Digital Logic Circuit Project

Matrix Multiplication

2015722025 정용훈

Abstract

I. Introduction

II. Project Specification

III. Design Details

IV. Design Verification Strategy and Results

V. Conclusion

VI. Reference

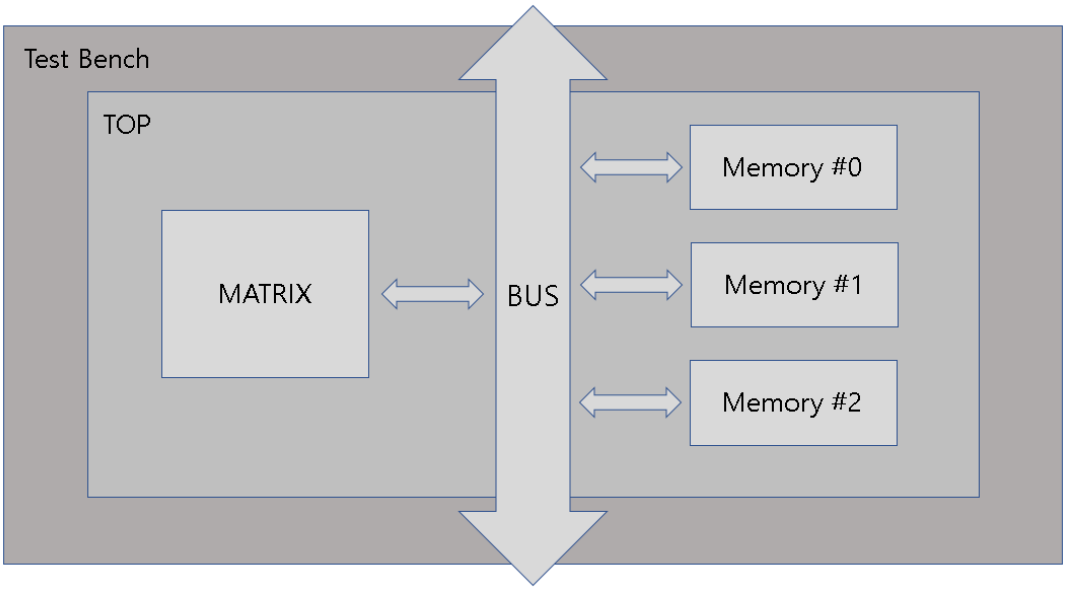
1. Introduction

Schedule

11	12	13	14	15	16	17
제안서 제출	코드 작성					
18	19	20	21	22	23	24
코드 작성	예비 검증 및 코드 수정					
25	26	27	28	29	30	12/1~2
보고서 개념 정리			본 검증 (결과 보고서 작성)		코드제출	결과 보고서 마무리 및 제출

Introduction of Project

이번 Digital logic circuit과목의 Project는 앞서 설계한 Multiplier, Adder, FIFO, Register File을 사용하여 해당 프로젝트에서 계산을 담당하는 Matrix module을 설계하여 2X2행렬을 계산할 수 있도록 하며, bus와 memory를 연결하여 검증하고 확인하는 것이 최종 목표이다. 미리 설계한 module은 Project에 맞게 약간씩 수정이 필요하며, 가장 중심이 되는 모듈은 Matrix module이다. Test bench와 핵심적인 module인 Matrix, Bus, Memory는 아래 그림과 같이 연결이 된다.

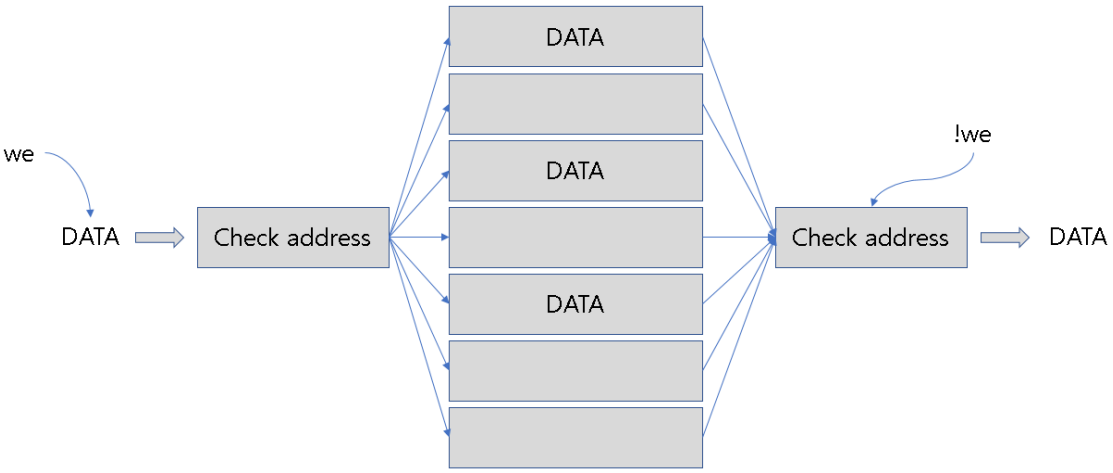


해당 TOP module의 대략적인 동작은 다음과 같다. Test bench를 통하여 Memory #0과 Memory #1에 값이 저장되면 Bus를 통하여 값이 Matrix로 이동 후 계산이 완료되면 다시 Bus를 통하여 Memory #2에 저장되는 방식이다. 각각 module의 설명은 다음과 같다. 해당 Project의 핵심이 되는 matrix는 Memory로부터 계산하게 될 값을 받으며 명령을 담당하는 주소가 값이 들어오게 되면 해당 명령을 통하여 동작하는 module이다. Bus는 Matrix와 Memory사이에서 명령과 데이터의 송수신을 맡아 데이터를 전달해주는 module이며 Memory는 실질적인 값을 저장하고 저장된 값을 사용할 수 있도록 도와주는 module이다. 저장된 값을 명령주소를 통하여 Matrix에 입력이 되면 Matrix에서는 세부적인 동작으로 곱셈을 하고 덧셈을 진행하며, 행렬의 결과 값들은 Matrix 내부의 Register File에 저장이 된다. Matrix는 slave, master 둘다 될 수 있는데 계산이 완료된 값을 Memory에 저장하기 위하여 master가 되어 값을 저장하게 된다. 모든 값이 저장이 되면 Matrix는 권한을 다시 Test bench에게 주며 slave가 된다.

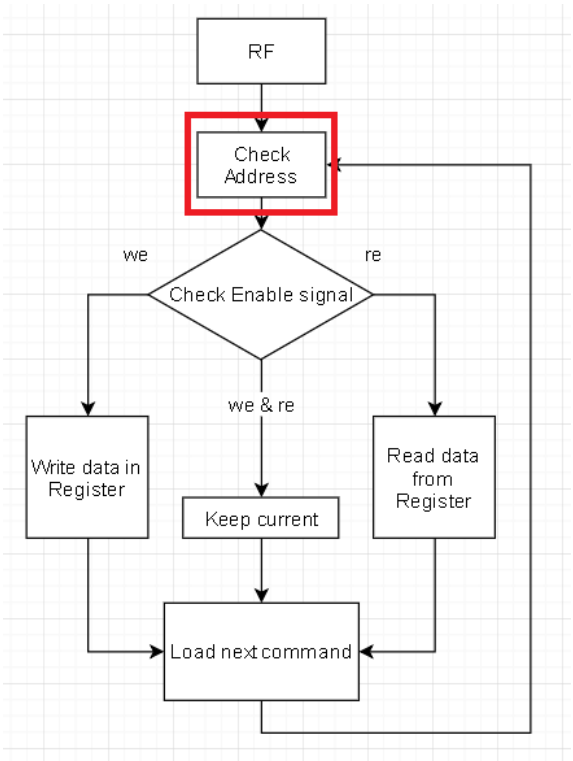
2. Project Specification

Register File

Register File(RF)는 값을 저장할 수 있는 register의 집합이다. FIFO와 같이 사용하지 않고 RF를 단독으로 사용하려면 주소 값을 test bench에서 설정해주어야 한다. RF는 데이터를 저장하고 읽는데 핵심적인 역할을 하는 module이라고 할 수 있다. 해당 module은 주소 값을 head와, tail로 나누어 순차적으로 업데이트하며 FIFO구조로 바꿀 수 있어 FIFO module에서의 중요한 역할이라고 할 수 있다.

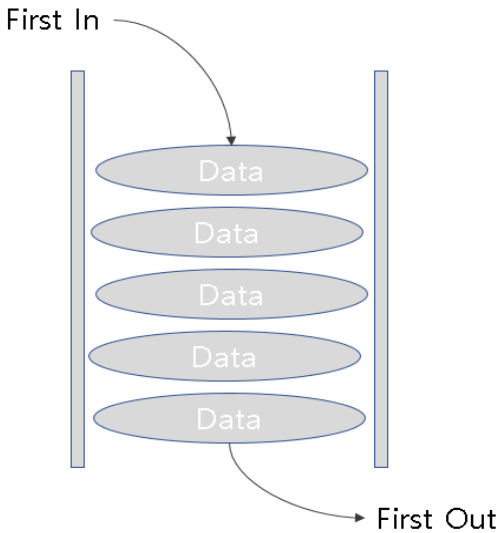


해당 그림은 RF자체의 동작만을 나타낸 그림이며 신호에 따라 불러오고, 저장할 주소의 위치를 찾아서 Data를 처리하는 동작을 하게 된다. 알고리즘 자체는 FIFO와 유사하며 아래와 같은 Flow chart로 나타낼 수 있다.

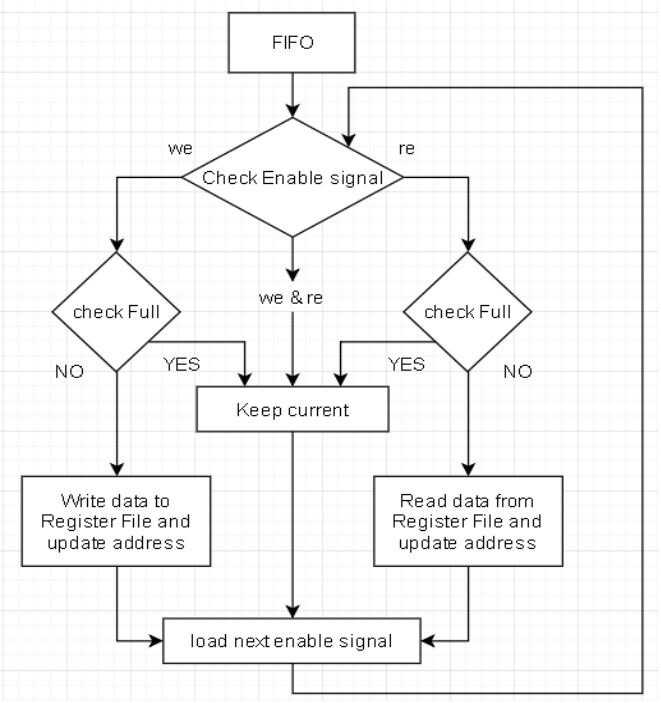


해당 그림은 RF의 Flow chart이며, FIFO와는 크게 다른 점이 없지만 빨간 네모부분을 보면 address를 check하는 logic이 있는 것을 확인할 수 있다. FIFO 동작을 할 수 있도록 address와 관련된 부분을 test bench가 아닌 자동적으로 Update할 수 있도록 설계하며 해당 모듈을 바탕으로 Multiplier와 Adder를 설계한다.

FIFO



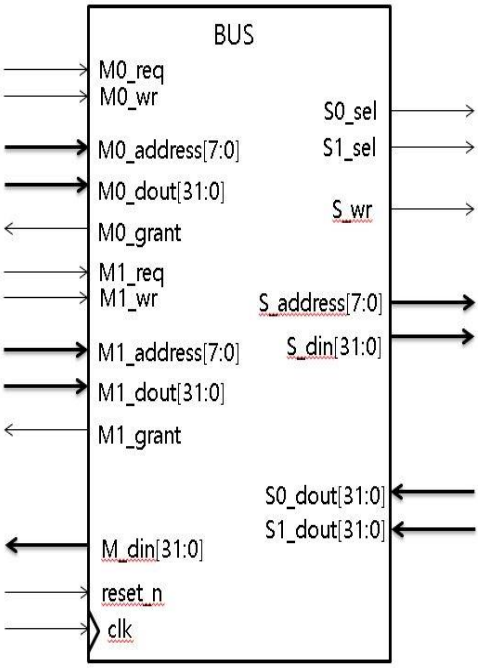
FIFO란 자료구조의 한 형태로 지정 공간에 어떤 자료가 가장 먼저 들어와 저장 되어있고 삭제 명령이나 일정 데이터를 읽는 명령을 받았다면 가장 먼저 들어왔던 데이터가 해당 명령의 대상이 되는 자료구조를 뜻한다. 예를 들면 가게의 입장을 위해 줄을 선다면 가장 먼저 줄을 선 사람부터 입장할 수 있는것과 같은 구조라고 생각하면 쉽다. 해당 FIFO의 개념은 다른 과목에서도 많이 언급되며, 우리가 가장 많이 다룬 자료구조 중 하나이다. FIFO가 이번 Project에서 응용이 되는데 Register File과 함께 응용이 된다. Register File은 데이터를 읽고 쓰는 기능을 하는 module이며 이는 FIFO와 응용을 한다면 Write enable신호를 받아 가장 먼저 쓰여 저장된 데이터는 나중에 데이터를 사용할 때 Read enable신호에 따라 가장 먼저 읽히게 되는 특징을 가지고 있다. 또한 FIFO의 특징으로 이미 읽힌 데이터는 다시 접근할 수 없다는 특징을 가지고 있다.



다음 그림은 FIFO동작의 전체적인 흐름을 나타낸 알고리즘이다. FIFO의 동작은 Enable signal을 통해 결정되며, we 신호가 켜지면 데이터를 저장하는 동작을 하며, re 신호가 켜지면 데이터를 읽는 동작을 한다. 두 신호가 함께 켜지면 동작을 하지 않는 특징을 가지고 있다. FIFO는 Matrix Module에서 곱셈과 덧셈의 결과를 저장하는데 쓰이는 중요한 역할을 담당하고 있다.

