

REPORT

IEEE Code of Ethics

(출처: <http://www.ieee.org>)

We, the members of the IEEE, in recognition of the importance of our technologies in affecting the quality of life throughout the world, and in accepting a personal obligation to our profession, its members and the communities we serve, do hereby commit ourselves to the highest ethical and professional conduct and agree:

1. to accept responsibility in making decisions consistent with the safety, health and welfare of the public, and to disclose promptly factors that might endanger the public or the environment;
2. to avoid real or perceived conflicts of interest whenever possible, and to disclose them to affected parties when they do exist;
3. to be honest and realistic in stating claims or estimates based on available data;
4. to reject bribery in all its forms;
5. to improve the understanding of technology, its appropriate application, and potential consequences;
6. to maintain and improve our technical competence and to undertake technological tasks for others only if qualified by training or experience, or after full disclosure of pertinent limitations;
7. to seek, accept, and offer honest criticism of technical work, to acknowledge and correct errors, and to credit properly the contributions of others;
8. to treat fairly all persons regardless of such factors as race, religion, gender, disability, age, or national origin;
9. to avoid injuring others, their property, reputation, or employment by false or malicious action;
10. to assist colleagues and co-workers in their professional development and to support them in following this code of ethics.

위 IEEE 윤리헌장 정신에 입각하여 report를 작성하였음을 서약합니다.

학 부: 전자공학과

제출일: 2024.12

과목명: Computer Organization and Architecture

교수명: 이효근 교수님

분 반: C014-1

학 번: 202020963

성 명: 안재형

<제목 차례>

1. Lab1	3
1.1. register_file.v	3
2. Lab2	7
2.1. cla16.v	7
2.2. alu.v	10
2.3. shifter.v	15
2.4. psr.v	17
3. Lab3	19
3.1. pc.v	19
3.2. instruction_reg.v	22
3.3. decoder.v	24
4. lab4	26
4.1. top_processor.v	26
4.2. Translate binary in imem_imginto human-readable assembly	31

1. Lab1

1.1. register_file.v

1.1.1. code implementation

```
module register_file
(
    input          rst_i,
    input          clk_i,
    input          wr_i,
    input[15:0]    data_i,
    input[3:0]     addr_a_i,
    input[3:0]     addr_b_i,
    output reg[15:0] data_a_o,
    output reg[15:0] data_b_o
);
reg[15:0] reg0_r;
reg[15:0] reg1_r;
reg[15:0] reg2_r;
reg[15:0] reg3_r;
reg[15:0] reg4_r;
reg[15:0] reg5_r;
reg[15:0] reg6_r;
reg[15:0] reg7_r;
```

입출력 포트 정의

16bit 크기를 갖는 8개의 register를 정의

```
always@(posedge rst_i or posedge clk_i) begin
    if(rst_i ==1'b1) begin
        reg0_r <=16'h0;
        reg1_r <=16'h0;
        reg2_r <=16'h0;
        reg3_r <=16'h0;
        reg4_r <=16'h0;
        reg5_r <=16'h0;
        reg6_r <=16'h0;
        reg7_r <=16'h0;
    end
    else begin
        /* TODO: write down your reg update code */
        if(wr_i) begin
            /* wr_i가 asset되었을 때 synchronus하게 register값 업데이트
             * demultiplexer로 합성되었으면 좋겠음
             */
            case(addr_b_i)
                0: reg0_r = data_i;
```

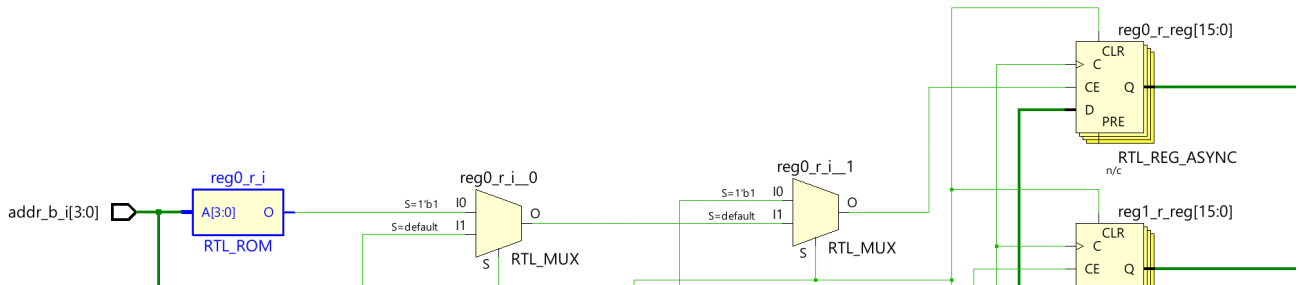
```

        1: reg1_r = data_i;
        2: reg2_r = data_i;
        3: reg3_r = data_i;
        4: reg4_r = data_i;
        5: reg5_r = data_i;
        6: reg6_r = data_i;
        7: reg7_r = data_i;
        default;;
    endcase
end
end
end

```

reset 입력이 1로 assert 될 때 레지스터들을 비동기 초기화 한다.

reset 입력이 deassert되고 wr_i 입력이 assert되었을 경우 clock positive edge에서 해당 주소의 레지스터 값을 업데이트 한다. 8x1 demux 합성을 의도하였지만, 위 case문을 통해서는 의도대로 합성되지 않았다. 아래와 같이 wr_i 조건을 확인하기 위한 2x1 MUX 앞에 주소값을 입력으로 갖는 ROM이 합성되었다.



RTL_ROM에 연결된 레지스터를 가리키는 주소가 입력되었을 때 1을 출력한다. 이는 wr_i 조건을 확인하는 MUX를 지나고 rst_i 조건을 확인하는 MUX를 지나 D flip flop의 Clock enable 입력으로 연결된다.

```

/* TODO: write down your reg reads (data*_o) code */
/* 출력 레지스터 비동기 업데이트
 * address가 업데이트 되었을 때 output register에 비동기 업데이트
 */
always@(addr_a_i or addr_b_i) begin
    //multiplexer로 합성되면 좋겠음
    case(addr_a_i[2:0])
        0: data_a_o = reg0_r;
        1: data_a_o = reg1_r;
        2: data_a_o = reg2_r;
        3: data_a_o = reg3_r;
        4: data_a_o = reg4_r;
        5: data_a_o = reg5_r;
        6: data_a_o = reg6_r;
        7: data_a_o = reg7_r;
        default;;
    endcase
    case(addr_b_i[2:0])
        0: data_b_o = reg0_r;

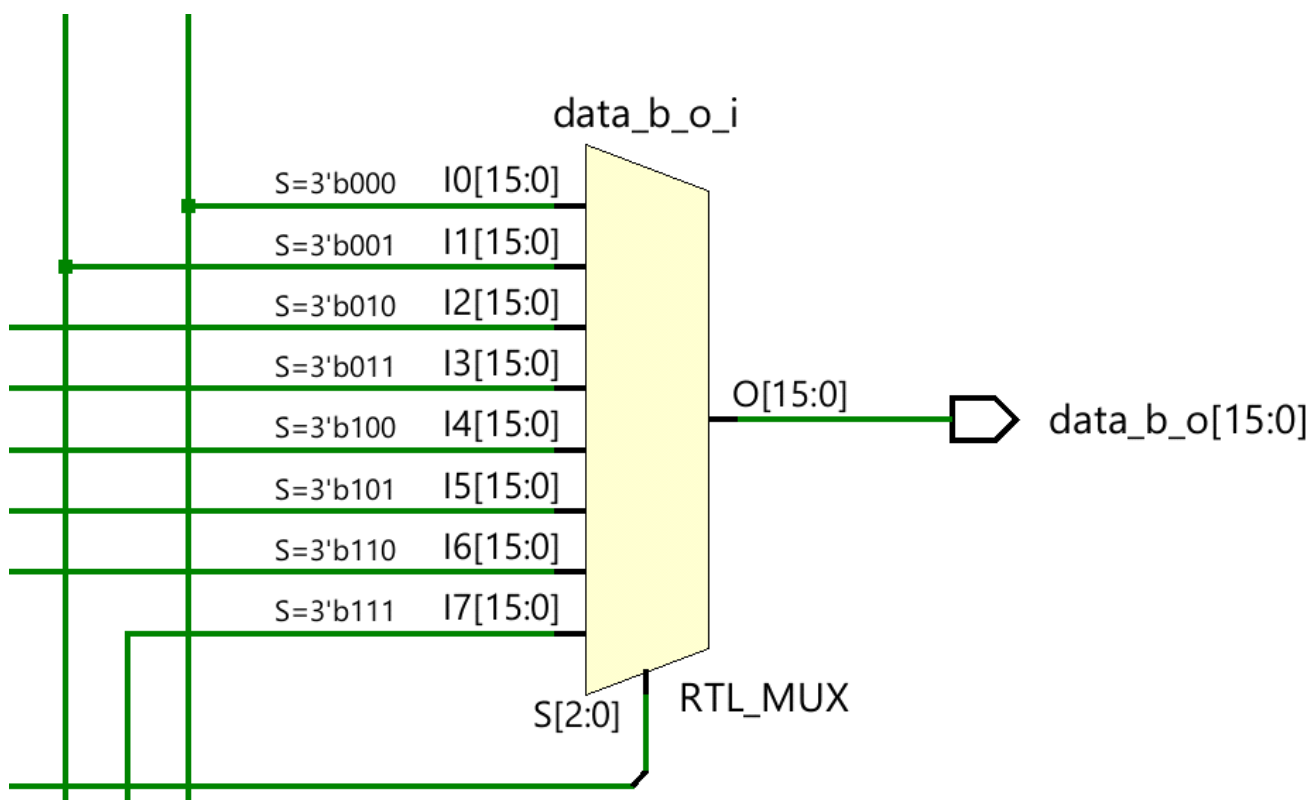
```

```

1: data_b_o =reg1_r;
2: data_b_o =reg2_r;
3: data_b_o =reg3_r;
4: data_b_o =reg4_r;
5: data_b_o =reg5_r;
6: data_b_o =reg6_r;
7: data_b_o =reg7_r;
default;;
endcase
end
endmodule

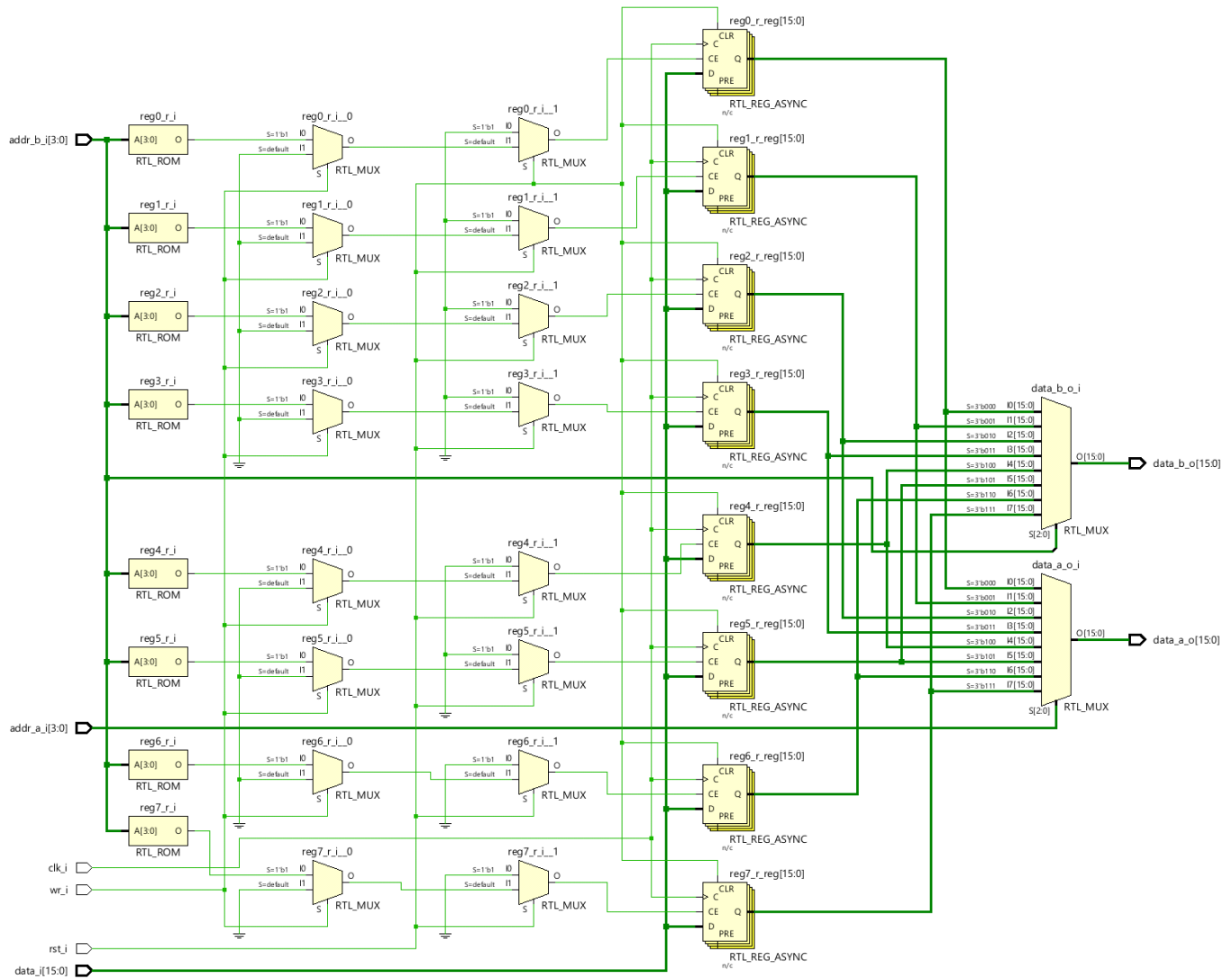
```

두 개의 비동기 출력 포트를 가지고 있으며, 입력되는 주소값에 대응되는 값을 내보낸다. 주소를 select 입력으로, 8x1 multiplexer로 합성되기를 의도하고 구성하였다.

