디지털 논리회로2

Term project 결과 보고서

제출일자: 2018년 12월 02일 (일)

학 과: 컴퓨터정보공학부

담당교수: 이준환 교수님

학 번: 2017202087

성 명: 홍 세 정

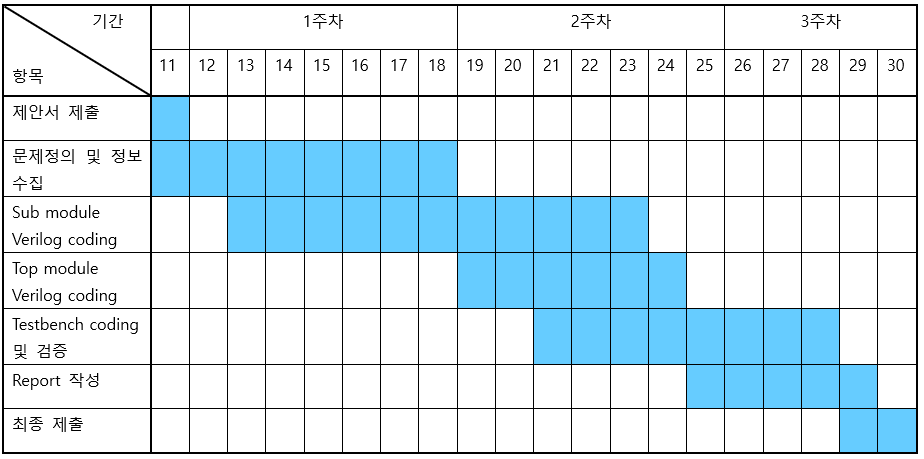
1. Introduction
   1. 제목

Multiplier, Adder, FIFO(first in first out), RF(register file)를 이용하여 Matrix를 설계하고, bus를 통해 Memory와 연결하여 검증.

* 1. 목적

Multiplier, Adder, FIFO, RF를 이용하여 Matrix를 설계하여 연속적인 연산들을 진행할 수 있다. 또한 testbench를 master로 하여 입력 값들을 저장하고 연산을 출력할 수 있다.

Matrix, Memory 그리고 Bus를 이용하여 Top module을 구현할 수 있다.

* 1. 일정
  2. 소개

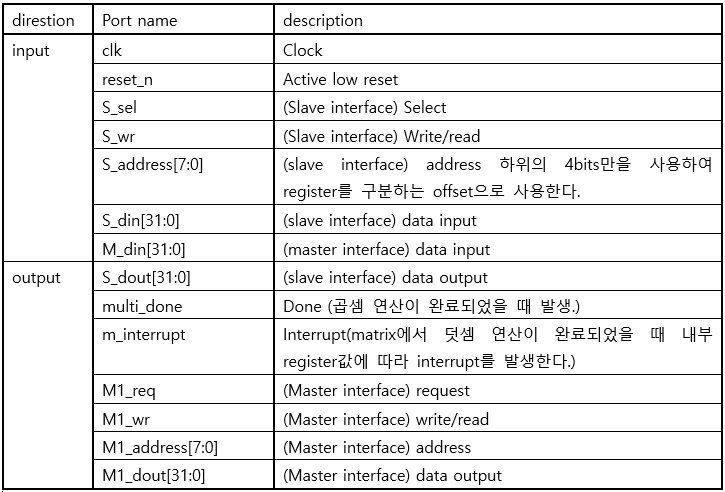
전체 시스템의 구성 요소는 크게 4가지로 matrix, bus, memory, testbench이다. Testbench로부터 memory#0과 memory#1에 2행 2열의 행렬 값을 serial하게 저장하고 testbench에서 matrix에게 memory에 저장된 값을 보내준다. Testbnech가 master가 되고, matrix는 전송받은 값을 내부에 있는 A Matrix, B Matirx fifo에 저장한다. Fifo에 serial하게 저장되어 있는 data들을 pop하여 mutiplier에서 A의 data와 B의 data들을 곱셈 연산을 수행하게 된다. 곱셈연산이 끝나게 되면 adder fifo에서 곱셈연산을 한 결과 값들을 RF에 저장한다. Adder에서는 fifo에 저장되어있는 data들을 순서대로 덧셈 연산을 하여 결과를 출력한다. RF에 저장된 값을 MEMORY#2의 첫 번째 register로 serial하게 전송하고 완료되면, test bench가 다시 master할 수 있다.



BUS와 MATRIX에서 slave와 master의 관계를 잘 파악하고 read/write할 수 있다. BUS는 grant에 따라 master를 결정해주며, grant에 따른 master만 bus를 사용할 수 있다. Master는 bus를 통해 data를 transfer하고자 할 때, 자신에게 해당하는 request signal을 1로 한 이후에, 그에 대한 확인으로 grant signal을 받은 후 data transfer를 올바르게 할 수 있다.

