **end** |
|  |  |
|  | **always@ (RegWrite or readReg1 or readReg2 or writeReg or writeData)** |
|  | **begin** |
|  |  |
|  | **if(RegWrite == 1)** |
|  | **begin** |
|  |  |
|  | **placeVal = 1;** |
|  | **readReg1INT=0;** |
|  | **readReg2INT=0;** |
|  | **for(i=0 ; i<5 ; i=i+1)** |
|  | **begin** |
|  | **if(readReg1[i] == 1)** |
|  | **readReg1INT = readReg1INT + placeVal;** |
|  |  |
|  | **if(readReg2[i] == 1)** |
|  | **readReg2INT = readReg2INT + placeVal;** |
|  |  |
|  | **placeVal = placeVal \* 2;** |
|  | **end** |
|  |  |
|  |  |
|  | **for(i=0 ; i<32 ; i=i+1)** |
|  | **begin** |
|  | **readData1[i] = RegMemory[readReg1INT][i];** |
|  | **readData2[i] = RegMemory[readReg2INT][i];** |
|  | **end** |
|  |  |
|  |  |
|  |  |
|  | **//binary to decimal address translation.** |
|  | **placeVal = 1;** |
|  | **writeRegINT=0;** |
|  | **for(i=0 ; i<5 ; i=i+1)** |
|  | **begin** |
|  | **if(writeReg[i] == 1)** |
|  | **writeRegINT = writeRegINT + placeVal;** |
|  |  |
|  | **placeVal = placeVal \* 2;** |
|  | **end** |
|  |  |
|  | **$display("before writing %d at %d", writeData, writeRegINT);** |
|  | **for(i=0 ; i<32 ; i=i+1)** |
|  | **begin** |
|  | **RegMemory[writeRegINT][i] = writeData[i];** |
|  | **end** |
|  | **$display("after writing %d at %d", writeData, writeRegINT);** |
|  |  |
|  | **end // Register Write** |
|  |  |
|  | **if(RegWrite == 0)** |
|  | **begin** |
|  | **//binary to decimal address translation.** |
|  | **placeVal = 1;** |
|  | **readReg1INT=0;** |
|  | **readReg2INT=0;** |
|  | **for(i=0 ; i<5 ; i=i+1)** |
|  | **begin** |
|  | **if(readReg1[i] == 1)** |
|  | **readReg1INT = readReg1INT + placeVal;** |
|  |  |
|  | **if(readReg2[i] == 1)** |
|  | **readReg2INT = readReg2INT + placeVal;** |
|  |  |
|  | **placeVal = placeVal \* 2;** |
|  | **end** |
|  |  |
|  |  |
|  | **for(i=0 ; i<32 ; i=i+1)** |
|  | **begin** |
|  | **readData1[i] = RegMemory[readReg1INT][i];** |
|  | **readData2[i] = RegMemory[readReg2INT][i];** |
|  | **end** |
|  |  |
|  |  |
|  | **end// Register Read** |
|  |  |
|  | **end //always@** |
|  |  |
|  | **endmodule** |

**SIGN EXTENDER**

|  |  |  |
| --- | --- | --- |
| **module SignExtender\_16to32(inputData, outputData);** | | |
|  | | |  | | |
|  | | | **input[15:0] inputData;** | | |
|  | | | **output[31:0] outputData;** | | |
|  | | | **reg [31:0] outputData;** | |
|  | | |  |
|  | | | **always@(inputData)** |
|  | | | **Begin** |
|  | | |  |
|  | | | **outputData[15:0] = inputData[15:0];** |
|  | | | **outputData[31:16] = {16{inputData[15]}};** |
|  | | |  |
|  | | | **end** |
|  | | | **endmodule** |

**SINGLE CYCLE MAIN PROGRA**

|  |
| --- |
| **module SingleCycleMain(initialPCval, run);** |
|  | **input [31:0]initialPCval;** |
|  | **input run;** |
|  |  |
|  | **reg [31:0]PC;** |
|  |  |
|  |  |
|  | **//instances of all the smaller modules of the processor.** |
|  |  |
|  | **reg [31:0]instrReg;** |
|  |  |
|  | **reg [31:0]instrAddress;** |
|  | **wire [31:0]instrWire;** |
|  | **InstructionMemory instrMem(instrAddress, instrWire);** |