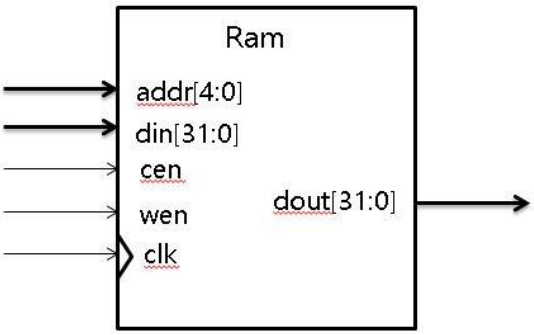
Bus

q



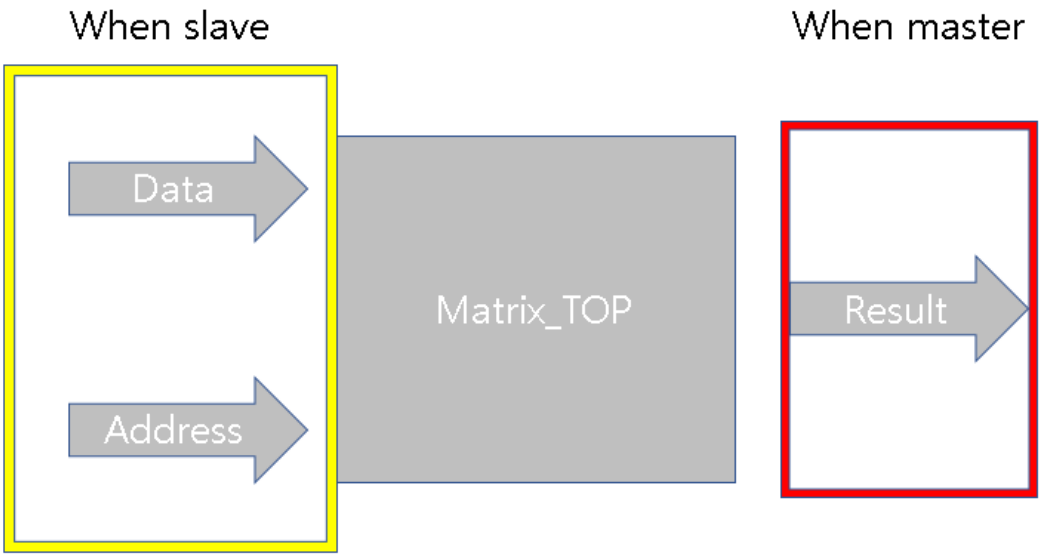
Bus는 컴퓨터 안의 부품들 간, 또는 컴퓨터 간에 데이터를 전송을 도와주는 역할을 하는 시스템이다. 데이터를 전송하는데 굳이 왜 Bus라는 모듈이 있어야 하는지 의문이 생길 수 있다. 이런 Bus는 연결을 좀 더 간소화하여 비용을 줄일 수 있으며 데이터의 전송을 훨씬 효율적으로 만들 수 있는 장점이 있다. 예를 들면 A의 정보를 1번에 주고 있는데, 시간이 지나 A의 정보를 2번에 주고 싶은 경우 Bus를 통해 Slave를 선택하여 데이터를 주는 대상자를 바꿀 수 있지만 만약 Bus가 개입되어 있지 않는다면 따로 연결을 해주어야 한다는 단점이 생긴다. 이렇게 버스는 이번 프로젝트에서도 Matrix와 Memory간의 데이터의 이동을 도와주는 역할을 담당하고 있다. 옆에 그림은 이전에 구현한 Bus의 Symbol이며 프로젝트에서 사용하게 될 Bus는 Slave를 4개 가지고 있으며 해당 그림은 Slave가 2개인 Bus이다. 해당 Bus의 알고리즘은 다음과 같이 나타낼 수 있다.

Memory

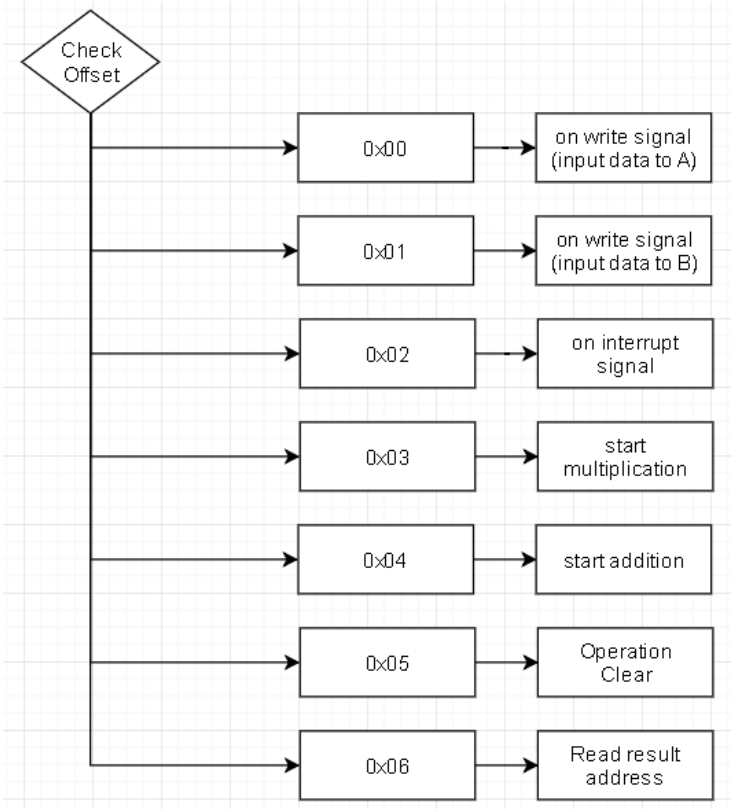


Memory는 쉽게 말하면 데이터를 저장하는 공간을 뜻한다. 데이터를 저장하는 장치로써 필요하면 정보를 읽을 수 있으며, 새로운 데이터를 저장하기도 한다. Memory의 종류로는 크게 RAM과 ROM으로 나누어지며 휘발성 메모리와 불 휘발성인 특징을 가지고 있다. Project에서 사용될 Memory는 앞서 설계한 Memory와 마찬가지로 실행되었을 때 모든 데이터 공간을 0으로 초기화 시켜주며, signal로는 wen, cen두가지를 받게 된다. Cen은 chip enable로써 해당 signal이 high를 유지해주어야 memory에 접근하여 데이터를 읽거나 쓸 수 있으며, wen은 Write enable로써 high를 유지하면 input으로 받은 주소에 data를 저장하는 명령을 실행하며 signal이 low이면 input으로 받는 주소의 데이터를 Read하는 명령을 수행한다.

Matrix



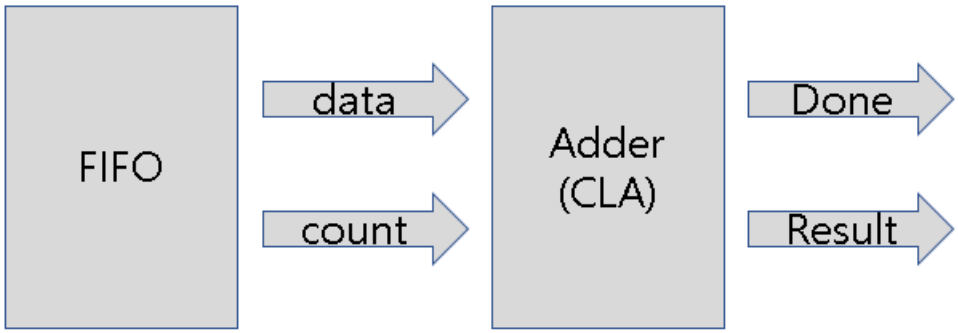
다음은 Matrix의 Symbol을 나타낸 것이다. Matrix는 Slave가 될 수 있고 Master가 될 수 있다. Slave의 경우 ram으로부터 bus를 통하여 데이터를 받고 알맞은 연산 명령을 받아 연산을 하여 최종적인 값을 RF에 저장하게 된다. 그 후 Master가 되면 Bus를 통하여 Result를 저장할 Ram에 최종적인 값을 serial하게 저장하게 된다. 다음은 Matrix가 offset을 통하여 연산하게 되는 과정을 그린 알고리즘이다.



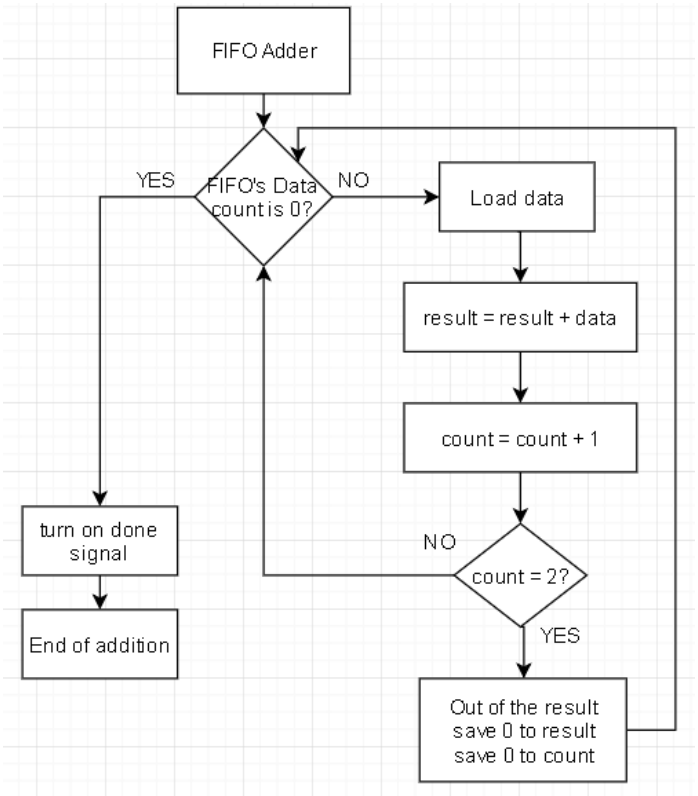
해당 알고리즘은 Slave가 명령을 받아 Matrix가 동작하는 연산을 추상적으로 만들어 놓은 것이다. 각각 연산의 자세한 사항은 다음 항목에서 계속 설명한다.

Adder (FIFO)

이번 프로젝트에서 행렬의 곱셈을 하기 위해서 필요한 연산으로는 곱셈 연산기와 덧셈 연산기가 필요하다. 해당 연산기들은 모두 FIFO와 연결되어 동작하며 이번 항목에서 설명할 adder의 간단한 연결을 확인하면 다음과 같이 나타낼 수 있다.



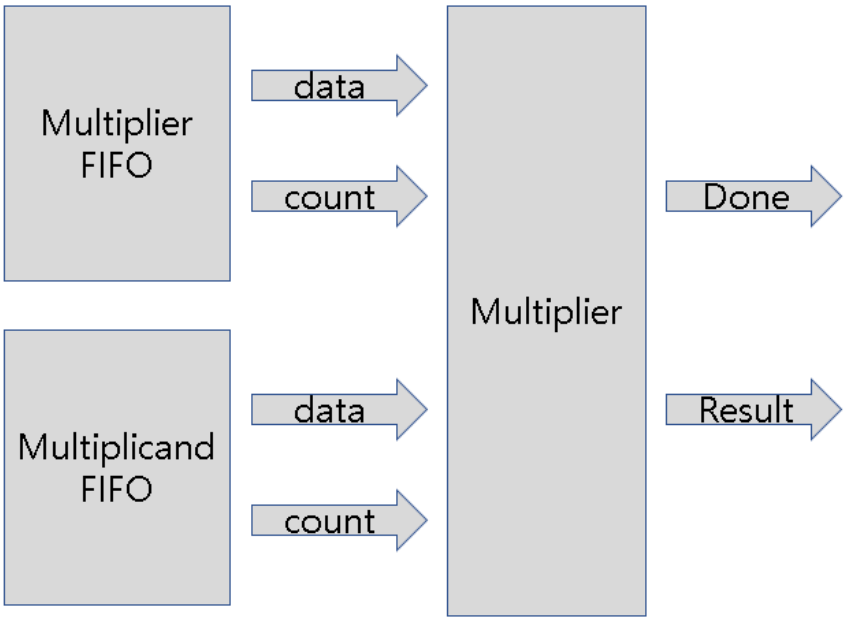
해당 Adder는 CLA를 기반으로 FSM으로 설계한 adder이다. 위에 제시된 그림과 같이 FIFO에서는 Adder에게 Clock마다 data의 값과 count를 전달하며, adder는 내부의 count와 FIFO의 count를 기반으로 next sate를 정하게 된다. 내부의 카운터로는 2가 count될때마다 Out state로 이동하며 덧셈의 결과를 표출하고 Out state가 되는 순간 FIFO의 data count를 check하여 count가 0 이면 Done state로 그렇지 않으면 값을 초기화하고 다시 덧셈준비를 한다. 또한 덧셈의 원리로는 선언한 result라는 변수에 계산된 값을 저장하여 다음 계산이 실행될 때 current상태로 들어오는 데이터를 CLA를 통해 계산해서 다시 result에 저장하는 방식을 사용하고 있다.



다음 그림은 FIFO와 Adder의 간단한 알고리즘을 flow chart로 나타낸 것이다.

Multiplier (FIFO)

다음은 연산에 필요한 곱셈기이다. 말그대로 데이터의 곱을 담당하는 logic으로써 adder와 마찬가지로 FIFO와 연결하여 사용한다. 곱셈연산이 모두 완료되면 adder와 연결되어 있는 FIFO에 값을 넘겨주며 순차적으로 연산을 진행할 수 있도록 되어있다.

