

컴퓨터 공학 기초 실험2 보고서

실험제목: Multiplier

실험일자: 2023년 11월 06일 (월)

제출일자: 2023년 11월 22일 (수)

학 과: 컴퓨터공학과

담당교수: 이준환 교수님

실습분반: 월요일 0, 1, 2

학 번: 2022202064

성 명: 최봉규

1. 제목 및 목적

A. 제목

Multiplier

B. 목적

이진수의 곱셈 연산이 음수(2의 보수)의 경우에도 통용되는 multiplier를 학습한다. Booth algorithm을 이용하여, 직접 multiplicand와 multiplier의 곱을 계산하는 프로그램을 설계하는 실습을 진행한다. 본인이 설계한 multiplier가 제대로 된 결과값을 출력하는지 검증한다.

2. 원리(배경지식)

Multiplier는 multiplicand(피승수)와 multiplier(승수)를 곱하여 결과값을 도출하는 hardware이다. Multiplicand와 multiplier의 각각의 bit length는 64bits이며, 곱의 결과값은 그의 두배인 128bits로 나온다.

다음은 radix-2인 경우 booth multiplication의 규칙이다.

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	-1
1	1	Shift only	String of ones	0

다음은 radix-2의 경우 booth algorithm을 이용해 계산하는 예시이다.

Ex $6 \times -6 = -36$

U	V	X_i	X_{i-1}	operation
0000	0000	0	0	Shift
0000	0000	1	0	Sub
1101	0000	0	1	Add
0001	1000	1	0	Sub
1101	1100	-	-	-

중간 연산을 하고 shift하는 과정은 제외하고 최대한 단순하게 한 사이클이 끝난 직후의 결과 값을 표현해 보았다. 계산한 결과 8'b1101_1100으로 이를 2의 보수로 취한 결과 8'b0010_00100으로 계산 결과 36의 2의 보수임을 알 수 있었다. Radix-2의 경우 cycle 수가 곱하는 multiplicand의 bit수와 동일하다. 해당 예시에서도 4bit의 값들을 서로 곱했을

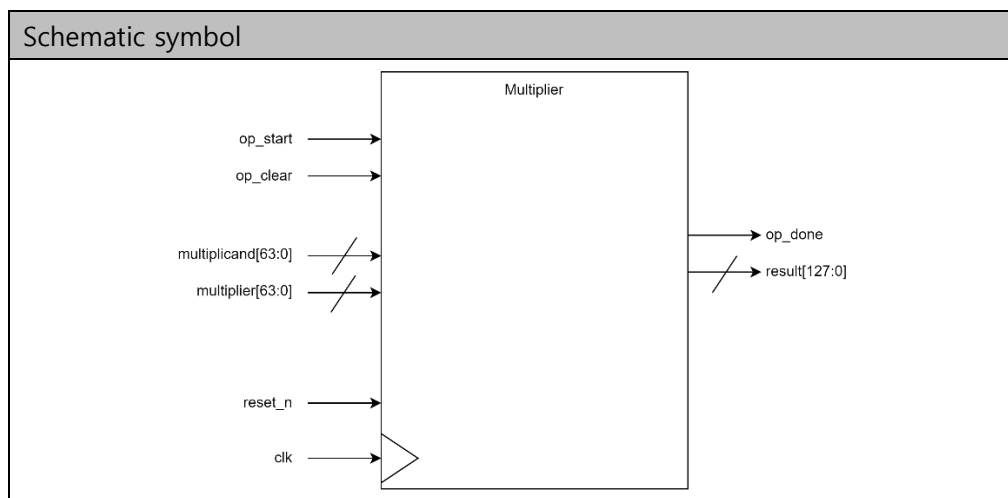
때, 4cycle만에 연산이 완료된 것을 확인할 수 있었다.

다음은 radix-4인 경우 booth multiplicand의 규칙이다.