|  |  |
|  | **reg [31:0]inputToShiftLeft;** |
|  | **wire [31:0]outputFromShiftLeft;** |
|  | **LeftShifter\_2bit instLftShft(inputToShiftLeft, outputFromShiftLeft);** |
|  |  |
|  | **//reg [31:0]PC; //already declared** |
|  | **reg [31:0]constantFour;** |
|  | **wire [31:0]nextPCval;** |
|  | **wire overflow1;** |
|  | **Adder32Bit nextPCvalue(PC, constantFour, nextPCval, overflow1);** |
|  |  |
|  | **// wire [31:0]nextPCval;** |
|  | **// wire [31:0]outputFromShiftLeft;** |
|  | **wire [31:0]nextPCvalPlusOffset;** |
|  | **wire overflow2;** |
|  | **Adder32Bit PCafterBranch(nextPCval, outputFromShiftLeft, nextPCvalPlusOffset, overflow2);** |
|  |  |
|  | **reg [31:0]dataAddress;** |
|  | **reg [31:0]inputData;** |
|  | **wire[31:0]outputData;** |
|  | **reg MemRead, MemWrite;** |
|  | **DataMemory dataMem(dataAddress, inputData, outputData, MemRead, MemWrite);** |
|  |  |
|  |  |
|  | **reg [4:0]inputReg1;** |
|  | **reg [4:0]inputReg2;** |
|  | **reg RegDst;** |
|  | **wire [4:0]writeRegWire;** |
|  | **MUX\_2to1\_5bit regDstMUX(inputReg1, inputReg2, RegDst, writeRegWire);** |
|  |  |
|  |  |
|  | **reg [4:0]readReg1, readReg2, writeReg;** |
|  | **reg [31:0]writeData;** |
|  | **wire [31:0]readData1, readData2;** |
|  | **reg RegWrite;** |
|  | **RegisterFile regFile(readReg1, readReg2, writeReg, writeData, readData1, readData2, RegWrite);** |
|  |  |
|  |  |
|  | **reg [15:0]inputDataSEXT;** |
|  | **wire [31:0]outputDataSEXT;** |
|  | **SignExtender\_16to32 signExt(inputDataSEXT, outputDataSEXT);** |
|  |  |
|  |  |
|  | **//wire [31:0]readData2; //already declared.** |
|  | **//wire [31:0]outputDataSEXT; //already declared.** |
|  | **reg aluSrc;** |
|  | **wire [31:0] ALUSrc2;** |
|  | **MUX\_2to1 aluSrc2MUX(readData2, outputDataSEXT, aluSrc, ALUSrc2);** |
|  |  |
|  |  |
|  | **reg [5:0]FunctField;** |
|  | **reg [1:0]ALUOp;** |
|  | **wire [2:0]ALUCtrl1;** |
|  | **ALU\_Control aluCtrlInstance(FunctField, ALUOp, ALUCtrl1);** |
|  |  |
|  |  |
|  | **reg [31:0]ALUSrc1;** |
|  | **//wire [31:0] ALUSrc2;** |
|  | **reg [2:0]ALUCtrl;** |
|  | **wire[31:0]ALUout;** |
|  | **wire ZeroOUT;** |
|  | **ALU\_Core aluCoreInstance(ALUSrc1, ALUSrc2, ALUCtrl, ALUout, ZeroOUT);** |
|  |  |
|  | **reg Branch;** |
|  | **reg ZeroIN;** |
|  | **wire BranchEnabled;** |
|  | **and branchAND(BranchEnabled, Branch, ZeroIN);** |
|  |  |
|  |  |
|  | **// wire [31:0]nextPCval;** |
|  | **// wire [31:0]nextPCvalPlusOffset;** |
|  | **// wire BranchEnabled;** |
|  | **wire [31:0]nextPCactual;** |
|  | **MUX\_2to1 pcSrcMUX(nextPCval, nextPCvalPlusOffset, BranchEnabled, nextPCactual);** |
|  |  |
|  |  |
|  | **//wire[31:0]ALUout; //already declared** |
|  | **//wire[31:0]outputData; //already declared** |
|  | **reg MemtoReg;** |
|  | **wire [31:0]writeDataToReg;** |
|  | **MUX\_2to1 mem2regSrcMUX(ALUout, outputData, MemtoReg, writeDataToReg);** |
|  |  |
|  |  |
|  | **//Requisite Datastructures for manipulation.** |
|  | **reg [5:0]OpCode;** |
