Xi'an Jiaotong-Liverpool University

西交利物浦大学

EEE 339: Digital System Design with HDL

MIPS Single-cycle Processor

Name: Kai-Yu Lu

ID: 1614649

English Name	Memory locations (X,Y,Z,T)	Data bus size	Design 1
Kai-Yu Lu	2, 3, 4, 5	20	ori

Task 1.

MIPS Code:

lw $2, 8(0)
lw $3, 12(0)
add $4, $2, $3
sw $4, 16(0)
lw $5, 16(0)

Machine Code:

+0	100011 00000 00010 0000000000000010
+1	100011 00000 00011 0000000000000011
+2	000000 00010 00011 00100 00000 100000
+3	101011 00000 00100 0000000000000100
+4	100011 00000 00101 0000000000000100

Table 1: 32-bit machine code for Task 1.

Task 2.

Simulation Result

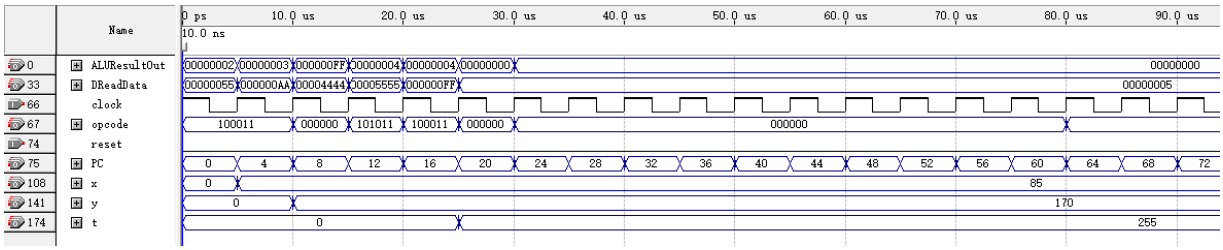

Figure 1: Simulation result for Task 1.

From Figure 2, the default values in memory of X, Y, Z and T are 85, 170 and 255 in decimal correspondingly.

```
always @(posedge clock or posedge reset)begin
        if (reset) begin
            DMem[0]=32'h00000005;
            DMem[1]=32'h0000000A;
            DMem[2]=32'h00000055;
            DMem[3]=32'h000000AA;
            DMem[4]=32'h00005555;
            DMem[5]=32'h00008888;
            DMem[6]=32'h00550000;
            DMem[7]=32'h00004444;
```

Figure 2: The default values in memory of X, Y, Z and T

It could be observed that the value in X register became 85 after the first load and the value in Y register became 170 after the second load. The values summed up by X and Y should be 225 and Figure 1 shows the value in Z register is 255 after adding operation, therefore it is correct. Finally, the value in T register became 225, which satisfies the condition that the value in T register was loaded from Z memory location.

Task 3.

Condition 1: BEQ satisfies.
MIPS Code:

lw $2, 8(0)
lw $3, 8(0)
beq $2, $3, 2

Machine Code:

+0	100011 00000 00010 0000000000000010
+1	100011 00000 00011 0000000000000010
+2	000100 00010 00011 0000000000000010

Table 2: 32-bit machine code for Task 3 when BEQ satisfied.

Simulation Result

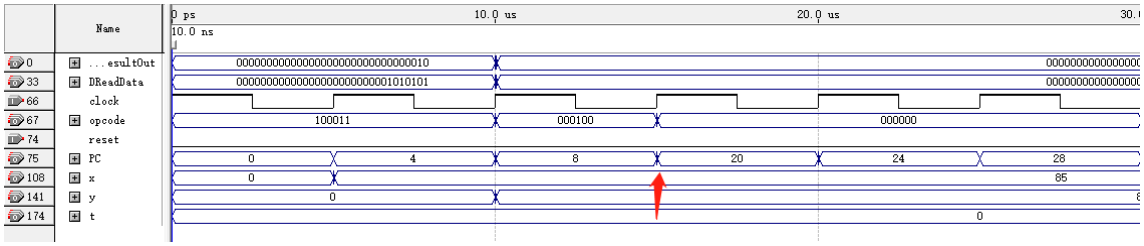

Figure 3: Simulation result for Task 3 when BEQ is satisfied.