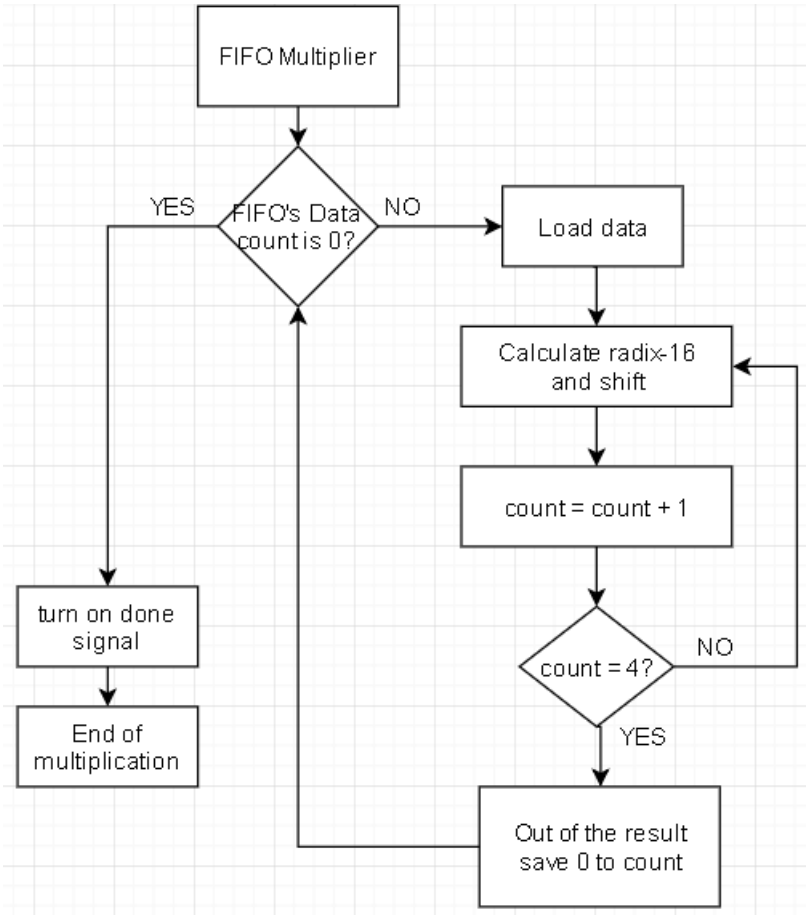


adder와는 다르게 입력 받는 값이 승수, 피승수로 나누어지기 때문에 FIFO또한 두개가 필요하며, 출력되는 값으로는 계산의 완료를 알리는 Done과 Result로 나눌 수 있다. 이번 프로젝트에서는 앞선 실습에서 구현한 Radix-4 multiplier를 응용하여 Radix-16 multiplier를 구현하게 되었는데 Radix-16에 관련된 정보는 아래 표를 참고하면 된다.

X_i	X_{i-1}	X_{i-2}	X_{i-3}	X_{i-4}	0peration	Y_i	Y_{i-1}	Y_{i-2}	Y_{i-3}	Y	Description
0	0	0	0	0	0	0	0	0	0	0	only 4bit shift
0	0	0	0	1	A	0	0	0	1	1	add A and 4bit shift
0	0	0	1	0	A	0	0	1	-1	1	add A and 4bit shift
0	0	0	1	1	2A	0	0	1	0	2	add 2A and 4bit shift
0	0	1	0	0	2A	0	1	-1	0	2	add 2A and 4bit shift
0	0	1	0	1	3A	0	1	-1	1	3	add 3A and 4bit shift
0	0	1	1	0	3A	0	1	0	-1	3	add 3A and 4bit shift
0	0	1	1	1	4A	0	1	0	0	4	add 4A and 4bit shift
0	1	0	0	0	4A	1	-1	0	0	4	add 4A and 4bit shift
0	1	0	0	1	5A	1	-1	0	1	5	add 5A and 4bit shift
0	1	0	1	0	5A	1	-1	1	-1	5	add 5A and 4bit shift
0	1	0	1	1	6A	1	-1	1	0	6	add 6A and 4bit shift
0	1	1	0	0	6A	1	0	-1	0	6	add 6A and 4bit shift
0	1	1	0	1	7A	1	0	-1	1	7	add 7A and 4bit shift
0	1	1	1	0	7A	1	0	0	-1	7	add 7A and 4bit shift
0	1	1	1	1	8A	1	0	0	0	8	add 8A and 4bit shift

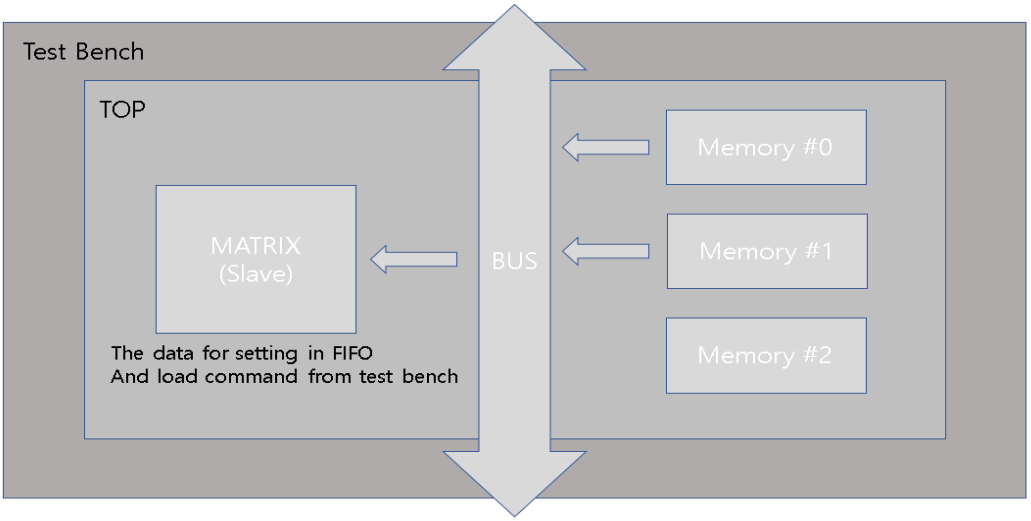
1	0	0	0	0	-8A	-1	0	0	0	-8	subtract 8A and 4bit shift
1	0	0	0	1	-7A	-1	0	0	1	-7	subtract 7A and 4bit shift
1	0	0	1	0	-7A	-1	0	1	-1	-7	subtract 7A and 4bit shift
1	0	0	1	1	-6A	-1	0	1	0	-6	subtract 6A and 4bit shift
1	0	1	0	0	-6A	-1	1	-1	0	-6	subtract 6A and 4bit shift
1	0	1	0	1	-5A	-1	1	-1	1	-5	subtract 5A and 4bit shift
1	0	1	1	0	-5A	-1	1	0	-1	-5	subtract 5A and 4bit shift
1	0	1	1	1	-4A	-1	1	0	0	-4	subtract 4A and 4bit shift
1	1	0	0	0	-4A	0	-1	0	0	-4	subtract 4A and 4bit shift
1	1	0	0	1	-3A	0	-1	0	1	-3	subtract 3A and 4bit shift
1	1	0	1	0	-3A	0	-1	1	-1	-3	subtract 3A and 4bit shift
1	1	0	1	1	-2A	0	-1	1	0	-2	subtract 2A and 4bit shift
1	1	1	0	0	-2A	0	0	-1	0	-2	subtract 2A and 4bit shift
1	1	1	0	1	-A	0	0	-1	1	-1	subtract A and 4bit shift
1	1	1	1	0	-A	0	0	0	-1	-1	subtract A and 4bit shift
1	1	1	1	1	0	0	0	0	0	0	only 4bit shift

해당표는 Radix-16으로 연산하는 경우 Radix의 경우를 따져 operation 만큼 더해준 후 4bit를 shift해주면 된다. 다음그림은 multiplier에 대한 flow chart이다.

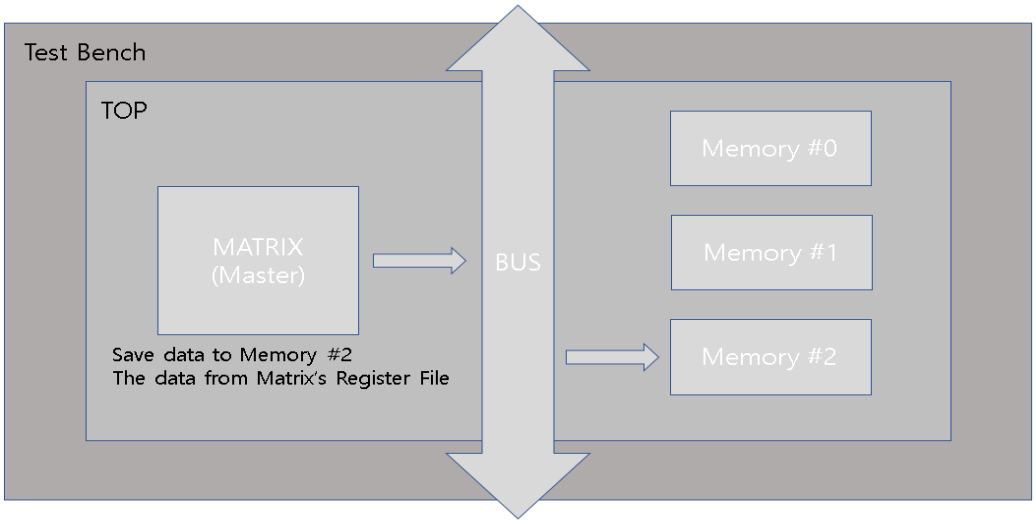


Slave & Master

Slave와 Master의 기능은 Bus를 통해서 미리 언급한 내용이 있다. Slave란 일정 명령이나 신호를 받아 해당하는 명령의 따라 동작하고, 값을 출력하는 역할을 하게 된다. 이번 Matrix module에는 Slave logic이 있는데 이는 Matrix가 Slave가 될 수 있다는 것을 뜻하며, 버스를 통해 명령을 받으면 ram으로부터 값을 가져와 연산을 진행하게 된다. Offset의 관련된 flow chart는 matrix항목에 설명되어 있으며, Slave의 간단한 동작그림은 다음과 같이 나타낼 수 있다.



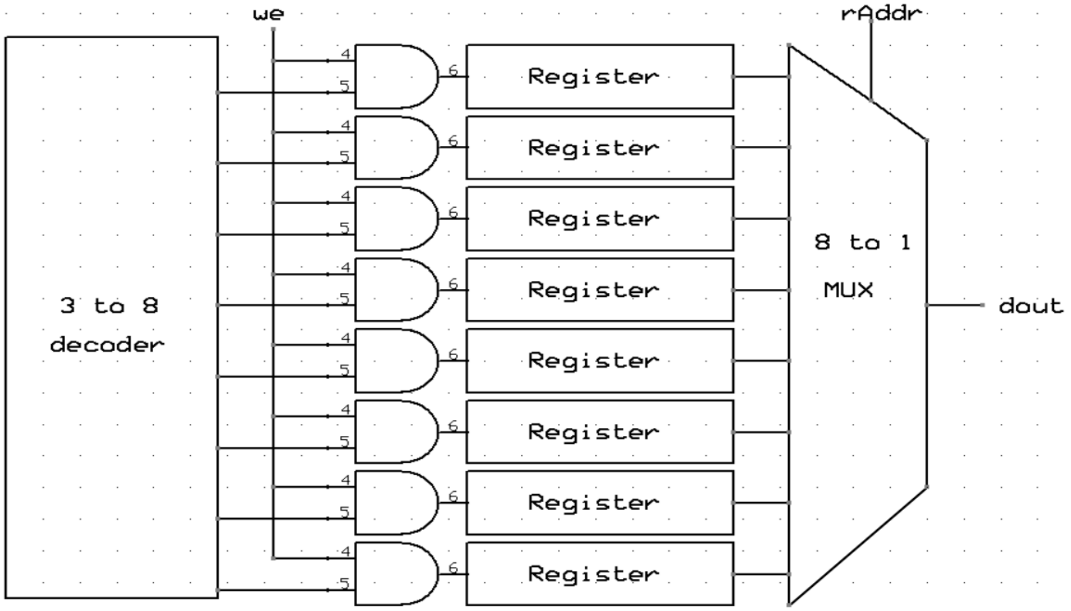
명령 주소에 따라 Memory에서 읽은 값을 저장하는 위치가 달라지며(multiplicand, multiplier) 값이 준비가 되면 test bench로부터 곱셈연산을 시작하는 명령주소를 받고, done signal이 on 되기 전까지 반복하며, done signal이 on되면 덧셈연산을 시작한다. 이렇게 행렬의 모든 연산이 끝나게 되면 Matrix는 Master가되어 Bus를 통해 Memory에 값을 저장하는데 관련된 간단한 그림은 다음과 같이 나타낼 수 있다.



Master란 명령을 전달하여 Slave가 동작할 수 있도록 하며, Slave에서 특정 명령을 수행할 수 있도록 하거나 값을 출력해오는 역할을 하는 장치이다. 이번 Project의 matrix는 Slave가 될 수 있을 뿐 아니라 Master도 될 수 있으며, Master가 되었을 경우 동작은 위 그림과 같이 Bus를 통해 Matrix내부에 존재하는 Register File에서 결과값을 Memory로 전달하여 최종적인 값을 저장하는 역할을 하게 된다. 이번 프로젝트에서는 해당 동작을 수행하면 Matrix는 Master의 권한을 포기하게 되는 특징을 가지고 있다.

3. Design Details

Register File



Register file은 데이터를 읽고 쓰는 장치로써 위 그림과 같은 형태로 존재한다. We는 write enable로 해당 신호가 켜지면 and gate들이 활성화되며 decoder를 통해 쓰게 될 레지스터를 선택하여 해당 레지스터의 데이터를 저장하는 방식이다. 또한 데이터를 읽는 방법은 read address signal을 통하여 원하는 레지스터에서 값을 빼내어 data out으로 값을 출력하는 방법이다. 이러한 Register를 구성하는 logic은 크게 Decoder, Mux, 로 나뉘는데 아래와 같이 설명할 수 있다.

1. 3 to 8 decoder

   설계하게 된 decoder는 3bits의 binary를 통하여 8bits의 one-hot encoding으로 바꾸어 값이 출력되게 된다. Ex) 3'b010 -> 8'b00000010 출력된 값은 wire로 각각 선언 되어있으며 instance된 and gate에 알맞은 번호에 맞춰 신호가 들어가게 되며 and gate가 활성화되어 선택된 Register의 Enable signal이 켜지며 Register의 값이 현재 data로 update되며, 저장한다.

