SE273 Term Project Report

13조
201911189 한현영, 201911127 이승우

I. Introduction

A. 목적

본 프로젝트의 궁극적인 목적은 다양한 ALU를 구현하는데 있다. Adder Subtractor, Multiplier를 그 목적에 맞게 Combinational, Sequential logic / Hierarchical design / Synchronous design과 같이 다양한 방식으로 설계한다. 이후 test bench를 작성하여 고안한 logic이 설계 목적에 맞는지, input 대비 output이 정상적으로 도출되는지에 대해 검증하는 과정을 거친다.

B. 목표 및 기준 설정

목표	기준
Adder 정상 작동	Testbench 내 임의의 두 8bits operand 간의 가산이 정상적으로 작동하는지 검증한다. (e.g. 00000110 + 00001101 = 00010011)
Subtractor 정상 작동	Testbench 내 임의의 두 8bits operand 간의 감산이 정상적으로 작동하는지 검증한다. (e.g. 00000110 - 11110011 = 00010011)
Multiplier 정상 작동	Testbench 내 임의의 두 8bits operand 간의 감산이 정상적으로 작동하는지 검증한다. (e.g. 00001111 * 00001010 = 10010110)
전체적인 state 정상 작동	Testbench 내 임의의 두 8bits operand 가산, 감산, 곱셈이 정상적으로 작동하는지 검증한다. 더불어 여러 signal(SYS_CLK, SYS_RESET_B, OP_CODE 등)에 logic이 정상적으로 대응하는지 관찰한다. 또한 DATA_A, DATA_B에 대한 DATA_C 가 정상적으로 출력되는지 관찰한다.

C. 팀원간 역할

이름(학번)	역할
한현영 (201911189)	SM diagram design / Adder(Subtractor), Top calculator, Testbench implementation / code version management(Git)
이승우 (201911127)	Multiplier(shift-add) / SM chart design/ Modify testbench

