LDLAB – FINAL

수리과학부 2019-16022 박채연

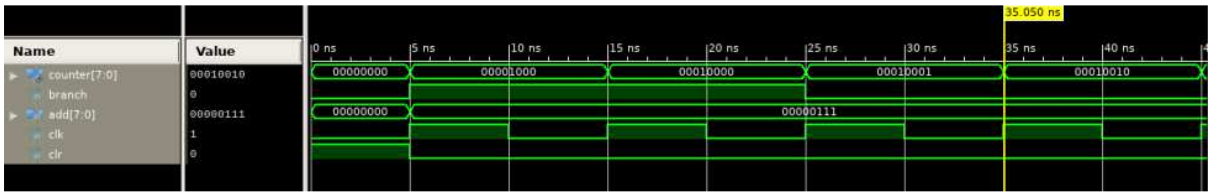
1. Implement the 8 bit microprocessor in Verilog and simulate the behavior.

LDLAB Final Project 관련 PPT에 제시된 스펙을 따라 ALU, PC, Control Unit 등의 모듈을 각각 구현하고 이를 Microprocessor 모듈 내에서 하나로 합치는 방법을 택하였다. 각 모듈별 구현 방식 및 시뮬레이션 결과를 정리하면 아래와 같다.

(1) PC

```
module PC(
    input branch,
    input [7:0] add,
    input clk,
            input clr,
            output reg [7:0] counter
    );

    always@(posedge clk or posedge clr)
    begin
                                if (clr == 1)
                                        begin
                                                                counter = 8'b00000000; //Clear the counter
                                        end
                                        else begin
                                                                if (branch == 1) begin
                                                                        counter = counter + add + 8'b00000001; // branch target
                                                                end
                                                                else begin
                                                                        counter = counter + 8'b00000001; // next instruction
                                                                end
                                end
                        end

endmodule
```

PC란 Program Counter의 약자로, 다음에 실행해야 하는 명령어의 주소를 가리키는 역할을 수행한다. clr(=reset) 버튼이 눌릴 경우 처음부터 명령어를 다시 실행해야 하므로 counter(=PC) 값을 0으로 초기화하였다. 일반적인 경우 n 번째 명령어를 실행한 다음에는 n+1 번째 명령어를 실행해야하므로 counter 값을 1씩 증가시키지만 branch 값이 1인 경우, 즉, jump를 수행해야하는 경우에는 현재의 counter 값에 add + 1만큼 더해주어야 한다.

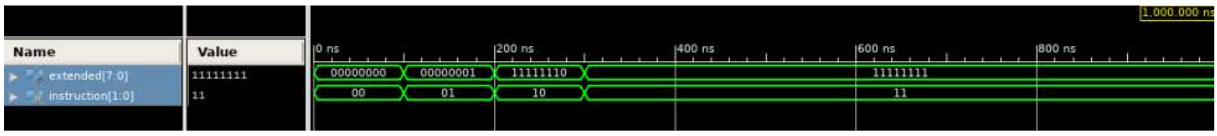


PC의 시뮬레이션 결과는 위의 이미지와 같다. branch가 0일 때는 카운터가 1씩 증가하지만, branch가 1일 때는 add + 1 값만큼 카운터가 증가하는 모습을 확인가능하다.

(2) SignExtend

```
module Signextend(
    input [1:0] instruction,
    output [7:0] extended
    );
            //00 0 8'b00000000
            //01 1 8'b00000001
            //10 -2 8'b11111110
            //11 -1 8'b11111111

            // imm value is 2 bit long, so we extended it to 8 bits
            assign extended = (instruction == 2'b00) ? 8'b00000000:
                                                                    (instruction == 2'b01) ? 8'b00000001:
                                                                    (instruction == 2'b10) ? 8'b11111110:
                                                                      8'b11111111;
endmodule
```

PC의 input인 add 값은 imm을 8비트로 늘린 값을 의미한다. imm은 instruction[1:0]에 해당하는 값으로 부호를 가지는 값이다. 즉, instrution[1:0]이 2'b00이면 8비트로 표현된 0을 output으로 반환하는 식이다. 따라서 이를 주석과 같이 instruction[1:0]이 2'b00일 때, 2'b01일 때, 2'b10일 때, 2'b11일 때 각각에 해당하는 8비트 값을 정리하였고 이를 output으로 반환하는 모듈을 작성하였다.