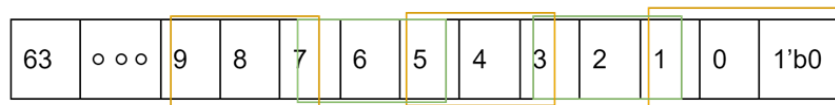
x_i	x_{i-1}	x_{i-2}	Operation	y_i	y_{i-1}	y
0	0	0	$0 + 0 = 0$	0	0	0
0	0	1	$0 + A = A$	0	1	+1
0	1	0	$2A - A = A$	0	1	+1
0	1	1	$2A + 0 = 2A$	1	0	+2
1	0	0	$-2A + 0 = -2A$	-1	0	-2
1	0	0	$-2A + A = -A$	0	-1	-1
1	1	0	$0 - A = -A$	0	-1	-1
1	1	1	$0 + 0 = 0$	0	0	0

3. 설계 세부사항

I/O table		
Direction	Port name	Description
Input	clk	Clock
	reset_n	Active low reset
	multiplier [63:0]	승수
	Multiplicand [63:0]	피승수
	op_start	Start operation
	op_clear	Clear operation
Output	op_done	Done operation
	result [127:0]	Multiplier result



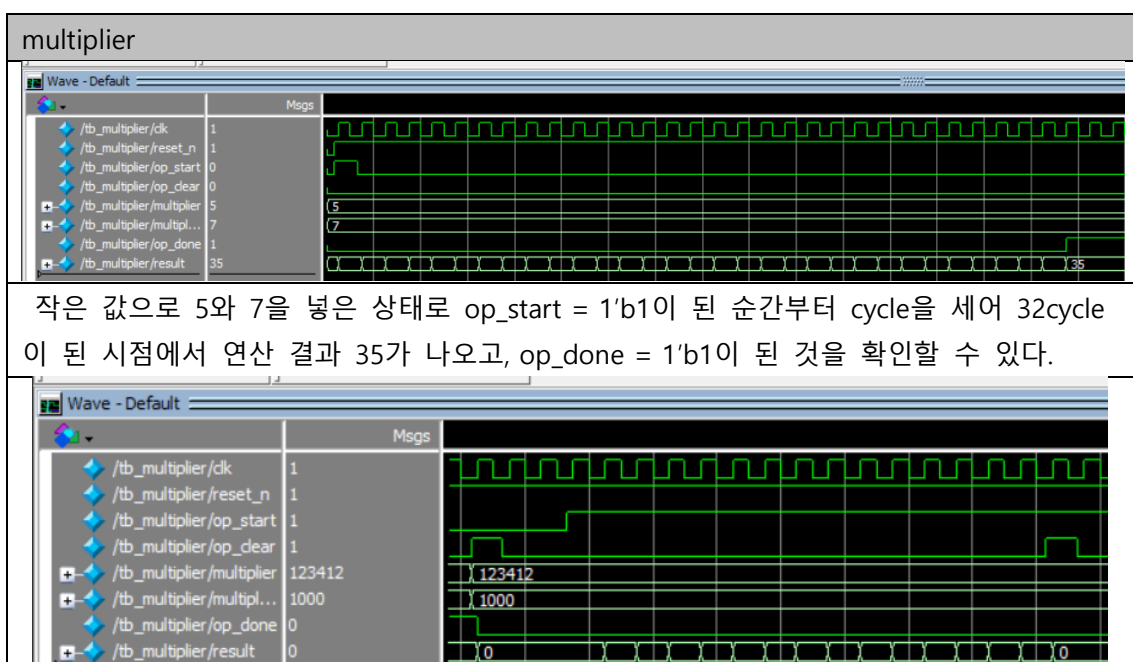
op_clear는 op_start 보다도 우선순위를 갖는다. 곱셈의 진행은 op_start = 1'b1인 경우 시작된다. 이후 op_start = 1'b0이 되어도 연산은 연산 종료까지 진행된다. 만약 연산 진행 도중에 op_clear = 1'b1이 된 경우에는 연산이 종료되고 result = 128'h00이 된다. 현재 구현한 multiplier는 radix-4다. Radix-2의 경우 64 clock cycle이 진행된 후에 연산 결과가 나오지만, radix-4의 경우 32 clock cycle이 진행된 후에, 연산 결과가 도출된다. 32cycle이 되는 순간 op_done = 1'b1의 값을 갖으며, 연산이 완료되었음을 알려준다. 연산이 완료된 후에도 특별한 signal이 없는 경우, 연산이 완료된 값을 그대로 유지한다. 다음은 next_x를 결정하는