This is the condition for BEQ is satisfied. From the machine code provided in Table 2, PC will skip one instruction which means PC value will be added by 4 directly if BEQ is satisfied. Because

$$New\ PC\ Value = Old\ PC\ Value + 4 + Offset * 4$$

In Table 2, it loaded the same value into registers X and Y, then the value in X and Y would be equal. The expected value of PC would be 20 after operating the third machine code, because the PC value of third instruction was 8 and the offset value is 2, which gives

$$Old\ PC\ Value\ (8) + 4 + Offset(2) * 4 = \ New\ PC\ Value\ (20)$$

From Figure 3, the new PC value has become 20, which means the task was completed correctly when BEQ was satisfied.

Condition 2: BEQ not satisfies.
MIPS Code:

lw $2, 8(0)
lw $3, 12(0)
beq $2, $3, 2

Machine Code:

+0	100011 00000 00010 0000000000000010
+1	100011 00000 00011 0000000000000011
+2	000100 00010 00011 0000000000000010

Table 3: 32-bit machine code for Task 3 when BEQ satisfied.

Simulation Result

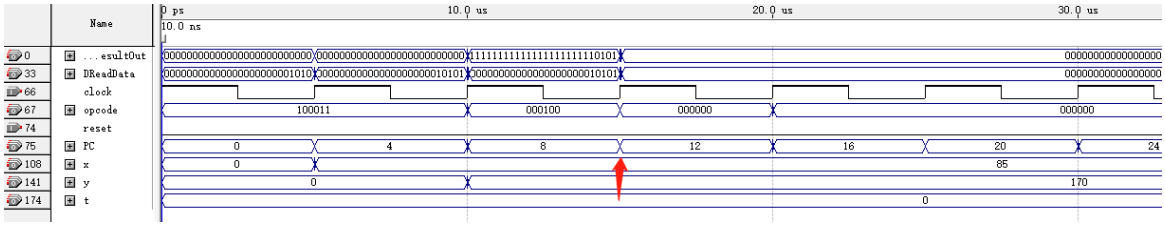

Figure 4: Simulation result for Task 3 when BEQ is not satisfied.

This is the condition for BEQ is not satisfied. From the machine code provided in Table 3, PC will skip one instruction which means PC value will be added by 4 directly if BEQ is satisfied. Because

$$New\ PC\ Value = Old\ PC\ Value + 4 + Offset * 4$$

In Table 3, it loaded the same value into registers X and Y, then the value in X and Y would be different. Therefore, the expected value of PC would be 12 after operating the third machine code, because the PC value of third instruction was 8, which gives

$$New\ PC\ Value = Old\ PC\ Value + 4$$

From Figure 4, the new PC value has become 12, which means the task was completed correctly when BEQ was not satisfied. After examining two condition of BEQ, it could be concluded that this task was finished totally correct.

Task 4.

Modified Verilog Code for processor:

