<디지털 논리회로 term project 보고서>

202111133 이석민
201811185 천새루

1) 도입부

A. project 목적

-> 덧셈과 뺄셈, 곱셈을 수행하는 simple calculator를 VHDL을 이용하여 FPGA level에서 설계하고 하드웨어에서 정상적으로 작동하는지 확인한다.

B. 목표 및 기준 설정

-> 이미 구현되어있는 +,-등의 연산기능을 사용하지 않고, calculator를 설계한다. 또, multiplier는 shift and add multiplier로 구현한다. 우리 팀은 먼저 state diagram을 통해 calculator의 기본 형태를 디자인하고, logisim이라는 툴을 이용하여 simulation을 통해 logic gate level에서 우리가 디자인한 회로가 정상적으로 작동하는지 확인한다. 그 이후, 마지막으로 VHDL을 이용하여 calculator를 FPGA level에서 만들어본다.

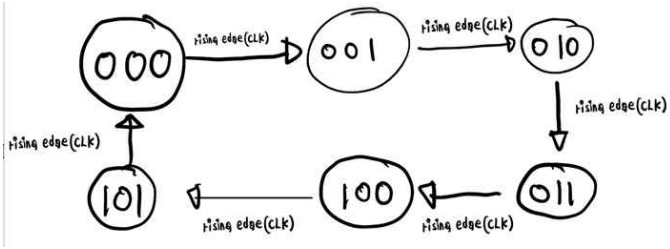
C. 팀원간 역할 명시

-> 기여도는 50%, 50%입니다. 회로 설계, logisim을 이용한 구현, vivado에서 코딩, 보고서 모두 두명이 함께 힘을 합쳐 해낼 수 있었습니다.

2) 합성 및 분석

A. State Diagram

-> counter를 설계할 때에 State Diagram을 사용하였습니다. Shift and add multiflier를 디자인 할 때에는 Counter의 State와 함께 해당 state에서 각 register에서 일어나는 일을 표로 작성하였습니다. 밑의 그림에 해당 내용을 첨부하였습니다. 각 그림에 대한 설명은 해당 보고서 **4) 전체 Schematic 설명** 부분에 있습니다.



<그림 1 : State diagram of counter>

CNT	Shift register_A	Shift register_B	Sum register	Ans register
000	x	x	x	x
001	0000A	B4 B3 B2 B1	00000000	x
010	000A0	B1 B4 B3 B2	00...0(8bit)+(0000A*B1)=K1	x
011	00A00	B2 B1 B4 B3	K1+(000A0*B2) = K2	x
100	0A000	B3 B2 B1 B4	K2+(00A00*B3) = K3	x
101	A0000	B4 B3 B2 B1	K3+(0A000*B4) = **Answer**	x
000	x	x	x	**Answer**
001	0000A	B4 B3 B2 B1	00000000	**Answer**
010	000A0	B1 B4 B3 B2	00...0(8bit)+(0000A*B1)=K1	**Answer**
011	00A00	B2 B1 B4 B3	K1+(000A0*B2) = K2	**Answer**
100	0A000	B3 B2 B1 B4	K2+(00A00*B3) = K3	**Answer**

<표1 : state and Q_value_of_register>

B. 시뮬레이션을 통한 모듈 검증

->제출한 project파일을 보면 임의의 VHD파일에 대하여 해당 파일이름 뒤에 _tb 라는 이름을 가진 파일이 하나 더 존재합니다. 이 파일이 바로 각모듈에 대한 시뮬레이션 검증 파일이며, 모든 모듈의 시뮬레이션이 정상적으로 동작하는 것을 확인했습니다. 주요 모듈의 정상적인 작동 시뮬레이션은 해당 보고서 **4) 전체 Schematic 설명** 부분에 있습니다.

3) 결과 및 논의

A. 결과 도출

-> 해당 보고서의 **4) 전체 Schematic 설명** 부분의 주요 모듈의 시뮬레이션 결과와, live demo를 통한 검증을 토대로, calculator의 작동이 정상적으로 이루어짐을 알 수 있었다.

B. 설계요소 및 제한요소

-> 부호를 고려하는 calculator를 설계하였고, adder and subtractor는 combinational logic으로, multiplier는 shift and add multiplier 형태로 설계하였다. 코드에는 -,+ 등의 연산자를 일절 이용하지 않았으며, hirarchy 구조를 이용하여 calculator를 설계하였다.

C. VHDL의 3가지 표현 방식

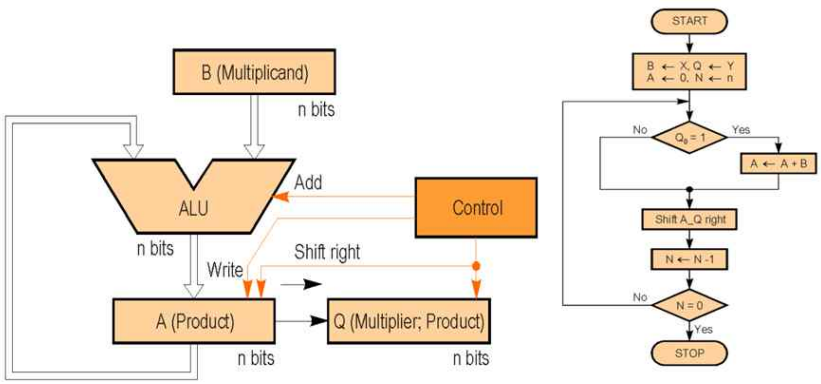
-> 거의 모든 모듈들을 디자인 할 때에 signal들을 이용하여, 한 gate의 output이 다른 gate의 input으로 쓰일 수 있도록 디자인하였고, 이를 통해 Data Flow Description method를 사용하였다.

-> 조건문이 필요한 경우에 process 문을 사용하여 상황을 해결해 나갔고, 특히 rising edge에서 활성화되는 f/f을 설계할 때에 clk에 대한 조건문을 사용하기 위해서 process 문을 많이 사용하였다. sequential하게 동작하는 모든 모듈 파일들에서 process 문을 볼 수 있다. 이를 통해 Behavioral Description method를 사용하였다.

-> 하나의 파일에 한번에 calculator를 디자인 한다면, 에러가 생겼을 때 어떠한 부분에서 에러가 나는지 확인하기 힘들 것이다. 우리는 이와 같은 이유로, hirarchy design을 택했고, 상위 모듈을 위한 하위 모듈들을 먼저 디자인 해가면서 각 모듈이 정확하게 동작하는지 시뮬레이션을 통해 확인했다. 이렇게 hirarchy design을 하면서, component 문을 통해 하위 모듈을 상위 모듈을 설계할 때 사용하였다. 이를 통해 Structural Description method를 사용하였다.