2. Register

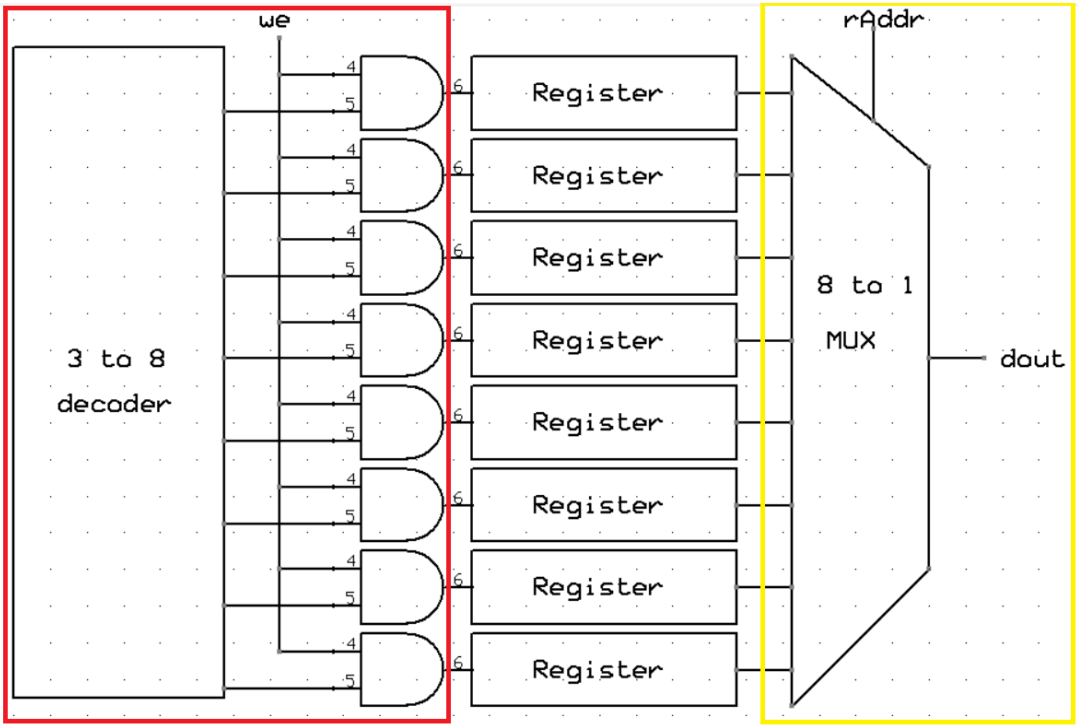
   Project에서 쓰인 Register는 32bits 8개가 쓰였으며, reset signal과 enable signal로 신호를 받는다. And gate를 통하여 enable signal을 받으며, reset신호가 들어오게 되면 다른 값에 영향을 받지 않고 바로 reset에 대한 명령을 수행한다.

3. 8 to 1 Mux

   설계된 Mux는 각각 Register output에 연결이 되어있으며, rAddr신호를 기반으로 결과 값을 선택하게 된다. rAddr은 3bits의 binary정보이며, binary정보에 따라 알맞은 Register가 선택되어 저장된 정보를 받아볼 수 있다.

추가적으로 위 데이터들의 흐름과 연결은 Write operation, Read operation이 담당하고 있다. 해당 모듈들의 구성을 살펴보면 다음과 같이 나타낼 수 있다.

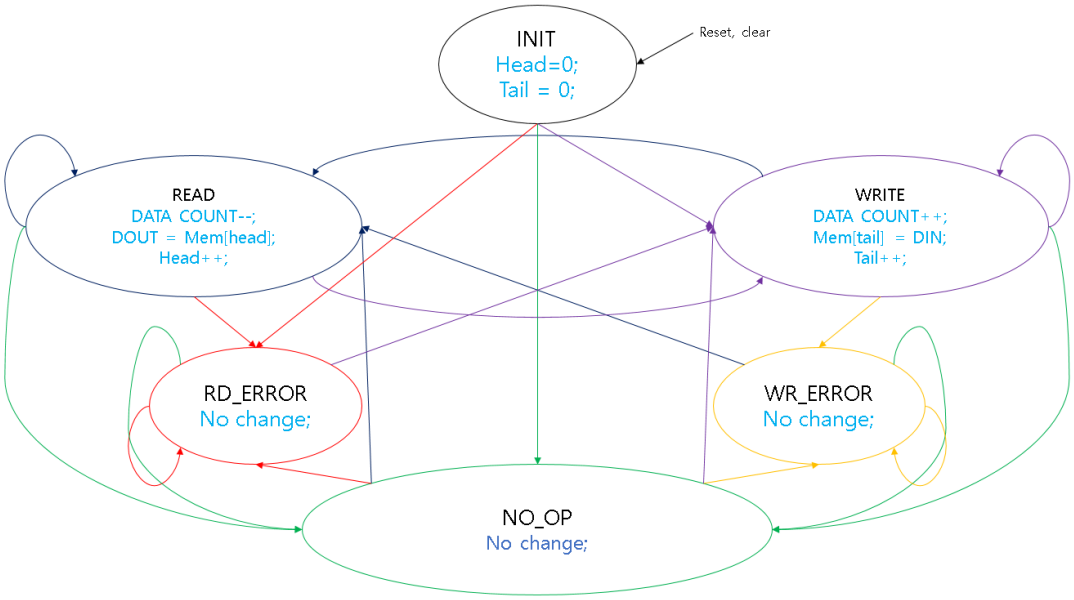
Write & Read operation



작성된 코드를 기반으로 빨간 네모 부분이 Write operation에 해당하는 부분이며, 노란색 네모 부분이 Read operation에 해당하는 부분이다. 최종적으로 Register file이 구성되기 위해서는 Write operation, Read operation, Register set이 instance되어 Top model을 이루면 된다. 아래는 RF의 PIN 정보이다.

Direction	Port name	Description
Input	clk	Clock
	clear	Active high clear
	we	Write enable
	wData[31:0]	For saving data
	wAddr[2:0]	Write address
	rAddr[2:0]	Read address
Output	rData[31:0]	The data of readied

FIFO



State	INIT, READ, RD_ERROR, WRITE, WR_ERROR, NO_OP
Input	rd_en, wr_en, din, clear
Output	Data_count, Dout

State

INIT: 초기상태를 뜻하며, reset을 통하여 값을 0으로 초기화 시키며 상태에 진입한다.

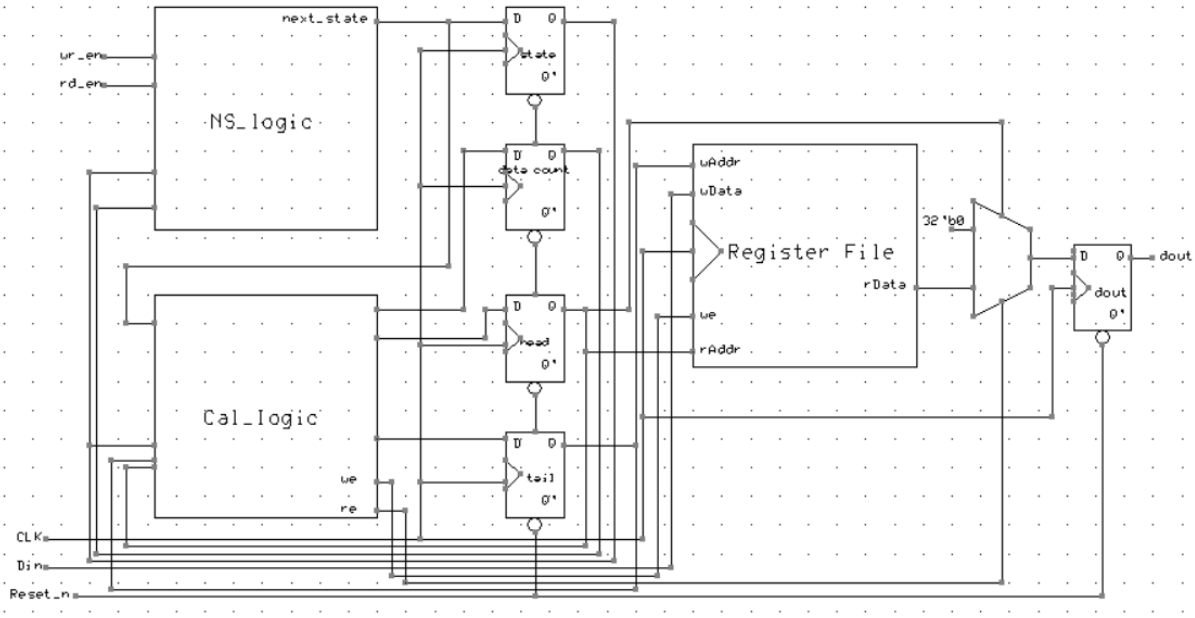
NO_OP: 명령을 수행하지 않는 상태로 enable의 값이 잘못 들어오는 경우도 해당한다.

WRITE: 들어온 데이터를 레지스터에 저장할 수 있는 상태를 뜻한다. Data의 count가 증가하며, 8개의 count가 모두 쓰인다면 full signal이 반응한다.

READ: 기존에 저장된 데이터를 읽는 명령을 수행하는 상태로, Data의 count가 하나 줄어들며, FIFO의 특징으로 읽힌 Head가 increase되어 읽힌 데이터는 접근하지 못한다. 또한 Data count를 판단하여 0인 경우 Empty signal이 반응한다.

WR_ERROR: Data count를 판단하여 Full인 상태이며, 특정한 연산은 진행하지 않는다.

RD_ERROR: Data count를 판단하여 Empty인 상태이며, 특정한 연산은 진행하지 않는다.



다음 그림은 FIFO의 회로이다. 각각 로직의 대한 설명은 아래와 같다.

NS_logic (fifo_ns): next state를 결정해주는 logic으로써 각각의 상태와 input으로 들어오는 조건(clear, data count, write enable, read enable...등등)에 따라 다음 상태가 결정된다. 예를 들어 현재 상태인 INIT에서 write enable의 값이 high가 되면 다음 state는 WRITE state로 결정되는 것을 확인할 수 있으며 각각의 상태마다 다음 상태가 결정되는 조건은 다양하다.

Cal_logic (fifo_cal): 현재 상태를 판단하고, data count를 판단하여 실질적인 계산을 담당해주는 logic이다. 예를 들어 현재의 state는 WRITE이며, data count가 7이 아니면 tail의 값을 하나 증가시켜주고 data count가 하나 증가되며 head는 현상태를 유지한다. 이처럼 실질적으로 logic에서 사용되는 data들의 update가 일어나는 logic이라고 생각하면 쉽다.

Register File (RF): 값을 읽고 쓰는데 실질적인 동작을 하는 logic이다. 값을 읽고 쓰는 조건과 저장할 주소 읽을 주소는 전에 설명한 logic에서 결정되어 정보가 전달되게 된다. 세부적으로 mux가 사용되어 값을 선택할 수 있다. 지속적인 언급과 실습을 통해 익숙한 logic이다.

Direction	Port name	Description
Input	clk	Clock
	reset_n	Active low reset
	opclear	Operation clear
	din[31:0]	Data input
	wr_en	Write enable
	rd_en	Read enable
	dout[31:0]	Data output
Output	data_count[3:0]	Count of data
	dout[31:0]	Data output

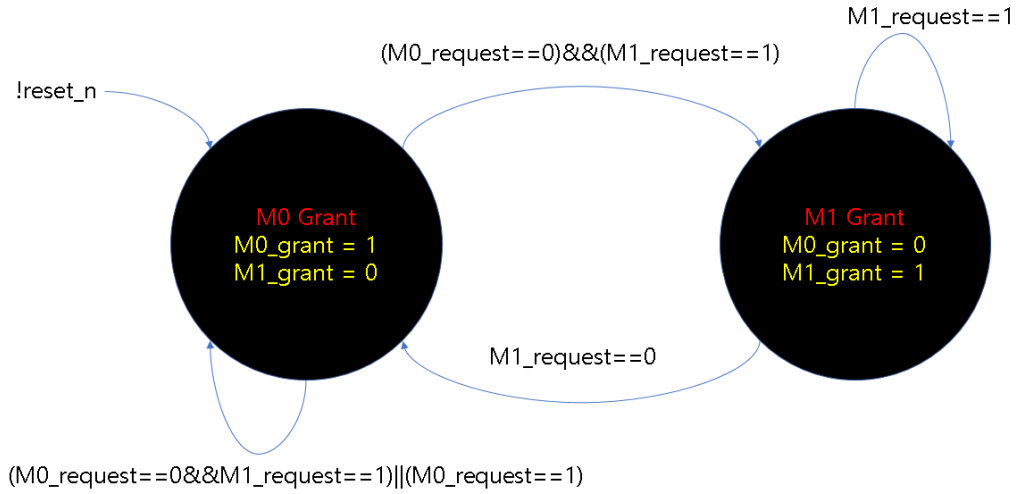
BUS

버스는 앞서 설명한 개념과 어떤 logic의 중간에서 데이터의 이동을 제어해주는 logic이다. 기본적인 버스의 개념에는 Master와 Slave가 존재하는데 Master는 Bus를 통해 명령을 주어 Slave로 선택된 logic의 동작을 제어할 수 있도록 한다. 예를 들면 원하는 메모리의 정보를 받아오는 것과 같은 것이다. 방금 설명한 것과 마찬가지로 Slave는 Master의 명령에 따라 동작하는 logic을 말한다. 이를 제어해주는 역할을 하는 것이 Bus이다. 해당 프로젝트에서는 2개의 Master Test bench와 Matrix, 4개의 Slave ram1, 2, 3 Matrix가 존재한다. 아래는 BUS의 PIN 정보이다.

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

-Arbiter

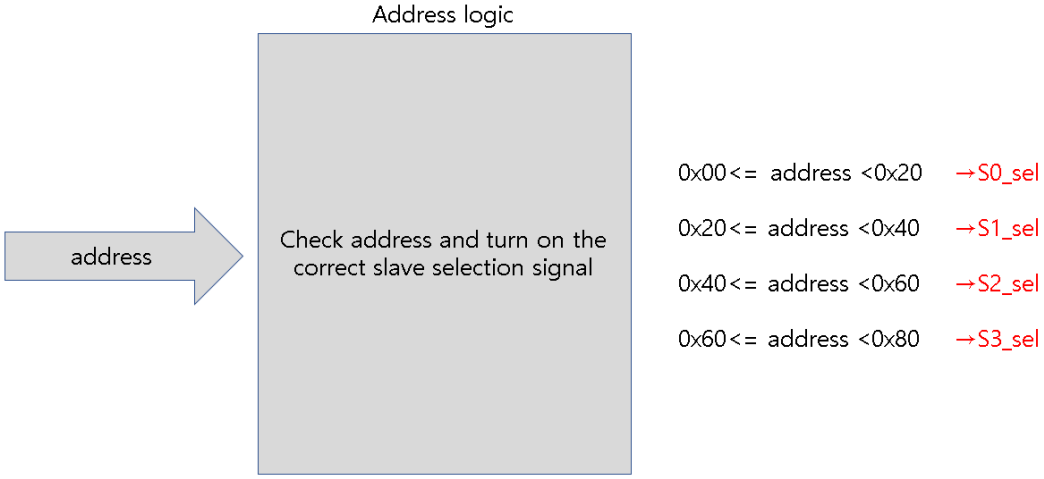
Bus를 설계하는 과정에서 중요한 module로는 Arbiter가 있다. Arbiter란 master가 되는 logic으로부터 request신호를 받아 Bus를 통하여 명령을 주는 Master를 결정하는 역할을 하게 된다. 다음 그림은 Arbiter의 대한 동작을 diagram으로 나타낸 것이다.



