[디지털 논리회로: Simple Calculator design]

201911013 곽현우 & 201911058 박병현

1. 도입

A. 목적

- Combinational Logic, Sequential Logic 설계 능력 배양
- VHDL 설계 능력 및 Simulator를 이용한 검증 능력 배양
- Hierarchical Design, Synchronous Design에 대한 이해도 함양

B. 목표 및 기준

a) 설계 목표

- 덧셈, 뺄셈, 곱셈을 수행하는 Calculator 설계

b) 설계 기준

- 최소한의 logic을 사용해 SM chart를 바탕으로 연산 알고리즘 구현
- Top-Down 방식으로 설계하며 각 블록의 설계를 검증한 후, 전체 블록 완성
- Test Bench를 통한 Simulation 검증

C. 역할 분담

- 201911013 곽현우

  : Binary_Adder_Substractor_8bit 설계

  : calculator_top state machine 구현

  : 조교님 컨택 및 피드백

- 201911058 박병현

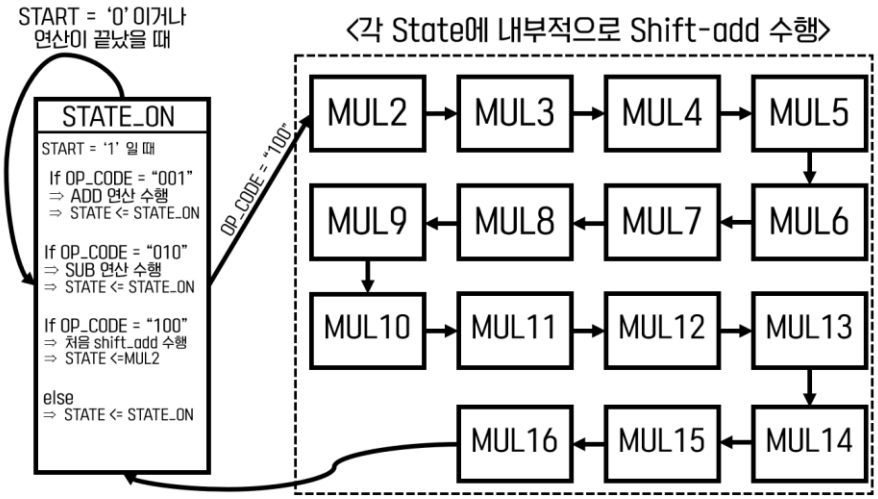
  : shift and add algorithm state machine 설계

  : multiplier 8bit RCA 설계(초기의 잘못된 접근으로 추후 수정)