RAM는 address에 기반하여 데이터를 저장하는 hardware이다. Address의 bandwidth는 5bit이고, data의 bandwidth는 32bit이다. RAM의 내부에 데이터를 address에 기반하여 저장한다. 사용자가 자유롭게 내용을 읽고 쓰고 지울 수 있는 기억장치이다. Random Access Memory로 어디든 똑같은 시간으로 접근이 가능하다는 의미로서, 메모리의 주소만 알고 그곳을 지정하면 별다른 지연없이 바로 접근할 수 있다. RAM에 기억된 내용은 전원이 끊기면 지워지는 휘발성 기억 장치이다.

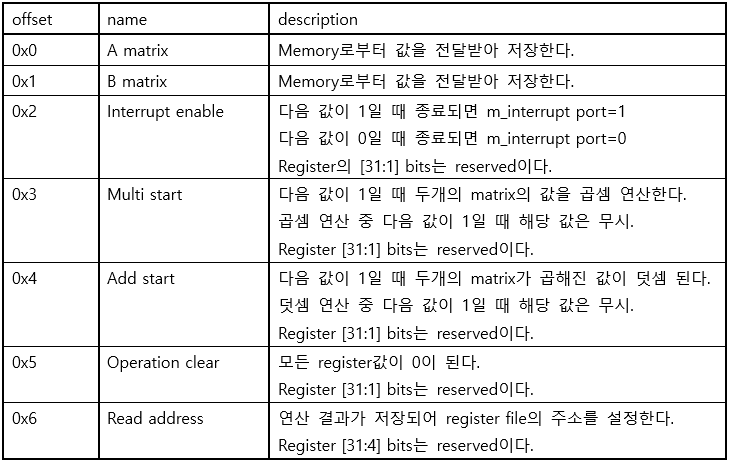
1. Project specification
   1. MATRIX





FIFO, Multiplier, Adder, Register File을 이용하여 Matrix를 구현할 수 있다.

행렬을 pop하는 fifo 두 개, 곱셈연산을 하는 multiplier, 곱셈연산을 완료한 결과를 pop하는 fifo, 덧셈 연산을 하는 adder, 연산을 저장하는 register file을 연결하여 연산할 수 있다. Matrix가 slave일 때는 offset마다 기능을 구현하여 각 fifo, multiplier, adder의 기능을 실행시킬 수 있다. Matirx가 master일 때는 bus에 명령을 주어 slave가 연산을 할 수 있도록 한다. Slave를 통해 연산을 실행할 수 있고, master를 통해 연산 결과를 출력한다고 볼 수 있다.

다음 offset을 보고 register에 저장을 할 수 있다.

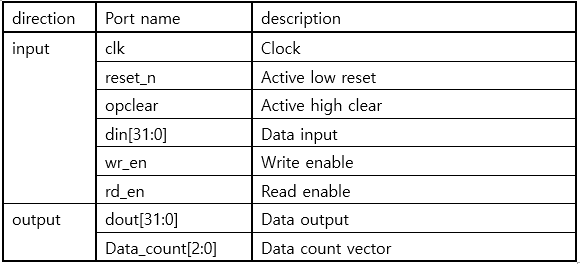
S\_address에서 하위 4비트만을 사용하여 register를 구분하는 offset으로 사용하여 구분하여 연산을 시작하거나 끝내거나 지정을 지정해 줄 수 있다.

Multi\_we : multi연산을 할 때, fifo에서 값을 읽어 주기 위해 쓰인다.

Multi\_start : multi 연산은 op\_start가 1일때만 수행한다. 따라서 op\_start를 조건에 따라서 읽어줄 수 있다.

Adder\_start : adder 연산도 마찬가지로 op\_start가 1일때만 수행한다. 따라서 op\_start를 조건에 따라서 읽어줄 수 있다.

* 1. FIFO



나중에 집어넣은 데이터가 먼저 나오는 스택과 반대되는 개념이다. 선입선출의 자료 구조 데이터가 들어오는 위치는 가장 뒤에 있고, 데이터 나가는 위치는 가장 앞에 있어서, 먼저 들어오는 데이터가 먼저 나가고 나중에 들어온 데이터는 가장 나중에 나간다.

여기서 queue의 구조를 이용하여 module FIFO를 구현한다.

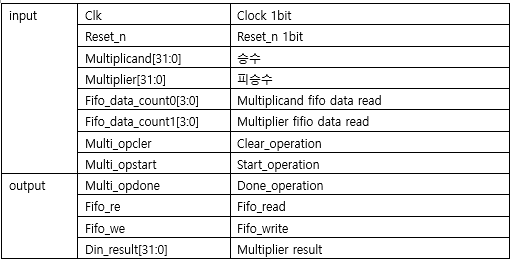
project에서 구현한 fifo는 앞서 실습에서 구현했던 fifo와 거의 비슷하게 구현할 수 있었다. Ouput인 Handshake(wr\_ack, wr\_err, rd\_ack, rd\_err)은 이번 project에서 사용하지 않으므로 삭제할 수 있다.