II. Composition and Analysis

A. State Diagram

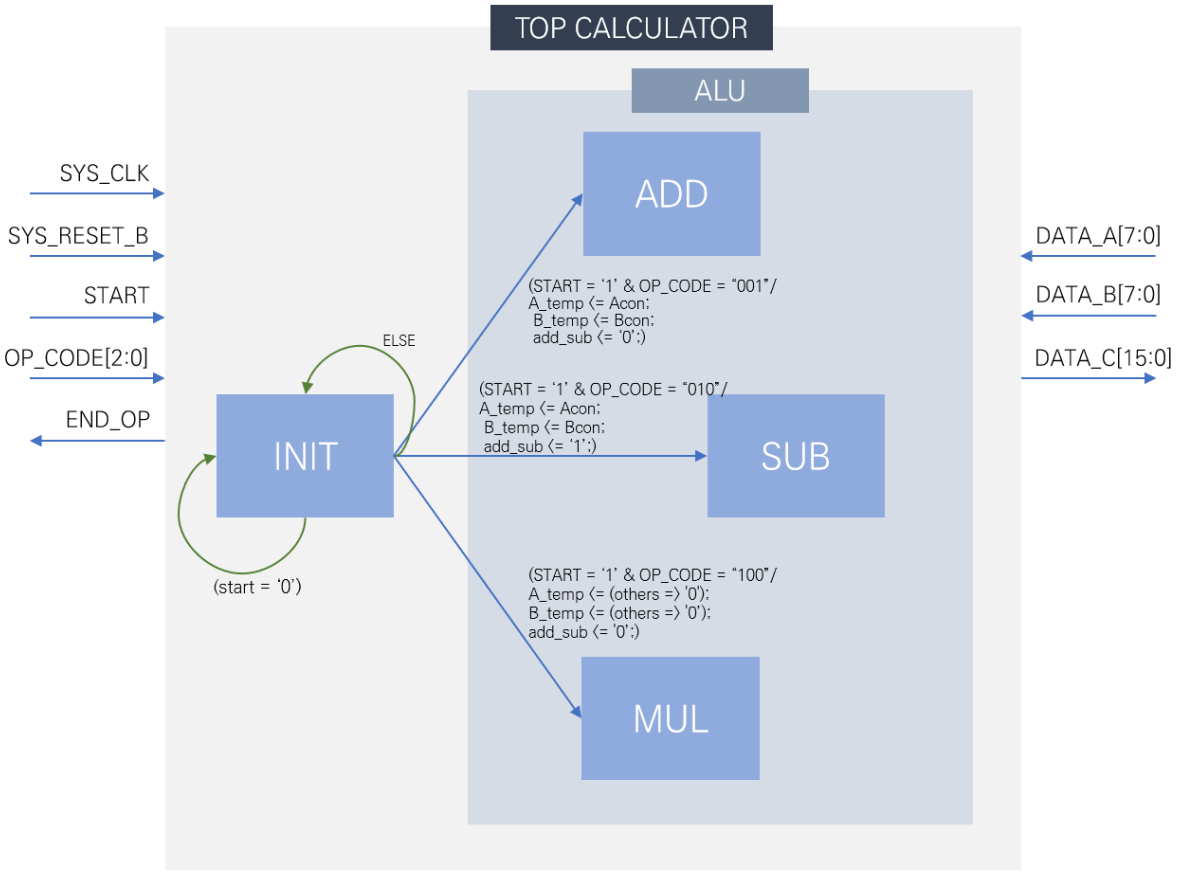

fig 1. 전체 state diagram

전체적인 state diagram은 위 그림과 같다. 먼저, initial state는 INIT으로 START 신호에 따라 상태가 달라지게 된다. (START='1') 신호를 받게 되면 OP_CODE에 따라서 각기 다른 연산 과정으로 진입한다. 가령 OP_CODE='001'이면 Adder, OP_CODE='010'이면, Subtractor, OP_CODE='100'이면 Multiplier로의 역할을 한다. 이 때, 위 세가지 OP_CODE를 제외한 다른 값이 입력되면 다시 INIT상태로 되돌아가고, DATA_C[15:0]의 경우 '0000000000000000'을 반환하게 된다. 기본적으로 ADDER의 경우 전형적인 Full Adder의 형식을 따랐다. 이를 기반으로 Subtractor, Multiplier가 작동하게 된다.

Multiplier의 state diagram은 다음과 같다.

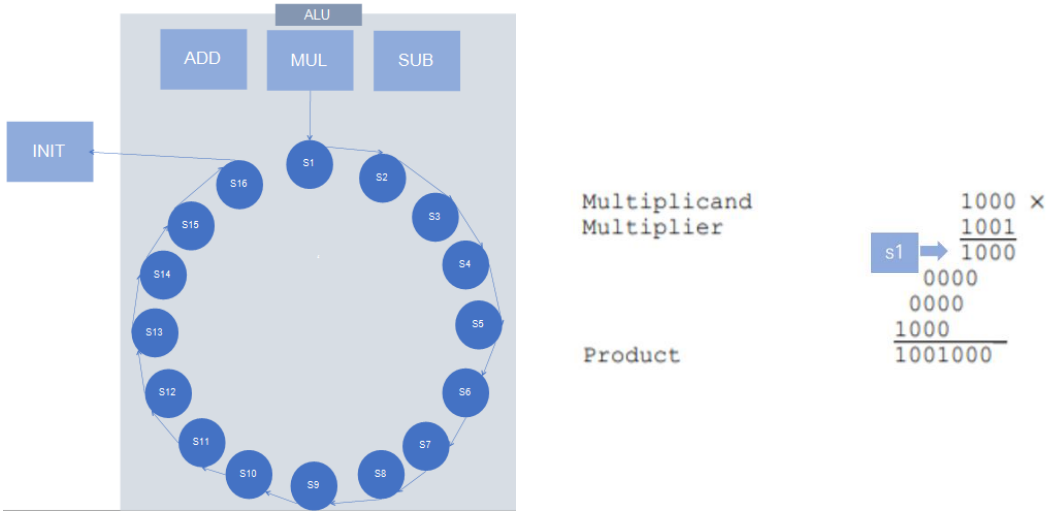

fig 2. Multiplier의 state diagram

OP_CODE='100' 신호를 받게 되면, s1 state로 진입한다. 이때 's1'상태는 shift add를 통해 곱셈을 계산할 경우 첫번째로 계산되는 과정을 말한다 (위의 오른쪽 그림 참고). 그 후, 순차적으로 's2', 's3', ..., 's16' 상태를 지나 곱셈 연산을 완료하게 된다. 마지막으로, 다시 'INIT'상태로 되돌아가 새로운 OP_CODE를 입력받아 다음 연산을 진행하게 되는 것이다. (곱셈 과정은 '토의' 부분에서 더 자세하게 다루도록 하겠다.)