The followings will show the Verilog Code for the processor, the modified codes will be highlighted in yellow. The explanations for every modified code will be appended nearby and highlighted in yellow.

```
// Register File
module RegisterFile (Read1,Read2,Writereg,WriteData,RegWrite, Data1, Data2,clock,reset, x, y, t); // In order to show the value of x, y, t.
    input   [4:0] Read1,Read2,Writereg; // the registers numbers to read or write
    input   [31:0] WriteData;                   // data to write
    input   RegWrite;                           // The write control
    input   clock, reset;                       // The clock to trigger writes
    output [31:0] Data1, Data2, x, y, t;        // In order to show the value of x, y, t.
    reg     [31:0] RF[31:0], x, y, t;           // In order to show the value of x, y, t.
```

```
        integer k;

        // Read from registers independent of clock
        assign Data1 = RF[Read1];
        assign Data2 = RF[Read2];
        // write the register with new value on the falling edge of the clock if RegWrite is high
        always @(posedge clock or posedge reset)
        begin
                if (reset) for(k=0;k<32;k=k+1) RF[k]<=32'h00000000;
                // Register 0 is a read only register with the content of 0
                else    if (RegWrite & (Writereg!=0)) RF[Writereg] <= WriteData;
                begin // In order to give the value to register x, y and t.
                        x <= RF[2];
                        y <= RF[3];
                        t <= RF[5];
                end
        end
endmodule

//ALU Control
module ALUControl (ALUOp, FuncCode, ALUCtl);
        input  [1:0]  ALUOp;
        input  [5:0]  FuncCode;
        output [3:0]  ALUCtl;
        reg    [3:0]  ALUCtl;

        always@( ALUOp, FuncCode)
        begin
        case(ALUOp)
        2'b00: ALUCtl = 4'b0010;
        2'b01: ALUCtl = 4'b0110;
        2'b10: case(FuncCode)
                        6'b 100000: ALUCtl = 4'b 0010;
                        6'b 100010: ALUCtl = 4'b 0110;
                        6'b 100100: ALUCtl = 4'b 0000;
                        6'b 100101: ALUCtl = 4'b 0001;
                        6'b 101010: ALUCtl = 4'b 0111;
                        default:ALUCtl = 4'b xxxx;
                        endcase
        default:ALUCtl = 4'b xxxx;
        endcase
        end
endmodule
//ALU
```

```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
	input   [3:0]    ALUctl;
	input   [31:0]  A,B;
	output [31:0]  ALUOut;
	output Zero;
	reg      [31:0] ALUOut;

	assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0
	always @(ALUctl, A, B) begin //reevaluate if these change
	case (ALUctl)
		0: ALUOut <= A & B;
		1: ALUOut <= A | B;
		2: ALUOut <= A + B;
		6: ALUOut <= A - B;
		7: ALUOut <= A < B ? 1:0;
		// .... Add more ALU operations here
		default: ALUOut <= A;
		endcase
	end
endmodule

// Data Memory
module DataMemory(Address, DWriteData, MemRead, MemWrite, clock, reset, DReadData);
input   [31:0]  Address, DWriteData;
input                    MemRead, MemWrite, clock, reset;
output [31:0]  DReadData;
reg      [31:0]  DMem[7:0];

assign   DReadData = DMem[Address[2:0]];
always @(posedge clock or posedge reset)begin
		if (reset) begin
			DMem[0]=32'h00000005;
			DMem[1]=32'h0000000A;
			DMem[2]=32'h00000055;
			DMem[3]=32'h000000AA;
			DMem[4]=32'h00005555;
			DMem[5]=32'h00008888;
			DMem[6]=32'h00550000;
			DMem[7]=32'h00004444;
			end else
			if (MemWrite) DMem[Address[2:0]] <= DWriteData;
		end
endmodule
```

```
// Main Controller
module Control (opcode,RegDst,Branch,MemRead,MemtoReg,ALUOp,MemWrite,ALUSrc,RegWrite);
input   [5:0]   opcode;
output [1:0]    ALUOp;
output RegDst,Branch,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite;
reg             [1:0]   ALUOp;
reg     RegDst,Branch,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite;

parameter R_Format = 6'b000000, LW = 6'b100011, SW = 6'b101011, BEQ=6'b000100, J=6'b000010; // The opcode of jump is set as 000010.
always @(opcode)begin
        case(opcode)
                R_Format: {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp}= 9'b 100100010;
