컴퓨터 공학 기초 실험2 보고서

실험제목: Multiplier

실험일자: 2016년 11월 03일 (목)

제출일자: 2016년 11월 17일 (목)

학 과: 컴퓨터공학과

담당교수: 이혁준 교수님

실습분반: 목요일 0, 1, 2

학 번: 2013722095

성 명: 최재은

1. 제목 및 목적
   1. 제목

Multiplier

* 1. 목적

외부로부터 입력된 multiplier(승수)와 Multiplicand(피승수) 곱을 연산하는 multiplier를 구현해본다. 이를 위해 곱셈을 진행하는 방법을 공부한다. 이에 대한 내용으로 Binary Multiplication과 Booth’s Multiplication을 알아보고 공부하여 원리를 이해하고 이를 실험 내용에 적용하여 모듈을 구현한다. 이 때 내부적으로 어떤 동작을 하며 출력되는 signal이 언제, 어떠한 조건에서 출력되는지 확인한다.

1. 원리(배경지식)

Binary Multiplication은 피승수에 승수의 모든 자리 값의 비트들을 1대 1 방식으로 곱하여 그 결과들을 모두 합하는 방식으로 우리가 일상에서 사용하는 일반적인 곱셈방법과 동일하다고 볼 수 있다. 설계하기 간단하다는 장점이 있으나 승수의 모든 bit를 확인하고 곱한다는 점에서 그에 소요되는 시간이 길다는 단점이 있으며 음수의 승수를 곱할 경우에는 제일 마지막에 피승수의 보수 값을 더해주어야 한다는 추가적인 절차가 생긴다.

이를 보완한 multiplication 중 하나인 booth’s multiplication은 승수의 자리 값을 이동하면서 그 자리 값의 bit(Xi)와 이전 자리 값의 bit(Xi-1)를 보고 그들의 조합에 따라서 각각 다른 동작을 수행하도록 설계되어있다. 단 처음에 X[0]을 볼 때는 Xi-1을 0으로 하여 시작한다. Xi와 Xi-1에 대한 경우의 수는 아래와 같다.

|  |  |  |  |
| --- | --- | --- | --- |
| Xi | Xi-1 | Operation | Description |
| 0 | 0 | Shift only | String of zeros |
| 0 | 1 | Add and shift | End of a string of ones |
| 1 | 0 | Sub and shift | Beginning of a string of ones |
| 1 | 1 | Shift only | String of ones |

Booth’s multiplication을 수행하는데 있어서 U와 V라는 개념이 적용되는데 이들의 초기값은 0으로 시작한다. Xi와 Xi-1이 각각 01 / 10인 경우에 Add와 sub를 해주게 되는데 이때

각각의 경우에 U에 피승수 값을 더하거나 빼준다. 이후 shift라는 동작을 수행할 때, V를 LSL 1 한 뒤, U의 LSB가 V의 MSB가 되며, U는 add/sub후의 결과값을 ASR 1하게 된다.

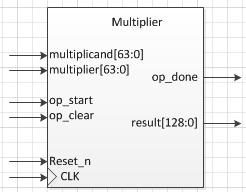
이러한 동작을 승수 값인 X의 모든 자리 값을 확인할 때까지 반복하게 된다.

1. 설계 세부사항

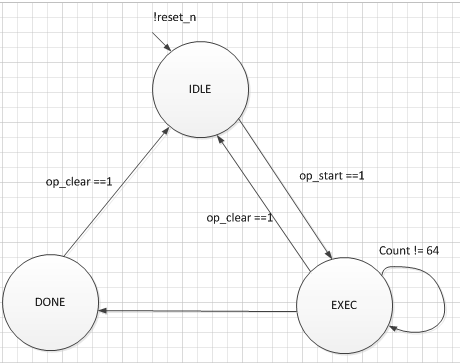
- X의 모든 자리 값 비트들을 확인하는 과정에 있어서, X를 clk이 뛸 때마다 circular shift하도록 설계하였습니다. 이렇게 shift된 값이 multiplier(피승수)와 동일해지면 연산을 다 한 것으로 보고, 동작을 멈춥니다.

