컴퓨터 공학 기초 실험2 보고서

실험제목: Booth Multiplier

실험일자: 2022년 11월 19일 (토)

제출일자: 2022년 11월 19일 (토)

학 과: 컴퓨터공학과

담당교수: 공영호 교수님

실습분반: 화요일 0, 1, 2

학 번: 2019202053

성 명: 신윤석

1. 제목 및 목적

A. 제목

Booth multiplier

B. 목적

Booth multiplier는 기본적으로 multiplier로, 입력받은 두 개의 숫자를 곱해주는 장치이다. 곱셈은 덧셈을 여러 번 반복하는 것을 의미한다. 이러한 연산을 빠르게 해결할 수 있는 알고리즘 중에 Andrew D. Booth의 알고리즘을 적용하여 곱셈을 bit shift와 덧셈으로 빠르게 연산하는 장치를 만들어보자.

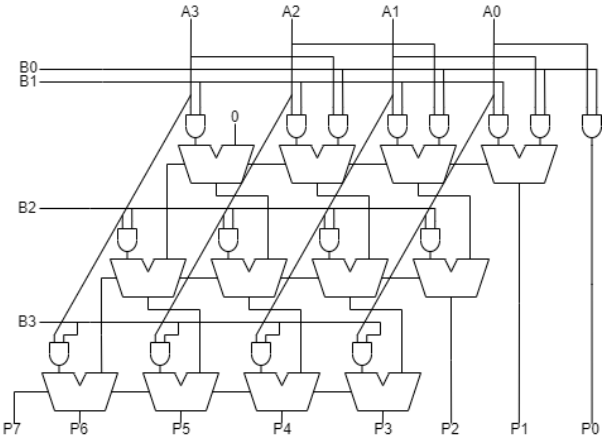
2. 원리(배경지식)

일반적으로 곱셈의 정의는 $a \times b$ 가 있다고 할 때, a를 b번 더하라는 의미가 있다. 7×8 을 한다고 하면 7을 7+7+7+7+7+7+7+7 = 56이 나오는 것이다. 사람은 곱셈을 10진수로 한다. 10진수로 곱셈을 할 때는 보통 직접 더해서 곱셈을 하지 않고 한 자릿수의 곱셈을 해서 나오는 경우의 수를 모조리 외워서 곱셈을 수행한 뒤 각 자릿수에 맞게 shift를 해서 더해서 계산을 한다.

이진수에 이러한 곱셈 방법을 적용시켜 계산할 수 있다. n bit 2진수 a와 b의 곱셈을 하는 경우에 a와 b의 n번째 bit와의 곱셈을 n번, 그리고 그렇게 나온 결과 n개를 shift하고 더해야 한다. 이와 같은 방식으로 hardware를 설계한다면 n번 반복하는 과정을 병렬화를 통해 쉽게 해결할 수 있다. 각 bit별 곱셈은 and gate로 표현할 수 있다. 어차피 0과 1의 곱셈이니 입력이 둘 다 1이면 결과는 1, 그렇지 않다면 결과는 0. and gate의 계산결과와 똑같다.

인간의 방식 그대로 곱셈기를 재현하면 다음과 같다.

$$
\begin{array}{cccccccc}
 & & & A_3 & A_2 & A_1 & A_0 & \\
 & & \times & B_3 & B_2 & B_1 & B_0 & \\
\hline
 & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 & \\
 & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & & \\
 & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$



위와 같은 곱셈식을 오른쪽 그림과 같은 회로로 바꿀 수 있다. 해당 회로의 장점은 1cycle만에 바로 곱셈 결과가 나온다는 점이다. 대신 delay가 긴 Ripple Carry Adder(RCA)를 이용하고, 또 겹겹이 이용하기 때문에 delay가 무척 길어진다는 단점이 있다. clock 속도를 빠르게 하기 위해 pipelining을 하는 경우에는 각 RCA 사이에, 위 Logic gate회로 기준으로 가로로 사이사이에 Register를 넣을 수 있을 것이다. 해당 회로의 결정적 문제는 bit가 많아질수록 RCA가 bit수만큼 많아진다는 점이 있다.

이러한 계산방법을 개선하기 위한 Andrew D. Booth의 알고리즘이 있다. Andrew D. Booth의 알고리즘은 다음과 같은 순서로 적용을 하게 된다.

Radix-2 Booth's multiplication:
n bit 피승수인 *multiplicand,* n bit 승수인 *multiplier,* 중간 계산 결과인 2n bit *shifted_number*, 최종 계산 결과인 2n bit *result*가 있다고 하자.
*multiplier*의 LSB 다음 bit에 0을 붙여서 n+1 bit 이진수로 확장한다. 이때 *multiplier*의 LSB는 마지막에 추가로 이어 붙인 0이며, *multiplier*의 i번째 비트를 m_i라 하자.
다음을 n번 반복한다:

case($m_1 m_0$):
$m_1 m_0$이 00인 경우 : *shifted_number* := 0
$m_1 m_0$이 01인 경우 : *shifted_number*[2n-1 : n] := *multiplicand*
$m_1 m_0$이 10인 경우 : *shifted_number*[2n-1 : n] := -*multiplicand*
$m_1 m_0$이 11인 경우 : *shifted_number* := 0
endcase
result = *result* + *shifted_number*
result := *result* >> 1 (Arithmetic Shift Right)
multiplier := *multiplier* >> 1 (Arithmetic Shift Right)
여기까지 반복한다.

해당 방법은 radix-2 booth's multiplication으로 한 cycle에 1bit씩 숫자를 오른쪽으로 밀어가면서 계산하는 방법이다. n비트 이진수 두개를 곱하기 위해 n번의 cycle이 필요하게 된다.

이 알고리즘의 원리는 승수의 패턴을 보고 피승수를 더하고 빼며 컨트롤 하는 것이다. 우선 이진 양수의 패턴은 기본적으로 0으로 가득 차 있는데 1이 중간에 어느 위치에 몇 개가 끼어져 있는지로 볼 수 있다. 예를 들어 $0001110100110_{(2)}$의 경우 2^1위치부터 1이 시작해서 2^2위치에서 1이 끝난다. 마찬가지로 2^5에 위치에는 1이 시작하자마자 끝난다. 2^7의 위치에서 1이 시작하여 2^9의 위치에서 끝난다. 컴퓨터에서 이진수 체계는 양수는 MSB가 무조건 0으로 끝나므로 1로 끝나지 않는다는 보장이 있다. 또한 이 알고리즘은 LSB보다 한 비트 낮은 위치에 0을 첨가한다. 1은 시작과 끝이 보장되어 있다. 알고리즘에서 "$m_1 m_0$이 10인 경우"는 1이 시작한다는 의미를 갖고 $m_1 m_0$이 01인 경우는 1이 끝난다는 의미를 갖는다.

또 한가지의 원리는 위 알고리즘에서는 *multiplier*와 *result*를 ASR을 하면서 진행한다. 즉, 다음 실행은 이번 실행의 2배의 의미를 갖는다. 즉 *shifted_number*에 ±*multiplicand*를 넣고 *result*에 더하는데 ±*multiplicand*가 반복문이 실행될 때마다 *result*입장에서 2배씩 커진다.

모든 결과를 종합하면 만약 *multiplier*에서 1이 2^n에서 시작해서 2^m에서 끝난다고 할 때 이 숫자를 위 알고리즘에 적용시키면 *result*에는 $(2^m - 2^n) \cdot multiplicand$의 결과가 남

게 된다. 이때, $2^m - 2^n = 2^{m-1} + 2^{m-2} + \cdots + 2^{n+1} + 2^n$ 이므로 결론적으로 알맞게 곱한 결과가 된다.

*multiplier*가 음수인 경우를 생각해 보자. $multiplier = r = \sum_{k=0}^{n-1} 2^k r_k$라 하자. *r*이 음수라는 의미는 $2^n - r$은 양수라는 의미이다. 앞서 설명했던 *multiplier*가 양수인 경우의 연산을 하기 위해 $multiplicand \times (2^n - r) = 2^n multiplicand - r \cdot multiplicand$가 나오게 된다. 즉 마지막 $2^n multiplicand$만 계산결과에서 빼 주면 원하던 $-r \cdot multiplicand$만 남아 음수의 곱셈이 완성된다. 그런데 음수는 MSB가 0이 아닌 1이므로 위의 알고리즘에서 $m_1 m_0$이 10인 경우나 $m_1 m_0$이 11인 경우로만 끝나게 된다. $m_1 m_0$이 10인 경우, 마지막 연산에서 $-2^n multiplicand$ 해주므로 결론적으로 별도의 처리 없이 위 알고리즘만으로 곱셈이 작동한다. $m_1 m_0$이 11인 경우 역시 어디선가 1이 시작되었고 1이 끝나지 않은 상태로 종료하므로 *r*이 양수인 곱셈에서는 $(2^n - 2^m) \cdot multiplicand$ 로 끝나야 했을 곱셈에서 $2^n multiplicand$가 사라져 $-2^m multiplicand$만 남아 결론적으로 $-2^n multiplicand$된 효과가 생긴다. 결론적으로 Radix-2 Booth's multiplication알고리즘은 *multiplicand*와 *multiplier* 모두 양수와 음수에 상관없이 연산이 가능하다.