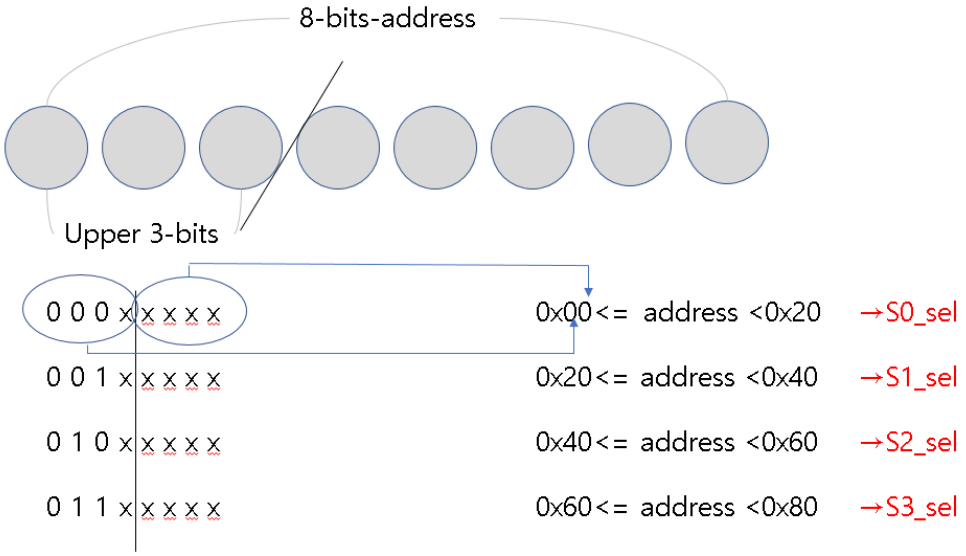
Grant Signal은 선택된 master의 승인을 표시해주는 signal이다. 해당 signal을 통하여 현재 선택되 마스터가 무엇인지 알 수 있는 지표가된다. 설계하게 된 Bus는 그림과 같이 두상태로 이루어져 있으며 grant signal이 동시에 켜지는 경우가 없기 때문에 두개의 마스터가 동시에 Bus의 주도권을 가질 수 없는 형태로 되어있다.

-address logic of bus

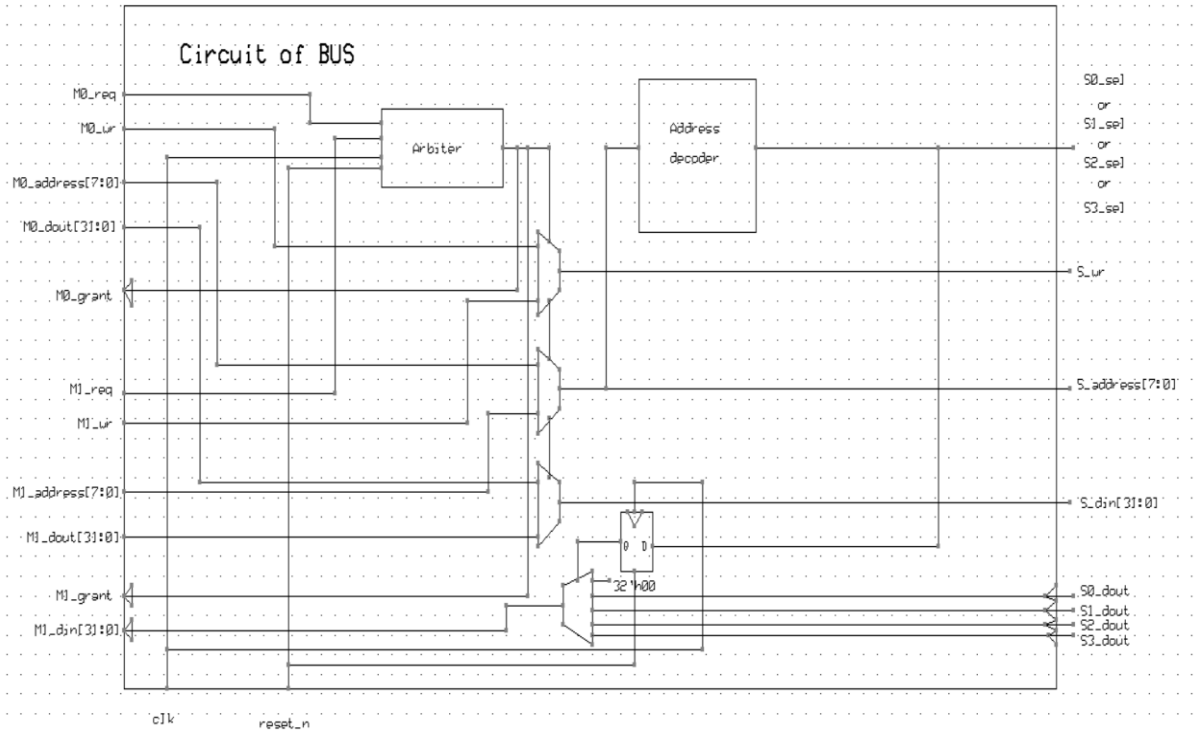
Bus는 arbiter와 master가 주는 address를 통해 어떤 slave를 선택할지에 대한 판단을 해주는 address logic으로 나뉘게 된다.



Slave를 선택하는 방법으로는 위 그림과 같이 전체 address를 비교하여 Slave를 선택하는 방법과 상위 3bits를 이용하여 선택되는 Slave를 결정하는 방법이 있다. 아래 그림은 상위 3bits를 이용한 방법을 나타낸 그림이다.

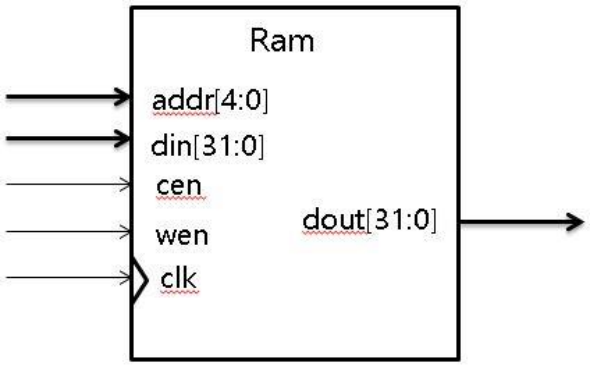


그 후 하위 5-bits를 이용하여 RAM(memory)내의 위치한 알맞은 address를 찾아 값에 접근할 수 있다. 단 상위 3-bits가 000이면 matrix가 선택되는 것으로 하위 5-bits는 offset이다. 아래 그림은 Project를 위해 설계한 BUS의 회로도다.



기존 실습했던 회로와는 크게 다르지 않지만 Slave의 개수가 늘어나면서 Slave에 대한 select신호가 2개 늘어난 것이 특징이다. 각각의 Mux는 2개의 master에게 모두 정보를 받고 있으며 Arbiter를 통하여 출력되는 grant signal을 통해 mux로 연결되어 있는 master의 데이터 중에 해당하는 grant signal의 해당하는 master의 값을 선택하여 출력하는 것을 확인할 수 있다. 각각의 출력에는 address, data, write read signal이 있다. 아래 있는 mux는 상위 decoder에 입력된 주소 값을 기반으로 선택된 Slave의 값을 선택할 수 있도록 하는 Mux이다. 이처럼 BUS는 master와 slave간의 데이터 교류를 도와주는 역할을 하게 된다.

Memory



해당 Symbol은 memory, 즉 Ram을 나타낸 것이다. Project에서는 승수와 피승수로 나뉘어 연산의 재료로 쓰일 data를 저장할 Ram 2개와 결과 값을 저장할 Ram 1개가 필요하다. Input으로 들어오는 address는 master가 주는 하위 5-bits에 해당하며, 해당 위치 data에 접근할 때 사용된다. 아래는 Memory의 Pin정보다.

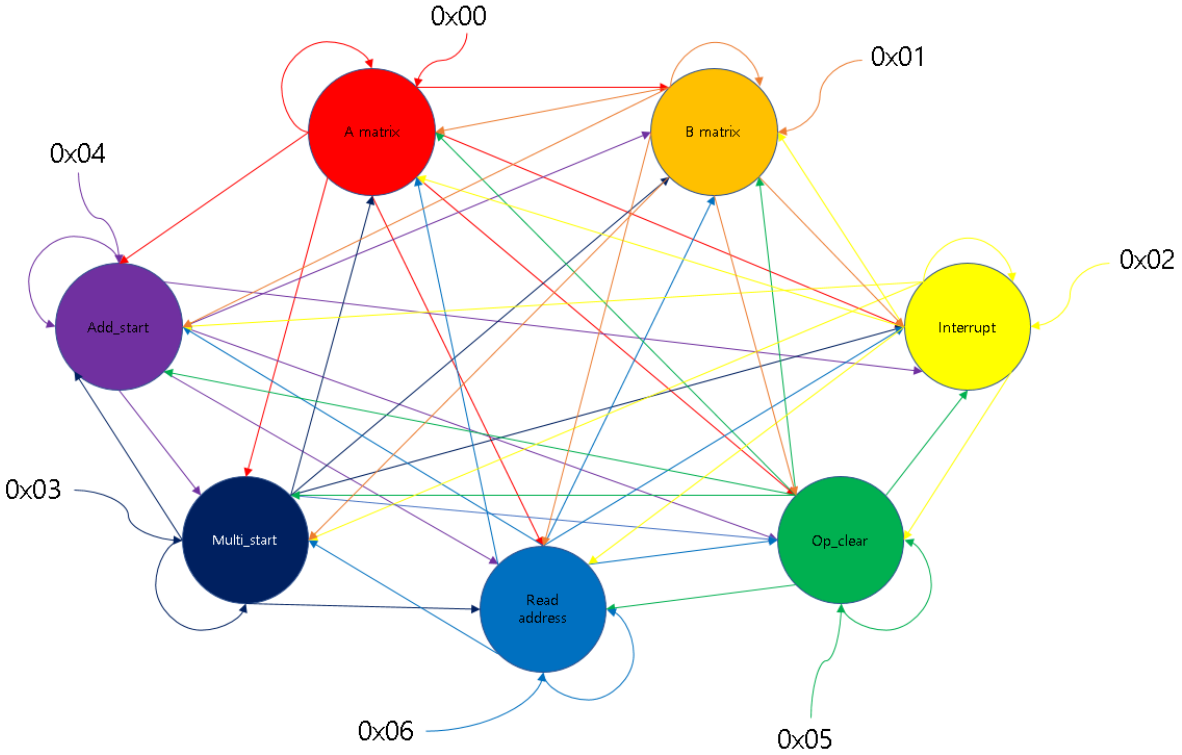
Direction	Port name	Description
Input	clk	Clock
	cen	Chip enable signal
	wen	Write enable signal
	addr[4:0]	Address
	din[31:0]	Data in
Output	dout[31:0]	Data out

Cen signal: Chop enable signal로써 해당 signal이 high상태에 있어야 Memory에 접근하여 data를 load하거나 save하는 동작을 하게 된다. Signal이 low에 위치한다면 dout의 값이 0으로 출력될 뿐 다른 동작을 하지 않게 된다.

Wen signal: Write enable signal로 data를 read할지 write할지 결정하는 signal이다. 조건으로는 Cen signal이 항상 high라는 조건에서 해당 signal이 존재하는데 의의를 갖게 된다. 해당 signal이 1로 high상태면 address를 check하여 범위에 해당하는 위치에 값을 업데이트 하게 된다. Signal이 0이라면 해당 범위의 값을 출력하게 된다. 데이터가 없다면 for조건으 통하여 해당하는 RAM을 모두 초기화 시켰기 때문에 RAM을 0으로 초기화 시켰기 때문에 0이 출력된다.

