컴퓨터 공학 기초 실험2 보고서

실험제목: Multiplier

실험일자: 2022년 11월 8일 (화)

제출일자: 2022년 11월 21일 (월)

학 과: 컴퓨터정보공학부

담당교수: 공영호 교수님

실습분반: 화요일 0,1,2

학 번: 2019202050

성 명: 이강현

1. 제목 및 목적
   A. 제목

      Multiplier

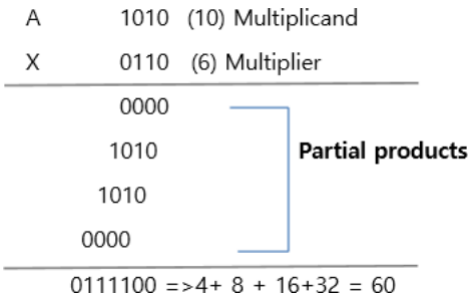
   B. 목적

      verilog에서 곱셈을 연산하는 방법을 알고 binary Multiplication과 booth Multiplication의 차이를 안다. Booth Multiplication방법을 사용한 곱셈연산을 verilog에서 구현해본다.

2. 원리(배경지식)

   <binary multiplication>

   Multiplicand에 multiplier를 1비트씩 나누어 곱해주며 자릿수를 맞춰주는 연산 방식이다.



   위와 같은 예시를 통해 설명하면 1010에 0110의 맨 오른쪽 비트 0을 곱해준 0000을 적고 또 0110의 오른쪽에서 두번째 비트 1을 1010에 곱해주어 자릿수를 맞춰 적고 이를 반복하고 모두 더해주는 방식이다. 양수간의 곱셈연산은 그대로 위와같이 진행되지만 음수를 곱하는 경우는 multiplicand의 보수를 취한 값을 추가적으로 더해주어 결과를 얻는다. 단순하고 직관적으로 계산할 수 있다는 장점을 가지나 곱해주는 값에 0이 많다면 결국 0을 더해주는 무의미한 연산을 하게 된다는 단점이 존재한다.

   <booth multiplication>

   Booth multiplication은 승수의 LSB에 0을 붙여주어 2bit씩 묶어 00,01,10,11일 때 arithmetic logical right shift연산과 add,sub을 이용하여 연산하는 방식이다.

   U를 00,01,10,11에 의한 연산결과를 저장하는 비트라고 하고 V를 overflow값이라 값 할 때 {U,V}한 값이 곱셈연산의 결과 값이 된다.

   이때 V는 01,10일때는 add,sub의 연산을 마친 결과의 LSB가 V의 MSB이고, 11,00

일때는 shift만 한 결과의 LSB가 MSB가 된다. 아래는 booth multiplication의 예시이다.

x_i	x_{i-1}	Operation	Comments	y_i
0	0	shift only	string of zeros	0
1	1	shift only	string of ones	0
1	0	subtract and shift	beginning of a string of ones	$\bar{1}$
0	1	add and shift	end of a string of ones	1

A		1 0 1 1	
X	x	0 0 1 1	
Y		0 1 0 $\bar{1}$	
$Y_0 = \bar{1}$ Add –A		0 1 0 1	
Shift		0 0 1 0	1
$Y_1 = 0$ Shift		0 0 0 1	0 1
$Y_2 = 1$ Add		1 0 1 1	
		1 1 0 0	0 1
Shift		1 1 1 0	0 0 1

Y는 X의 맨 뒤에 0을 붙였을 때 위의 표에 따라 구할 수 있다.

3. 설계 세부사항

Multiplier를 radix-4로 구현했기 때문에 연산 횟수는 32clock이다.

