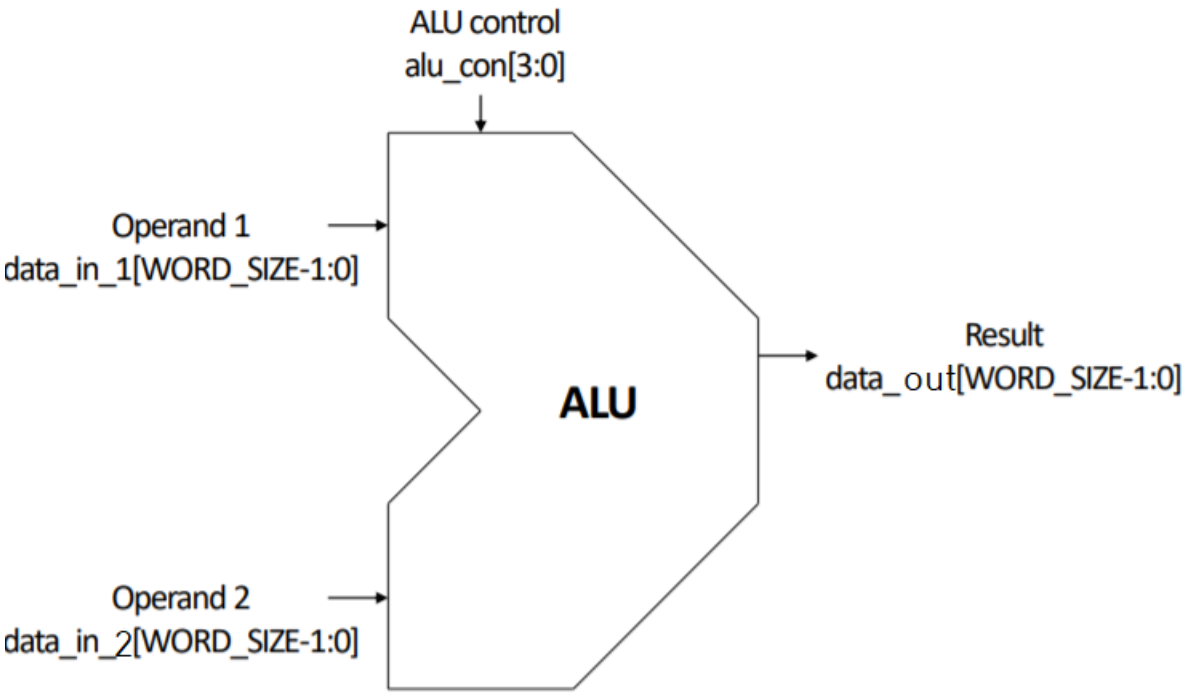
디지털시스템설계 LAB1

2016310936 우승민

이번 과제의 목표는 주어진 **logic gate** 인 **AND, OR, XOR** 을 이용하여 기본적인 ALU 인 **Bitwise OR / AND / NOR** 과 **Signed Addition, Signed Subtraction** 을 구현하는 것입니다.



Opcode	Instruction
0010	Signed Addition
0110	Signed Subtraction
0000	Bitwise AND
0001	Bitwise OR
1111	Bitwise NOR

우선 저는 **Opcode** 에 따라 결과값인 **data_out** 이 정해지도록 아래와 같이 assign 문을 사용해 주었습니다. **NOR** 일 경우는 **OR** 의 결과값을 **NOT** 해주기만 하면 됩니다.

```
localparam ADD = 4'b0010,
           SUB = 4'b0110,
           AND = 4'b0000,
           OR  = 4'b0001,
           NOR = 4'b1111;
```

```
assign data_out = (alu_con==ADD)? S0 :
                  (alu_con==SUB)? S1 :
                  (alu_con==AND)? A0 :
                  (alu_con==OR)? A1 :
                  (alu_con==NOR)? ~A1 : 0;
```

여기서 **S0, S1** 등 결과로 넣어줄 값들은 작성한 코드 상단부에 **WORD_SIZE** 크기에 맞도록 wire 로 선언해 주었습니다.

```
////////////////////////////////////////////////////////
//////////////write your code here//////////////////////
wire [WORD_SIZE-1   :0] A0, A1, S0, S1, N0, N1, M0, M1, C0, C1, k, A2;

assign k = ~data_in_2;
```

여기서 **k** 는 나중에 **subtraction** 을 할 때 덧셈과 같은 방식으로 만들되 **2's complement** 를 사용하여 구현하기 위해 사용하였습니다.