B. 시뮬레이션을 통한 모듈 검증

- Adder

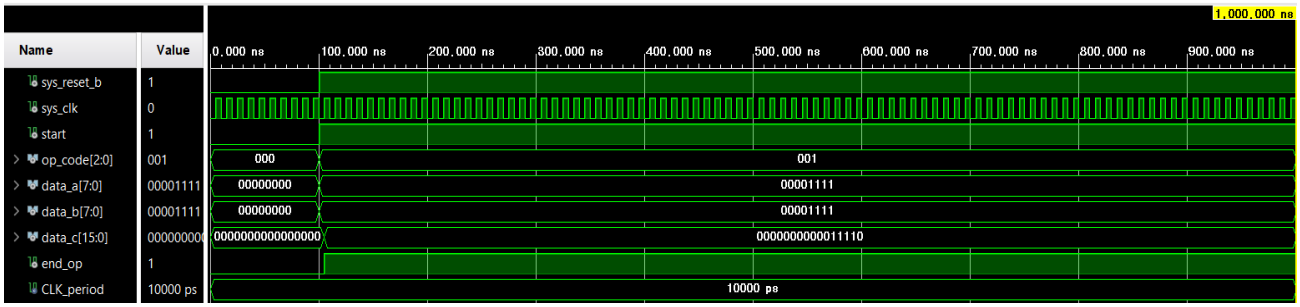

fig 3. Adder_simulation(DATA_A='00001111', DATA_B='00001111')

- Subtractor

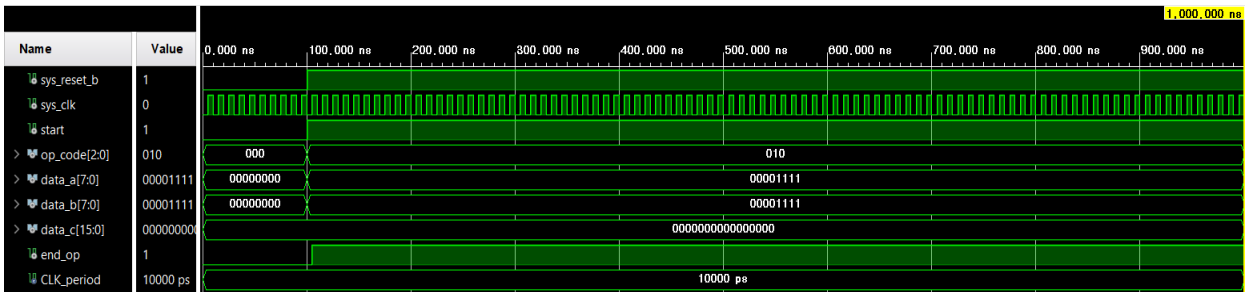

fig 4. Subtractor_simulation(DATA_A='00001111', DATA_B='00001111')

- Multiplier

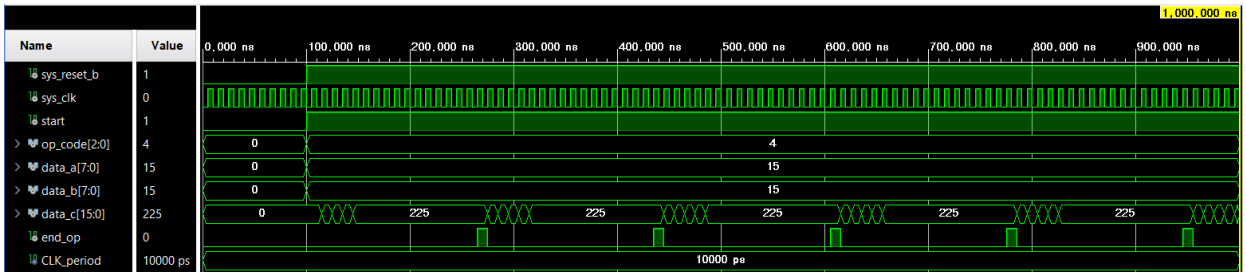

fig 5. Multiplier_simulation(DATA_A='00001111', DATA_B='00001111')

각각의 경우, 모두 옳은 결과값을 도출해내는 것을 시뮬레이션을 통해 검증할 수 있었다.