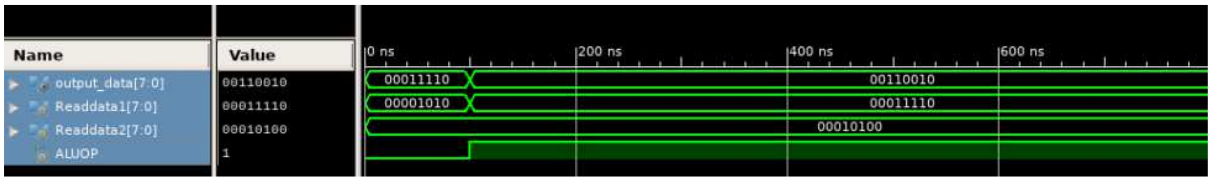


Signextend 모듈의 시뮬레이션 결과는 위의 이미지와 같다. 2비트의 signed 값을 8비트로 정상적으로 연장시켜주는 것을 확인 가능하다.

(3) ALU

```
module ALU(
    input [7:0] Readdata1,
    input [7:0] Readdata2,
            input ALUOP,
    output [7:0] output_data
    );

            assign output_data = Readdata1 + Readdata2;
endmodule
```

PPT에서는 ALUOP 신호가 1일 때만 output_data를 Readdata1과 Readdata2를 더한 값으로 지정하고 그렇지 않을 경우 어떠한 변화도 일어나지 않는다고 되어있으나 실제로는 ALUOP 신호가 0일 때도 Readdata1과 Readdata2를 더한 값을 output으로 반환해야 한다. 즉, PPT에서는 add를 수행할 때만 Readdata1과 Readdata2의 값을 더하도록 되어있으나 lw와 sw를 수행할 때에도 두 값을 더해주어야 한다. 예를 들면, $s0 = Mem[$s3+0]을 수행할 때, Readdata1에는 레지스터 3에 저장된 값이, Readdata2는 8비트로 늘어난 imm 값(=0)이 저장된다. Data Memory에서는 $s3+0에 해당하는 값을 input인 Address로 받아야 하기 때문에 ALU에서는 lw일 때에도 역시 두 값을 더해주어야 한다. 마찬가지로 sw를 수행할 때 역시 두 값을 더해주어야 한다.



ALU의 시뮬레이션 결과는 위의 이미지와 같다. ALUOP 값과 상관없이 output인 output_data의 값이 Readdata1 + Readdata2의 값으로 정상적으로 지정되는 모습을 확인 가능하다.

(4) Control Unit

```
module Control(
    input [1:0] instruction,
            output RegDst,
    output RegWrite,
    output ALUSrc,
    output Branch,
    output MemRead,
    output MemWrite,
    output MemtoReg,
    output ALUOP
    );

            wire op1 = instruction[1];
            wire op0 = instruction[0];

            //Used K-map to minimize the equations (Ref: Microprocessor Design - ISA)
            assign RegDst = ~op1 & ~op0;
            assign RegWrite = ~op1;
            assign ALUSrc = (~op1 & op0)|(op1 & ~op0);
            assign Branch = op1 & op0;
            assign MemRead = ~op1 & op0;
            assign MemWrite =op1 & ~op0;
            assign MemtoReg = op0;
            assign ALUOP = ~op1 & ~op0;

endmodule
```

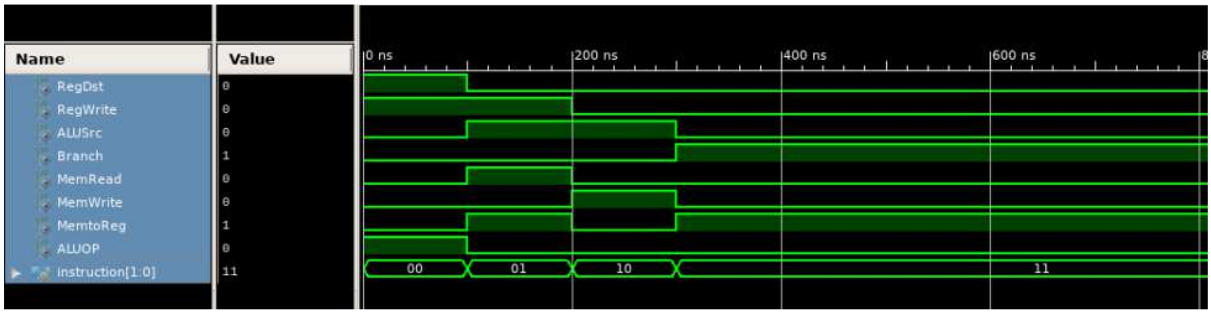
Instruction	RegDst	RegWrite	ALUSrc	Branch	MemRead	MemWrite	MemtoReg	ALUOP
R-format	1	1	0	0	0	0	0	1
lw	0	1	1	0	1	0	1	0
sw	x	0	1	0	0	1	x	0
j	x	0	0	1	0	0	x	0