이제 다음으로 결과값을 만들어주는 코드를 살펴보겠습니다. 우선 가장 간단한 **AND** 와 **OR** 을 보면

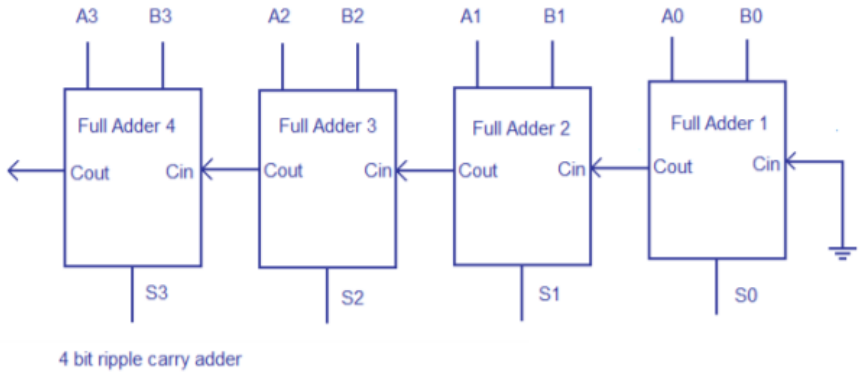
```
genvar idx;
for (idx = 0; idx <WORD_SIZE; idx = idx + 1)
        begin
        logic_and x0(.data_in_1(data_in_1[idx]), .data_in_2(data_in_2[idx]), .data_out(A0[idx]));
        end

for (idx = 0; idx <WORD_SIZE; idx = idx + 1)
        begin
        logic_or y0(.data_in_1(data_in_1[idx]), .data_in_2(data_in_2[idx]), .data_out(A1[idx]));
        end
```

Generate for 문을 사용하여 최하위 비트부터 최상위 비트까지 loop 를 돌며 각각 A0 와 A1 에 입력되도록 만들어 주었습니다.

NOR 문은 위에서 설명했듯이 **OR** 결과값에 **NOT** 을 붙이면 되므로 따로 만들지는 않았습니다.

다음으로 Addition 코드를 살펴보기 전에 과제의 기준인 **Ripple carry adder** 는 아래와 같은 형태이고



4 bit ripple carry adder

이 형태에 맞는 **Boolean equation** 은 아래와 같습니다. 여기서 C_{in}은 이전 비트의 C_{out} 입니다.

$$S = Cin \oplus A \oplus B$$

$$Cout = AB + Cin(A \oplus B)$$

이제 **Boolean equation** 에 맞추어 구현한 코드를 보면

```
for (idx = 0; idx <WORD_SIZE; idx = idx + 1)
        begin
        logic_xor a0(.data_in_1(data_in_1[idx]), .data_in_2(data_in_2[idx]), .data_out(N0[idx]));
        end
```

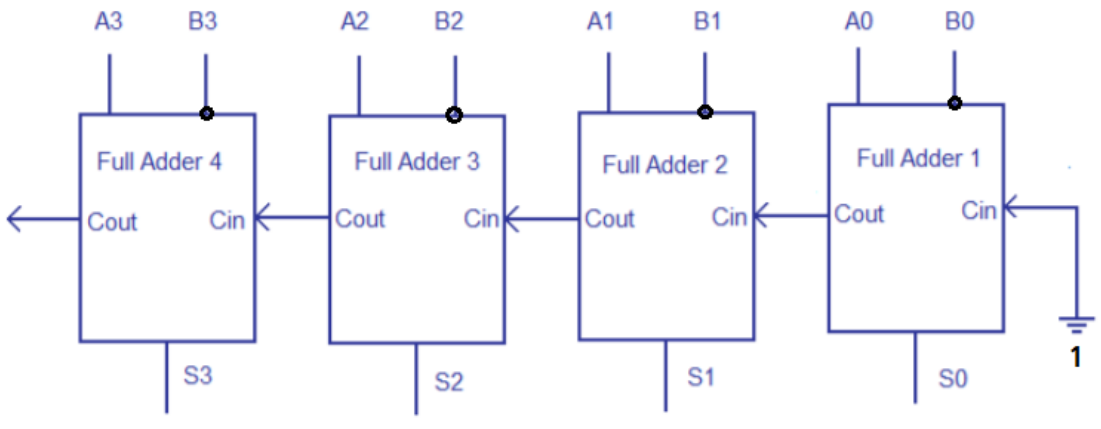
우선 **N0** 라는 wire 에 **data_in_1** 과 **data_in_2** 의 **XOR** 한 값을 넣어주었습니다. **data_in_1** 과 **data_in_2** 의 **AND** 의 결과값은 이전에 **A0** 에 넣어주었기 때문에 그대로 사용하였습니다.

이후의 코드는 같은 for 문안에서 한번에 구현하였는데

```
for (idx = 0; idx <WORD_SIZE; idx = idx + 1)
        begin
        if(idx ==0)
             begin
             logic_and b0(.data_in_1(N0[idx]), .data_in_2(0), .data_out(M0[idx]));
             logic_xor c0(.data_in_1(N0[idx]), .data_in_2(0), .data_out(S0[idx]));
             logic_or d0(.data_in_1(A0[idx]), .data_in_2(M0[idx]), .data_out(C0[idx+1]));
             end
        else if(idx == WORD_SIZE-1)
             begin
             logic_and b0(.data_in_1(N0[idx]), .data_in_2(C0[idx]), .data_out(M0[idx]));
             logic_xor c0(.data_in_1(N0[idx]), .data_in_2(C0[idx]), .data_out(S0[idx]));
             end
        else
             begin
             logic_and b0(.data_in_1(N0[idx]), .data_in_2(C0[idx]), .data_out(M0[idx]));
             logic_xor c0(.data_in_1(N0[idx]), .data_in_2(C0[idx]), .data_out(S0[idx]));
             logic_or d0(.data_in_1(A0[idx]), .data_in_2(M0[idx]), .data_out(C0[idx+1]));
             end
        end
```

이유는 이전 C_{out} 의 값이 다음 **SUM** 과 C_{in}에 계속 입력되어야 하므로, 순차적으로 실행되게 하기 위한 목적입니다. For 문 안의 if 문은 처음에 C_{in} 값은 없으므로 **0** 으로 넣어주어야 하고, 마지막 bit 의 결과에서 나온 C_{out}은 **WORD_SIZE** 의 bit 를 넘어서기 때문입니다.