D. SM chart

-> Shift and add algorithm을 알고리즘의 형태로 정리해주는 SM chart를 작성한후 multiplier를 설계하였다. SM chart를 작성하고나니, 확실히 VHDL 코딩할 때 편리한 것을 깨달을 수 있었다.



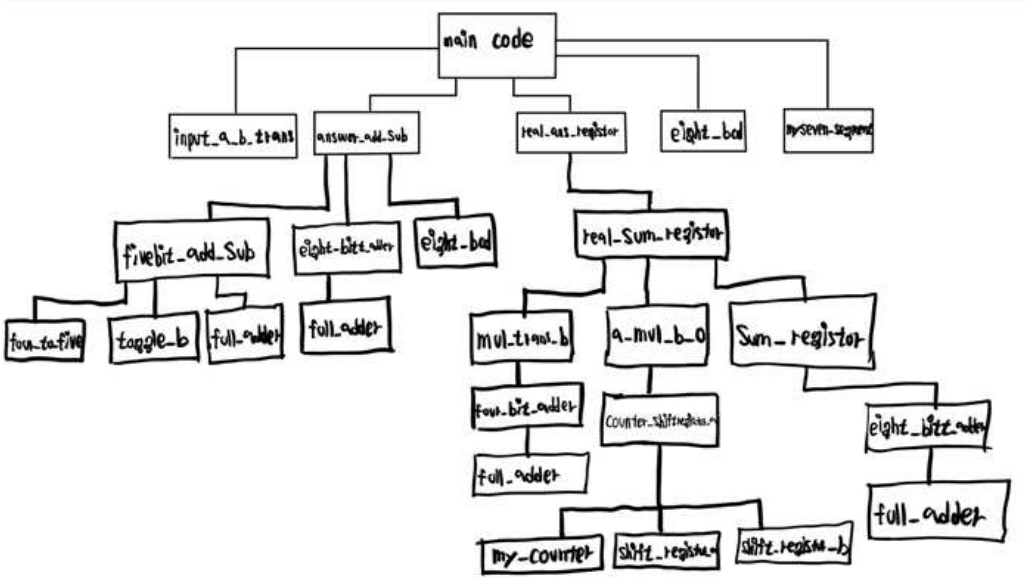
<그림 2 : Shift and add multiflier 동작 논리표와 SM Chart>

E. 부호 고려성

-> **4) 전체 Schematic 설명** 부분을 보면, 설계한 calculator가 부호를 고려하는 것을 알 수 있다.

F. Top-down, Hierarchical Design

-> 코드의 재활용성과 유지보수를 쉽게 하기 위하여 top-down 즉 Hierarchical design을 택할 수 밖에 없었다. 밑의 그림은 전체 코드의 Hierarchical design을 나타낸 것이다.



<그림 3 : Hierarchical design>

G. Synchronous Design

-> shift and add multiplier를 설계할 때, Synchronous Design은 필수적이다. 해당 보고서의 **2)-A**를 보면, Synchronous Design을 하면서 사용한 State diagram 과 state 마다의 register의 Q 값 분석표를 볼 수 있다.

H. Testbench

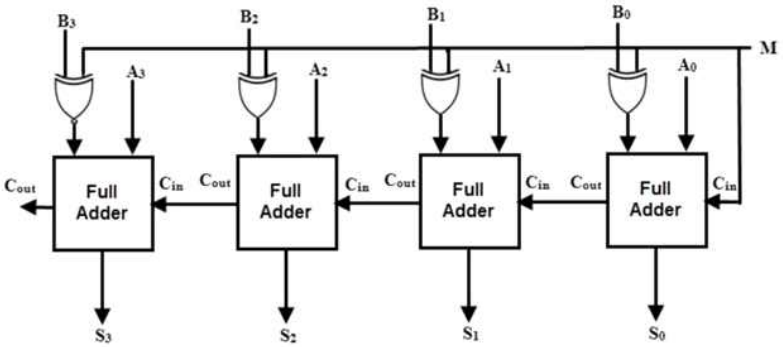
-> 프로젝트 내의 파일들을 보면 알 수 있듯이, 모든 모듈에 대하여 tb파일을 만들어 시뮬레이션을 진행하였다. 모든 모듈에 대해서 시뮬레이션이 정상적으로 작동하는 것을 알 수 있었고, 주요 파일들의 시뮬레이션 결과는 해당 보고서의 **4) 전체 Schematic 설명** 부분에 첨부하였다.

4) 전체 Schematic 설명

A. Adder and subtractor

1. 이론적 배경

-> Add/subtractor를 구현은 아래 회로를 사용했다. Full Adder는 3개의 Input(A, B, Cin)을 받는다. 이 FULL adder를 이용하여 4bit adder로 구현하려면 4개의 full adder를 이어붙이면 된다. subtract의 경우에는 b의 부호를 바꾸어 더해주면 된다.(a-b에서) b의 부호를 바꾸어준다는 얘기는, b의 2's complement를 구한다는 것과 같은데, 이는 b를 toggle 시키고 1을 더해주면 된다. 이를 표현하기 위해서, XOR과 seletion signal M을 사용하면 된다. 본 프로젝트에서는 signed 4bit간의 add/subtraction을 구현하기 때문에 출력값의 범위가 –16~14이다. 이 값은 4bit으로 표현할 수 있는 값의 최대치를 넘어서기 때문에 overflow가 발생할 수 있다. 따라서 overflow를 잘 고려해야할 것이다.



