컴퓨터 공학 기초 실험2 보고서

실험제목: Multiplier

실험일자: 2023년 11월 03일 (수)

제출일자: 2023년 11월 20일 (월)

학 과: 컴퓨터정보공학부

담당교수: 이준환 교수님

실습분반: 수요일 0, 1, 2

학 번: 2022202075

성 명: 우나륜

1. 제목 및 목적

A. 제목

Multiplexer

B. 목적

이전의 과제에서 Verilog로 구현한 Carry Look-ahead Adder를 사용하여 64-bit의 두 값을 곱하여 128-bit의 연산결과를 계산하는 multiplier를 구현할 수 있다. 연산자 op_start, op_clear, op_done을 사용하여 multiplier의 흐름을 조작하고, clock의 한 cycle마다 multiplier의 진행 단계를 출력한다.

2. 원리(배경지식)

A. Multiplier

Multiplier는 곱셈기로 디지털 회로에서 두 이진값을 곱하는 목적을 가지는 하드웨어 회로이다. Multiplier는 연산자로 multiplicand (피승수)와 multiplier (승수)를 가진다. 64-bit multiplicand에 64-bit multiplier를 곱하여 128-bit의 결과값을 계산하는 것이 본 과제의 목적이다. 본 과제에서 설계하는 multiplier는 입력값으로 multiplicand와 multiplier 뿐만 아니라 op_start, op_clear, rst, clk를 추가로 갖는다. 연산자 op_start를 사용하여 곱셈기의 시작 흐름을 조작하고, op_clear를 사용하여 multiplier 내의 모든 레지스터의 값들을 0으로 초기화시킨다. 또한, reset의 역할을 수행하는 rst를 사용하여 multiplier를 초기화할 수 있고, clock인 clk을 사용하여 clock의 rising edge에서 결과값이 할당되어 결과화면에 나타날 수 있도록 한다.

B. Booth multiplication algorithm

Booth multiplication algorithm은 2의 보수 표기법을 사용하여 두 개의 부호 (+, -)에 있는 이진수를 곱하는 곱셈 알고리즘이다. 이 알고리즘은 Andrew Donald Booth가 발명하여 그의 이름을 따 booth라는 이름이 붙게 되었다. Booth algorithm은 최하위 두 개의 비트를 사용하여 곱셈을 진행한다.

x_i	x_{i-1}	Operation	Description	y_i
0	0	Shift only	String of zeros	0
0	1	Add and shift	End of a string of ones	1
1	0	Subtract and shift	Beginning of a string of ones	$\bar{1}$
1	1	Shift only	String of ones	0

Table 1. Radix-2 Booth Multiplication table

[출처] 광운대학교 이준환 "Booth multiplication" 강의자료

본 과제에서 사용한 booth multiplication은 2-radix로 multiplicand의 맨 마지막 비트와 직전 연산에 사용한 비트를 사용하여 operation을 결정한다. 위에 표에서 확인할 수 있듯이, x_i는 현재 multiplicand의 최하위 비트이며 x_{i-1}은 직전 연산에 사용한 비트이고, 00과 11일때는 arithmetic right shift 연산만 수행한다. 만약 x_i=1, x_{i-1}=0이면, multiplier의 보수를 취한 뒤, 더하고 arithmetic right shift 연산을 수행한다. 만약, x_i=0, x_{i-1}=1이면, multiplier의 값을 그대로 더하고 arithmetic right shift 연산을 수행한다. 또한, booth multiplication algorithm은 3비트씩 보면서 연산하는 4-radix도 존재한다. 이러한 곱셈 연산 알고리즘은 계산 속도가 빠르고 덧셈 및 shift 연산을 최적화 하여 곱셈 연산을 수행할 수 있다. 또한, 이진수의 덧셈과 shift 연산을 사용하기 때문에 하드웨어로 간단하게 구현할 수 있다는 장점이 있다. 하지만, booth algorithm은 부분 곱셈을 하기 위해 추가로 연산을 수행해야 하고, 큰 숫자를 다룰 때 overflow 문제가 발생할 수 있으며, 정확도가 떨어진다는 단점을 가지게 된다.