Final Project PPT 8 페이지에 있는 표를 참고하여 각 신호별 boolean 식을 K-map 방식을 통해 minimize하였다. 예를 들어, RegDst 신호에 대해 K-map을 그리면 아래와 같다. 이때, 편의상 instruction[7]를 i7, instruction[6]을 i6로 표시하였다.

i7₩i6	**0**	**1**
0	1	0
1	x	x

따라서 RegDSt = (instruction[6])'로 표기 가능하다. (한편, 두 개의 operation code 만을 고려하면 되기 때문에 실제로는 위와 같이 K-map을 그리지 않고 머릿속으로 계산하였는데 이로 인하여 Discussion – (2)에서 제시한 것과 같은 실수가 발생하였다. 또한, 방금 보인 RegDst의 K-map을 살펴보면 RegDst = ~op0로도 표현가능하나 암산 과정에서 실제 코드를 작성할 때는 Don't care 경우를 미처 고려하지 못해 RegDst = ~op0 & ~op1로 표시하였다. MemtoReg 역시 Don't care 경우를 미처 고려하지 못해 처음에는 MemtoReg = ~op1 & op0와 같이 표시하였다. 검토를 하면서

MemtoReg만을 MemtoReg = op0로 지정해주었으나, RegDSt 역시 RegDst = ~op0로 표기하는 것이 옳다. 그러나 Don't care 조건이기 때문에 어떤 방식으로 지정하든 성능에는 영향을 미치지 않는다. 보고서 작성 과정에서 해당 실수를 발견하였고, 조원들에게 해당 내용을 공유하였다.)

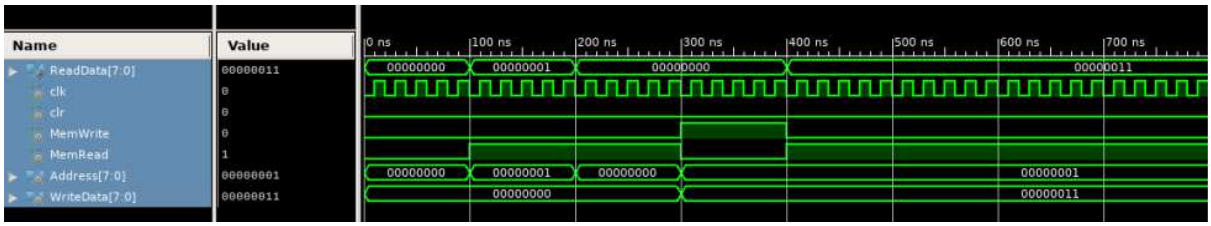


Control Unit의 시뮬레이션 결과는 위의 이미지와 같다. Operation code에 해당하는 instruction[7:6]이 2'b00, 2'b01, 2'b10, 2'b11일 때 각각 RegDst, RegWrite, ALUSrc, Branch, MemRead, MemWrite, MemtoReg, ALUOP 신호가 올바르게 설정되는 것을 확인 가능하다.