<그림 4 : 4bit add and subtractor>

2. overflow

-> overflow란 컴퓨터에서의 연산 결과가 허용범위의 최대치를 넘어섰을 때 생기는 오류를 의미한다. 본 project에서는 Input signal이 4bit이기 때문에 Adder/subtracter구현에서 overflow가 발생할 수 있다. 실제 예시를 보면, 다음과 같다.

ex)

실제 연산

-8-8=-16 > 나와야 하는 값

Computer 연산

-8(1000)-8(1000)=0(0000) > 의도한 값이 아님

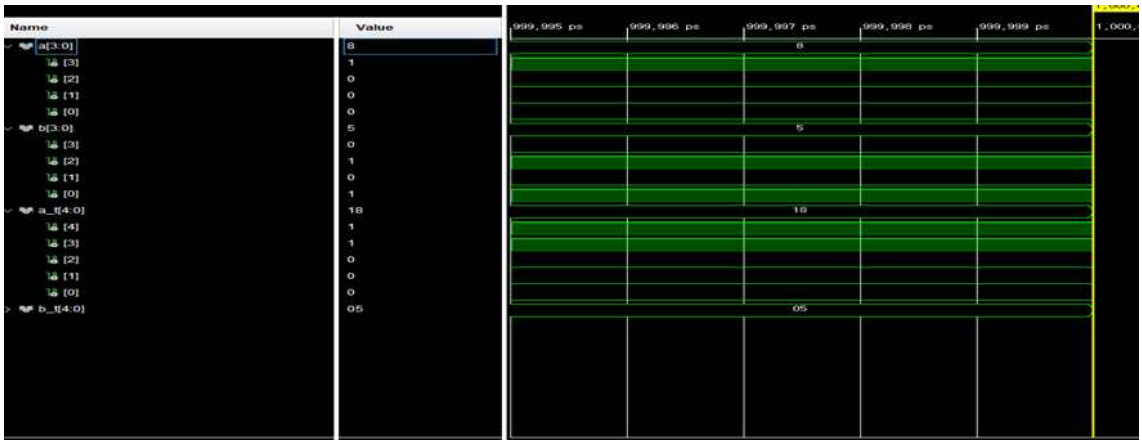
Add/subtractor 구현에서 Overflow현상을 막기 위해서 4 bit singed를 5 bit signed 로 입력값을 변환한 뒤에 회로를 설계했다. 4bit의 입력을 5bit으로 변환해서 연산을 하면, 5bit의 출력 범위가 결과값의 출력범위보다 크기 때문에 overflow가 발생하지 않는다.

아래 표는 4bit의 입력 (I3, I2, I1, I0를 5bit의 Output(Q5, Q4, Q3, Q2, Q1)으로 바꾼 결과값이다.

I3	I2	I1	I0	real value	Q4	Q3	Q2	Q1	Q0
1	0	0	0	-8	1	1	0	0	0
1	0	0	1	-7	1	1	0	0	1
1	0	1	0	-6	1	1	0	1	0
1	0	1	1	-5	1	1	0	1	1
1	1	0	0	-4	1	1	1	0	0
1	1	0	1	-3	1	1	1	0	1
1	1	1	0	-2	1	1	1	1	0
1	1	1	1	-1	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0	1
0	0	1	0	2	0	0	0	1	0
0	0	1	1	3	0	0	0	1	1
0	1	0	0	4	0	0	1	0	0
0	1	0	1	5	0	0	1	0	1
0	1	1	0	6	0	0	1	1	0
0	1	1	1	7	0	0	1	1	1

<표 2 4-bit to 5-bit convertor truth table>

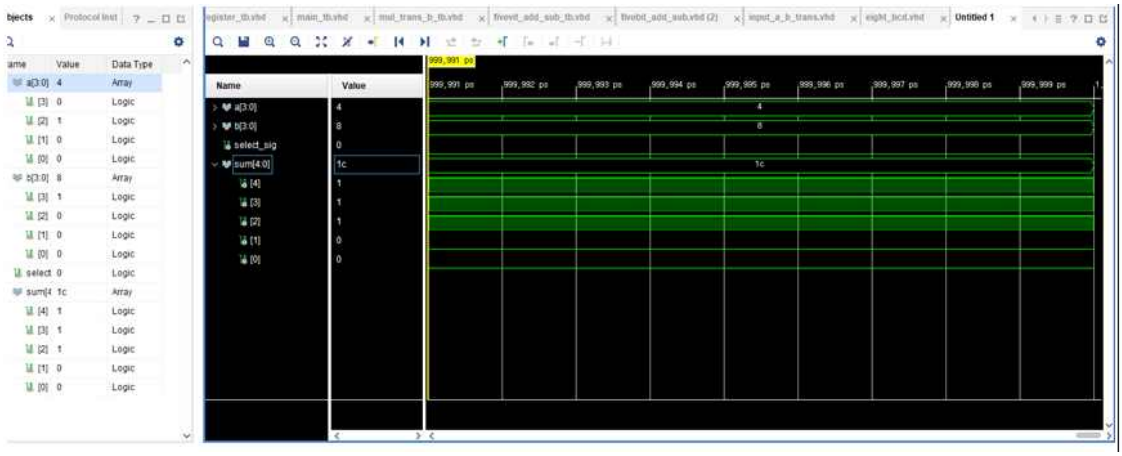
아래 그림 5는 1000(-8)과 0101(5)의 Input을 각각 5bit으로 변환한 시뮬레이션 결과이다. -8은 11000으로, 5는 00101로 변환되었음을 확인할 수 있다.



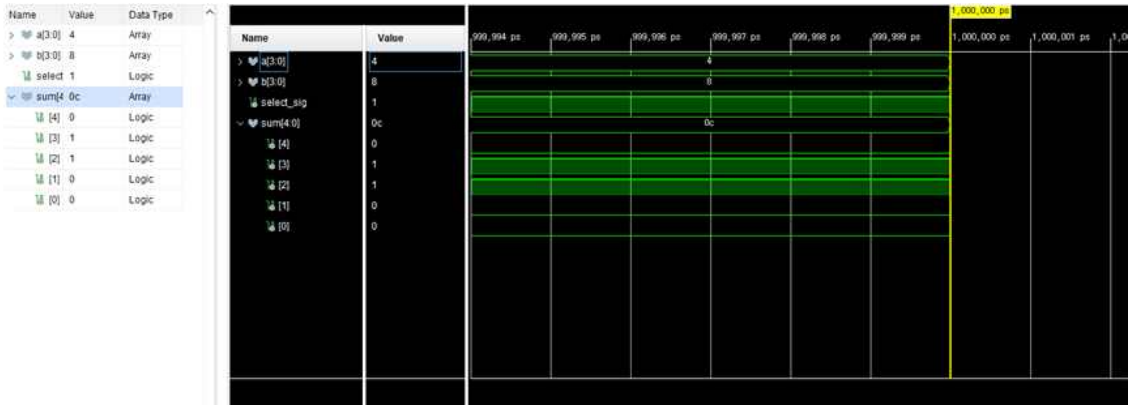
<그림 5 : 4bit to 5bit convertor simulation>

3. 5-bit adder simulation

-> 5 bit로 변환한 값들에 대하여, 5 bit Add/subtract를 구현했다. 그림 2을 보면 sel=0(add)일 경우 4+(-8)을, 그림3을 보면 sel=1(sub)일 경우에 4-(-8)을 구현하도록 했다. Add의 경우 –4(11100), Sub의 경우 12(01100)으로 두 경우 모두 시뮬레이션 결과가 바르게 나왔음을 확인할 수 있다.



