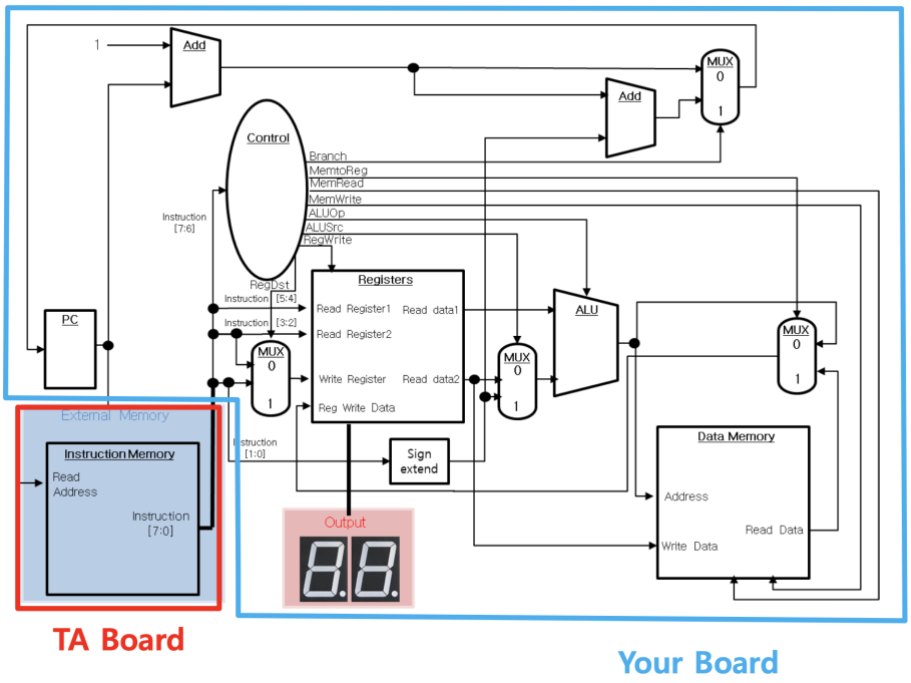
Final Project Report

Park Chaehyun

1. Overall design and structure



For the overall design, we referenced the example design given in the lab session. We made a separate module for each module shown in the diagram. And besides the modules shown in the diagram, we implemented frequency divider module that receives 50Mhz clock oscillator as input and outputs a 1Hz clock. And for simulation purposes, we made an instruction memory module to check that the functions are implemented correctly. When programming the FPGA board, the instruction memory module was not used and was left to be received as input to the board (output of the external instruction memory), and the board gives an additional output, which is the output of the PC, so that the external instruction memory can receive it as input.

Overall structure:

(1) PC keeps track of the address of the instruction, which is given as input the instruction memory.
(2) Output of PC is fed to pcAndOneAdder, where 1 is added to the current PC value. Then MUXtoPC decides whether the 1-added value is fed back to PC or an additional constant (which is translated from the last 2 bits of the instruction by the Sign Extend module) is added first and fed back to PC. This depends on the 'Branch' flag it receives from the control module, which will correctly give the flag depending on whether the operation is 'jump' or not. So the new PC value at the next clock will be (old value + 1) by default and (old value + 1 + constant) if the operation is 'jump'.
(3) The instruction memory outputs the 8-bit instruction at the given address. The instruction is fed to the control module and register module.
(4) Depending on the instruction, control module gives flag outputs fed to multiplexers, register, ALU and memory so that the correct operation can be carried out.
(5) The register receives which registers to read, which registers to write (determined by MUXtoWriteRegister) and what data to write (determined by MUXtoRegWriteData) gives outputs readData1 and readData2, which is fed as the inputs to the multiplexer and ALU. It also outputs the register write data to the 7-segment decoder module, which decodes the 8 bit value to be displayed on 2 7-segment displays in hexadecimal, 4-bit by 4-bit.
(6) Depending on the operation, MUXfromRegToALU chooses either the second data from register or the constant value (which is translated from the last 2 bits of the instruction by the Sign Extend module). And the first data from the register and the output of the MUXfromRegToALU is passed to the ALU, so that both data can be added (adding register values), or the first data and the constant value can be added(address of the memory). The added data from ALU is fed to the memory and MUXtoRegWriteData.