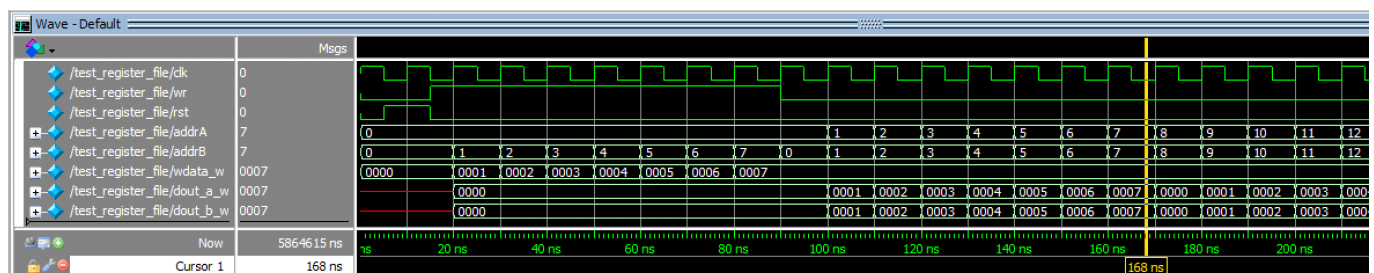


의도한 바와 같이 합성되었음을 확인하였다.

1.1.2. Schematic



1.1.3. Waveform



2. Lab2

2.1. cla16.v

2.1.1. code implementaion

```
You, 14 minutes ago | 1 author (You)
1  /*
2   * CLA does not know A & B are treated as signed or not
3   * So, it passes both F and C flags
4   * opcode selects one of them as jump/branch condition
5   * If opcode is signed (e.g., signed overflow, signed compare),
6   * F will be used
7   */
8  module cla16
9  (
10     input [15:0]    A_i,
11     input [15:0]    B_i,
12     input           CARRYIN_i,
13
14     output          CARRYOUT_no,
15     output [1:0]    flag_overflow_o,
16     output [15:0]   sum_o
17 );
18
19 wire [15:0]    A_used_w;    //B-A를 위해 A*(signed)
20 wire [15:0]    prop_w;      //피연산자1, 2 half sum
21 wire [15:0]    gen_w;       //피연산자1, 2 half carry out
22 wire [15:0]    carry_w;
23 wire [3:0]     gprop_w;     // group-propagate
24 wire [3:0]     ggen_w;     // group-generate
25 wire [3:0]     gcarry_w;    // group-carry
```

cla16 모듈은 3개의 input port, 3개의 output port를 가진다. 각각의 포트의 입출력은 다음과 같다.

A_i: 16 bit 피연산자

B_i: 16 bit 피연산자

CARRYIN_i: 캐리인, 빼기 연산 시 B의 2의 보수를 취해주기 위한 flag로 활용

CARRYOUT_no: flag를 구하는 데 이용될 수 있으나, 이용하지 못했다.

flag_overflow_o: 각 인덱스는 signed, unsigned 연산에 대한 overflow유무를 반환한다.

sum_o: ADD 결과를 출력한다.

```
27 //carry in에 1이 들어왔을 시 B-A연산 실행, 반전 후 cla4_0 모듈의 carry_i포트 1 전달
28 assign A_used_w = (CARRYIN_i == 1'b0) ? A_i : ~A_i;
29
30 assign prop_w = A_used_w ^ B_i;
31 assign gen_w = A_used_w & B_i;
```

뺄셈을 구현하기 위해 CARRYIN_i이 1인 경우 A의 보수를 취한다.

cla4 모듈에 전달할 propagate과 generate를 계산한다.

```

33  cla4
34 >  cla4_0( ...
41  );
42
43  cla4
44 >  cla4_1( ...
51  );
52
53  cla4
54 >  cla4_2( ...
61  );
62
63  cla4
64 >  cla4_3( ...
71  );
72
73  lcu4
74 >  lcu4_0( ...
79  );

```

각 모듈에 나누어 propagate, generate을 전달한다. 이때 lcu는 group propagate과 generate를 이용하여 각 cla 모듈의 carry out을 예측한다. 그리고 이 값은 다시 cla의 carry in으로 feedback되어 계산된다. 여기서 흥미로운 점은, 한 개의 cla 모듈 단위로 carry in, group propagate, group generate을 정의함으로써 일반적으로 carry out을 구하는 과정과 동일한 방법으로 group carry out을 구할 수 있다는 점이었다. cla에서 propagate과 generate을 이용해 carry out을 구하는 방법은 아래와 같다.

- generate: carry in과 무관하게 carry out을 발생 시키는 경우
 $G(A, B) = A \& B$
A, B 모두 1인 경우 항상 carry out이 발생된다. 이 경우 생성된다고 한다.
- propagate: carry in에 종속적으로 carry out을 발생시키는 경우
 $P(A, B) = A \wedge B$
그러나 우리는 $P(A, B) = A | B$ 식을 이용할 수 있다. 이는 Carry out을 예측하는 과정에서 설명된다.
- carry out의 예측
 $C_{i+1} = G_i + P_i C_i$ 의 진리표

A	B	Cin	G = A&B	A B	A^B	Cout
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	1	1	0
0	1	1	0	1	1	1
1	0	0	0	1	1	0
1	0	1	0	1	1	1
1	1	0	1	1	0	1
1	1	1	1	1	0	1

P(A, B)를 결정하기 위하여 carry out을 결정하는 진리표를 볼 때 A|B와 A^B의 값이 다른 {A, B} == 'b11'인 경우 G항이 1이므로 같은 결과가 나타나게 된다. 따라서 더 구현이 간편한 A|B를 이용하기로 한다.

```

81  assign sum_o[0] = CARRYIN_i ^ prop_w[0];
82  assign sum_o[15:1] = carry_w[14:0] ^ prop_w[15:1];

```

첫 번째 결과 비트를 계산하기 위해 A&B의 결과인 propagate와 carry in을 XNOR 한다. 이는 전가산기에서 sum을 구하는 것과 동일한 과정이다.


```

84  /* TODO: please write down code for flag_overflow[0] and [1]
85  * [0] (C-flag): overflow that is meaningful for unsigned operations
86  * ADD에서 carry out == 1일 때 overflow
87  * SUB에서 signed_bit == 1 & carry out == 0 일 때 overflow
88  * MSB까지 이용하여 표현된 unsigned operand의 경우 MSB가 1인 경우에도 carry out이 1이라면 overflow 아니다.
89  * [1] (F-flag): overflow that is meaningful for signed operations (2s complement)
90  */
91
92  assign flag_overflow_o = {carry_w[14] ^ gcarry_w[3],
93  |~CARRYIN_i & gcarry_w[3] | CARRYIN_i & sum_o[15] & ~gcarry_w[3]};
94  assign CARRYOUT_no = !gcarry_w[3];
95
96  endmodule

```

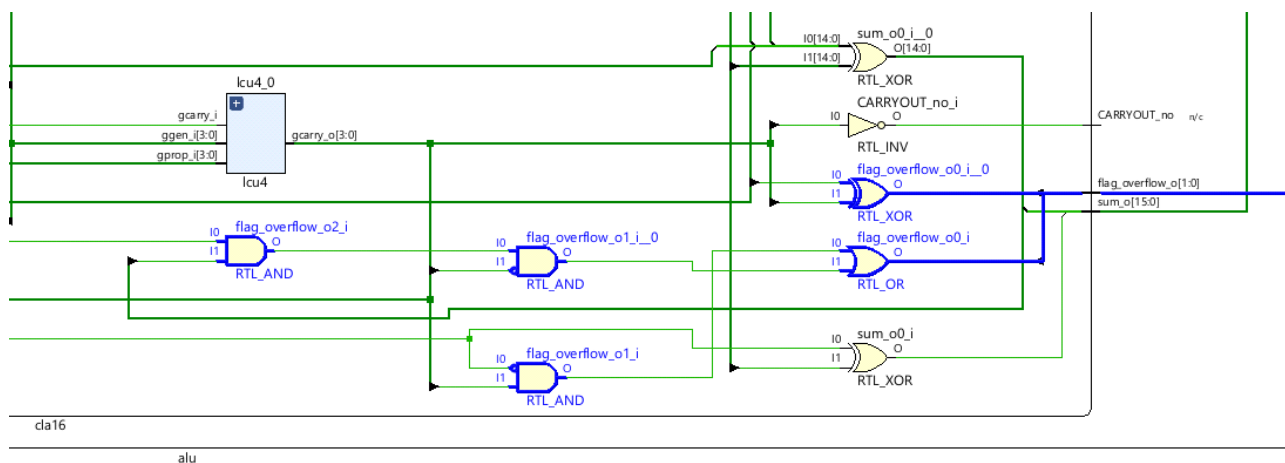
overflow를 감지하기 위한 로직이다.

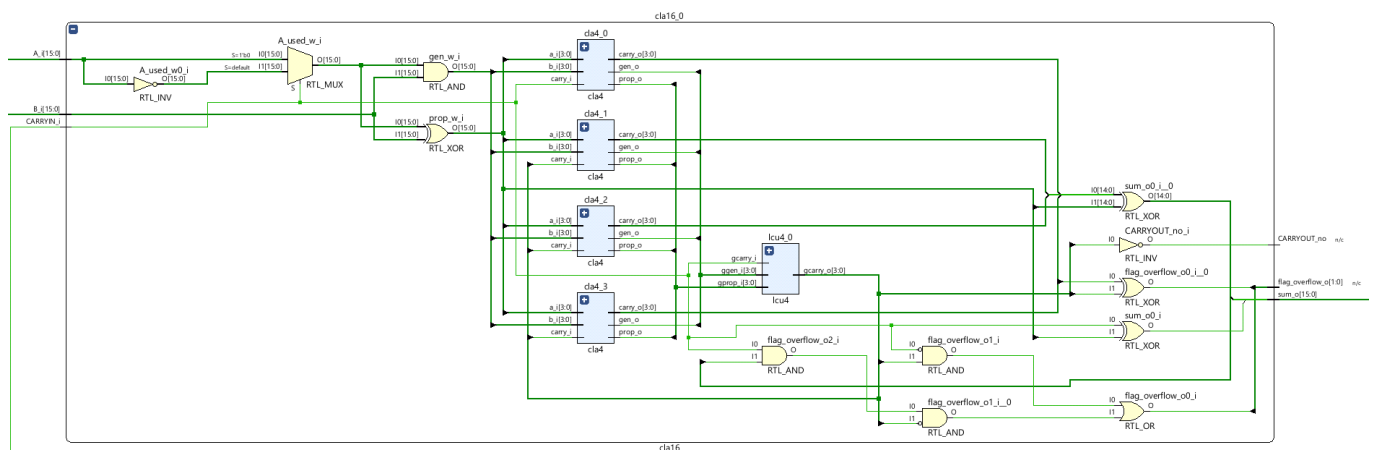
- signed operation인 경우
MSB의 carry in과 carry out 같지 않으면 overflow가 발생되게 된다.
- unsigned operation인 경우
CARRYIN_i == 0일 때(ADD operation), MSB에서 carry out이 발생한 경우 overflow가 발생한 것이다.
CARRYIN_i == 1일 때(SUB operation), MSB가 1일 때 carry out이 발생하지 않은 경우 overflow가 발생한 것이다.

unsigned] [signed]	B [unsigned] [signed]	CLA16b.ADD (carryout, 16b)	CLA16b.SUB (carryout, 16b)	Unsigned		Signed		F		C	
				True.ADD	True.SUB	True.ADD	True.SUB	ADD	SUB	ADD	SUB
0xDEAD [57005] [-8531]	0xCAFE [51966] [-13570]	1, 0xA9AB	0, 0xEC51	0x1_A9AB (>65535)	-5039 (not unsigned!)	0xA9AB	0xEC51	0	0	1	1
0x10AF [4271] [4271]	0xBEEF [48879] [-16657]	0, 0xCF9E	1, 0xAE40	0xCF9E	0xAE40	0xCF9E	0xAE40	0	0	0	0
0xBAAD [47789] [-17747]	0xC0DE [49374] [-16162]	1, 0x7B8B	1, 0x0631	0x1_7B8B (>65535)	0x0631	-33909 (0x7B8B>0)	0x0631	1	0	1	0
0x0003	0x0004	0, 0x0007	1, 0x0001	0x0007	0x0001	0x0007	0x0001	0	0	0	0
0x4B1D [19229] [19229]	0x10AF [4271] [4271]	0, 0x5BCC	0, 0xC592	0x5BCC	-14958 (not unsigned!)	0x5BCC	0xC592	0	0	0	1

위 case에 대해 동일한 F, C flag를 보여준다.