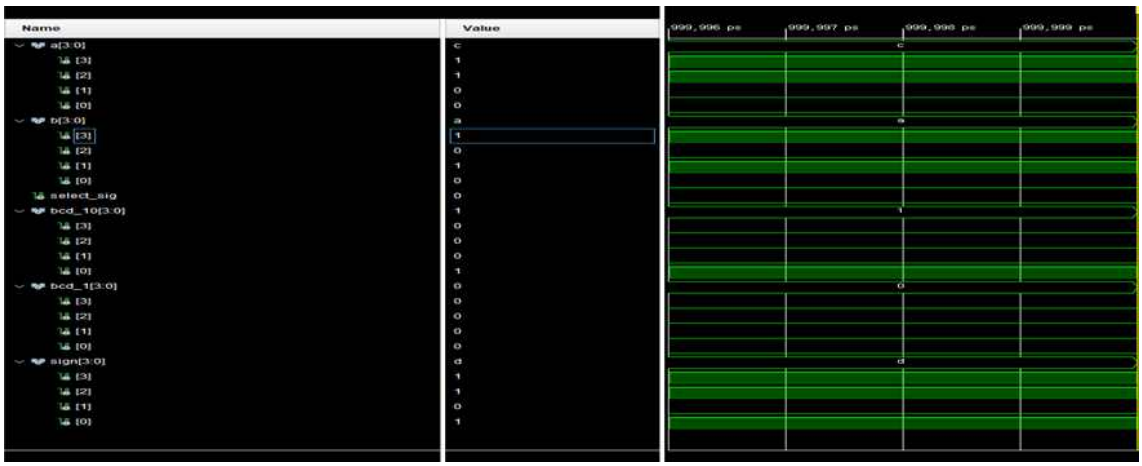
<그림 6 : 4+(-8)=-4 5bit adder>



<그림 7 : 4-(-8)=12 5bit adder>

4. 5-bit add/subtractor output to BCD

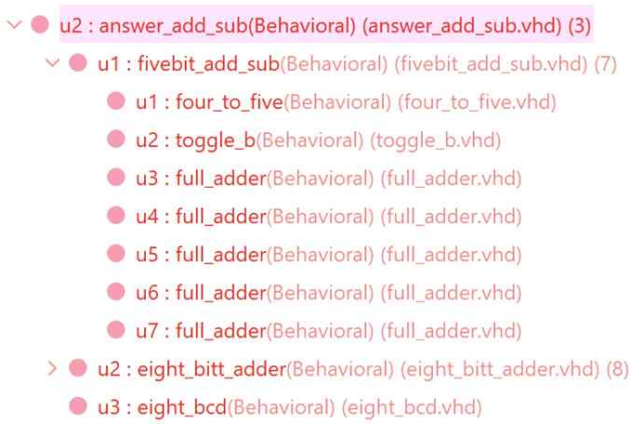
-> 5bit으로 나온 결과값을 BCD로 변환하기 위해서 이후에 나올 8bit to BCD code를 활용했다. 다면 결과값이 5bit이기 때문에, 앞에 3bit를 추가해서 bit 수를 맞춰줬다. 아래 그림은 5bit to BCD의 Simulation 결과이다. 1100(-4) + 1010(-6)의 연산을 했을 때, 10의 자리 BCD에는 0001(1), 1의 자리 BCD에는 0000(0)이 나오는 것을 확인했다.



<그림 8 : add/subtractor output to bcd>

5. 전체 구조

-> 구조의 이해를 돕기 위해 아래 그림 9의 hierarchy Design을 보면 맨 위의 u2 design아래에 4bit을 5bit으로 바꾸고 연산을 해주는 fivebit_add_sub block이 존재한다. 또, fivebit_add_sub block의 결과값인 signed 5 bit를 unsigned 8bit로 바꾸어주는 과정에서 eight_bitt_adder block을 사용한다 (fivebit_add_sub block의 결과값이 음수일 때, 2's complement 취해주는 과정에서). 이 unsigned 8 bit를 10의 자리 BCD와 1의 자리 BCD로 변환해주는 block이 바로 eight_bcd block이다. 최종적으로, signed 4 bit input을 넣으면, signed 4bit > signed 5bit> unsigned 8bit> bcd로 변환되는 과정을 거쳐서 BCD 코드가 나오게 되는 것이다.



<그림 9 : 전체 구조>

B. Shift and add Multiplication

1. 이론적 배경

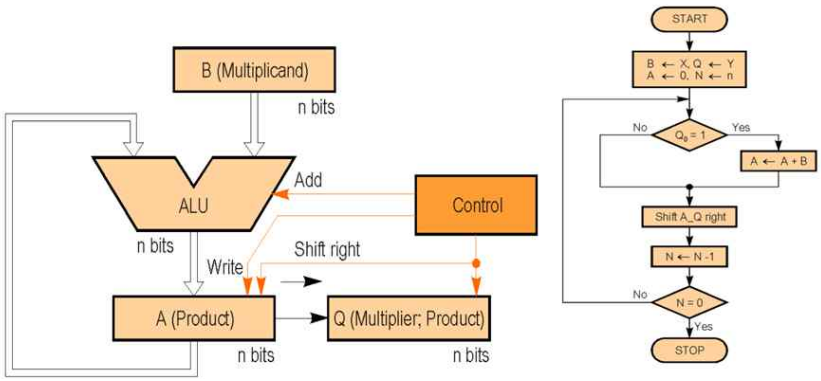
->A(n-bit) B(n-bit)의 곱셈 연산
n-bit의 input A,B가 들어왔을 때, 먼저 2n bit의 register(register_a)에 Q&A를 할당하고,(Q와 A를 이어붙인 것, Q=0, n-bit) n bit의 register(register_b)에 b를 할당한다. 그리고 다른 2n bit의 register(sum_register)의 값은 0으로 초기화한다.

이후 n번만큼 아래 동작을 구현한다.

i) register_b의 LSB가 0인 경우
-> register_a의 bit배열을 한칸씩 왼쪽으로 shift 하고, register_b의 bit 배열을 한칸씩 오른쪽으로 shift 한다.

ii) register_b의 LSB가 1인 경우
-> register_a의 bit배열값을 sum register의 이전 값과 더하여 sum register의 새로운 값에 할당한다. 이후 register_a의 bit배열을 한칸씩 왼쪽으로 shift 하고, register_b의 bit 배열을 한칸씩 오른쪽으로 shift 한다.