(7) The data memory receives address, write data and read / write flags as input. It reads the data at the given address and outputs it if the MemRead flag is raised, and writes the given data at the given address if write flag is raised.

(8) MUXtoRegWriteData receives the output of ALU and data memory as input and chooses the correct output depending on the operation.

2. Functionality of each module

(1) Microprocessor

```
module microprocessor(
    output [7:0] read_address,
    input [7:0] inst, //commented out when testing
    input clk,
    input rst,
    output [6:0] segA,
    output [6:0] segB,
    output [6:0] segC,
    output [6:0] segD
    );

    //wire [7:0] inst; //uncommented when testing

    wire new_clk;
    wire branch;
    wire memToReg;
    wire memRead;
    wire memWrite;
    wire ALUSrc;
    wire regWrite;
    wire regDst;

    wire [7:0] regWriteData;
    wire [7:0] readData1;
    wire [7:0] readData2;

    wire [1:0] writeRegister;

    wire [7:0] signExtendData;


    wire [7:0] readDataFromMUX;
    wire [7:0] addedData;

    wire [7:0] memoryData;
    wire [7:0] pc;
    wire [7:0] new_pcCount;

    assign read_address = new_pcCount;

    wire [7:0] addedCount;

    wire [7:0] jumpedPC;

    freq_divider T1(.clr(rst), .clk(clk), .clkout(new_clk));
    control T2(.clk(new_clk), .op(inst[7:6]), .branch(branch), .memToReg(memToReg), .memRead(memRead),
.memWrite(memWrite), .ALUSrc(ALUSrc), .regWrite(regWrite), .regDst(regDst));
```

```
    MUXtoWriteRegister T3(.inst32(inst[3:2]), .inst10(inst[1:0]), .RegDst(regDst), .out(writeRegister));
    register T4(.rst(rst), .clk(new_clk), .regWrite(regWrite), .readRegister1(inst[5:4]), .readRegister2(inst[3:2]),
.writeRegister(writeRegister), .regWriteData(regWriteData), .readData1(readData1), .readData2(readData2));
    MUXfromRegToALU T5(.readData(readData2), .signExtend(signExtendData), .ALUSrc(ALUSrc),
.out(readDataFromMUX));
    ALU T6(.readData1(readData1), .readDataFromMUX(readDataFromMUX), .addedData(addedData));
    MUXtoRegWriteData T7(.memToReg(memToReg), .ALUData(addedData), .MemoryData(memoryData),
.outData(regWriteData));
    memory T8(.rst(rst), .address(addedData), .writeData(readData2), .memRead(memRead),
.memWrite(memWrite), .clk(clk), .readData(memoryData));
    signExtend T9(.two_bit(inst[1:0]), .eight_bit(signExtendData));
    pc T10(.rst(rst), .inputCount(pc), .clk(new_clk), .out(new_pcCount));
    pcAndOneAdder T11(.pcCount(new_pcCount), .addedCount(addedCount));
    SignExtendAdder T12(.PCAdded(addedCount), .jump(signExtendData), .out(jumpedPC));
    MUXtoPC T13(.addedPC(addedCount), .jumpedPC(jumpedPC), .branch(branch), .out(pc));
    //IMEM_ppt T18(.read_address(new_pcCount), .instruction(inst)); // uncommented when testing
    sevenSegDecoder T15(.data(regWriteData), .segA(segA), .segB(segB));
    sevenSegDecoder T17(.data(new_pcCount), .segA(segC), .segB(segD));

endmodule
```

---

microprocessor makes an instance of every module and connects the inputs and outputs of the modules correctly, according to the diagram. The output read_address will be given to the external instruction memory so that it can receive the 8-bit instruction from it. The outputs segA, segB are the decoded 7-segment data to display the regWriteData in hexadecimal. The outputs segC, segD are the decoded 7-segment data to display the current PC value in hexadecimal.