3. 설계 세부사항

아래의 Figure 1은 top module인 multiplier의 schematic symbol을 나타낸다. Module multiplier는 외부로부터 연산할 두 대상인 multiplier (승수)와 multiplicand (피승수)를 입력받고, module의 연산 흐름을 제어하는 operation인 op_start, op_clear와 rst (reset), clk (clock)을 입력받는다. Multiplier내의 연산을 통해 연산이 완료되면 op_done = 1이되고 결과값은 매 cycle마다 result를 통해 얻을 수 있다.

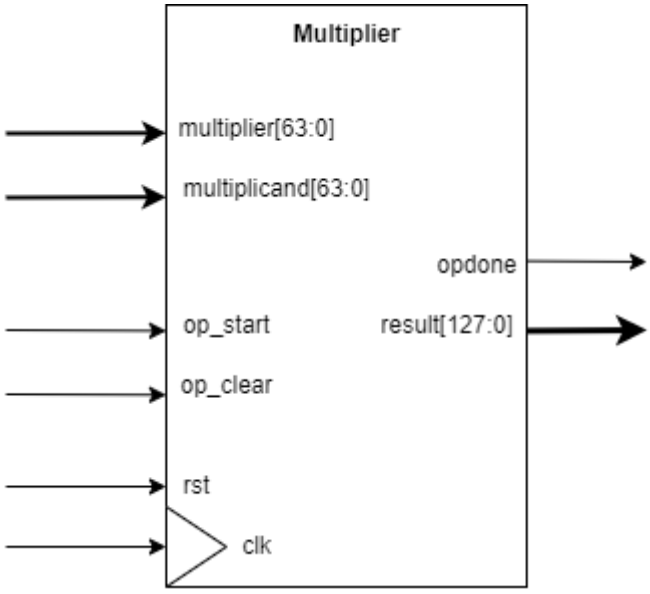


Figure 1. Schematic symbol of multiplier

본 과제에서 설계한 multiplier의 state를 나타내면 아래의 Table 2와 같다. 상태는 총 3가지 INIT, EXEC, DONE을 가지고 각각, 초기 상태 또는 clear = 1인 상태(INIT), 곱셈 연산이 수행되고 있는 상태(EXEC), 곱셈 연산이 완료된 상태(DONE)을 나타낸다. 각 상태를 binary encoding을 사용하여 INIT = 2'b00, EXEC = 2'b01, DONE = 2'b10로 정의하였다. 각 모듈 내에서 이 상태들은 parameter로 선언되어 사용된다.

State	Description	Binary encoding
INIT	Initial and clear state	00
EXEC	Calculation executing state	01
DONE	Calculation done state	10

Table 2. State encoding table of multiplier.

정의한 상태를 바탕으로 multiplier의 state transition diagram을 그리면 아래의 Figure 2와 같다. 만약, op_start = 0 또는 op_clear = 1이면 INIT 상태를 갖는다. 만약 op_start = 1 && op_done = 0 && op_clear = 0인 상태가 되면 입력받은 두 수를 곱셈하는 상태 EXEC가 된다. 64-cycle을 계산하는 cnt의 값이 0이 아니고 op_clear = 0이면 64 사이클이 될 때까지 연산을 수행한다. 그런데 EXEC 상태 도중에 op_clear의 값이 1이 되면 연산을 멈추고 INIT 상태로 돌아간다. 연산이 모두 완료되고 cnt의 값이 0이 되면, DONE 상태가 된다. 이때, op_clear의 값이 1이 되기 전까지 연산한 결과의 값을 유지한다. 그리고 op_clear = 1이되면 INIT 상태로 돌아간다.