- U와 V의 값을 만들어 줄 때에는 concatenation 하여 그 코드 사이즈를 최소화 하였습니다.

- 아래와 같은 간단한 형태로 표현될 수 있습니다.



- 아래와 같은 state진행을 갖습니다.



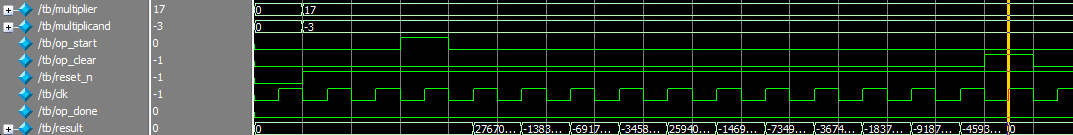
- 사용되는 인자들은 다음과 같습니다.

|  |  |  |  |
| --- | --- | --- | --- |
| 구분 | 이름 | bits | 설명 |
| Input | Multiplier | 64 | 승수 |
| Multiplicand | 64 | 피승수 |
| Clk | 1 | Clock |
| Reset\_n | 1 | Active low reset |
| Op\_start | 1 | Start signal |
| Op\_clear | 1 | Clear signal |
| Output | Op\_done | 1 | Done signal (when multiplication is over) |
| Result | 128 | Result of multiplication |
| Reg | x | 64 | Same with multiplier (circular shift) |
| x1 | 1 | X’s Previous LSB |
| u | 64 | Add/sub할 때의 피연산자(Initialize 0) |
| v | 64 | shift할 때 u의 범위를 벗어난 값이 MSB로 들어옴.(Initialize 0) |
| count | 7 | X에 대해 Circular shift를 몇 번했는가 |
| state | 2 | State를 나타내는 값 |
| add | 64 | U와 multiplicand를 add한 값 |
| sub | 64 | U와 multiplicand를 sub한 값 |
| next\_x | 64 | Next x |
| next\_x1 | 1 | Next X1 |
| next\_u | 64 | Next U |
| next\_v | 64 | Next V |
| next\_state | 2 | Next State |
| next\_count | 7 | Next count |

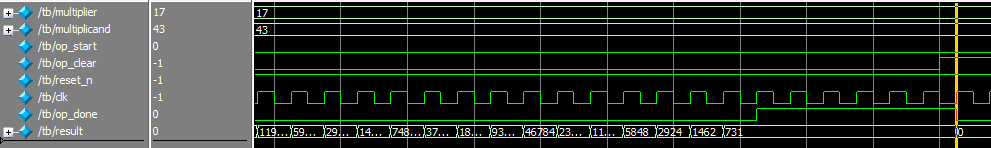
- always문을 사용하여 reset\_n에 따라서 state를 천이하는 부분(reset\_n이 0이면 state를 IDLE로 유지, 그 이외에는 next\_state값을 state로 넘겨줌), clk와 reset\_n에 따라서 값을 초기화 하거나, next 변수들을 각각에 해당하는 변수들에 넘겨주는 부분(reset\_n이 0이면 reg 변수들을 모두 0으로 초기화, 그 외의 경우에는 next\_변수들을 각각의 변수들에 넘겨줌), 현재 state와 count, op\_start, op\_clear signal에 따라서 next\_state를 결정해주는 부분(상태천이도와 동일하게 동작), 실질적으로 x, x1(x-1), u, v를 통해서 booth multiplication을 진행하는 부분, result 출력을 위해서 현재 state가 done이면 결과를 출력하고 op\_done을 1로 출력, 그 이외에는 둘 다 0으로 출력하는 부분으로 나누어 구현하였습니다.