Matrix

Matrix의 동작은 명령주소에 따라 동작을 진행하게 된다. 해당 동작들의 diagram은 다음 그림과 같이 나타낼 수 있다.



offset	Type	Name	Description	Default value
0x00	R/W	A Matrix	해당 offset이 들어오면 write enable을 on 시켜 multiplicand_fifo에 값을 저장한다.	0x00
0x01	R/W	B Matrix	해당 offset이 들어오면 write enable을 on 시켜 multiplier_fifo에 값을 저장한다.	0x00
0x02	W	INTERRUPT ENABLE	해당 register의 [0] bit가 1이면 MATRIX의 adder_opdone 값이 0x01일 때 (종료되었을 때), m_interrupt port에서 1이 출력된다. 해당 register의 [0]bit가 0이고 adder_opdone 값이 0x01일 때, m_interrupt port에서 0이 출력된다. 해당 register의 [31:1]bits는 reserve이다.	0x00
0x03	W	MULTISTART	해당 register의 [0] bit가 1이 써지면 FIFO에서 POP하며 A Matrix의 값을 이용한 곱셈 연산이 시작된다. 곱셈 연산 중 해당 register의 [0]에 1이 써지면 해당 값은 무시된다. 해당 register의 [31:1]bits는 reserved이다.	0x00
0x04	W	ADDERSTART	해당 register의 [0]bit에 1이 써지면 FIFO에서 POP하여 행렬이 곱해진 값을 이용한 덧셈 연산이 시작된다. 덧셈 연산 중 해당 register의 [0]에 1이 써지면 해당 값은 무시된다. 해당 register의 [31:1]bits는 reserved 이다.	0x00
0x05	W	OPERATION CLEAR	해당 register의 [0]bit에 1이 써지면 모든 register의 값이 default value가 된다. 해당 register의 [31:1]bits는 reserved이다.	0x00
0x06	R/W	READ ADDRESS	해당 register를 통해 연산 결과가 저장되어 있는 register file의 주소를 설정한다. 해당 register의 [31:4]bits는 reserved 이다.	0x00

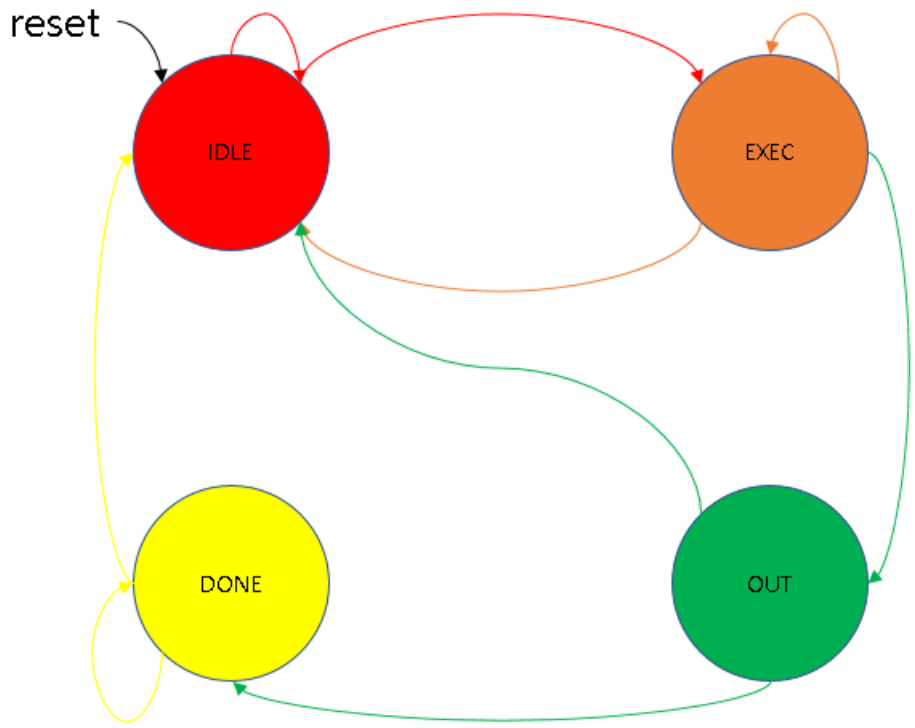
다음 설명은 해당 offset에 대한 Matrix의 동작과 동작에 알맞은 기능을 설명한 것이다.

다음은 Matrix에 대한 Pin정보다.

Direction	Port name	Description
Input	clk	Clock
	reset_n	Active low reset
	S_sel	(Slave interface) Select
	S_wr	(Slave interface) Write/read
	S_address[7:0]	(Slave interface) Address
	S_din[31:0]	(Slave interface) Data input
	M_din[31:0]	(Master interface) Data input
Output	S_dout[31:0]	(Slave interface) Data output
	multi_done	Done
	m_interrupt	Interrupt
	M1_req	(Master interface) Request
	M1_wr	(Master interface) Write/Read
	M1_address[7:0]	(Master interface) Address
	M1_dout[31:0]	(Master interface) Data output

해당 설명까지는 Matrix가 명령주소를 받아 수행하게 되는 동작에 대한 간단한 설명과 Matrix의 Port에 관련한 설명이었다. 다음으로 소개될 module들은 Matrix의 구성 module들이며, offset에 의한 동작을 도와주는 module로써 Matrix의 세세한 동작을 알아보는 항목이다.

Adder



해당 그림은 이번 프로젝트에서 설계된 adder의 diagram이다. 특징으로는 실습에서 제시된 multiplier등의 diagram과 다르게 OUT state가 존재하는 것이 특징이다. DONE state는 count를 판단하여 필요한 값이 생성되었을 경우 값을 Matrix의 최종적인 Register File에 저장해주기 위한 state이며, OUT state는 단순히 진행되는 과정과 signal들을 가시적으로 보여주기 위한 state라고 할 수 있다. 각각의 state의 자세한 설명은 아래와 같이 정의할 수 있다.

IDLE: IDLE상태는 계산의 초기를 뜻하며 계산에 대한 준비를 하는 상태이다. 간단한 동작을 설명하면 모든 계산에 필요한 값들을 0으로 초기화 해주는 동작을 한다.

EXEC: 실질적인 계산을 하는 state로써 해당 module은 add이므로 들어오는 값에 대하여 순차적으로 덧셈이 되는 것을 확인할 수 있으며, count를 판단하여 OUT state로 이동한다.

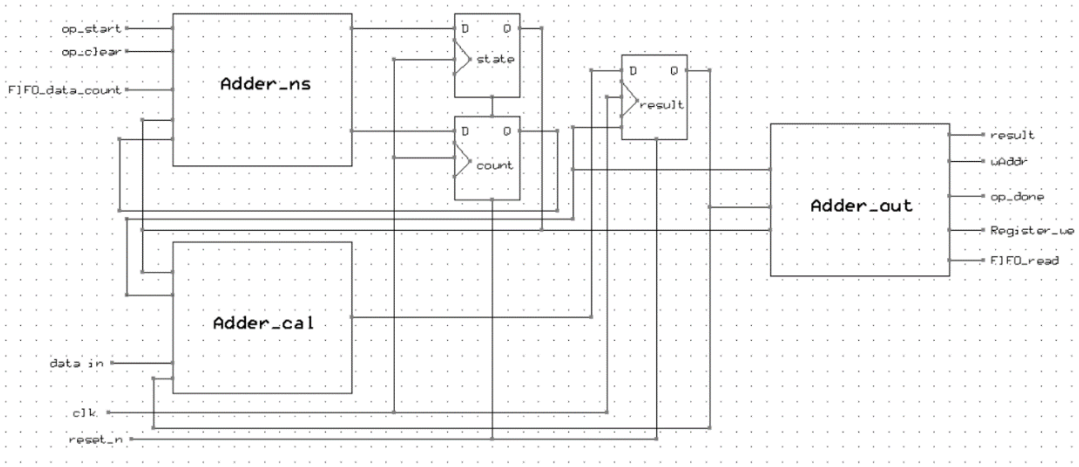
OUT: 계산된 값을 출력해주는 state로 복잡한 동작은 없는 state이다. Data count를 판단하여 다시 IDLE로 가서 계산을 할지 DONE state로 가서 계산의 끝을 알리는지 판단하게 된다.

DONE: OUT state에서 data count를 판단하여 넘어오게 되는 state이다. 모든 덧셈의 끝을 알리며, operation start나 clear signal이 따로 존재하지않으면 자동적으로 IDEL state로 넘어가게 된다.

다음 표는 Adder의 PIN정보를 나타낸다.

Direction	Port name	Description
Input	clk	Clock
	reset_n	Active low reset
	Opclear	Clear signal
	Opstart	Start signal
	din[31:0]	Data in
	Fifo_data_count[3:0]	Number of fifo data
Output	Out_result[31:0]	Data out
	Opdone	Done signal
	Fifo_re	FIFO read enable
	RF_we	Register file write enable
	wAddr[2:0]	Register file address

다음 그림은 adder에 대한 circuit이다.

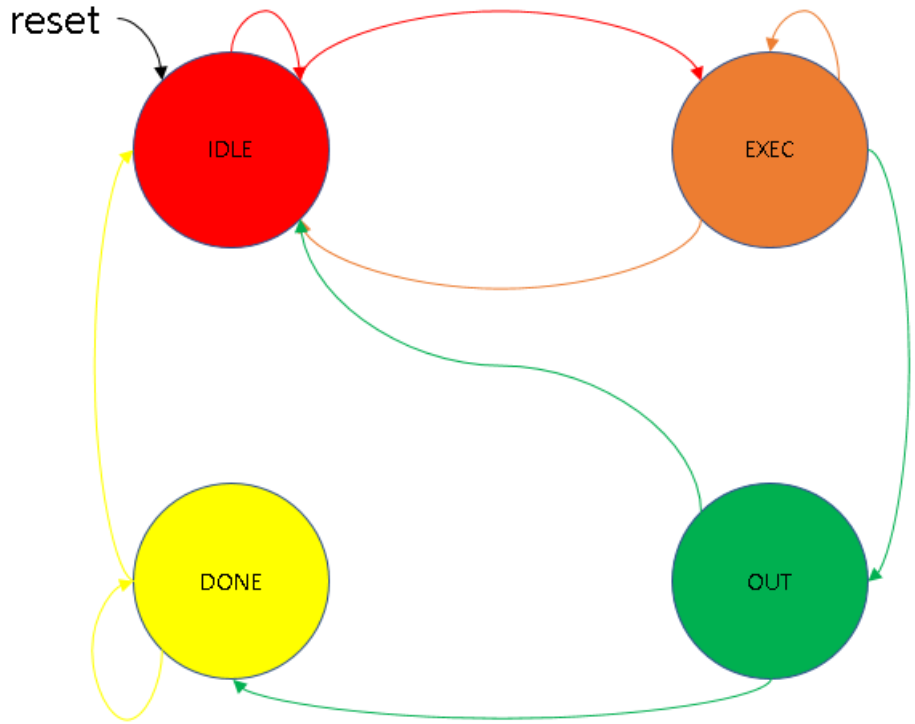


다음은 각각 logic에 대한 간단한 설명이다.

logic	Description
Adder_ns	Adder의 Next state를 결정해주는 logic으로써 다음 동작을 결정하는 조건으로는 op_clear, op_start, adder내부의 count, 정보를 받아오는 FIFO의 data_count로 이루어져 있다.
Adder_cal	Adder의 실질적인 계산을 담당하는 logic으로써 해당 기능의 의미는 EXEC state일 때 의미를 갖게 된다. 해당 logic안에 CLA가 선언되어 있으며, 덧셈의 next_result에 각각의 결과 값을 저장하게 된다.
Adder_out	전체적인 결과값에 대한 output과 data결과에 대한 output을 관리해주는 logic이다. 각각의 state에 맞게 각종 결과 값을 출력해준다.

Multiplier

이전 실습을 통하여 구현한적이 있는 multiplier다. 다른 점으로는 FIFO와 같이 연결해야 하며, Out state가 포함되어 있으며, radix또한 더 느리게 되었으며, 설계자체를 조금 바꾸게 되었다. 아래 그림은 이번에 구현하게 된 multiplier다.



해당 Diagram은 adder와 동일하며, 기본적인 state의 역할도 비슷하다. 각각의 state에 대한 설명은 아래와 같다.

IDLE: adder의 IDLE과 마찬가지로 계산을 준비하는 state다. 또한 다른 state에서 reset과 clear signal이 감지되면 해당 상태로 넘어오게 된다. 모든 값들이 0으로 초기화되며, 특별한 signal이 없다면 EXEC state로 넘어가며 계산을 시작한다.

EXEC: 곱셈을 진행하는 state다. Radix가 16으로 설계되었기에 기존 radix 4로 설계된 multiplier보다 적은 cycle을 기준으로 Out state로 넘어가게 되어있다. Count나 다른 특정 signal이 없다면 계속해서 계산을 하는 state에 남아있게 된다.

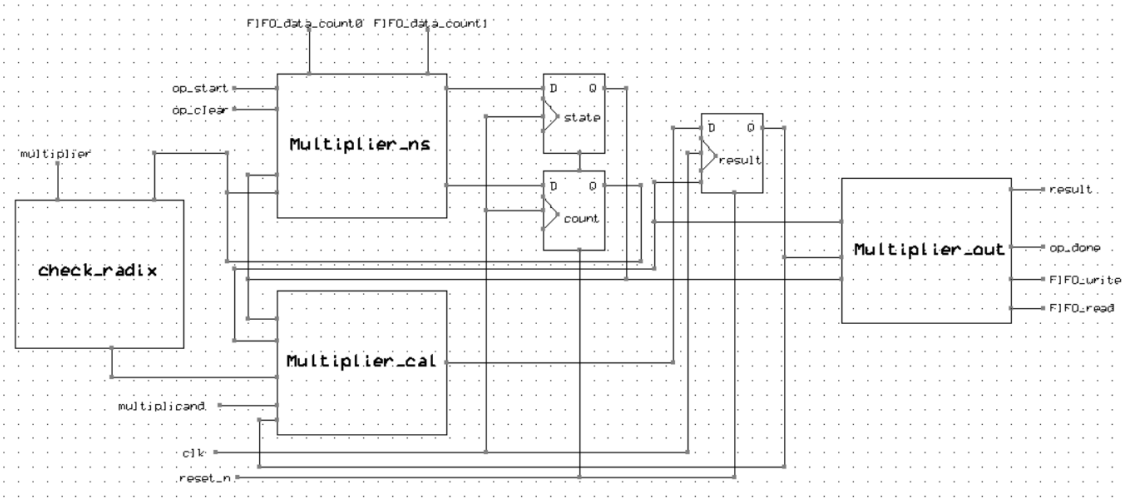
OUT: 이번 프로젝트를 진행하면서 새로 설계한 state로써 특별한 계산은 존재하지 않는 state다. Out state에 도달하면 FIFO의 data count를 비교하여 DONE state로 갈지 바로 IDLE state로 갈지 결정하게 된다.

DONE: Out state에서 FIFO count가 0이되면 이동하게 되는 state로써 더 이상 계산할 값이 존재하지 않기 때문에 해당 logic의 done 값을 on시키며, 특정 조건이 없으면 자동적으로 IDLE state로 넘어가게 된다.

다음 표는 해당 multiplier의 PIN정보다.

Direction	Port name	Description
Input	clk	Clock
	reset_n	Active low reset
	Op_clear	Clear signal
	Op_start	Start signal
	multiplier[31:0]	Data in
	multiplicand[31:0]	Data in
	Fifo0_data_count[3:0]	Number of data in fifo
	Fifo1_data_count[3:0]	Number of data in fifo
Output	Out_result[31:0]	result
	Opdone	Done signal
	FIFO_read	FIFO read enable
	FIFO_write	Register file write enable

다음 그림은 Multiplier의 회로도다.