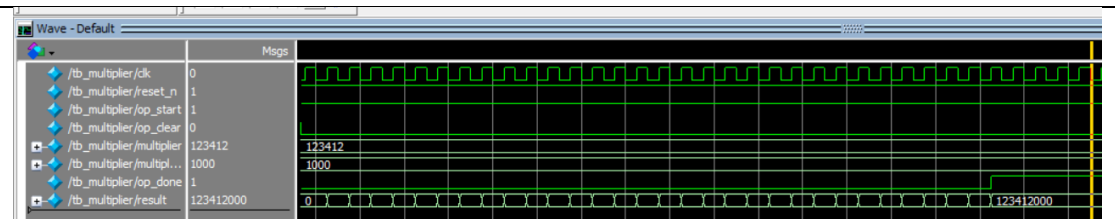
초기 곱셈은 multiplicand[2], multiplicand[1], 1'b0을 가지고 연산한다. 다음 연산할 것을 미리 저장하기 위해서는 현재 cycle수를 이용하여 다음 위치를 미리 지정하면서 연산한다. 다음 위치는 multiplicand[3:1]이 되어야 하므로, 현재 cycle인 0인 수에서 $0 + 3 \sim 0 + 1$ 로 이루어진다. 해당 규칙을 이용하여, cycle은 2의 배수로 증가하며, 다음 연산할, x들을 찾을 때, 그 cycle에서 3을 더한값과 1을 더한값의 범위를 갖게 된다. $next_x_i = cycle + 3$, $next_x_{i-1} = cycle + 2$, $next_x_{i-2} = cycle + 1$ 해당 규칙을 이용하여 곱셈이 가능해진다.

4. 설계 검증 및 실험 결과

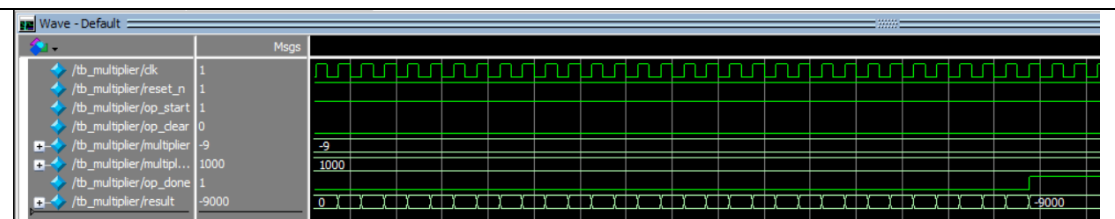
A. 시뮬레이션 결과



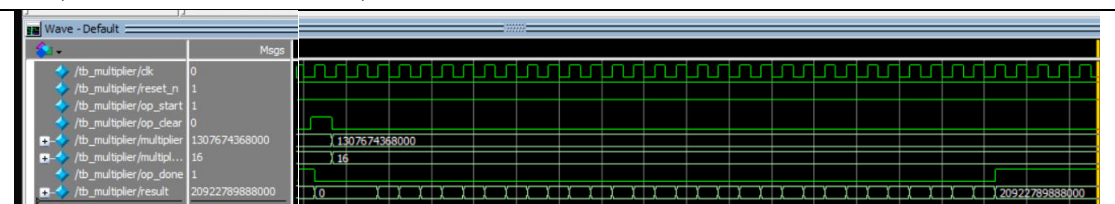
해당 결과는 op_clear = 1'b1이 된후에 result가 0의 값을 갖는지 확인한 것으로, op_clear = 1'b1이 될 때 연산이 진행되던 도중에도 result가 0의 값을 갖는 것을 확인할 수 있다.



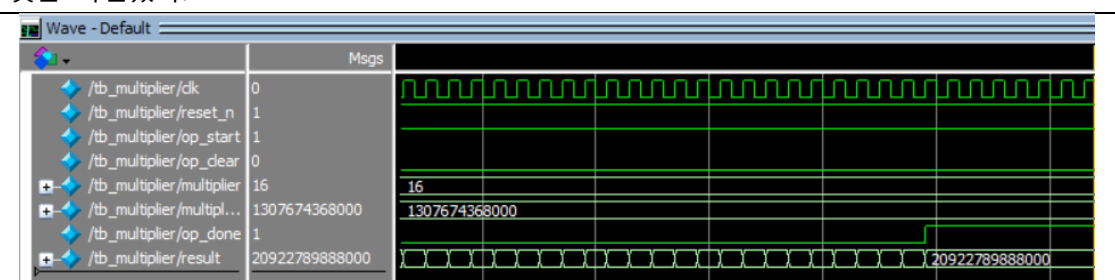
이 구간의 waveform은 op_clear = 1'b0이 되고, op_start = 1'b1이 될 때, 바로 연산이 시작되는지 확인한 것으로, 이 경우에도 32cycle이 된 시점에서 결과 123412000가 제대로 나온 값을 확인할 수 있다.