(5) Data Memory

```
module data_memory(
    input clk,
    input clr,
    input MemWrite,
    input MemRead,
    input [7:0] Address,
    input [7:0] WriteData,
    output [7:0] ReadData
    );
            integer i;

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

            //output
            assign ReadData = MemRead ? mem[Address] : 0;

            //Initialize the Data Memory
            initial begin
                        for(i = 0; i < 16; i= i+ 1)
                                    begin
                                                mem[i] = i;
                                                mem[i + 16] = -i;
                                    end
            end

            always @(posedge clk or posedge clr) begin
                        if (clr) begin
                                                //Initialize the Data Memory
                                                for(i = 0; i < 16; i= i+ 1)
                                                            begin
                                                                        mem[i] = i;
                                                                        mem[i + 16] = -i;
                                                            end
                        end
                        else begin
                                    //Store the value of 'WriteData' to memory
                                    if (MemWrite) begin
                                                                        mem[Address] = WriteData;
                                                            end
                                    end

                        end
endmodule
```

우선, 처음과 clr(=reset) 버튼이 눌릴 경우 32개의 Data Memory를 for 문을 사용하여 PPT에서 제시된 바와 같이 초기화해주었다. Data Memory는 주소인 Address를 input으로 받는데, MemWrite 신호가 true일 경우에는 mem[Address]에 WriteData 값을 저장하고, MemRead 신호가 true일 경우에는 output인 ReadData에 mem[Address] 값을 저장하였다. if (MemWrite) 문에 이어서 if (MemRead) 문을 작성하고자 하였으나 이상하게도 계속해서 오류가 발생하였고, 이를 always 문 바깥으로 빼주니 해당 오류가 사라졌다. (이 오류는 wire에 register 값을 저장하려고 시도하여 발생하는 오류이다.) 아래는 data memory에 대한 시뮬레이션 결과이다.



(6) registers

```
module registers(
        input clr,
        input clk,
        input RegWrite,
        input [1:0] rs,
        input [1:0] rt,
        input [1:0] writeRegister,
        input [7:0] writeData,
        output [7:0] output_data1,
        output [7:0] output_data2
        );

        //4 registers
        reg[7:0] data1;
        reg[7:0] data2;
        reg[7:0] data3;
        reg[7:0] data4;
```

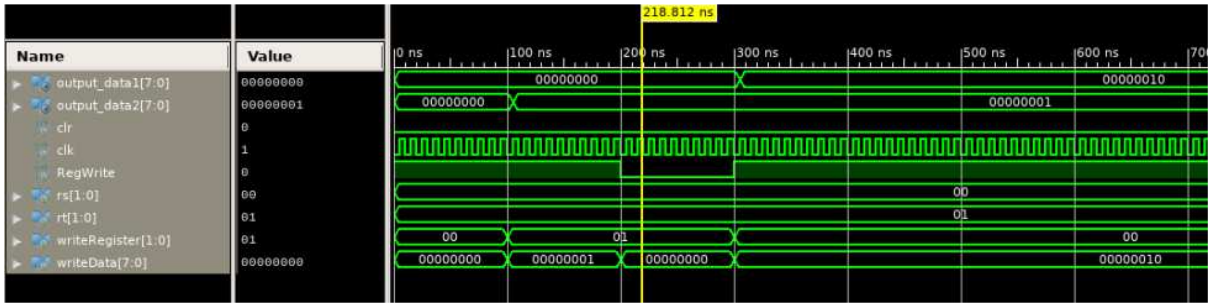
우선, register 모듈의 input과 output을 살펴보면 위의 이미지와 같다. 또한, 모듈 내부에 data1, data2, data3, data4라는 4개의 레지스터가 존재한다.

```
//output data determined by rs and rt
assign output_data1 = (rs == 2'b00) ? data1:
                      (rs == 2'b01) ? data2:
                      (rs == 2'b10) ? data3:
                      data4;

assign output_data2 = (rt == 2'b00) ? data1:
                      (rt == 2'b01) ? data2:
                      (rt == 2'b10)? data3:
                      data4;

initial begin
        data1 = 8'b00000000;
        data2 = 8'b00000000;
        data3 = 8'b00000000;
        data4 = 8'b00000000;
end
```

위의 코드를 통해 프로그램 실행 시 data1, data2, data3, data4라는 4개의 레지스터를 모두 0으로 초기화하였다. 다음으로, rs와 rt 값에 따라 각각 output_data1과 output_data2의 값을 설정하였다. 예를 들어, rs, rt의 값이 각각 2'b00, 2'b01이면 output_data1에는 data1을 output_data2에는 data2를 저장해야 한다.

```
always @(posedge clk or posedge clr)
        begin
                if (clr)
                        begin
                                //Initialize data
                                data1 = 8'b00000000;
                                data2 = 8'b00000000;
                                data3 = 8'b00000000;
                                data4 = 8'b00000000;
                        end
                else if (RegWrite) begin
                        case (writeRegister)
                                //Write data to the register corresponding to the value of WriteRegister
                                2'b00: data1 = writeData;
                                2'b01: data2 = writeData;
                                2'b10: data3 = writeData;
                                2'b11: data4 = writeData;
                        endcase
                end
        end
endmodule
```

또한, clr (=reset) 버튼이 눌렸을 때에도 모든 레지스터의 값을 0으로 초기화 해주어야 한다. 또한, RegWrite 신호가 True일 경우 writeRegister의 값에 따라 적절한 레지스터에 writeData를 저장해주었다. 아래는 register 모듈에 대한 시뮬레이션 결과이다.



