

명지 공학인 윤리 서약서

보고서 및 논문 윤리 서약

1. 나는 보고서 및 논문의 내용을 조작하지 않겠습니다.
2. 나는 다른 사람의 보고서 및 논문의 내용을 내 것처럼 무단으로 복사하지 않겠습니다.
3. 나는 다른 사람의 보고서 및 논문의 내용을 참고하거나 인용할 시 참고 및 인용 형식을 갖추고 출처를 반드시 밝히겠습니다.
4. 나는 보고서 및 논문을 대신하여 작성하도록 청탁하지도 청탁받지도 않겠습니다.
나는 보고서 및 논문 작성 시 위법 행위를 하지 않고, 명지인으로서 또한 공학인으로서 나의 양심과 명예를 지킬 것을 약속합니다.

시험 윤리 서약

1. 나는 대리시험을 청탁하거나 청탁받지 않겠습니다.
2. 나는 허용되지 않은 교과서, 노트 및 타학생의 답안지 등을 보고 답안지를 작성하지 않겠습니다.
3. 나는 타인에게 답안지를 보여주지 않겠습니다.
4. 나는 감독관의 지시와 명령에 따라 시험 과정에 참여하겠습니다.
나는 시험에 위법 행위를 하지 않고, 명지인으로서 또한 공학인으로서 나의 양심과 명예를 지킬 것을 약속합니다.

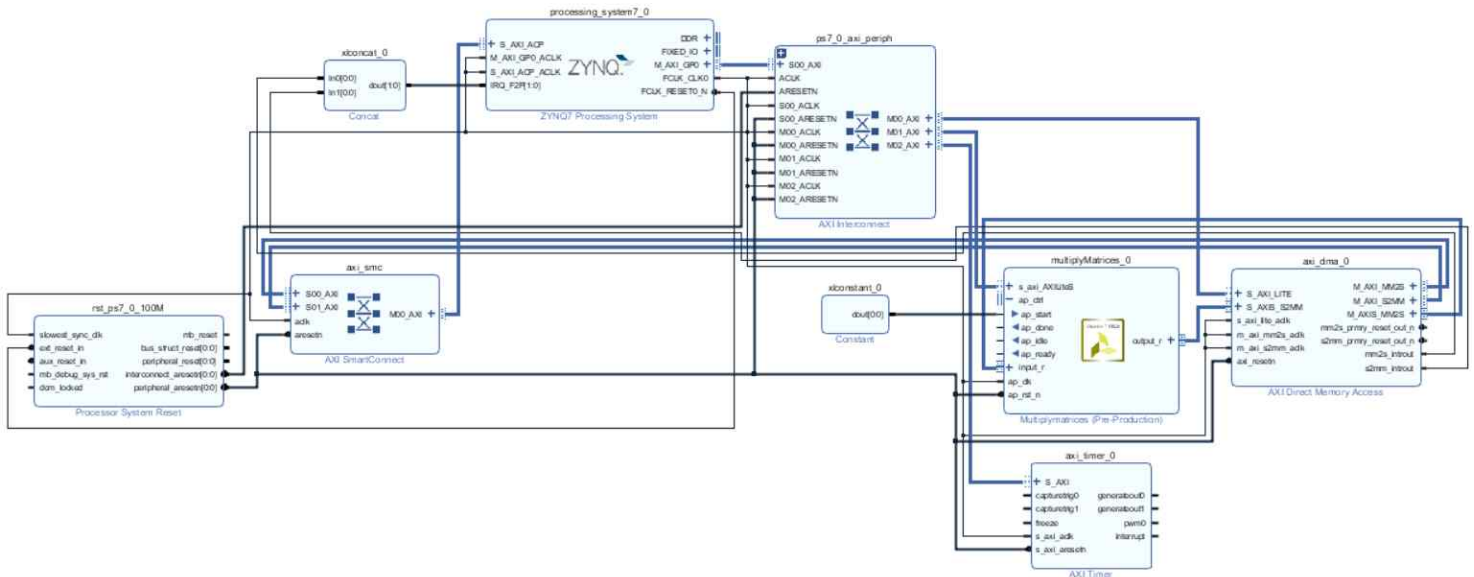
2023년 06월 16일

서약자

(학번) 60171878

(성명) 허무혁 (인)