해당 waveform은 음수와 양수를 곱하면서 음수의 값이 제대로 나오는 가를 확인한 경우로, -9와 1000을 곱했을 때, -9000이란 값이 제대로 나왔음을 확인할 수 있다.



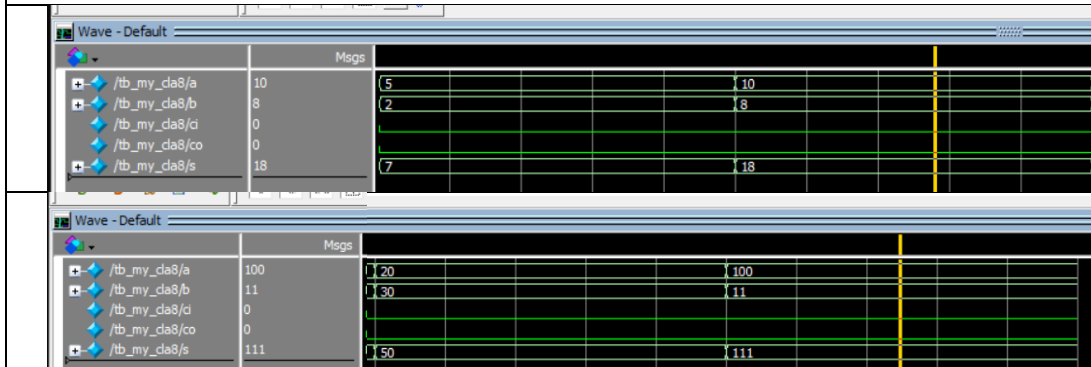
이번 waveform은 큰 수를 구했을 경우에 대해 관찰하고 싶었고, multiplier는 15! 의 수가 들어가 있으며 결과는 16!에 대한 수를 구하는 것을 보고 싶었고, 예상 결과가 나온 것을 확인했다.



해당 결과는 앞서 보였던 값이 multiplier와 multiplicand의 값이 반대로 넣어도 같은 결과가 나오는 지 확인한 것으로, 이 또한 앞에 보였던 값인 20,922,789,888,000의 수가 제대로 들어간 것을 확인할 수 있었다.

my_cla8

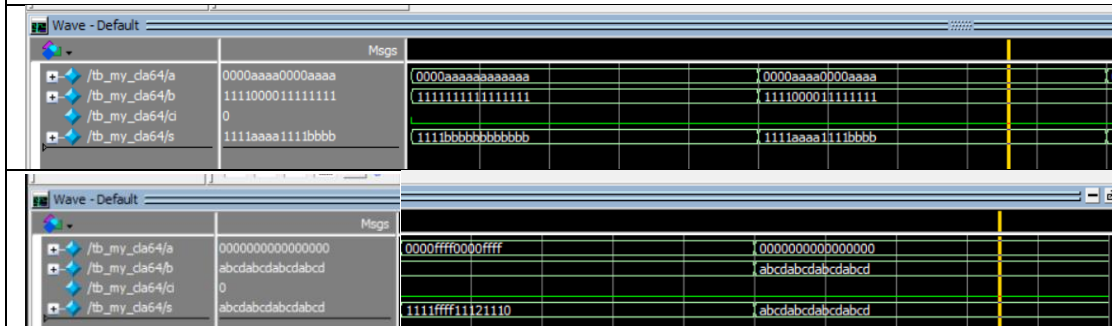
해당 모듈은 8bit를 가산할 수 있는 장치이다.



