SoC설계

Lab#12

Arithmetic Logic Unit

컴퓨터공학과

201402439

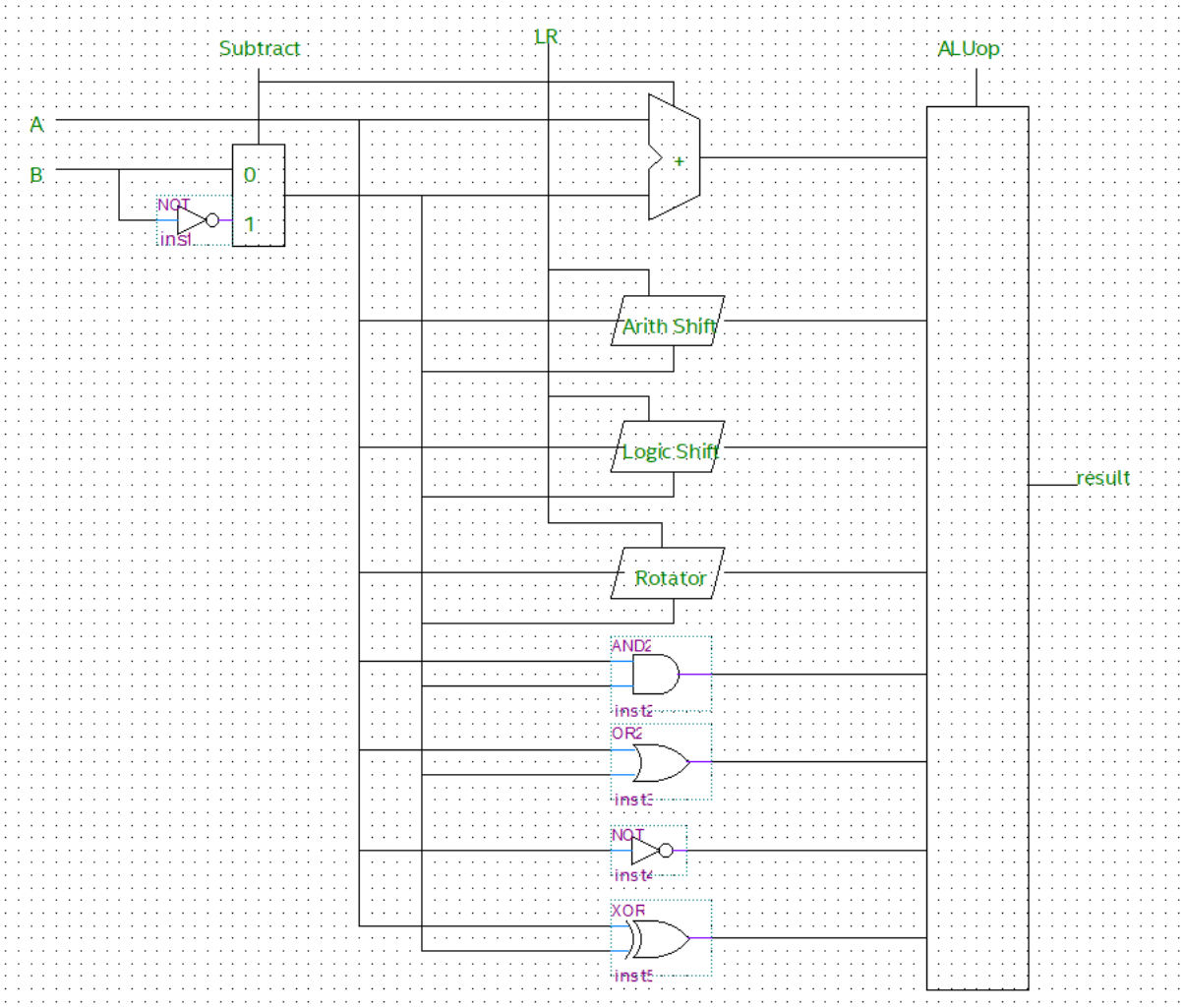
천원준

1. Purpose of the lab

이번 실습의 목표는 이전 주차에 만들었던 산술 연산 모듈들과 논리게이트를 이용하여 종합적인 연산을 수행하는 ALU를 설계하는 것입니다.