이렇게 n번 수행한 뒤에, 나온 sum register의 2n bit 결과값이 multiplication의 결과값이 된다.



<그림 10 : Shift and add multiflier 동작 논리표와 SM Chart>

2. Multiplication 구현

-> 먼저 signed 4 bit를 multiplication을 하기 위해서 부호를 저장해두고, unsigned 4 bit로 바꿔주었다.

-> Multiplication을 구현하기 위해서 3-bit Counter를 사용했다. 4-bit에 대해서 Shift and

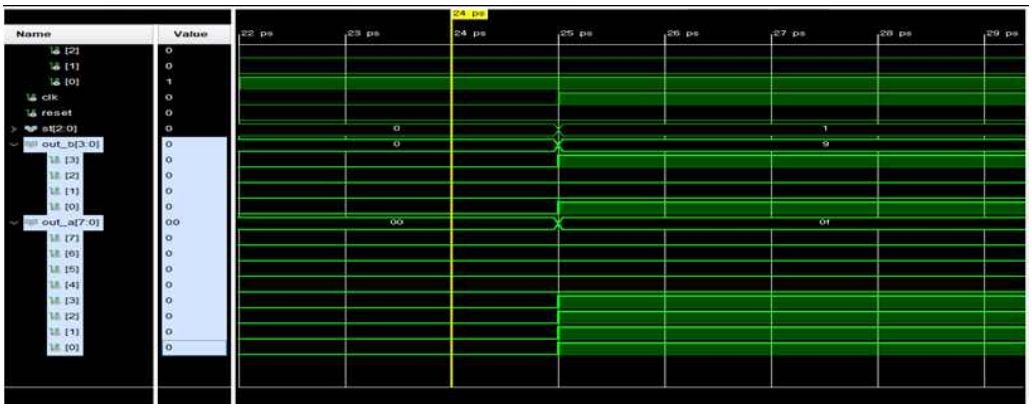
Add를 구현하기 위해서는 적어도 4개의 state가 필요하다. 우리 팀은 6 state를 사용했고, 이에 기존의 3-bit counter에서 살짝 변형된 3- bit counter를 사용하였다. state 가 000일때는 register_a와 register_b, sum_register의 D F/F의 입구(보통 D라고 표시함)에 초기값을 update해 주었다. 이로 인해 state가 001일 때, register_a와 register_b, sum_register의 초기값들이 register_a와 register_b, sum_register의 D F/F의 출구(보통 Q라고 표시함)에 update 된다. 그와 동시에 register_a와 register_b, sum_register의 Q값을 이용하여, 한 stage 연산이 수행되고, 그 연산의 결과가 register_a와 register_b, sum_register의 D F/F의 입구(보통 D라고 표시함)에 update 되면서, 다음 state에 register_a와 register_b, sum_register의 D F/F의 출구(보통 Q라고 표시함)에 update 된다. 이렇게 State마다 이 방식을 반복하면, counter의 state가 101일 때, sum register의 Q값에 답이 update되게 되고, ans register는 이때 D F/F의 입구에 ans가 update되게 된다. 이후 ANS register는 counter의 한 주기동안 그 값을 유지하며, 다음 주기의 000에서 다음 ANS를 update 하게 된다.

CNT	Shift register_A	Shift register_B	Sum register	Ans register
000	x	x	x	x
001	0000A	B4 B3 B2 B1	00000000	x
010	000A0	B1 B4 B3 B2	00...0(8bit)+(0000A*B1)=K1	x
011	00A00	B2 B1 B4 B3	K1+(000A0*B2) = K2	x
100	0A000	B3 B2 B1 B4	K2+(00A00*B3) = K3	x
101	A0000	B4 B3 B2 B1	K3+(0A000*B4) = **Answer**	x
000	x	x	x	**Answer**
001	0000A	B4 B3 B2 B1	00000000	**Answer**
010	000A0	B1 B4 B3 B2	00...0(8bit)+(0000A*B1)=K1	**Answer**
011	00A00	B2 B1 B4 B3	K1+(000A0*B2) = K2	**Answer**
100	0A000	B3 B2 B1 B4	K2+(00A00*B3) = K3	**Answer**