Radix-4 Booth's multiplication:
n bit 피승수인 *multiplicand,* n bit 승수인 *multiplier,* 중간 계산 결과인 2n bit *shifted_number*, 최종 계산 결과인 2n bit *result*가 있다고 하자.
*multiplier*의 LSB 다음 bit에 0을 붙여서 n+1 bit 이진수로 확장한다. 이때 *multiplier*의 LSB는 마지막에 추가로 이어 붙인 0이며, *multiplier*의 i번째 비트를 m_i라 하자.
다음을 n/2번 반복한다:

case($m_2 m_1 m_0$):
$m_2 m_1 m_0$이 000인 경우 : *shifted_number* := 0
$m_2 m_1 m_0$이 001인 경우 : *shifted_number*[2n-1 : n] := *multiplicand*
$m_2 m_1 m_0$이 010인 경우 : *shifted_number*[2n-1 : n] := *multiplicand*
$m_2 m_1 m_0$이 011인 경우 : *shifted_number*[2n : n] := *multiplicand* × 2
$m_2 m_1 m_0$이 100인 경우 : *shifted_number*[2n : n] := -*multiplicand* × 2
$m_2 m_1 m_0$이 101인 경우 : *shifted_number*[2n-1 : n] := -*multiplicand*
$m_2 m_1 m_0$이 110인 경우 : *shifted_number*[2n-1 : n] := -*multiplicand*
$m_2 m_1 m_0$이 111인 경우 : *shifted_number* := 0
endcase
result = *result* + *shifted_number*
result := *result* >> 2 (Arithmetic Shift Right)
multiplier := *multiplier* >> 2 (Arithmetic Shift Right)
여기까지 반복한다.

해당 방법은 radix-2 booth's multiplication으로 한 cycle에 2bit씩 숫자를 오른쪽으로 밀어가며 하는 계산 방법이다. n비트 이진수 두개를 곱하기 위해 n번의 cycle이 필요하게 된다. 대신 계산이 조금 복잡하다. 기본적인 계산 원리는 radix-2와 동일하게 *multiplier*의

패턴을 분석하는 것이다. 그런데 이젠 두개 bit씩 본다. *multiplier*의 1이 1개만 나오는 경우와 2개 이상 무리 지어서 나오는 경우로 나눠서 생각할 수 있다. 우선 3개 bit를 볼 때, $m_2m_1m_0$에서 m_0는 이전 bit이고 기준 bit는 m_1이며 m_2는 두배로 생각해서 취급한다. 우선 1이 2개 이상 뭉쳐 다니는 경우를 생각해 볼 수 있다. 그런 경우 $m_2m_1m_0$이 100이거나 $m_2m_1m_0$이 110인 경우로 1이 시작되게 되고, $m_2m_1m_0$이 001이거나 $m_2m_1m_0$이 011인 경우로 1이 끝나게 된다. 인 경우, m_2에서 1이 시작했으니 기준보다 두배로 큰 수인 $-multiplicand \times 2$를 해주게 된다. $m_2m_1m_0$이 110인 경우는 기준인 m_1에서 1이 시작했으니 radix-2 알고리즘과 마찬가지로 $-multiplicand$를 해준다. 끝나는 것도 마찬가지이다. $m_2m_1m_0$이 001인 경우 기준인 m_1에서 1이 끝났기 때문에 *multiplicand*를 더해주게 된다. $m_2m_1m_0$이 011인 경우 한 칸 더 와서 1이 끝났기 때문에 $multiplicand \times 2$를 해주게 된다. $m_2m_1m_0$이 010이거나 101인 경우 중간에 1이나 0이 하나 들어왔다고 보고 각 계산을 해주게 된다. 결론적으로 각 *multiplicand*마다 자릿수 2^n을 곱해주면 적절한 계산임을 알 수 있다.

3. 설계 세부사항

이번 과제에서는 Radix-4 Booth's multiplier를 구현한다. 이때, 각 register에 있는 값을 기준으로 계산하기 때문에 sub module은 각 register에 있는 값을 control하기 위한 컨셉으로 설계한다. 필요한 각 register는 다음과 같다.

- 2bit register : state를 저장한다.
- 5bit register : count를 하기 위한 수를 저장하는 register. multiplier가 64bit이고 Radix-4 Booth's multiplication로 구현하기 때문에 32까지의 숫자를 count할 필요가 있으며, 실질적으로는 0~31까지가 32개의 숫자이 때문에 5bit로 설정한다.
- 64bit register : multiplicand를 저장한다. op_start신호가 들어온 이후부터는 입력 값이 일정하다는 보장이 없기 때문에 register에 따로 저장해 둔다.
- 65bit register : multiplier를 저장한다. LSB에 추가로 0을 붙이기 위해서 65bit register가 필요하다.
- 128bit register : 결과값을 저장하기 위한 register. 계산 중간중간에 나오는 수를 더하기 위해 op_done신호가 출력되기 전 까지는 값이 변한다.

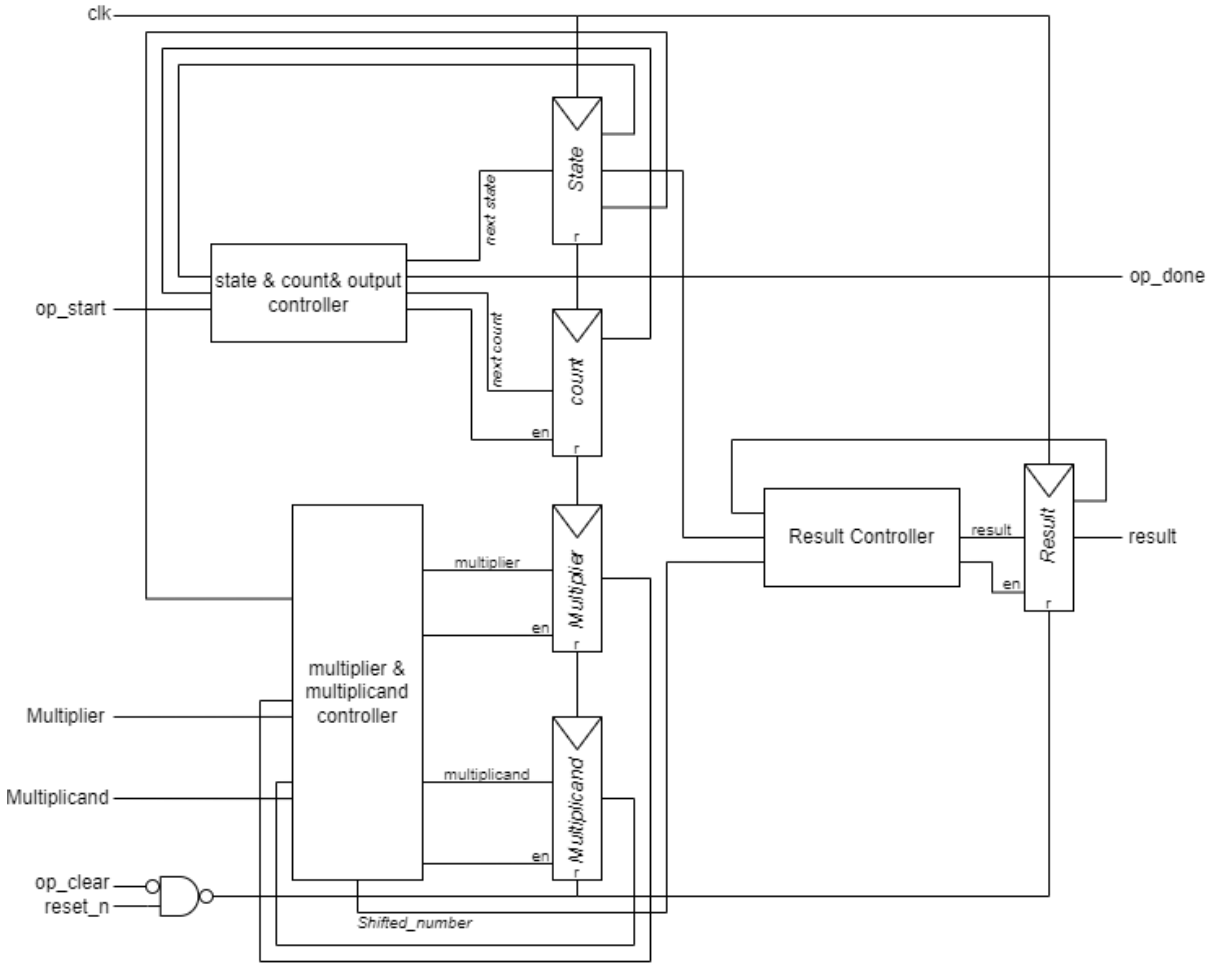
해당 regster를 control하기 위한 sub module은 다음과 같다.