  : 팀프로젝트 보고서 작성

2. 합성 및 분석

A. State Diagram

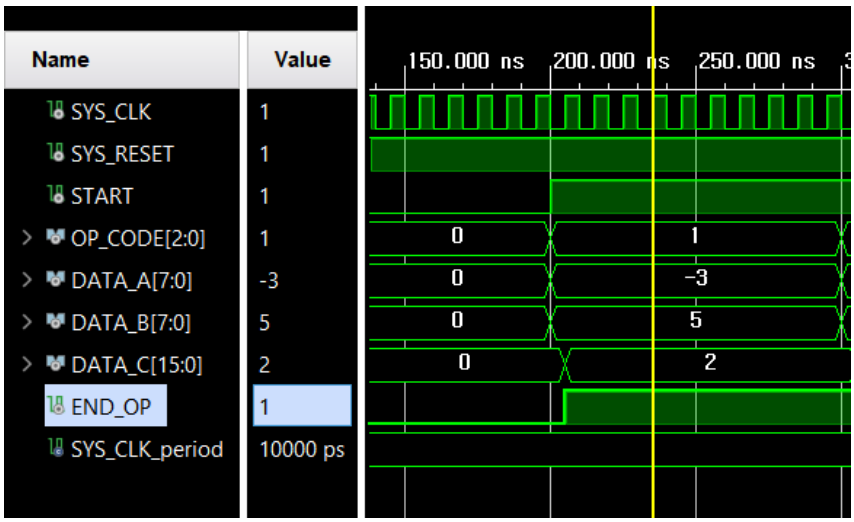


B. 시뮬레이션 모듈검증

: Test Bench를 작성해, 의도한 결과값과 시뮬레이션 결과값 비교함

1) Test1: (-3) + (+5) = +2

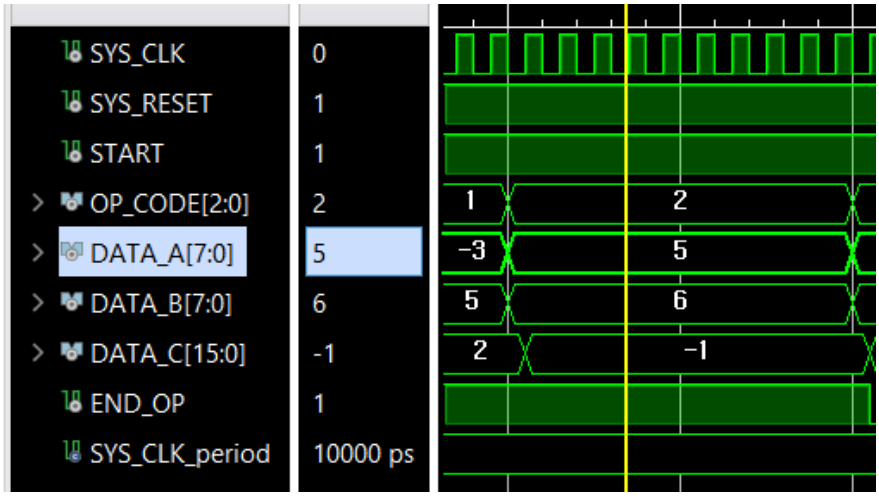
```
wait for SYS_CLK_period * 10;
    START <= '1';
    DATA_A <= "11111101"; -- -3
    DATA_B <= x"05"; -- +5
    OP_CODE <= "001";    --add : +2
```



의도한 결과(DATA_C = +2)와 시뮬레이션 결과(DATA_C = +2)가 일치함

2) Test2: (+5) – (+6) = -1

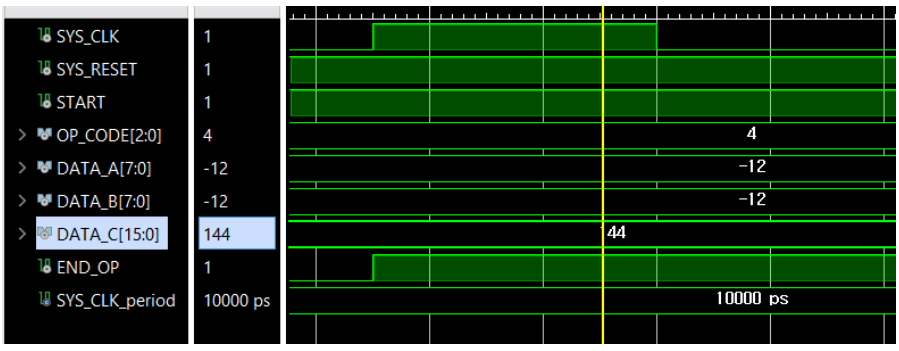
```
wait for SYS_CLK_period * 10;
    DATA_A <= x"05"; -- +5
    DATA_B <= x"06"; -- +6
    OP_CODE <= "010";   --substract : -1
```



의도한 결과(DATA_C = -1)와 시뮬레이션 결과(DATA_C = -1)가 일치함

3) Test5: (-12) * (-12) = +144

```
wait for SYS_CLK_period * 10;
    DATA_A <= "11110100";   -- -12
    DATA_B <= "11110100";   -- -12
    OP_CODE <= "100";   --mul : +144
```