2.1.2. schematic





2.2. alu.v

2.2.1. Code Implementaion

```

You, 2 hours ago | 1 author (You)
1  module alu
2  (
3      input [15:0] A_i,
4      input [15:0] B_i,
5      input [5:0]  alu_sel_i,
6
7      output      [4:0]  flags_o, // F, L, C, N, Z
8      output reg [15:0] alu_o
9  );
10
11  `include "ALU_modes.vh"
12
13  reg CARRYIN_w;
14  wire CARRYOUT_nw;
15  wire [1:0] alu_ovf_w;
16  wire [15:0] cla_sum_w;
17
18  cla16
19      cla16_0(
20          .A_i(A_i),
21          .B_i(B_i),
22          .CARRYIN_i(CARRYIN_w),
23          .CARRYOUT_no(CARRYOUT_nw),
24          .flag_overflow_o(alu_ovf_w),
25          .sum_o(cla_sum_w)
26      );

```

input, output 포트 정의 및 변수, net 정의. cla16 인스턴스화

```
28 /* TODO: Please write down codes for flags_o[4], ..., [0]
29 * [4:0] flags_o = {F, L, C, N, Z} in order
30 * F flag : overflow / underflow @ signed ADD/ SUB
31 * L flag : B<A @ unsigned CMP -> cla16에서 B-A < 0
32 *
33 * C flag : overflow / underflow @unsigned ADD/SUB
34 * N flag : B<A @ signed CMP -> 결과가 음수일 때
35 * Z flag : A==B @ CMP -> sum이 0일 때
36 */
37 assign flags_o = {alu_ovf_w[1], alu_ovf_w[0], alu_ovf_w[0], alu_ovf_w[1] | cla_sum_w[15], (~|cla_sum_w) & ~|alu_ovf_w};
```

FC field는 앞서 전달받은 값 그대로 이용.

L: unsigned 비교에서 B<A를 만족하기 위해서는 B-A<0이 되어야 함. overflow가 생길 조건과 일치하므로 unsigned ADD, SUB overflow flag를 그대로 연결해 주었다.

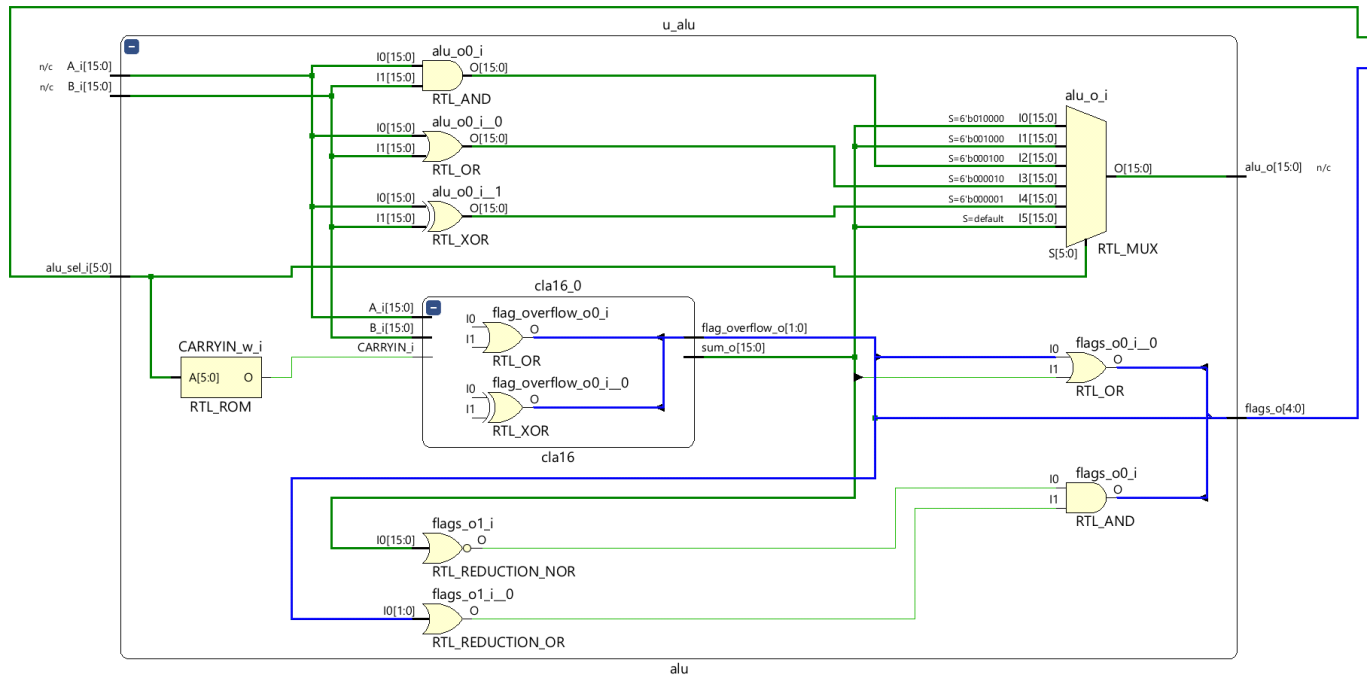
N: signed B<A를 만족하기 위해서는 B-A<0이므로, MSB가 1 또는 표현 가능한 최소 값보다 작아지는 underflow 발생 조건이 이다.

Z: B-A=0이므로 sum의 모든 비트가 0이고 overflow가 발생하지 않았을 조건이다.

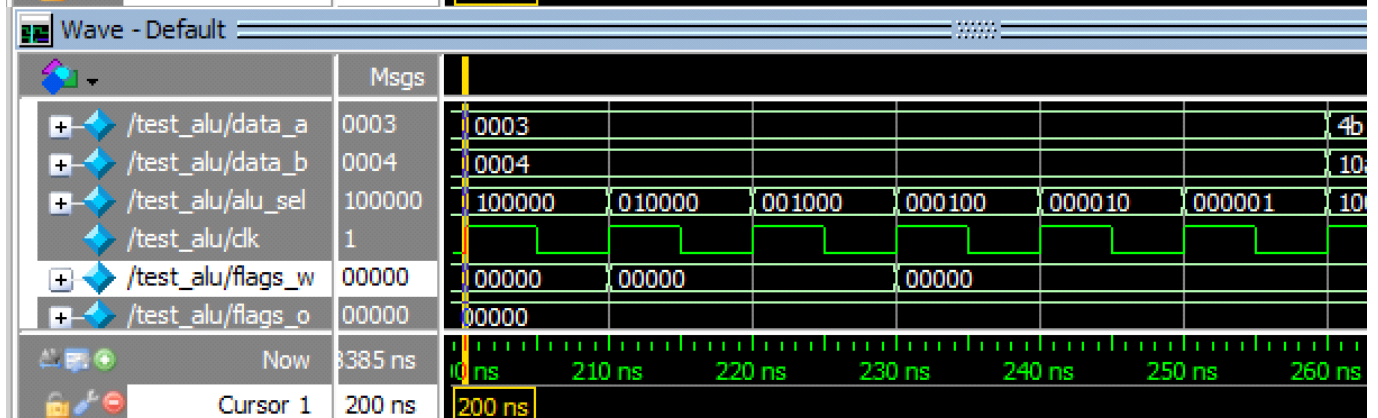
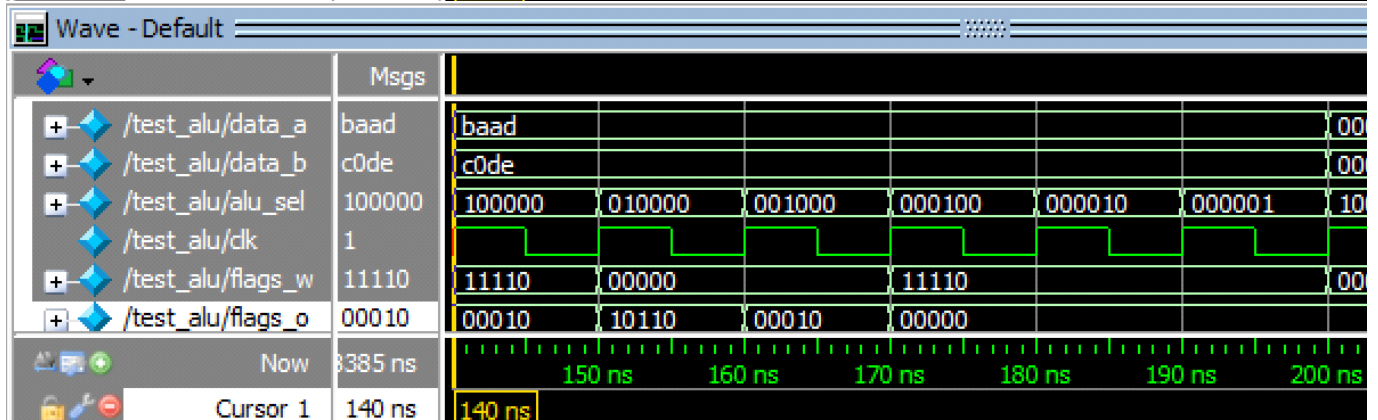
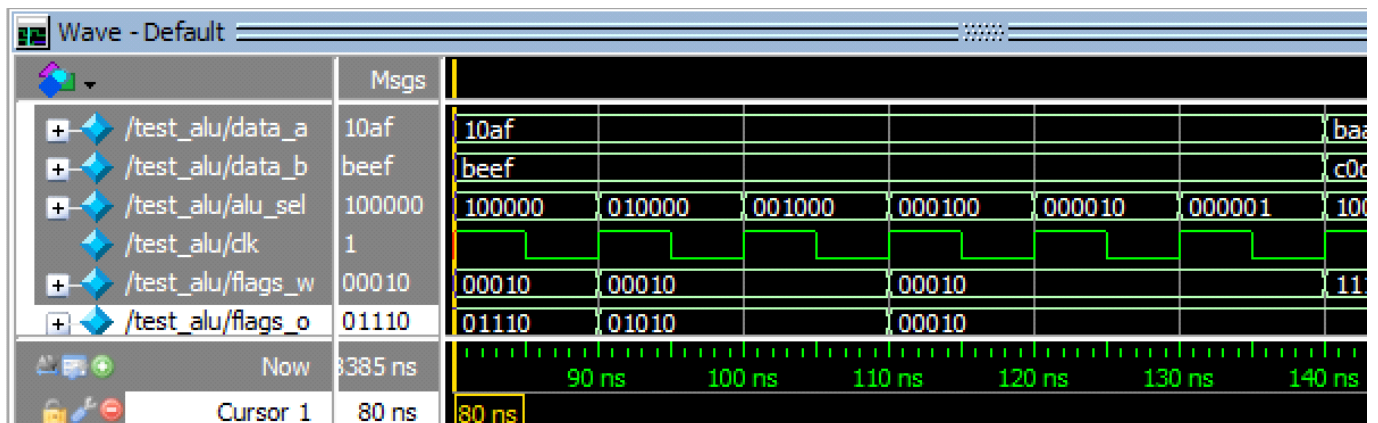
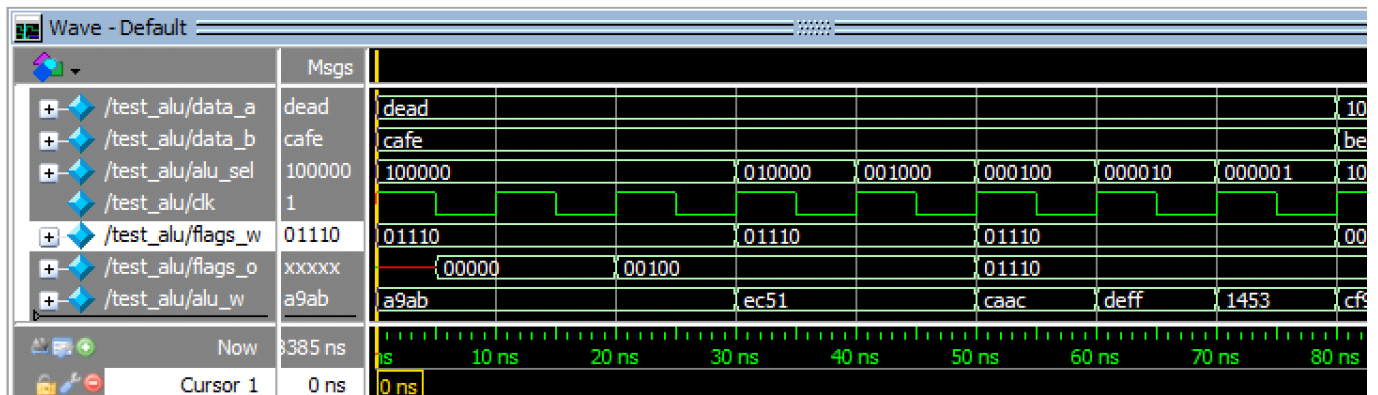
```
39 always @(*) begin
40     CARRYIN_w = 1'b0;           You, last we
41     alu_o = cla_sum_w;
42
43     case (alu_sel_i)
44     ALU_SEL_SUB:
45         | CARRYIN_w = 1'b1;
46     ALU_SEL_CMP:
47         | CARRYIN_w = 1'b1;
48     ALU_SEL_AND:
49         | alu_o = A_i & B_i;
50     ALU_SEL_OR:
51         | alu_o = A_i | B_i;
52     ALU_SEL_XOR:
53         | alu_o = A_i ^ B_i;
54     default:
55         | ;
56     endcase
57 end
58
59 endmodule
```