- `state & count & output controller` : 원래는 next state logic으로 만들려고 했는데 하다 보니 state에 의해 output(`op_done`신호)이 결정되어서 output도 출력하고, 어차피 count도 계산중인 상태에서는 계속 1씩 올려야 하므로 세 개를 묶어서 설계하였다.
- `multiplier & multiplicand controller` : 2.원리(배경지식)에 있는 알고리즘

표 속 *shifted_number*를 계산하기 위한 sub module이었다. *shifted_number*를 계산하기 위해선 *multiplier*와 *multiplicand*를 둘 다 다룰 필요가 있기 때문에 해당 register도 관리해 준다.

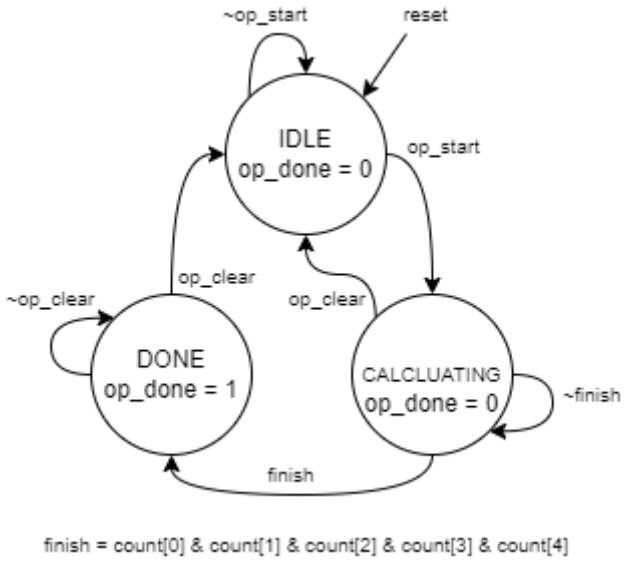
- `result controller` : *result*를 관리한다. *result*에 *shifted_number*를 더해주는 역할을 한다.

이번 과제의 sub module과 register는 다음과 같이 배치된다.



state register를 제외하고는 state에 따라 값이 갱신되어야 하는 경우도 있고, 갱신되면 안되는 경우가 있기 때문에 enable신호를 각 controller에게서 받는다. 이를 통해 원하는 state일 때만 register에 data를 갱신하게 된다.

우선 `state & count & output controller`는 State를 control하기 위해 state encode를 정의한다. 이에 따른 state encode와 state diagram은 다음 도표와 같다.



state	encode
IDLE	00
CALCULATING	01
DONE	10

해당 도표에서 IDLE state는 곱셈 계산이 시작하기 전 *mutilplier*와 *multiplicand*를 받는 준비를 한다. op_start가 1이 되면 CALCU-LATE state가 되어 계산을 시작하게 된다. count가 31이 되어 곱셈 계산이 끝나면 state는 자동으로 DONE이 되며 op_done에 1을 출력하고 멈추게 된다. 그리고 해당 상태에서 reset이나 op_clear신호가 들어오기 전까지는 DONE상태를 유지하게 된다. 이번 과제물이 op_clear신호를 받았을 때의 행동과 reset신호를 받았을 때의 행동이 모든 register를 0으로 초기화하고 *state* register가 0이 되면 다시 IDLE로 돌아가기 때문에 두 신호를 and gate로 묶어 각 flip flop reset신호로 넣어준다. 추가로 `state & count & output controller`는 *count* register와 `output`도 관리를 해줘야 한다. *count* register의 경우 현 state가 CALCULATING인 경우 현 *count*출력에 +1한 값을 *count* register의 입력으로 넣어준다. 이 행동을 위해 `counter enabler` 또한 현 state가 CALCULATING인경우에만 1을, 나머지 state의 경우에는 0을 출력하여 register를 control한다. output은 단순히 encode의 MSB와 동일하게 된다. 위의 기술된 내용을 바탕으로 procedural하게 sub module 설계를 진행한다.

`multiplier & multiplicand controller`는 *multiplier* register와 *multiplicand* register를 다루고, 2.원리(배경지식)에 기술된 Radix-4 Booth's multiplication algorithm의 *shifted_number*를 계산하게 된다. 이번 설계에서는 위 알고리즘처럼 더하고 shift를 하는 것이 아닌 shift를 먼저하고 더하게 된다. 따라서 *shifted_number*는 $m_2m_1m_0$이 000이거나 111 경우 0이되는건 맞지만 $m_2m_1m_0$이 001이거나 010 경우 *shifted_number*는 *multiplicand*를 상위 64bit에 쓰고 Arithmetic Shift Right를 2만큼 한 결과가 된다. $m_2m_1m_0$이 011인 경우 *multiplicand*를 *shifted_number*의 상위 64bit에 쓰고 Arithmetic Shift Right를 1만큼 한 결과를 넣고 $m_2m_1m_0$ 이 100인 경우 -*multiplicand*를 *shifted_number*의 상위 64bit에 쓰고 Arithmetic Shift Right를 1한 값을 출력한다. 이렇게 하면 원래는 우로 두 칸 shift를 해야하는데 그 상태에서 좌로 한 칸 shift한 것이 되어 곱하기 2의 효과가 있게 된다. $m_2m_1m_0$이 101인 경우와 $m_2m_1m_0$이 110인 경우는

*shifted_number*는 *-multiplicand*를 상위 64bit에 쓰고 Arithmetic Shift Right를 2만큼 한 결과가 된다. 또한 *multiplicand*와 *multiplier* register를 관리해주게 되는데 *multiplicand* register의 경우에는 IDLE state에서만 enabler신호가 true가 되면서 수정이 가능하고, *multiplier*는 DONE state를 제외한 나머지 state일 때 enabler가 1이 되면서 수정이 가능해진다. IDLE state에서는 외부 `multiplier`입력을 받아 *multiplier* register의 값을 갱신해 주다가 CALCULATING state가 되면 *multiplier*를 clk가 rising edge일때마다 *multiplier* ASR 2한 값을 넣어주게 된다.

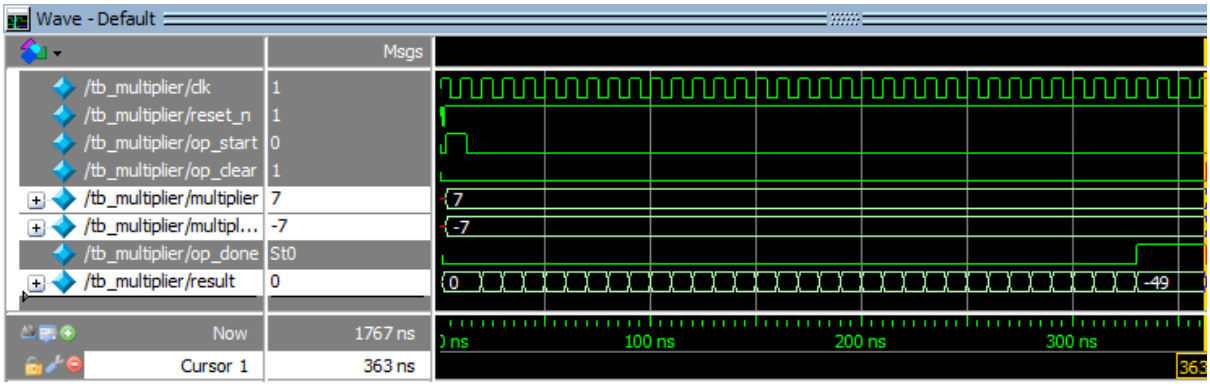
마지막으로 `result controller`는 *result* register를 관리하는 역할을 한다. 현 state가 CALCULATING일때만 enabler신호가 true가 되면서 *result*의 값이 갱신된다. 이때 갱신은 단순히 현 *result*값에 `multiplier & multiplicand controller` 유닛에서 나오는 shifted_*multiplicand* 신호의 값을 더해서 갱신할 새 *result*값을 출력하는 역할을 한다.

