컴퓨터 공학 기초 실험2 보고서

실험제목: Multiplier

실험일자: 2023년 11월 13일 (월)

제출일자: 2023년 11월 22일 (수)

학 과: 컴퓨터정보공학부

담당교수: 이준환 교수님

실습 분반: 월요일 0, 1, 2

학 번: 2020202031

성 명: 김재현

1. 제목 및 목적

A. 제목

Multiplier

B. 목적

Multiplier는 multiplicand와 multiplier를 곱하여 결과값을 도출하는 hardware입니다. Multiplicand와 multiplier의 각각의 bit length는 64bits이며, 곱의 결과값은 128bits입니다. 64bist multiplier와 multiplicand를 입력 받아 booth multiplier를 진행하는 곱셈기를 구현해봅니다.

2. 원리(배경지식)

1) Carry Look-ahead Block

$$G_i = A_i B_i$$
$$P_i = A_i + B_i$$

라고 정의 하고 full adder의 carry out에 적용하면 다음과 같습니다.

$$C_{i+1} = A_i B_i + (A_i + B_i) C_i = G_i + P_i C_i$$

이를 적용하여 4-bits CLA를 위한 carry를 미리 계산하면 다음과 같습니다.

$$C_1 = A_0 B_0 + (A_0 + B_0) C_0 = G_0 + P_0 C_0$$
$$C_2 = A_1 B_1 + (A_1 + B_1) C_1 = G_1 + P_1 C_1$$
$$C_3 = A_2 B_2 + (A_2 + B_2) C_2 = G_2 + P_2 C_2$$
$$C_{out} = A_3 B_3 + (A_3 + B_3) C_3 = G_3 + P_3 C_3$$

C_{out}*을* C_1, C_2, C_3*을 사용하여 정리하면*

$$C_{out} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$$

2) 32-bits CLA with clock

32-bit cla에 앞 뒤로 flip-flop을 추가하여 clock과 연결하는 module이다. 해당 모듈에서 clock을 인가하여 valid한 결과가 나오는 데 필요한 clock period를 확인하는 것을 목표로 한다.

3) 128-bits CLA with clock

32-bits cla를 4개 이어 붙인다. 이전 CLA의 co를 다음 CLA의 cin으로 전달해주면 128-bits cla가 된다.

4) cal_radix4

multiplicand의 마지막 3비트를 읽어서 어떤 연산을 진행할지 판단한 후 현재 진행중인 연산 값을 shift하거나, multiplier와 덧셈 혹은 뺄셈 연산을 진행한다.

