Logic Design Final Project

2021-14284
Taehyun yang
Class2- team 20
5/24

A. Structure of our Micro processor

We used the following modules in order to create a micro processor:

1. BCD to 7 segment: was used to represent binary codes to hexadecimals into 7segmented signals.

It takes 4bit inputs and is named as BCD_to_7seg.v. It takes [3:0] bcd as inputs and outputs as [6:0] seg and any numbers above the decimal point is represented with alphabet letters.

2. Frequency divider: was used to convert the 25MHZ clock to 1hz.

The frequency divider module is called freq_divider.v. This module takes clk and clr as the input and outputs clkout. When the positive edge of the input is active and when clr is also active, the counter resets to 0 and increase the counter otherwise. In order to reverse the clkout every 0.5 seconds and output 1hz clkout, if counter has reached 25*10^6, it reverses the clkout and resets the counter back to 0.

3. Data Memory: was used to contain previous data

Data memory module is named as DMEM.V. This module utilizes inputs of memA, memWD the representes memory address and data that will be used in the memory. Control signal then reads and writes over memRE and memWE. We used register 0~7 and MEMbyte 0~31 to store data and when memRD and memRE is high, it sets output as 0.

4. CONTROL unit: is the module that produces the correct control signal based on each operation

Control unit module is named as CTRL.v. This control unit receives 0~1 op as input and has 0~7 op as outputs. For corresponding operation, the control sets the control signal for each operation as 8bit signals. The signals are in the same order as given in the lab instructions. From REGDST, REGWRITE to ALUOP.

5. Register FILE: Is the register file module that manages the 8bit registers

This Register file module is named as RFILE.V

This register takes 2 bit inputs as regRI1, regRI2, and regWI in order to take register value from the instruction reading 2values at once while writing 1 value. It then uses that information to 8 bit ouput as regRD1, regRD2, and sends regWD as another input. This register file takes clk and clr as asynchronous resets as inputs. They work with the always function whenever each clk and clr positive edge is triggered and resets with all registers going back to 0

6. Processor:

Processor is the center of all the modules that are listed above. This processor performs one instruction per cycle and is named as PROC.V

It takes inputs as [7:0] inst, clk and clr both as 25MHZ asynchronous inputs. We use freq_divider module to convert these inputs to 1hz clocks and input them through other modules. For every positive edge on the clk, we change the pc to next PC and whenever positive edge is met or enable is shown on the clr, we reset the pc back to 0.

and output as [7:0] pc. It fetches each instruction and outputs to the pc. [6:0] num 1 and num 2is used by BCD to 7 seg module to translate the inputs take from RFile into 7 segments. In order to decode the instruction code, we take the instruction as op, rs1, rs2, rd, imm, immEXt. To read the register file, we input these decoded values to RFILE to find rVAL1 and rVal2. And to calculate the register values and immEXT, we use ALUSRC signals to calculate the alures value. Also we used the BRANCH signal to change nextPC as pc+1 or pc+1+immEXT.

To read and write the memories, we used control signals and allures as addresses and connect them to rVal2 and memRD values to read them.

For Writebacks, depending on the REGDST, the regWI would alternate between rd or rs2 values. regWD would change to memRD or alures values depending on MEMTOREG. These values are also linked back to RFILE modules.

B. Lab Results

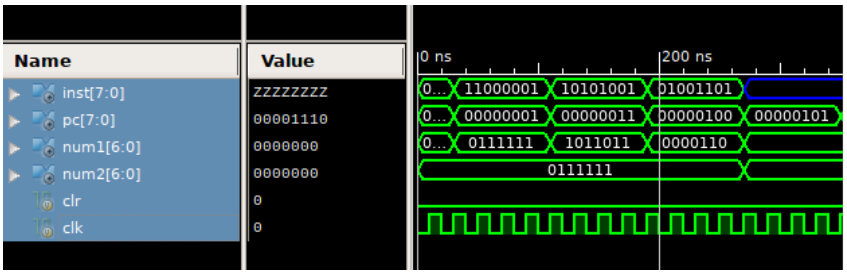
Inorder to upload these codes onto SNU Logic board, we used planahead to initialize the pins like so:

PC and inst was used to communicate to instruction memory, num1 and 2 were linked inorder for 7segment conversions, clk was connected to p57 which is a 25MHZ oscillator and clr was connected to p47, and we used impact to upload through the FPBG board.