1. 설계 검증 및 실험 결과
   1. 시뮬레이션 결과

- 초기 두 입력 값을 주고 연산 중간에 op\_clear를 주어 연산이 중지되는지를 확인하였고 그 이후 값을 주고 연산 결과가 출력된 후 op\_clear를 주기 전까지 그 값을 유지하다가 op\_clear를 주면 값이 바로 초기화 되는지를 보았습니다.

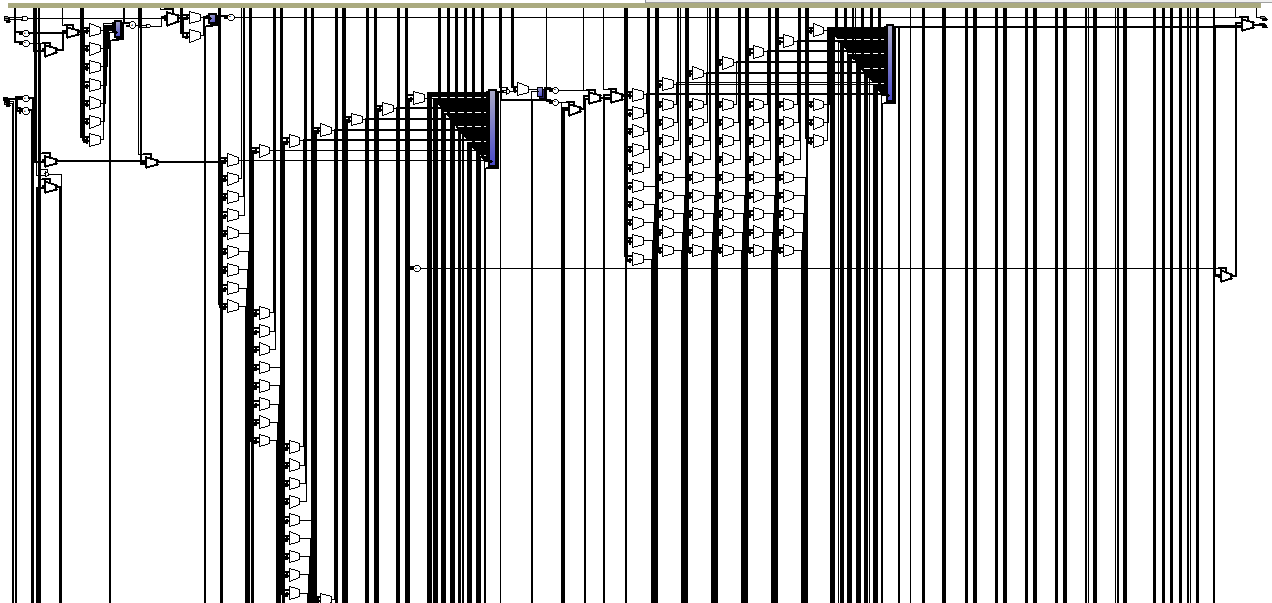


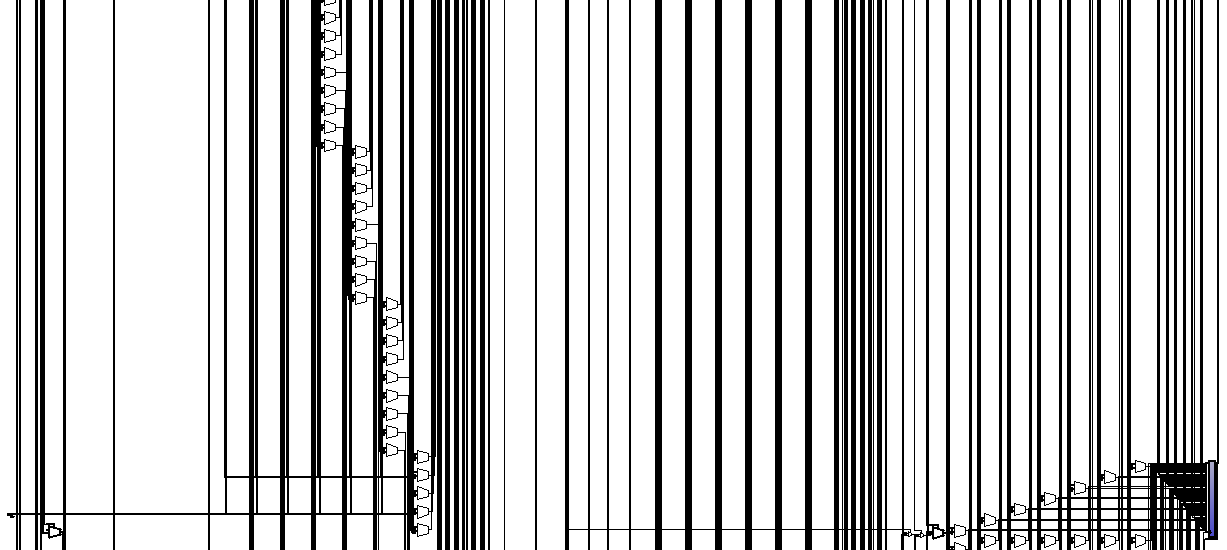
-위의 사진을 확인하면 op\_clear가 1이되고 바로 다음 rising edge에서 결과값을 0으로 초기화되는 것을 확인할 수 있었습니다.