                LW:     {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp}= 9'b 011110000;
                SW:     {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp}= 9'b x1x001000;
                BEQ:    {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp}= 9'b x0x000101;
                J:      {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp}= 9'b xxx0001xx;
                // Regarding to the control signal of jump, it will be set to xxx0001xx, the reason will be explained later.
                // .... Add more instructions here
                default: {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp}= 9'b xxxxxxxxx;
                endcase
        end
endmodule

// Datapath
module DataPath(RegDst, Branch, MemRead, MemtoReg, ALUOp, MemWrite,
ALUSrc, RegWrite, clock, reset, opcode,ALUResultOut ,DReadData, x, y, t, PC); // In order to give the value to register x, y, t and PC.

input   RegDst,Branch,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite,clock, reset;
input   [1:0]   ALUOp;
output [5:0]    opcode;
output [31:0]  ALUResultOut ,DReadData, x, y, t, PC; // In order to give the value to register x, y, t and PC.

reg     [31:0] PC, IMemory[0:31];
wire    [31:0] SignExtendOffset, PCOffset, PCValue, ALUResultOut,
                IAddress, DAddress, IMemOut, DmemOut, DWriteData, Instruction,
                RWriteData, DReadData, ALUAin, ALUBin;
wire    [3:0] ALUctl;
wire    Zero;
wire    [4:0] WriteReg;

//Instruction fields, to improve code readability
wire [5:0]      funct;
wire [4:0]      rs, rt, rd, shamt;
wire [15:0] offset;
```

```
//Instantiate local ALU controller
ALUControl alucontroller(ALUOp,funct,ALUctl);

// Instantiate ALU
MIPSALU ALU(ALUctl, ALUAin, ALUBin, ALUResultOut, Zero);

// Instantiate Register File
RegisterFile REG(rs, rt, WriteReg, RWriteData, RegWrite, ALUAin, DWriteData,clock,reset,x, y, t);

// Instantiate Data Memory
DataMemory datamemory(ALUResultOut, DWriteData, MemRead, MemWrite, clock, reset, DReadData);

// Instantiate Instruction Memory
IMemory         IMemory_inst (
        .address ( PC[6:2] ),
        .q ( Instruction )
        );

// Synthesize multiplexers
assign WriteReg      = (RegDst)      ? rd                              : rt;
assign ALUBin        = (ALUSrc)      ? SignExtendOffset    : DWriteData;
assign PCValue       = (Branch & Zero) ? ((opcode==6'b000010)    ?       PCOffset: (PC+4+PCOffset)) : PC+4;
// Both Jump and BEQ will use Branch. However, if the opcode for jump is operated, it will go to the desired PC value, which is PCOffset.
assign RWriteData    = (MemtoReg)   ? DReadData           :       ALUResultOut;
//assignPCValue_j     = (Branch & ALUOp)? PCOffset_j     : PCOffset;
// Acquire the fields of the R_Format Instruction for clarity
assign {opcode, rs, rt, rd, shamt, funct} = Instruction;
// Acquire the immediate field of the I_Format instructions
assign offset = Instruction[15:0];
assign SignExtendOffset = { {16{offset[15]}} , offset[15:0]};
// Multiply by 4 the PC offset
assign PCOffset = SignExtendOffset << 2;
// Write the address of the next instruction into the program counter
always @(posedge clock ) begin
if (reset) PC<=32'h00000000; else
        PC <= PCValue;
end
endmodule

module EEE339(clock, reset,opcode, ALUResultOut ,DReadData, x, y, t, PC); // In order to give the value to register x, y, t and PC.
        input   clock,  reset;
        output [5:0]    opcode;
        output [31:0]  ALUResultOut ,DReadData, x, y, t, PC; // For simulation purposes
```

```
        wire [1:0] ALUOp;
        wire [5:0] opcode;
        wire [31:0] SignExtend, ALUResultOut ,DReadData;
        wire RegDst,Branch,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite;

        // Instantiate the Datapath
        DataPath MIPSDP (RegDst,Branch,MemRead,MemtoReg,ALUOp,
        MemWrite,ALUSrc,RegWrite,clock, reset, opcode, ALUResultOut ,DReadData,x, y, t, PC);

        //Instantiate the combinational control unit
        Control MIPSControl (opcode,RegDst,Branch,MemRead,MemtoReg,ALUOp,MemWrite,ALUSrc,RegWrite);
endmodule
```