III. Result and Discussion

A. 결과 도출

아래 결과처럼, 설계한 Adder, Subtractor, Multiplier 모두 input에 따라 정상적으로 작동함을 알 수 있다. 연산 결과 뿐 아니라, 외부적인 signal의 I/O 처리도 정상적으로 함을 볼 수 있다. 또한, 값을 반환할 때 end_op='1'을 출력하도록 설계한 것까지 작동함을 알 수 있다.

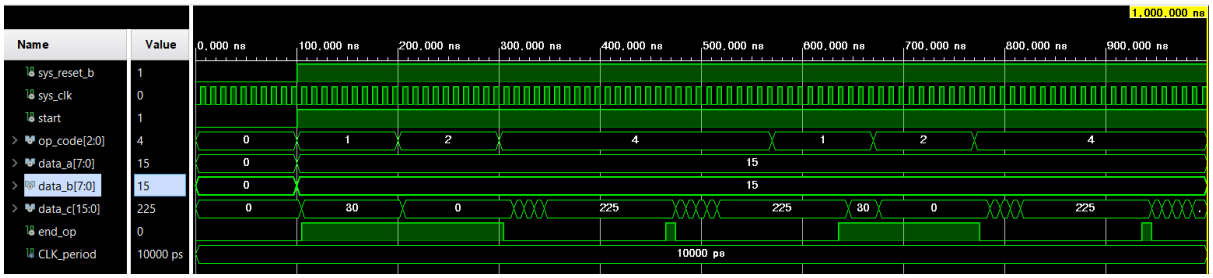

fig 6. 임의의 입력에 대한 출력 결과

B. 토의

i. 제시된 주요 설계요소 및 제한요소를 만족하는가

제시된 Reference에서 언급된 Block Diagram, Signal Description 모두 포함하여 설계하였고, 주어진 OP_CODE(001, 010, 100) 외 다른 값이 들어올 경우 예외까지 처리하였다.

1. VHDL의 세 가지 표현 방식인 Structural Description, Behavioral Description, Data Flow Description의 방법을 모두 사용하여 설계하였는가

```
elsif (rising_edge (sys_clk)) then
    case state is

        when init =>
            A_temp <= (others => '0');
                    B_temp <= (others => '0');
             if start = '1' then
                if op_code = "001" then
                    end_op_temp<='1';
                    A_temp <= Acon;
                    B_temp <= Bcon;
                    add_sub <= '0';
                    state <= init;

                elsif op_code = "010" then
                    end_op_temp<='1';
                    A_temp <= Acon;
                    B_temp <= Bcon;
                    add_sub <= '1';
                    state <= init;

                elsif op_code = "100" then
                    end_op_temp<='0';
                   -- A_temp <= (others => '0');
                    --B_temp <= (others => '0');
                    add_sub <= '0';
                    state <= s1;
```

fig 7. Behavioral Description

```
entity ADDER is
    Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           Cin : in STD_LOGIC;
           S : out STD_LOGIC;
           Cout : out STD_LOGIC);
end ADDER;

 architecture Behavioral of ADDER is

begin

    S <= A XOR B XOR Cin;
    Cout <= (A AND B) OR (Cin AND A) OR (Cin AND B);

end Behavioral;
```

fig 8. Dataflow Description

```
entity EIGHTBIT_ADDER is
    Port ( Ain : in STD_LOGIC_VECTOR (15 downto 0);
           Bin : in STD_LOGIC_VECTOR (15 downto 0);
           oper : in STD_LOGIC;
           C : out STD_LOGIC_VECTOR (15 downto 0));
end EIGHTBIT_ADDER;

architecture Behavioral of EIGHTBIT_ADDER is
component ADDER is
     Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           Cin : in STD_LOGIC;
           S : out STD_LOGIC;
           Cout : out STD_LOGIC);
end component;

result : EIGHTBIT_ADDER port map(Ain => A_temp, Bin => B_temp, C => C_temp , oper => add_sub);
```

fig 9. Structural Description

8 bit adder module을 만들 때, Data flow description, 8 bit-ALU module을 만들 때, Structural description, 전체적인 state machine을 설계할 때, Behavioral description을 사용하였다.