Input은 clk,reset_n,multiplicand,multiplier,op_start,op_clear이고 output은 op_done, result로 실습과제에 올라온 표대로 구성하였다. 연산과정중 필요한 reg는 temp1, temp2은 각각 multiplier와 multiplicand를 저장하는 변수이고 temp3에는 값이 연산되는 저장소이다. Temp3은 두배로 곱셈되는 경우가 존재하기에 65비트로 구성하였고 이는 sum,sum2,sub,sub2변수 또한 마찬가지이다. 또한 temp1[1]은 xi ,temp1[0] $xi-1$, lastbit은 $xi-2$를 나타내도록 구성하였다. count변수를 통해 연산이 얼마나 진행되어야 하는지에 대한 값을 넣어주었다. 또한 add와 sub 그리고 add2와 sub2를 사용했는데 이곳에는 temp[1:0]과 lastbit의 값에 따라 연산해야하는 방법이 다르므로 현재상태에서 한번 연산했을 때 값들을 각각 저장해두고 이

를 if문에서 사용하고 전체비트를 shift시키는 방법으로 구현하였다.

Op_start가 1일 때 데이터를 temp 변수들에 할당하고 op_start와 op_clear가 모두 0일 때 연산이 진행되게끔 프로그램을 설계하였다.

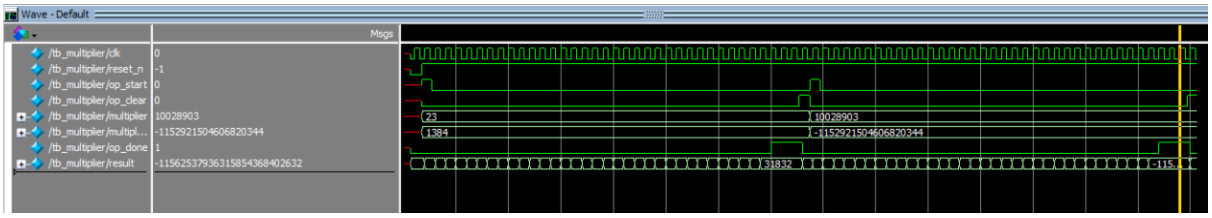
연산이 진행되면 count가 1감소하고 count가 0일 때 연산을 마무리하고 op_done의 값을 1로 set하고 op_clear가 1이 되어 값을 초기화하기 전에는 값들을 계속 유지하게끔 구현하였다.

연산의 마지막은 result가 128비트이므로 {temp3[63:0],temp1}처럼 두변수를 결합하여 값을 할당하였고 이는 temp3에서 값을 처리한 후 shift가 모든 곱셈연산의 마지막 연산이므로 결론적으로 부호비트의 복사이다. 따라서 앞의 비트를 잘라 128비트를 맞추어주었다.

4. 설계 검증 및 실험 결과

A. 시뮬레이션 결과

<multiplier>(top module)



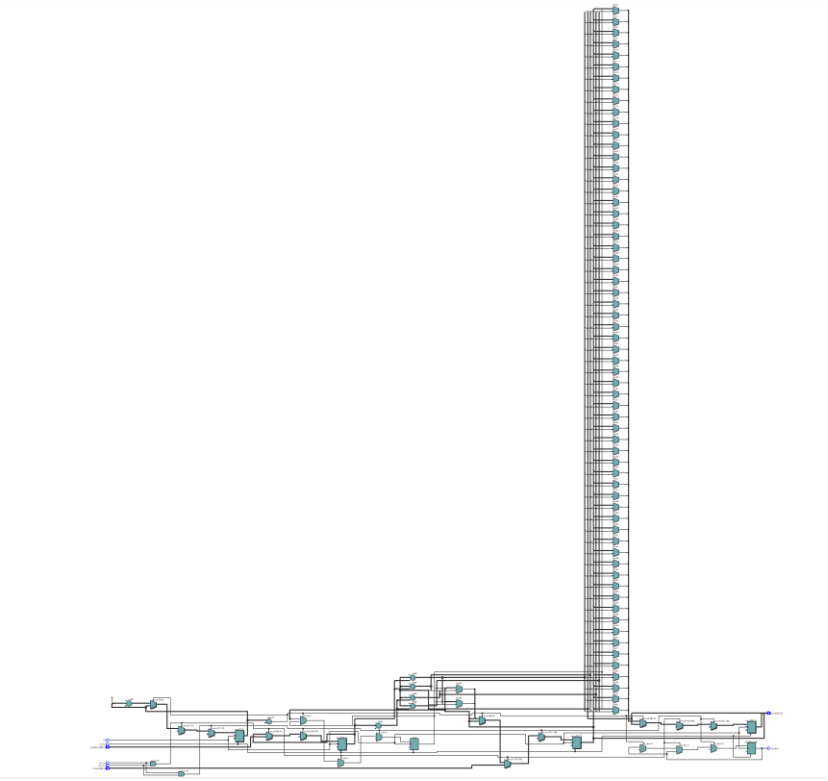
값을 23과 1384를 넣었을 때 31832라는 값이 나와 정상적으로 곱셈연산이 수행됨을 알 수 있다. Op_start가 1로 셋되어 값들이 변수에 저장된 후 0으로 떨어지고 나서의 clock부터 총 32번의 연산이 이루어졌고 최종적으로 32번째에 연산이 이루어진 후 count가 0이 되어 결과값이 저장되고 op_done의 값은 1이 되었는데 이는 32번째 연산이 끝난 후 적용되는 것이므로 33번째 클록에서 op_done의 값이 1로 정상적으로 변경되는 것을 확인할 수 있었다.