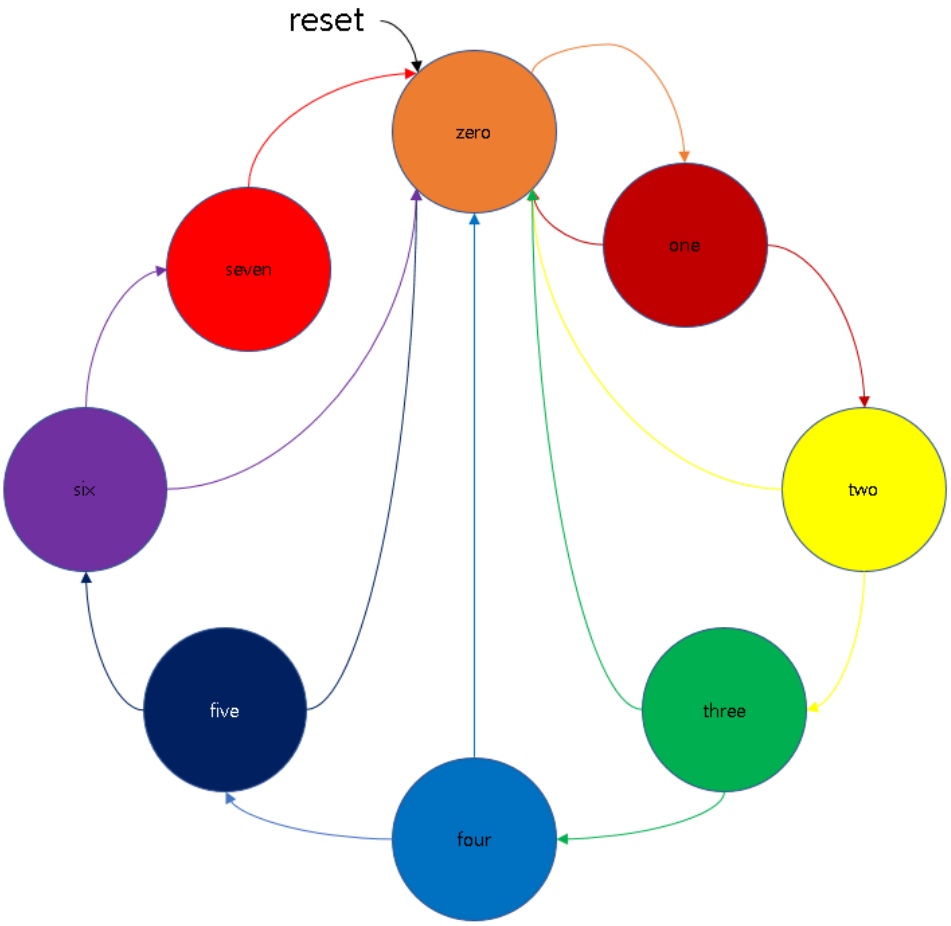


다음은 Multiplier각 logic에 대한 간단한 설명이다.

logic	Description
Sel_radix	Multiplier의 정보를 받아 계산되는 count를 비교하여 현재 계산에 알맞은 승수의 값을 radix로 변환하여 calculation logic에 input으로 보내준다 radix는 여러가지 계산 방법 중 한가지의 경우의 수를 정한다.
multiplier_cal	곱셈의 계산을 담당하는 logic이다. 해당 logic내부는 radix-16을 사용한 방법으로 선택된 radix의 경우에 맞춰 cla를 통하여 덧셈이나 뺄셈의 횟수와 shift를 결정하게 된다.
multiplier_ns	Multiplier의 다음 stat를 결정해주는 logic으로 clear signal과 start signal이 관여 하고있으며, 가장 중요한 조건으로는 count가 있다. 내부의 count는 EXEC에서 OUT stat로 이동시켜주는 지표이며, data_count는 OUT에서 DONE이나 IDLE state로 이동시켜주는 지표이다.
multiplier_out	Adder의 OUT logic과 마찬가지로 각종 signal과 결과 값을 결정해주는 logic이다. 각각 state에 맞는 결과값을 출력해주며, 연산에는 크게 관여하지 않는 logic이다.

Master

Matrix 내부에 존재하는 master module에 관련한 내용이다. 해당 module의 state는 아래와 같은 diagram으로 나타낼 수 있다.

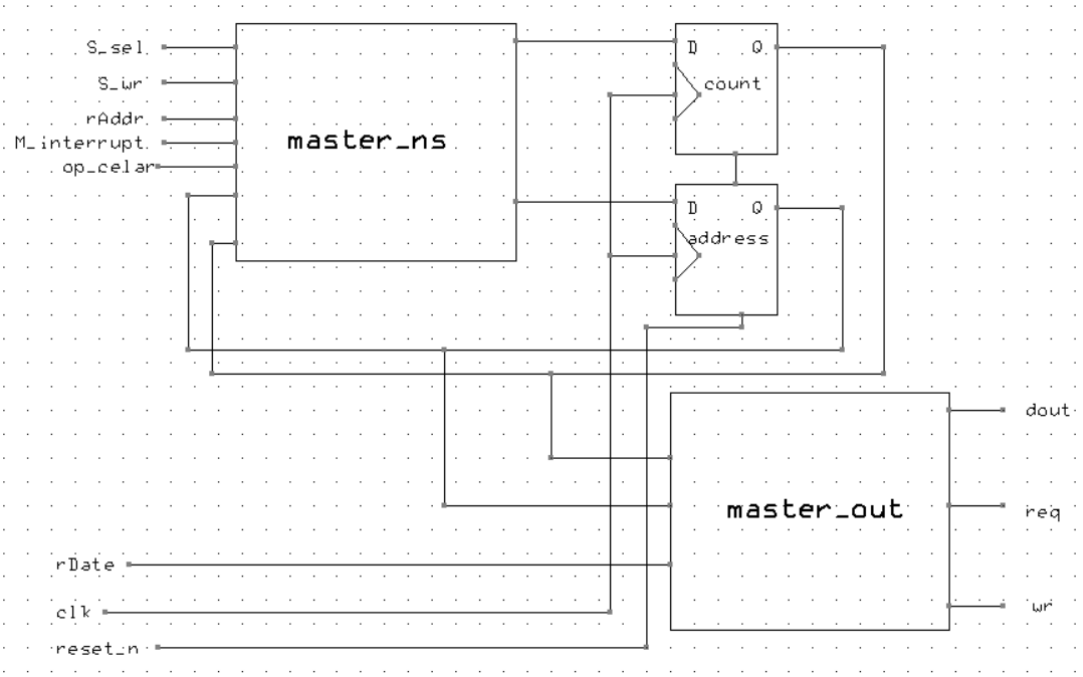


해당 state는 zero부터 seven까지 나눠진다. 이유는 polling방식을 사용하여 값을 불러오고 저장하고 불러오고 저장하는 방식을 연달아서 하기 때문에 해당 state가 8가지로 나뉘게 된다. Polling방식이란 Interrupt방식과 비교되는데 다음과 같은 차이점이 있다. 우선 Interrupt방식은 조건에 알맞은 event가 발생하면 해당 함수가 발생하는 것이다. 이와 다르게 polling방식은 어떤 event가 있는지 계속해서 monitoring을 하게 된다. 이런 특징 때문에 설계한 matrix계산기는 값이 저장될 때 값이 이어져서 순차적으로 나오는 것이 아니라 값이 나오고 끊기고 나오고 끊기고 하는 방식이다. (값을 읽고 쓰고 읽고 쓰는 방식) 그래서 state를 읽는 state와 쓰는 state를 나눠 8개로 구분하게 된 것이다.

다음은 Master logic에 대한 PIN정보다.

Direction	Port name	Description
Input	clk	Clock
	reset_n	Active low reset
	Opclear	Clear signal
	S_sel	Select signal
	S_wr	Read & write signal
	M_interrupt	Interrupt signal
	rAddr[2:0]	Register file address
	rDate[31:0]	Data in
output	M1_dout[31:0]	Data out
	M1_req	Master request
	M1_wr	Master output read & write signal
	M1_address[7:0]	Master output address

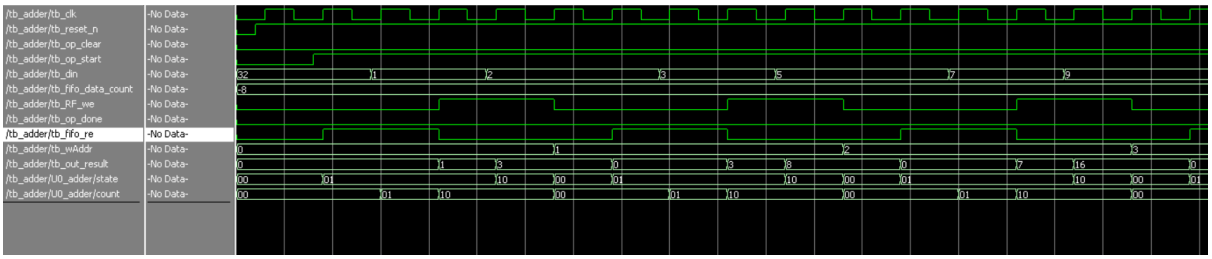
다음은 Master의 Circuit이다.



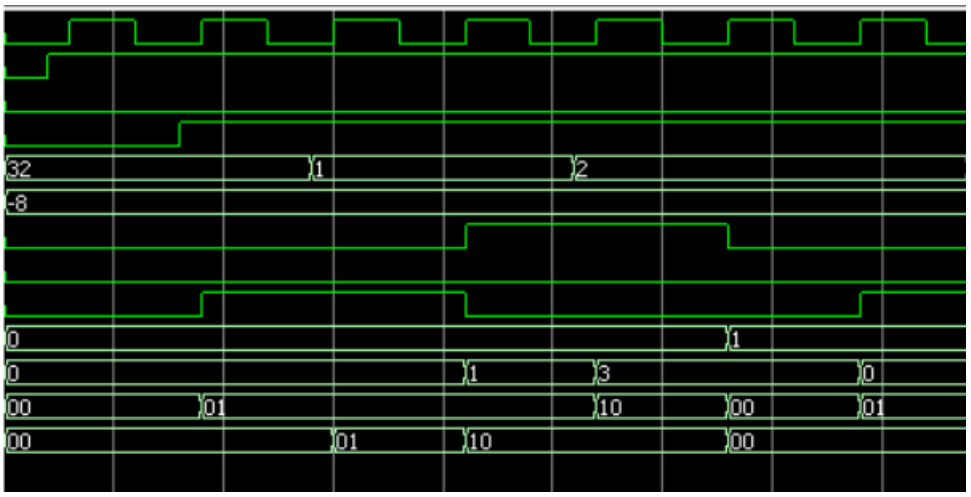
4. Design Verification Strategy and Results

Adder

Adder의 검증 전략으로는 해당 module을 실행했을 때 값이 정확히 계산되는지 판단하며, 설정해준 count에서 state가 잘 넘어가는지 확인해야하는 목적이 있었다. 다음 그림은 adder module을 실행한 결과이며, 해당 wave form에 대한 간단한 설명이다.



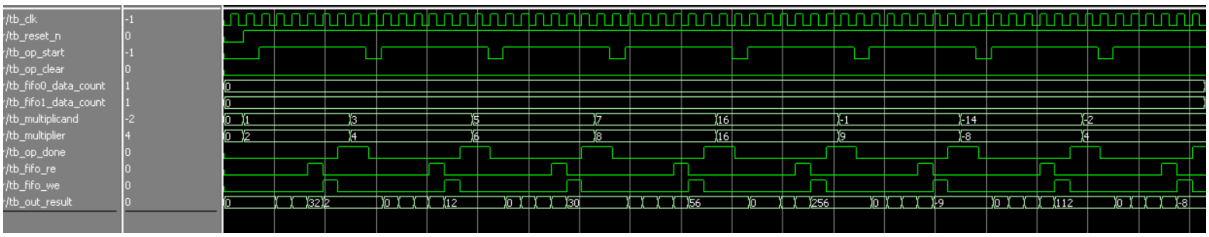
다음은 adder의 전체적인 wave form을 나타낸 것이다. 순차적으로 들어온 값을 읽어 두개의 값을 더하게 되면 다시 초기화되어 다른 data값을 읽어 덧셈을 진행하는 방식이다 좀 더 자세하게 보면 아래 wave form을 참고하면 된다.



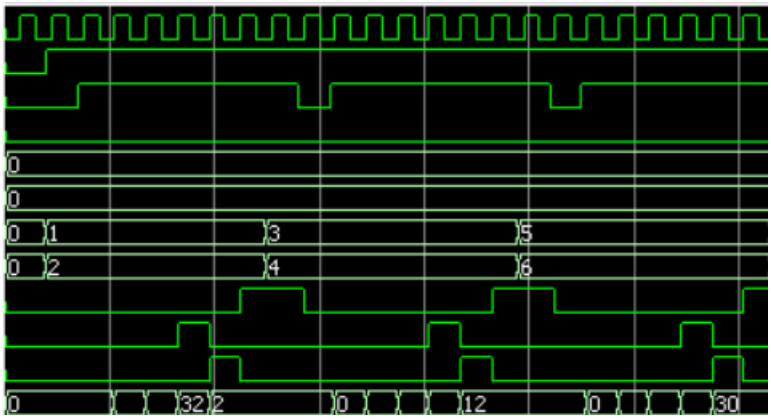
현재 adder는 FIFO가 연결되지 않았기 때문에 read enable과 write enable은 큰 뜻을 가지고 있지 않다. Data 연산을 진행하는 state는 01(EXEC)로써 해당 state로 이동했을 때부터 값을 읽어 계산하는 것을 확인 할 수 있다. 계산이 완료되는 시점은 count로 판단하게 되며 count가 2가 되면 다음 state는 IDLE로 결정되며 값을 초기화 하게 된다. 현재는 의미 없지만 write enable을 통하여 나온 결과값에 대한 정보를 Register file에 저장하게 된다.

Multiplier

Multiplier의 검증 전략은 adder와 마찬가지로 정확한 값이 잘 나오는지 판단하는데 있다. adder와는 다르게 전 실습에서 이미 각각의 state를 구분하여 설계를 하였기 때문에 새로 설계하는데 크게 무리가 없었다. 다음 이미지는 전체적인 wave form을 나타낸 것이다.