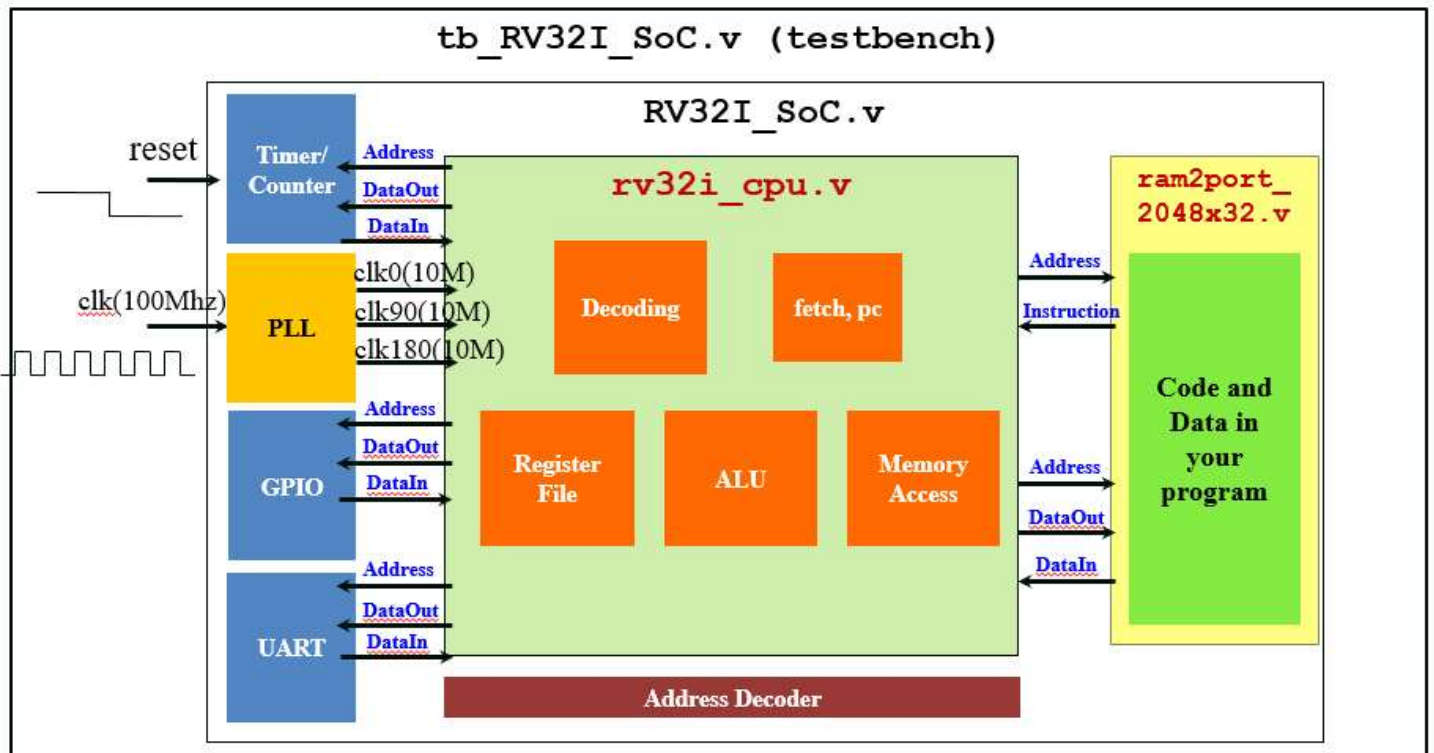
60171878 허무혁



목차

1. 설계 방법
2. Source Code
3. Discussion
4. 결론

1. 설계 방법



Riscv-soc 내부에는 pll이 있다. RISC-V를 싱글 사이클로 설계하기 때문에, 3개의 클락을 사용하며 clk0은 main clock, clk 90은 instruction 클락, clk180은 data 클락으로 사용한다.

The phase is calculated relative to the active input clock.

Output Clock	Port Name	Output Freq (MHz)		Phase (degrees)			Duty Cycle
		Requested	Actual	Requested	Actual	Requested	
<input checked="" type="checkbox"/> clk_out1	clk0	10	10.000	0.000	0.000	50.000	50.000
<input checked="" type="checkbox"/> clk_out2	clk90	10	10.000	90	0.000	50.000	50.000
<input checked="" type="checkbox"/> clk_out3	clk180	10	10.000	180	0.000	50.000	50.000
<input type="checkbox"/> clk_out4	clk_out4	100.000	N/A	0.000	N/A	50.000	50.000
<input type="checkbox"/> clk_out5	clk_out5	100.000	N/A	0.000	N/A	50.000	50.000
<input type="checkbox"/> clk_out6	clk_out6	100.000	N/A	0.000	N/A	50.000	50.000
<input type="checkbox"/> clk_out7	clk_out7	100.000	N/A	0.000	N/A	50.000	50.000

Riscv-soc 내부에는 RAM이 있다. 메모리 모델은 block memory를 사용하고 사이즈는 2048x32이다.

Component Name ram_2port_2048x32

Basic	Port A Options	Port B Options	Other Options	Summary
Interface Type	Native		<input type="checkbox"/> Generate address interface with 32 bits	
Memory Type	True Dual Port RAM		<input type="checkbox"/> Common Clock	

Riscv-soc하위에는 Address Decoder가 있다. 상위 하이러키로부터 32bit 주소를 입력받아 주소가 위치한 곳이 GPIO인지, DATA 영역인지, 주변장치 영역인지 등을 알아내어 출력으로 내보낸다.

```
module Addr_Decoder(
    input [31:0] addr,
    output reg cs_mem,
    output reg cs_gpio
);
```

```
// 0xFFFF_FFFF -----
//
//          Reserved
//
// 0xFFFF_3000 -----
//          GPIO          4kB
// 0xFFFF_2000 -----
//          Reserved for Peripheral-2 4kB
// 0xFFFF_1000 -----
//          Reserved for Peripheral-1 4kB
// 0xFFFF_0000 -----
//
//          Reserved
//
// 0x0000_2000 -----
//
//          instruction & data 8kB
//
// 0x0000_0000 -----
```

Riscv-soc 내부에는 GPIO가 있다. GPIO에서는 Chip select 신호와, 12비트 주소, 32비트 데이터, Write Enable 신호와, Read Enable 신호를 입력받아 출력으로 32비트 데이터와 7-Segment, LED를 내보내준다.

```
module GPIO(
    input clk,
    input rst,
    input CS,
    input REN,
    input WEN,
    input [11:0] Addr,
    input [31:0] DataIn,
    output reg [31:0] DataOut,
    output reg [6:0] HEX0,
    output reg [6:0] HEX1,
    output reg [6:0] HEX2,
    output reg [6:0] HEX3,
    output reg [6:0] HEX4,
    output reg [6:0] HEX5,
    output reg [7:0] LEDS
);
```

Risc-v cpu 에서는 instruction을 해석하여 op코드에 해당하는 control 신호와 datapath를 설정해주는 역할을 한다.

```

1  `define OP_LUI      7'b0110111
2  `define OP_AUIPC   7'b0010111
3  `define OP_JAL     7'b1101111
4  `define OP_JALR    7'b1100111
5  `define OP_B       7'b1100011
6  `define OP_L_LOAD  7'b0000011
7  `define OP_S       7'b0100011
8  `define OP_L_ARITH 7'b0010011
9  `define OP_R       7'b0110011
10
11
12 module rv32i_cpu(
13     input  clk,
14     input  rst,
15     output reg [31:0] pc, //program counter (address for instruction)
16     input  [31:0] inst, //instruction from memory
17     output MemWen,
18     output [31:0] MemAddr,
19     output [31:0] MemWdata,
20     input  [31:0] MemRdata
21 );

```

Risc-v cpu의 내부에는 register file이 있다. 32비트 폭에 32의 높이를 가진다. Address는 cpu로부터 받아오며, Data는 양방향으로 소통한다.

```

module regfile(
    input  clk,
    input  we,
    input  [4:0] rs1, rs2, rd,
    input  [31:0] rd_data,
    output reg [31:0] rs1_data, rs2_data);

```

Risc-v cpu의 내부에는 ALU가 있으며 CPU로부터 5비트 control 신호와 alusrc1, alusrc2를 입력받아 연산결과값과 N, Z, C, V 플래그를 출력으로 내보낸다.

```

module alu(
    input [31:0] a, b,
    input [4:0] control,
    output reg [31:0] result,
    output N,
    output Z,
    output C,
    output V
);

```

ALU 내부에는 Ripple Carry Adder를 가지게 설계하였다.

```

module RippleAdder (
    input[31:0] a,
    input[31:0] b,
    input ci,
    output[31:0] s,
    output co,
    output last_ci
);
    wire [32:0] c;

    assign c[0] = ci;
    assign co = c[32];
    assign last_ci = c[31];

```

```

    FullAdder fa0(a[0], b[0], c[0], s[0], c[1]);
    FullAdder fa1(a[1], b[1], c[1], s[1], c[2]);

```

Ripple Carry Adder 에는 32개의 Full Adder를 사용한다.

실습에 사용할 coe 파일의 생성을 위해 리눅스 환경에서 cross compiler를 사용해 risc-v용 기계어 코드로 컴파일 하는 과정을 거쳤다.