MIPS Code:

lw $2, 8(0)
lw $3, 12(0)
j 0

Machine Code:

+0	100011 00000 00010 0000000000000010
+1	100011 00000 00011 0000000000000011
+2	000010 00000000000000000000000000

Table 4: 32-bit machine code for Task 4.

Simulation Result

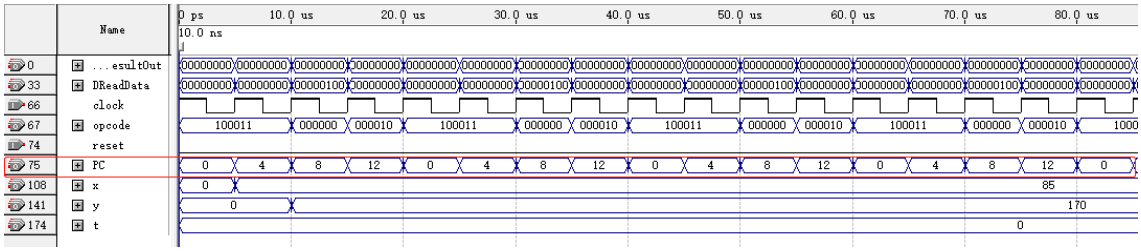

Figure 5: Simulation result for Task 4.

Table 4 indicated the first two instructions load the values to registers and the third instruction was for jump to the desired PC values, which is 0, because

$$Address\ in\ Jump = \frac{New\ PC\ Value}{4}$$

From Figure 5, it could be discovered that PC values jumped to 0 after the third machine code was operated. Therefore, the result is consistent with the expectation.

Task 5 and Task 6.

Modified Verilog Code for processor:

The followings will show the Verilog Code for the processor, the modified codes will be highlighted in yellow. The explanations for every modified code will be appended nearby and highlighted in yellow.

```
// Register File
module RegisterFile (Read1,Read2,Writereg,WriteData,RegWrite, Data1, Data2,clock,reset, x, y, t); // In order to show the value of x, y and t.
	input	[4:0] Read1,Read2,Writereg; // the registers numbers to read or write
	input	[31:0] WriteData;			// data to write
	input	RegWrite;					// The write control
	input	clock, reset;				// The clock to trigger writes
	output [31:0] Data1, Data2, x, y, t;	// the register values read; // In order to show the value of x, y and t.
	reg	[31:0] RF[31:0], x, y, t;		// 32 registers each 32 bits long // In order to show the value of x, y and t.
	integer k;

	// Read from registers independent of clock
	assign Data1 = RF[Read1];
	assign Data2 = RF[Read2];
	// write the register with new value on the falling edge of the clock if RegWrite is high
	always @(posedge clock or posedge reset)
	begin
		if (reset) for(k=0;k<32;k=k+1) RF[k]<=32'h00000000;
		// Register 0 is a read only register with the content of 0
		else	if (RegWrite & (Writereg!=0)) RF[Writereg] <= WriteData;
		begin
			x <= RF[2];
			y <= RF[3];
			t <= RF[5];
		end
	end
endmodule

//ALU Control
module ALUControl (ALUOp, FuncCode, ALUCtl);
	input	[1:0]	ALUOp;
	input	[5:0]	FuncCode;
	output [3:0]	ALUCtl;
	reg	[3:0]	ALUCtl;

	always@( ALUOp, FuncCode)
	begin
	case(ALUOp)
```

```
        2'b00: ALUCtl = 4'b0010;
        2'b01: ALUCtl = 4'b0110;
        2'b11:   ALUCtl = 4'b0001;              //ORI: ALUOp is 11 and ALU control is 0001 which is same as the one for OR.
        2'b10: case(FuncCode)
                        6'b 100000: ALUCtl = 4'b 0010;
                        6'b 100010: ALUCtl = 4'b 0110;
                        6'b 100100: ALUCtl = 4'b 0000;
                        6'b 100101: ALUCtl = 4'b 0001;
                        6'b 101010: ALUCtl = 4'b 0111;
                        default:ALUCtl = 4'b xxxx;
                        endcase
        default:ALUCtl = 4'b xxxx;
        endcase
        end
endmodule
//ALU
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
        input   [3:0]   ALUctl;
        input   [31:0]  A,B;
        output [31:0]  ALUOut;
        output Zero;
        reg     [31:0] ALUOut;

        assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0
        always @(ALUctl, A, B) begin //reevaluate if these change
        case (ALUctl)
                0: ALUOut <= A & B;
                1: ALUOut <= A | B;
                2: ALUOut <= A + B;
                6: ALUOut <= A - B;
                7: ALUOut <= A < B ? 1:0;
                // .... Add more ALU operations here
                default: ALUOut <= A;
                endcase
        end
endmodule

// Data Memory
module DataMemory(Address, DWriteData, MemRead, MemWrite, clock, reset, DReadData);
input   [31:0]  Address, DWriteData;
input   MemRead, MemWrite, clock, reset;
output [31:0]  DReadData;
reg     [31:0]  DMem[7:0];
```

```
assign   DReadData = DMem[Address[2:0]];
always @(posedge clock or posedge reset)begin
        if (reset) begin
            DMem[0]=32'h00000005;
            DMem[1]=32'h0000000A;
            DMem[2]=32'h00000055;
            DMem[3]=32'h000000AA;
            DMem[4]=32'h00005555;
            DMem[5]=32'h00008888;
            DMem[6]=32'h00550000;
            DMem[7]=32'h00004444;
            end else
            if (MemWrite) DMem[Address[2:0]] <= DWriteData;
        end
endmodule

// Main Controller
module Control (opcode, RegDst, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite);

input   [5:0]   opcode;
output [1:0]    ALUOp;
output RegDst,Branch,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite;
reg     [1:0]   ALUOp;
reg     RegDst,Branch,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite;

parameter R_Format = 6'b000000, LW = 6'b100011, SW = 6'b101011, BEQ=6'b000100, J=6'b000010, ori=6'b001101; // The opcode for ORI is 001101
always @(opcode)begin
    case(opcode)
        R_Format: {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp}= 9'b 100100010;