(2) PC

```
module pc(
    input [7:0] inputCount,
    input clk,
    input rst,
    output reg [7:0] out
    );

    always @(posedge clk or posedge rst) begin
    if (rst)
        out <= 8'b00000000;
    else
        out <= inputCount;
    end

endmodule
```

PC receives the new value as input from the output of “(7) MUX to PC”, and gives the new value as output at the next rising edge of the clock. The output goes into “(3) Instruction memory” (when testing) or as the board’s output read_address (when programming FPGA board) and to “(4) PC and 1 adder”.

(3) Instruction memory (for testing)

```
module IMEM_ppt(
	input [7:0] read_address,
	output [7:0] instruction
	);

	wire [7:0] MemByte[31:0];

	assign MemByte[0] = {2'b01, 2'b00, 2'b10, 2'b01};

	assign MemByte[1] = {2'b11, 2'b00, 2'b00, 2'b01};

	assign MemByte[2] = {2'b00, 2'b01, 2'b10, 2'b00};

	assign MemByte[3] = {2'b10, 2'b10, 2'b10, 2'b01};

	assign MemByte[4] = {2'b01, 2'b00, 2'b11, 2'b01};

	assign instruction = MemByte[read_address];
endmodule
```

The instruction memory receives read_address as input from “(2) PC” and gives the stored value at the read_address index. 5 values are stored as example. This is only used in simulation by uncommenting the lines in microprocessor.v and the result of this example is shown in “3. Simulation result”.

(4) PC and 1 adder

```
module pcAndOneAdder(
	input [7:0] pcCount,
	output [7:0] addedCount
	);

	assign addedCount = pcCount + 1;

endmodule
```

This adds the output of PC and 1. The output goes into “(6) PC and constant adder” and “(7) MUX to PC”.

(5) Sign extend 2-bit to 8-bit (for constant)

```
module signExtend(
	input [1:0] two_bit,
	output reg [7:0] eight_bit
	);

		always @(*) begin
		case (two_bit)
			2'b00: eight_bit <= 8'b00000000;
			2'b01: eight_bit <= 8'b00000001;
			2'b10: eight_bit <= 8'b11111110;
			2'b11: eight_bit <= 8'b11111111;
		endcase
	end
```

```
endmodule
```

This receives the last two bits of the instruction (inst[1:0]) as input and converts the 2-bit value into the corresponding 8-bit 2's complement value. The output is fed to "(6) PC and constant adder" and "(11) MUX from register to ALU".

(6) PC and constant adder

```
module SignExtendAdder(
    input [7:0] PCAdded,
    input [7:0] jump,
    output [7:0] out
    );

        assign out = PCAdded + jump;

endmodule
```

It receives the 1-added value of the current PC and the jump constant as inputs, and outputs the addition of the two values. The output is fed to "(7) MUX to PC".

(7) MUX to PC

```
module MUXtoPC(
    input [7:0] addedPC,
    input [7:0] jumpedPC,
    input branch,
    output [7:0] out
    );

        assign out = (branch == 1'b0) ? addedPC : jumpedPC;

endmodule
```

This receives the 1-added value of the current PC from "(4) PC and 1 adder" and 1-and-constant-added value from "(6) PC and constant adder". If branch is 0, this is not a jump operation so it outputs (4), and when branch is 1, this is a jump operation so it outputs (6). The output is fed to PC.