사용한 Risc-v 명령어의 해석은 dump 파일을 보며 하였으며, pc값이 증가했을 때 어떤 명령어를 만나 연산을하고, jump를 할 때 어느 레이블로 점프를 하는지 파일을 보고 알 수 있다.

insts_data.coe labcode.dump

파일 편집 보기

```
00000000 <SevenSeg-0x10>:
.text
.align 4

    la sp, stack
0:  40000113          li    sp,1024
    j      SevenSeg
4:  00c0006f          j      10 <SevenSeg>
    ...

00000010 <SevenSeg>:
.text
.align 2
.globl SevenSeg
.type SevenSeg, @function
SevenSeg:
    addi    sp,sp,-32
10:  fe010113          add    sp,sp,-32
    sw      s0,28(sp)
14:  00812e23          sw      s0,28(sp)
    addi    s0,sp,32
18:  02010413          add    s0,sp,32
    li      a5,-57344
1c:  ffff27b7          lui     a5,0xfffff2
    addi    a5,a5,12
20:  00c78793          add    a5,a5,12 # ffff200c <stack+0xfffff1c0c>
    sw      a5,-20(s0)
24:  fef42623          sw      a5,-20(s0)
    lw      a5,-20(s0)
28:  fec42783          lw      a5,-20(s0)
    li      a4,91
2c:  05b00713          li      a4,91
    sw      a4,0(a5)
30:  00e7a023          sw      a4,0(a5)
.L2:
    nop
```

위 는 milestone 1의 dump 파일이며 첫 번째 명령은 sevenseg 레이블로 jump를 하라는 명령이다. 이후 jump 이후 add, sw, add, lui, add 등의 명령들을 만나는데, 해당 코드가 어떤 어셈블리 코드를 수행하는 지 파악한 후 Verilog 코드를 작성하여 어셈블리 코드에 해당하는 내용을 추가하는 과정을 거쳐주었다.

insts_data.coe labcode.dump

파일 편집 보기

```
memory_initialization_radix=16;
memory_initialization_vector=
40000113,
00C0006F,
00000000,
00000000,
FE010113,
00812E23,
02010413,
FFFF27B7,
00C78793,
FEF42623,
FEC42783,
05B00713,
00E7A023,
00000013,
FFDF006F,
00000000,
00000000,
00000000,
00000000,
00000000,
```


위는 milestone 1의 coe파일이며, 메모리에 적재할 명령어들을 Hex값으로 보여준다. 위 coe 파일과 dump 파일을 활용하고 Vivado의 Simulation 기능을 활용하여, PC값의 증가와 메모리의 명령들을 잘 수행하는지, GPIO와 ALU연산이 적절하게 수행되는지를 검증하여 설계하였다.



2. 소스 코드 (핵심 부분만)

스켈레톤 코드에서 추가로 작성한 코드만 첨부합니다.

module rv32i_cpu

```
assign beq = {funct3 == 3'b000};
assign bne = {funct3 == 3'b001};
assign bit = {funct3 == 3'b100};
assign bge = {funct3 == 3'b101};
assign bitu = {funct3 == 3'b110};
assign bgeu = {funct3 == 3'b111};
```

```
assign beq_flag = branch & beq & Zflag;
assign bgeu_flag = branch & bgeu & Cflag;
```

```
// register for pc
```

```
always @ (posedge clk, posedge rst)
begin
    if (rst)
        pc <= 0;
    else if (jalr)
        pc <= {aluout[31:1], 1'b0};
    else if (jal)
        pc <= pc + {{11{inst[31]}}, {inst[31], inst[19:12], inst[20], inst[30:21], 1'b0}};
    else if (bgeu_flag || beq_flag)
        pc <= pc + {{19{inst[31]}}, {inst[31], inst[7], inst[30:25], inst[11:8], 1'b0}};
    else
        pc <= pc+4;
end
```

branch 오퍼레이션 코드와 pc값 컨트롤 부분.

```

case (opcode)
  `OP_R: //R-type
    case ({funct7,funct3})
      10'b0000000_000: alucontrol = `ALU_ADD;
      10'b0100000_000: alucontrol = `ALU_SUB;
      10'b0000000_001: alucontrol = `ALU_SLL;
      10'b0000000_010: alucontrol = `ALU_SLT;
      10'b0000000_011: alucontrol = `ALU_SLTU;
      10'b0000000_100: alucontrol = `ALU_XOR;
      10'b0000000_101: alucontrol = `ALU_SRL;
      10'b0100000_101: alucontrol = `ALU_SRA;
      10'b0000000_110: alucontrol = `ALU_OR;
      10'b0000000_111: alucontrol = `ALU_AND;
      default: alucontrol = `ALU_ADD;
    endcase
  `OP_I_ARITH: //I-Type Arithmetic
    case (funct3)
      3'b000 : alucontrol = `ALU_ADD; //ADDI
      3'b010 : alucontrol = `ALU_SUB; //SLTI
      3'b011 : alucontrol = `ALU_SUB; //SLTIU
      3'b100 : alucontrol = `ALU_XOR; //XORI
      3'b110 : alucontrol = `ALU_OR; //ORI
      3'b111 : alucontrol = `ALU_AND; //ANDI
      default: alucontrol = `ALU_ADD;
    endcase
  `OP_LUI : alucontrol = `ALU_ADD;
  `OP_JAL : alucontrol = `ALU_ADD;
  `OP_JALR : alucontrol = `ALU_ADD;
  `OP_I_LOAD : alucontrol = `ALU_ADD;
  `OP_AUIPC : alucontrol = `ALU_ADD;
  `OP_S : alucontrol = `ALU_ADD;
  `OP_B : alucontrol = `ALU_SUB;

```

op코드를 해석하여 alu 연산 수행을 지정해주는 부분. 가독성 향상을 위해 define을 이용해 alu 명령 비트를 작성해 주었다


```

case (opcode)
  `OP_R: begin //R-type
    alusrc = 1'b0;
    regwrite = 1'b1;
    lui = 1'b0;
    memwrite = 1'b0;
    jal = 1'b0;
    jalr = 1'b0;
    mem2reg = 1'b0;
    branch = 1'b0;
  end
  `OP_I_ARITH: begin //I-type
    alusrc = 1'b1;
    regwrite = 1'b1;
    lui = 1'b0;
    memwrite = 1'b0;
    jal = 1'b0;
    jalr = 1'b0;
    mem2reg = 1'b0;
    branch = 1'b0;
  end
  `OP_LUI: begin
    alusrc = 1'b0;
    regwrite = 1'b1;
    lui = 1'b1;
    memwrite = 1'b0;
    jal = 1'b0;
    jalr = 1'b0;
    mem2reg = 1'b0;
    branch = 1'b0;
  end
  `OP_S: begin
    alusrc = 1'b0;
    regwrite = 1'b0;
    lui = 1'b0;
    memwrite = 1'b1;
    jal = 1'b0;
    jalr = 1'b0;
    mem2reg = 1'b0;
    branch = 1'b0;
  end
  `OP_JAL: begin
    alusrc = 1'b0;
    regwrite = 1'b1;
    lui = 1'b0;
    memwrite = 1'b0;
    jal = 1'b1;
    jalr = 1'b0;
    mem2reg = 1'b1;
    branch = 1'b0;
  end
  `OP_JALR: begin
    alusrc = 1'b1;
    regwrite = 1'b1;
    lui = 1'b0;
    memwrite = 1'b0;
    jal = 1'b0;
    jalr = 1'b1;
    mem2reg = 1'b0;
    branch = 1'b1;
  end
  `OP_B: begin
    alusrc = 1'b0;
    regwrite = 1'b0;
    lui = 1'b0;
    memwrite = 1'b1;
    jal = 1'b0;
    jalr = 1'b0;
    mem2reg = 1'b0;
    branch = 1'b1;
  end
  `OP_I_LOAD: begin
    alusrc = 1'b1;
    regwrite = 1'b1;
    lui = 1'b0;
    memwrite = 1'b0;
    jal = 1'b0;
    jalr = 1'b0;
    mem2reg = 1'b1;
    branch = 1'b0;
  end
  default: begin
    alusrc = 1'b0;
    regwrite = 1'b0;
    lui = 1'b0;
    memwrite = 1'b0;
  end
endcase