        LW:     {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp}= 9'b 011110000;
        SW:     {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp}= 9'b x1x001000;
        BEQ:    {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp}= 9'b x0x000101;
        J:        {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp}= 9'b xxx0001xx;
        ori:      {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp}= 9'b 010100011;
        // Regarding to the control signal of ori, it will be set to 010100011, the reason will be explained later.
        // .... Add more instructions here
        default: {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp}= 9'b xxxxxxxxx;
        endcase
    end
endmodule

// Datapath
module DataPath(RegDst, Branch, MemRead, MemtoReg, ALUOp, MemWrite,
ALUSrc, RegWrite, clock, reset, opcode,ALUResultOut ,DReadData, x, y, t,PC);
```

```
input   RegDst,Branch,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite,clock, reset;
input   [1:0]   ALUOp;
output [5:0]    opcode;
output [31:0]  ALUResultOut ,DReadData,x, y, t, PC;

reg     [31:0] PC, IMemory[0:31];
wire    [31:0] SignExtendOffset, ZeroExtendOffset, PCOffset, PCValue, ALUResultOut,
                IAddress, DAddress, IMemOut, DmemOut, DWriteData, Instruction,
                RWriteData, DReadData, ALUAin, ALUBin;
wire    [3:0] ALUctl;
wire    Zero;
wire    [4:0] WriteReg;

//Instruction fields, to improve code readability
wire [5:0]      funct;
wire [4:0]      rs, rt, rd, shamt;
wire [15:0] offset;

//Instantiate local ALU controller
ALUControl alucontroller(ALUOp,funct,ALUctl);

// Instantiate ALU
MIPSALU ALU(ALUctl, ALUAin, ALUBin, ALUResultOut, Zero);

// Instantiate Register File
RegisterFile REG(rs, rt, WriteReg, RWriteData, RegWrite, ALUAin, DWriteData,clock,reset,x, y, t);

// Instantiate Data Memory
DataMemory datamemory(ALUResultOut, DWriteData, MemRead, MemWrite, clock, reset, DReadData);

// Instantiate Instruction Memory
IMemory         IMemory_inst (
        .address ( PC[6:2] ),
        .q ( Instruction )
        );

// Synthesize multiplexers
assign  WriteReg        = (RegDst)          ? rd                                    : rt;
assign  ALUBin          = (ALUSrc)    ?    ((opcode==6'b001101)   ? ZeroExtendOffset : SignExtendOffset): DWriteData;
// ALUBin is used for arithmetic calculation. If ALUSrc is high, it will do arithmetic calculation. However, if the opcode for ori is operated, it will
//operate ZeroExtendOffset, which is for immediate arithmetic instructions.
assign  PCValue         = (Branch & Zero) ? ((opcode==6'b000010)     ?          PCOffset : (PC+4+PCOffset)) : PC+4;
assign  RWriteData      = (MemtoReg)    ? DReadData :    ALUResultOut;
```

```
//assignPCValue_j      = (Branch & ALUOp)? PCOffset_j      : PCOffset;
// Acquire the fields of the R_Format Instruction for clarity
assign {opcode, rs, rt, rd, shamt, funct} = Instruction;
// Acquire the immediate field of the I_Format instructions
assign offset = Instruction[15:0];
assign ZeroExtendOffset = { {16{1'b0}} , offset[15:0]};
// For immediate arithmetic instructions.
assign SignExtendOffset = { {16{offset[15]}} , offset[15:0]};
// Multiply by 4 the PC offset
assign PCOffset = SignExtendOffset << 2;
// Write the address of the next instruction into the program counter
always @(posedge clock ) begin
if (reset) PC<=32'h00000000; else
        PC <= PCValue;
end
endmodule

module EEE339(clock, reset,opcode, ALUResultOut ,DReadData, x, y, t, PC);
        input   clock,  reset;
        output [5:0]    opcode;
        output [31:0]  ALUResultOut ,DReadData, x, y, t, PC; // For simulation purposes

        wire [1:0] ALUOp;
        wire [5:0] opcode;
        wire [31:0] SignExtend, ZeroExtend, ALUResultOut ,DReadData;
        wire RegDst,Branch,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite;

        // Instantiate the Datapath
        DataPath MIPSDP (RegDst,Branch,MemRead,MemtoReg,ALUOp,
        MemWrite,ALUSrc,RegWrite,clock, reset, opcode, ALUResultOut ,DReadData,x, y, t, PC);

        //Instantiate the combinational control unit
        Control MIPSControl (opcode,RegDst,Branch,MemRead,MemtoReg,ALUOp,MemWrite,ALUSrc,RegWrite);
endmodule
```