두 수의 덧셈이 잘 진행된 것을 확인할 수 있다.

my_cla64

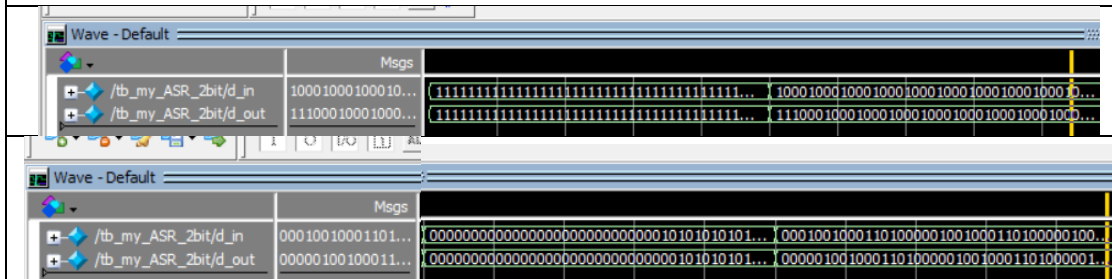
해당 모듈은 64bit를 가산할 수 있는 장치이다. 이번 cla의 경우 cout을 제외했다.



이번 모듈 또한 두 수의 덧셈이 잘 진행된 것을 확인할 수 있다.

my_ASR_2bit

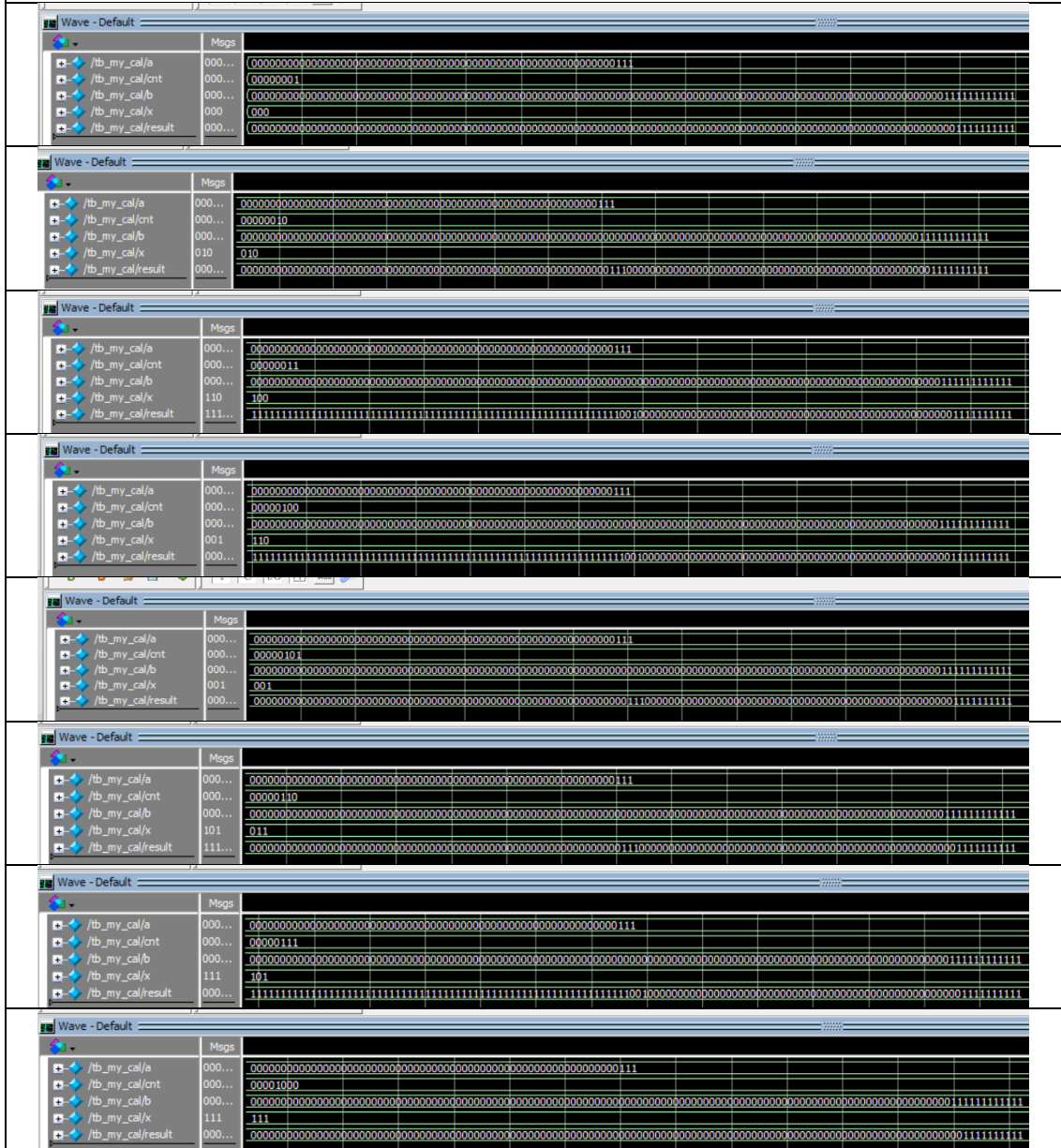
해당 모듈은 arithmetic shift right를 2bit 진행하게 된다.