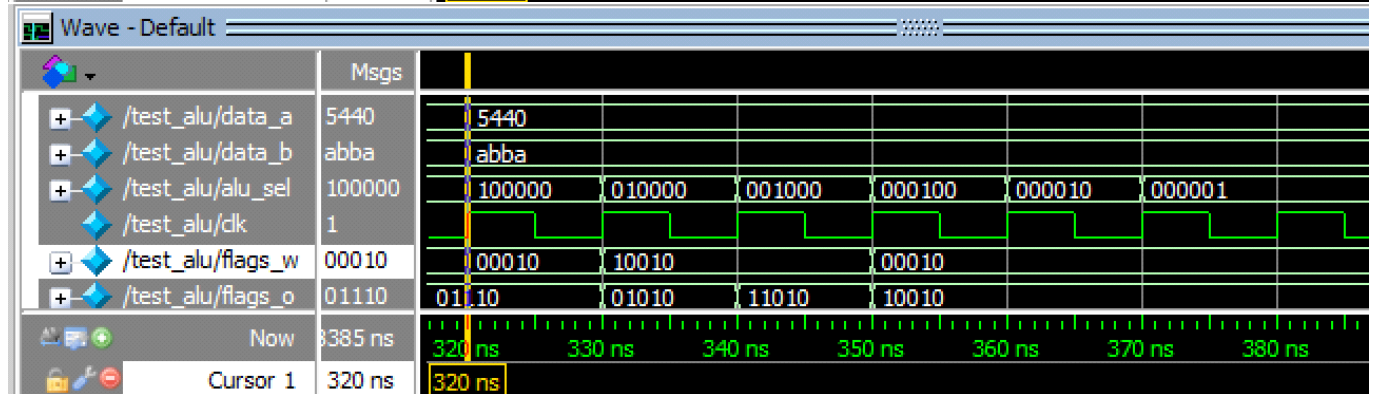
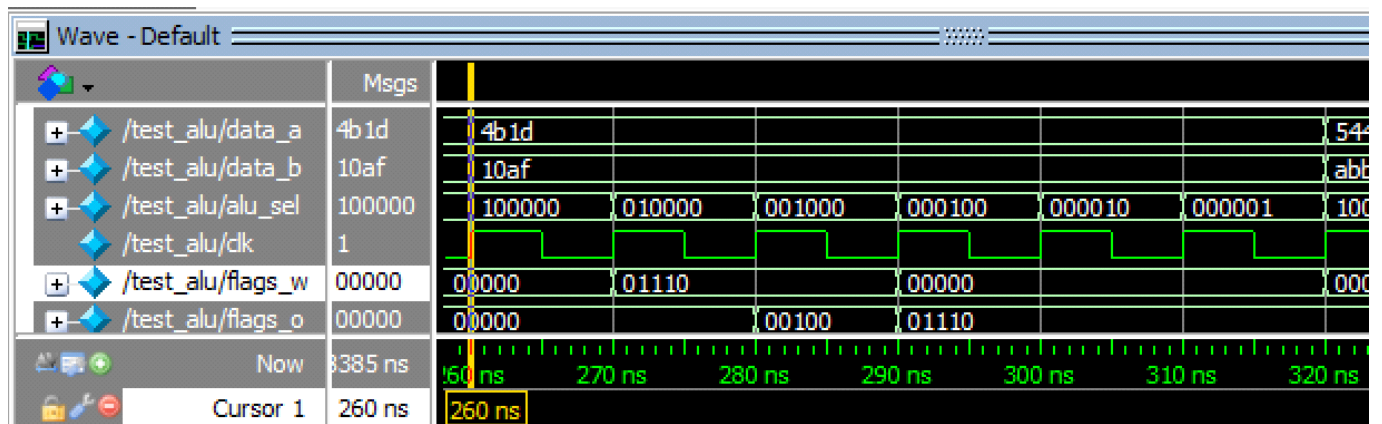
alu의 operation을 결정하는 로직이다.

2.2.2. schematic



2.2.3. test_alu waveform





2.3. shifter.v

2.3.1. Code Implementation

```
You, 2 days ago | 1 author (You)
1  module shifter
2  (
3      input [15:0]    data_i,
4      input [4:0]     rl_shift_amt_i,
5      input          lui_i,
6
7      output [15:0]   data_o
8  );
9
10 wire [30:0] stage_0;
11 wire [22:0] stage_1;
12 wire [18:0] stage_2;
13 wire [16:0] stage_3;
14 wire [15:0] stage_4;
```

입출력 포트 정의 및 16, 8, 4, 2, 1 bit shift를 위한 stage net 정의

data_i: shift 연산할 데이터

rl_shift_amt_i: shift 연산할 크기, 이때 음수이면 오른쪽으로 shift 한다.

lui_i: asserted 시 항상 왼쪽으로 8bit shift 한다.

data_o: 출력 데이터


```

16  /* TODO: Please write down codes for each stage */
17  /* 오른쪽 shift 연산만을 이용해 양방향 shift 구현
18   * amount가 음수일 때 오른쪽 shift, 양수일 때 왼쪽 shift
19   * <Right shift> key: abs = ~amount + 1
20   * 오른쪽 shift는 음수의 절댓값 만큼 이동시켜야 하므로, stage_0에서 1만큼 오른쪽 shift
21   * 이후 amount를 반전하여 해당하는 크기만큼 shift (2의 보수 절댓값)
22   * <Left shift> key: 왼쪽으로 15bit shift 후 되돌려 놓기
23   * 왼쪽으로 정보 손실 없이 15bit shift 시킨 후 amount 반전시켜 오른쪽으로 돌려놓는다
24   * amount == 00001 인 경우
25   * {data, 15'b0} -> {data, 7'b0} -> {data, 3'b0} -> {data, 1'b0} -> {data[14:0], 1'b0}
26   * 결과적으로 왼쪽으로 한 bit shift 된다.
27   * amount를 반전시켜 이용하는 것이 핵심
28  */
29  assign stage_0 = rl_shift_amt_i[4] == 1'b0 ? {data_i, 15'b0} : {1'b0, data_i[15:1]};
30  assign stage_1 = rl_shift_amt_i[3] == 1'b0 ? stage_0[30:8] : stage_0[22:0];
31  assign stage_2 = rl_shift_amt_i[2] == 1'b0 ? stage_1[22:4] : stage_1[18:0];
32  assign stage_3 = rl_shift_amt_i[1] == 1'b0 ? stage_2[18:2] : stage_2[16:0];
33  assign stage_4 = rl_shift_amt_i[0] == 1'b0 ? stage_3[16:1] : stage_3[15:0];
34
35  //lui input: treat rl_shift_amt_i ad dont'care and simply shift left for 8 bits
36  assign data_o = (lui_i == 1'b1) ? {data_i[7:0], 8'h0} : stage_4;
37
38  endmodule

```

문제에 주어진 stage의 크기가 핵심이다. 16+15bit 크기의 net이 첫 번째 stage로 주어지며, 우선 왼쪽 shift 후 연산을 진행해야 함을 유추할 수 있음.

오른쪽 shift 연산만을 이용해 양방향 shift 구현

amount가 음수일 때 오른쪽 shift, 양수일 때 왼쪽 shift

<Right shift> 핵심 아이디어: $abs = \sim amount + 1'b1$

오른쪽 shift는 shift_amt의 절댓값만큼 이동시켜야 하므로, stage_0에서 1만큼 오른쪽 shift 이후 amount를 반전하여 해당하는 크기만큼 shift한다. 이는 결과적으로 amount를 보수 취하여 절댓값 만큼 이동 시킨 것과 같다.

<Left shift> 핵심 아이디어: 왼쪽으로 15bit shift 후 되돌려 놓기

왼쪽으로 정보 손실 없이 15bit shift 시킨 후 amount 반전시켜 오른쪽으로 돌려놓는다

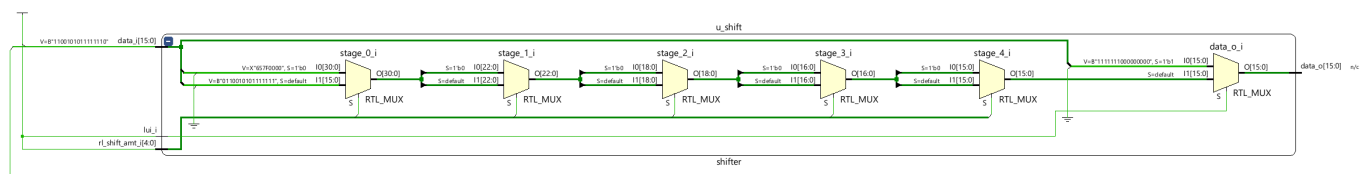
amount == 00001 인 경우

{data, 15'b0} -> {data, 7'b0} -> {data, 3'b0} -> {data, 1'b0} -> {data[14:0], 1'b0}

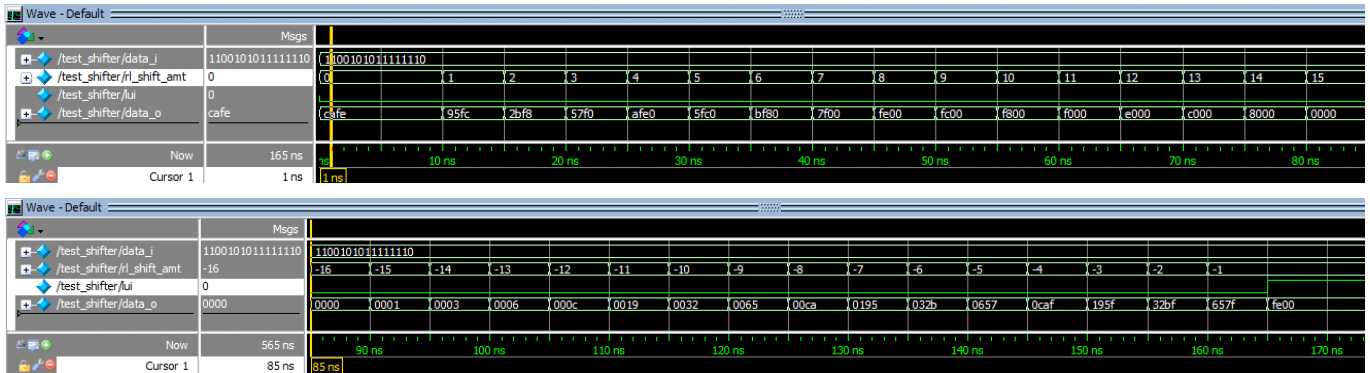
결과적으로 왼쪽으로 한 bit shift 된다.

amount를 반전시켜 이용하는 것이 핵심.