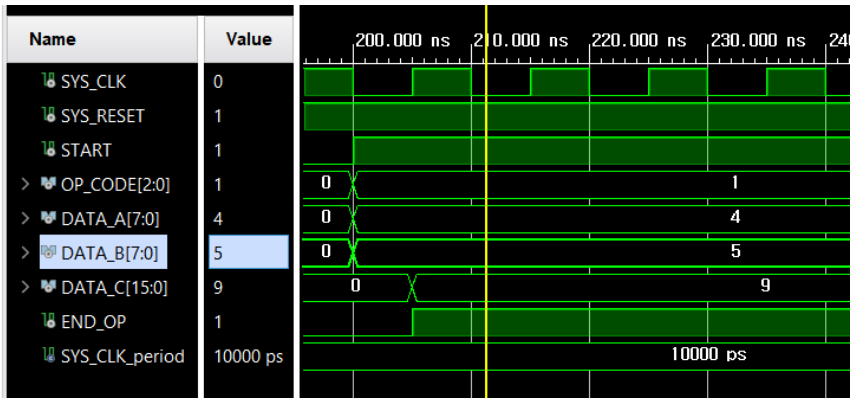
의도한 결과(DATA_C = +144)와 시뮬레이션 결과(DATA_C = +144)가 일치함

3. 결과 및 논의

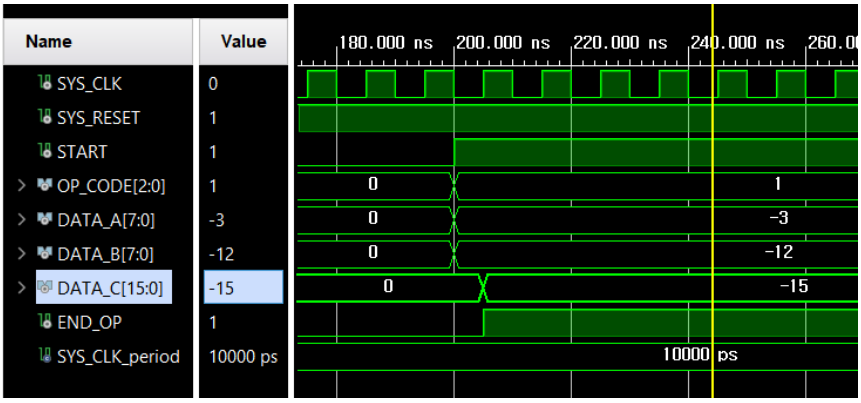
A. 결과 도출

1) ADDER

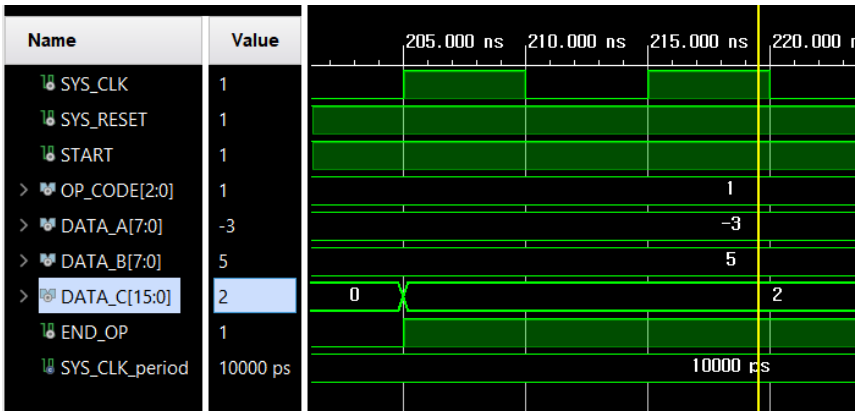
- OP_CODE <= "001"에 대해 작동함을 확인함
- 두 양수에 대한 결과 검증결과, 예측값과 결과값 일치