마지막 3비트가

000, 111일 경우: ASR ASR

001, 010일 경우: ADD ASR ASR

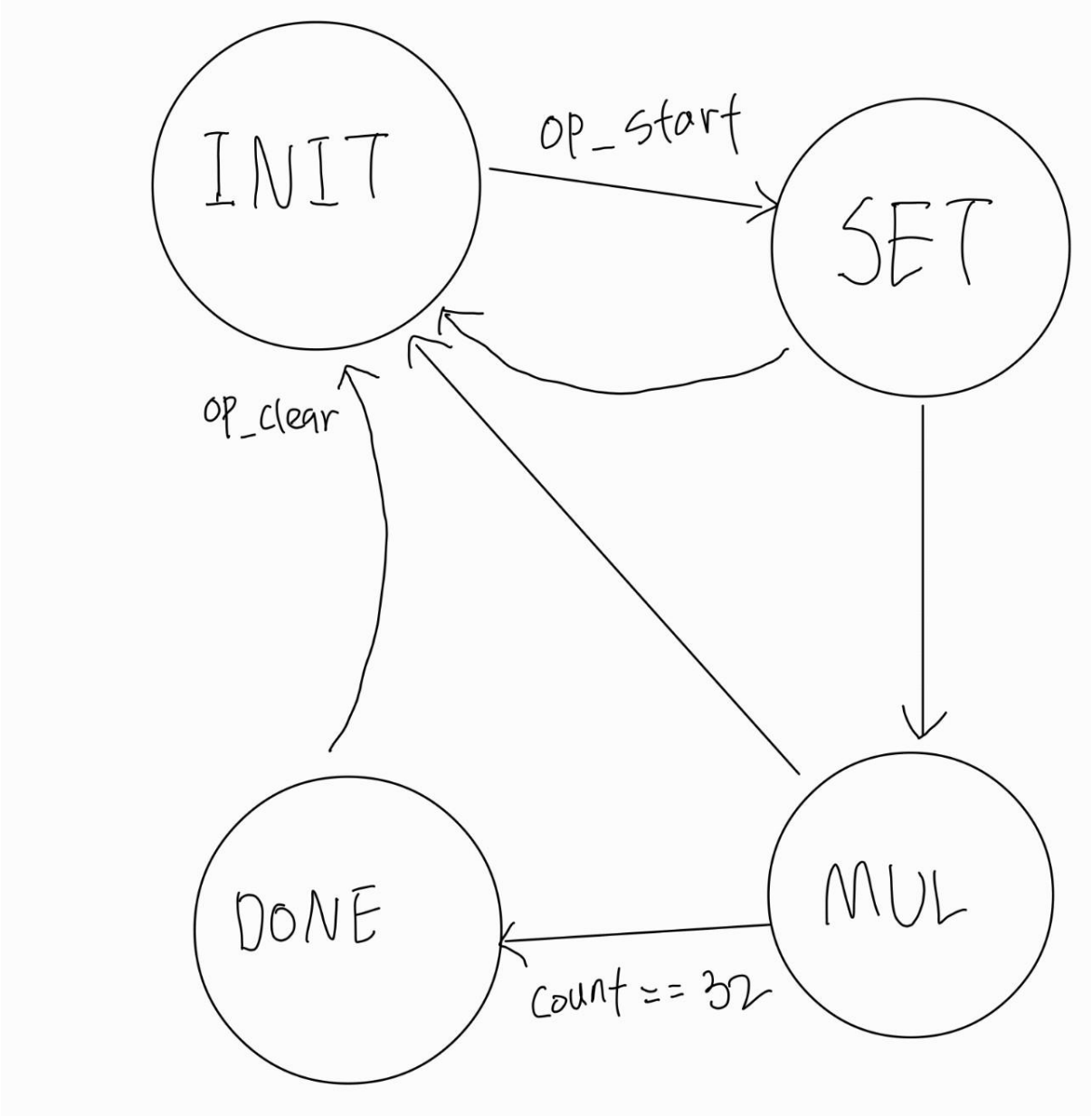
011일 경우 : ASR ADD ASR

100일 경우 : ASR SUB ASR

101, 110일 경우: SUB ASR ASR

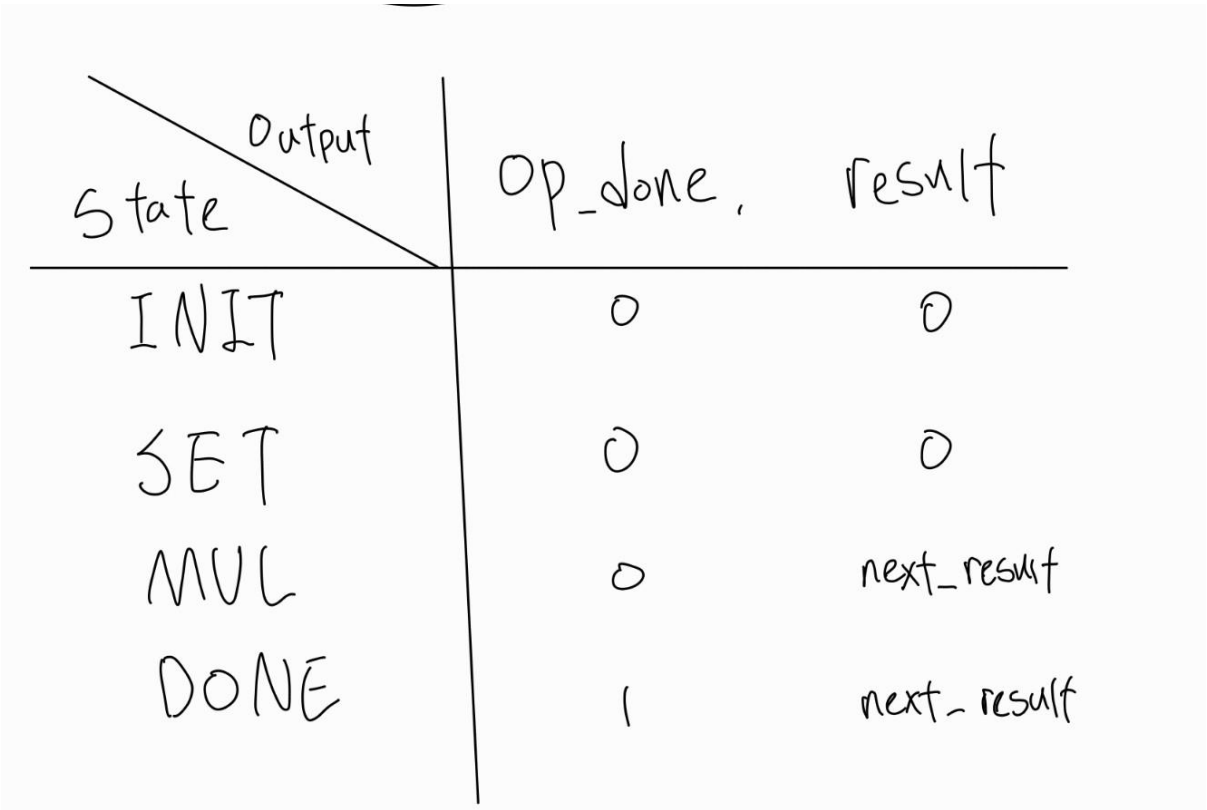
순서로 연산을 진행합니다.

3. 설계 세부사항



booth_multiplier의 state diagram입니다. op_start, op_clear, count를 조건으로 next_state를 계산하게 됩니다. INIT에서 op_start가 1일 때 clk rising

edge를 만나면 SET으로, SET에서 clk rising edge를 만나면 MUL로, MUL에서 count가 32가 되면 DONE으로 state가 이동하게 되고, 어떤 state에서든 op_clear가 1일 때 clk rising edge를 만나면 INIT으로 이동합니다.