(8) Control

```
module control(
    input clk,
    input [1:0] op,
    output reg branch,
    output reg memToReg,
    output reg memRead,
    output reg memWrite,
    output reg ALUSrc,
    output reg regWrite,
    output reg regDst
    );

    initial begin
        branch = 0;
    end

    always @(op) begin
```

```
        branch <= (op == 2'b11) ? 1'b1 : 1'b0;
        memToReg <= (op == 2'b00) ? 1'b0 : 1'b1;
        memRead <= (op == 2'b01) ? 1'b1 : 1'b0;
        memWrite <= (op == 2'b10) ? 1'b1 : 1'b0;
        ALUSrc <= (op == 2'b00) ? 1'b0 : 1'b1;
        regWrite <= (op == 2'b00 || op == 2'b01) ? 1'b1 : 1'b0;
        regDst <= (op == 2'b00) ? 1'b1 : 1'b0;
    end

endmodule
```

This receives the first 2-bits of instruction(inst[7:6]) as input and gives flags as outputs to other modules.
00: add the values of 2 registers and save to a register. (add)
01: read register, add a constant to the value, and use it as read address of memory, and save that value of the memory to a register. (load from memory)
10: read register, add a constant to the value, and use it as read address of memory. Read another register, and save that value of the register at the address of the memory. (save to memory)
11: add a constant to the next value of PC (jump)

i) branch
This is raised when input is 11, meaning this is a jump operation. This is given to "(7) MUX to PC" so that it can output the correct next PC value.

ii) memToReg
This is raised when input is not 00. Only when input is 00, the writing data is the added value of two registers. Otherwise, the writing data is the value referenced by the address in the memory. So this memToReg flag is given to the "MUX to regWriteData", which takes both added value and memory read value, so that it can choose the correct output depending on the flag.

iii) memRead
This is raised when input is 01, and given to the memory. When it is raised, the memory will read the value stored at its input "address" and outputs it.

iv) memWrite
This is raised when input is 10, and given to the memory. When it is raised, the memory will write the value of its input "writeData" at its input "address" and outputs it.

v) ALUSrc
This is 0 when input is 00 and given to "MUX to ALU". When 0, "MUX to ALU" will choose the second readData from the register and gives it to the ALU, so that ALU can add the first readData and second readData, performing add operation (00). Otherwise, it will add the constant to the first readData value, for 01 or 10 operation.

vi) regWrite
This is raised when input is either 00 or 01 and is given to the register. These operations require writing to the register, so the register will only write the value given as its input "regWriteData"

vii) regDst
This is raised when input is 00 and given to "MUX to write register". When adding (00), the register to be written on is in inst[1:0], and when loading (01), it is on inst[3:2]. Using this flag, the MUX can correctly choose the 2-bit data and feed it to the register.

(9) MUX to writeRegister

```
module MUXtoWriteRegister(
    input [1:0] inst32,
    input [1:0] inst10,
    input RegDst,
```

```
    output reg [1:0] out
    );

    always @(*) begin
        out = (RegDst == 1'b0) ? inst32 : inst10;
    end

endmodule
```

This MUX receives inst[3:2], inst[1:0] and RegDst flag as inputs. Depending on the flag, it will output the correct data which is fed to the register (as explained in “(8) Control-vii)regDst”).

(10) Register

```
module register(
    input rst,
    input regWrite,
    input clk,
    input [1:0] readRegister1,
    input [1:0] readRegister2,
    input [1:0] writeRegister,
    input [7:0] regWriteData,
    output reg [7:0] readData1,
    output reg [7:0] readData2
    );

    reg [7:0] data0, data1, data2, data3;

    initial begin
        data0 = 8'b0;
        data1 = 8'b0;
        data2 = 8'b0;
        data3 = 8'b0;
    end

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            data0 <= 8'b0;
            data1 <= 8'b0;
            data2 <= 8'b0;
            data3 <= 8'b0;
        end
        else if (regWrite) begin
            case (writeRegister)
                2'b00: data0 <= regWriteData;
                2'b01: data1 <= regWriteData;
                2'b10: data2 <= regWriteData;
                2'b11: data3 <= regWriteData;
            endcase
        end
    end

    always @(*) begin
        case (readRegister1)
            2'b00: readData1 <= data0;
            2'b01: readData1 <= data1;
```

```
            2'b10: readData1 <= data2;
            2'b11: readData1 <= data3;
        endcase

        case (readRegister2)
            2'b00: readData2 <= data0;
            2'b01: readData2 <= data1;
            2'b10: readData2 <= data2;
            2'b11: readData2 <= data3;
        endcase
    end

endmodule
```