위의 해당사항을 verilog의 procedural assignment를 통해 구현한다.

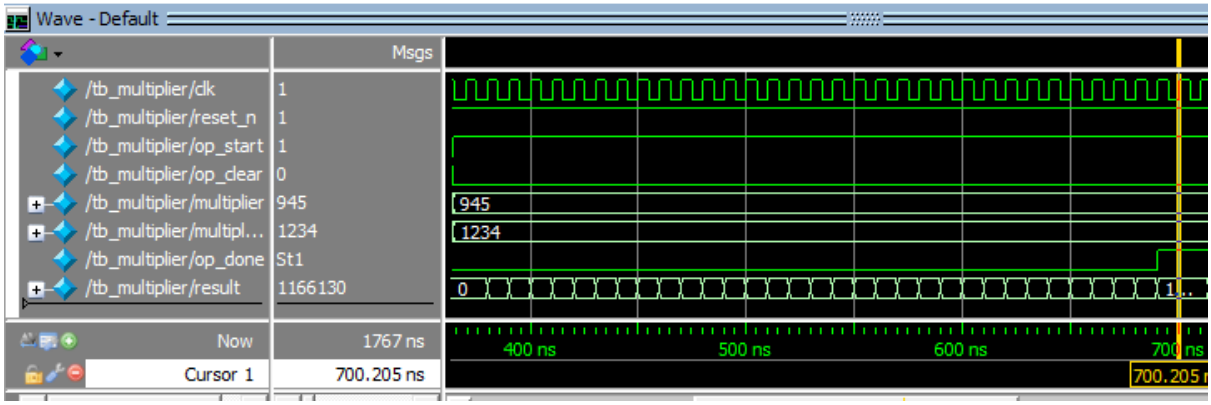
4. 설계 검증 및 실험 결과

A. 시뮬레이션 결과

이번 시뮬레이션에서는 Top module인 `Multiplier`를 test bench에 instantiate해서 계산 결과가 알맞게 나오는지 확인을 하고 test bench내부에 `Multiplier`의 sub module을 instantiate하고 top module과 동일하게 배선을 꾸며 특수한 상황에 특수한 작동이 잘 되는지 확인을 할 것이다. 특수한 상황이라 함은 op_start가 들어오고 난 이후에 입력값이 바뀌어도 IDLE state 때 입력받은 값에 대한 계산 결과를 잘 내놓는지, 중간에 op_clear나 reset_n신호가 들어오면 reset이 잘 되는지를 검증한다.

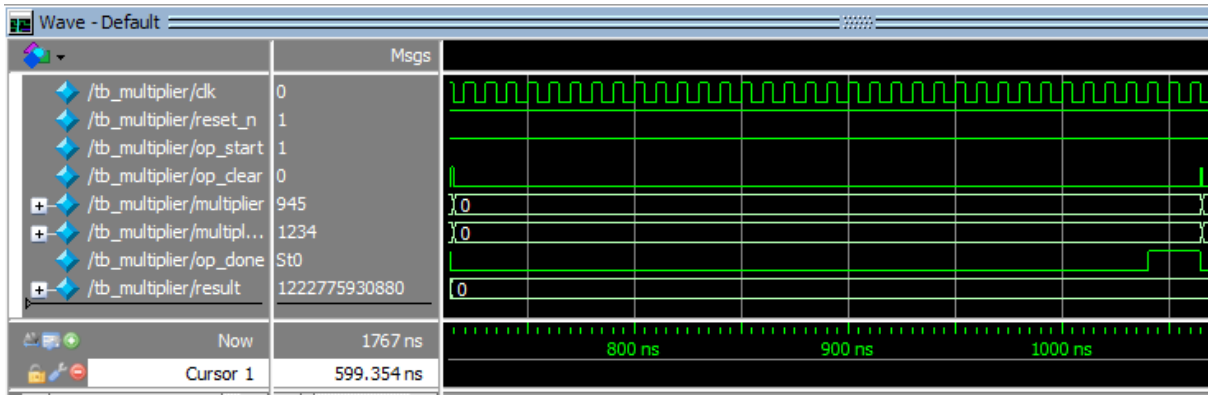


우선 예제로 나온 7*(-7)을 해 보았다. 32cycle만에 계산결과 -49가 잘 나왔다.

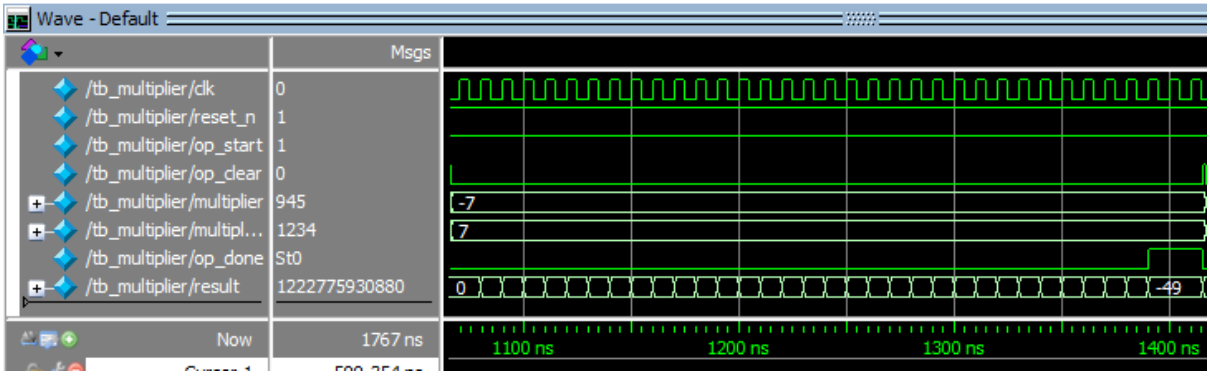


이번엔 임의의 양수 곱셈 945*1234를 해 보았다.

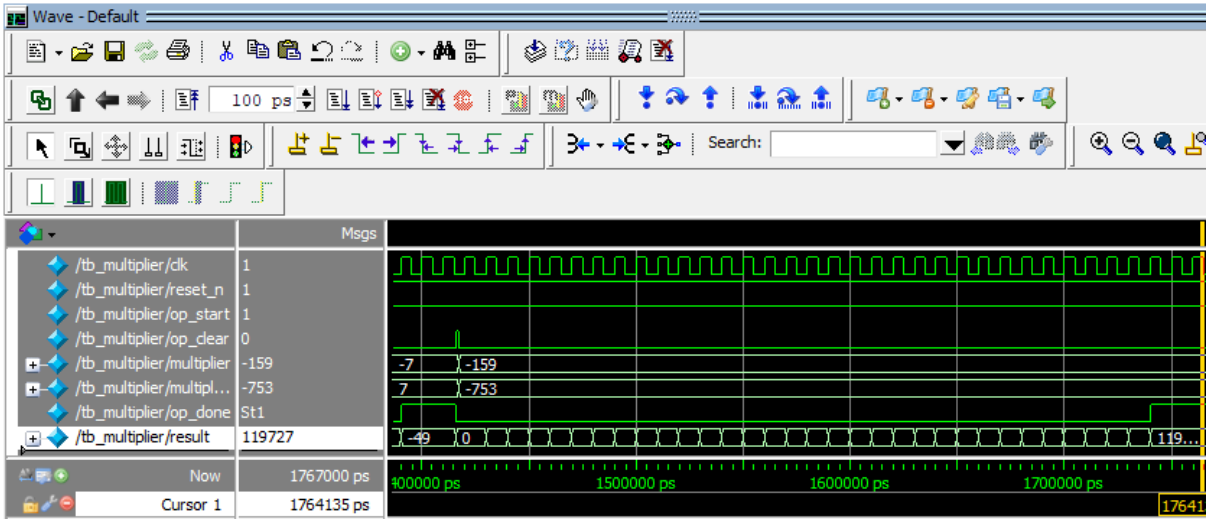
계산기의 결과인 1,166,130이 test bench wave form에도 계산 결과로 잘 나와 있다.



0*0의 경우는 *result*의 변화 없이 0이 나왔다.



