컴퓨터 공학 기초 실험2 보고서

실험제목: Multiplier

실험일자: 2020년 10월 30일 (금)

제출일자: 2020년 11월 13일 (금)

학 과: 컴퓨터공학과

담당교수: 이준환 교수님

실습분반: 금요일 5,6,7

학 번: 2019202009

성 명: 서여지

1. 제목 및 목적

A. 제목

Multiplier

B. 목적

Booth 알고리즘을 이용하여 64비트 곱셈을 계산하는 Multiplier를 구현한다. Booth 알고리즘을 이해하고 직접 설계할 수 있도록 한다. Booth 알고리즘의 radix-2와 radix-4의 차이를 설명할 수 있다. 구현한 Multiplier를 이용해 얻은 계산 결과를 분석할 수 있다.

2. 원리(배경지식)

1) Booth 알고리즘

Booth 알고리즘은 multiplicand와 multiplier를 곱셈 결과를 multiplicand의 연속성을 확인하여 구하는 알고리즘이다. 한 번에 읽는 multiplicand의 bit수에 따라 radix-2와 radix-4 등으로 표현할 수 있다. 이번 실험에서 구현한 multiplier는 한 번의 clock cycle에 multiplicand를 2 bit 읽어 계산하는 radix-2 booth algorithm을 이용하였다. 계산 과정은 다음과 같다.

1. multiplicand의 마지막부터 시작하여 계산할 자리수의 multiplicand값을 읽는다.
2. 1.에서 읽은 multiplicand의 바로 뒤에 위치한 1 bit를 읽는다. 이때 1. 에서 읽은 값이 multiplicand의 마지막 bit인 경우 2.에서 읽는 값은 0으로 지정한다.
3. 1.과 2.에서 읽은 값에 따라 result의 상위 bit에 multiplier를 더하거나 빼거나 연산하지 않는다. (1.의 결과, 2.의 결과)의 순서쌍이 (1,1)이나 (0,0)인 경우 result에 multiplier를 더하지 않는다. 반면 (1,0)인 경우 result의 상위 bit에 multiplier를 빼고, (0,1)인 경우 더한다.
4. result를 1bit만큼 ASR 연산 한다. result의 값을 1bit씩 오른쪽으로 이동시키고, 기존 sign bit의 값을 result의 최상위 bit에 저장한다.
5. 다시 1.로 돌아가 이번에 읽은 multiplicand bit의 바로 앞 bit를 읽는다. multiplicand를 모두 읽어 계산한 뒤 반복을 종료한다.

위의 과정은 2bit를 읽어 계산하는 radix-2의 경우이다. 3개의 bit를 읽어 계산하는 radix-4는 읽은 3개의 bit를 조합하여 나오는 8가지 경우에 대해 계산을 진행한다는 차이가 있다. 또한 radix-4를 이용한 과정에서는 한 번에 2bit의 multiplicand를 계산하므로 1 clock cycle이 지난 뒤, 이용하는 다음 multiplicand bit는 이전에 읽은 bit에서 2bit 떨어져있는 bit이다.

(2) radix-2와 radix-4

radix-2와 radix-4는 곱셈 과정에서 읽는 multiplicand의 bit수에 차이가 있다. radix-2는 2bit를 읽고, radix-4는 3bit를 읽어 사용한다. radix-2에서 이용하는 계산은 다음과 같이 정리할 수 있다.

multiplicand[x]	multiplicand[x-1]	operation	result에 더하는 값
0	0	LSR	0
0	1	ADD, LSR	multiplier
1	0	SUB, LSR	-multiplier
1	1	LSR	0

radix-4의 경우 이용하는 3bit를 앞의 2bit와 뒤의 2bit로 나누어서 radix-2의 계산 방법을 적용한다. 앞의 2bit는 계산결과에 지수인 2를 곱한 결과를 이용한다. 이것을 정리하면 다음과 같이 나타난다.