The register receives as inputs the data to be written, the register for the data to be written on, which register to read. If the regWrite flag is raised, it will write “regWriteData” to the register correspoding to the index given by “writeRegister”. And it will give the value stored at the register of the index given by “readRegister1” and “readResgister2” and gives as outputs “readData1” and “readData2” respectively. “readData1” is given to ALU, and “readData2” is given to the memory as “writeData” and to “MUX from register to ALU”.

(11) MUX from register to ALU

```
module MUXfromRegToALU(
    input [7:0] readData,
    input [7:0] signExtend,
    input ALUSrc,
    output [7:0] out
    );

    assign out = (ALUSrc == 1'b0) ? readData : signExtend;

endmodule
```

This MUX receives readData2 from the register and sign extended data from “(5) Sign extend 2-bit to 8-bit”. It will correctly output the data which is fed to the ALU (as explained in “(8) Control-v) ALUSrc”).

(12) ALU

```
module ALU(
    input [7:0] readData1,
    input [7:0] readDataFromMUX,
    output [7:0] addedData
    );

    assign addedData = readData1 + readDataFromMUX;

endmodule
```

This ALU adds the “readData1” from register and the data from “MUX from register to ALU”. Depending on the operation, the MUX will give either the second register value or a constant value, and the ALU just has to add the two inputs. The added data is fed to the memory as read address and “MUX to regWriteData”.

(13) Memory

```
module memory(
    input [7:0] address,
    input [7:0] writeData,
    input rst,
```

```
input memRead,
input memWrite,
input clk,
output reg [7:0] readData
);

reg [7:0] memory [31:0];

initial begin
	memory[0] = 8'b00000000;
	memory[1] = 8'b00000001;
	memory[2] = 8'b00000010;
	memory[3] = 8'b00000011;
	memory[4] = 8'b00000100;
	memory[5] = 8'b00000101;
	memory[6] = 8'b00000110;
	memory[7] = 8'b00000111;
	memory[8] = 8'b00001000;
	memory[9] = 8'b00001001;
	memory[10] = 8'b00001010;
	memory[11] = 8'b00001011;
	memory[12] = 8'b00001100;
	memory[13] = 8'b00001101;
	memory[14] = 8'b00001110;
	memory[15] = 8'b00001111;

	memory[16] = 8'b00000000;
	memory[17] = 8'b11111111;
	memory[18] = 8'b11111110;
	memory[19] = 8'b11111101;
	memory[20] = 8'b11111100;
	memory[21] = 8'b11111011;
	memory[22] = 8'b11111010;
	memory[23] = 8'b11111001;
	memory[24] = 8'b11111000;
	memory[25] = 8'b11110111;
	memory[26] = 8'b11110110;
	memory[27] = 8'b11110101;
	memory[28] = 8'b11110100;
	memory[29] = 8'b11110011;
	memory[30] = 8'b11110010;
	memory[31] = 8'b11110001;
end

always @(posedge clk or posedge rst) begin
	if (rst) begin
		memory[0] <= 8'b00000000;
		memory[1] <= 8'b00000001;
		memory[2] <= 8'b00000010;
		memory[3] <= 8'b00000011;
		memory[4] <= 8'b00000100;
		memory[5] <= 8'b00000101;
		memory[6] <= 8'b00000110;
		memory[7] <= 8'b00000111;
		memory[8] <= 8'b00001000;
		memory[9] <= 8'b00001001;
		memory[10] <= 8'b00001010;
```

```
                memory[11] <= 8'b00001011;
                memory[12] <= 8'b00001100;
                memory[13] <= 8'b00001101;
                memory[14] <= 8'b00001110;
                memory[15] <= 8'b00001111;

                memory[16] <= 8'b00000000;
                memory[17] <= 8'b11111111;
                memory[18] <= 8'b11111110;
                memory[19] <= 8'b11111101;
                memory[20] <= 8'b11111100;
                memory[21] <= 8'b11111011;
                memory[22] <= 8'b11111010;
                memory[23] <= 8'b11111001;
                memory[24] <= 8'b11111000;
                memory[25] <= 8'b11110111;
                memory[26] <= 8'b11110110;
                memory[27] <= 8'b11110101;
                memory[28] <= 8'b11110100;
                memory[29] <= 8'b11110011;
                memory[30] <= 8'b11110010;
                memory[31] <= 8'b11110001;
            end
            else if (memWrite) begin
                memory[address] <= writeData;
            end
        end

        always @(memRead) begin
            if (memRead) begin
                readData <= memory[address];
            end
        end
endmodule
```