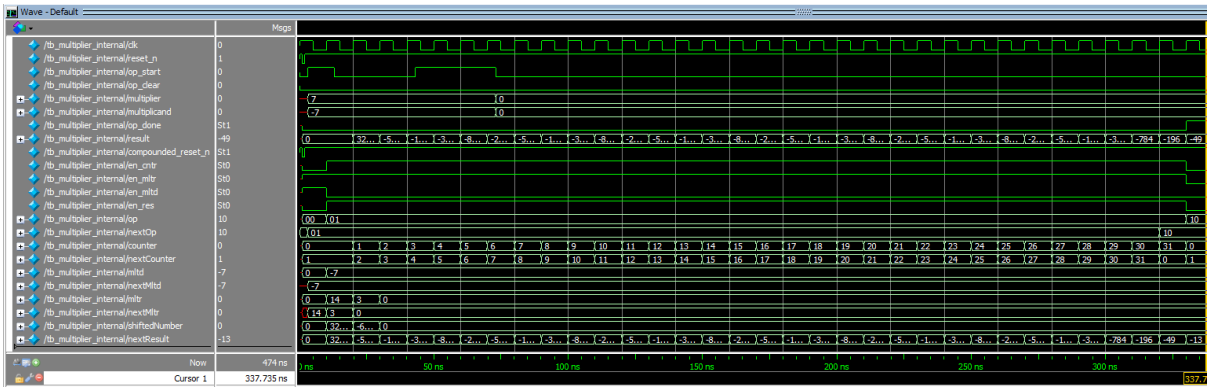
이번엔 (음수)*(양수)로 -7*7을 해 보았다. 그리고 기대값인 -49가 잘 나왔다.



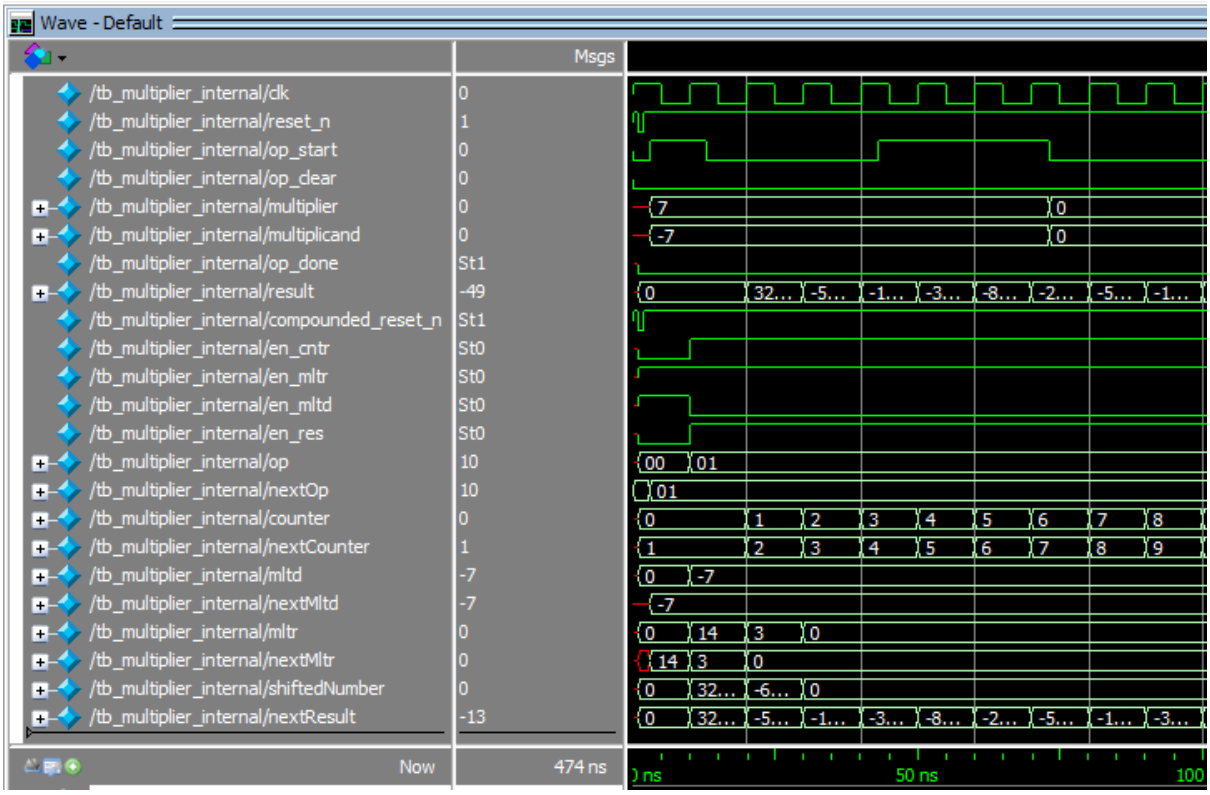
마지막으로는 (음수)*(음수) 곱셈을 위해 임의의 숫자인 -159*(-753)을 곱했다.

결과는 계산기와 동일하게 119,727이 잘 나왔다.

다음은 submodule 계산결과이다.

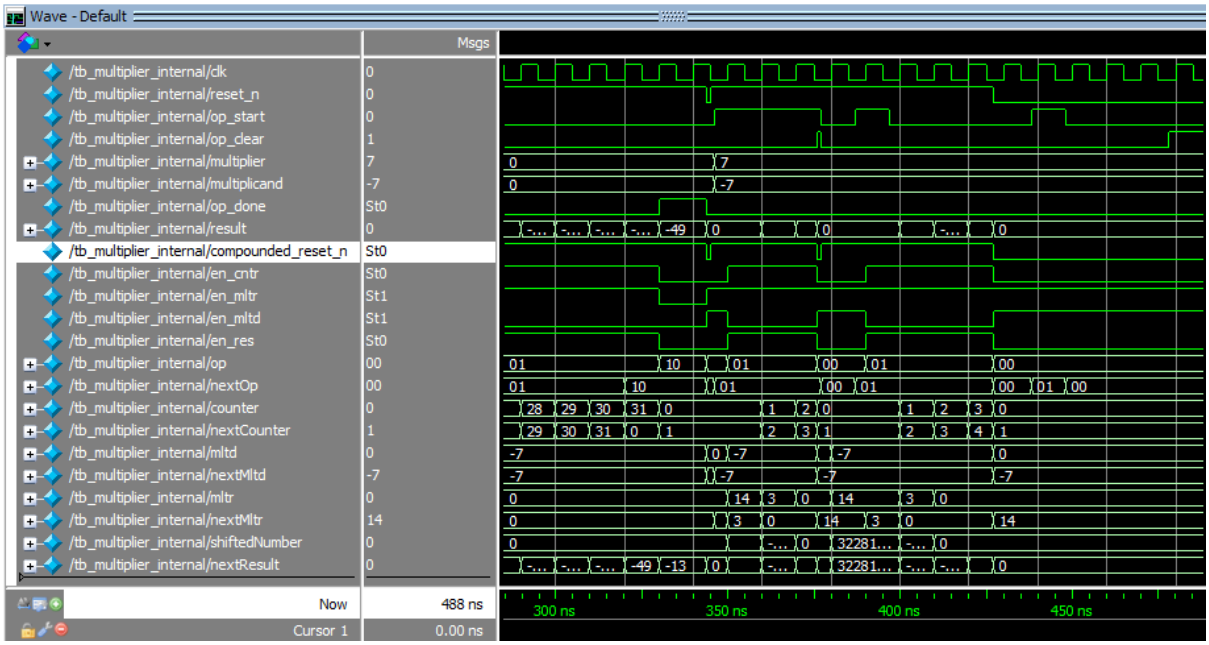


위와 같은 -7*7 전체 과정 중에서



0~100ns까지 입력 값의 변화가 생긴다. 이때, 10ns에 `op_start`를 인식해서 `op`가 01이 되면서 CALCULATING 상태로 진입한다. 해당 상태에서는 register에 필요한 값이 저장되었기 때문에 입력 값이 어떻든 간에 정상적으로 작동하는 모습을 보인다. 0~10ns까지 IDLE상태에서 새로운 값을 받기 위해 `en_mltd`(multiplicand), `en_mltr`(multiplier)이 1이 되어 register에 새로운 값을 쓰기 위해 준비한다. CALCULATING 상태로 진입하면서 `en_cntr`(counter), `en_mltr`, `en_res`(result)가 1이 되며 쓰기 활성화 상태로 진입하고 `en_mltd`는 0이 되며 *multipliand*에 새로운 값을 넣지 않게 된다. 이에 `op_start`가 0이 되거나 1이 되든 상관없이 `counter`신호가 31이 될 때까지 state는 CALCULATING을 유지한다. 또한 `multiplier`, `multiplicand`입력이 바뀌어도 CALCULATING상태에선 입력과 상관없이 내부 값을 알아서 잘 갱신하며 계산을 진행한다. 그래서 나온 결과값은 다음 사진과 같이 330ns

에서 -49를 잘 계산해 냈다.



