컴퓨터 공학 기초 실험2 보고서

실험제목: Multiplier

실험일자: 2022년 11월 09일 (수)

제출일자: 2022년 11월 22일 (화)

학 과:컴퓨터정보공학부

담당교수: 공진흥 교수님

실습분반: 수요일 0, 1, 2

학 번: 2019202021

성 명: 정 성 엽

1. 제목 및 목적

A. 제목

Multiplier

B. 목적

Multiplicand와 Multiplier를 곱하여서 그 결과를 출력하는 hardware를 설계하고 이때 Multiplier는 Booth Multiplication을 이용한다. 이 때 각각의 bit length는 64bits이고 결과는 128bits이다.

2. 원리(배경지식)

A. Booth Multiplication

i. Booth Multiplication은 승수의 bit x_i, x_{i-1}의 패턴을 읽고 연산 수행을 반복하여 곱셈을 수행한다. 이때 승수의 패턴에 따른 연산은 아래의 표와 같다. 이 표는 Radix-2를 나타낸 것이다.

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

x_i와 x_{i-1}이 연속 된다면 shift만 진행하며 만약 순서가 달라진다면 01인 경우 add 연산 10인 경우 sub 연산을 진행한다.

ii. 위 표와 다르게 패턴을 더 추가하여 Radix-4 또한 만들 수 있다. 이 패턴을 추가할 수록 더욱 적은 횟수로 연산이 가능하지만 그만큼 더 많은 공간을 차지한다. 아래의 표는 Radix-4의 표이다.

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

이번 실험에서는 Radix-2 booth multiplication만을 설계하였으며 아래 예시로. 7 * -7을 진행하였다.

A	0111(7)	7*-7
X	1001(-7)	
Sub A	1001	Sub and shift (10)
Shift	11001	
Add A	0011	Add and shift (01)
Shift	000111	
Shift	0000111	Shift (00)
Sub A	1001	Sub and shift (10)
Shift	11001111	

결과는 1100111(-49)가 나온다.

B. Multiplier

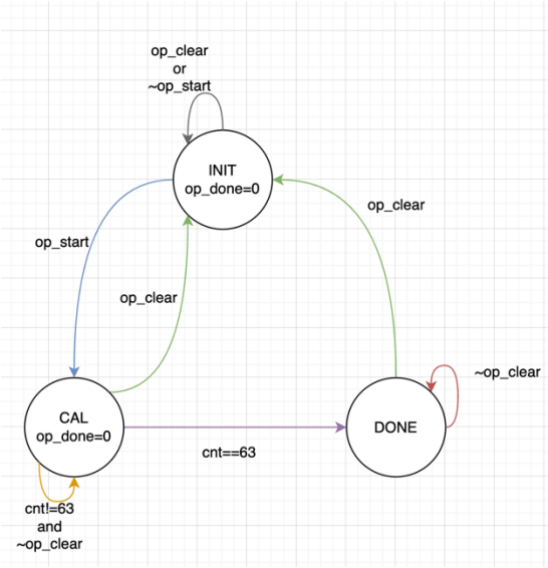
i. 승수와 피승수를 입력 받아 곱하여 결과값을 도출하는 hardware로 만약 64bit길이의 승수와 피승수라면 128bit의 result가 출력된다. 이는 각 자릿수마다 덧셈 또는 뺄셈 또는 생략 진행 후 shift 연산이 한번 씩 진행되기 때문에 64번의 shift가 진행되면 128bit의 결과가 도출되는 것이다.