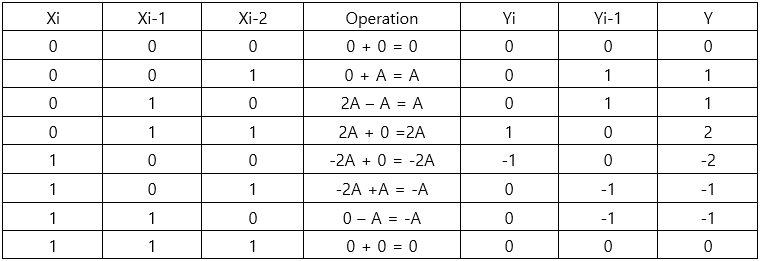
Multiplier fifo

multiplier에서 필요한 fifo는 총 8개의 data가 read/write된다. 곱셈연산에서는 multiplier, multiplicand 두개의 수가 필요하다. 두개의 fifo에서 각각 수를 받아와서 연산하게 된다.

Adder fifo

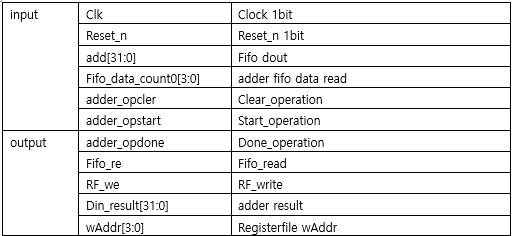
총 8개의 data가 read/write 된다. Multiplier와 다르게 한 개의 fifo에서 data를 두개 받아와서 연산을 하게 된다.

* 1. MULTIPLIER

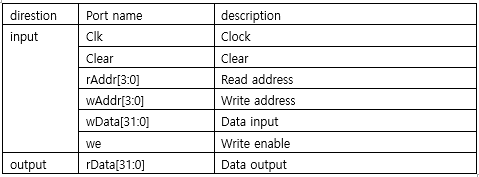


이전에 실습했던 multiplier에서는 radix-2 booth multiplier로 구현할 수 있었다. 이번 프로젝트에서는 testbench로 검증하였을 때 너무 많은 clk가 사용되어 성능이 좋지 않아 radix-4 booth multiplier로 구현할 수 있었다.

* 1. ADDER



Fifo에서 data를 받아와서 empty가 될 때까지 연산을 한다. Adder\_fifo에 들어있는 data는 총 8개의 data를 가지고 있다. fifo에서 두개의 data를 받아와서 덧셈 연산을 한 후 result를 출력하게 된다. 따라서 adder의 최종적인 result는 4개가 나온다. Fifo\_re는 fifo에서 값을 read하기 위해 출력되는 값이다. 그러므로 연산하기 전에 data의 값을 하나씩 받아올 수 있다. RF\_we는 덧셈을 연산이 완료되었을 때 1로 바뀌게 되고, adder\_opdone은 fifo에서 더 이상 받아올 data가 없을 때 1로 출력된다.

* 1. RF(register file)

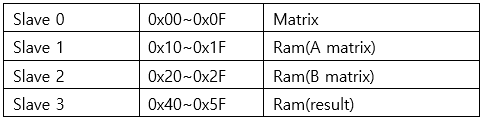
이전 실습했던 register file과는 다르게 reset\_n이 사용되지 않고 clear로 사용되게 된다. Reset\_n이 0일 때 모든 값이 초기화 되는 이전 방식과는 다르게 clear이 1이 되면 모든 값이 초기화 된다. Write\_operation, register32\_8, read\_operation 3개의 module을 이용하여 Register file을 구현할 수 있다.

register은 write enable에 의해 결정되고, read register은 rAddr로 선택된 register의 값이 출력된다. Decoder를 통해 address를 해석하여 해당 register enable을 지정한다. Mux를 통해서 register 중 한 개를 선택하여 출력할 수 있게 된다.

* 1. SLAVE

Slave는 bus에서 memory를 받아 slave가 되기 때문에 matirx안에서도 필요하다. 명령을 받고 그 명령에 따라 동작을 수행하도록 한다. S\_address를 통해서 offset을 정하고 register를 선언하여 각 역할을 수행할 수 있도록 할 수 있다.

* 1. BUS



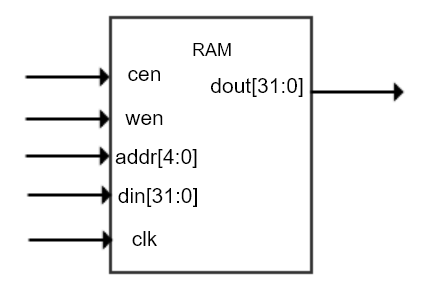
BUS는 grant에 따라 master를 결정해주며, grant에 따른 master만 bus를 사용할 수 있다.