booth_multiplier의 각 state에 따른 OUTPUT 값을 나타낸 진리표입니다.

state가 INIT일 때, op_done은 0, result는 0을 출력합니다.

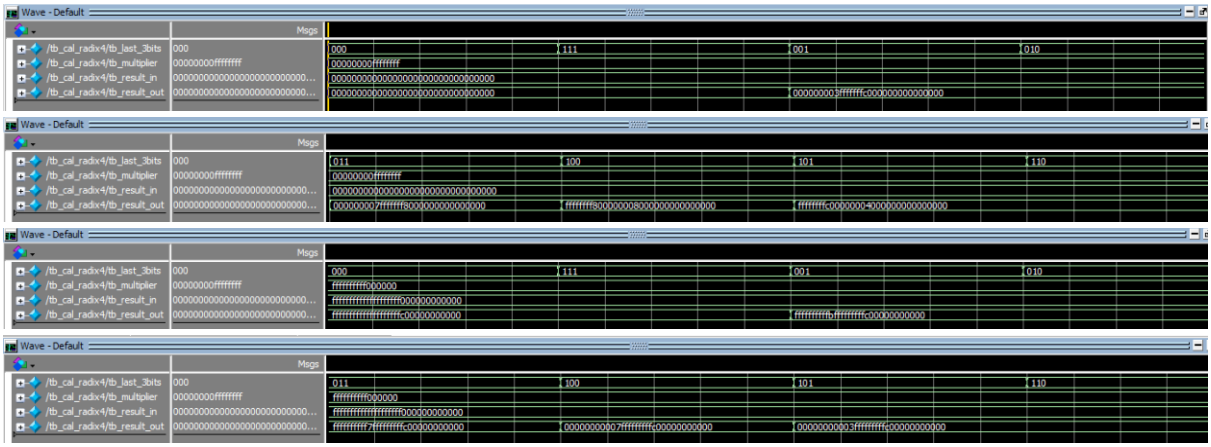
state가 SET일 때, op_done은 0, result는 0을 출력합니다.

state가 MUL일 때, op_done은 0, result는 cal_radix4를 통해 연산이 진행된 값을 출력합니다.

state가 DONE일 때, op_done은 1, result는 cal_radix4를 통해 연산이 진행된 값을 출력합니다. DONE에서 result가 next_result를 출력하는 이유는 next_result이 이미 이전 사이클에서 cal_radix4를 통해 도출된 결과 값이므로 반영해줘야하기 때문입니다.