2. Design Procedure

- Block Diagram



3. Simulation

- Logical Shifter

```
module LShifter(shift, lr, in, out);
//lr? 1 == left, 0 == right
//logical shifter

   input [3:0] shift;
   input lr;
   input [15:0] in;

   output [15:0] out;

   wire [15:0] st1, st2, st3;

   assign st1 = shift[0]? (lr? {in[14:0], 1'b0}:{1'b0, in[15:1]}) : in[15:0];
   assign st2 = shift[1]? (lr? {st1[13:0], 2'b00}:{2'b00, st1[15:2]}) : st1[15:0];
   assign st3 = shift[2]? (lr? {st2[11:0], 4'b0000}:{4'b0000, st2[15:4]}) : st2[15:0];
   assign out = shift[3]?
   (lr? {st3[7:0], 8'b00000000}:{8'b00000000, st3[15:8]}) : st3[15:0];

endmodule
```

기존의 Arithmetic Shifter 모듈을 일부 수정하여 Logical Shifter를 만들었습니다.

Shift 연산 후 빈 공간을 0으로 채웁니다.

- ALU

```
module ALU(valA, valB, aluop, sub, lr, cc, result);

   input [15:0] valA;
   input [15:0] valB;
   input [3:0] aluop; //need 8+ modules(3+ bit aluop)
   input sub; //Subtract or Add
   input lr; //shift Left or Right

   output [3:0] cc; //flags
   output [15:0] result;

   wire [15:0] and16b, or16b, not16b, xor16b, ashift_out, lshift_out, rotate_out, add_out, svalB, result; //svalB : subtract operand
   wire shift_co, add_co;
   wire [3:0] cc;
   wire N, Z, C, V; //condition flag

   assign and16b = valA & valB;
   assign or16b = valA | valB;
   assign not16b = ~valA;
   assign xor16b = valA ^ valB;

   Shifter arith_shifter(.shift(valB[3:0]), .lr(lr), .in(valA), .out(ashift_out)); //arithmetic shifter
   LShifter logic_shifter(.shift(valB[3:0]), .lr(lr), .in(valA), .out(lshift_out)); //logical shifter
   Rotator rotator(.shift(valB[3:0]), .lr(lr), .in(valA), .out(rotate_out));

   assign svalB = sub ? ~valB : valB;

   KSA ksadder(.A(valA), .B(svalB), .Cin(sub), .Sum(add_out), .Cout(add_co));

   assign result =
      (aluop == 4'b0000) ? add_out : //4'b0000 -> ADD, SUB
      (aluop == 4'b0001) ? ashift_out : //4'b0001 -> arithmetic shift
      (aluop == 4'b0010) ? lshift_out : //4'b0010 -> logical shift
      (aluop == 4'b0011) ? rotate_out : //4'b0011 -> rotate
      (aluop == 4'b0100) ? and16b : //4'b0100 -> AND
      (aluop == 4'b0101) ? or16b : //4'b0101 -> OR
      (aluop == 4'b0110) ? not16b : //4'b0110 -> NOT
      (aluop == 4'b0111) ? xor16b : //4'b0111 -> XOR
      16'bx;

   assign N = result[15]; //Negative(sign bit)
   assign Z = ~|result; //Zero
   assign C = (aluop == 4'b0000) & add_co; //Carry out
   assign V = ((aluop == 4'b0001) | (aluop == 4'b0010) | (aluop == 4'b0011)) ?
              lr & (valA > result) : 1'b0;
```

16비트 입력값 valA, valB와 연산을 결정하는 4비트 aluop, 뺄셈 연산을 결정하는 sub, Shift 연산을 결정하는 lr을 입력으로 받고, FLAG 값을 나타내는 4비트 cc와 연산 결과값인 16비트 result를 출력합니다.

Kogge Stone Adder, Arithmetic Shifter, Logical Shifter, Rotator 모듈을 불러와 ALU를 구성합니다.

입력이 들어오면, 각 모듈 마다 연산 결과를 내놓고, aluop값에 따라 특정 모듈의 결과값을 선택하여 최종적인 result를 결정하는 구조입니다.

ADD, SUB 연산(aluop == 4'b0000)은 2의 보수 연산을 이용하므로, 뺄셈일 경우(sub == 1'b1), valB의 값을 반전시킨 후 1을 더하는 식으로 연산합니다.