2. Shift and add algorithm에 의해 SM chart를 작성한 후 multiplier를 설계하였는가

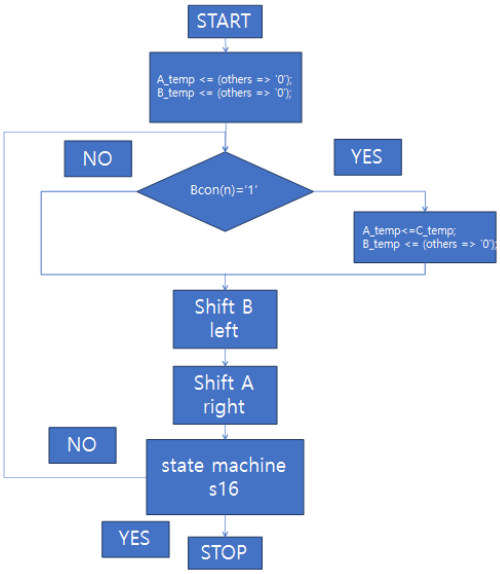

fig 10. SM Chart

Multiplier 설계에 사용한 SM Chart는 위와 같다.

먼저, 사용된 shift 과정에 대해서 이야기 하겠다.

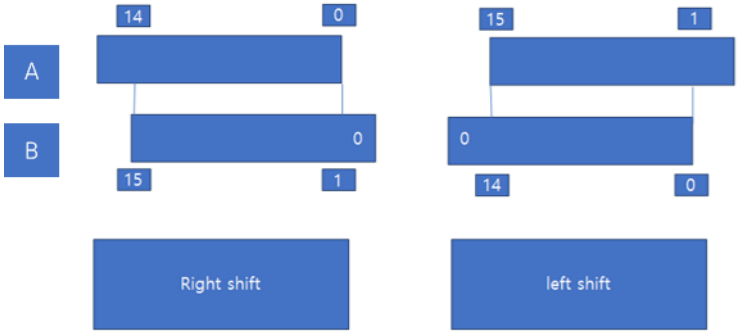

fig 11. Multiplier에서의 Shift methio

오른쪽으로 shift하는 경우, A의 14->0 부분을 B의 15->1 부분에 대입한 뒤, 맨 끝자리에 0을 추가하면서 진행된다. 반대로 왼쪽으로 shift하는 경우, A의 15->1 부분을 B의 14->0 부분에 대입한 뒤 맨 앞자리에 0을 추가한다. 이런 과정을 통해 shift를 진행하였다.

```
Multiplicand          1000 ×
Multiplier            1001
                      1000
                     0000
                    0000
                   1000
Product            1001000
```

fig 12. shift-add method

그 다음, 중요한 점은 Multiplier의 맨끝자리가 0인지 1인지에 따라 ADD 여부가 결정되었다. 맨끝자리가 1인 경우에는 ADD를 진행하였고, 0인 경우에는 ADD를 진행하지 않았다. 물론, 두 경우 모두 SHIFT 과정이 일어났다. 여기서 사용한 ADD는 계산값을 Product에 누적하기 위한 목적이다. 위 사진처럼 '1000', '0000', '0000', '0000' 등을 더하여 product에 저장하였다. 해당 과정을 S1 상태부터 S16상태까지 반복하고 난 뒤, 최종 계산결과가 나오게 되면 다시 INIT 상태로 되돌아가 다른 연산을 할 준비를 하게 된다.

3. 설계한 Adder, Subtractor, Multiplier가 부호를 고려하는가

부호를 고려한다. 아래 Testbench를 통해서도 확인 가능하다.