(7) freq_divider

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

            reg[31:0] cnt;
            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
```

SNU Logic Design Board의 pin 57은 5MHz의 clock 역할을 수행한다. 이 clock의 파동이 총 5천만번 oscillate하면 1초가 흐른다. 따라서 2천 5백만 번 oscillate 했을 때 clkout을 0에서 1로 증가시키고 또 다시 2천 5백만 번 oscillate 했을 때 clkout을 1에서 0으로 감소시킨다면 clkout은 1Hz 클럭의 역할을 수행한다. 해당 코드는 Lab7 PPT에 제시된 freq_divider 모듈의 코드를 그대로 사용한 것이다.

(8) bcd_to_7

Lab7 PPT에 제시된 bcd_to_7 모듈의 코드를 확장하여 0에서 15까지의 정수를 16진수로 표현하는 bcd_to_7 모듈을 작성하였다. input인 bcd는 4비트의 이진수(0~15)인데, 이를 7 segment display로 나타낼 수 있도록 output인 seg의 값을 정해주는 모듈이다.

```
module bcd_to_7(
    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;
                            4'd11: seg <= 7'b1111100;
                            4'd12: seg <= 7'b0111001;
                            4'd13: seg <= 7'b1011110;
                            4'd14: seg <= 7'b1111001;
                            4'd15: seg <= 7'b1110001;
                    endcase
            end

endmodule
```

(9) Microprocessor

```
assign value = MemtoReg ? data_out : alu_res;
assign display = MemWrite ? read_data2 : value;

//signed extended signal
wire [7:0] extended;

//slow clock (1Hz)
freq_divider T0(.clr(clr), .clk(clk_in), .clkout(clk));

//Program Counter
PC T1(.branch(Branch),.add(extended), .clk(clk), .clr(clr), .counter(counter));

//Module for check
//IMEM T2(.Read_Address(counter-1), .instruction(instruction));

//Control Unit
Control T3(.instruction(instruction[7:6]), .RegDst(RegDst), .RegWrite(RegWrite), .ALUSrc(ALUSrc), .Branch(Branch),.MemRead(MemRead), .MemWrite(MemWrite), .MemtoReg(MemtoReg), .ALUOP(ALUOP));

//Registers
registers T4(.clr(clr), .clk(clk), .RegWrite(RegWrite), .rs(instruction[5:4]), .rt(instruction[3:2]), .writeRegister(RegDst?instruction[1:0]:instruction[3:2]), .writeData(value), .output_data1(read_data1), .output_data2(read_data2));

//7 segments
bcd_to_7 T6(.bcd(display[7:4]), .seg(seg1));
bcd_to_7 T7(.bcd(display[3:0]), .seg(seg2));
bcd_to_7 T00(.bcd(counter[7:4]), .seg(seg3));
bcd_to_7 T01(.bcd(counter[3:0]), .seg(seg4));
bcd_to_7 T02(.bcd(alu_res[7:4]), .seg(seg5));
bcd_to_7 T03(.bcd(alu_res[3:0]), .seg(seg6));

//Sign Extend
Signextend T8(.instruction(instruction[1:0]), .extended(extended));

//ALU
ALU T10(.Readdata1(read_data1), .Readdata2(ALUSrc?extended:read_data2), .ALUOP(ALUOP), .output_data(alu_res));