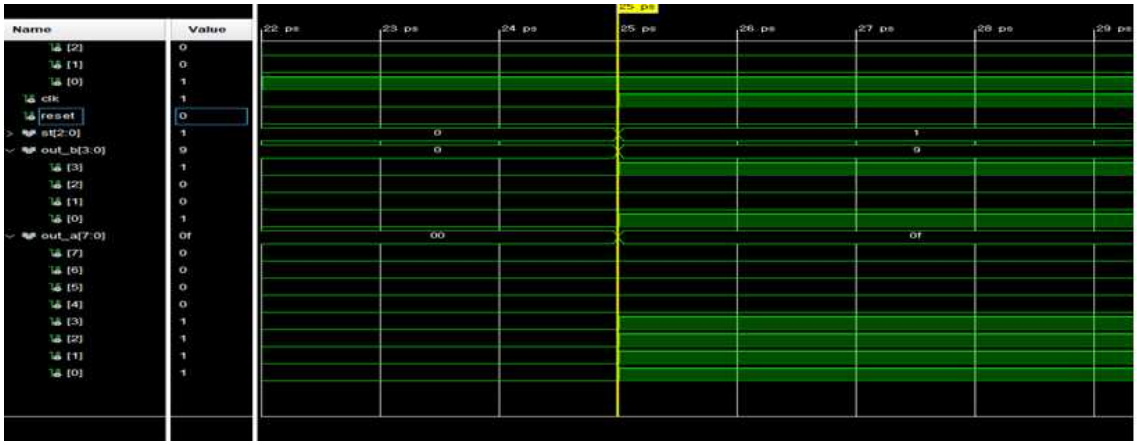
<표3 : state and Q_value_of_register>

3. Shift and Add Multiplication Simulation 결과

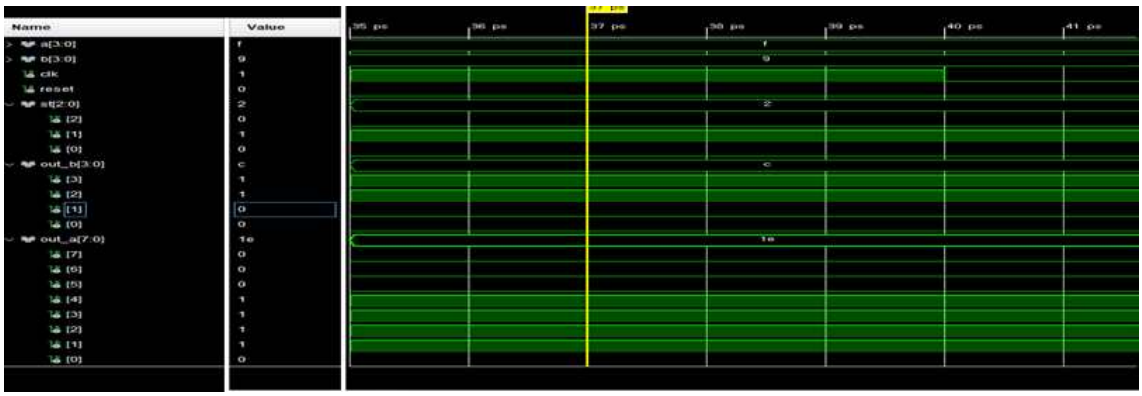
-> 아래 그림들은 A="1111", B="1001"일 때, counter에 동작에 따라서 shifting을 구현하는 Shifter register A, B의 동작을 simulating한 것이다. 아래 그림들을 보면 알 수 있듯이, counter의 state에 따라서 shifting이 잘 구현되고 있음을 알 수 있다.



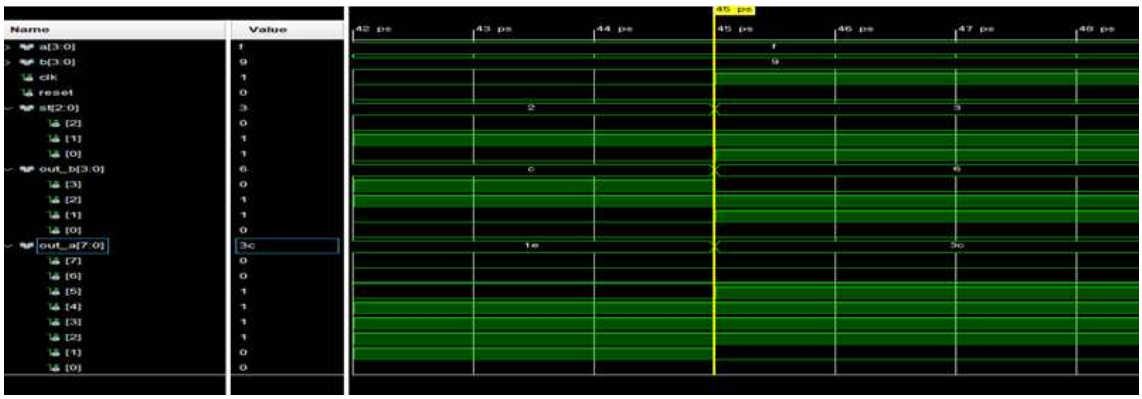
<그림 11 : State = 000 register_b=0000, register_a=00000000>