multiplicand [x]	multiplicand [x-1]	multiplicand [x-2]	operation 1	operation 2	result에 더하는 값
0	0	0	LSR	LSR	0
0	0	1	LSR	ADD, LSR	multiplier
0	1	0	ADD, LSR	SUB, LSR	multiplier
0	1	1	ADD, LSR	LSR	2 multiplier
1	0	0	SUB, LSR	LSR	-2 multiplier
1	0	1	SUB, LSR	ADD, LSR	- multiplier
1	1	0	LSR	SUB, LSR	- multiplier
1	1	1	LSR	LSR	0

3. 설계 세부사항

(1) 입출력

사용된 입출력은 다음과 같다.

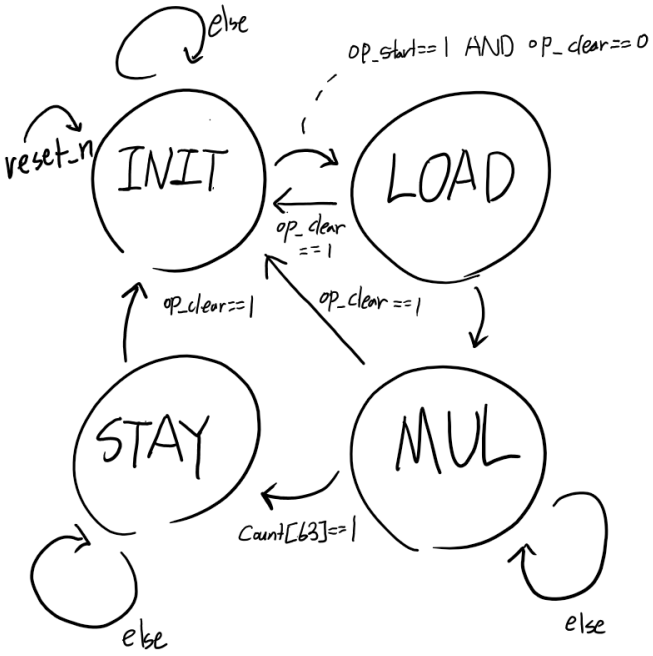
input/output	name	크기(bit)	설명
input	clk	1	clock
	reset_n	1	reset(active low)
	op_start	1	연산 시작
	op_clear	1	초기화
	multiplier	64	승수
	multiplicand	64	피승수
output	op_done	1	연산 종료
	result	128	연산결과

(2) 상태정의, 인코딩

설계에 사용한 state는 다음과 같다.

state	encoding	설명
INIT	00	op_start 신호를 대기하는 초기 상태
LOAD	01	multiplier와 multiplicand의 값을 register에 저장
MUL	10	곱셈 결과를 구하는 과정
STAY	11	계산 완료 후 op_clear 신호를 대기하는 상태

state transition diagram



state transition table

current state		op_start	op_clear	count[63]	next state	
INIT	00	0	X	X	INIT	00
		1	0	X	LOAD	01
LOAD	01	X	1	X	INIT	00
		X	0	0	MUL	10
MUL	10	X	1	X	INIT	00
		X	0	0	MUL	10
		X	0	1	STAY	11
STAY	11	X	1	X	INIT	00
		X	0	X	STAY	11