Master는 bus를 통해 data를 transfer하고자 할 때, 자신에게 해당하는 request signal을 1로 한 이후에, 그에 대한 확인으로 grant signal을 받은 후 data transfer를 올바르게 할 수 있다. Bus는 memory와 testbench와 matirx를 연결해준다. 각 연결해준 기능들은 slave와 master의 관계를 가지고 있어서 그 관계를 잘 파악하고, 연결해줄 수 있어야한다.

|  |  |  |
| --- | --- | --- |
| Direction | Port name | Description |
| Input | clk | Clock |
| reset\_n | Active low reset |
| M0\_req | Master 0 request |
| M0\_wr | Master 0 write/read |
| M0\_address[7:0] | Master 0 address |
| M0\_dout[31:0] | Master 0 data output |
| M1\_req | Master 1 request |
| M1\_wr | Master 1 write/read |
| M1\_address[7:0] | Master 1 address |
| M1\_dout[31:0] | Master 1 data out |
| S0\_dout[31:0] | Slave 0 data out |
| S1\_dout[31:0] | Slave 1 data out |
| S2\_dout[31:0] | Slave 2 data out |
| S3\_dout[31:0] | Slave 3 data out |
| Output | M0\_grant | Master 0 grant |
| M1\_grant | Master 1 grant |
| M\_din[31:0] | Master data input |
| S0\_sel | Slave 0 select |
| S1\_sel | Slave 1 select |
| S2\_sel | Slave 2 select |
| S3\_sel | Slave 3 select |
| S\_din[31:0] | Slave data input |
| S\_address[7:0] | Slave address |
| S\_wr | Slave write/read |
| S\_din[31:0] | Slave data input |

* 1. MEMORY

|  |  |  |
| --- | --- | --- |
| Direction | Port name | Description |
| Input | clk | Clock |
| cen | Chip enable signal |
| wen | Write enable signal |
| addr[4:0] | Address |
| din[31:0] | Data in |
| Output | dout[31:0] | Data out |

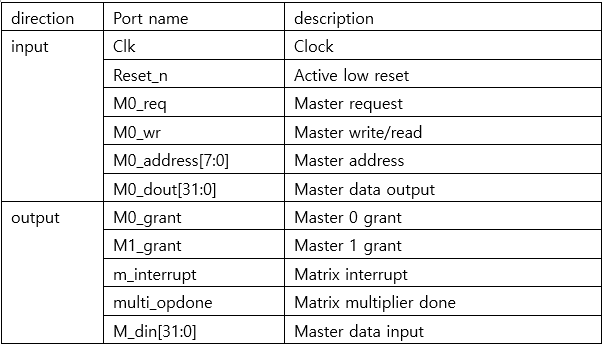


Address의 bandwidth는 5bit이고, data의 bandwidth는 32bit이다. RAM의 내부에 32개의 데이터를 address에 기반하여 저장한다는 특징을 가지고 있다.

MEMORY#1은 2행 2열의 행렬이 serial하게 저장되어 있다.

MEMORY#2은 2행 2열의 행렬이 serial하게 저장되어 있다.

MEMORY#3은 MEMORY#1과 MEMORY#2에서 저장된 행렬을 곱셈한 결과가 저장되어 있다.

* 1. TOP

1. Design details
   1. MATRIX

곱셈연산 할 때 multiplier, multiplicand를 pop하기 위한 fifo module 2개, 곱셈 연산을 위한 multiplier, multiplier의 result, 덧셈 연산을 하기 위한 fifo 1개, 덧셈 연산을 위한 adder, register file, slave, master 총 8개의 module을 instance하여 matirx의 module을 구현할 수 있다.

* 1. FIFO

이전에 실습한 fifo module과 거의 비슷하게 구현할 수 있다.

Handshake(wr\_ack, wr\_err, rd\_ack, rd\_err)은 matrix 구현에 필요없으므로 삭제한다.



다음과 같이 6개의 state로 fsm을 설계할 수 있다.

Wr\_en이 1일 때 write를 하고 rd\_ed 이 1일 때 read를 할 수 있다.

Fifo에서는 총 4개의 module로 구현할 수 있다.

Fifo\_ns, fifo\_cal, register file module을 instance하여 top fifo를 구현할 수 있다.

Write는 slave에서 알맞게 input을 넣어줄 수 있고 read는 연산하는 과정에서 연산을 완료할 때 마다 알맞게 input값을 줄 수 있다.