The memory holds 32 8-bit data. Initially, the first 16 data are set to 0, 1, 2, …, 15 respectively. And the last 16 data are set to 0, -1, -2, …, -15 respectively (in 2's complement). If memWrite flag is raised, it will save writeData to the index given by the input. When memRead flag is raised, it will output the data stored at the index given by the input .The output is given immediately so that the register module can use the readData to write to its register, while memory writing is done at the next rising edge of the clock.

(14) MUX to regWriteData

```
module MUXtoRegWriteData(
    input memToReg,
    input [7:0] ALUData,
    input [7:0] MemoryData,
    output [7:0] outData
    );

    assign outData = (memToReg == 1'b1) ? MemoryData : ALUData;


endmodule
```

This MUX gives the data from the memory if memToReg flag is raised. Otherwise, it will give the data from the ALU, because it will be an add operation. The output goes into the register and to the 7-segment decoder so that

it can be displayed.

(15) 7-segment decoder

```
module sevenSegDecoder(
      input [7:0] data,
      output reg [6:0] segA,
      output reg [6:0] segB
      );

   always @(data) begin
      case (data[7:4])
         4'b0000: segA <= 7'b0111111;
         4'b0001: segA <= 7'b0000110;
         4'b0010: segA <= 7'b1011011;
         4'b0011: segA <= 7'b1001111;
         4'b0100: segA <= 7'b1100110;
         4'b0101: segA <= 7'b1101101;
         4'b0110: segA <= 7'b1111101;
         4'b0111: segA <= 7'b0000111;
         4'b1000: segA <= 7'b1111111;
         4'b1001: segA <= 7'b1100111;
         4'b1010: segA <= 7'b1110111;
         4'b1011: segA <= 7'b1111100;
         4'b1100: segA <= 7'b0111001;
         4'b1101: segA <= 7'b1011110;
         4'b1110: segA <= 7'b1111001;
         4'b1111: segA <= 7'b1110001;
      endcase

      case (data[3:0])
         4'b0000: segB <= 7'b0111111;
         4'b0001: segB <= 7'b0000110;
         4'b0010: segB <= 7'b1011011;
         4'b0011: segB <= 7'b1001111;
         4'b0100: segB <= 7'b1100110;
         4'b0101: segB <= 7'b1101101;
         4'b0110: segB <= 7'b1111101;
         4'b0111: segB <= 7'b0000111;
         4'b1000: segB <= 7'b1111111;
         4'b1001: segB <= 7'b1100111;
         4'b1010: segB <= 7'b1110111;
         4'b1011: segB <= 7'b1111100;
         4'b1100: segB <= 7'b0111001;
         4'b1101: segB <= 7'b1011110;
         4'b1110: segB <= 7'b1111001;
         4'b1111: segB <= 7'b1110001;
      endcase
   end

endmodule
```

This decoder receives 8-bit data as input. And it will decode the first 4 bits (data[7:4]) and the last 4 bits (data[3:0]) to the seven-segment display of the corresponding hexadecimal value.