//Data Memory
data_memory T11(.clk(clk), .clr(clr), .MemWrite(MemWrite), .MemRead(MemRead), .Address(alu_res), .WriteData(read_data2), .ReadData(data_out));

endmodule
```

마지막으로 PPT에서 제시된 바와 같이 각 모듈들을 하나로 합쳐주었다. 이때, 랩 시간에 제공된 동영상과 같이 1, 2번째 display에는 Reg Write Data를, 3, 4번째 display에는 counter를 표시하였다. 마지막으로, 동영상을 통해서는 5, 6번째 display에 어떤 데이터가 표시되고 있는지 알기가 어려웠고, 임의로 alu의 output 값을 나타내도록 하였다.

2. Discussion

(1) Concepts that I learned during the lab session

팀원 중 한 명이 data memory를 구현하였는데, data memory를 초기화하는 과정에서 for 문을 사용하였다. integer i를 index 로 사용하여 for 문 내에서 data memory를 초기화한 것을 보면서 새로운 문법을 알게 되었다. for 문 문법은 C언어와 동일하지만 C언어의 int 자료형이 베릴로그에서 integer 자료형에 대응된다는 것을 알게 되었다.

이번 파이널 프로젝트를 하기 전에도 PC 등 CPU를 구성하는 모듈들의 역할에 대해서는 알고 있었는데, 이번 프로젝트를 하면서 어떻게 개별 모듈들이 상호 연결되어 프로세서를 구성하는지 배웠다. 본 프로젝트에서 구현해야하는 ALU는 ALUOP 값과 관계없이 무조건 두 개의 input을 더하여 output으로 반환하면 됐는데, 실제 컴퓨터에서 사용되는 복잡한 프로세서에는 몇 개의 control input이 사용되는지 연산 종류에는 어떤 것이 있는지 궁금하여 찾아보기도 하였다.

printf 로 출력이 되는 C 언어 등 일반적인 프로그래밍 언어와는 다르게 값 출력이 불가능하고 오직 시뮬레이션을 통해서만 모듈이 올바르게 동작하는지 알 수 있었기 때문에 베릴로그 언어의 디버깅이 굉장히 까다롭게 느껴졌다. 본 프로젝트를 수행하면서 디버깅을 효율적으로 하는 법을

배웠다. SNU Logic Design Board에는 총 6개의 7 segment display가 있기 때문에 LAB 7 PPT에 제시되어 있는 bcd_to_7 모듈을 사용하여 확인하고자 하는 값들을 display에 나타내 확인하는 방식을 사용하는 것이 가장 효율적이었다. 처음에 각각의 모듈을 모두 구현하고 이를 Microprocessor 모듈 내에서 연결하여 SNU Logic Design Board을 통해 결과를 확인하였으나 Reg Write Data의 값이 00에서 움직이지 않았고 원인을 파악하고자 각종 Signal 들과 ALU와 같은 개별 모듈들의 output을 출력해보면서 수정해나갔다. 마침내 올바르게 동작하는 Microprocessor를 구현할 수 있었다.

data memory 모듈에서 clr(=reset) 버튼이 눌렸을 때 외에도 처음 실행이 될 때 PPT에 주어진 바와 같이 초기화가 되어야 했는데 이러한 상황에서 initial begin – end 문법이 사용된다는 것을 배웠다. initial begin – end 문 내에서 for 문을 사용하여 data memory를 초기화해주면 처음 실행이 될 때도 정상적으로 초기화가 가능하다.

(2) Any errors that our team made and How to correct our errors

Instruction	RegDst	RegWrite	ALUSrc	Branch	MemRead	MemWrite	MemtoReg	ALUOP
R-format	1	1	0	0	0	0	0	1
lw	0	1	1	0	1	0	1	0
sw	x	0	1	0	0	1	x	0
j	x	0	0	1	0	0	x	0

Control Unit을 구현할 때 PPT에 주어진 위의 표를 참고하여 RegDst, RegWrite 등에 대한 boolean 식을 K-map 방법을 적용하여 minimize하고자 하였다. 이때, operation code에 해당하는 instruction[7]과 instruction[6]만을 고려하면 되었기 때문에 K-map을 직접 종이에 그리지 않고 머릿속으로 계산하였다. 이로 인해 ALUSrc를 (~op0 & op1) | (op0 & ~op1)이 아닌 (~op0 & op1) & (op0 & ~op1)로 잘못 assign 하는 실수를 했다. 이러한 실수로 인해 ALUSrc 값이 항상 false가 되어 Microprocessor가 올바르게 동작하지 못했다. 처음에 디버깅을 할 때 해당 문제를 인지하지 못했고, SNU Logic Design Board의 display가 00에서 아예 바뀌지 않아 다른 모듈에서 그 원인을 찾고자 하였다.