```

OP코드를 해석하여 Control 신호를 지정해주는 부분

```

always @* begin
  if (lui) alusrc1 = 0;
  else if (jal) alusrc1 = pc;
  else alusrc1 = rs1_data;
end

always @* begin
  if (alusrc) alusrc2 = {{20{inst[31]}}, inst[31:20]};
  else if (lui) alusrc2 = {inst[31:12], 12'b0};
  else if (memwrite) alusrc2 = {{20{inst[31]}}, inst[31:25], inst[11:7]};
  else if (jal) alusrc2 = 4;
  else alusrc2 = rs2_data;
end

```

alusrc1, 2을 지정해주는 부분. R타입일 경우 레지스터에 담긴 정보를 이용하고, I 타입일 경우 imm를 sign extension, lui 일 경우 upper immediate 연산을 한다. memwrite control 신호를 만났을 경우도 작성해주었고, jal일 경우에는 alusrc2는 4이며, alusrc1은 pc였다.

```
// assign rd_data = aluout;
always@* begin
    if(mem2reg) rd_data = MemRdata;
    else if(jalr) rd_data = pc+4;
    else rd_data = aluout;
end
assign MemAddr = aluout;
assign MemWdata = rs2_data;
assign MemWen = memwrite;
```

rd_data의 경우에도 control 신호에 맞게 만들어 주었다. mem2reg의 경우 rd_data를 mem에서 읽은 데이터로 채워준다. jalr의 경우 현재 pc의 다음 값 pc+4를 레지스터에 저장해둔다. 나머지 경우에는 alu의 연산값을 저장한다.

module alu

```
`define ALU_ADD 5'b0_0000
`define ALU_SUB 5'b1_0000
`define ALU_OR 5'b0_0010
`define ALU_AND 5'b0_0001
`define ALU_XOR 5'b0_0011
`define ALU_SRL 5'b0_0101
`define ALU_SRA 5'b0_0110
`define ALU_SLL 5'b0_0100
`define ALU_SLT 5'b1_0111
`define ALU_SLTU 5'b1_1000
```

alu에서는 가독성을 높이기 위해 define으로 각 비트에 해당하는 명령들을 쉽게 적어주었다.

```
always @ (*) begin
    case (control)
        `ALU_ADD: begin
            result = a + b;
        end
        `ALU_SUB: begin
            result = a - b;
        end
        `ALU_OR: result = a | b;
        `ALU_AND: result = a & b;
        `ALU_XOR: result = a ^ b;
        `ALU_SRL: result = a >> b;
        `ALU_SRA: result = a >>> b;
        `ALU_SLL: result = a << b;
        `ALU_SLT: result = (a<b) ? 1 : 0;
        `ALU_SLTU: begin
            if (a[31])
                tempA = (~a) + 1;
            else
                tempA = tempA;
            if (b[31])
                tempB = (~b) + 1;
            else
                tempB = tempB;
            result = (tempA < tempB) ? 1 : 0;
        end
        default: result = 0;
    endcase
end
```

컨트롤 신호에 해당하는 값을 읽어 연산을 진행한 후 결과로 내보내는 코드를 추가해 주었다.

```
assign sign = control[4];
assign bxor = (sign)? ~b + 1 : b;
```

```
RippleAdder rca (
    .a(a),
    .b(bxor),
    .ci(1'b0),
    .s(sum),
    .co(c[32]),
    .last_ci(c[31])
);
assign N = sum[31];
assign Z = (sum == 32'b0);
assign C = c[32]; //c32 = cout
assign V = c[31] ^ c[32];
```

N, Z, C, V 플래그 계산을 위해 작성한 코드이며, 뺄셈을 수행한 결과값을 활용해야 하기에 b값을 2의 보수로 만들어 주었다.
이후 Ripple Carry Adder를 거쳐 연산결과를 뱃어낸다.

```
module RippleAdder (
    input[31:0] a,
    input[31:0] b,
    input ci,
    output[31:0] s,
    output co,
    output last_ci
);

    wire [32:0] c;

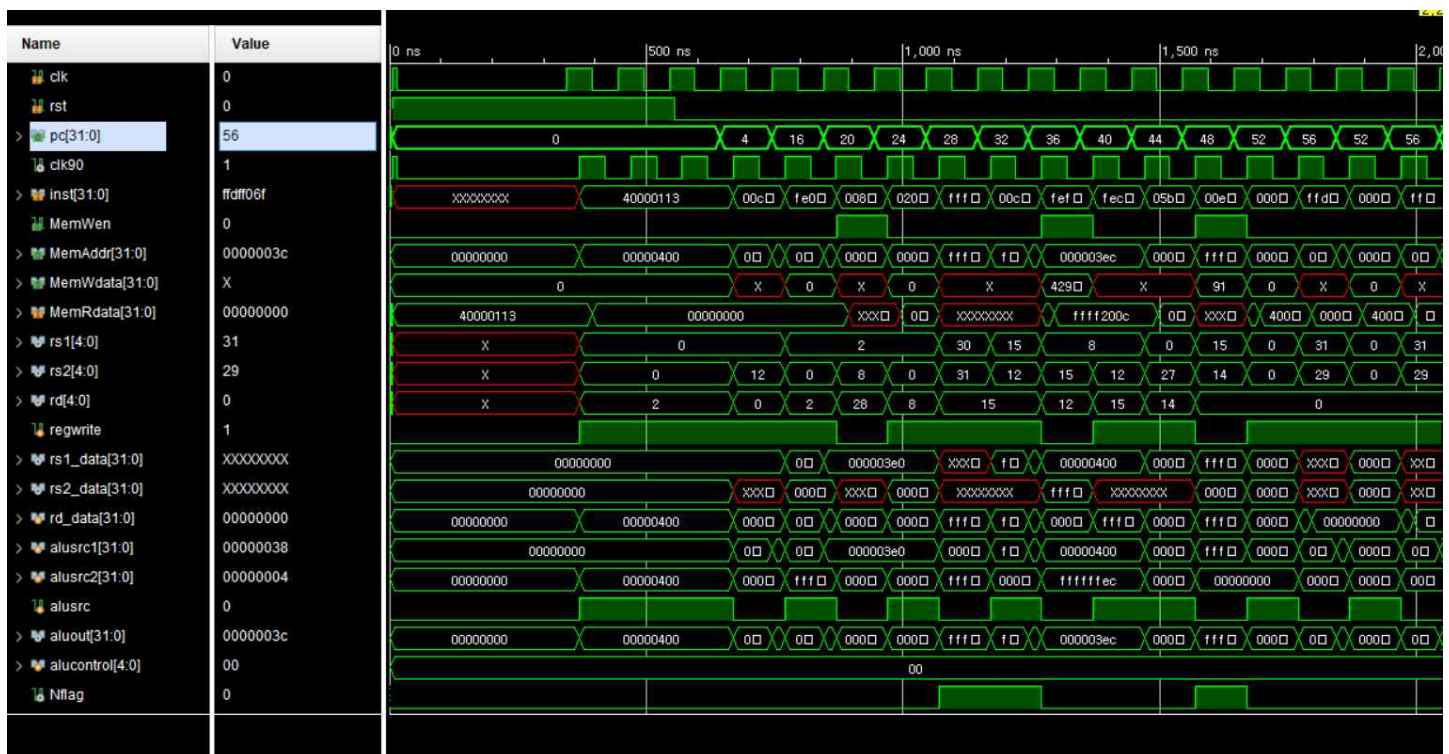
    assign c[0] = ci;
    assign co = c[32];
    assign last_ci = c[31];

    FullAdder fa0(a[0], b[0], c[0], s[0], c[1]);
    FullAdder fa1(a[1], b[1], c[1], s[1], c[2]);
    FullAdder fa2(a[2], b[2], c[2], s[2], c[3]);
    FullAdder fa3(a[3], b[3], c[3], s[3], c[4]);
```