4. 설계 검증 및 실험 결과

   A. 시뮬레이션 결과



tb_cal_radix4의 testbench입니다.

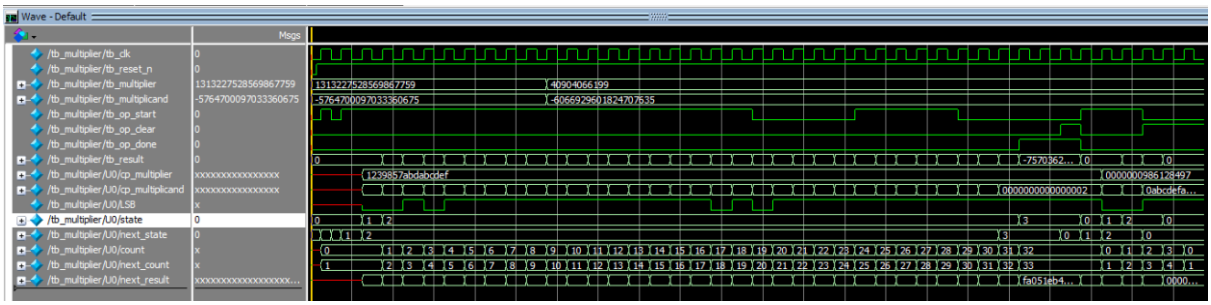
tb_last_3bis가 000, 777이면 tb_result_in을 ASR shift 2번 해준 결과 값을 tb_result_out으로 출력해줍니다.

tb_last_3bis가 001, 010이면 tb_result_in에 tb_multiplier를 더한 값을 ASR shift 2번 해준 결과 값을 tb_result_out으로 출력해줍니다.

tb_last_3bis가 011이면 tb_result_in를 ASR shift 1번 해준 값에 tb_multiplier를 더한 값을 ASR shift 1번 해준 결과 값을 tb_result_out으로 출력해줍니다.

tb_last_3bis가 100이면 tb_result_in를 ASR shift 1번 해준 값에 tb_multiplier를 빼준 값을 ASR shift 1번 해준 결과 값을 tb_result_out으로 출력해줍니다.

tb_last_3bis가 001, 010이면 tb_result_in에 tb_multiplier를 빼준 값을 ASR shift 2번 해준 결과 값을 tb_result_out으로 출력해줍니다.



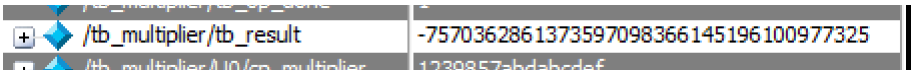
tb_multiplier의 testbench입니다.

tb_multiplier, tb_multiplicand, tb_result, count, next_count는 Decimal로,
나머지는 전부 Hexadecimal로 표기했습니다.

op_start가 1인 상황에서 clk의 rising edge를 만나면, state가 SET이 되고, cp_multiplier, cp_multiplicand를 multiplier, multiplicand로 초기화해줍니다. 다음 clk rising edge에서 state가 MUL로 넘어가게 됩니다. state가 MUL로 넘어감과 동시에 count가 1씩 증가하고, 한 사이클의 연산 결과가 출력됩니다. 연산이 진행되는 과정에서 count가 증가함에 따라,

state MUL에서의 연산도 계속해서 진행됩니다. state MUL인 상태에서 multiplier와 multiplicand가 바뀌고 op_start가 0과 1로 반복해서 스위칭해도 연산에 영향을 주지 않는 것을 확인할 수 있습니다.