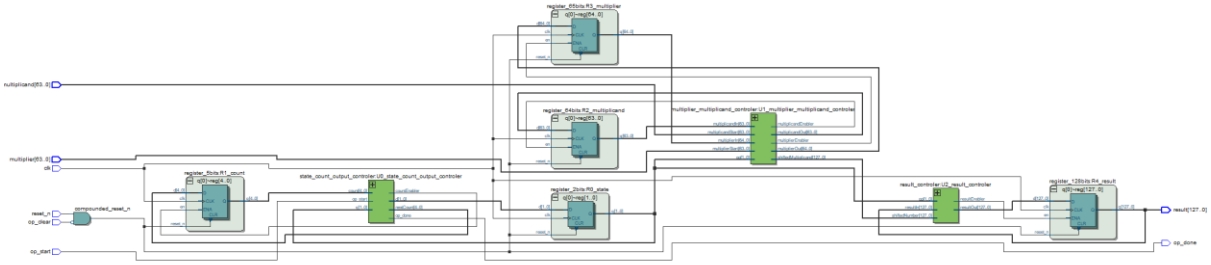
340ns 부터는 reset_n과 op_clear신호에 대한 테스트이다. 근본적인 원리는 두 신호 다 Flip Flop clear 신호로 들어가서 해당 신호가 1이 되면 Flip Flop이 0으로 초기화 되는 방식이다. compounded_reset_n신호가 바로 ~op_clear & reset_n 신호로써 해당 신호를 Flip Flop의 clear신호로 들어가게 되는 것이다. 345ns 부근에서 op_done 상태를 초기화하기 위해 reset_n이 잠시 0을 찍고 1이 되어서 상태가 IDLE로 리셋된다. 이후 정상적으로 계산을 하다가 370ns 부근에서 op_clear신호가 들어왔고 compounded_reset_n이 0이 되면서 모든 register가 초기화가 된다. 마찬가지로 455ns부근에서 reset_n이 0이 되며 compounded_reset_n신호가 0이 되면서 계속 초기화 된 상태로 FSM이 멈춰있는 모습을 확인할 수 있다.

이로써 검증을 완료한다.

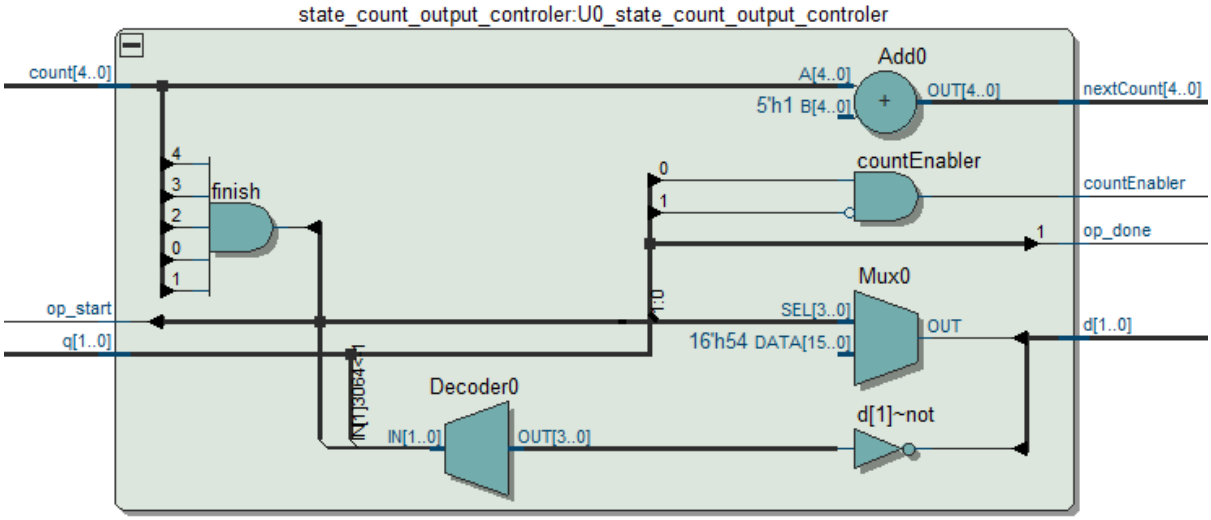
B. 합성(synthesis) 결과

Flow Summary	
Flow Status	Successful - Sat Nov 19 16:28:31 2022
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)	N/A
Total registers	264
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

해당 합성 결과는 multiplier에 대한 합성결과이다. 2022년 11월 19일 토요일 16시 28분 31초에 합성이 완료되었음을 알려준다.



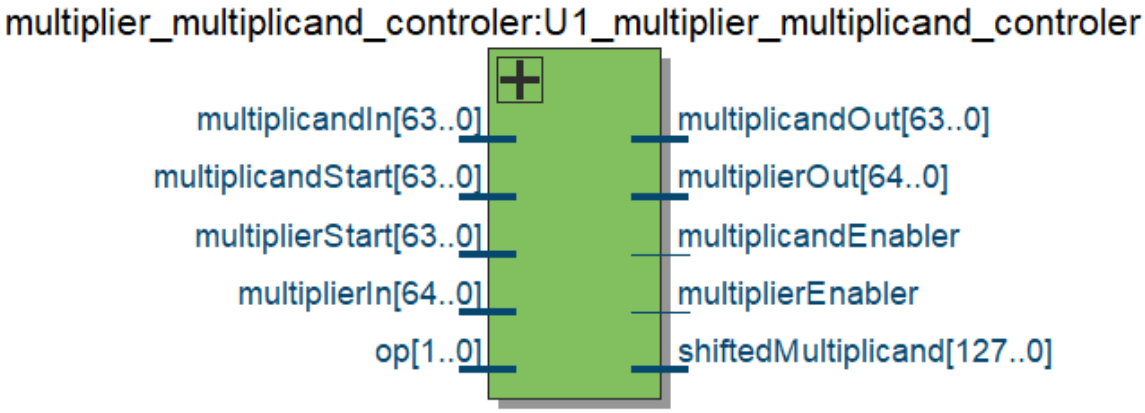
RTL viewer로 본 전체 모습은 다음과 같다. sub-module별로 차근차근 살펴보자.



우선 `state_count_output_controller`이다. 해당 logic은 *state, count* register와

output인 `op_done`신호를 구하기 위해 procedural assignment로 구현을 했다. 이를 위해 verilog의 casex 구문과 +연산자의 assignment를 하였고 이에 adder와 알 수 없는 Decoder0, Mux0가 생겼다. 정확히 어떤 방식으로 구동하는지는 모르겠지만 해당 위 검증을 통해 해당 모듈이 의도한대로 잘 작동한다는 사실을 알 수 있었다.

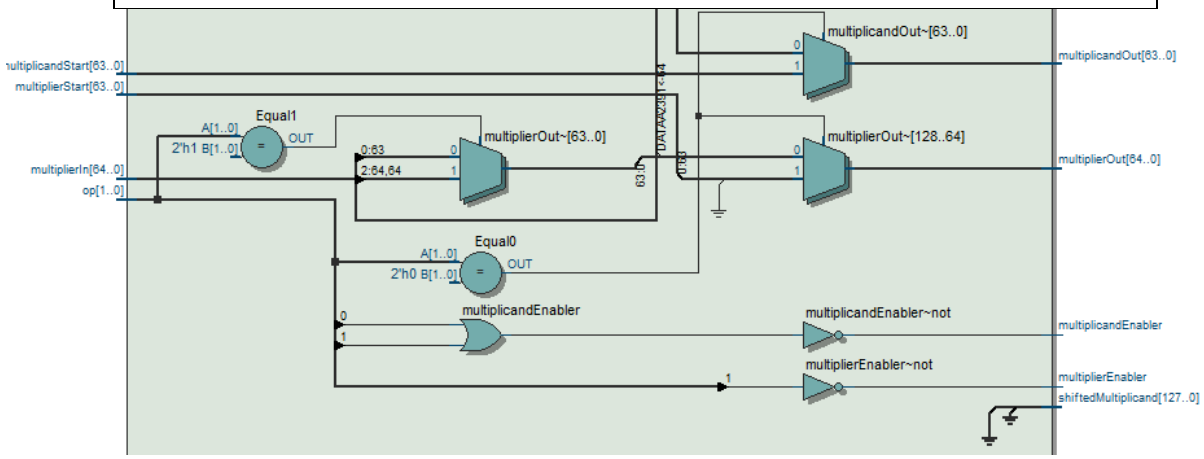
이번 모듈은 `multiplier_multiplicand_condtroller`이다.



register2개를 관리하고 출력으로 radix-4 booth's multiplication algorithm에서 다음 표의 내용과 같이 128bit짜리 *shifted_number*를 계산하기 때문에 Mux가 상당히 많이 들어있다.