2.3.2. schematic



2.3.3. test_shifter waveform



2.4. psr.v

2.4.1. Code Implementation

```

1  You, last week | 1 author (You)
2  /*      You, last week • init
3  * Program status register
4  * FLCNZ for each bit [4:0]
5  */
6  module psr
7  (
8      input      rst_i,
9      input      clk_i,
10     input [5:0] alu_sel_i,
11     input [4:0] flags_i,      //FLCNZ
12
13     output reg [4:0] flags_o  //selected FLCNZ
14 );
15
16 `include "ALU_modes.vh"

```

포트 정의 및 parameter header include

```

18 always @(posedge rst_i or posedge clk_i) begin
19     if (rst_i == 1'b1)
20         flags_o <= 0;
21     else begin
22         /* TODO: write down codes for propagating flags */
23         case (alu_sel_i)
24             ALU_SEL_ADD: {flags_o[4], flags_o[2]} = {flags_i[4], flags_i[2]};
25             ALU_SEL_SUB: {flags_o[4], flags_o[2]} = {flags_i[4], flags_i[2]};
26             ALU_SEL_CMP: {flags_o[3], flags_o[1], flags_o[0]} = {flags_i[3], flags_i[1], flags_i[0]};
27             default;;
28         endcase
29     end
30 end
31
32 endmodule

```

현재 연산에 대응되는 flag 값만을 update하기 위하여 case문을 이용해 flag register에 값을 저장하는 로직을 구현하였다.

```

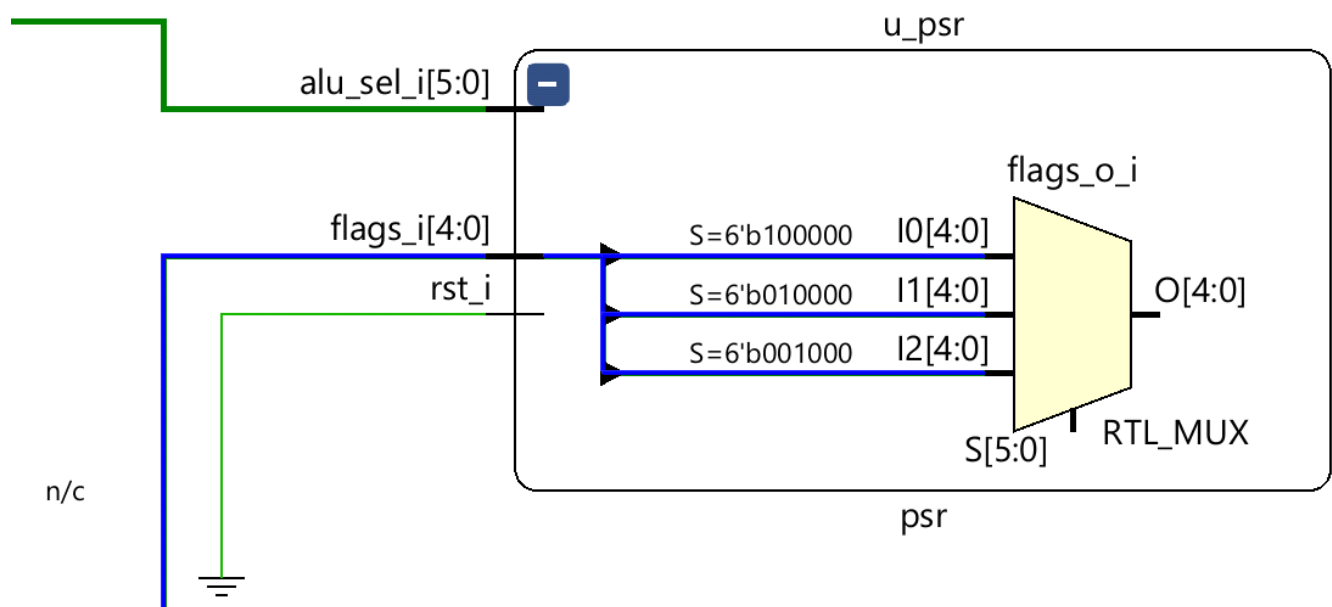
if clk.posedge {
  if reset {
    reset registers
  } else if alu_sel = add or sub {
    reg(F,C) = flag_i(F,C)
  } else if alu_sel = cmp {
    reg(L,N,G) = flag_i(L,N,Z)
  }
}
flags_o = reg{F,L,C,N,Z}

```

위 세 가지 연산에 해당하지 않는 경우 모든 레지스터의 state는 유

지된다.

2.4.2. schematic



3. Lab3

3.1. pc.v

3.1.1. Code Implementation

```
1  `include "../lab2/cla16.v"
2  `include "../lab2/cla4.v"
3  `include "../lab2/lcu4.v"
4  module pc
5  (
6      input      rst_i,
7      input      clk_i,
8      input      jump_i,
9      input      branch_i,
10     input [7:0] displacement_i,
11     input [15:0] jump_tgt_i,
12
13     output [15:0] addr_imem_o // this is instruction pointer!
14 );
15
16 wire [15:0] disp_signEx_c;      //변위 부호 확장
17 wire [15:0] addr_incr_c;        //PC 증분 값
18 wire [15:0] next_addr_imem_w;  //calculated next PC
19 reg [15:0] curr_addr_imem_r;    //stored currnt PC
```

1~11행: 입출력 포트 지정

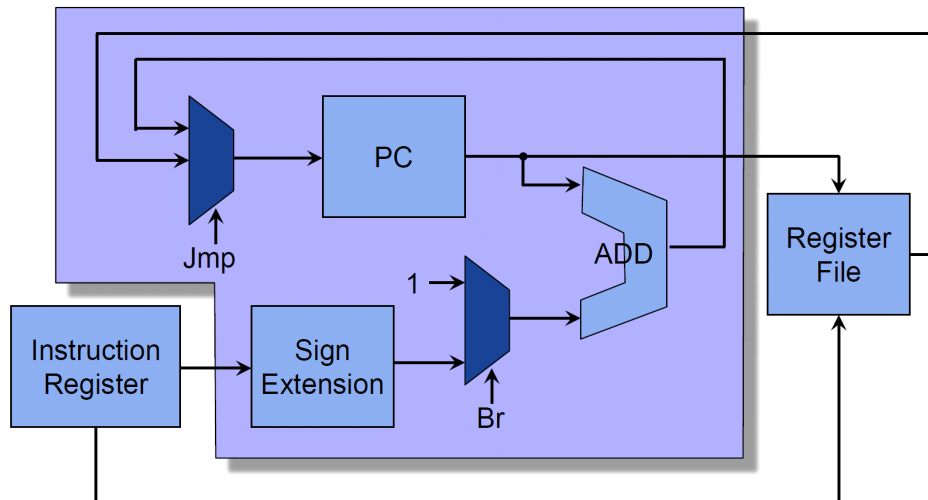
13~19행: net 및 variable 선언

- wire [15:0] disp_signEx_c; //입력된 displacement과 current PC를 더하기 위해 signed extension 한 값
- wire [15:0] addr_incr_c; //next PC를 구하기 위해 current PC에 가산할 값
- wire [15:0] next_addr_imem_w; //calculated next PC
- reg [15:0] curr_addr_imem_r; //stored currnt PC
- wire dummy_co_w; //cla port에 wire할 dummy net
- wire [1:0] dummy_fc_w; //cla port에 wire할 dummy net

```
21 wire dummy_co_w;
22 wire [1:0] dummy_fc_w;
23
24 assign disp_signEx_c[15:8] = {8{displacement_i[7]}};
25 assign disp_signEx_c[7:0] = displacement_i;
26 assign addr_incr_c = (branch_i == 1'b0) ? 16'b1 : disp_signEx_c; //branch아니면 1증가
27
28 assign addr_imem_o = curr_addr_imem_r;
29
30 cla16
31     cla16_0(      You, 2 months ago • init
32         .A_i(addr_incr_c),
33         .B_i(curr_addr_imem_r),
34         .CARRYIN_i(1'b0),
35         .CARRYOUT_no(dummy_co_w),
36         .flag_overflow_o(dummy_fc_w),
37         .sum_o(next_addr_imem_w)
38     );
```

- disp_signEX_c: 8bit로 입력되는 변위를 cla16에 피연산자로 넣기 위해 MSB 8bit를 앞쪽에 padding

- `addr_incr_c`: next PC를 연산하기 위한 증감분으로 branch 명령이면 변위를 일반 명령이면 1로 설정



PC address 연산을 위한 adder를 `cla16_0`이름으로 인스턴스화 함.

```

40  /* TODO: please write down logic for curr_addr_imem */
41  always@(posedge clk_i or posedge rst_i) begin
42      if (rst_i) curr_addr_imem_r <= 0;
43      else if (jump_i) curr_addr_imem_r <= jump_tgt_i;
44      else curr_addr_imem_r <= next_addr_imem_w;
45  end
46
47  endmodule
48

```

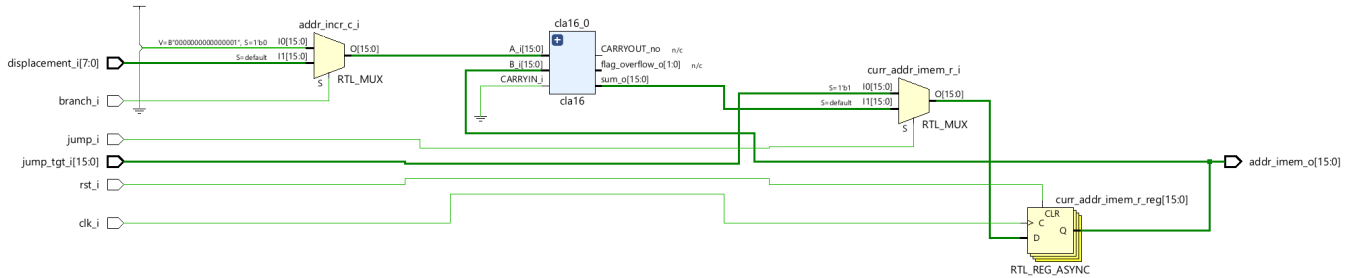
```

always@(posedge clk or negedge reset_b)
if (reset_b==0) output<=0;
else
    jmp=1 : addr_imem <= src;
    br =1 : addr_imem <= addr_imem+extend(displace)
    else : addr_imem++;

```

- Normal instruction: PC 값은 1 증가 `addr_incr_c = 1'b1`로 지정함으로써 `cla16`에서 1 더해진 값 출력
 - Conditional branch: `addr_incr_c` = displacement로 지정하여 변위만큼 증가한 PC 출력
 - Conditional jump, JAL: PC 출력을 target 주소로 업데이트
- 비동기 reset을 지원하고, 이 경우 현재 PC값을 0으로 초기화 한다.
- jump명령의 경우 입력되는 jump target address를 직접 PC값에 대입하기 때문에 조건문을 이용하였고, 나머지 branch 명령과 normal 명령은 조건에 따라 24~26행에서 계산되어 있으므로 위와같이 구현하였다.

3.1.2. Schematic



<pre> 24 assign disp_signEx_c[15:8] = {8{displacement_i[7]}}; 25 assign disp_signEx_c[7:0] = displacement_i; 26 assign addr_incr_c = (branch_i == 1'b0) ? 16'b1 : disp_signEx_c; </pre> <p>작성한 코드처럼 branch 명령을 표시하는 branch_i 입력을 select로 하는 2X1MUX가 현재 주소에 더해질 양을 선택한다.</p>	<p>MUX에서 선택된 addr_incr와 현재 주소를 앞서 설계한 look ahead adder를 통해 계산한다.</p>
<pre> 40 /* TODO: please write down logic for curr_addr_imem */ 41 always@(posedge clk_i or posedge rst_i) begin 42 if (rst_i) curr_addr_imem_r <= 0; 43 else if (jump_i) curr_addr_imem_r <= jump_tgt_i; 44 else curr_addr_imem_r <= next_addr_imem_w; 45 end </pre>	<p>본 모듈의 구현 내용으로 control signal에 따라 계산된 next 주소를 출력할지 입력된 target 주소를 출력할지 MUX로 선택한 뒤 현재 주소를 저장하는 레지스터에 저장한다.</p>

3.2. instruction_reg.v

3.2.1. Code Implementation

```
You, 1 hour ago | 1 author (You)
1  module instruction_reg
2  (
3      input      rst_i,
4      input      clk_i,
5      input      jump_i,
6      input      branch_i,
7      input [15:0] inst_i,
8
9      output reg [15:0] inst_o,
10     output reg [7:0] imm_o,
11     output reg [7:0] displacement_o,
12     output reg [3:0] addr_a_o,
13     output reg [3:0] addr_b_o
14 );
15
16 localparam NOP = 16'h0020;
```