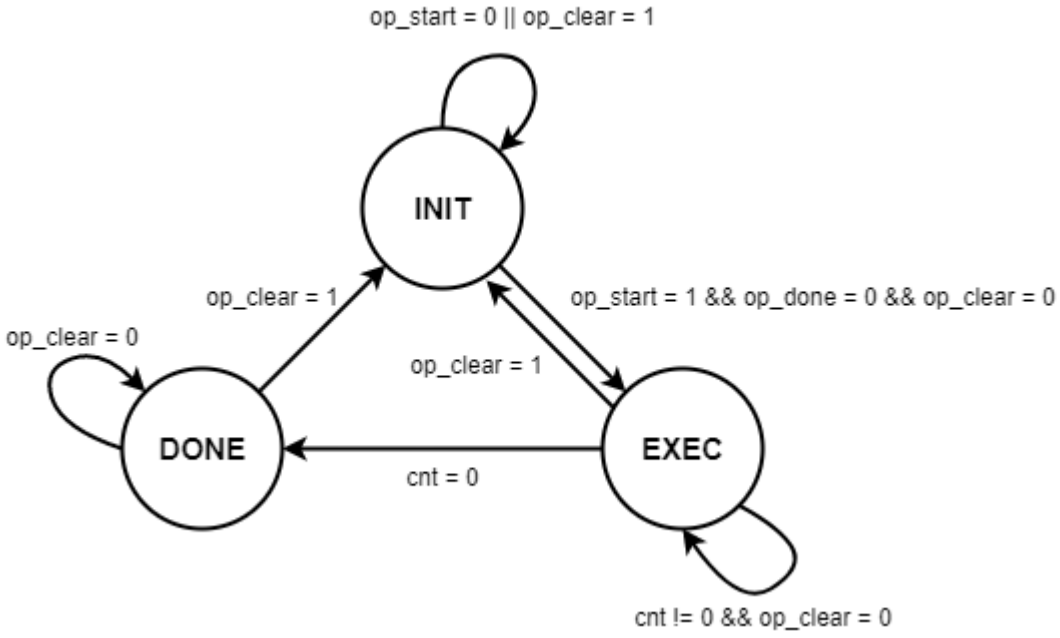


Figure 2. State diagram of multiplier

State transition diagram을 설계한 후, multiplier에 필요한 대표적인 module의 input/output을 표로 정리하면 Table 3과 같다. 본 과제의 top module인 multiplier와 multiplier의 next state를 연산할 multiplier_ns, EXEC 상태에서 연산을 수행할 multiplier_os와 그 외의 module들로 이루어져 있다.

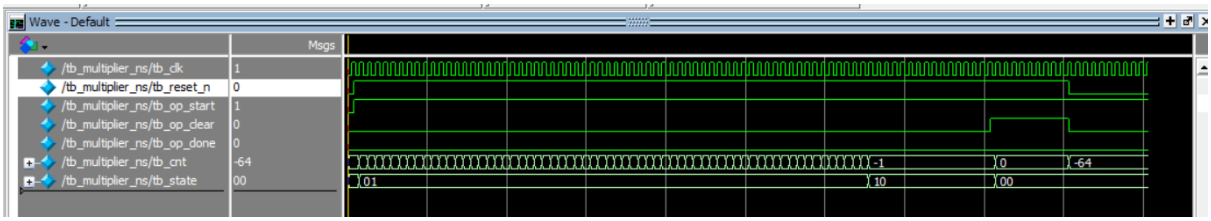
Module name	**Direction**	**Port name**	**Bit**	**Description**
multipler	Input	clk	1-bit	Clock
		reset_n		Active-low reset
		multiplier	64-bit	승수
		multiplicand		피승수
		op_start	1-bit	Start operation
		op_clear		Clear operation
	Output	op_done		Done operation
		result	128-bit	Multiplier result
multiplier_ns	Input	clk	1-bit	Clock
		reset_n		Active-low reset
		op_start		Start operation
		op_clear		Clear operation
		op_done		Done operation
	Output	cnt	7-bit	64-cycle counter
		state	2-bit	Module current state
multiplier_os	Input	clk	1-bit	Clock
		reset_n		Active-low reset
		op_clear		Clear operation
		state	2-bit	Module current state
		cnt	7-bit	64-cycle counter
		multiplier	64-bit	승수
		multiplicand		피승수
	Output	op_done	1-bit	Done operation
		result	128-bit	Multiplier result