3. 설계 세부사항

Multiplier

A. State diagram

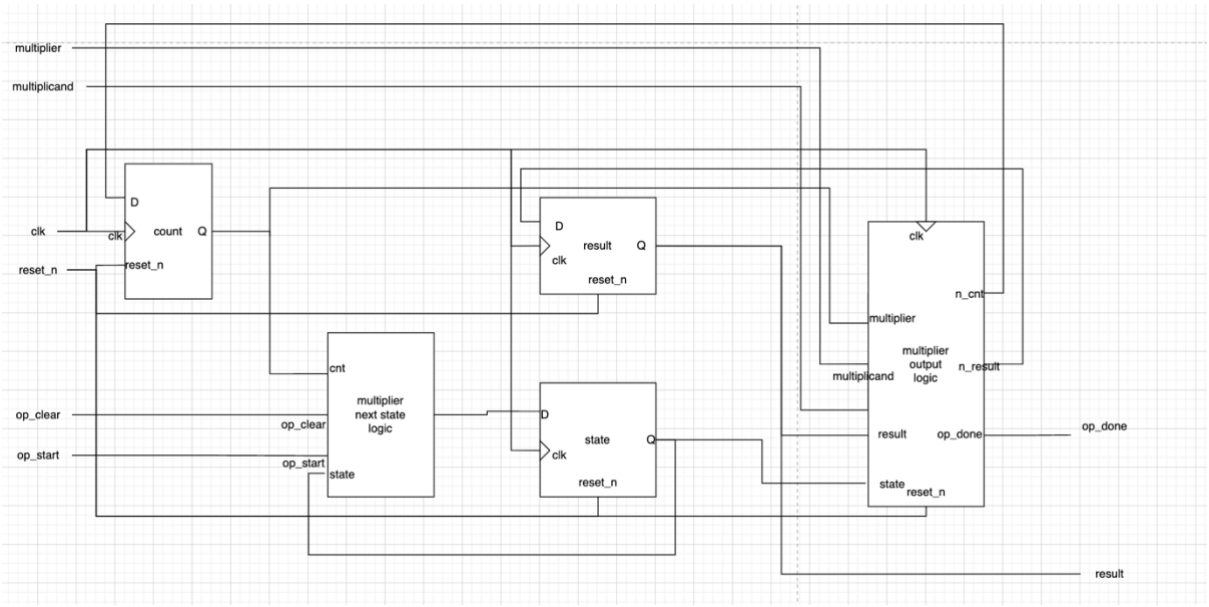
Binary encoding	
INIT	00
CAL	01
DONE	11



color	
Gray	op_clear or (op_clear and ~op_start)
Blue	~op_clear and op_start
Green	op_clear
Yellow	cnt!=63 and ~op_clear
Purple	cnt==63 and ~op_clear
red	~op_clear

Multiplier를 구현하기 위해 위와 같은 3단계의 FSM을 설계하였다. Op_start는 연산의 시작을, op_clear는 연산 초기화로 새로 연산을 시작할 때, op_done은 연산의 완료를 뜻한다. 이 때 op_clear는 연산을 진행하건 안하든 모든 값을 초기화 하는 것으로 한다.

위의 state diagram처럼 INIT state에서는 시작점을 뜻하고 만약 op_start가 1로 set이 된다면 Binary Multiplication 연산을 하는 CAL state로 이동하고 이 state에서는 cnt가 0부터 시작해 63이 될 때까지 연산을 한다. 63번 연산이 끝나면 DONE state로 이동하는데 이는 연산이 종료됨을 뜻하고 op_done을 1로 set한다. Op_clear가 1로 set 된다면 모든 값을 초기화한다.



위에서 설계한 FSM에 따라 op_clear 그리고 op_start의 입력을 받아 multiplier next state에서 계산하고 state를 저장할 state 2bit d-flipflop을 두었다.

CAL state에서는 연산 횟수를 계산해야 하므로 63번 카운트 하기위해 7bit의 D-flipflop을 통해 저장하였다.

Multiplier output logic에서는 CAL state에서 multiplication 연산을 수행하고, INIT이나 DONE에서는 op_done을 출력한다. 그리고 result를 저장하기 위한 128bit D-flipflop이 필요하다.

B. Module Configuration

	Name	Description
Top module	Multiplier	Multiplier top module
Sub module	Cla64	64bit carry look adder
	Multiplier_ns	Multiplier next state logic
	Multiplier_out	Output logic