- 두 음수에 대한 결과 검증결과, 예측값과 결과값 일치



- 양수, 음수에 대한 결과 검증결과, 예측값과 결과값 일치

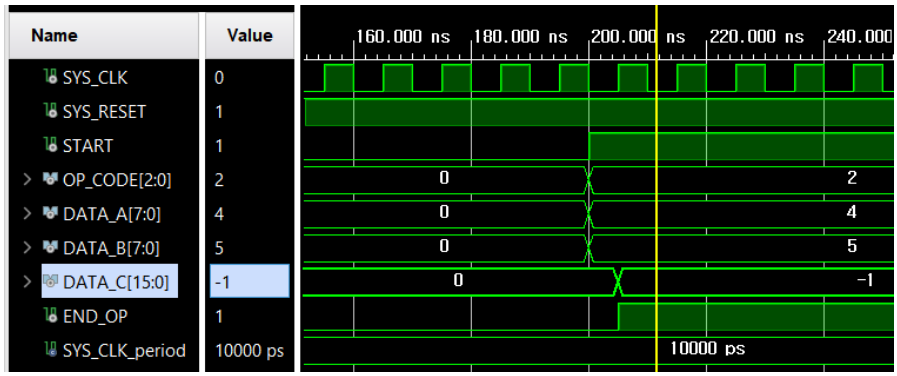


- 따라서, 잘 설계한 것으로 판단함

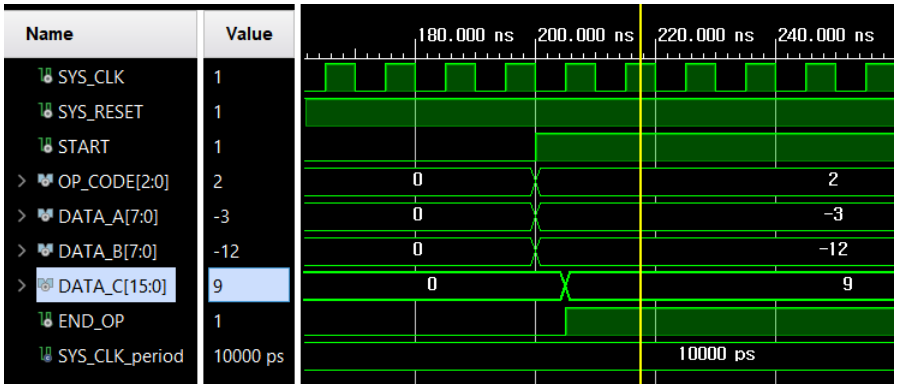
2) SUBTRACTOR

- OP_CODE <= "010"에 대해 작동함을 확인함

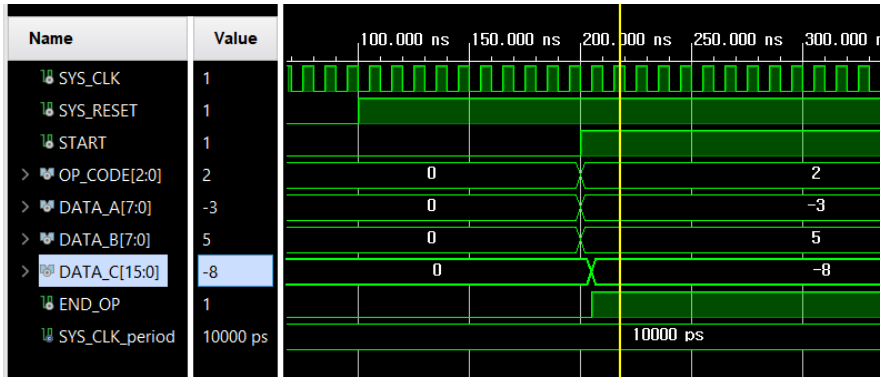
- 두 양수에 대한 결과 검증결과, 예측값과 결과값 일치



- 두 음수에 대한 결과 검증결과, 예측값과 결과값 일치



- 양수, 음수에 대한 결과 검증결과, 예측값과 결과값 일치

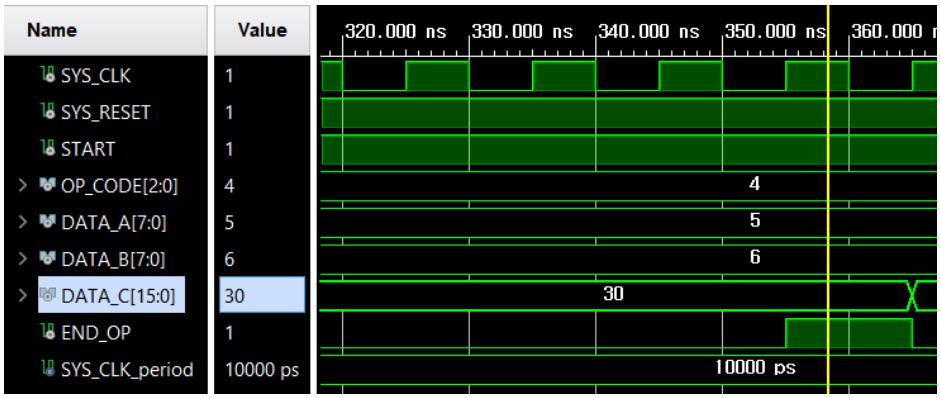


- 따라서, 잘 설계한 것으로 판단함

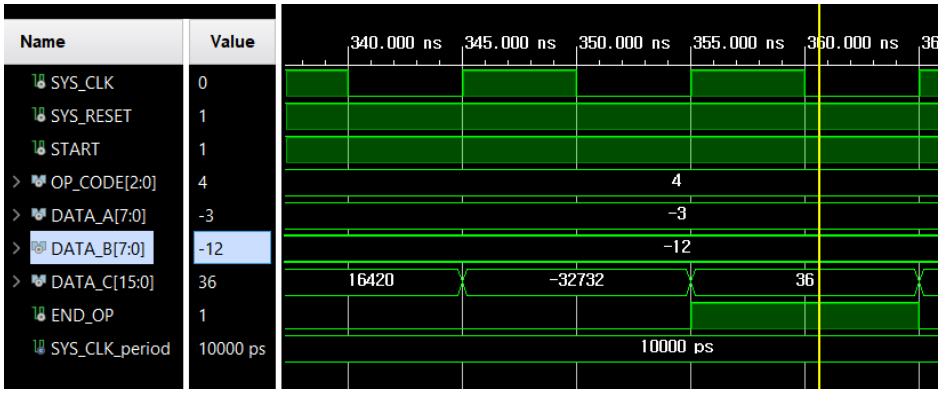
3) MULTIPLIER

- OP_CODE <= "100"에 대해 작동함을 확인함

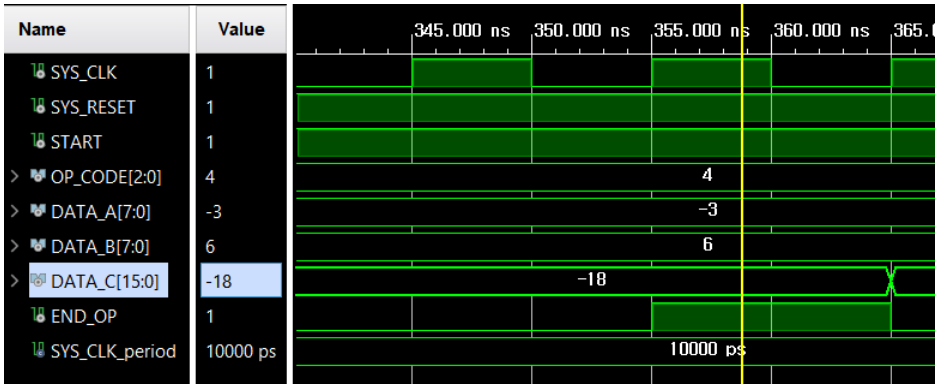
- 두 양수에 대한 결과 검증결과, 예측값과 결과값 일치



- 두 음수에 대한 결과 검증결과, 예측값과 결과값 일치



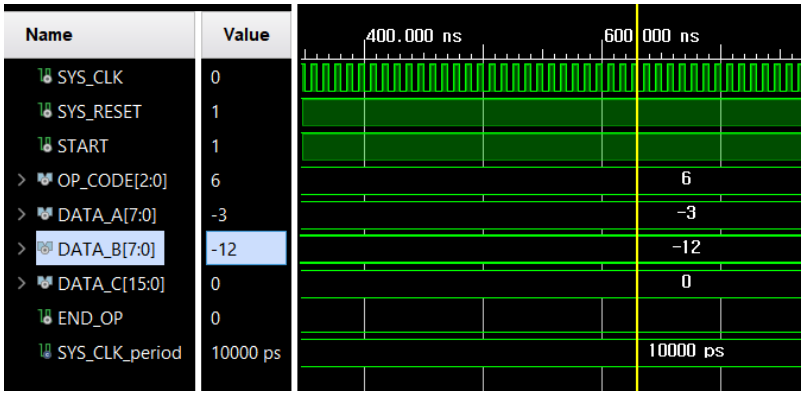
- 양수, 음수에 대한 결과 검증결과, 예측값과 결과값 일치



- 따라서, 잘 설계한 것으로 판단함

4) NONE

- OP_CODE가 "001", "010", "100"이 아닌 경우, DATA_C = 0



B. 토의

1) Design Constraints Check

a) 3 Descriptions of VHDL (Structural, Behavioral, Data Flow)

- 8bit Adder module (Data Flow Description): Full_Adder in Data Flow Description

```
entity Full_Adder is
    Port ( x : in STD_LOGIC;
           y : in STD_LOGIC;
           cin : in STD_LOGIC;
           cout : out STD_LOGIC;
           s : out STD_LOGIC);
end Full_Adder;

architecture Behavioral of Full_Adder is

begin
s <= x xor y xor cin;
cout <= (x and y) or (cin and x) or (cin and y) ;