Table 3. Module I/O configuration

4. 설계 검증 및 실험 결과

A. 시뮬레이션 결과

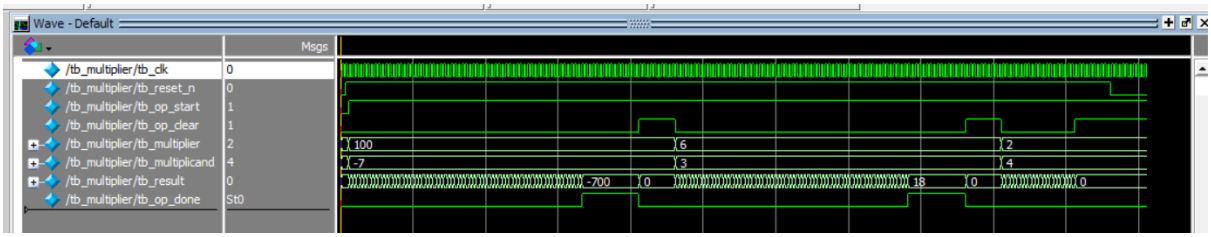
1) Testbench for multiplier_ns.v



위의 그림은 multiplier_ns의 testbench waveform을 전체적으로 본 모습이다. 64-cylcle 동안 EXEC 상태에서 cnt를 계산한 후 cnt의 값이 -1이 되었고, op_clear = 1이 되자 cnt의 값이 0으로 초기화 된 것을 확인할 수 있다. 그리고 op_clear = 0이 되어 다시 7'b1000000의 값을 가지는 것을 알 수 있다.

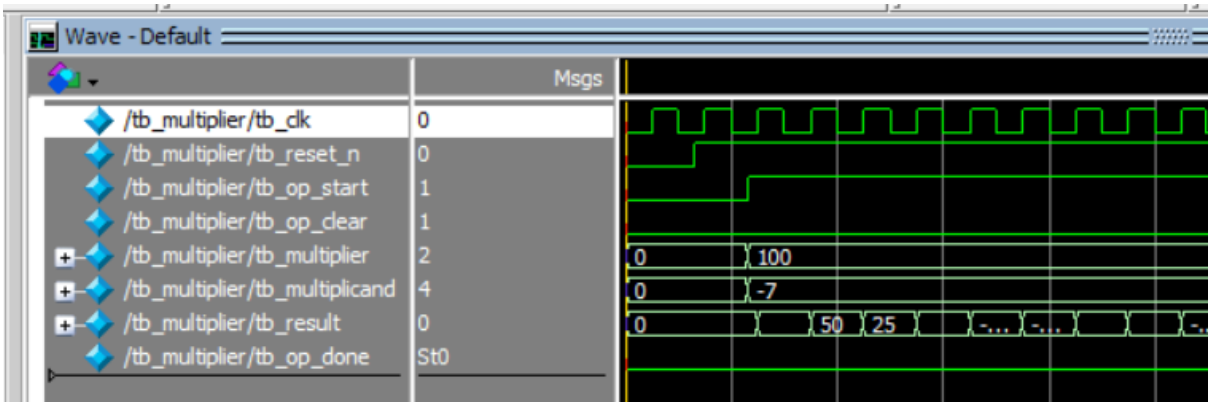
위 그림들을 통해 64-cycle이 제대로 수행되고 이에 따른 State도 변화하는 것을 확인할 수 있다.

2) testbench for multiplier.v

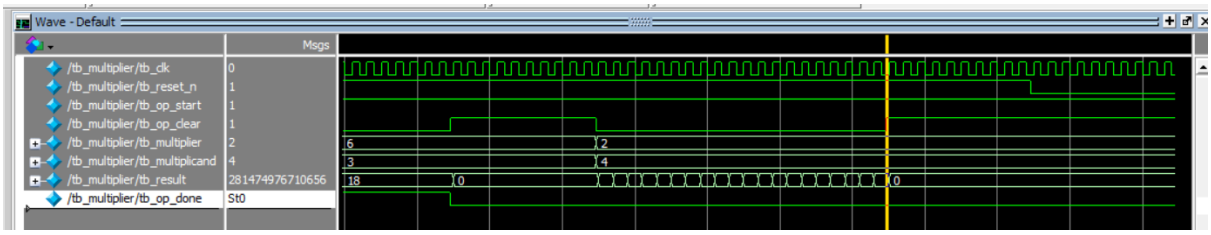


위의 결과화면은 top module인 multiplier.v의 테스트벤치 waveform이다. 처음에 100과 -7를 할당하고, 64-cycle 이후에 결과값인 -700이 올바르게 계산되고 op_done = 1인 것