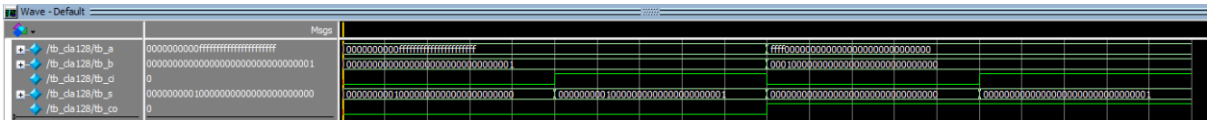
처음 입력된 multiplier 값 10진수 131322752856986775 9와 multiplicand 값 10진수 -57647000970333606 75를 곱한 결과로



다음과 같이 -757036286137359709836614519610097732 5 가 출력되는 것을 확인할 수 있습니다.

state가 DONE일 때 op_clear이 1이면 다음 clk rising edge에서 state가 INIT으로 넘어갑니다. state가 MUL일 때도 마찬가지로 op_clear가 1이면 다음 clk rising edge에서 state가 INIT으로 넘어갑니다.

state가 DONE일 때는 op_done이 1을 출력하고, 나머지 state에서는 op_done이 0을 출력하는 것을 확인할 수 있습니다.

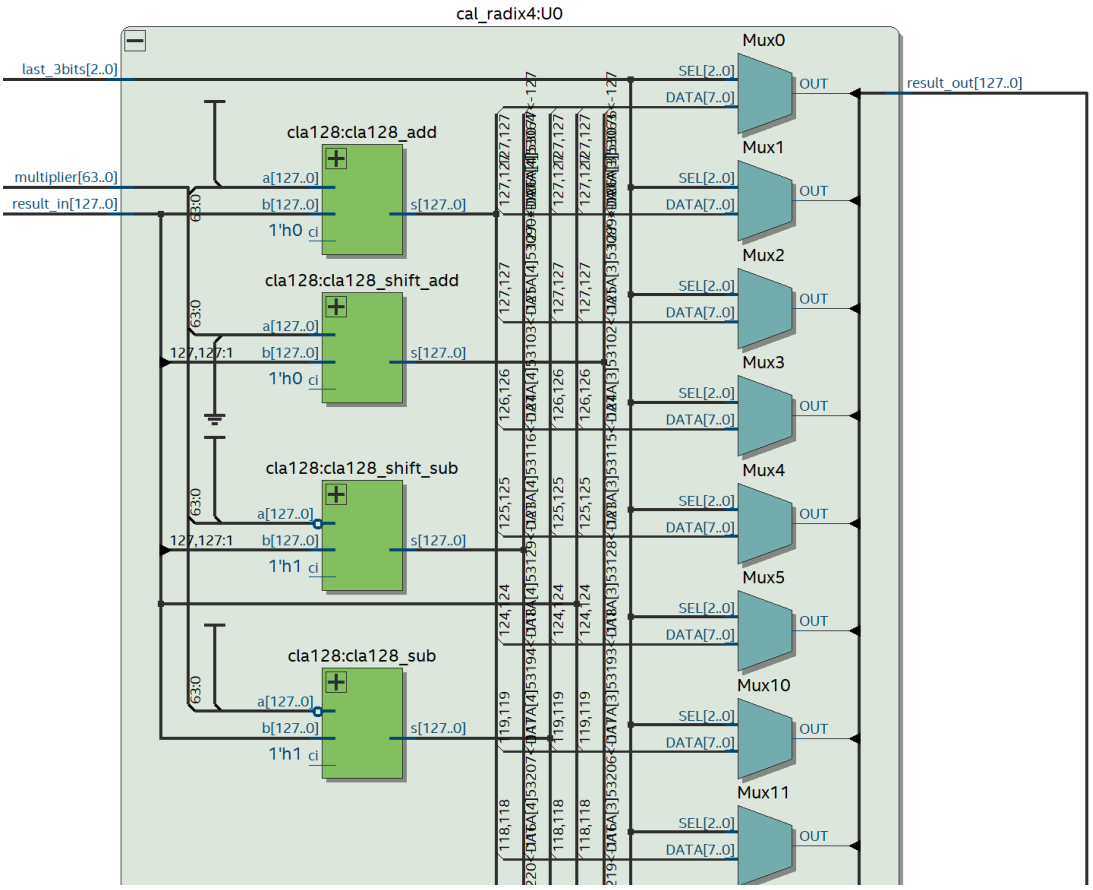


tb_cla128의 testbench입니다.