end Behavioral;
```

조건 만족함

- 8bit ALU module (Structural Description): Binary_Adder_Substractor_8bit in Structural Description

```
entity Binary_Adder_Substractor_8bit is
    Port ( a : in STD_LOGIC_VECTOR (15 downto 0);
           b : in STD_LOGIC_VECTOR (15 downto 0);
           oper : in STD_LOGIC;
           result : out STD_LOGIC_VECTOR (15 downto 0));
end Binary_Adder_Substractor_8bit;

architecture Behavioral of Binary_Adder_Substractor_8bit is
component Full_Adder is
    Port(
    x: in std_logic;
    y: in std_logic;
    cin: in std_logic;
    s: out std_logic;
    cout: out std_logic);
end component;
```

```
-- oper가 0 이면 덧셈, 1 이면 뺄셈
signal c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15, over, b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12, b13, b14, b15, b16 : std_logic;

begin

b1 <= b(0) xor oper;
b2 <= b(1) xor oper;
b3 <= b(2) xor oper;
b4 <= b(3) xor oper;
b5 <= b(4) xor oper;
b6 <= b(5) xor oper;
b7 <= b(6) xor oper;
b8 <= b(7) xor oper;
b9 <= b(8) xor oper;
b10 <= b(9) xor oper;
b11 <= b(10) xor oper;
b12 <= b(11) xor oper;
b13 <= b(12) xor oper;
b14 <= b(13) xor oper;
b15 <= b(14) xor oper;
b16 <= b(15) xor oper;


F1:  Full_Adder port map(x => a(0), y => b1, cin => oper, s => result(0), cout => c1);