을 확인할 수 있다. 그리고 op_clear = 1이 되어 0으로 값이 초기화 되었다. 두번째로 6과 3을 할당하고 64-cycle 이후에 18이 계산되고 op_done = 1인 것을 확인할 수 있다. 그리고 op_clear를 사용하여 결과값을 초기화하였다. 세번째로 2와 4를 할당하고, cycle이 끝나기 전에 op_clear = 1로 바꾸어 연산을 중단하였고 그에 대한 결과값으로 0이 출력되는 것을 확인할 수 있다.



맨 처음 module이 실행되었을 때, multiplier, multiplicand, result의 값이 0으로 초기화되어 있는 거슬 확인할 수 있다.

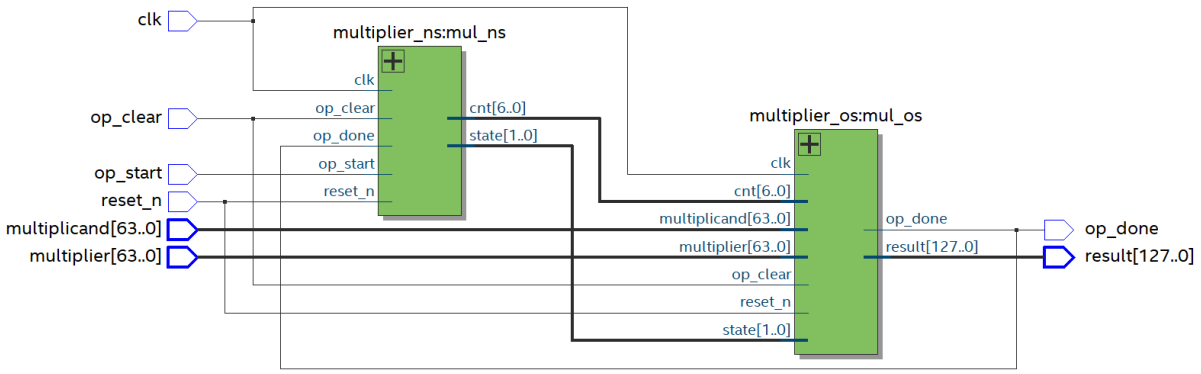


세번째 연산을 진행하는 도중에 op_clear = 1이 되어 연산이 중단되고 0을 출력하는 것을 확인할 수 있다.