- Adder

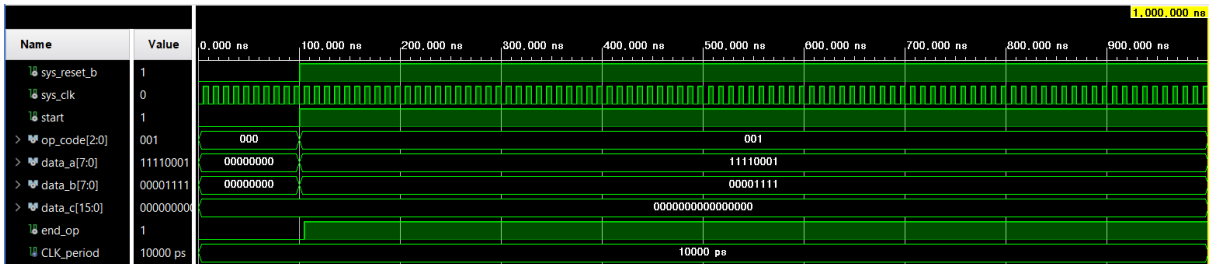

fig 13. TB_ADDER

- Subtractor

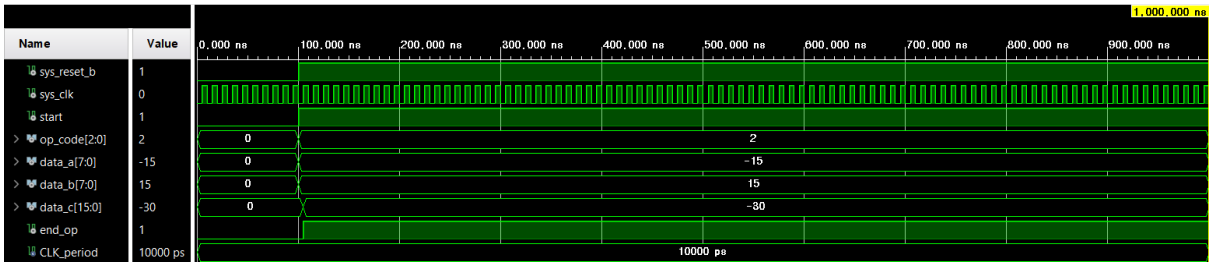

fig 84. TB_SUBTRACTOR

-Multiplier

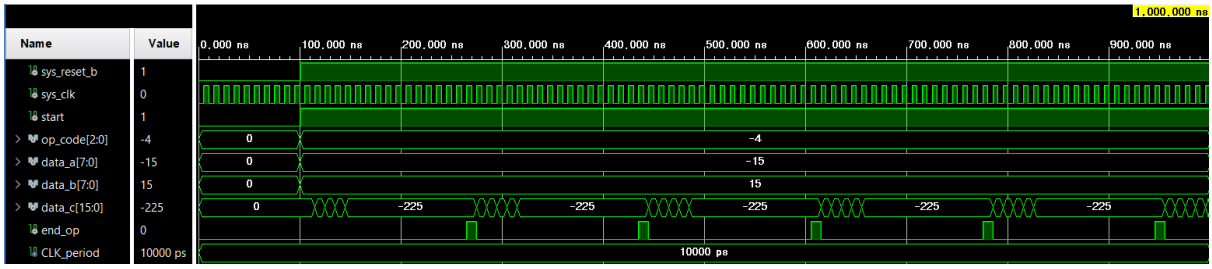

fig 15. TB_MULTIPLIER

ii. Top-down 방식의 설계가 적용되었는가

구현에 앞서 각 state를 잇는 조건과 해당 state에서 어떠한 action을 취해야 하는지에 대한 State Machine을 구상하였다. 이후 ALU에 진입하였을 때 어떠한 algorithm을 사용하여 연산을 할지 궁리하였다.

iii. Hierarchical Design 방식의 설계가 적용되었는가

가령 ALU의 경우 세 operation 모두 ADDER를 기반으로 작동한다. 기본적으로 ADDER를 구현한 뒤, 덧셈과 곱셈은 Sign <= '0'인 ADDER를, 뺄셈은 Sign <= '1'인 ADDER에 기인하여 동작한다.

iv. Synchronous Design 방식의 설계가 적용되었는가

Process를 이용하여 주어진 Calculator_top이 clock에 의존하면서 실행되기 때문에, 이는 Synchronous Design 방식의 설계가 적용되었다는 것을 알 수 있다.

v. Testbench의 작성 요령 및 임의의 입력에 대해서 안정적으로 동작하는가

Testbench 작성시 가장 중요시하게 여겼던 점은, multiplier에서 생기는 delay를 해결하고 정확한 값이 출력될 수 있도록 하는 것 이었다. s1에서 s16까지의 여러 상태들과, 계속 값이 누적이 되도록 알고리즘을 구현했기 때문에, delay가 발생하였다. 이를 해결하기 위해서 곱셈기에서 testbench의 wait time을 170ns으로 해주었을 때, 가장 정확한 결과가 도출됨을 알 수 있었다.