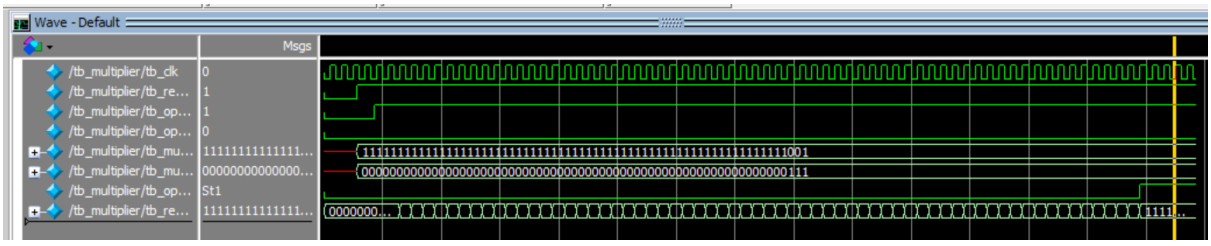
(3) MUL 연산과정

calculate module에서 진행되는 곱셈연산은 크게 3개의 부분으로 나뉘어 실행된다. 각각 현재 multiplicand의 값을 1 bit LSR하여 next multiplicand를 구하는 부분, 현재의 counter를 1 bit LSL하여 next_counter를 구하는 부분, 현재 multiplicand에 대한 연산을 진행하여 result를 구하는 부분이다. result를 구하는 부분은 다시 두 과정으로 나뉘어 result의 상위 64 bit를 구하는 과정과 128 bit전체 정보를 1 bit ASR하는 과정으로 구성된다. counter는 one hot encoding을 이용하여 계산이 64번 실행되도록 하고, 순환하는 Logic Shift Left를 이용하여 register에 저장할 next_counter를 계산한다. result의 상위 64 bit는 CLA를 사용하여 구한 기존 result의 상위 64 bit와 multiplier의 덧셈과 뺄셈 결과를 이용한다.

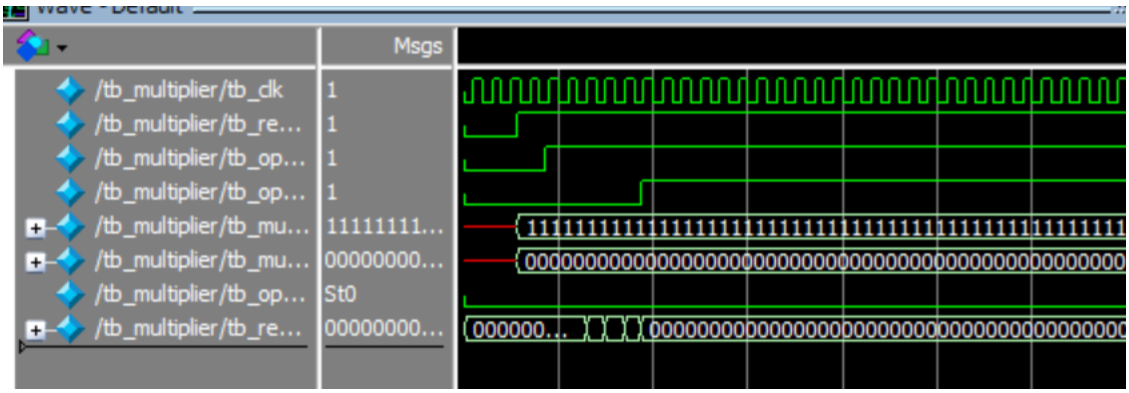
4. 설계 검증 및 실험 결과

A. 시뮬레이션 결과

testbench는 강의자료의 예시인 -7 * 7에 대해 작성하였다. op_start신호 이후 곱셈계산이 시작되어 op_done이 1이 되었을 때 곱셈 결과를 정상적으로 출력하는 것이 확인되었다.