FullAdder를 32개 가지고 있는 RCA 모듈

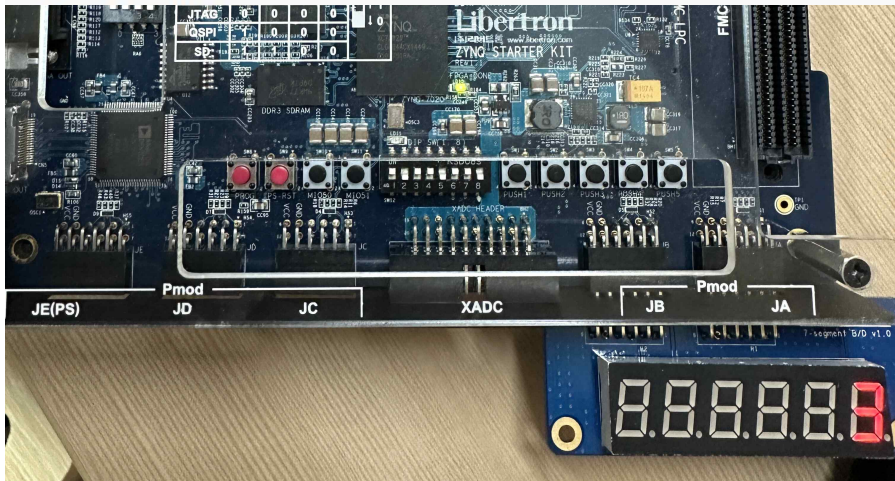
3. Discussion

milestone 1



먼저 clk와 clk90의 클락을 보면 위상이 90만큼 이동한 모습을 볼 수 있다. 이후 pc는 4씩 증가된다.
inst 부분을 보면 coe 파일의 내용대로 잘 수행하는 것을 알 수 있다.

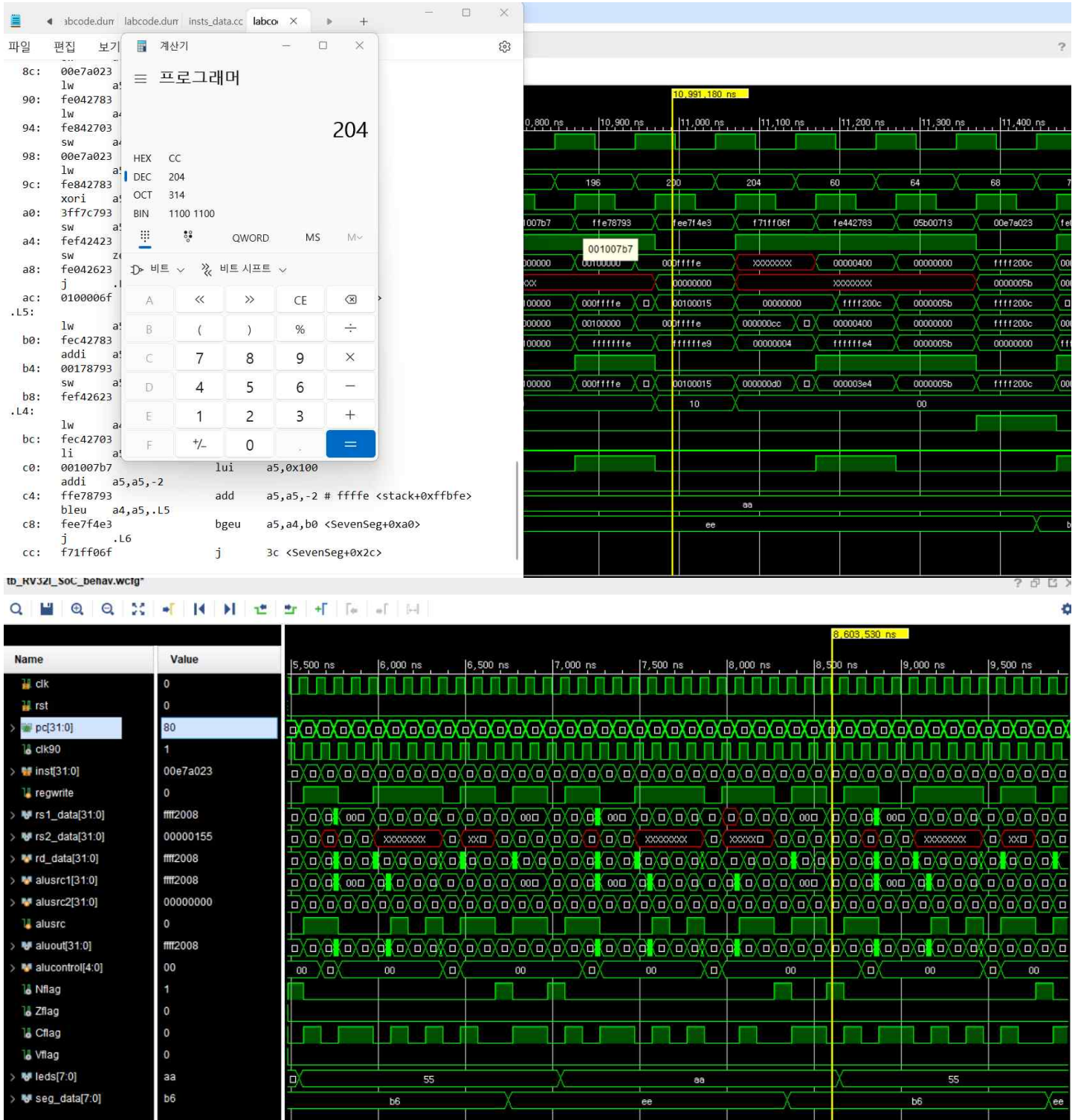

```
memory_initialization_radix=16;  
memory_initialization_vector=  
40000113,  
00C0006F,  
00000000,  
00000000,  
FE010113,  
00812E23,  
02010413,  
FFFF27B7,  
00C78793,  
FEF42623,  
FEC42783,  
05B00713,  
00E7A023,  
00000013,  
FFDFF06F,
```



3을 표시해주는 모습

milestone 2-1

milestone2-1의 경우에는 bgeu 명령이 수행되어야 한다.