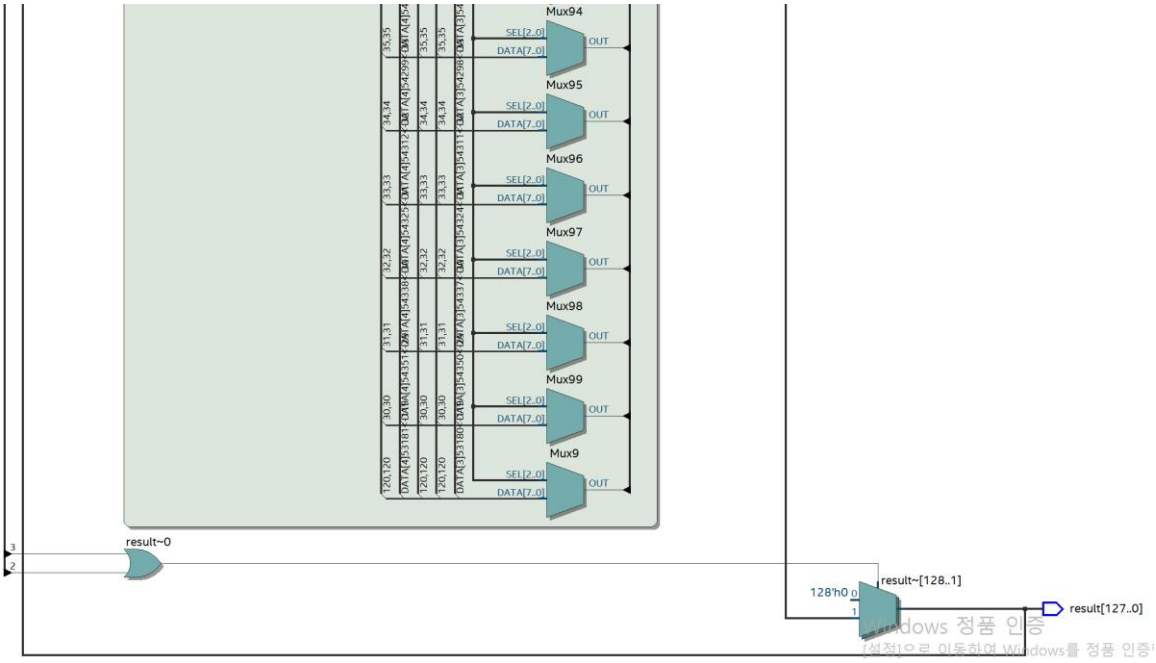
덧셈 결과에 따라 co와 s가 잘 출력되는 것을 확인할 수 있습니다.

B. 합성(synthesis) 결과

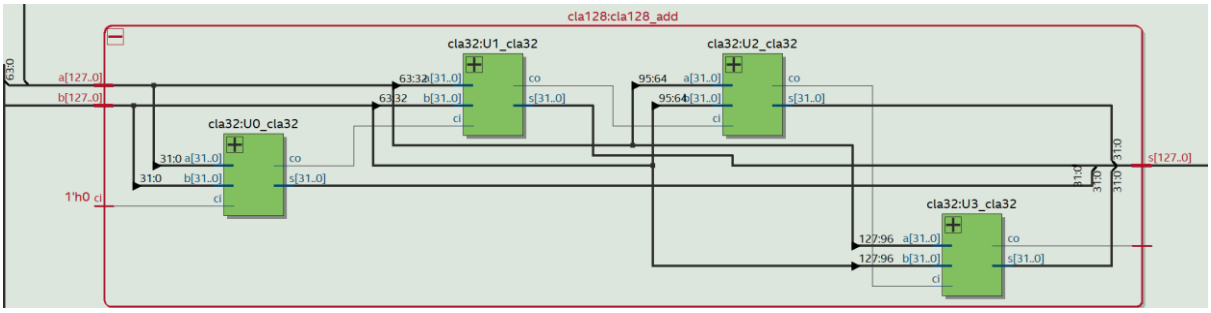
cal_radix4의 RTL Viewer입니다.