포트 정의. 현재 명령이 jump, branch, normal인지 decoder로부터 입력받는다.
명령을 쪼개어 각 포트로 출력한다.

```
16 localparam NOP = 16'h0020;
17
18 always @(posedge rst_i or posedge clk_i) begin
19     /* TODO: please write down logic for each output */
20     if (rst_i) begin
21         inst_o <= 16'b0;
22         imm_o <= 8'b0;
23         displacement_o <= 8'b0;
24         addr_a_o <= 4'b0;
25         addr_b_o <= 4'b0;
26     end else begin
27         if (jump_i | branch_i) begin
28             inst_o <= NOP;
29             addr_a_o <= 4'h0;          //$rs
30             addr_b_o <= 4'h0;          //$rd      You, 1 hour ago • lab4temp
31             imm_o <= 8'h20;           //immediate
32             displacement_o <= 8'h20;   //immediate
33         end else begin
34             inst_o <= inst_i;
35             addr_a_o <= inst_i[3:0];    //$rs
36             addr_b_o <= inst_i[11:8];  //$rd
37             imm_o <= inst_i[7:0];      //immediate
38             displacement_o <= inst_i[7:0]; //immediate
39         end
40     end
41 end
42
43 endmodule
```

```

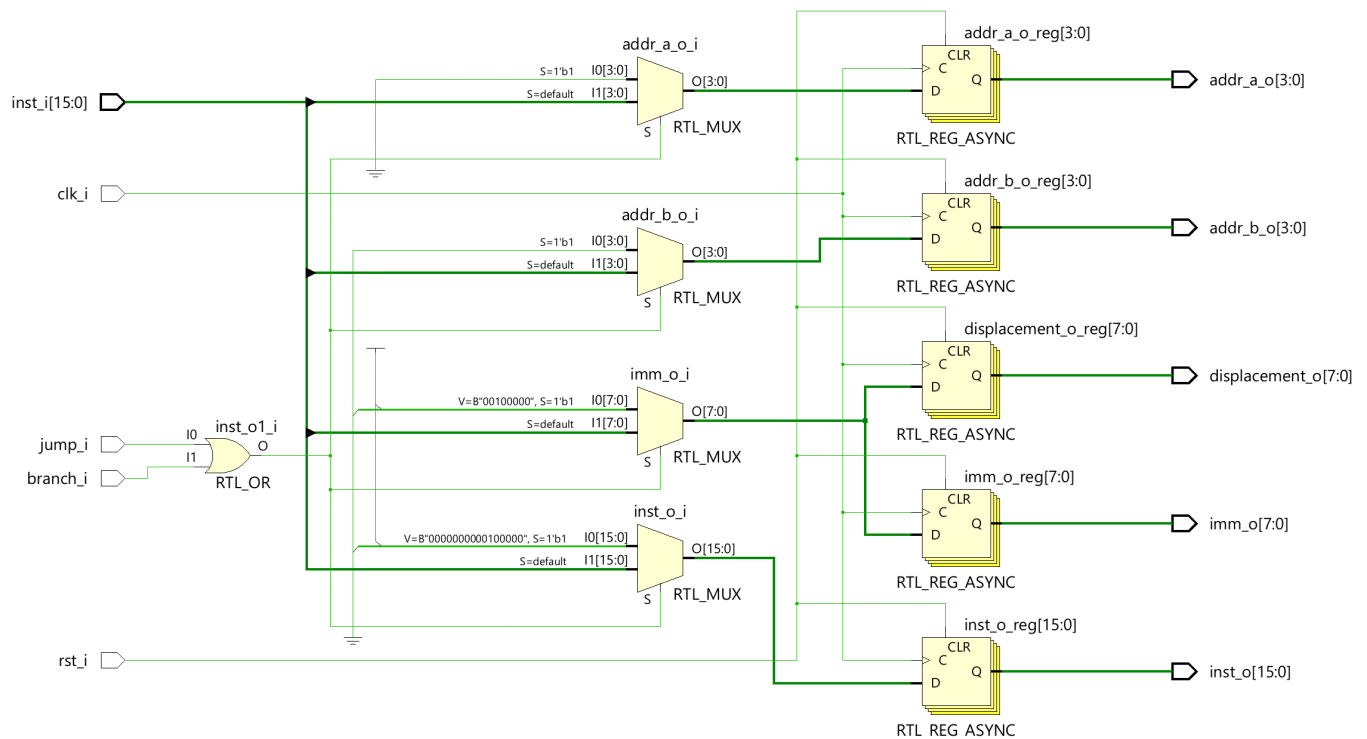
always@(posedge clk or negedge reset_b)
  if (reset_b==0)
    reset;
  else
    if (jmp==1||br==1)
      instruction_out = NOP;
    else
      instruction_out = inst_in;

  addr_a = instruction_out[3:0];
  addr_b = instruction_out[11:8];
  immediate = instruction_out[7:0];
  disp      = instruction_out[7:0];

```

reset 신호가 인가된 경우

3.2.2. Schematic



3.3. decoder.v

3.3.1. Code Implementation

```

40  function Jcond_Bcond_table;
41      input [3:0] Rdest;
42      input [4:0] psr_flags_i;    //FLCNZ
43      begin
44          case (Rdest)
45              4'b0000: Jcond_Bcond_table = psr_flags_i[0];
46              4'b0001: Jcond_Bcond_table = ~psr_flags_i[0];
47              4'b1101: Jcond_Bcond_table = psr_flags_i[1] | psr_flags_i[0];
48              4'b0010: Jcond_Bcond_table = psr_flags_i[2];
49              4'b0011: Jcond_Bcond_table = ~psr_flags_i[2];
50              4'b0100: Jcond_Bcond_table = psr_flags_i[3];
51              4'b0101: Jcond_Bcond_table = ~psr_flags_i[3];
52              4'b1010: Jcond_Bcond_table = ~psr_flags_i[3] & ~psr_flags_i[0];
53              4'b1011: Jcond_Bcond_table = psr_flags_i[3] | psr_flags_i[0];
54              4'b0110: Jcond_Bcond_table = psr_flags_i[1];
55              4'b0111: Jcond_Bcond_table = ~psr_flags_i[1];
56              4'b1000: Jcond_Bcond_table = psr_flags_i[4];
57              4'b1001: Jcond_Bcond_table = ~psr_flags_i[4];
58              4'b1100: Jcond_Bcond_table = ~psr_flags_i[1] & ~psr_flags_i[0];
59              4'b1110: Jcond_Bcond_table = 1'b1;
60              4'b1111: Jcond_Bcond_table = 1'b0;
61              default::;
62          endcase
63      end
64  endfunction

```

문제에서 주어진 table을 참고하여 각 조건별 의도한 컨트롤 신호를 출력하는 function을 만들었다. function의 입력은 Decoder table과 같이 condition에 맞는 출력을 내기 위한 Rdest와 psr flag를 받고, table의 출력을 낸다.

```

106  //////////////////////////////////////
107  // JMP set
108  //////////////////////////////////////
109  JMP = 1'b0;    // default value
110  /* TODO: write logic for JMP, you need to consider JAL and Jcond */
111  if ( (opcode == MEM_OP) && (opcode_ex == JAL_OPex) ||
112      (opcode == MEM_OP) && (opcode_ex == Jcond_OPex) )
113      begin
114          JMP = Jcond_Bcond_table(Rdest, psr_flags_i);
115      end

```

	tri_sel	alu_sel	WR	jmp	br	mux_sel0	mux_sel1	shift_imm	lui	WE	imm_ex_sel
ALUi	2	?	?	0	0	1	x	x	x	0	?
ALU	2	?	?	0	0	0	x	x	x	0	x
movi	4	x	1	0	0	1	x	x	x	0	0
mov	4	x	1	0	0	0	x	x	x	0	x
lshi	1	x	1	0	0	x	0	1	0	0	x
lsh	1	x	1	0	0	x	0	0	0	0	x
lui	1	x	1	0	0	x	1	x	1	0	x
load	16	x	1	0	0	x	x	x	x	0	x
stor	0	x	0	0	0	x	x	x	x	1	x
jal	8	x	1	1	0	x	x	x	x	0	x
jcond	0	x	0	1	0	x	x	x	x	0	x
bcond	0	x	0	0	1	x	x	x	x	0	x

JUMP 명령의 경우 위 테이블에서와 같이 JAL, Jcond인 경우에 assertion되어야 하므로 위와같은 조건을

추가했다.

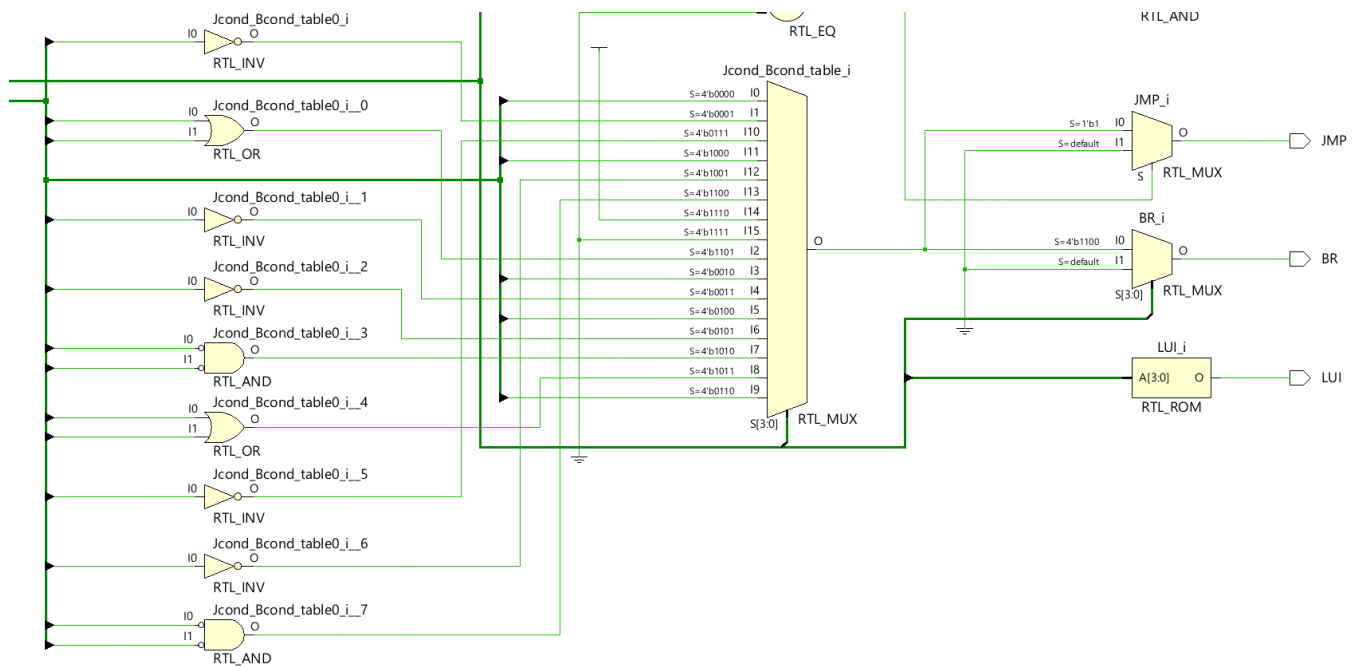
```

117 ///////////////////////////////////////////////////
118 // BR set
119 ///////////////////////////////////////////////////
120 BR = 1'b0; // default value
121 /* TODO: write logic for JMP, you need to consider JAL and Jcond */
122 if ( (opcode == Bcond_OP) )
123 begin
124     BR = Jcond_Bcond_table(Rdest, psr_flags_i);
125 end

```

Branch 제어 신호도 동일하게 Bcond 명령에서만 assert되도록 위의 조건을 추가하였다.

3.3.2. Schematic



- function으로 구현했던 부분이 10X1 MUX로 합성되었다. select입력으로 Rdest를 받고, 주어진 table에 맞게 psr flag를 조합하여 출력으로 내보낸다.