Fifo\_ns : 외부로 wr\_en과 rd\_en을 받고 내부에서 현재 stat와 data\_count를 받아 다음 state를 출력. next state를 출력하는 module을 구현한다. Wr\_en, rd\_en, state, data\_count의 값을 받아 next state를 정해준다.

FIFO\_cal : state, data\_count, head, tail을 받아 다음 stat에 대한 head, tail값(register의 주소값)과 현재 data의 개수를 계산

각 state에서 head, tail data\_count값을 변경시켜준다.

INIT – 초기값이므로 값이 바뀌지 s않는다.

NO\_OP – INIT state와 마찬가지로 값이 바뀌지 않는다.

WRITE – ­­next\_head=head tail+1, data\_count+1 이다. We =1 로 값을 write한다.

READ – head+1, tail, data\_count+1 이다. Re = 1로 값을 read 한다.

Register file: 사용자로부터 받은 입력을 저장하거나 알맞은 register의 값을 출력

이전 실습에서 진행한 register file을 포함하고 이용할 수 있다.

* 1. MULTIPLIER

곱셈연산을 하는 module이다. fifo에서 multiplicand와 multiplier를 받아와서 연산을 할 수 있어야한다. 이전 실습했던 multiplier와는 다르게 3개의 state가 아닌 fifo에서 값을 받아들여야 하기 때문에 state를 5개로 늘릴 수 있다. 이전의 실습에서는 radix\_2 booth multiplier로 32번의 data\_count가 필요했지만 이번 프로젝트에서는 radix4\_booth로 16번의 data\_count가 필요하다.



IDLE : 초기 상태 곱셈 연산을 하기 위한 준비 상태이다.

POP : fifo에서 data들을 pop하여 rd\_en의 출력이 1로 바뀌면서 multiplicand fifo와 multiplier fifo에서 각각 data값들을 받아들일 수 있다.

EXEC : 받아들인 두개의 data들을 연산한다. 구현한 multiplier는 radix-4이다. 따라서 16번의 clk가 rising edge를 완료하면 result값이 나오게 된다. Data\_count가 16이 될 때까지 EXEC를 반복한다.

WRITE : 연산의 결과를 출력한다. EXEC에서의 data\_count가 16이 되면, result값이 출력되고 값을 write하는 adder fifo에서 write가 1이 된다.

DONE : 8개의 data가 모두 연산이 완료되면 multi\_done이 1이 되어 모든 연산이 완료되었다는 것을 확인할 수 있다.

Multiplier cal, multiplie\_ns, multiplier out, multiplier\_top으로 multiplier를 구성할 수 있다.

* 1. ADDER

덧셈 연산하는 module이다. fifo에서 받아오는 값들을 덧셈 연산 완료 후 출력할 수 있다.

Multiplier와 같은 state를 가지게 된다. 덧셈 연산인 cla를 instance하여 연산을 할 수 있다. Cla를 fsm으로 구현한다. Multiplier와는 다르게 adder은 한 개의 fifo에서 두개의 data를 받아와 연산을 한다. 그래서 pop에서 읽어오는 read\_count가 2일 때 만 연산을 시작한다.



IDLE : 초기 상태이다 연산을 하기 위한 준비상태. Clear가 1이면 어느 state에서든 IDLE로 돌아간다.

POP : 연산을 하기 위해 fifo에서 data값들을 받아온다. adder에서 data값을 순차적으로 두개를 받아와야 하므로 하나의 data를 받아올 때마다 read\_count를 +1 하고, data 값들을 저장할 수 있다. 두개를 읽어 read\_count가 2이면 EXEC state로 이동하여 연산을 한다.

EXEC : 저장된 data값 두개를 덧셈하여 result에 저장할 수 있다.

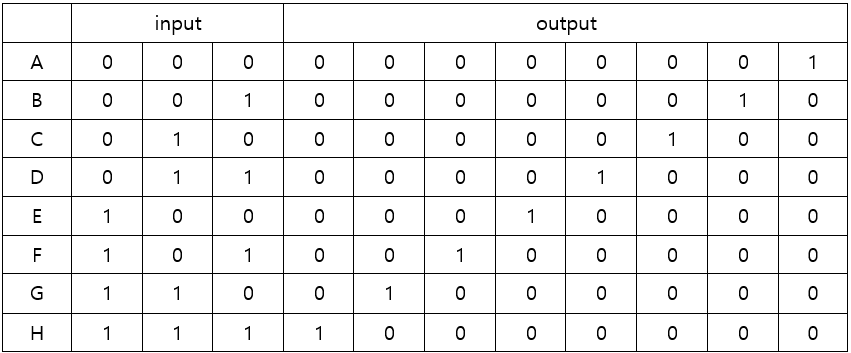
WRITE :

DONE : 8개의 data가 모두 연산이 완료되면 adder\_done이 1이 되어 모든 연산이 완료되었다는 것을 확인할 수 있다.