아래 그림과 같이 임의의 입력에 대해서 Adder, Subtractor, Multiplier 모두 안정적으로 동작하는 것을 볼 수 있다.

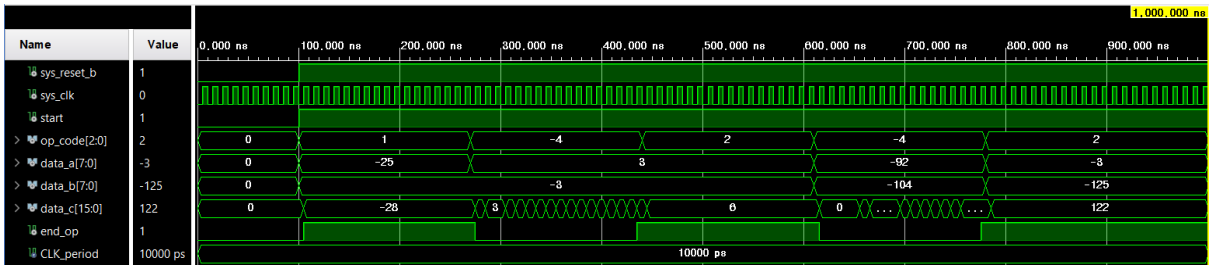

fig 16. 종합적인 TB result

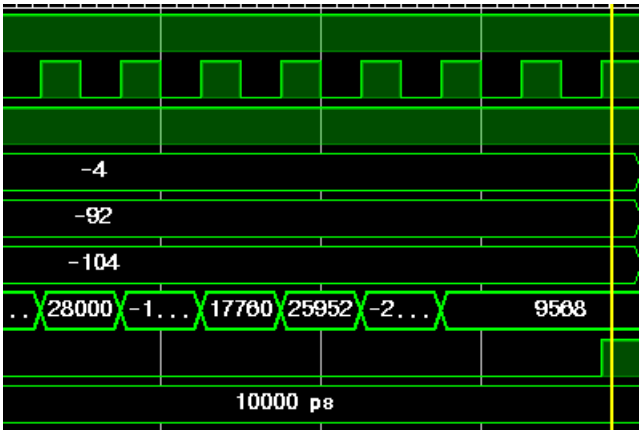

fig 17. (−92) * (−104) = 9,568

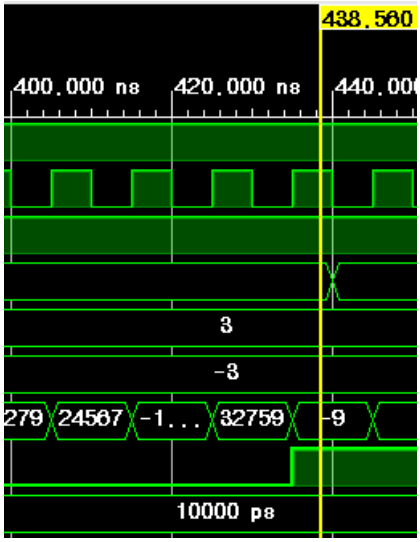

fig 18. 3 * (−3) = −9

vi. 기타 논의사항

Multiplier 구현 시 매 state마다 shift and add를 해주는 방식으로 설계하였다. 이 때문에 코드가 길어졌는데, 이를 간결하게 표현할 수 있는 방안에 대한 논의가 필요하다. 가령 각 adder. shifter를 modularize하여 표현할 수 없는지 생각해보아야 한다. 또한 multiplier 구현에서, 아무리 end_op = '1'일 때의 출력 값(DATA_C)을 본다고 하더라도 최종 출력 전의 garbage 값을 출력하지 않는 방안에 대해 살펴볼 필요가 있다.