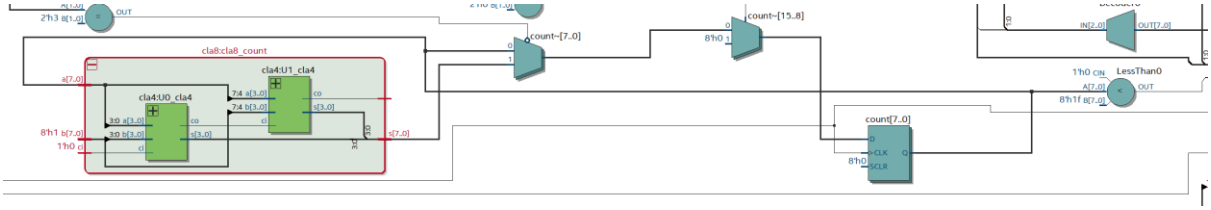
3bits로 만들 수 있는 8가지의 연산 경우의 수 중 last_3bits로 한 가지의 연산만을 선택하여 128비트 모든 자리수에 연산을 진행시킵니다.



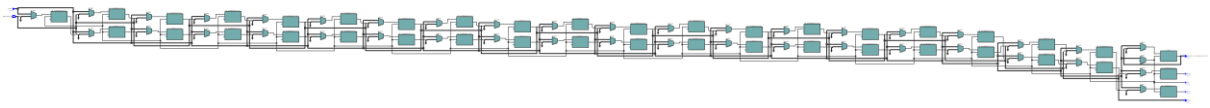
연산을 완료하면 현재 result로 출력함과 동시에 다시 cal_radix4의 입력으로 들어갑니다.



cal_radix4 내부에 존재하는 cla128의 RTL Viewer입니다. cla32를 4개가 연결되어 128비트 cla를 구성하고 있는 것을 확인할 수 있습니다.



cla8의 RTL Viewer입니다.



multiplier의 RTL Viewer입니다. case문과 if문을 중복해서 많이 쓰다 보니 latch가 좌우로 길게 늘어뜨러진 것을 확인할 수 있습니다.

Flow Summary

<<Filter>>

Flow Status	Successful - Wed Nov 22 20:50:04 2023
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	multiplier
Top-level Entity Name	multiplier
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	506 / 41,910 (1 %)
Total registers	10
Total pins	261 / 499 (52 %)
Total virtual pins	0
Total block memory bits	0 / 5,662,720 (0 %)
Total DSP Blocks	0 / 112 (0 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 15 (0 %)
Total DLLs	0 / 4 (0 %)

Logic utilization은 506이고, Total pin은 clk 1bit + reset_n 1bit + multiplier 64bits + multiplicand 64bits + op_start 1bit + op_clear 1bit + op_done 1bit + result 128bits = 261 입니다.

5. 고찰 및 결론

A. 고찰

cal_radix4를 구현할 때, cla 인스턴스 하나로 모든 경우의 연산을 다 하려고 계획했었습니다. 그러나 result_in을 shift한 후 덧셈 혹은 뺄셈을 진행하는 경우, shift 없이 바로 덧셈 혹은 뺄셈을 진행하는 경우로 총 4가지의 경우의 수가 나오기 때문에 cla 인스턴스를 4개 선언해서 구현해줬습니다.

B. 결론

정리하자면, 하드웨어 설계는 모든 경우의 수에 대해 모듈의 인스턴스를 하나하나 다 구현해줘야 한다는 것을 알 수 있었습니다.

radix2뿐만 아니라 radix4를 통해 booth multiplier를 직접 구현해 봄으로써 booth multiplier의 작동원리와 구조를 잘 이해할 수 있었습니다.

6. 참고문헌

이준환 교수님/디지털논리회로2/광운대학교(컴퓨터정보공학부)/2023

이준환 교수님/컴퓨터공학기초실험2/광운대학교(컴퓨터정보공학부)/2023