<그림 12 : State = 001 register_b=1001, register_a=00001111>

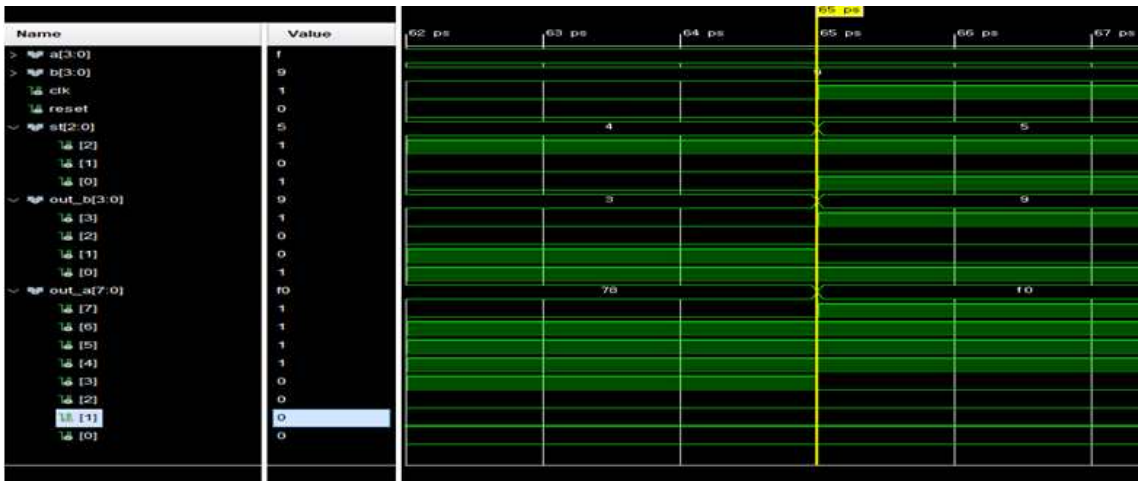


<그림 13 : State = 010 register_b=1100, register_a=00011110>

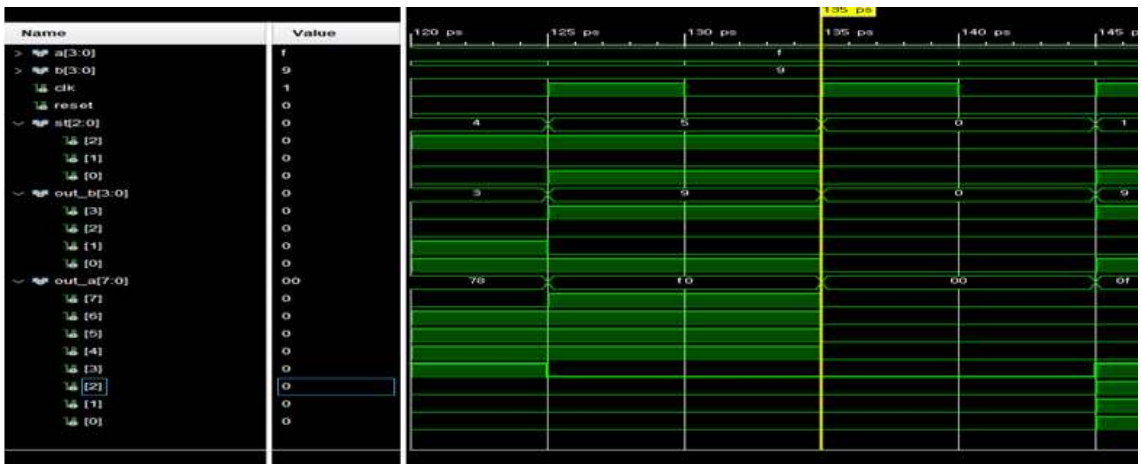


<그림 14 : State = 011 register_b=0110, register_a=00111100>

<그림 15 : state=100, register_b=0011 register_a=01111000>

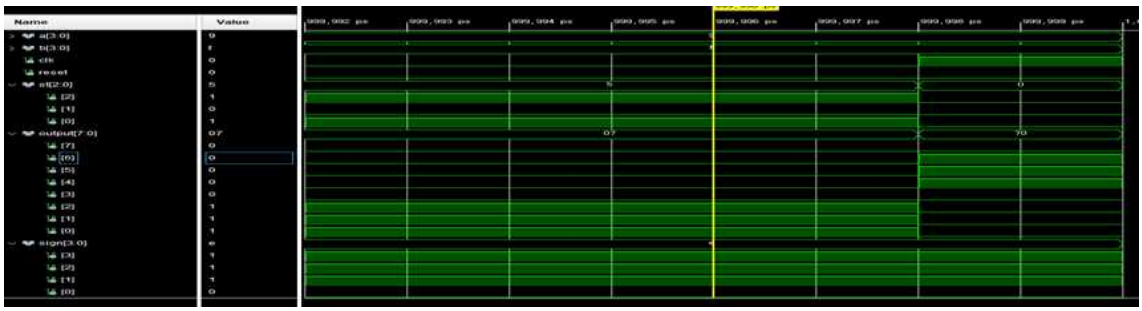


<그림 16 : state 101, register_b=1001, register_a=11110000>



<그림 17 : state 000 register_b=0000, register_a=00000000(reset)>

다음은 sum_register의 simulation 결과이다. state가 101일 때, sum_register에 답이 나와야 한다. 아래 그림을 보면 알 수 있듯이, 1001(-7)*1111(-1)=00000111(7)의 결과를 얻음을 확인할 수 있다.



<그림 18 : 1001 * 1111 곱하기 = 00000111, sign=1110(+)>

4. Multiplication 8-bit answer to BCD Output 구현

-> logisim을 이용하여 logic gate level에서 설계할때는 gate를 줄여줄 수 있는 알고리즘을 설계하였지만, vhdl에서는 코드의 가독성을 위해서, 65(0~64)가지 경우를 모두 나누어서 코드를 아래와 같이 작성했다. 그림 19는 시뮬레이션 결과이다. 63(00111111)이 input일 때, BCD code값이 10_BCD와 1_BCD가 6,3으로 정확하게 나오는 것을 확인할 수 있다.

<그림 19 : 8 bit to BCD>

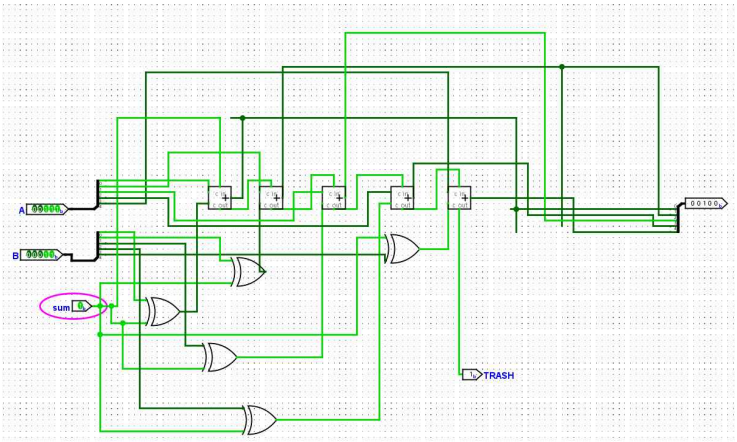
C. Sign 출력과 sel 출력

-> input, output의 sign을 +인 경우 1110, -인 경우 1101로 출력해야 했다. 또, sel 에 따라 다른 output을 출력해야 했다. signed n-bit(2's complement)인 경우 msb만 보면 부호를 판별할 수 있기 때문에 process문에서 if/else 문을 이용해서 쉽게 sign output을 구현할 수 있었다. sel도 마찬가지로 process문에서 if/else 문을 이용해서 쉽게 sel output을 구현할 수 있었다.