|  | **reg [4:0]rs, rt, rd, shamt;** |
|  | **reg [25:0]target;** |
|  | **reg [31:0]jumpTarget;** |
|  | **integer counter, prevInstrWasJ;** |
|  |  |
|  | **initial** |
|  | **begin** |
|  | **PC = initialPCval;** |
|  | **//instrReg = 32'b001001\_10010\_10011\_00000\_00000\_000100; //addi $s1, $s2 ,4;** |
|  | **constantFour = 32'h0000\_0004; //updated it was 'constantFout'** |
|  | **counter = 0;** |
|  | **end** |
|  |  |
|  | **always@(run) //this will make it work like trace. to make the execution sequential & automatic use always(PC)** |
|  | **begin** |
|  | **if(counter != 0) //that is if we are not executing for the first time, we will not take the previously calculated pc value.** |
|  | **PC = nextPCactual;** |
|  |  |
|  | **if(prevInstrWasJ == 1) //that is if the previous instruction was jump, take next pc from a specified collection register.** |
|  | **PC = jumpTarget;** |
|  |  |
|  | **instrAddress = PC;** |
|  | **#10** |
|  | **instrReg = instrWire; //updated, this link was not there.** |
|  | **//Now the wire instrReg has the Instruction corresponding to this PC.** |
|  | **OpCode = instrReg[31:26];** |
|  |  |
|  | **writeReg = 5'b00000;** |
|  | **writeData = 32'h1111\_1111;** |
|  |  |
|  | **RegDst = 0;** |
|  | **$display("value after init, %d, %d", writeReg, writeRegWire);** |
|  |  |
|  | **counter = 1; //to keep track of the fact that we are not executing our first instruction.** |
|  |  |
|  | **/\* Beginning of ##CRITICAL INTERCONNECTIONS##. DO NOT MODIFY\*/** |
|  | **rs = instrReg[25:21];** |
|  | **rt = instrReg[20:16];** |
|  | **rd = instrReg[15:11];** |
|  | **inputDataSEXT = instrReg[15:0];** |
|  | **shamt = instrReg[10:6];** |
|  | **FunctField = instrReg[5:0];** |
|  |  |
|  |  |
|  | **//the inputs to the register file.** |
|  | **readReg1 = rs;** |
|  | **readReg2 = rt;** |
|  |  |
|  | **//the two inputs to the RegDst mux.** |
|  | **inputReg1 = rt;** |
|  | **inputReg2 = rd;** |
|  | **#10** |
|  | **//$display("value of writeRegWire after input1 input2, %d", writeRegWire);** |
|  |  |
|  | **//now the output of the register file should be correctly assigned to aluSrc1 and aluSrcMux1.** |
|  |  |
|  |  |
|  | **//#50** |
|  | **//$display("before value of ALUSrc1, readData1 %d %d", ALUSrc1, readData1);** |
|  | **//ALUSrc1 = readData1;** |
|  | **//$display("after value of ALUSrc1, readData1 %d %d", ALUSrc1, readData1);** |
|  | **//ALUSrc2\_input1 = readData2; this has been already hardwired.** |
|  | **//ALUSrc2\_input2 = outputDataSEXT; this has been already hardwired.** |
|  |  |
|  | **//output from ALUControl being passed into ALUCore.** |
|  | **ALUCtrl = ALUCtrl1;** |
|  |  |
|  | **//now the output of the mux needs to be redirected to the writeReg of the register file** |
|  | **// #10** |
|  | **//writeReg = writeRegWire;** |
|  | **$display("value after first mod, %5d, %5s", writeReg, writeRegWire);** |
|  | **//connecting ALUout with memory input and memory mux.** |
|  | **//ALUout already connected to mem2regMUX.** |
|  | **dataAddress = ALUout;** |