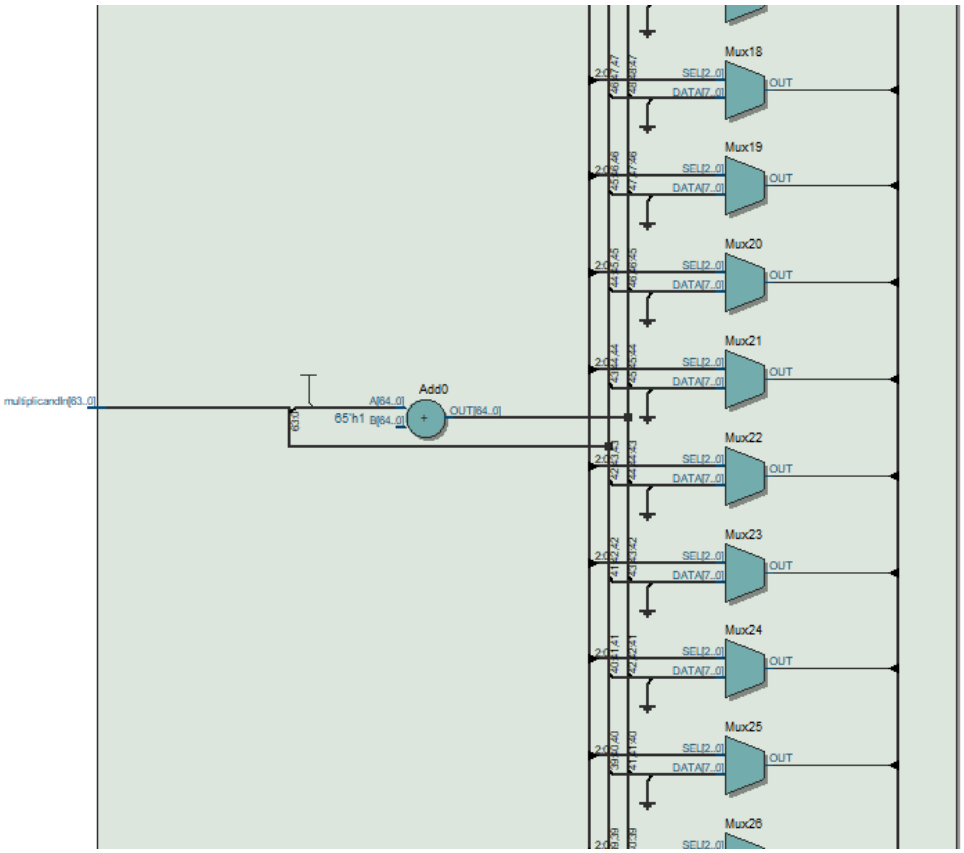
case($m_2m_1m_0$):
- $m_2m_1m_0$이 000인 경우 : *shifted_number* := 0
- $m_2m_1m_0$이 001인 경우 : *shifted_number*[2n-1 : n] := *multiplicand*
- $m_2m_1m_0$이 010인 경우 : *shifted_number*[2n-1 : n] := *multiplicand*
- $m_2m_1m_0$이 011인 경우 : *shifted_number*[2n : n] := *multiplicand* $\times 2$
- $m_2m_1m_0$이 100인 경우 : *shifted_number*[2n : n] := -*multiplicand* $\times 2$
- $m_2m_1m_0$이 101인 경우 : *shifted_number*[2n-1 : n] := -*multiplicand*
- $m_2m_1m_0$이 110인 경우 : *shifted_number*[2n-1 : n] := -*multiplicand*
- $m_2m_1m_0$이 111인 경우 : *shifted_number* := 0

endcase

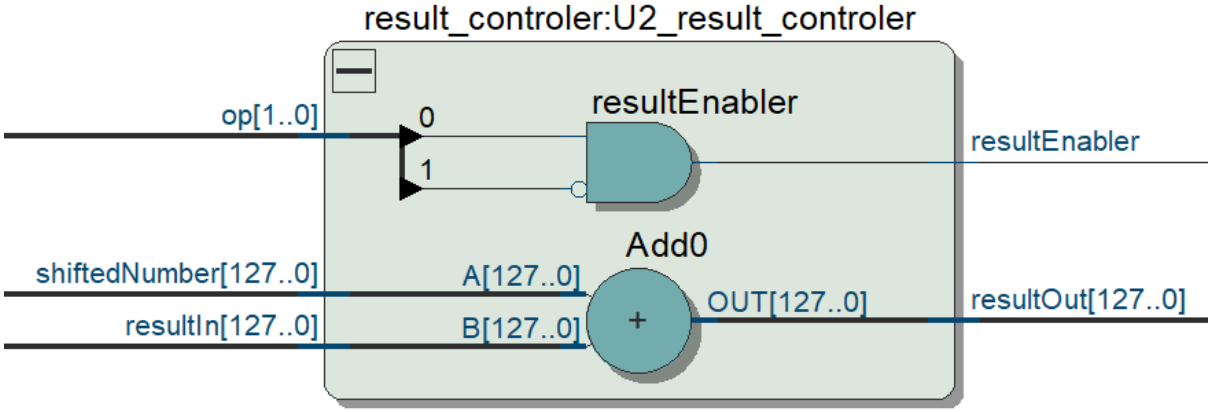


RTL viewer상에서 `multiplier_multiplicand_condtroller`는 상단부의 수많은 mux와 하단에 위와 같은 회로로 나오게 되는데, enabler는 op에 맞춰서 출력되는 모

습을 볼 수 있다. 또한 `Equal0`, `Equal1` cell이 조건을 비교해서 각 mux의 selector 신호로 쓰이는 출력을 내고 있다. 각 mux는 selector신호에 맞춰 필요한 값을 출력한다. 이때, `shifted_number` 출력은 ground신호를 출력하는 것이 아니라 해당 viewer상단부의 mux의 출력을 bus로 출력하고 있는 것이다.



수많은 mux 가운데 있는 Adder는 *multiplicand*의 음수를 구하기 위한 Adder이다. 또한 Adder아래로 따로 나온 bus는 원본 *multiplicand*를 의미한다. 저 많은 mux는 각 bit별로 *multiplier*의 [2:0]를 SEL신호로 받아 각 bit별 나와야 하는 숫자를 고를 것이다.



마지막 모듈은 `result_controller`로 *result* register를 관리한다. 단순히 `shifted_number`신호와 기존 *result* register에 들어있는 값을 더해서 출력한다. 또한 `op`가 01, CALCULATING상태일 때만 register를 활성화시키는 신호를 출력한다.

5. 고찰 및 결론

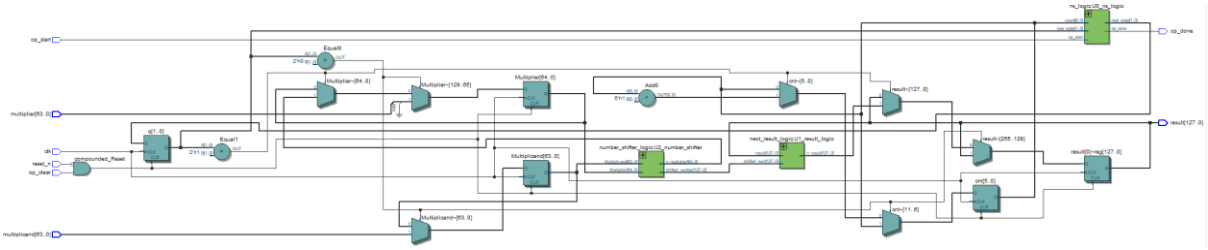
A. 고찰

우선 이번 과제에서 multiplier로 radix-4 Booth's multiplier로 구현을 했다. 가산점을 준다기에 고민을 해서 설계를 해 보았다. 그런데 $\pm 2multiplicand$부분에서 걸렸다. result를 이분했을 때의 상위 절반에 해당하는 부분에 $\pm 2multiplicand$를 더하기 위해선 1bit가 더 필요한 것이다. 그래서 생각한 부분이 shift와 덧셈의 순서를 바꾸는 것이었다. m ASR n연산자는 $m \cdot 2^{-n}$로 대체 가능하다. 즉 결론적으로 $(result + shifted\_number)$ ASR $2 = (result + shifted\_number) \cdot 2^{-n} = result \cdot 2^{-n} + shifted\_number \cdot 2^{-n}$ 로 분배법칙이 성립하게 된다. 그렇기 때문에 해당 연산을 할 때 덧셈을 먼저 하는 것이 아닌 두 수의 shift 연산을 먼저 한 뒤에 덧셈을 하는 방식을 취했다. 사실 이 방법이 아니더라도 *result*와 *shifted_number*의 범위를 MSB방향으로 2bit 더 확장시켜도 될 것 같긴 한데 그냥 register를 아끼기 위해 순서를 바꾸는 설계를 하였다.