테스트 벤치 값을 확인하면 부호 비트를 포함해 2bits shift되는 것을 확인할 수 있다.

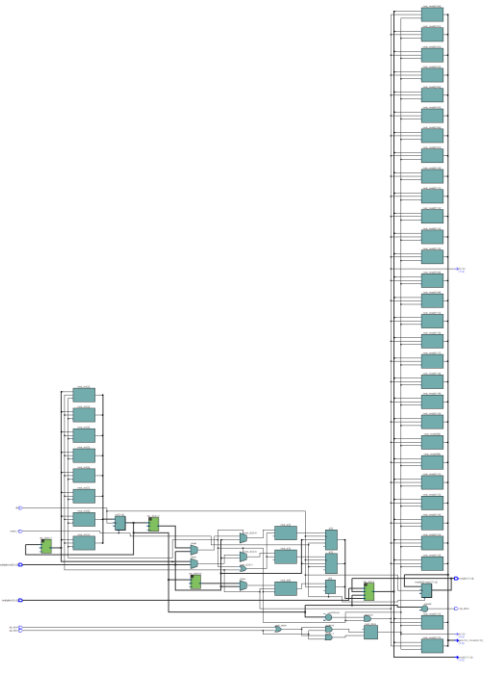
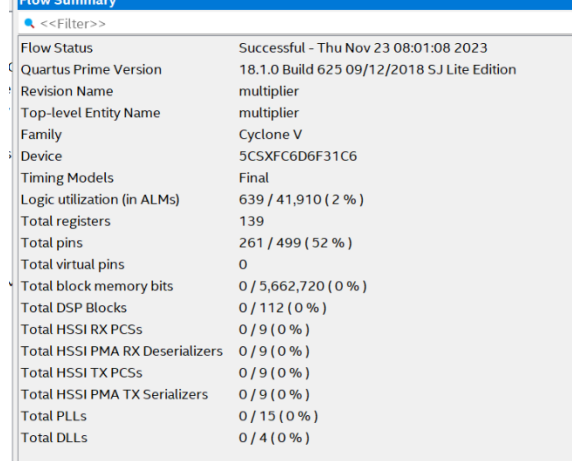
my_cal

해당 모듈은 64bits a, 128bits b, result, 3bits x를 이용하여 x의 값에 의해 radix-4일 때의 연산 결과가 제대로 나오는지 확인한다.



x의 값들에 의해 shift, +A, -A, +2A, -2A를 하는 것을 확인할 수 있다.