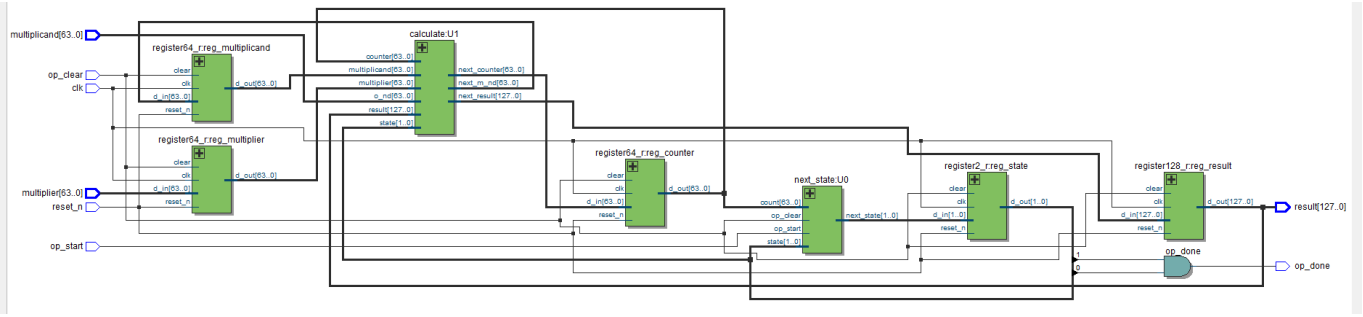


op_clear가 입력될 경우 register에 저장된 값이 0으로 초기화된다.



B. 합성(synthesis) 결과

RTL Viewer는 다음과 같이 나타났다. multiplier, multiplicand, state, result, counter를 저장하는 register가 모듈 내부에 존재하고, 다음 state를 계산하는 next_state 모듈과 곱셈계산과 counter를 증가시키는 calculate module이 위치한다.



Flow Summary는 다음과 같이 나타났다. multiplier(64), multiplicand(64), op_start(1), op_clear(1), clk(1), reset_n(1) 이 input pin을 이용하여 총 132개의 pin을 차지하고, result(128), op_done(1)이 129개의 pin을 이용하여 모두 261개의 pin이 사용되었다. register는 result(128), multiplier(64), multiplicand(64), counter(64), state(2)가 이용되었다.

Compilation Report - multiplier

Table of Contents

Flow Summary

Flow Status	Successful - Fri Nov 13 00:07:29 2020
Quartus Prime Version	15.1.0 Build 185 10/21/2015 SJ Lite Edition
Revision Name	multiplier
Top-level Entity Name	multiplier
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	224 / 41,910 (< 1 %)
Total registers	369
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

처음 state를 정의했을 때는 LOAD없이 INIT state에서 바로 multiplier와 multiplicand를 register에 저장하는 방법을 이용했다가, register의 기존 출력인 64'b0 이 지정한 값과 교대로 출력되는 오류가 발생하여 LOAD state를 추가하였다. 그러나 오류의 실체는 sensitivity list의 오류로 인해 register에 정상적인 입력이 2주기에 한 번 이루어진 것이 원인이었다. 원인을 뒤늦게 발견하여 LOAD state를 완전히 삭제하지 못하여 계산에 필요한 clock cycle의 수가 늘어나게 되었다. 모듈을 상위 모듈에 instance하기 전, 각 모듈을 따로 검증하는 절차의 필요성을 느끼게 되었다.

B. 결론

고찰 및 결론에 자기가 구현한 곱셈기가 어떤 곱셈기인지 언급하고 구현한 곱셈기

의 특징 또는 장점에 대해서 반드시 작성한다.

이번 과제는 booth algorithm을 이용한 multiplier를 구현하는 것이었다. 내가 구현한 multiplier는 radix-2를 이용하여 한 번에 2개의 multiplicand의 bit를 읽어 동작한다. radix-2를 이용하면 multiplicand의 각 bit에 대해 1 clock cycle이 필요하므로 모두 multiplicand의 길이만큼의 clock cycle이후에 계산이 종료된다. radix-2의 경우 구현이 상대적으로 간단하다는 장점이 있다. 반면 radix-4를 이용하는 경우에는, 한 번에 2 bit에 대한 곱셈연산을 할 수 있으므로 radix-2가 요구하는 시간의 절반으로 계산을 마칠 수 있다는 장점이 있다.

6. 참고문헌

이준환교수님, 디지털논리회로2 강의자료, 광운대학교 컴퓨터정보공학과,2020

이준환교수님, 컴퓨터공학기초실험2 강의자료, 광운대학교 컴퓨터정보공학과,2020