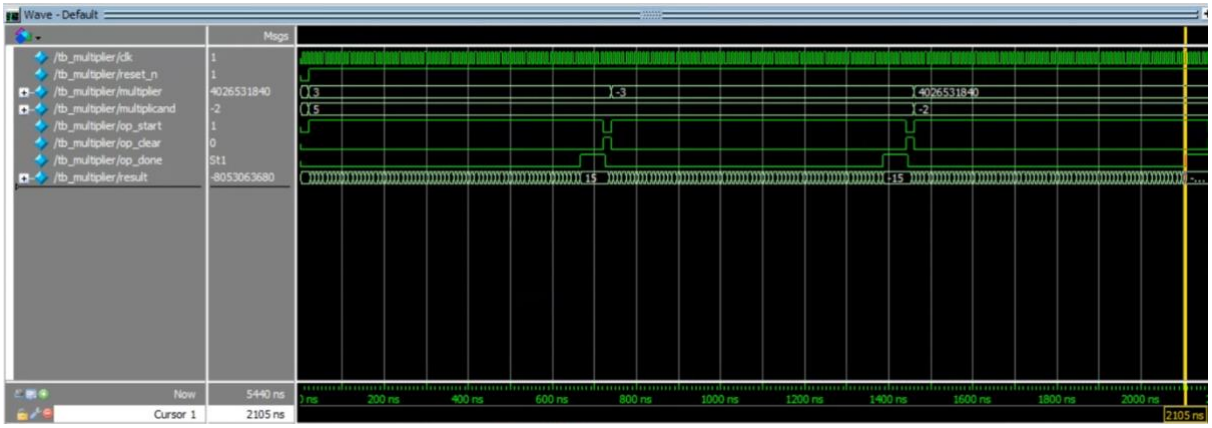
C. I/O configuration

Module	i/o	Signal	n-bit	Description
Multiplier	Input	Clk	1bit	Clock
		Reset_n	1bit	Active low하게 작동하는 reset
		Multiplier	64bit	승수
		Multiplicand	64bit	피승수
		Op_start	1bit	Start operation
		Op_clear	1bit	Clear operation
	Output	Op_done	1bit	Done operation
		result	128bit	Multiplier result

4. 설계 검증 및 실험 결과

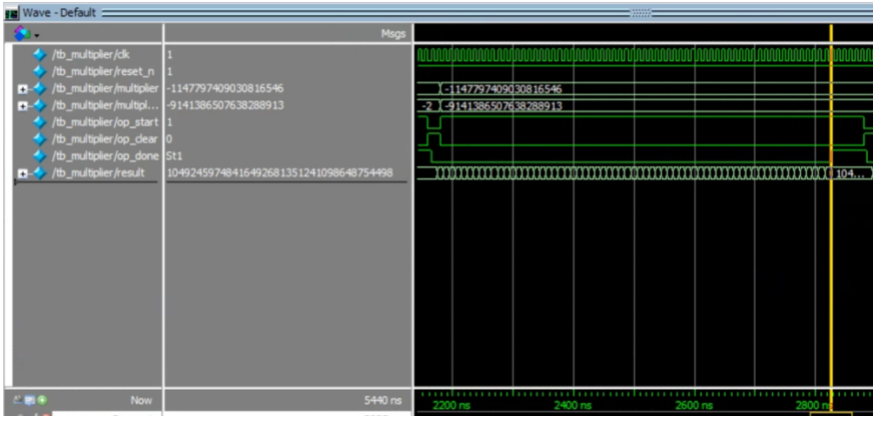
A. 시뮬레이션 결과

1) 양수*양수 , 음수*양수, 양수*음수



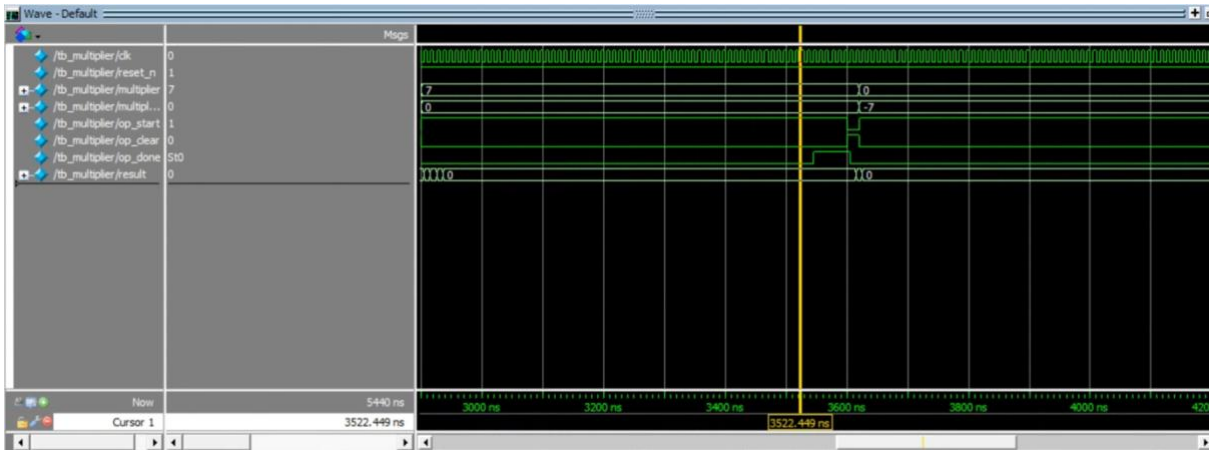
과제의 예시였던 5*3와 5*-3 그리고 임의로 지정한 큰 수와 작은 수를 지정하여 연산을 진행하였고 5*3은 15 5*-3은 -15 4026541840*-2는 -8053063680으로 정상적 연산이 수행 되었음을 볼 수 있다.