1. Example code given from lab instructions:

```
1   lw      $s2 , 1( $s0) ;
2   j       1;
3   add     $s0 , $s1 , $ s 2 ;
4   sw      $s2 ,  1( $s2) ;
5   lw      $s3 ,  1( $s0) ;
```

Result:



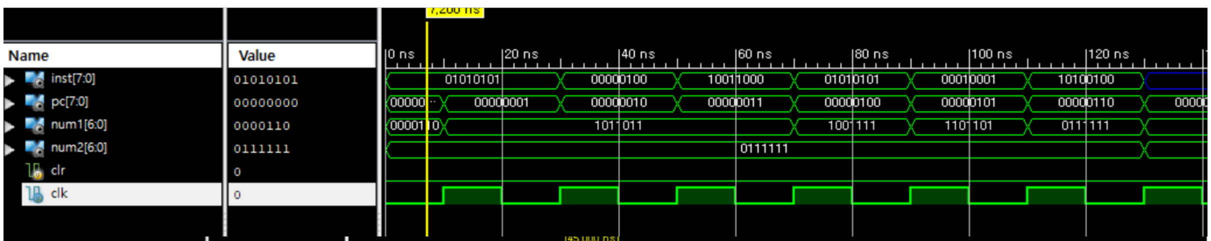
Name	Value
inst[7:0]	ZZZZZZZZ
pc[7:0]	00001110
num1[6:0]	0000000
num2[6:0]	0000000
clr	0
clk	0

The simulation result from the lab result is expected to give 1, 0, 2, 1 and we can check so in our test bench

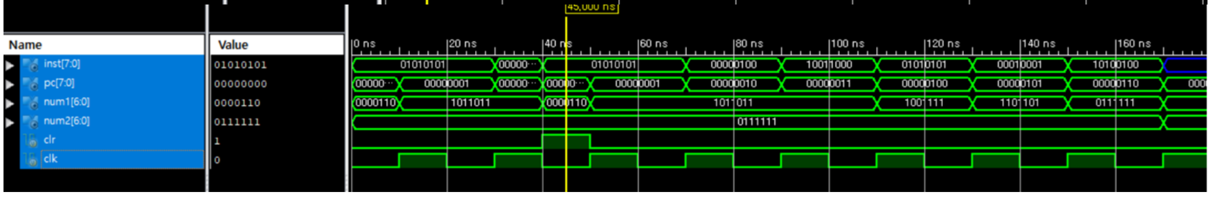
2. Loading and Storing

This code is used to load and store program

```
1   lw      $s1 ,  1 ( $ s 1 ) ;
2   lw      $s1 ,  1 ( $ s 1 ) ;
3   add     $s0 ,  $s1 , $ s 0 ;
4   sw      $s2 ,  0 ( $ s 1 ) ;
5   lw      $s1 ,  1 ( $ s 1 ) ;
6   add     $s1 ,  $s0 , $ s 1 ;
7   sw      $s1 ,  0 ( $ s 2 ) ;
```



Name	Value
inst[7:0]	01010101
pc[7:0]	00000000
num1[6:0]	0000110
num2[6:0]	0111111
clr	0
clk	0



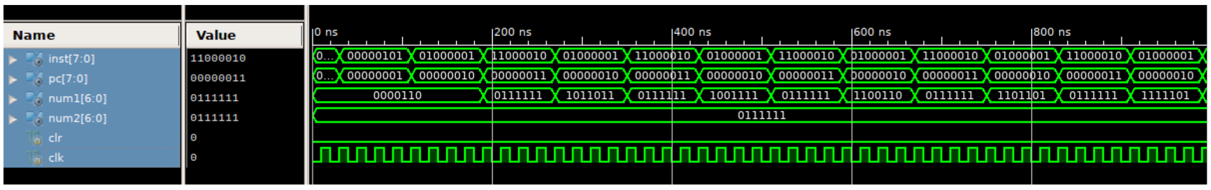
Name	Value
inst[7:0]	01010101
pc[7:0]	00000000
num1[6:0]	0000110
num2[6:0]	0111111
clr	1
clk	0

The correct output should be as 1, 2, 2, 2, 3, 5, 0 and we can see from the first image that it gives proper results.
From the second image, we can see that pc and output is properly reset when given clr input.