F2:  Full_Adder port map(x => a(1), y => b2, cin => c1, s => result(1), cout => c2);
F3:  Full_Adder port map(x => a(2), y => b3, cin => c2, s => result(2), cout => c3);
F4:  Full_Adder port map(x => a(3), y => b4, cin => c3, s => result(3), cout => c4);
F5:  Full_Adder port map(x => a(4), y => b5, cin => c4, s => result(4), cout => c5);
F6:  Full_Adder port map(x => a(5), y => b6, cin => c5, s => result(5), cout => c6);
F7:  Full_Adder port map(x => a(6), y => b7, cin => c6, s => result(6), cout => c7);
F8:  Full_Adder port map(x => a(7), y => b8, cin => c7, s => result(7), cout => c8);
F9:  Full_Adder port map(x => a(8), y => b9, cin => c8, s => result(8), cout => c9);
F10: Full_Adder port map(x => a(9), y => b10, cin => c9, s => result(9), cout => c10);
F11: Full_Adder port map(x => a(10), y => b11, cin => c10, s => result(10), cout => c11);
F12: Full_Adder port map(x => a(11), y => b12, cin => c11, s => result(11), cout => c12);
F13: Full_Adder port map(x => a(12), y => b13, cin => c12, s => result(12), cout => c13);
F14: Full_Adder port map(x => a(13), y => b14, cin => c13, s => result(13), cout => c14);
F15: Full_Adder port map(x => a(14), y => b15, cin => c14, s => result(14), cout => c15);
F16: Full_Adder port map(x => a(15), y => b16, cin => c15, s => result(15), cout => over);
```