파형을 분석하면, led와 seg가 바뀌는 것을 알 수 있다.

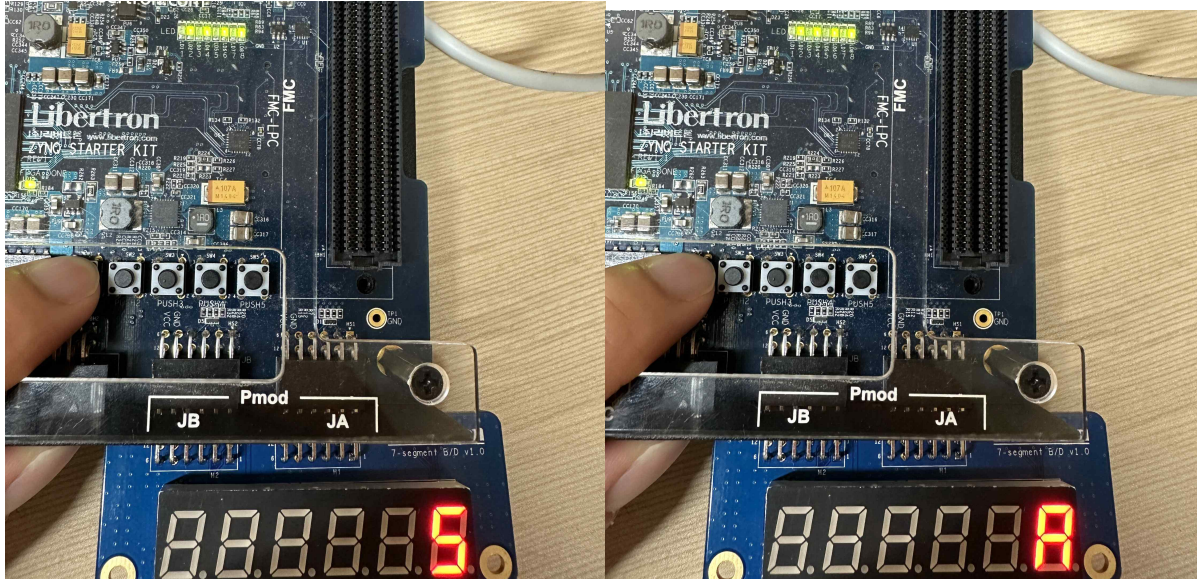
≡ 프로그래머

244

display:

```
.size 2
.align 2
.globl display
.type display, @function

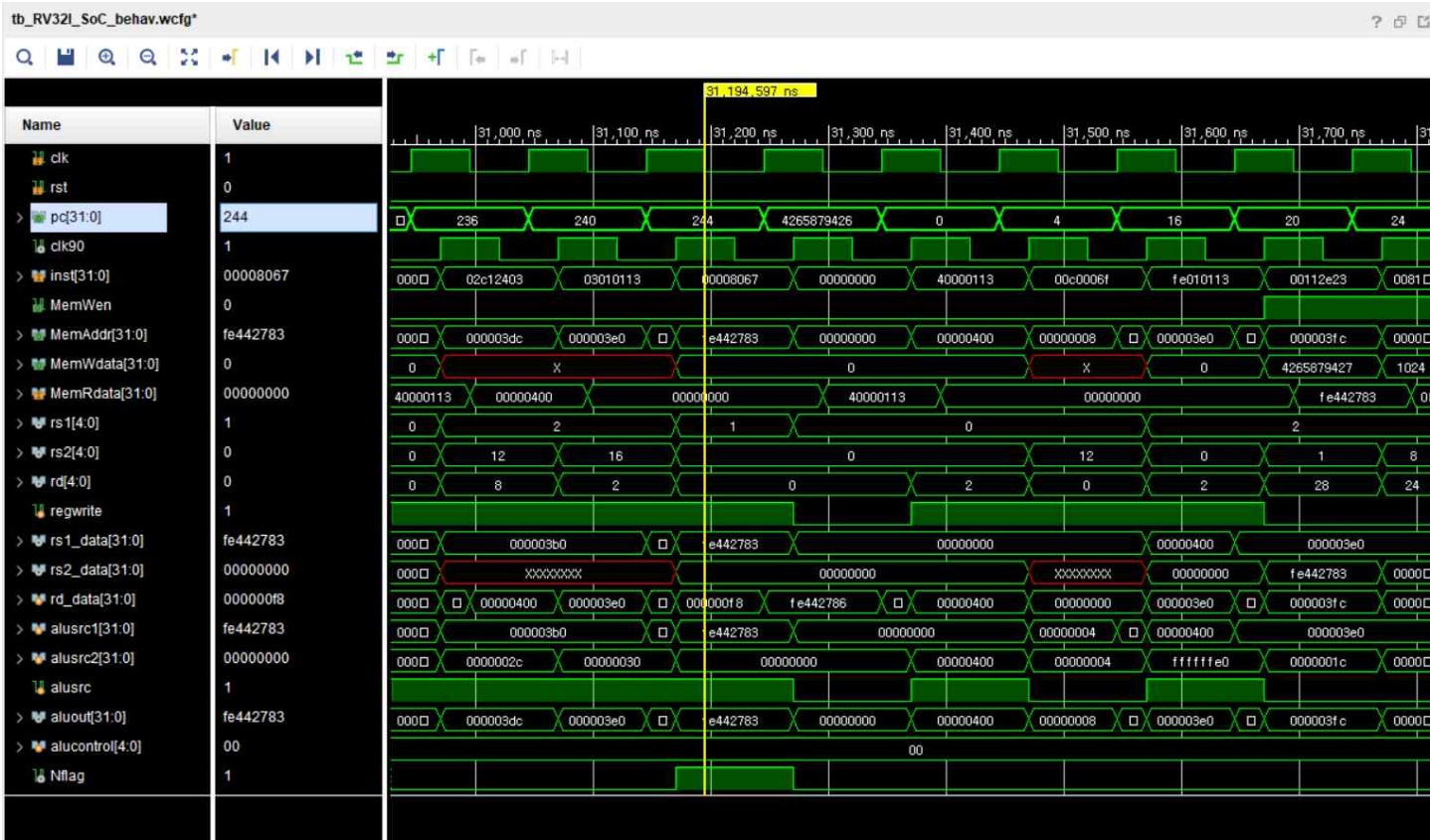
display:
    addi    sp, sp, -16
    c0:     fd010113    HEX    F4
    sw      s0, c0
    c4:     02812623    DEC    244
    addi    sp, sp, 4
    c8:     03010413    OCT    364
    sw      s0, c8
    cc:     fca42e23    BIN    1111 0100
    li      a0, c0
    d0:     ffff27b7
    addi    a0, a0, 1
    d4:     00c78793
    or      a0, a0, 0xffff1c0c
    sw      a0, d4
    d8:     fef42623
    lw      a0, d8
    dc:     fdc42703
    lw      a0, dc
    e0:     fec42783
    sw      a0, e0
    e4:     00e7a023
    nop
    e8:     00000013
    lw      s0, 44(sp)
    ec:     02c12403    lw      s0, 44(sp)
    addi    sp, sp, 48
    f0:     03010113    add    sp, sp, 48
    jr      ra
    f4:     00008067    ret
```