* 1. RF(register file)

<register file>

Write\_operation, register32\_8, read\_operation 3개의 module을 이용하여 Register file을 구현할 수 있었다.

 register은 write enable에 의해 결정되고, read register은 rAddr로 선택된 register의 값이 출력된다. Decoder를 통해 address를 해석하여 해당 register enable을 지정한다. Mux를 통해서 register 중 한 개를 선택하여 출력할 수 있게 된다.

* 1. MEMORY

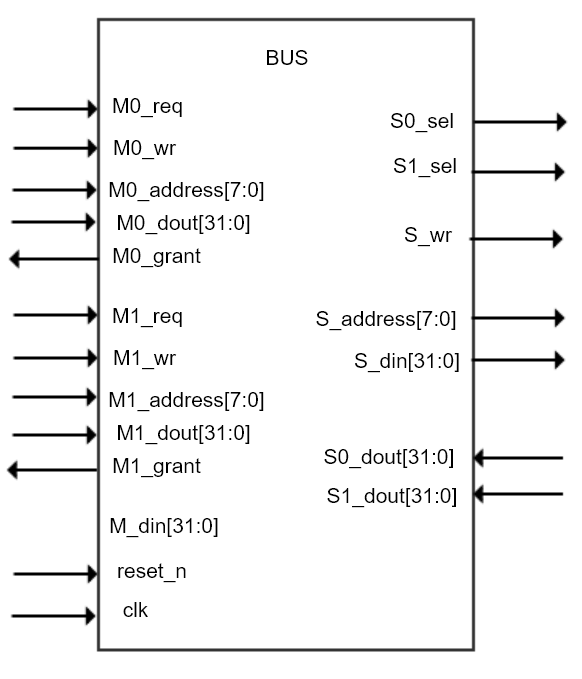
이전 구현하였던 RAM과 똑 같은 방식으로 구현할 수 있었다.

기억장치로써 initial 구만 안에 반복문을 사용하여 데이터를 초기화 할 수 있다.

Cen과 wen이 모두 1이면 address가 가리키는 memory에 din을 write한다. 이때는 dout은 0을 출력한다.

Cen이 1, wen이 0이면 address가 가리키는 memory의 값을 dout에 write한다.

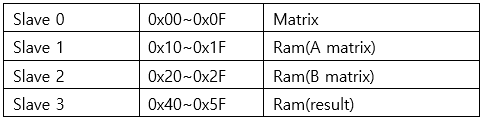
* 1. BUS



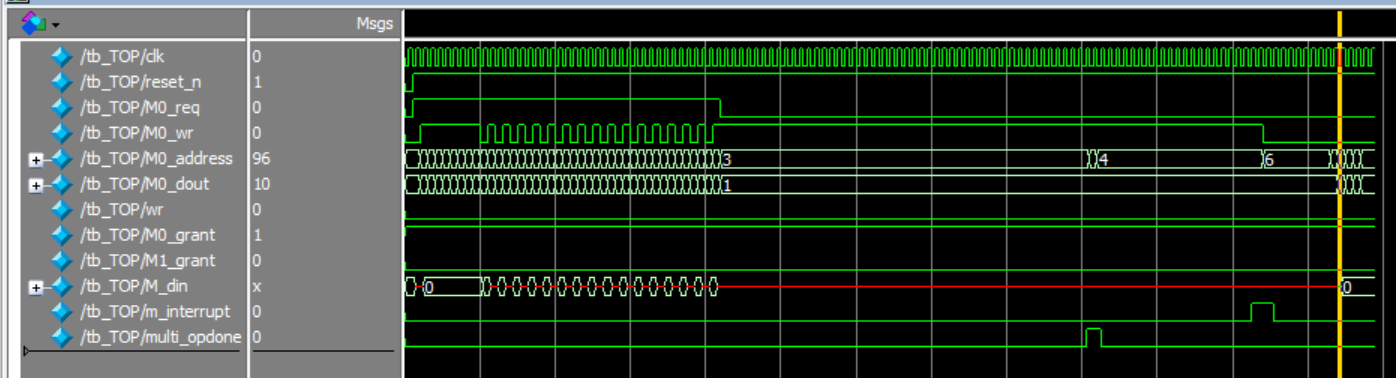
BUS는 grant에 따라 master를 결정해주며, grant에 따른 master만 bus를 사용할 수 있다.

Master는 bus를 통해 data를 transfer하고자 할 때, 자신에게 해당하는 request signal을 1로 한 이후에, 그에 대한 확인으로 grant signal을 받은 후 data transfer를 올바르게 할 수 있다.