MIPS Code:

lw $2, 8(0)
ori $2, $2, 170

Machine Code:

+0	100011 00000 00010 0000000000000010
+1	001101 00010 00010 0000000010101010

Table 5: 32-bit machine code for Task 6.

Simulation Result

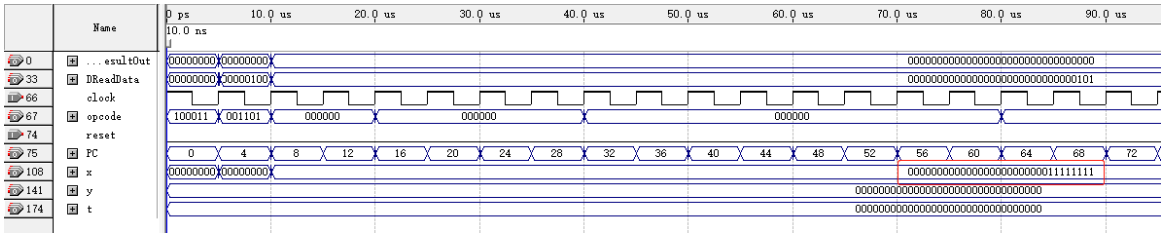

Figure 6: Simulation result for Task 5.

Table 5 is the provided machine code. The first instruction is to load the data stored in the X location of the data memory into X register. The second instruction is to do ori operation on X register and the set actual number whose value is 170. The expected result would be

X	0000000001010101
170	0000000010101010
X after ori	0000000011111111

Therefore, the expected of the result is 0000000011111111. From Figure 6, it could be seen that the final new value in X register has become 0000000011111111. In summary, this task was completed correctly.

Task 7.

Modified Verilog Code for processor:

The followings will show the Verilog Code for the processor, the modified codes will be highlighted in yellow. The explanations for every modified code will be appended nearby and highlighted in yellow. For this section, the only part is to change the number of data bus, therefore there are just two modifications. The rest codes are the same as the ones in previous tasks.

```
module EEE339(clock, reset,opcode, ALUResultOut ,DReadData, x, y, t, PC);
        input   clock,  reset;
        output [5:0]    opcode;
        output [19:0]  ALUResultOut ,DReadData, x, y, t, PC;      // The size of the data bus is 20
        wire [1:0] ALUOp;
        wire [5:0] opcode;
        wire [19:0] SignExtend, ZeroExtend, ALUResultOut ,DReadData;    // The size of the data bus is 20
        wire RegDst,Branch,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite;

        // Instantiate the Datapath
        DataPath MIPSDP (RegDst,Branch,MemRead,MemtoReg,ALUOp,
        MemWrite,ALUSrc,RegWrite,clock, reset, opcode, ALUResultOut ,DReadData,x, y, t, PC);
        //Instantiate the combinational control unit
        Control MIPSControl (opcode,RegDst,Branch,MemRead,MemtoReg,ALUOp,MemWrite,ALUSrc,RegWrite);
endmodule
```

MIPS Code:

lw $2, 8(0)
lw $3, 12(0)
add $4, $2, $3
sw $4, 16(0)
lw $5, 16(0)

Machine Code:

+0	100011 00000 00010 0000000000000010
+1	100011 00000 00011 0000000000000011
+2	000000 00010 00011 00100 00000 100000
+3	101011 00000 00100 0000000000000100
+4	100011 00000 00101 0000000000000100

Table 6: 32-bit machine code for Task 7.

Simulation Result

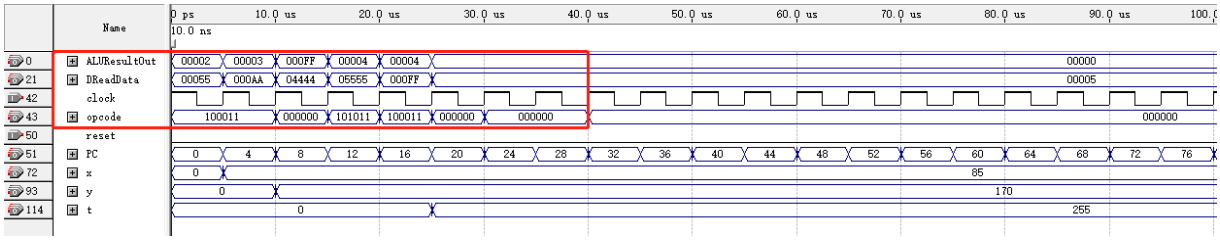

Figure 6: Simulation result for Task 7.

According to the requirement that the size of data bus should be changed to 20, therefore all the codes related to the data bus size should be modified to 20, which could be referenced above. Although the size of the data bus has been changed, the processing will not be influenced because there are no other changes in the processor. Therefore, the simulation would be the same as the one in Task 1.