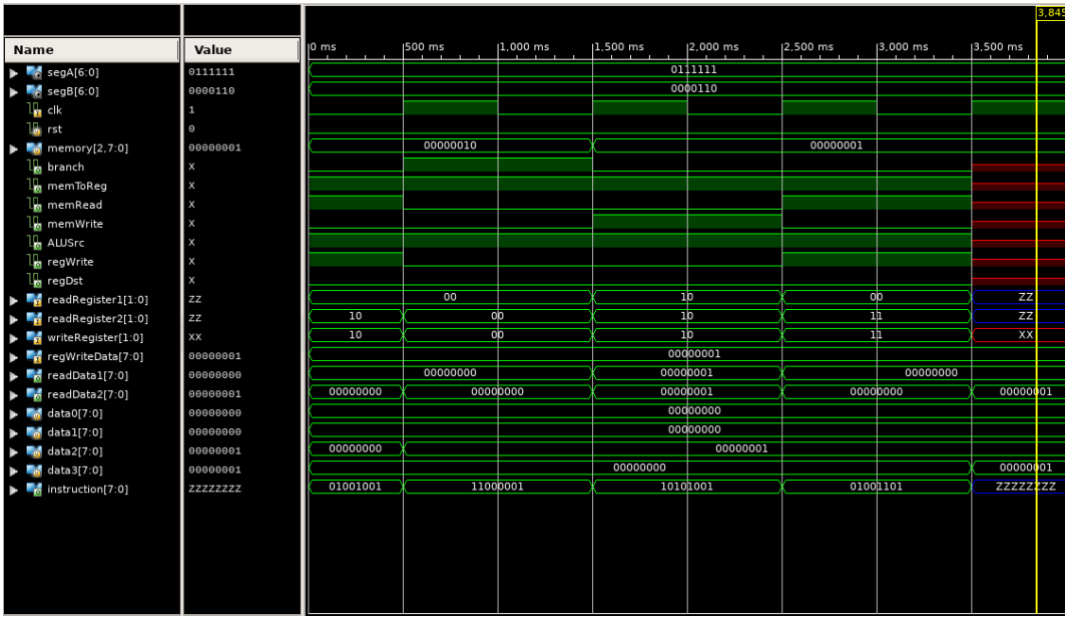
(16) Frequency divider

```
module freq_divider(
```