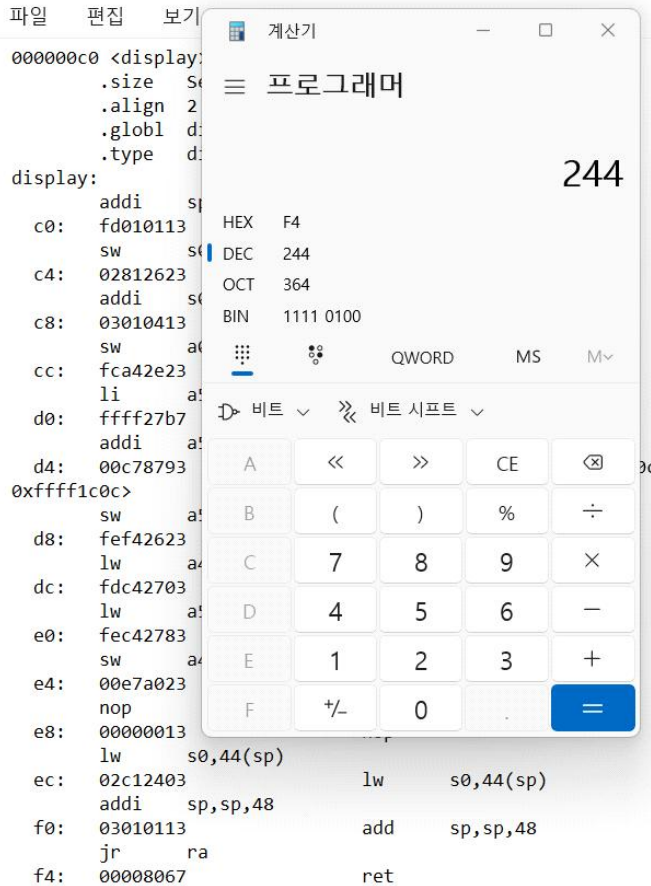


led 101010 -> 010101, 5 -> A로 바뀌는 모습

milestone 2-2

milestone 2-2에서는 jal-ret의 수행이 되어야 한다.
그러나 pc를 볼 때 정상적으로 증가하다 jal-ret 부분을 만나면 pc가 쓰레기 값이 들어가는 문제가 발생하여 설계를 완성하지 못했다.





244는 F4이므로 ret을 해야하지만 pc가 ret 되지 않고 쓰레기 값이 들어가 다시 0으로 초기화되어 버리는 문제가 있다.

milestone 3

```

always @* begin
    if (addr[31:13] == 19'h0) begin // instruction & data
        cs_mem <= 1'b1;
        cs_timer <= 1'b0;
        cs_uart <= 1'b0;
        cs_gpio <= 1'b0;
    end
    else if (addr[31:12] == 20'hFFF2) begin // GPIO
        cs_mem <= 1'b0;
        cs_timer <= 1'b0;
        cs_uart <= 1'b0;
        cs_gpio <= 1'b1;
    end
    else if (addr[31:12] == 20'hFFF1) begin // uart
        cs_mem <= 1'b0;
        cs_timer <= 1'b0;
        cs_uart <= 1'b1;
        cs_gpio <= 1'b0;
    end
    else if (addr[31:12] == 20'hFFF0) begin // timer
        cs_mem <= 1'b0;
        cs_timer <= 1'b1;
        cs_uart <= 1'b0;
        cs_gpio <= 1'b0;
    end
end
end

```

마일스톤 3는 타이머와, 유아트 주변장치 두 개를 추가해야한다. 이 프로젝트에서는 memory mapped io 이기 때문에 address 디코더를 통해 주소번지를 해석해 chip select 신호를 발생시키는 코드를 추가했다.

Rv32I_soc.v(하이라키의 top)에

```
TimerCounter TimerCounter(
    .clk(clk),
    .rst(rst),
    .CS(cs_timer),    //from Address Decoder
    .RD(~data_we),    //~MemWrite
    .WR(data_we),     //from rv32i_opu (MemWrite)
    .Addr(data_addr), // from rv32i_opu (MemAddr)
    .DataIn(write_data), // from rv32i_opu (MemWriteData)
    .DataOut(read_data_timer), // rv32i_opu selects DataOut from I/O Peripherals
    .Intr()    // open
);

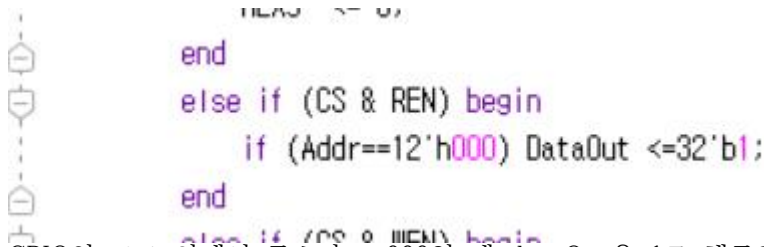
uart_top uart_top(
    .clk(clk),
    .rst(rst),
    .uart_tx_en(cs_uart),
    .uart_tx_data(sw),
    .uart_txd(uart_txd),
    .uart_rxd(uart_rxd),
    .uart_rx_data(leds)
);
```