Arithmetic Shift 연산(aluop == 4'b0001)은 valA 값을 valB 만큼 lr 방향으로 산술 shift 하는 식으로 연산합니다.

Logic Shift 연산(aluop == 4'b0010), Rotate 연산(aluop == 4'b0011)도 마찬가지로 연산합니다.

AND연산(aluop == 4'b0100), OR연산(aluop == 4'b0101), XOR연산(aluop == 4'b0111)은 valA와 valB의 각 비트를 논리 연산하여 연산합니다.

NOT연산(aluop == 4'b0110)은 valA의 각 비트를 반전하여 연산합니다.

N은 결과값이 음수임을 나타내는 flag 비트입니다.

Z는 결과값이 0임을 나타내는 flag 비트입니다.

C는 결과값에 Carry가 발생했음을 나타내는 flag입니다.

V는 결과값이 overflow가 발생했음을 나타내는 flag입니다. 이번 과제에서는 shift연산의 경우만 판별하도록 설계했습니다.

```
    assign cc = {N, Z, C, V};
endmodule
```

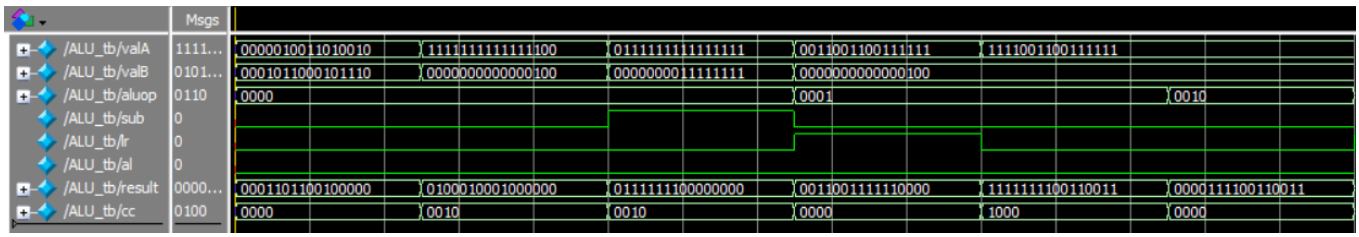
N, Z, C, V flag들을 묶어 cc 라는 출력을 만듭니다.

- ALU testbench

```
`timescale 1ns/100ps

/*
   ADD & SUB 4'b0000
   arithmetic shift 4'b0001
   logical shift 4'b0010
   rotate 4'b0011
   AND 4'b0100
   OR 4'b0101
   NOT 4'b0110
   XOR 4'b0111
*/

module ALU_tb;

   reg [15:0] valA, valB;
   reg [3:0] aluop;
   reg sub, lr, al;

   wire [15:0] result;
   wire [3:0] cc;

   ALU alu(.valA(valA), .valB(valB), .aluop(aluop), .sub(sub), .lr(lr), .cc(cc), .result(result));

   initial begin
      sub = 1'b0;
      lr = 1'b0;
      al = 1'b0;

      //ADD
      aluop = 4'b0000;
      sub = 1'b0;
      valA = 1234;
      valB = 5678;

      #50;
      //ADD(carry generation)
      aluop = 4'b0000;
      sub = 1'b0;
      valA = 16'b1111111111111100;
      valB = 16'b0000000000000100;

      #50;
      //SUB
      aluop = 4'b0000;
      sub = 1'b1;
      valA = 16'b0111111111111111;
      valB = 16'b0000000011111111;

      #50;
      //left shift
      aluop = 4'b0001;
      sub = 1'b0;
      lr = 1'b1;
      valA = 16'b0011001100111111;
      valB = 16'b0000000000000100; //4bit left shift

      #50;
      //Arithmetic right shift
      aluop = 4'b0001;
      lr = 1'b0;
      valA = 16'b1111001100111111;
      valB = 16'b0000000000000100; //4bit arithmetic right shift

      #50;
      //Logical right shift
      aluop = 4'b0010;
      lr = 1'b0;
      valA = 16'b1111001100111111;
      valB = 16'b0000000000000100; //4bit logical shift
```

```
        #50;
        //left rotate
        aluop = 4'b0011;
        lr = 1'b1;
        valA = 16'b1111001100111111;
        valB = 16'b0000000000000100;

        #50;
        //right rotate
        aluop = 4'b0011;
        lr = 1'b0;
        valA = 16'b1111001100111111;
        valB = 16'b0000000000000100;

        #50;
        //AND
        aluop = 4'b0100;
        valA = 16'b1111000011110000;
        valB = 16'b0101010101010101;

        #50;
        //OR
        aluop = 4'b0101;

        #50;
        //NOT
        aluop = 4'b0110;

        #50;
        //XOR
        aluop = 4'b0111;

        #50;
        //NOT(Zero generation)
        aluop = 4'b0110;
        valA = 16'b1111111111111111;

        #50;

        $finish;
    end
endmodule
```