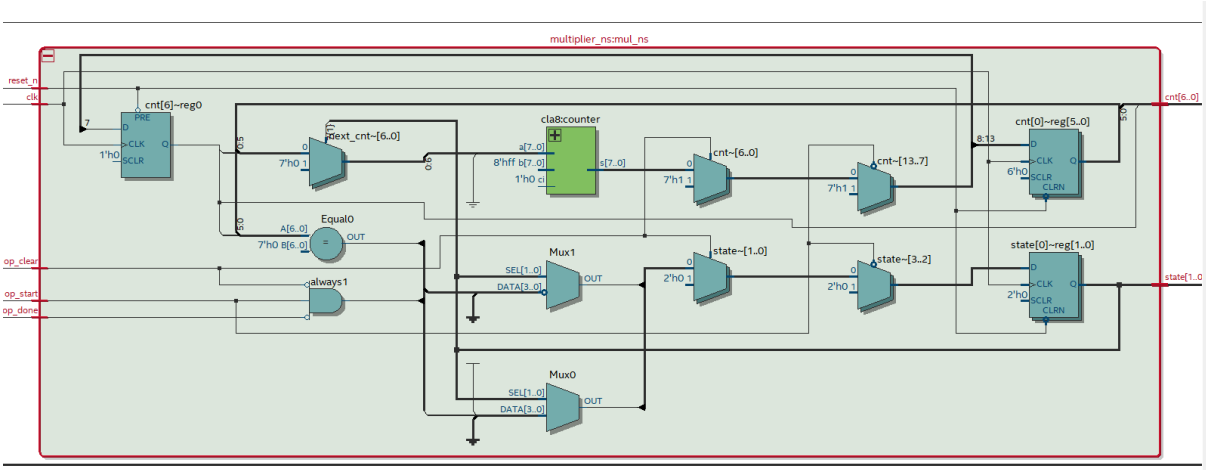
B. 합성(synthesis) 결과

Flow Summary	
Flow Status	Successful - Mon Nov 20 17:53:04 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)	327 / 41,910 (< 1 %)
Total registers	244
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 %)

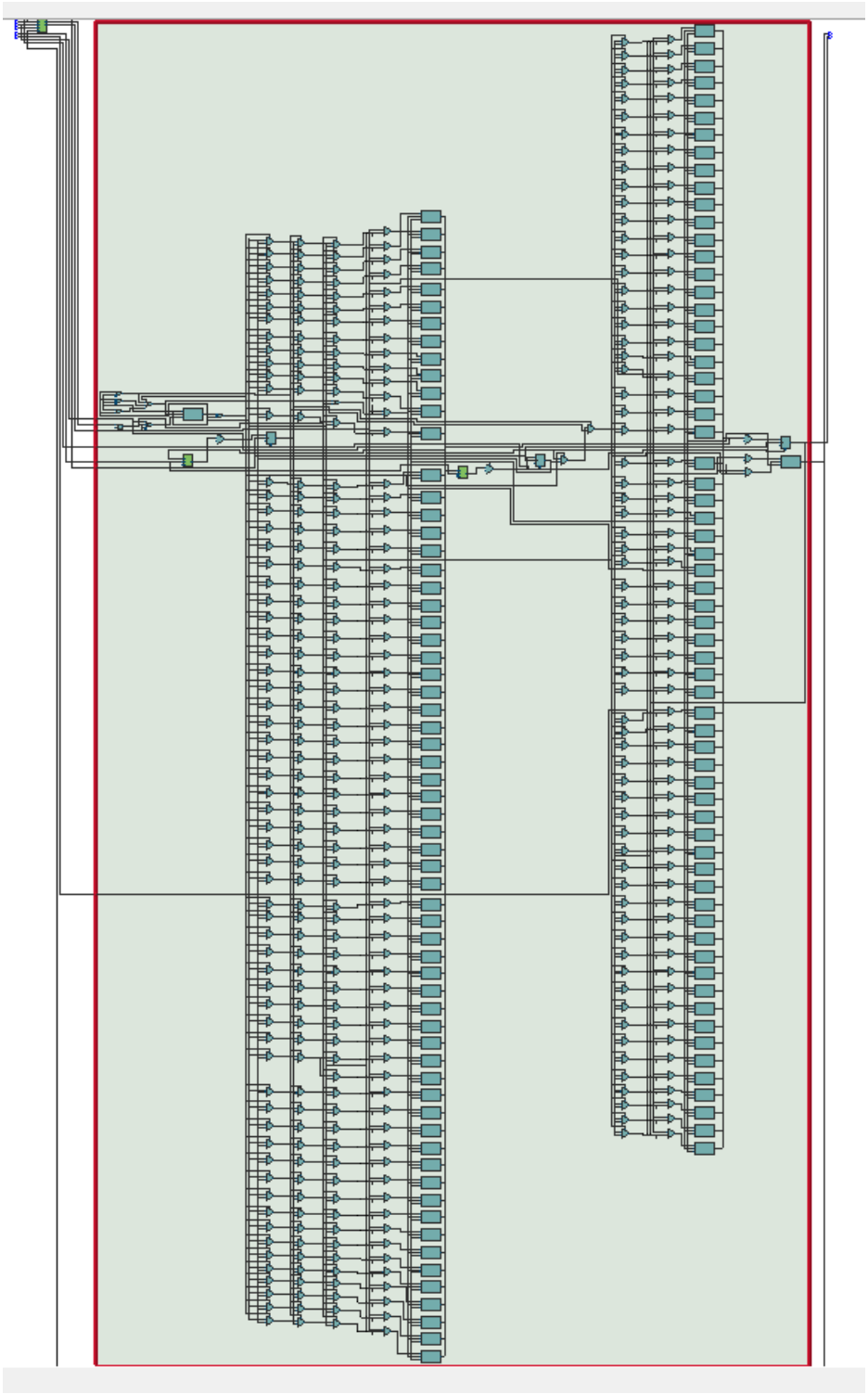
본 과제에서 multiplier는 logic utilization이 327 로 1%미만이고 total registers = 244, total pins = 261 (52%)인 것을 확인할 수 있다.