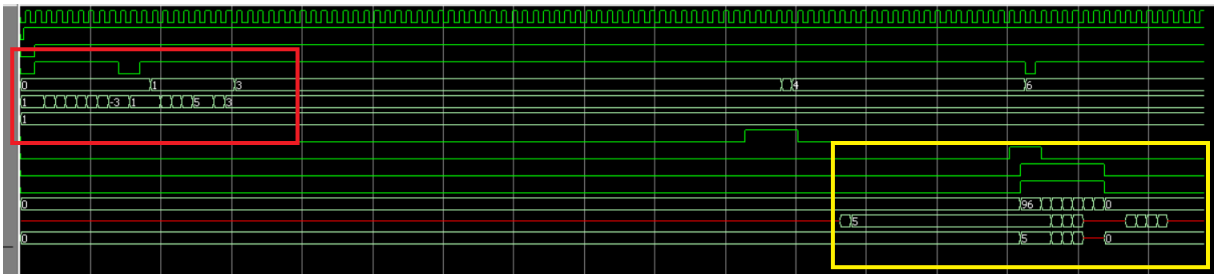
다음은 Multiplier의 전체적인 wave form이다. 이전 실습에서 구현한 multiplier와 크게 차이가 없으며 결과값이 나옴에 따라 값을 넘겨주기 위한 write signal이 rising하는 것을 확인할 수 있다. 아래 이미지는 좀더 자세한 wave form이다.



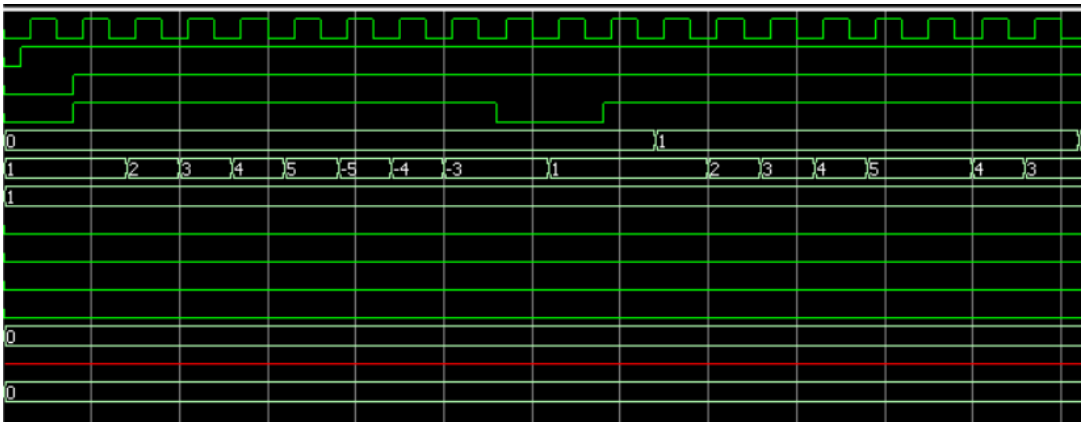
이전 실습과는 다르게 radix-16으로 설계하였기 때문에 4번의 cycle로 계산이 완료되는 것을 확인할 수 있다. 또한 계산이 완료됨에 따라 adder FIFO에 값을 넘겨주기 위하여 write signal이 rising하는 것을 확인할 수 있다.

Matrix

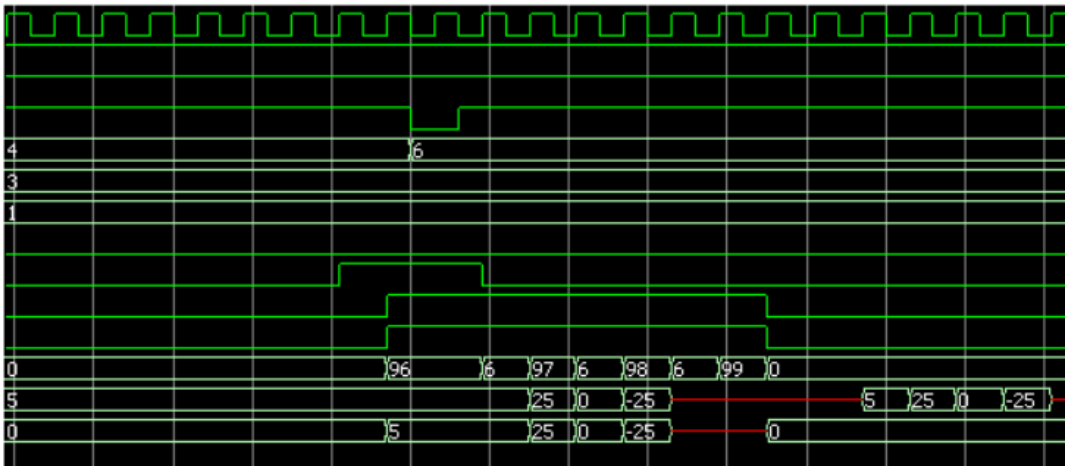
Matrix는 연산이 재료가 되는 정보를 넣고 넣은 정보가 잘 들어갔는지 확인하며 연산을 통하여 최종적으로 마지막 Register File에 값이 잘 들어가 있는지 확인하는 것이 목적이 된다. 아래는 matrix에 대한 wave form이다.



빨간 네모부분에서 test bench를 통하여 값이 잘 들어가는지 확인하며, 중간 연산을 거친 후 노란 부분에서 최종적인 값이 잘 나오는지 확인해야 한다. 해당 부분을 세부적으로 확인하면 다음과 같다.



다음 wave form은 빨간 네모 부분이며, 값이 순차적으로 들어간 것을 확인 할 수 있다. 연산은 (1 * 1) + (2 * 2) ..... (3 * 3) + (4 * 4) ..... 순서로 계산될 예정이며, 해당 결과 값이 마지막 결과 값에서 잘 나오는지 확인하면 된다.

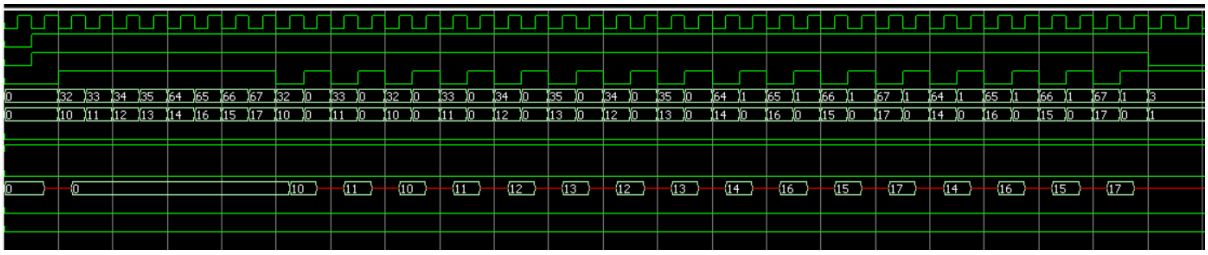


해당 wave form은 노란 네모 부분이며, 결과 값을 확인하면 된다. 각 연산에 관련하여 최종적인 값이 정확히 나온 것을 확인할 수 있다.

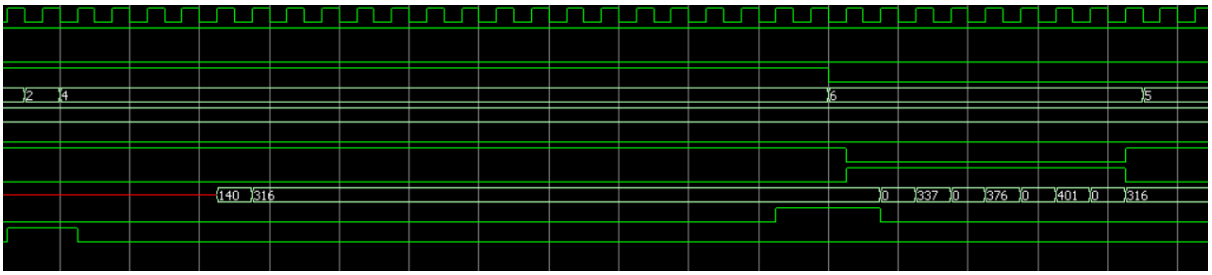
TOP

TOP모듈에서의 검증은 Project의 목적인 Memory에 값이 잘 들어가는지 확인만 해주면 된다. 이미 앞선 검증에서 계산의 절차와 Matrix가 잘 작동하는 것을 확인하였기에 TOP에서는 많은 검증을 실시할 필요가 없다. 아래 wave form은 TOP module을 실행하여 나온 wave form이다.

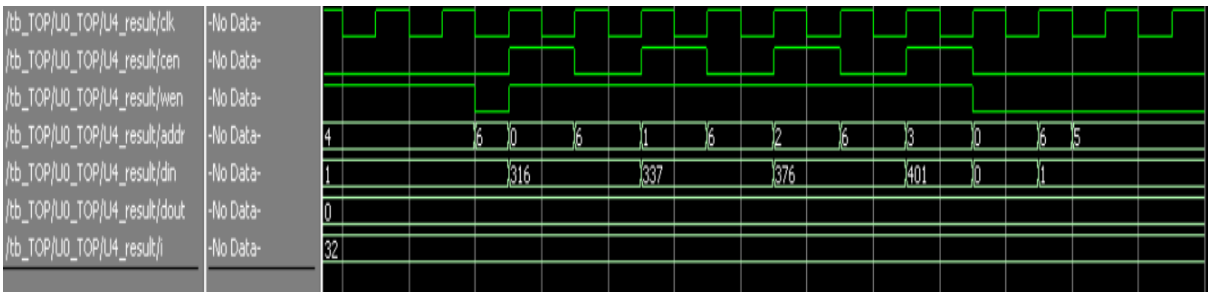
Data In 부분



Data Out(Save) 부분



TOP module 또한 Matrix와 마찬가지로 데이터가 들어오는 부분과 연산이 되어 최종적인 data를 out(save)하는 부분으로 나눌 수 있다. 결과가 나오는 것은 확인이 되었으므로 #2 Memory에 저장된 것을 확인해야 하는데 해당 결과는 아래 wave form을 보면 확인할 수 있다.



Result_ram에서 순차적으로 주소가 증가하며 값이 들어오는 것을 확인할 수 있다.

5. Conclusion

해당 프로젝트는 지금까지 실습을 통해 구현한 하드웨어를 바탕으로 완벽하게 각각의 하드웨어를 이해하고있다면 어렵지 않은 과제였을 것이다. 하지만 프로젝트를 진행하면서 진행한 실습에 대한 부족한 개념들이 있었고 특히 프로젝트 전에 구현하게 된 Bus는 제대로 이해하지 못한 상황이었기 때문에 Matrix에서의 Slave와 Master를 구현하고 이해하는데 많은 어려움이 있었다. 처음 프로젝트를 진행하면서 가장 먼저 실행한 작업으로는 Multiplier를 다시 구현하는 것과 Adder를 구현하여 FIFO에 연결하여 값이 잘 들어오는지 확인하는 작업을 하게 되었다. 처음 작업인 Multiplier와 Adder를 FIFO에 연결시키는 문제도 적지 않게 일어났다. 특히 multiplier가 FIFO에서 data를 읽어올 때 첫 값이 0이 읽혀오는 문제가 발생하여 첫 값이 0이 아닌 head의 값을 가리킬 수 있도록 모듈을 수정하며 몇번의 실험과 친구들의 의견을 종합하며 모듈을 수정하여 문제를 해결하게 되었다. Adder와 FIFO를 연결하는 데서는 clock에 동기화되는 문제가 발생하였다. Clock을 맞추어 연산을 진행하게 되면 마지막 연산에서 문제가 발생하였고 이 문제를 해결하고자 clock을 늘려주면 오히려 앞부분의 연산이 제대로 출력되지 않는 문제가 발생하였다. Clock을 변경하였지만 해결하지 못할 것 같다고 판단하여 모듈 수정에 초점을 맞추게 되었고 오류의 특징과 조건의 특징을 종합으로 확인하여 tail의 조건을 변경하며 문제를 해결하게 되었다. 다음으로 문제가 된 module은 Matrix내부의 master를 설계하는 과정이었다. 처음 master를 설계하는데 있어 메모리에 접근하여 저장하는 방법이 아니라 단순히 값을 출력하는 방법으로 wave form상에서만 값이 보이도록 설계하게 되었다. 허나 해당 방법은 예비검증을 통하여 잘못된 방법이란 것을 알고 메모리에 접근하기 위해 interrupt방식이 아니라 polling방식으로 설계하여 메모리에 접근하는 동작과 Register File에 접근하는 동작을 순차적으로 실행할 수 있도록 설계하며 문제를 해결하게 됐습니다. 해당 프로젝트를 설계하며 기대되는 학습 효과로는 무작정 코딩하는 습관에서 사전설계를 충분히 마치고 설계를 순차적으로 효율적이게 할 수 있는 방법을 익히게 되었습니다. 그리고 clock의 타이밍에 따라 값이 확연하게 달라지는 것을 확인할 수 있었으며, 타이밍의 중요성을 알게 되었습니다. 또한 순차적인 동작이 필요할 때는 FSM이 중요하다는 것을 알게 되었습니다. 이렇게 FSM을 많이 다루어 봄으로써 앞으로 비슷한 logic이나 프로젝트를 받게 되어도 충분한 사전설계와 해당 프로젝트를 설계한 경험을 토대로 복잡한 문제도 해결할 수 있다고 생각됩니다. 또한 지금까지 했던 실습 내용을 토대로 진행하는 프로젝트였기 때문에 이번 2학기에 진행한 디지털논리 2에대한 개념을 정리하고 다시 적립할 수 있는 계기가 되었습니다.

6. Reference

김영민/Finite State Machine/광운대학교/2018-2
김영민/FIFO & FIFO_new/광운대학교/2018-2
김영민/Register Files/광운대학교/2018-2
김영민/Simple Bus/광운대학교/2018-2
김영민/Booth Multiplication/2018-2

버스 관련 사전 개념/ https://en.wikipedia.org/wiki/Bus_(computing)
Mater와 Slave의 개념/ https://en.wikipedia.org/wiki/Master/slave_(technology)

Booth's multiplication algorithm
/ https://en.wikipedia.org/wiki/Booth%27s_multiplication_algorithm
/ https://www.youtube.com/watch?v=L9uYLLBXfLk

대부분의 이미지 자료와 개념 자료는 이전 실습을 통한 보고서와 개념을 많이 참고하여 사용하였습니다.

프로젝트에 도움을 준 사람들
2015722022 임유섭, 2015722047 홍진혁, 22015722027 손현식, 2015722059 오윤제