2) 음수*음수



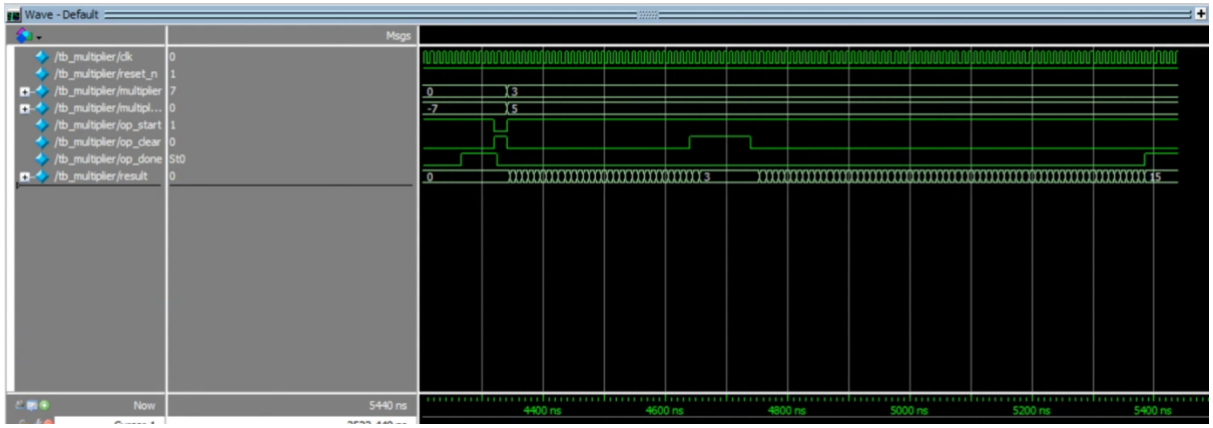
매우 큰 수에 대한 연산을 음수와 음수의 곱으로 확인하였다. -11477974090308165 46 * -914138650763828891의 곱을 진행하였고 결과값으로 10492459748416482681351241098648754498 값이 나왔다.

3) 0의 곱셈



0을 곱하는 경우 다른 연산에 비해 일찍 곱셈 연산이 끝나는 것을 확인하였다. 값 또한 0으로 정상 출력되었다.