그리고 추가적으로 음수와 양수의 곱셈또한 정상적으로 수행되었다.

B. 합성(synthesis) 결과

<multiplier>(top module)

<RTL viewer>



64개의 비트를 shift해야하기 때문에 위와같이 위로 솟구친 모양의 RTL viewer를 얻을 수 있었고 아래에는 논리에 맞게 회로가 구현되었다.

<flow summary>

Flow Summary	
<<Filter>>	
Flow Status	Successful - Mon Nov 21 13:47:00 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)	N/A
Total registers	199
Total pins	261
Total virtual pins	0
Total block memory bits	0
Total DSP Blocks	0
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0
Total DLLs	0

컴파일이 성공적으로 완료되었음을 확인한다.

5. 고찰 및 결론

A. 고찰

Reset_n와 op_clear의 차이를 알 수 없어 조교님께 질문드린 결과 reset은 비동기로 구현하고 clear는 동기로 구현하라는 답변을 받아 reset은 전체 회로의 초기화를 목적으로 구현하고 clear는 연산이 마무리된 후 값들을 초기화하는 느낌으로 구현하였다. 한 모듈내에서 always 구문과 조건문을 통해 구현하려고 하였는데 구현하던중 서로 다른 always구문에서 같은 변수에 값을 넣으려고 하면 오류가 생긴다는 것을 알고 같은 always문에서 구현하려고 하였다. 또한 단순한 숫자로 값을 넣는 것은 쿼터스에서 32비트로 인식하는 것이 default라는 것을 구글링을 통해 알았다.

B. 결론

Booth 알고리즘을 통한 multiplier 구현을 하는 실습이었는데 booth 알고리즘이 잘 이해되지 않아 실습자료와 구글링을 통해 개념을 학습하는데 더 많은 시간을 사용한 것 같다. 또한 개념을 이해하고 나서 코드로 구현하려니 또 다른 어려움이 많이 생겼던 것 같다. 실습을 통해 완전히 코드를 구현한 결과 쿼터스에서 코드를 구현하는데 주의해야할 점 같은 것과 오류에서 발견한 새로운 지식이 있었다. 이번 실습에서 구현한 multiplier 모듈은 radix-4로 구현하여 3비트씩 비교하여 연산하고 2bit shift하는 방법을 사용하였다. 단순한 binary multiplier와는 다르게 booth algorithm의 원리에 따라 2진수의 1들은 두개의 이진수의 차로 나타낼 수 있다는 것을 이용하여 연산의 수를 줄이고 1을 만났을 때 승수의 shift를 활용하여 연산을 하는 특징때문에 효율적이었다. booth multiplier방법을 사용해 보면서 지식이 쌓일수록 구현은 복잡해지나 시간적인 것을 고려하여 회로를 구현할 수 있는 실력이 쌓인 것 같았다.

6. 참고문헌

공영호 교수님/컴퓨터공학기초실험2/2022

Booth algorithm/https://suintodev.tistory.com/3