위의 그림은 multiplier의 RTL map viewer이다. 해당 top module내에 두개의 module인 multiplier_ns와 multiplier_os가 instance된 것을 확인할 수 있다.



위 그림은 multiplier_ns instance를 확대한 것이다.



위 그림은 multiplier_os instance를 확대한 것이다.

5. 고찰 및 결론

A. 고찰

1. "A time value could not be extracted from the current line" 오류가 발생하였다. 이는

testbench에서 2000ns를 실행시켰는데 해당 딜레이 타임이 너무 커 발생한 오류였다. 따라서 실행 ns를 800ns로 줄여주니 해결되었다.

2. <multiplier_os> module에서 cla64 instance가 실행되지 않는 오류가 발생하였다. 64bits multiplicand의 inverse 값을 저장하기 위해 _inv_64bits를 생성한 후 wire인 inv_multiplicand 에 저장하였었다. 하지만 testbench를 실행하니 _inv_64bits에서 다음 instance로 넘어가지 않아 testbench가 실행되지 않았다. 따라서 multiplicand의 inverse값을 따로 저장하지 않고 cla64의 SUB instance의 인자로 ~를 붙여 바로 넣어주니 해결할 수 있었다.
3. <multiplier_ns>에서 next_cnt의 값을 구하기 위해 8-bit CLA를 구현하였다. 하지만 counter는 최대 64까지로 7비트이기 때문에 그대로 값을 넣을 수 없었다. 따라서 결합 연산자 ({ })를 사용하여 7-bit counter의 MSB에 0을 붙여 8-bit CLA에 넣어 -1를 더해주어 값을 구하였다. 또한, 발생하는 carry는 필요가 없으므로 공백으로 두어 값을 저장하지 않았다.
4. 첫번째 곱셈 연산에서는 완전히 64 cycle이 수행되어 올바른 결과값이 나왔지만, 두번째 연산에서는 63 cycle만이 수행되어 올바른 값이 계산되지 않았다. 이는 multiplier_ns module에서 op_clear = 1일 때, counter의 값을 7'b1000000로 고정시키지 않아 7'b1000000 - 1이 된 값으로 INIT 상태가 되었고 이로 인해 1cycle이 덜 수행된 것이었다.

B. 결론

1. 본 과제에서 구현한 multiplier는 booth multiplication algorithm을 사용하여 구현하였다. 그 중에서도 2개의 값을 가지고 연산을 수행하는 2-radix로 구현하였다. 이 multiplier는 위에서 언급한 것과 같이 계산 속도가 빠르고 덧셈 및 shift 연산을 최적화하여 연산을 수행한다. 따라서 이진수의 덧셈과 shift 연산을 사용하기 때문에 하드웨어로 간단하게 구현할 수 있다.
2. Booth multiplication algorithm을 사용한 multiplier는 이후의 factorial computation system에 팩토리얼을 계산하는데 사용될 수 있다.

6. 참고문헌

[1] 곱셈기 / https://ko.wikipedia.org/wiki/%EA%B3%B1%EC%85%88%EA%B8%B0

[2] 이준환 교수 / Booth Multiplication / 광운대학교 컴퓨터정보공학부 / 2023년

[3] Verilog 결합연산자 / https://wh00300.tistory.com/166

[4] Booth multiplier algorithm / https://zrr.kr/d4fm

[5] Multiplication / https://hi-guten-tag.tistory.com/254