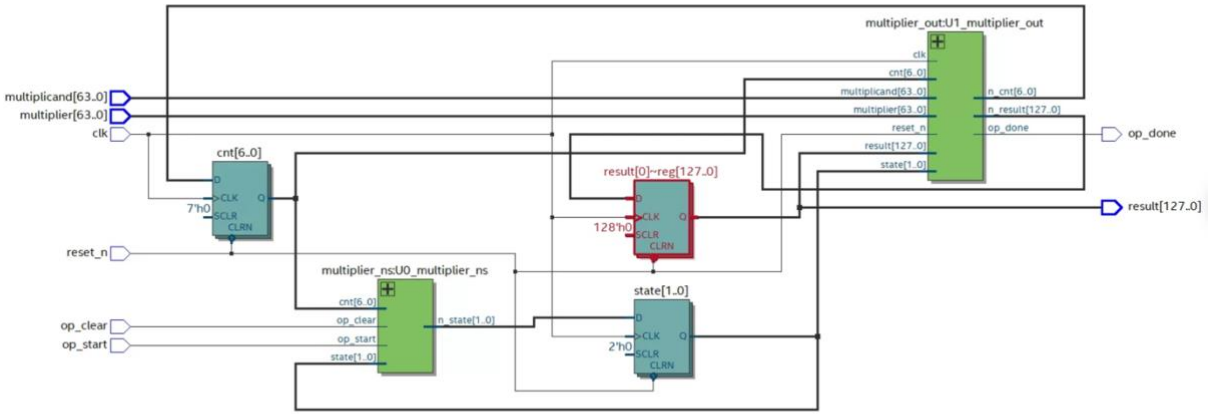
4) 연산 중 op_clear이 1로 set



연산 중 중간에 op_clear를 1로 set 하였고 INIT으로 이동하면서 값이 3으로 다시 초기화 되었다. 다시 op_clear를 0으로 reset하고 그 후 연산이 끝났을 때 정상적으로 15의 값이 저장되어 출력되었다.

B. 합성(synthesis) 결과

1) RTL viewer



2) Flow summary

Flow Summary

<<Filter>>

Flow Status	Successful - Tue Nov 22 21:07:20 2022
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)	324 / 41,910 (< 1 %)
Total registers	138
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 %)

위에서 설계한 Block diagram과 같은 구조를 하고 있으며 설계한 것처럼 count를 하는 모듈, result를 저장하는 모듈, state를 저장하는 모듈이 모두 있고 multiplier_ns를 통해 다

음 state를 지정하고 multiplier_out을 통해 n_result와 op_done을 지정한다.

5. 고찰 및 결론

A. 고찰

이번 과제에서 스스로에게 매우 아쉬웠던 점은 시간적 issue로 인해 radix-4를 완성시키지 못했다는 것이다. 시간이 된다면 다음 프로젝트를 위해 radix-4를 완성해볼 생각이다.

Booth multiplication을 이용하여 radix-2의 곱셈기를 구현하였고 top module만이 주어졌을 때 FSM을 설계하여 next state logic과 output logic 모듈을 구현하여 제작하였다. 이때 op_clear와 op_start 그리고 cnt 값에 따라 FSM을 설계하였고 이것이 정상적으로 작동하였다.

또한 result, state, count 모듈을 d-flipflop으로 구현하였고 각 값을 저장할 수 있도록 하였다. 이때 result는 64bit의 multiplicand와 multiplier가 곱해지면 최대 128bit의 값이 저장될 수 있으므로 128bit로 지정하고 State에서는 INIT, CAL, DONE으로 3가지 state를 가지고 있기 때문에 2bit로, count는 0부터 63까지 계산해야하고 부호를 구분하기 위해 7bit로 구현하였다.

Output logic에서 x_i와 x_{i-1}의 패턴을 읽어서 비교해야 하는데 임시로 저장하도록 d-flipflop을 두었고, 연산이 끝난 후에는 d-flipflop에 다음 승수 패턴을 넣어서 다음 연산을 하면 그 다음 패턴 1개만 읽어서 가져오면 되므로 쉽게 비교가 가능했다.

B. 결론

Booth multiplication에서 radix-2는 패턴이 비교적 간단하지만 연산이 느리고 radix-4는 패턴이 더 복잡해지지만 연산이 더욱 빨라진다. 이는 승수의 비트 패턴을 늘리면 늘릴수록 계산의 cycle이 줄어들기 때문이다. 다만 그 패턴을 비교하고 각 패턴에 대한 logic을 더 많이 설계해야 하므로 logic 사이즈가 더 커진다.

설계한 radix-2의 booth multiplication은 add연산과 sub연산 그리고 shift연산을 통해 기존 곱셈보다 빠르게 계산이 가능하다. 이유는 shift 연산에서 나오는데 기존의 곱셈은 add연산을 반복해서 진행해서 큰 값의 곱셈의 경우 속도가 더욱 느려지지만 booth multiplication은 64bit의 multiplier와 multiplicand인 경우 64번만 반복하면 되기 때문에 훨씬 빠르다.

6. 참고문헌

1) 공영호 교수님/디지털논리회로2/광운대학교 컴퓨터정보공학부/2022

2) 공진흥 교수님/컴퓨터공학기초실험2/광운대학교 컴퓨터정보공학부/2022