B. 합성(synthesis) 결과

	RTL map viewer																																										
 <table border="1"><thead><tr><th colspan="2">Flow Summary</th></tr><tr><th colspan="2"><<Filter>></th></tr></thead><tbody><tr><td>Flow Status</td><td>Successful - Thu Nov 23 08:01:08 2023</td></tr><tr><td>Quartus Prime Version</td><td>18.1.0 Build 625 09/12/2018 SJ Lite Edition</td></tr><tr><td>Revision Name</td><td>multiplier</td></tr><tr><td>Top-level Entity Name</td><td>multiplier</td></tr><tr><td>Family</td><td>Cyclone V</td></tr><tr><td>Device</td><td>5CSXFC6D6F31C6</td></tr><tr><td>Timing Models</td><td>Final</td></tr><tr><td>Logic utilization (in ALMs)</td><td>639 / 41,910 (2 %)</td></tr><tr><td>Total registers</td><td>139</td></tr><tr><td>Total pins</td><td>261 / 499 (52 %)</td></tr><tr><td>Total virtual pins</td><td>0</td></tr><tr><td>Total block memory bits</td><td>0 / 5,662,720 (0 %)</td></tr><tr><td>Total DSP Blocks</td><td>0 / 112 (0 %)</td></tr><tr><td>Total HSSI RX PCSs</td><td>0 / 9 (0 %)</td></tr><tr><td>Total HSSI PMA RX Deserializers</td><td>0 / 9 (0 %)</td></tr><tr><td>Total HSSI TX PCSs</td><td>0 / 9 (0 %)</td></tr><tr><td>Total HSSI PMA TX Serializers</td><td>0 / 9 (0 %)</td></tr><tr><td>Total PLLs</td><td>0 / 15 (0 %)</td></tr><tr><td>Total DLLs</td><td>0 / 4 (0 %)</td></tr></tbody></table>	Flow Summary		<<Filter>>		Flow Status	Successful - Thu Nov 23 08:01:08 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)	639 / 41,910 (2 %)	Total registers	139	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 %)	Flow summary를 보았을 때, Logic utilization은 639, total register는 139, pins는 261인 것을 확인할 수 있다.
Flow Summary																																											
<<Filter>>																																											
Flow Status	Successful - Thu Nov 23 08:01:08 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)	639 / 41,910 (2 %)																																										
Total registers	139																																										
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 %)																																										

5. 고찰 및 결론

A. 고찰

Submodule을 모두 완성하고 multiplier를 통해 재조립을 할 때, sequential part와 combinational part를 분리해서 만들었는데, combinational한 곳을 만드는 도중에 always 구문을 사용할 때 always @ (*) 이런 것을 사용해 보았다. Always에 넣을 인자를 무엇으로 해야 할지 고민하고 있을 때, 좀 더 간편하게 하고자 always에 대한 정보를 찾아보았다. 찾던 중 인자에 *을 넣는 구문을 보았고, 해당 구문에 대해 찾아본 결과 다음과 같은 설명이 있었다. "always 문 내의 모든 변수를 포함" 해당 문구를 확인하고, 설계한 결과

제대로 작동되는 것을 확인할 수 있었다. 원래 always 문에 넣을 이벤트 detect를 고민을 여러 번 했지만, 해당 문구 하나로 끝낼 수 있다는 것이 좋았다.

B. 결론

이번 실습에서는 radix-4를 이용하여 multiplier를 설계했다. Radix-4는 radix-2에 비해 cycle수를 반으로 줄일 수 있다. Radix-2는 한 칸 씩 이동하게 되지만 radix-4의 경우는 두 칸 씩 이동하면서 곱셈을 진행하기 때문이다. 이번 실습을 진행할 때 Radix-2를 먼저 구현하면서 규칙을 찾게 되었다. 그래서 radix-4를 도전해 보았고, 곱셈기가 제대로 작동하게 되면서, 해당 radix로 설계를 끝냈지만, 이를 통해 radix-16으로도 충분히 할 수 있겠다는 생각이 들었다. 상태가 2^5 로 32개의 케이스를 정의해야 하지만, 어떤 식으로 구현해야 될지 아는 상태에서 radix를 늘리는 건 시간이 충분하다면 설계하는 것도 나쁘지 않겠다는 생각이 든다.


6. 참고문헌

이준환 교수님/디지털논리회로2/광운대학교(컴퓨터정보공학부)/2023

이준환 교수님/컴퓨터공학기초실험2/광운대학교(컴퓨터정보공학부)/2023

김동규 교수님/보안 회로 설계/한양대학교

(더 자세한 정보를 확인할 수 없어, 자료 링크를 달겠습니다. 해당 자료는 구글에서 "verilog always문 *"으로 검색 후 찾았습니다.)

 Hanyang University
http://esslab.hanyang.ac.kr > lecture_note > [Se...
[Secu-2] Verilog 기초문법.pdf
• Verilog 순차 처리문. Page 4. 4. Verilog HDL(1/2). • Verilog HDL 특성. - 국제 ... - always 문
내의 신호는 reg혹은 integer로 선언. - @(*) : always 문 내의 모든 ...
페이지 39개

링크:

http://esslab.hanyang.ac.kr/uploads/security_circuit_design/lecture_note/8046f8d687beab9b5bc99594ff3be505/%5BSecu-2%5D%20Verilog%20%EA%B8%B0%EC%B4%88%EB%AC%B8%EB%B2%95.pdf