조건 만족함

- State Machine (Behavioral Description): calculator_top in Behavioral Description

```
type STATE_TYPE is ( STATE_ON, MUL2, MUL3, MUL4, MUL5, MUL6, MUL7, MUL8, MUL9, MUL10, MUL11, MUL12, MUL13, MUL14, MUL15, MUL16);
signal STATE : STATE_TYPE;

begin
a_1(7 downto 0) <= DATA_A;
a_1(15 downto 8) <= (others => DATA_A(7));
b_1(7 downto 0) <= DATA_B;
b_1(15 downto 8) <= (others => DATA_B(7));
none <= (others => '0');
--초기값 설정


process (SYS_CLK, SYS_RESET_B)
   begin
      if SYS_RESET_B = '0' then
       temp_a <= (others => '0');
       temp_b <= (others => '0');
       temp_end_op <= '0';

       shifted_a <=  (others => '0');
       shifted_b <=  (others => '0');
       add_sub <= '0';
```

(이하 생략)

조건 만족함

b) SM chart for multiplier applying Shift and Add algorithm

```
when MUL2 =>
   if shifted_b(0) = '0' then
       --shift
       shifted_a(15 downto 1) <= shifted_a(14 downto 0);
       shifted_a(0) <= '0';
       shifted_b(15) <= '0';
       shifted_b(14 downto 0) <= shifted_b(15 downto 1);

       temp_a <= alu_result; -- 결과 그대로 보존
       temp_b <= none;
       add_sub <= '0';

   else
       --add and shift
       temp_a <= shifted_a;
       temp_b <= alu_result;
       add_sub <= '0';

       shifted_a(15 downto 1) <= shifted_a(14 downto 0);
       shifted_a(0) <= '0';
       shifted_b(15) <= '0';
       shifted_b(14 downto 0) <= shifted_b(15 downto 1);

   end if;


ALU: Binary_Adder_Substractor_8bit port map( a => temp_a,
                                             b => temp_b,
                                             oper => add_sub,
                                             result => alu_result);
```

Shift and Add Algorithm에 해당하는 SM chart를 코드로 구현한 결과이다.