3. Jump and ADD

After it has completed lw and add, it jumps back to lw and j repeating the two.

```
1   lw       $s1 , 1( $s0) ;
2   add      $s1 , $s0 , $ s 1 ;
3   lw       $s0 , 1( $s0) ;
4   j        −2;
```



The results should be 1, 1, 1, 0, 2, 0, 3,0, 4, 0, 5… and constantly increments after giving a 0 as output. The image above shows that it represents proper outputs

C. Source Codes

BCD_to_7seg

BCD_to_7seg.v

```
`timescale 1ns /  1ps

module BCD_to_7seg(
        input [3:0] bcd,
                output reg [6: 0] seg
        );

                always@( bcd)begin
                 case(bcd)
                        4'd0:      seg  <=  7'b0111111;
                        4'd1:      seg  <=  7'b0000110;
                        4'd2:      seg  <=  7'b1011011;
                        4'd3:      seg  <=  7'b1001111;
                        4'd4:      seg  <=  7'b1100110;
                        4'd5:      seg  <=  7'b1101101;
                        4'd6:      seg  <=  7'b1111101;
                        4'd7:      seg  <=  7'b0000111;
                        4'd8:      seg  <=  7'b1111111;
                        4'd9:      seg  <=  7'b1101111;
                        4'd10:     seg  <=  7'b1110111;     // A
                        4'd11:     seg  <=  7'b1111100;     // B (b)
                        4'd12:     seg  <=  7'b0111001;     // C
                        4'd13:     seg  <=  7'b1011110;     // D (d)
                        4'd14:     seg  <=  7'b1111001;     // E
                        4'd15:       seg   <=   7'b1110001;     // F
                        default:             seg <= 7'b0000000;
                endcase
        end

endmodule
```

Freq divider

freq_divider.v

```
`timescale 1ns /  1ps

module  freq_divider(
        input clk,
        input clr,
        output reg clkout = 0
        );

          reg [31 : 0] cnt =0;
          always@ ( posedge clk , posedge clr) begin
                if( clr)begin
                        cnt <=  0;
                        clkout <=  0;
                end
                else  if(cnt  ==  32'd25000000)  begin
                        cnt <=  0;
                        clkout <=~ clkout;
                end
                else  begin
                        cnt <=  cnt+1;
                end
        end

endmodule
```

Control Source code

CTRL.v

```
`timescale 1ns/1ps

module Control(op,  ctrl);
        input [1: 0]op;
        output reg [7: 0] ctrl;

          always@(op) begin
                  case(op)
                            2'd0: ctrl = 8'b11000001;
                            2'd1: ctrl = 8'b01101010;
                            2'd2: ctrl = 8'b00100100;
                            2'd3: ctrl = 8'b00010010;
                            default: ctrl =  0;
                  endcase
          end

endmodule
```

DMEM source code

DMEM.v

```
`timescale    1ns / 1ps

module DMEM (
        input [7:0] memA,
        output [7:0] memRD,
        input memRE,
        input [7:0] memWD,
        input memWE,
        input clk,
        input clr
        );

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

        assign memRD = memRE?MemByte[memA]:0;

        initial begin
                reset();
        end

        always @( posedge clk , posedge clr) begin
                if( clr)begin
                        reset();
                end
                else if( memWE) begin
                        MemByte[ memA] <=memWD;
                end

        end

        task reset;
                begin
                   MemByte[0]    <=   8'h00;
                   MemByte[1]    <=   8'h01;
                   MemByte[2]    <=   8'h02;
                   MemByte[3]    <=   8'h03;
                   MemByte[4]    <=   8'h04;
                   MemByte[5]    <=   8'h05;
                   MemByte[6]    <=   8'h06;
                   MemByte[7]    <=   8'h07;
                   MemByte[8]    <=   8'h08;
                   MemByte[9]    <=   8'h09;
                  MemByte[10]   <=   8'h0a;
                  MemByte[11]   <=   8'h0b;
                  MemByte[12]   <=   8'h0c;
                  MemByte[13]   <=   8'h0d;
                  MemByte[14]   <=   8'h0e;
                  MemByte[15]   <=   8'h0f;
                  MemByte[16]   <=   8'h00;
                  MemByte[17]   <=   8'hff;
                  MemByte[18]   <=   8'hfe;
                  MemByte[19]   <=   8'hfd;
                  MemByte[20]   <=   8'hfc;
                  MemByte[21]   <=   8'hfb;
                  MemByte[22]   <=   8'hfa;
                  MemByte[23]   <=   8'hf9;
                  MemByte[24]   <=   8'hf8;
                MemByte[25]  <= 8'hf7;
                MemByte[26]  <= 8'hf6;
                MemByte[27]  <= 8'hf5;
                MemByte[28]  <= 8'hf4;
                MemByte[29]  <= 8'hf3;
                MemByte[30]  <= 8'hf2;
                MemByte[31]  <= 8'hf1;
                end
        endtask

endmodule
```