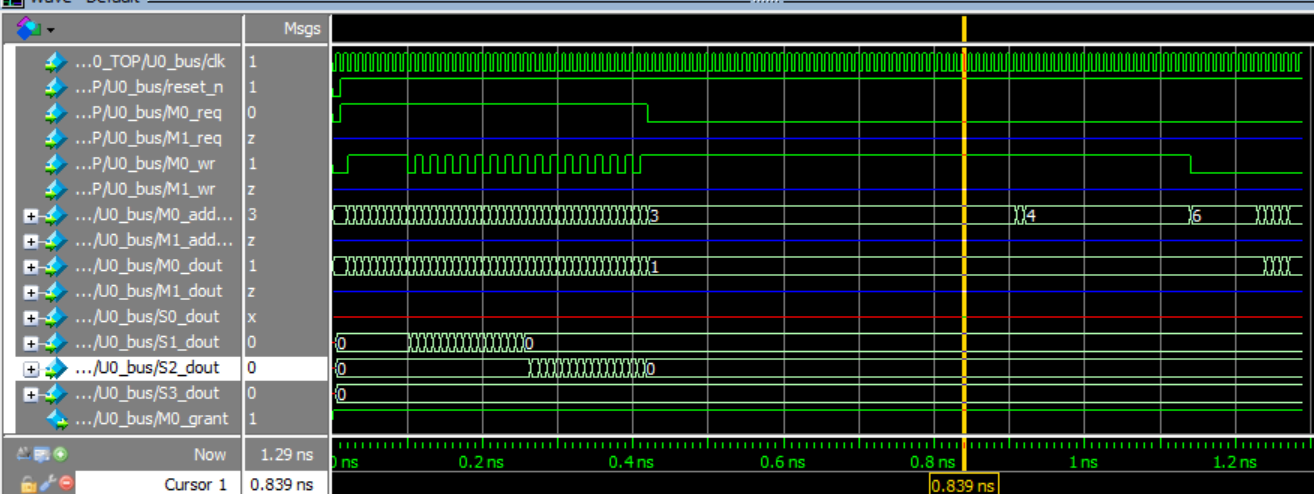
Master가 grant signal을 받은 후 request signal이 1인 동안에는 bus의 소유권을 빼앗기지 않고 data transfer를 계속 할 수 있다.

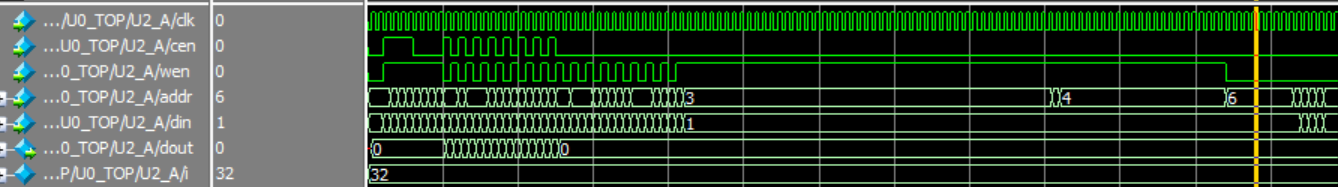


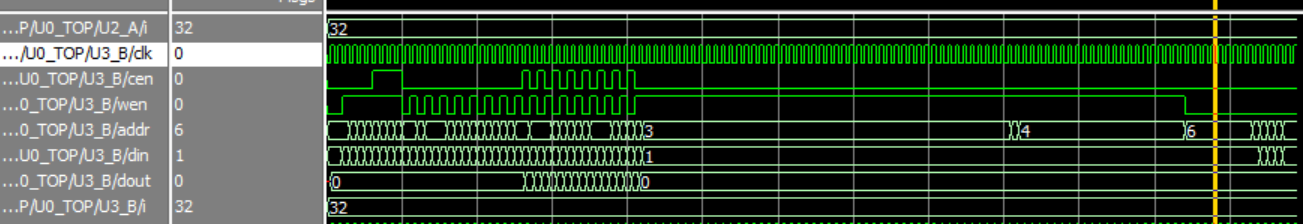


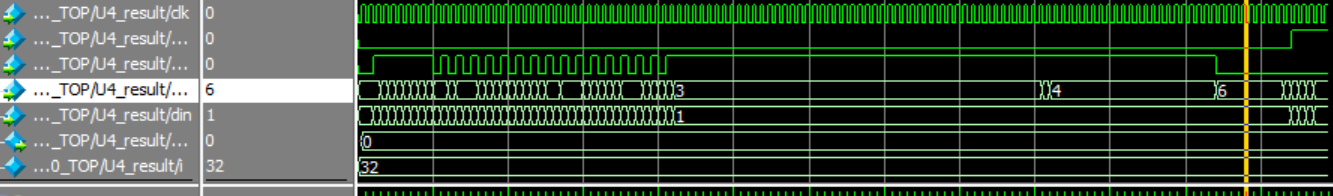
1. Design verification strategy and results
   1. TOP