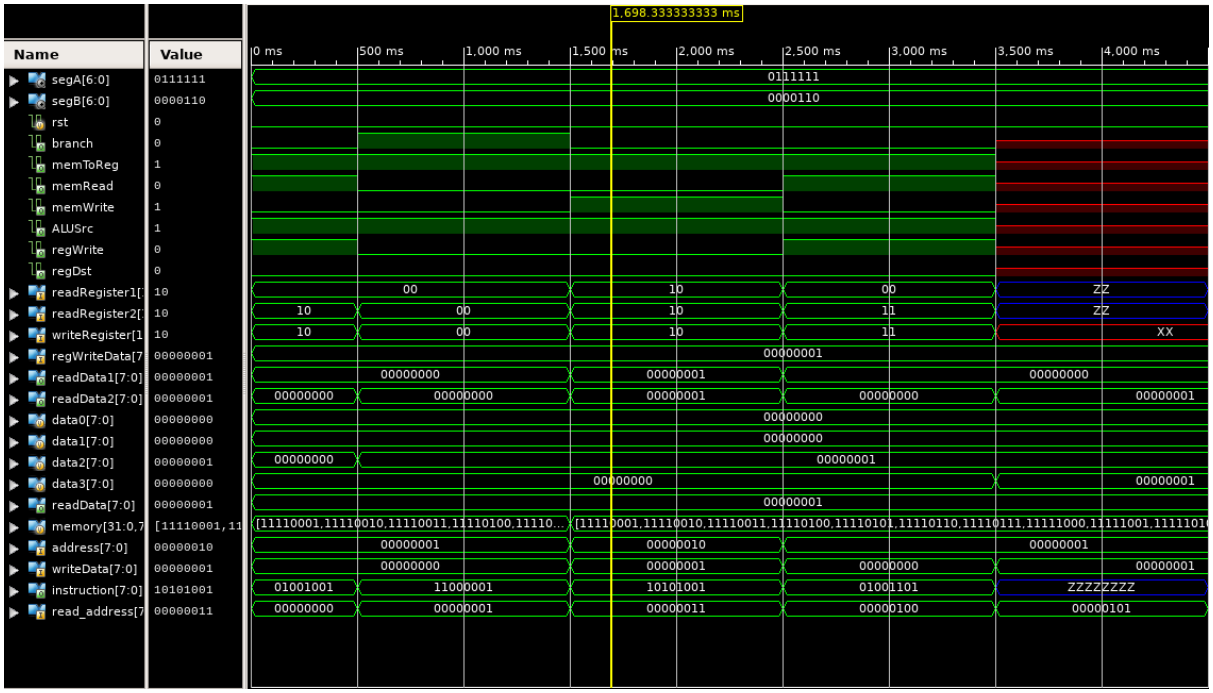
```
input clk,
input clr,
output reg clkout
);

reg[31:0] cnt;

initial begin
	cnt <=32'd0;
	clkout<= 1'b0;
end

always@ (posedge clk) begin
	if(clr) begin
		cnt <=32'd0;
		clkout <= 1'b0;
	end
	else if (cnt == 32'd25000000) begin
		cnt <= 32'd0;
		clkout <= ~clkout;
	end
	else begin
		cnt <= cnt + 1;
	end
end

endmodule
```

The frequency divider receives clock and clear as input, and gives another clock as output. It has an internal variable count. The count and the output clock is set to 0 when clear is 1. Then count is incremented by 1 at every positive edge of the input clock. When count reaches 25000000(=25M), the output clock is toggled. So if 50MHz clock is given as input, the output clock is flipped every 0.5s, meaning the period is 1s, meaning the frequency is 1Hz.

3. Simulation result



Memory[2] is changed at 1500ms



All operations work as intended

The instructions given are as follows:

01001001
11000001
00011000
10101001
01001101

We check that the clock created by the frequency divider is of frequency 1Hz, as intended. Now we check whether instructions are carried out correctly:

(i) First instruction: 01001001
This instruction is equal to “$s2 = Mem[$s0 + 1]”. Initially, $s0=$s1=$s2=$s3=0. So the value stored i n Mem[1] is saved $s2. Initially, Mem[1]=1. So we check that at the rising edge of the clock at 500 ms, data2 ($s2) is changed from 00000000 to 00000001.

(ii) Second instruction: 11000001
This instruction is equal to “go to (next PC + 1)”. So it should skip the next instruction and operate the instruction after that. We check that at the next rising edge of the clock at 1500ms, the instruct ion is changed to 10101001 instead of 00011000.

(iii) Third instruction: 00011000
Not executed

(iv) Fourth instruction: 10101001
This instruction is equal to “Mem[$s2+1] = $s2”. $s2 equals to 1, because of the result of (i). So the instruction is “Mem[2] = 1”. We check that at the rising clock edge 1500ms, the memory checks “m emWrite” flag and writes to the memory. We can see that memory[2] is changed from its initial valu e 00000010 to 00000001.

(v) Fifth instruction: 01001101
This instruction is equal to “$s3 = Mem[$s0+1]”. $s0 equals to 0, so this instruction is “$s3 = Mem[1]”. Mem[1] currently has its initial value 1. So we check that at the next rising edge of the clock at 3500ms, data3 ($s3) is changed from 00000000 to 00000001.