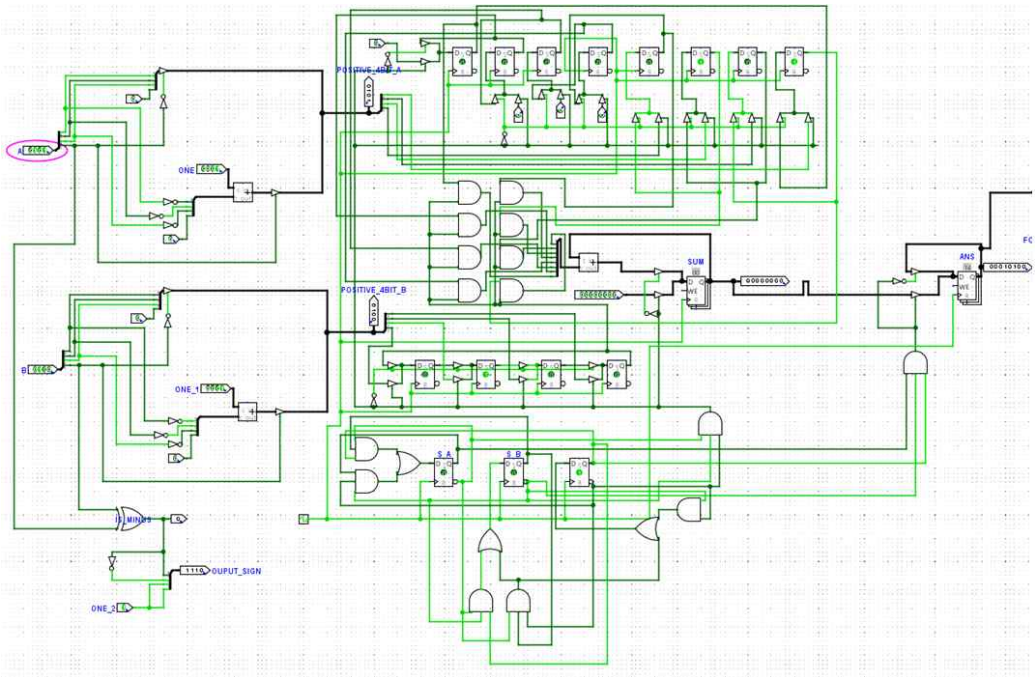
5) 부록

A) logic-gate-level design with logisim

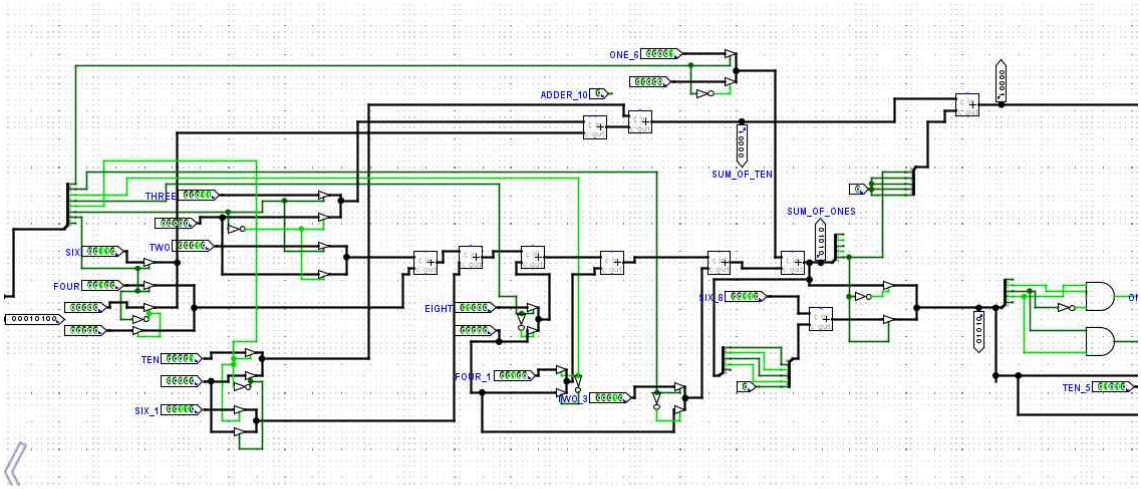
-> 우리는 본격적으로 vhdl 코드를 코딩하기 전에, 우리의 논리가 맞는지 확인하기 위해서 add/subtractor와 shift and add multiplier를 logic gate level에서 설계한 후 이를 시뮬레이션을 통해 검증하였다. 아래 그림은 logic gate level에서 설계한 add/subtractor와 shift and add multiplier이다. shift and add multiplier의 경우에는 결과값을 BCD로 바꾸는 것과 sign을 출력하는 것 까지 구현하였다.



<그림 20 : 5 bit add/subtractor 7-3 = 4>



<그림 21 :shift and add multiplier 왼쪽부분>



<그림 22 :shift and add multiplier 오른쪽부분>

B) logic-gate-level에서 8bit to BCD algorithm

Shift and Add multiplication의 output은 8-bit로 나온다. unsigned 8-bit를 BCD로 구현하기 위해서는 truth table을 사용하는 방법은 과도하게 복잡하기 때문에, 다른 알고리즘을 사용하여 BCD output을 구현했다.

I) Output signal의 8bit를 대응되는 10진수 표현으로 바꾸면 맨 위의 표와 같다. 다음 이 표에서 나온 수들을 10의 자리와 1의 자리로 분리한다(2번째 표).

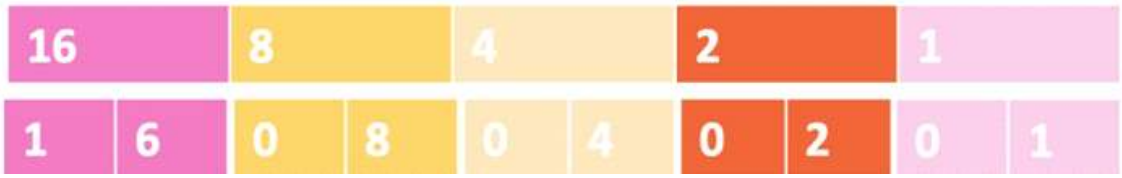
128	64	32	16	8	4	2	1
	6 4	3 2	1 6	0 8	0 4	0 2	0 1

<그림 23: 8-bit to BCD 알고리즘, bit를 decimal로 표현>

II) 8-bit output에서 1이 되는 bit에 대응되는 decimal들을 각각 10의 자리에 더하고, 1의 자리에 더해서 결과값을 출력한다.

ex) output = "0010010" =>
10의 자리 : 1+0 = 1
1의 자리 : 6+2 = 8
=> 결과 값: 18
BCD code 표현: 0001 / 1000

예외) 1의 자리에서 자리수 올림이 발생하는 경우
하지만 이렇게 계산을 하는 경우, 1의 자리에서 자리 올림이 발생하는 경우를 고려해줘야 한다. 같은 알고리즘을 한번 더 적용시켜야한다.



<그림 24 : 1의 자리에서의 자릿수 나누기>

Multiplication output의 1의 자리 최댓값은 "00111111"(binary) = 63(decimal)이므로,
MAX 1의 자리 : 2+6+8+4+2+1 = 23
10의 자리 = 3+1= 4
이 경우에는 1의 자리수가 23이기 때문에, 10의 자리에 +2를 해주고, 1의 자리에 3이 출력되어야 한다. 이를 계산한 알고리즘은 다음과 같다.

예외)-a. 1의 자릿수 합이 15 이하인 경우

이 경우에는 10의 자릿수 값들이 0이기 때문에, 값을 나누는 것이 의미가 없다. 때문에 자리수 올림이 발생하면서 값들이 15이하인, 15, 14, 13, 12, 11에 대해서는 따로 알고리즘을 추가해줘야 한다.

예외)-b. 1의 자릿수 합이 16~23인 경우

16부터 23인 경우에는 위의 알고리즘을 통해서 십의 자리 1을 분리해준다. 이후 다시 남은 1의 자리끼리 덧샘을 해준다(16~23 > 6~13). 이후 값이 15와 10 사이면 a, 10 이하이면 그대로 BCD값으로 변환해서 출력하는 연산을 해준다.

위에서 설명한 과정을 CASE별로 분류해서 설명하면 아래와 같다(1의 자리수 합 = x).

Case 1 : 16< x <23
10 자리에 +1을 해준 뒤, 1의 자리 값에 -10을 해주고, 값이 Case 2인지 Case 3인지 다시 비교해서 값을 출력한다.

Case 2 10< x <16
10의 자리에 +1을 해준 뒤 1의 자리에는 -10을 해준 값을 BCD 코드로 변환

Case 3 x <10
그대로 BCD 코드로 변환