다음과 같이 각 연산들을 수행할 수 있도록 testbench를 작성하였습니다.



첫 번째 결과는 일반적인 덧셈 연산입니다.

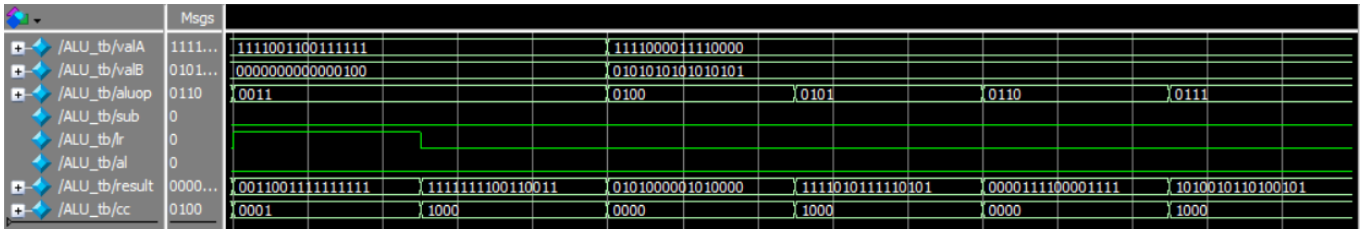
두 번째 결과는 Carry가 발생하는 덧셈 연산입니다.

세 번째 결과는 뺄셈 연산입니다.

네 번째 결과는 왼쪽 방향 산술 Shift 연산입니다.

다섯 번째 결과는 오른쪽 방향 산술 Shift 연산입니다. 음수가 발생했음을 확인할 수 있습니다.

여섯 번째 결과는 오른쪽 방향 논리 Shift 연산입니다.



일곱 번째 연산은 왼쪽 방향 Rotate 연산입니다. Overflow가 발생했음을 확인할 수 있습니다.

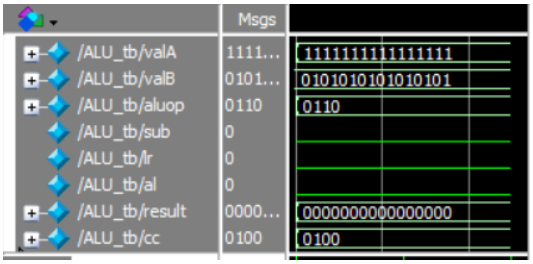
여덟 번째 연산은 오른쪽 방향 Rotate 연산입니다. 음수가 발생했음을 확인할 수 있습니다.

아홉 번째 연산은 AND 연산입니다.

열 번째 연산은 OR 연산입니다. 음수가 발생했음을 확인할 수 있습니다.

열한 번째 연산은 NOT 연산입니다.

열두 번째 연산은 XOR 연산입니다. 음수가 발생했음을 확인할 수 있습니다.



마지막 연산은 NOT 연산입니다. 0이 발생했음을 알 수 있습니다.

4. Evaluation

12가지 연산과 flag가 의도한 대로 나오는 것을 보아, 올바로 설계했음을 확인할 수 있습니다.

5. Discussion

이번 실습의 key는 각 연산들의 결과값을 aluop를 통해 선택하도록 만드는 것 입니다.

흥미로웠던 점은 이전에 만들었던 모듈들을 한 가지 모듈(ALU)로 가져와서 전부 처리할 수 있다는 점이었습니다.

이 design을 향상시킬 방법은 Multiplier 연산을 추가하고, 덧셈에 대한 Overflow flag bit 동작을 추가해주는 것입니다.