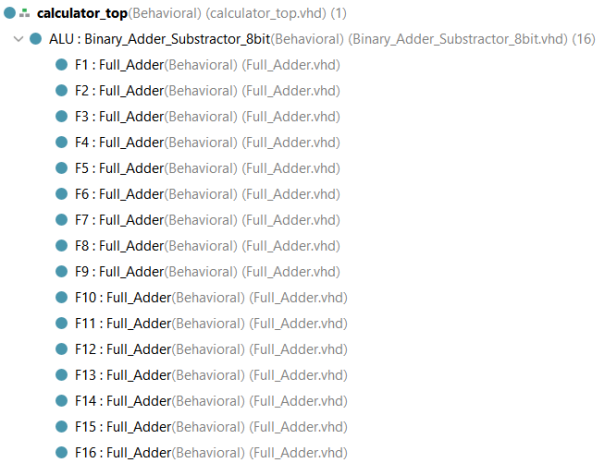
c) Consideration of Signed Data

- 초기에 DATA_A와 DATA_B를 16bit로 만든 임시변수에 부호에 맞는 값을 할당

- 이를 통해 연산과정에서 발생하는 Overflow를 대처

```
begin
a_1(7 downto 0) <= DATA_A;
a_1(15 downto 8) <= (others => DATA_A(7));
b_1(7 downto 0) <= DATA_B;
b_1(15 downto 8) <= (others => DATA_B(7));
none <= (others => '0');
```

2) Top-Down Design (Hierarchical Design)



calculator_top을 최상위 블록으로 설정하고, State Machine을 통해 필요한 상위 제어를 수행한다. 덧셈 및 뺄셈의 연산을 수행하기 위해 하위로 Binary_Adder_Substractor_8bit를 구성해 Ripple Carry Add를 수행한다. 또한, 이 Binary_Adder_Substractor_8bit의 하위에 16개의 Full_Adder를 구성하여 한 비트 별 Full Add를 수행한다.

3) Synchronous Design Check

- Synchronous Design을 위해 process에 CLK와 reset을 넣음.

```
process (SYS_CLK, SYS_RESET_B)
   begin
      if SYS_RESET_B = '0' then
       temp_a <= (others => '0');
       temp_b <= (others => '0');
       temp_end_op <= '0';

       shifted_a <=  (others => '0');
       shifted_b <=  (others => '0');
       add_sub <= '0';
```

4) Testbench 작성요령 & Stable Work Check

설계한 calculator의 동작을 검증하기 위해 여러 케이스를 나눠 Testbench를 작성하였다. 각 operation에 대한 DATA_A, DATA_B에 부호를 고려한 모든 케이스를 넣어 기대하는 값과 시뮬레이션 결과값을 비교함으로써 calculator를 검증하였다. 이에 대한 테스트 진행 결과는 3-A에 포함하였고, 원하는 결과가 출력됨을 확인할 수 있다.

5) 기타 논의사항

1. Signed 8bit와 8bit를 연산하는 과정에서 부호 고려

=> 8bit와 8bit를 연산했을 때 덧셈, 뺄셈을 연산하는 과정에서 Overflow가 발생될 수 있다. 이때 DATA_C에 할당해주지 않은 나머지 bit들에 어떤 값을 넣어줘야 하는 지 결정하기 힘든 상황들이 발생한다. 따라서 calculator_top 내부에서 DATA_A와 DATA_B를 16bit로 만든 임시 변수에 부호에 맞게 할당하여 연산하도록 하였다.

2. Multiplier를 모듈로 만들어서 제작했을 때 Multiplier 연산은 16 clock동안 수행되는 반면 calculator_top에서는 1 Clock 만에 Multipier의 값을 요구하면서 문제 발생

=> Multiplier를 모듈로 만들지 않고 calculator_top 내부적으로 STATE에 따라 동작하도록 설계한다. 즉 STATE_ON일 때 OP_CODE를 "100"으로 받으면 MUL2부터 MUL16까지 Shift_add를 각각 동작하도록 설계하고 마지막에 연산을 다 마치면 End_op를 1로 할당하였다.

3. 처음에 reset을 Synchronous하게 동작하도록 함

=> reset을 STATE에 따라 조절되지 않도록 Asynchronous 하게 동작하도록 한다.

4. 처음에 STATE에 대한 개념이 정확히 없어서 불필요한 Signal들을 내부적으로 많이 정의하여 연산을 수행하게 하여 원하는 Clock에 결과를 못 내놓는 경우도 생기고 이전에 연산에 사용되었던 Signal들이 다음에 수행될 연산에도 영향을 미쳐 값이 이상하게 나오는 경우가 발생함.

=> 불필요한 Signal들을 다 없애고 최소한의 Signal들로 연산을 수행하게 하였다.