|  | **ZeroIN = ZeroOUT; //ALU connections complete** |
|  |  |
|  |  |
|  | **//connecting the readData2 register output to the memory data input.** |
|  | **inputData = readData2;** |
|  | **//the outputData is already connected to the mem2regMUX.** |
|  |  |
|  | **//output from Sign extender being given to the shiftleft module.** |
|  | **inputToShiftLeft = outputDataSEXT;** |
|  | **#20** |
|  | **//now we need to check if the memory mux output is connected to the writeData of regFile.** |
|  | **//writeData = writeDataToReg;** |
|  |  |
|  |  |
|  |  |
|  | **/\* End of ##CRITICAL INTERCONNECTIONS##. DO NOT MODIFY\*/** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | **if(OpCode == 6'b000010 || OpCode == 6'b000011) //If the instruction is 'J' type.** |
|  | **begin** |
|  | **target = instrReg[25:0];** |
|  | **jumpTarget[27:2] = target;** |
|  | **jumpTarget[1:0] = 2'b00;** |
|  | **jumpTarget[31:28] = PC[31:28];** |
|  | **prevInstrWasJ = 1;** |
|  | **end** |
|  |  |
|  | **else if(OpCode == 6'b000000) // If the instruction is 'R' type.** |
|  | **begin** |
|  |  |
|  | **ALUOp = 2'b10;** |
|  | **RegDst = 1'b1;** |
|  | **Branch = 0;** |
|  | **MemRead = 0;** |
|  | **MemWrite = 0;** |
|  | **aluSrc = 0;** |
|  | **MemtoReg = 0;** |
|  | **RegWrite = 1;** |
|  |  |
|  |  |
|  | **end //'R' Type.** |
|  |  |
|  |  |
|  | **else // If the instruction is 'I' type.** |
|  | **begin** |
|  |  |
|  |  |
|  | **if(OpCode == 6'b100011) // lw instruction.** |
|  | **begin** |
|  | **//Control Signals** |
|  | **ALUOp = 2'b00;** |
|  | **RegDst = 1'b0;** |
|  | **Branch = 0;** |
|  | **MemRead = 1;** |
|  | **MemWrite = 0;** |
|  | **aluSrc = 1;** |
|  | **MemtoReg = 1;** |
|  | **RegWrite = 1;** |
|  |  |
|  | **end** |
|  |  |
|  | **if(OpCode == 6'b101011) //sw instruction** |
|  | **begin** |
|  | **//Control Signals** |
|  | **ALUOp = 2'b00;** |
|  | **RegDst = 1'b0; //irrelevant as data not being written into regfile.** |
|  | **Branch = 0;** |
|  | **MemRead = 0;** |
|  | **MemWrite = 1;** |
|  | **aluSrc = 1;** |
|  | **MemtoReg = 1; //irrelevant** |
|  | **RegWrite = 0;** |
|  |  |
|  | **end** |
|  |  |
|  | **if(OpCode == 6'b000100) //beq instruction** |
|  | **begin** |
|  | **//Control Signals** |
|  | **ALUOp = 2'b01;** |
|  | **RegDst = 1'b0; //irrelevant** |
|  | **Branch = 1;** |
|  | **MemRead = 0;** |
|  | **MemWrite = 0;** |
|  | **aluSrc = 0;** |
|  | **MemtoReg = 1; //irrelevant** |
|  | **RegWrite = 0; //irrelevant** |
|  |  |
|  | **end** |
|  |  |
|  | **if(OpCode == 6'b000101) //bne instruction** |
|  | **begin** |
|  | **//Control Signals** |
|  | **ALUOp = 2'b01; //for branch instruction.** |
|  | **RegWrite = 0;** |
|  | **RegDst = 0;** |
|  | **Branch = 1;** |
|  | **aluSrc = 0;** |
|  | **ZeroIN = ~ZeroOUT; //passing the negated value of ZeroOUT from the ALU to the PCSrc MUX.** |
|  | **MemRead = 0;** |
|  | **MemWrite = 0;** |
|  |  |
|  | **MemtoReg = 1;** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | **end** |
|  |  |