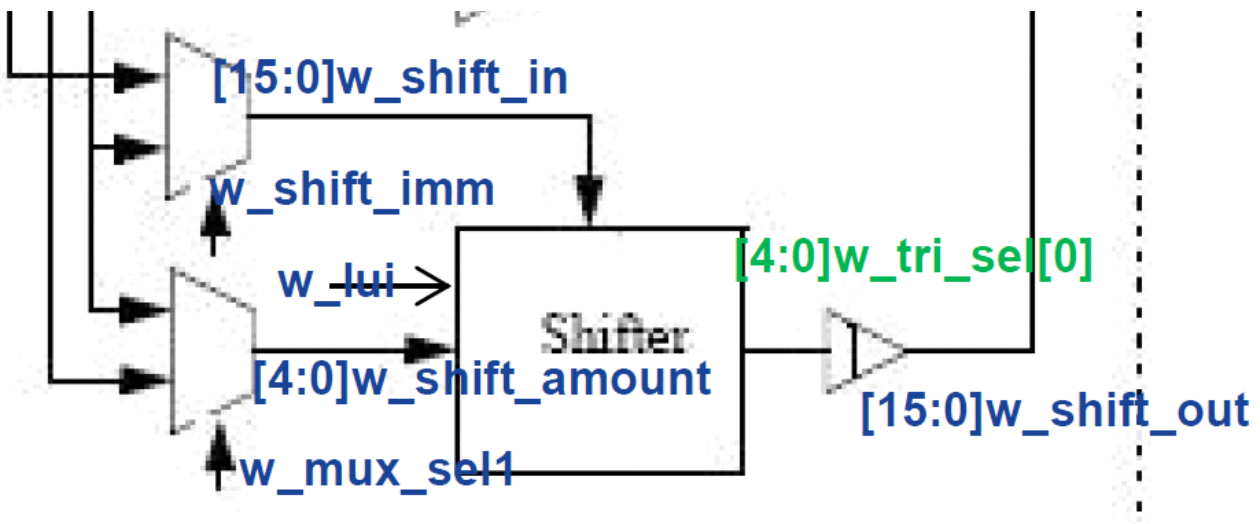
4. lab4

4.1. top_processor.v

4.1.1. code implementation

```
147  /* TODO: Please wire all components: shifter, RF, PSR, PC, IR, ALU, decoder */
148  shifter u_shift(
149  |   .data_i(MUX1_OUT),          //shift in data 16bit
150  |   .rl_shift_amt_i(MUX2_OUT),  //amount 4bit
151  |   .lui_i(LUI),
152  |   .data_o(TRI_BUF0_IN)
153  );
```

port wiring

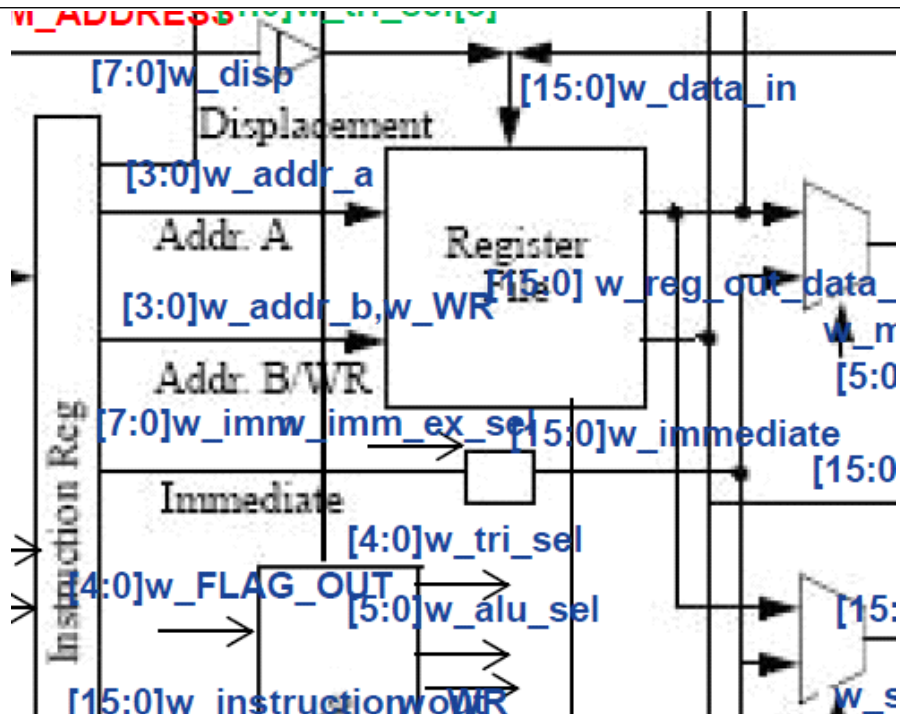


- `data_i`: shift 연산을 할 16bit 피연산 data register shift 연산과 immediate shift 연산을 모두 지원하기 위해 MUX_SEL1을 select입력으로 하는 2X1MUX의 출력을 wire
- `rl_shift_amt_i`: shift amount로, 16bit 피연산자의 shift amount를 나타내기 위해 5bit net 이용. SRC와 IMM을 2X1MUX로 select함
- `lui_i`: LUI연산을 위한 control 신호

```
35  //lui input: treat rl_shift_amt_i ad dont'care and simply shift left for 8 bits
36  assign data_o = (lui_i == 1'b1) ? {data_i[7:0], 8'h0} : stage_4;
```

- `data_o`: 연산 결과를 버스에 보내기 위한 tri state buffer로 입력

```
155  register_file u_rf(
156  |   .rst_i(RESET),
157  |   .clk_i(CLK),
158  |   .wr_i(WR),
159  |   .data_i(RF_DATA_IN),
160  |   .addr_a_i(ADDR_A),
161  |   .addr_b_i(ADDR_B),
162  |   .data_a_o(SRC),
163  |   .data_b_o(DEST)
164  );
```

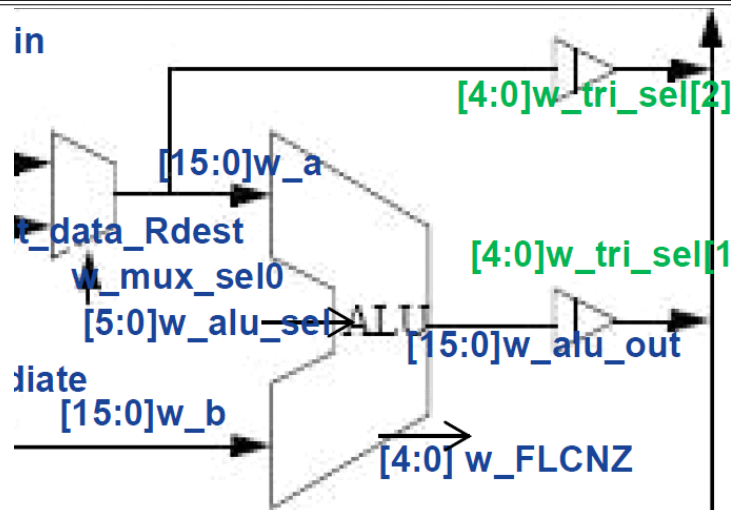


- wr_i: w_addr에 값을 쓰기 위한 control signal
- data_i: register에 데이터를 저장하기 위한 입력. tri buffer가 연결된 bus와 연결되어 있음
- addr_a_i, addr_b_i: Instruction Reg로부터 Fetch
- data_a_o: SRC 출력; jump 주소, memory addr, shift amount, adder 피연산자로 입력될 버스
- data_b_o: Rdest 출력; shifter 피연산자, adder 피연산자, memory에 저장할 데이터 출력 버스

```

166   alu u_alu(
167       .A_i(MUX0_OUT),
168       .B_i(DEST),
169       .alu_sel_i(ALU_SEL),
170       .flag_o(PSR_FLCNZ_IN),
171       .alu_o(TRI_BUF1_IN)
172   );

```



- A_i: src 또는 imm을 선택하는 MUX의 output
- B_i: 피연산자 2
- alu_sel_i: ALU 연산을 선택
- flag_o: overflow flag. processor state결정에 이용됨
- alu_o: 연산 결과 출력

```

174  psr u_psr(
175      .rst_i(RESET),
176      .clk_i(CLK),
177      .alu_sel_i(ALU_SEL),
178      .flags_i(PSR_FLCNZ_IN),
179      .flags_o(PSR_FLCNZ_OUT)
180  );

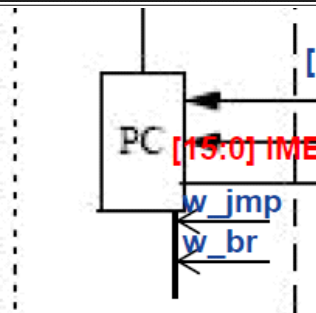
```

- flags_o: flag로부터 각 연산에 맞는 flag 출력 register

```

182  pc u_pc(
183      .rst_i(RESET),
184      .clk_i(CLK),
185      .jump_i(JMP),
186      .branch_i(BR),
187      .displacement_i(DISP),
188      .jump_tgt_i(SRC),
189      .addr_imem_o(PC_OUT)
190  );

```

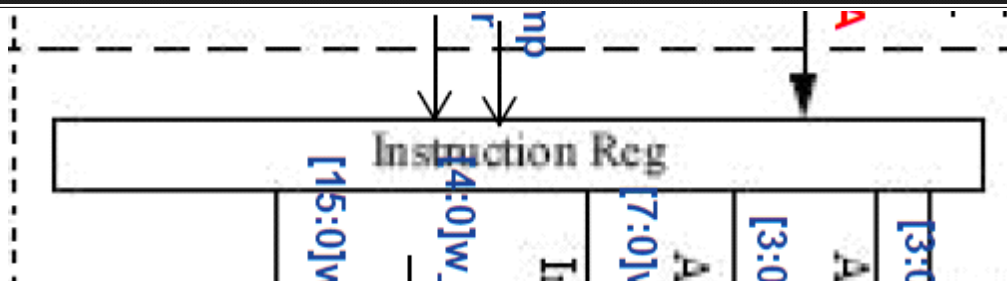


- jump_i: jump instruction에서 assert되는 control signal. jump target으로 PC값을 업데이트 한다.
- branch_i: control signal asserted when branch instruction. 현재 PC값에 branch address를 더해 출력하게 된다. 이때 displacement값이 이용된다.
- displacement_i: branch 되는 주소값의 변위
- jump_tgt_i: jump 명령으로 이동할 주소
- addr_imem_o: Instruction Memory로부터 읽어올 현재 PC address

```

192  instruction_reg u_ir(
193      .rst_i(RESET),
194      .clk_i(CLK),
195      .jump_i(JMP),
196      .branch_i(BR),
197      .inst_i(IMEM_DATA),
198      .inst_o(INST_REG_OUT),
199      .imm_o(IMM),
200      .displacement_o(DISP),
201      .addr_a_o(ADDR_A),
202      .addr_b_o(ADDR_B)
203  );

```



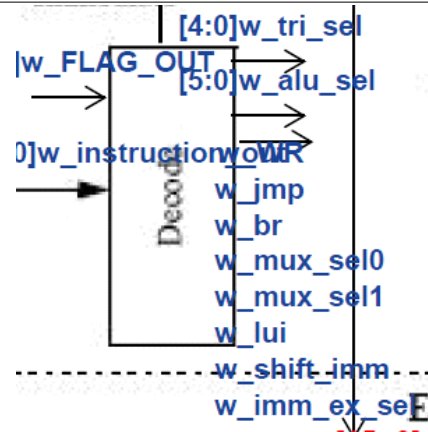
- jump_i, branch_i: jump, branch 명령의 경우 다음 PC instruction이 실행되면 안되므로, NOP를 pipe-line에 넣기 위한 signal
- inst_i: Instruction Memory로부터 읽은 Instruction Fetch

- inst_o: control signal을 생성하기 위해 입력된 instruction을 출력. timing을 맞추기 위함
- imm_o, displacement, addr_a, addr_b: instruction을 파싱하여 Fetch하기위한 준비를 함

```

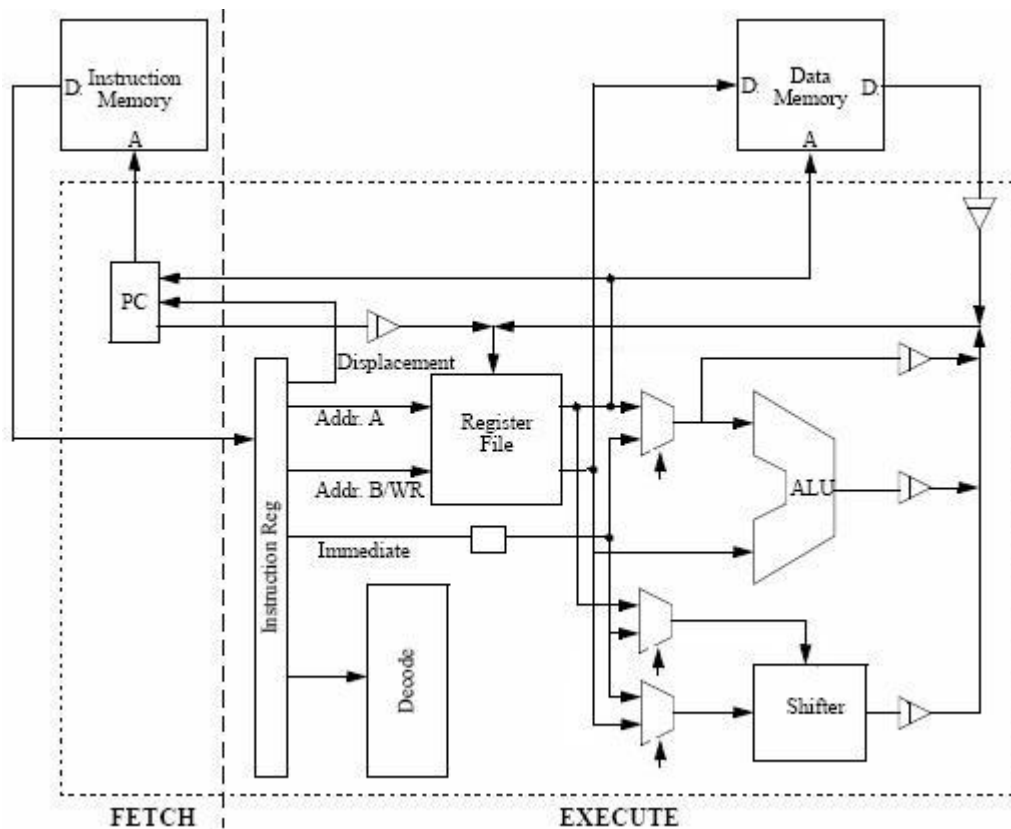
205 decoder u_id(
206     .inst_i(INST_REG_OUT),
207     .psr_flags_i(PSR_FLCNZ_OUT),
208     .TRI_SEL(TRI_SEL),
209     .ALU_SEL(ALU_SEL),
210     .WR(WR),
211     .JMP(JMP),
212     .BR(BR),
213     .IMM_EX_SEL(IMM_EX_SEL),
214     .MUX_SEL0(MUX_SEL0),      //Data input control of ALU
215     .MUX_SEL1(MUX_SEL1),      //Data input control of Shifter
216     .SHIFT_IMM(SHIFT_IMM),    //Data input control of Shifter
217     .LUI(LUI),
218     .WE(WE)
219 );
220 endmodule