코드를 추가하여 chip select를 읽어 동작하도록 추가해주었다. 유아트의 경우 cs_uart 가 들어오면 sw(스위치)의 데이터를 읽어 전송한다. rx의 경우 rx 데이터가 들어오면 leds에 표시해주도록 하였다.

```
1 #ifndef SEVENSEG_H
2 #define SEVENSEG_H
3
4 #define GPIO_BASE    0xFFFF2000
5 #define Button_Status GPIO_BASE + 0
6 #define SW_Status    GPIO_BASE + 1 //if pointer of int, increment(+1) is
7 #define LEDG         GPIO_BASE + 2
8 #define SevenSeg0    GPIO_BASE + 3
9 #define SevenSeg1    GPIO_BASE + 4
10 #define SevenSeg2    GPIO_BASE + 5
11 #define SevenSeg3    GPIO_BASE + 6
12 #define SevenSeg4    GPIO_BASE + 7
13 #define SevenSeg5    GPIO_BASE + 8
14 #define SEG_0        0x7E /* Display "0" on 7 Segment */
15 #define SEG_1        0x30 /* Display "1" on 7 Segment */
16 #define SEG_2        0x60 /* Display "2" on 7 Segment */
17 #define SEG_3        0x79 /* Display "3" on 7 Segment */
18 #define SEG_4        0x33 /* Display "4" on 7 Segment */
19 #define SEG_5        0x5B /* Display "5" on 7 Segment */
20 #define SEG_6        0x5F /* Display "6" on 7 Segment */
21 #define SEG_7        0x72 /* Display "7" on 7 Segment */
22 #define SEG_8        0x7F /* Display "8" on 7 Segment */
23 #define SEG_9        0x7B /* Display "9" on 7 Segment */
24 #define SEG_A        0x77 /* Display "A" on 7 Segment */
25 #define SEG_B        0x1F /* Display "B" on 7 Segment */
26 #define SEG_C        0x4E /* Display "C" on 7 Segment */
27 #define SEG_D        0x3D /* Display "D" on 7 Segment */
28 #define SEG_E        0x4F /* Display "E" on 7 Segment */
29 #define SEG_F        0x47 /* Display "F" on 7 Segment */
30 #define SEG_        0x01 /* Display "-" on 7 Segment */

1 #include "SevenSeg.h"
2
3 void display (int);
4
5 void SevenSeg()
6 {
7     unsigned int * uart_btn = (unsigned int *) Button_Status;
8     unsigned int * led_addr = (unsigned int *) LEDG;
9     unsigned int i, data;
10
11     data = 0x155;
12
13     while (1){
14         if(*uart_btn == 1){
15             *led_addr = data;
16         }
17
18         return;
19     }
20
21
22 }
```

c코드는 uart_btn을 선언하고 Button_status의 주소를 지정한 후, 무한루프 안에서 uart_btn == 1이면 led에 data를 표시하도록 하였다.



```

end
else if (CS & REN) begin
    if (Addr==12'h000) DataOut <=32'b1;
end
else if (CS & !REN) begin

```

GPIO의 코드 안에서 주소가 x_000일 때 dataOut을 1로 해주었다.

-> 최종적으로는 verilog 코드 오류와, c언어 data선언 등의 문제로 합성에 실패하였다.

오류 분석

- 1) milestone 2-2를 할 때 jalr의 명령을 수행하면 pc가 쓰레기가 차는 문제.
베릴로그 코드상의 jalr을 수행할 때 레지스터에 저장된 pc 값을 다시 불러와야 한다. 분기를 하기전 레지스터에 pc 값을 저장할 때는 pc+4를 맞게 저장하나, 후에 pc가 쓰레기가 되는 것을 볼 때 branch 후 레이블에 해당하는 코드를 수행하고 ret을 만나 다시 돌아오며 pc가 저장된 레지스터를 참조하는 과정에서 해당 레지스터의 값이 변하여 그런 것으로 추정하고 있다.
- 2) uart 설계 시 uart_tx_en의 생성문제
버튼 신호를 받을 때 uart_tx를 하고 싶다면, 그냥 로직 설계만 할 때에는 constraint 파일에 uart_tx_en 포트를 버튼의 핀맵에 할당하면 되었다. 그러나 RISC-V 코드 상에서 트리거를 줄 때에는 RISC-V가 GPIO의 버튼에 해당하는 주소에서 LW로 읽어와 트리거를 발동시켜주어야 한다. Verilog의 gpio 파일에서 CS와 REN이 모두 1일 때 gpio의 버튼 레지스터에 접근하는 방법까지는 알겠으나, 이후 상위 하이러키로 값을 알려주어 UART 모듈로 전달할 때, TOP 모듈에서 와이어로 트리거와 이어서 전송해야 하는지, 다른 방법이 있는지 잘 모르겠다.
- 3) C코드 작성 시 led 주소에 값을 쓰는 방법
예제로 주어진 C코드 상에서는 led에 값을 쓸 때, 고정된 변수를 사용하여 입력 해주었다. 그러나 milestone3에서는 uart에서 전송받은 값을 메모리에 저장해두고 그 메모리의 주소에 담긴 값을 읽어와 led에 써주어야 한다고 생각한다. 이 때 uart에서 전송받은 값을 저장할 메모리 주소를 uart_top 코드안에서 작성해 주어야 할 것 같은데, 그 방법이 명확히 생각나지 않았다. 또한 C코드 헤더에 uart에서 전송받은 값을 저장하는 메모리의 주소를 적어주어야 작동할 것 같으나 GPIO의 메모리 주소만 적혀있어, 헤더에 Periperal의 메모리 주소 값을 적어주는 것이 올바른 설계인지 모르겠다.
- 4) 메모리 IP에서 COE 파일을 받지 않는 문제
메모리 IP에서 같은 이름의 COE 파일을 교체할 때 문제가 생기는 일이 있어, 이름을 바꾸어주거나, vivado에서 저장하고 있는 coe파일을 모두 삭제해주어야 한다.
- 5) c코드 상에서 for (i=0; i<0xFFFFF; i++) ;를 수행하나 클락은 그만큼 기다리지 않는 문제.
10mhz의 클락을 사용하니 fffff * 10mhz 만큼의 시간이 지나야 다음 c코드를 수행하도록 코드를 짰으나, 돌릴 때 그만큼 기다리지 않고 10만개의 클락이 지날 때 까지 기다린 후, 상태가 바뀐다. //for (i=0; i<0x10; i++) ; 10클락 기다리는 c코드는 주석처리 해주었으나, 10클락 마다 pc상태가 바뀌는 현상이 발생하였다.

4. 결론

이번 프로젝트를 하면서, Cross Compiler를 사용하여 서로 다른 아키텍처 환경에서 실행파일을 만드는 방법을 배울 수 있었다. verilog코드로 실제로 risc-v 로직을 설계하며, risc-v 어셈블리 코드의 명령어를 해석하여 데이터 패스와 컨트롤 명령을 만드는 방법을 배울 수 있었으며, 부가적으로 verilog 코드를 사용해 register 파일을 설계하는 법을 배울 수 있었다. 실제로 10개 남짓한 명령어들을 기반으로 설계해보았으나, 실제 risc-v에는 더 많은 명령어들이 있고, 싱글 사이클을 사용하여 설계해주었기에 완전히 설계를 수행하지 못했다. 기회가 된다면 milestone3에서 발생한 오류들을 수정해보고, 나머지 명령어와, 파이프라이닝 혹은 멀티 사이클 시스템을 갖추도록 하여, 설계해보는 시간을 가져보려 한다.