display 값이 바뀌지 않는 원인을 찾는 과정에서 Data memory 모듈에서 clr (=reset) 버튼을 눌렀을 때 외에도 처음 실행과 동시에 data memory가 PPT 에 지정된 바와 같이 초기화되어야 하지만 이것이 구현되지 않았음을 깨달았다. 구글링을 통해 initial begin – end 내에 있는 코드가 프로그램 실행 시작과 동시에 실행된다는 것을 알게 되었고 이를 이용하여 정상적으로 초기화할 수 있었다.

그러나 여전히 정답은 출력되지 않았고, ALU 모듈의 작동 방식에 대해 생각을 해보던 중, ALUOP 값이 0일 때에도 두 개의 input이 더해져 output으로 반환되어야 함을 알게 되었다. PPT에서는 ALUOP 가 1일 때만 두 개의 input을 더해 output으로 반환하라고 명시되어 있는데, add를 수행할 때(=operation code가 2'b00일 때)만 ALUOP가 1이다. 그러나 lw와 sw 연산을 수행할 때에도 두 개의 input(=한 개의 레지스터 값과 한 개의 sign extended된 imm 값)을 더해 output

으로 반환해야 한다. 따라서 ALUOP 신호를 무시하고 무조건 두 개의 input을 더하여 output으로 반환하도록 수정하였다.

여러 모듈들을 수정하였음에도 불구하고 SNU Logic Design Board 에서 Reg Write Data 값이 바뀌지 않았다. TA 세션에 참가했을 때, 다른 조가 RegWrite가 동작하지 않는다고 하였고, 혹시 우리조도 RegWrite가 동작을 안해서 발생하는 문제인가 의심하게 되었다. 4비트로 구성된 wire 하나를 정의하고 모든 비트를 RegWrite로 초기화하여 display에 나타냈으나 올바르게 지정되는 것을 확인하였다. 다른 모든 Signal들도 각각 display에 나타내어 확인하는 과정에서 ALUSrc의 값이 아예 바뀌지 않는 것을 알게 되었다. 위에서 언급했던 것과 같이 잘못 지정되어 있는 것을 확인하여 이를 수정하였다. 검토하는 과정에서 MemtoReg의 boolean 식을 Don't care 조건을 활용하면 더 minimize 할 수 있다는 것을 깨달았고 결과에는 영향을 미치지 않지만 이를 반영하여 같이 수정해주었다.

디버깅을 통해 수정을 거치고 나서 SNU Logic Design Board에 연결했을 때 이번에는 Program Counter 값이 바뀌지만 clr(=reset) 버튼을 눌렀을 때 이상하게도 Program Counter가 0으로 초기화가 되는 것이 아니라 값이 멈췄다. 이전 랩 실습 때 00 – 99 counter를 구현하는 과정에서 if 문을 여러 개 이어서 사용하는 바람에 비슷한 문제를 겪었는데, 이번에도 if 문이 문제일 것이라고 예상하여 'counter = counter + 1; if (branch == 1) conter = counter + add'를 구역이 명확히 나누어지도록 'if (branch == 1) counter = counter + add + 1; else counter = counter + 1'과 같이 바꾸었으나 여전히 동일한 문제가 반복되었다. 혹시 현재의 instruction 주소와 다음에 수행해야 하는 instruction 주소를 따로 정의하여 관리해야 하는지 등 다양한 새로운 구현 방법을 고민하던 중 Microprocessor 모듈 내부에서 PC에 clr 를 전달하지 않았다는 것을 발견하였다. 이를 수정하고 나니 올바르게 동작하는 PC를 얻을 수 있었다.

Microprocessor가 랩 시간에 제공된 동영상에 있는 테스트 케이스를 통과하는 것을 확인하고 마지막으로 시뮬레이션을 실행해보고자 하였다. 그러던 중 data memory의 테스트 벤치 코드 이름을 기존에 있던 다른 파일의 이름과 동일하게 해버렸고 이로 인해 더 이상 기존 프로젝트 파일에서 data memory가 뜨지 않았다. 새로 파일을 만들었음에도 해당 오류가 계속 되었고 결국 새로운 프로젝트를 생성하여 모든 파일을 옮김으로써 문제를 해결하였다.