이번 과제를 구현하는데 있어 처음에는 파일 하나에 모듈 하나로 만들려고 시도를 해 보았다. 그러나 불현듯 드는 생각이 RTL viewer가 보기 어려워질 것 같고, wire와 bus가 많아지면서 보기도 불편하고 헷갈려서 submodule을 추가하는 쪽으로 설계의 방향을 돌렸다. 그렇게 막 설계를 하니 뭔가 꼬여서 계산이 안되고 state가 IDLE로 고정이 되어서 결국 다시 설계를 했다. 그러니까 이번 과제물은 두번째 시도만에 만든 것이다.

두번째로 과제물을 만들면서 이번에는 sub module이 담당 memory를 관리하게 하자는 컨셉으로 만들었다. 그러면서 연관성 있는 register는 각각의 submodule이 아닌 하나의 sub module로 통합하였고 그래서 지금처럼 state, count register를 하나의 sub module이 관리하고 multiplier, multiplicand register를 하나의 submodule이 관리를 하고 마지막 result도 submodule 하나를 만들어 관리를 하게 만들게 된 것이다.

그리고 이렇게 설계를 하고나서 첫번째 과제물이 왜 state가 변하지 않는지 원인을 알 수 있었다. 심지어 RTL viewer도 정리가 안되어 있어 뭐가 문제인지 더더욱 알기 어려웠다.



실패작의 RTL view는 module 밖에 무언가 알 수 없는 mux와 가산기가 너무 많았다. 그리고 위치도 원하는 곳에 정렬이 되어있는 것이 아니라서 더더욱 보기 어려웠

다.

처음에는 count register를 32까지 갈 것을 염두 해 두고 6bit짜리로 만들었는데 32에서 종료를 하니 원하는 값보다 4배 낮은 값이 나와서 31에서 finish신호가 나오게 만들어서 31에서 종료를 하니 딱 원하는 결과가 나왔다. 결론적으로 count register의 bit를 하나 줄여서 설계를 할 수 있었다.

term project가 발표되었고, term project에서도 이 booth's multiplier가 필요한데, 거기서 쓰이는 multiplier는 +, - operator를 쓸 수 없기 때문에 adder를 직접 구현해서 써야 할 것이다. 또한 count도 1씩 올리는 과정을 adder를 통해서 해결했는데 term project에서는 count는 register가 아닌 32bit짜리 ring counter를 구현하여 해결할 것이다. 그렇다면 finish신호는 count[0] & count[1] & count[2] & count[3] & count[4]였는데 ring counter에서는 one-hot encoding으로 나오기 때문에 count[31]이랑 finish신호가 동치일 것이다.

아무튼 sub module을 만드는 기준을 가지고 설계를 하였더니 RTL view도 깔끔해서 보기 좋고 오류가 생겨도 대략적으로 어디서 생긴 오류인지 알 수 있어서 좋았다. 그리고 wave form으로 봤을 때 이게 진짜 될까? 싶었는데 진짜 돼서 솔직히 좀 신기했다.

2. 원리(배경지식)에서 언급했던 인간의 방식처럼 각 bit별로 partial product를 만들어 adder로 더하는 multiplier와 이번 과제로 구현한 Booth Multiplier에 관한 생각이다. 우선 partial product를 만들어 adder로 더하는 multiplier는 Partial Product Multiplier(PPM)라고 칭하겠다. PPM은 n-bit adder가 n-1개 생기기 때문에 회로가 길어져서 delay가 필연적으로 길어질 수밖에 사용할 수 없다. clock신호를 이용하는 회로를 설계할 때 긴 delay를 만회하기 위해서 adder와 and gate사이에 register를 넣어 pipelining하는 경우를 생각해 볼 수 있다.

PPM과 비교해 봤을 때 Booth multiplier는 이미 pipelining된 회로라고 볼 수 있다. 특히 radix-2 multiplier는 n-bit 곱셈을 하는데 n번의 rising edge가 필요하다. 마찬가지로 pipelining된 PPM을 생각해 보면 n-bit PPM은 adder가 n-1개였고 마지막 adder뒤에 오는 register는 생략한다고 했을 때 register를 n-2개를 거치므로 n-1번의 rising edge를 기다려야 결과를 얻을 수 있다. radix-2 multiplier는 이번 과제 style로 했을 때 필요한 register가 state 2bit, count $\lceil \log_2 n \rceil$bit, multiplier, multiplicand 각 n bit, result 2n bit 해서 총 $2 + \lceil \log_2 n \rceil + 4n$ bit가 필요하다. pipelined n-bit PPM의 경우는 2n bit register가 n-2개 필요해서 $2n^2 - 4$ bit register가 필요하다.

PPM은 비록 느리고 pipelining을 해도 register가 더 많이 필요하고 배선이 복잡하다는 문제만 빼면 FSM설계를 안 해도 되고 shifter가 필요 없이 and gate와 adder만을 통해 비교적 쉽게 구현할 수 있다는 장점이 있다. 대신 delay가 길고 unsigned

이진수에 대해서만 곱셈이 가능하다는 단점도 있다. 그에 비해 Booth Multiplier는 FSM이기 때문에 clock을 쓰는 회로에 넣어도 잘 동작하고(대신 latency가 있긴 하지만) signed 이진수에 대해서도 잘 작동한다는 장점이 있다.

ARM processor에서도 MUL명령어를 위해 Booth multiplier가 쓰인다고 한다. 2bit 단위로 끊어서 clock cycle하나를 잡아먹고 뒤에 모두 0이거나 1이면 그 즉시 연산을 종료한다고 하는데 아무래도 radix-4 booth multiplier가 쓰인 듯싶다. 이를 참고해서 radix-4 booth multiplier를 만든다 치면 multiplier를 ASR하면서 Z flag와 all 1 flag를 만들어서 해당 신호가 나오면 result를 남은 count * 2한 만큼 ASR하고 끝내는 방법이 있을 텐데, 이런 구현을 하자면 32 to 1 mux(count가 32까지 있으므로)를 만들어서 각 bit별로 mux에 연결을 해줘서 Z flag나 all 1 flag가 set되면 새로운 state에 도달하여 shift를 하고 종료하는 방식으로 설계하는 방식이 있을 것이다. 혹은 shifter를 구현하는 또 다른 알고리즘이 있다면 그렇게 해도 될 것 같은데 너무 복잡해질 것 같아 생각만 해보고 그냥 위 알고리즘 그대로 작동시켰다. ARM Processor는 Operand2에 Barrel shifter가 32bit 숫자를 원하는 만큼 shift를 해주니 가능한 연산이지 않나 싶다. 그리고 MUL같은 여러 cycle을 잡아먹는 명령어를 수행할 때 Processor 내부에서 여러 cycle에 걸쳐 명령어 없이도 숫자가 원하는 대로 돌아다니는 것도 신기하긴 하다.

B. 결론

곱셈을 위한 전자회로는 and gate와 adder를 통해 구현할 수 있다. 하지만 그렇게 할 경우 n bit곱셈을 하는데 adder가 n-1개 필요하며, delay가 무척 길어지게 되며 unsigned binary number에 대해서만 곱셈이 가능하게 된다. 이를 해결하기 위해 Andrew D. Booth의 알고리즘을 통해 곱셈을 구현해 보았다. adder와 shifter만으로 구현할 수 있는 Booth의 알고리즘은 FSM 설계를 통해 곱셈을 계산하는 multiplier를 설계할 수 있음을 시사한다. 이 multiplier는 특히 음수와 양수 상관없이 계산이 가능하고 특정 조건이 맞을 경우 곱셈을 미리 종료하여 더욱 빠르게 연산을 수도 있다.

6. 참고문헌

이준환/디지털논리회로1/광운대학교(컴퓨터정보공학부)/2020

공영호/디지털논리회로2/광운대학교(컴퓨터정보공학부)/2022

정현우, 김수현/컴퓨터공학기초실험2/광운대학교(컴퓨터정보공학부)/2022

이형근/어셈블리프로그램설계및실습/광운대학교(컴퓨터정보공학부)/2022

Sarah L. Harris & David Money Harris, *Digital Design and Computer Architecture*, Elsevier Korea L.L.C, 2016.

논리회로, state diagram 작성도구 / Diagram Software and Flowchart Maker / https://www.diagrams.net/