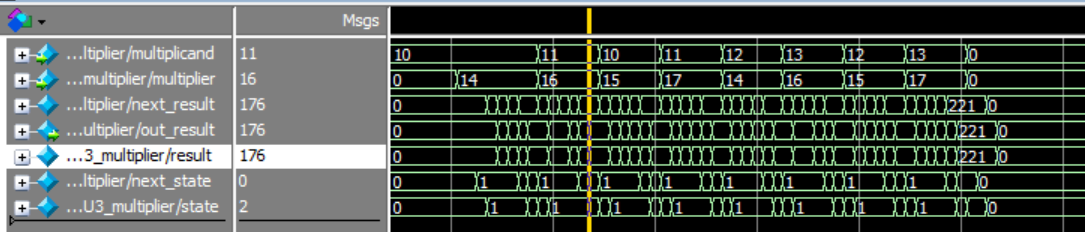
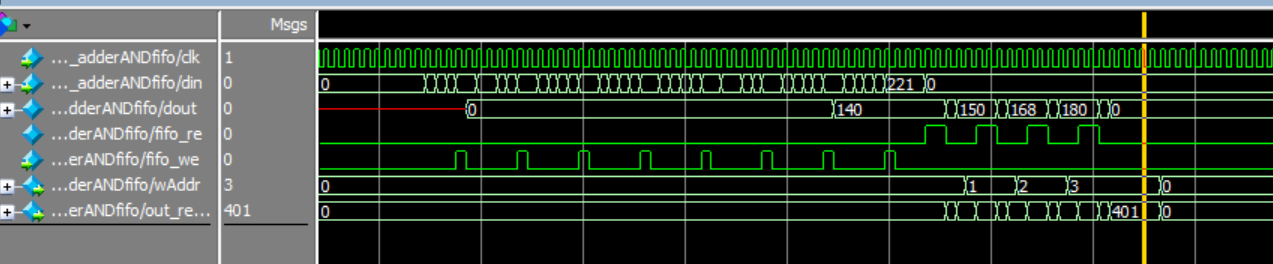
Master를 완성하지 못하여서 제대로 된 결과가 출력되지 않는다.

* 1. BUS
  2. MEMORY

MEMORY#1

MEMORY#2

MEMORY#3

* 1. MUTIPLIER
  2. Adder

1. Conclusion

여태 했던 실습들을 모두 응용하여 구현할 수 있는 프로젝트였다. Multiplier, Adder, FIFO(first in first out), RF(register file)를 이용하여 Matrix를 설계할 수 있다.

프로젝트를 진행하는 동안 1주일 이상을 프로젝트를 어떻게 해야 할 지 감을 못 잡고 제안서와 pdf를 계속 읽으면서 이해하도록 노력했다. 여러 사람들의 도움을 받으면서 프로젝트를 이해하고 구현할 수 있었다. Matrix module을 먼저 구현할 수 있었다. Fifo와 multiplier와 연결한 module을 검증하고, fifo와 adder를 연결한 module을 검증하고 multiplier, adder, register file을 연결한 module을 검증하면서 하나씩 차근차근 matrix를 구현할 수 있었다.

여태 실습하였던 fifo, multiplier, adder, Register file등을 가져와서 그냥 input port와 output port만 연결해주면 될 줄 알았는데 이 module도 matrix에 맞게 바꿔주어야 작동한다는 것을 알았다. 처음 multiplier를 구현할 때 예전에 구현했던 module에 fifo\_re와 fifo\_we의 output을 추가하여 구현하였지만, multiplier가 제대로 작동하지 않았다. 그래서 처음부터 다시 시작해서 state를 5개로 바꿔서 만들어 주고, fsm을 다시 작성하여 multiplier module을 완성할 수 있었다. 또한, 전에 만들었던 multiplier는 값을 유지 못하여서 이 후의 값들을 받아서 곱셈을 하였다. 그래서 fifo에서 받아온 값들을 유지하기 위해서 register를 하나 더 만들어 사용하면서 값들을 저장하고 그 저장한 값들을 받아와서 곱셈을 하였다.

Slave를 이해하는 부분이 제일 어려웠다. Register를 어떻게 사용하는지도 모르겠고, 구현자체를 어떻게 하는지 이해가 안돼서 여러 친구들과 선배들의 도움을 받아서 이해를 할 수 있었다. Slave와 master의 관계를 파악하여 출력할 수 있었다.

이번 프로젝트에서 master module을 구현하지 못하였다. 전체적인 흐름을 보면서 slave와 master를 이해할 수 있었지만 master까지 구현하기에는 시간이 부족하였다. 조금만 더 시간이 있었으면 master까지 구현할 수 있지 않았을까 하는 아쉬움이 많이 남았다. Project가 끝나고 master까지 구현하여 프로젝트를 완성하고 싶다.