```

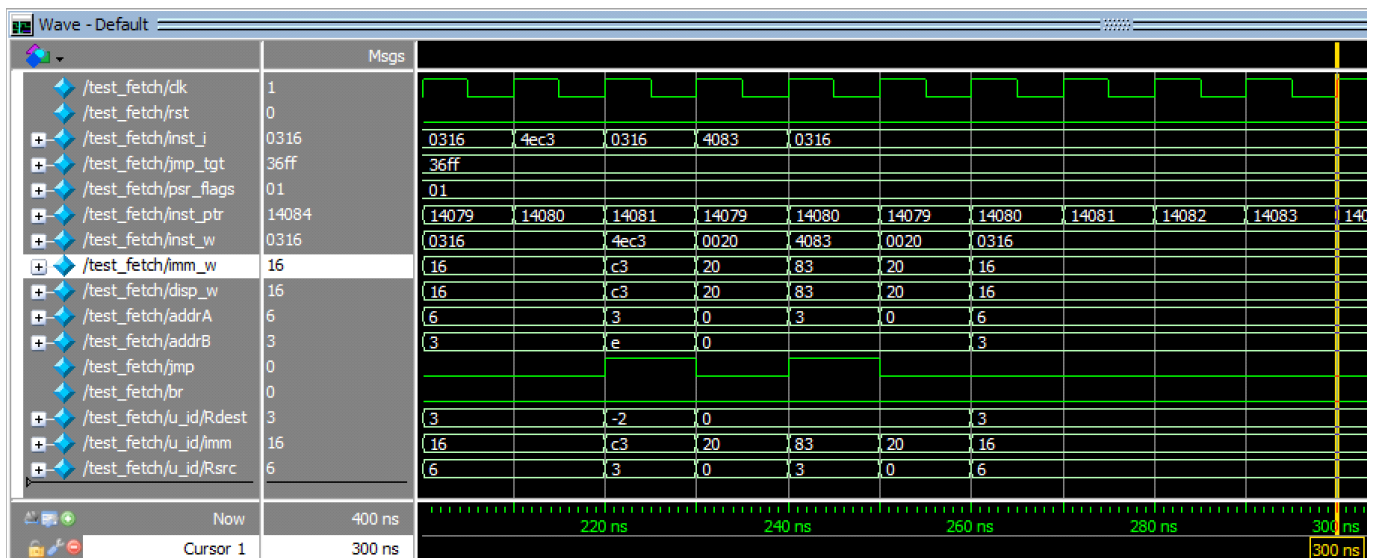
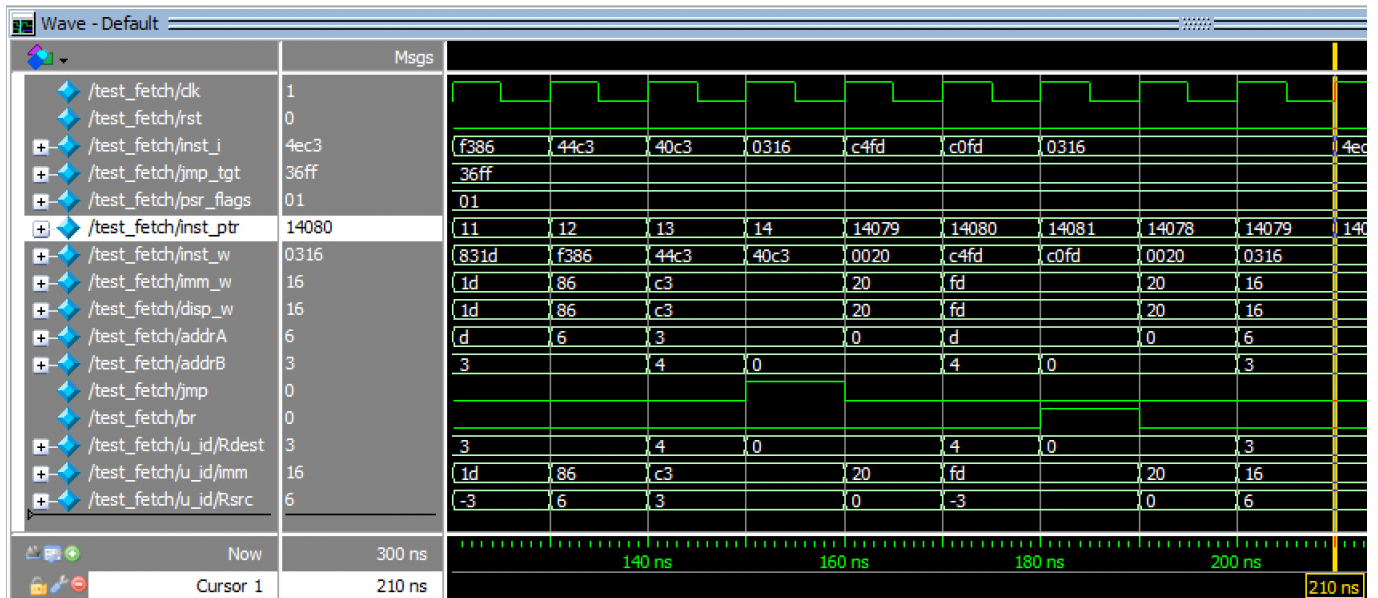
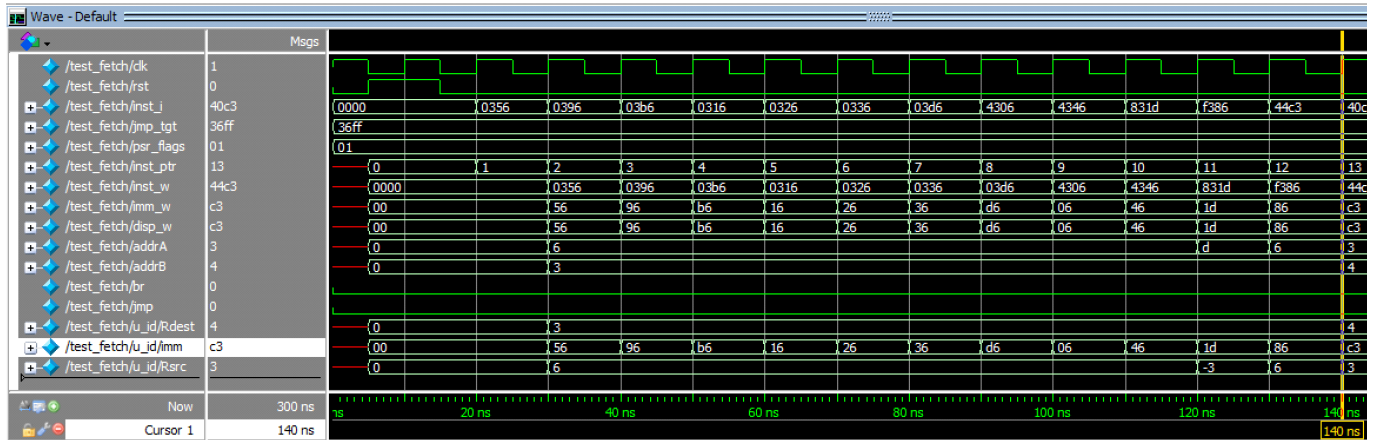


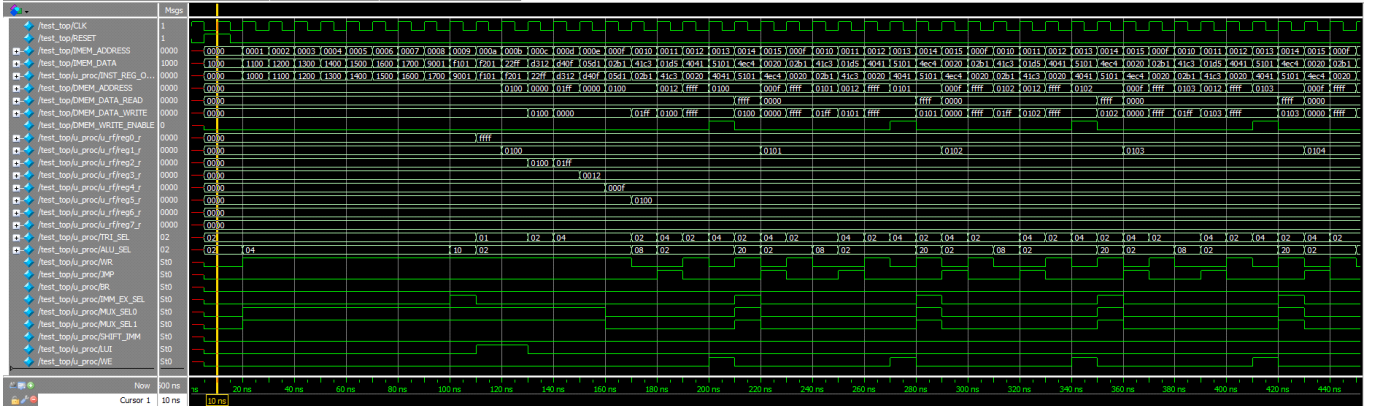
instruction을 디코딩하여 제어 신호를 출력한다.

4.1.2. schematic




4.1.3. wave form





4.2. Translate binary in imem_imginto human-readable assembly

	num	binary				assembly	
		op_code	R_dest	ex_op	R_src		
				Immediate			
	1	0001	0000	0000	0000	ANDI \$r0 8'b0	//r0 = 0
	2	0001	0001	0000	0000	ANDI \$r1 8'b0	//r1 = 0
	3	0001	0010	0000	0000	ANDI \$r2 8'b0	//r2 = 0
	4	0001	0011	0000	0000	ANDI \$r3 8'b0	//r3 = 0
	5	0001	0100	0000	0000	ANDI \$r4 8'b0	//r4 = 0
	6	0001	0101	0000	0000	ANDI \$r5 8'b0	//r5 = 0
	7	0001	0110	0000	0000	ANDI \$r6 8'b0	//r6 = 0
	8	0001	0111	0000	0000	ANDI \$r7 8'b0	//r7 = 0
	9	1001	0000	0000	0001	SUBI \$r0 8'b1	//r0 = 16'hFFFF
	10	1111	0001	0000	0001	LUI \$r1 8'b1	//r1 = 8'b1 << 8 = 16'h100 = 256
	11	1111	0010	0000	0001	LUI \$r2 8'b1	//r2 = 8'b1 << 8 = 16'h100 = 256
	12	0010	0010	1111	1111	ORI \$r2 8'hFF	//r2 = (16'h100 16'hFFFF) = 16'hFFFF
	13	1101	0011	0001	0010	MOVI \$r3 8'h12	//r3 = 16'h12
	14	1101	0100	0000	1111	MOVI \$r4 8'h0F	//r4 = 16'h0F
	15	0000	0101	1101	0001	MOV \$r5 \$r1	//r5 = r1 = 16'h100 = 256
	16	0000	0010	1011	0001	CMP \$r2 \$r1	//r2==r1(16'hFFFF==16'h100), PSR=xxx0
	17	0100	0001	1100	0011	(Jcond) NEQ \$r3	//JUMP to 0x0012 (10)18
	18	0000	0001	1101	0101	MOV \$r1 \$r5	//r1 = r5 = 16'h100 = 256
	19	0100	0000	0100	0001	STORE \$r0 \$r1	//r0<-\$r1 (0xFFFF) <- 6'h100 = 256
	20	0101	0001	0000	0001	ADDI \$r1 8'h01	//r1 = 16'h100+16'h001 = 16'h101
	21	0100	1110	1100	0100	(Jcond) JMP \$r4	//just JUMP to 0x000F