|  | **if(OpCode == 6'b001101) //ori instruction** |
|  | **begin** |
|  |  |
|  | **//Control Signals** |
|  | **prevInstrWasJ = 0;** |
|  | **ALUOp = 2'b01; //presently irrelevant** |
|  | **RegDst = 0;** |
|  |  |
|  | **Branch = 0;** |
|  | **MemRead = 0;** |
|  | **MemWrite = 0;** |
|  | **//now the output of the mux needs to be redirected to the writeReg of the register file** |
|  | **$display("value after first mod before second,%5d, %5d", writeReg, writeRegWire);** |
|  | **writeReg = writeRegWire;** |
|  | **$display("value after second mod,%5d, %5d", writeReg, writeRegWire);** |
|  | **RegWrite=1;** |
|  |  |
|  |  |
|  | **#50** |
|  | **$display("value of readData1, %d", readData1);** |
|  |  |
|  | **#50** |
|  | **$display("value of readData1, %d", readData1);** |
|  | **ALUSrc1 = readData1;** |
|  | **aluSrc = 1;** |
|  |  |
|  | **ALUCtrl = 3'b001; //for or** |
|  | **#20** |
|  |  |
|  | **$display("1 writeData = %d, writeDataToReg = %d", writeData, writeDataToReg);** |
|  |  |
|  | **MemtoReg = 0; //the aluout is to be redirected to the regwrite.** |
|  | **#20** |
|  | **$display("2 writeData = %d, writeDataToReg = %d", writeData, writeDataToReg);** |
|  |  |
|  | **writeData = writeDataToReg;** |
|  | **RegWrite = ~RegWrite; //this will write the initialized value 32'b1111\_1111** |
|  | **RegWrite = ~RegWrite; //this will run the next cycle of the reg file, writing the updated value.** |
|  |  |
|  | **$display("3 writeData = %d, writeDataToReg = %d", writeData, writeDataToReg);** |
|  |  |
|  |  |
|  |  |
|  |  |
|  | **end** |
|  |  |
|  | **if(OpCode == 6'b001000) //addi instruction** |
|  | **begin** |
|  | **//Control Signals** |
|  | **prevInstrWasJ = 0;** |
|  | **ALUOp = 2'b01; //presently irrelevant** |
|  | **RegDst = 0;** |
|  |  |
|  | **Branch = 0;** |
|  | **MemRead = 0;** |
|  | **MemWrite = 0;** |
|  | **//now the output of the mux needs to be redirected to the writeReg of the register file** |
|  | **$display("value after first mod before second,%5d, %5d", writeReg, writeRegWire);** |
|  | **writeReg = writeRegWire;** |
|  | **$display("value after second mod,%5d, %5d", writeReg, writeRegWire);** |
|  | **RegWrite = 1;** |
|  |  |
|  | **#20** |
|  | **$display("value of readData1, %d", readData1);** |
|  | **ALUSrc1 = readData1;** |
|  | **aluSrc = 1;** |
|  |  |
|  | **ALUCtrl = 3'b010; //for add** |
|  | **#20** |
|  |  |
|  | **$display("1 writeData = %d, writeDataToReg = %d", writeData, writeDataToReg);** |
|  |  |
|  | **MemtoReg = 0; //the aluout is to be redirected to the regwrite.** |
|  | **#20** |
|  | **$display("2 writeData = %d, writeDataToReg = %d", writeData, writeDataToReg);** |
|  |  |
|  | **writeData = writeDataToReg;** |
|  | **RegWrite = ~RegWrite;** |
|  | **RegWrite = ~RegWrite;** |
|  |  |
|  | **$display("3 writeData = %d, writeDataToReg = %d", writeData, writeDataToReg);** |
|  |  |
|  | **end** |
|  |  |
|  | **end //'I' Type.** |
|  |  |
|  |  |
|  | **end //always block** |
|  |  |
|  |  |
|  | **endmodule** |