-위의 사진을 보면 op\_clear가 1이 될 때까지 연산결과 값을 유지하고 있다가 op\_clear가 1이 되면서 다음 rising clk에서 초기화되는 것을 확인할 수 있었습니다.

* 1. 합성(synthesis) 결과





- 왼쪽 위부터 (input) op\_start, op\_clear, multiplicand[63:0], clk, reset\_n, multiplier[63:0]입니다.

- 오른쪽 위부터 op\_done, result[127:0]입니다.

- module instance 없이 구현하다보니 이렇게 나왔습니다…..죄송합니다…..

1. 고찰 및 결론
   1. 고찰 및 결론

- 제가 구현한 곱셈기는 booth’s multiplication으로 radix-2 multiplication입니다. Binary multiplication는 승수의 모든 자리 값의 bit를 확인하고 그에 대한 피승수와의 곱들의 합을 통해 연산 결과를 도출하게 됩니다. 반면에 booth’s multiplication을 사용하게 되면 승수의 bit를 더 간결하게 만들어((ex) 011111 -> 100000 - 1) add/sub가 아닌 shift만 하게 되는 경우를 만들어 주기 때문에 이러한 부분에 있어 연산 처리 속도의 감소가 있습니다.

이번 실험을 구현하면서 가장 난감했던 부분은 always 구문 안에서 case를 사용하는데, 이때 사용하는 next 인자들에 대해서 latch가 발생하면서 warning이 418개가 떴던 것이었습니다. ‘와 ….. 이러면 -318점인가? ‘라는 생각이 머릿속을 스치면서 구글링을 시작했습니다. 와…..역시 공대생이 졸업하는데 기여도의 80%는 구글입니다. 저 warning은 초기화를 해주지 않은 것에 대해 latch를 만드는 것이며 always begin과 case 사이에서 초기화를 해주면 된다고 하여 그리 해결을 했으나, 이로 인해 발생한 문제는 state가 done로 넘어가고 result를 출력했으나, 그 다음 클럭에 자동으로 값이 0으로 바뀌는 것이었습니다. 이는 초기화로 인해 next 변수들이 0으로 바뀌어 값이 바뀌는 것이었기 때문에 이를 해결하기 위해서 출력에 관한 always문을 추가적으로 만들어 이에 대해 현재 state가 done이라면 next인자에 각각에 해당하는 기본 인자들을 넣어주도록 함으로써 그 값을 지속적으로 유지하게끔 만들었습니다. 물론 IDLE에서는 그에 해당하는 0을 출력합니다.

그 이외에도 사소한 문제들이 몇 개 있었지만 원만하게 해결하였습니다. 다만 문제는 프로젝트라는 생각이 머리를 떠나지 않습니다…….

1. 참고문헌

Wikipedia

Google.com