Subtraction 의 코드는 **Addition** 과 동일한 방법을 사용하되, **data_in_2** 의 값을 **NOT** 한 **k** 값을 사용해 주었고, 첫 비트의 C_{in} 의 값을 **1** 로 해주었습니다.



4 bit ripple carry adder

Addition 에서는 이전에 **data_in_1** 과 **data_in_2** 의 **AND** 값이 있었지만, 이번에는 ~data_in_2 과 해주어야 하므로 for 문에 추가해 주었습니다.

```
assign k = ~data_in_2;
```

이전에 **k** 에 ~data_in_2 값을 넣어주었기 때문에 **k** 를 사용하였습니다.

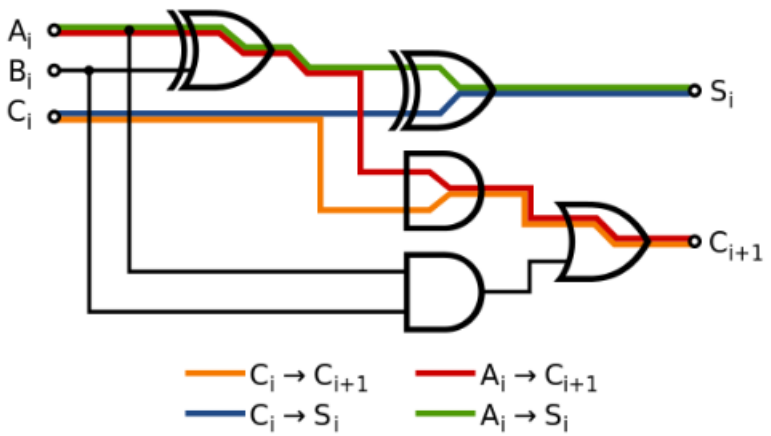
```
for (idx = 0; idx <WORD_SIZE; idx = idx + 1)
        begin
        logic_xor aa0(.data_in_1(data_in_1[idx]),.data_in_2(k[idx]),.data_out(N1[idx]));
        logic_and e0(.data_in_1(data_in_1[idx]), .data_in_2(k[idx]), .data_out(A2[idx]));
        end

for (idx = 0; idx <WORD_SIZE; idx = idx + 1)
        begin
        if(idx ==0)
              begin
              logic_and f0(.data_in_1(N1[idx]), .data_in_2(1), .data_out(M1[idx]));
              logic_xor g0(.data_in_1(N1[idx]), .data_in_2(1), .data_out(S1[idx]));
              logic_or h0(.data_in_1(A2[idx]), .data_in_2(M1[idx]), .data_out(C1[idx+1]));
              end
        else if(idx == WORD_SIZE-1)
              begin
              logic_and f0(.data_in_1(N1[idx]), .data_in_2(C1[idx]), .data_out(M1[idx]));
              logic_xor g0(.data_in_1(N1[idx]), .data_in_2(C1[idx]), .data_out(S1[idx]));
              end
        else
              begin
              logic_and f0(.data_in_1(N1[idx]), .data_in_2(C1[idx]), .data_out(M1[idx]));
              logic_xor g0(.data_in_1(N1[idx]), .data_in_2(C1[idx]), .data_out(S1[idx]));
              logic_or h0(.data_in_1(A2[idx]), .data_in_2(M1[idx]), .data_out(C1[idx+1]));
              end
        end
```

이 후는 **Addition** 과 같은 방식으로 만들어 주었습니다.

제 코드에서 **critical path** 를 찾으면, 일단 AND, OR, NOR 보다는 **ADD** 와 **SUB** 가 훨씬 많은 게이트를 지나기 때문에 **ADD** 와 **SUB** 으로 생각하고, 과제에서 NOT gate 의 delay 가 주어지지 않았음으로 **ADD** 와 **SUB** 이 동일하게 **critical path** 를 가지게 될 것입니다.

1 bit **ADD** 로 살펴보면 아래 그림에서 **critical path** 는 300ns delay 가 될 것입니다. (XOR + XOR 혹은 XOR + AND + OR)



이후로는 여기서 나온 C_{out}값이 다음 **Adder**로 들어가 150ns delay가 추가될 것입니다. (SUM으로 가는 XOR 혹은 C_{in}으로 가는 AND + OR)

위의 과정이 bit 마다 반복될 것이고, 따라서 전체의 **Critical path** 의 delay 는

150(WORD_SIZE + 1)ns 가 될 것입니다.