디지털 논리회로2 프로젝트 제안서

Matrix Multiplication

학　　과: 컴퓨터공학과

담당교수: 김영민 교수님

실습분반: 수요일 0, 1, 2

학　　번: 2015722025

성　　명: 정용훈

1. Title & Object

A. Title

Matrix multiplication

B. Object

2행 2열의 행렬 값을 계산

해당 프로젝트는 미리 실습을 통해 만들었던 Multiplier, Adder, FIFO, RF를 한번 더 이해하며 사용하고 행렬의 곱을 계산할 수 있는 Matrix logic를 설계하며 Memory에서 계산할 값을 불러와 Matrix를 통해 계산을 하고 Bus를 다시Memory에 저장하여 검증하는 것이 목적이다.

2. Component concept

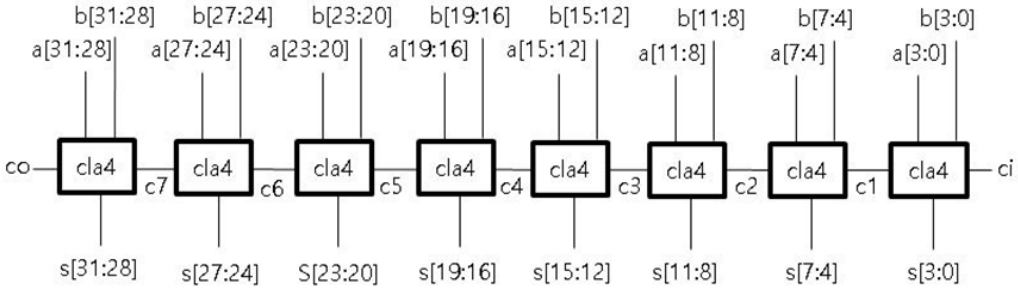
시스템의 전체적인 구성을 하는 가장 중요한 module로는 Matrix, Memory, Register File, FIFO, BUS, TOP으로 나뉠 수 있다. 아래는 Project전에 구현한 각 module의 개념과 특징을 PDF기반으로 정리한 것이다. Project를 진행하며 알맞게 module을 변경할 예정이다.

a. Matrix

MATRIX는 A Matrix와 B Matrix의 data값을 기반으로 곱하여 결과값을 도출하는 hardware이다. 각 A, B Matrix의 bit length는 최대 16bits이며 결과 값의 bit length는 최대 32bits가 된다. 또한 MATRIX는 Bus의 master interface를 통하여 data를 전송할 수 있으며 op_start signal을 통하여 곱셈과 덧셈을 시작하고 계산이 완료되면 op_done signal이 출력되며 op_clear signal을 통하여 상태를 초기화할 수 있다. Matrix는 여러가지 module로 구성되는데 가장 중요한 module은 실질적인 계산을 담당하는 Multiplier와 Adder입니다. Matrix를 구성하는 Module은 다음 항목에 계속 언급됩니다.

b. Adder

해당 Project에서 adder는 곱 연산을 한 후 그 결과가 Result FIFO에 저장된 후 Adder logic으로 값을 전달해주어 연산 된 두 곱을 더해주는 hardware입니다. Adder는 이전에 구현하였던 cla를 응용하여 사용하지 않는 output과 input을 없애면 무리 없이 사용할 수 있을 것을 예상합니다.



I/O Description(Including Wire)

포트	이름	비트단위	설명
Input	a, b	32 bit	Input data
	ci	1 bit	Carry in
Output	co	1 bit	Carry out
	s	32 bit	Summation
wire	c	1 bit	Internal carry

Module Description

구분	이름	설명
Module	cla32	32-bit carry look-ahead adder(CLA)
instance	U0~U7_cla4	4-bit carry look-ahead adder(CLA)

c. Multiplier

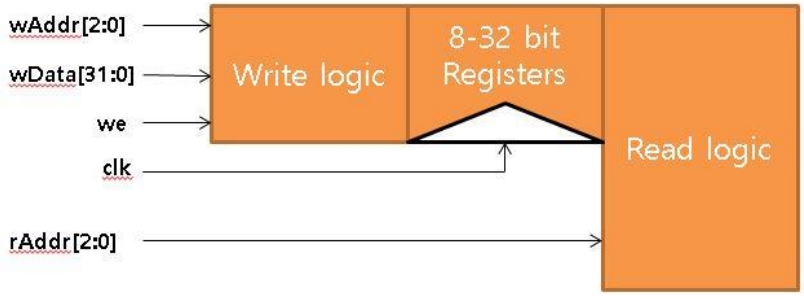
Multiplier는 가장 최근 만든 module로써 radix 4를 사용하여 계산하는 곱셈기를 구현했습니다. 이번 프로젝트를 통해서는 가능한 radix를 조금 더 증가시켜볼 생각이며 좀 더 빠른 cycle로 계산을 끝낼 수 있도록 설계할 예정입니다. Input port와 output port는 프로젝트를 구현하며 알맞게 맞출 생각이며 해당 곱셈기의 가장 중요한 개념은 다음 표를 확인하면 됩니다.

X_i	X_{i-1}	X_{i-2}	Operation		Y_i	Y_{i-1}	Y
0	0	0	0+0 = 0		0	0	0
0	0	1	0+A = A		0	1	+1
0	1	0	2A-A = A		0	1	+1
0	1	1	2A+0 = 2A		1	0	+2
1	0	0	-2A+0 = -2A		-1	0	-2
1	0	1	-2A+A= -A		0	-1	-1
1	1	0	0-A = -A		0	-1	-1
1	1	1	0+0 = 0		0	0	0

Radix 4의 계산 방법이며 기본적으로 해당 개념을 바탕으로 곱셈기를 구현할 계획입니다.

d. Register File

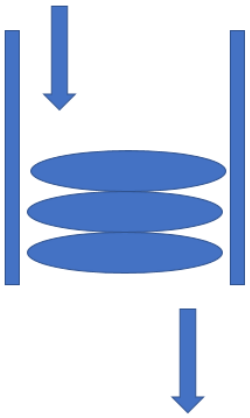
Register file은 앞서 실습한 logic으로 읽고 쓸 수 있는 register들로 구성되어 있는 set을 말하며 mux를 통해 address에 따라서 접근할 수 있는 저장 장치를 뜻한다. Logic을 구성하는 정보로는 32bit register 8개를 포함하고 있으며 Write enable을 통하여 write기능이 활성화하고 Write/Read address에 따라 어느 주소에 값을 읽고 쓸지 조정이 가능한 특징이 있다. 미리 실습한 RF와 동작을 한다. 다음 이미지와 표는 RF를 구현하는데 필요한 정보이다.



구분	이름	설명
Top module	register_file	top module
Sub module	register32_8	8개의 32bit register의 module
	write_operation	write address decode module
	read_operation	Select register using read address

Module 이름	구분	이름	비트 수	설명
register_file	input	clk	1	clock
		reset_n	1	reset which operating at active low register goes to 0
		we	1	write enable
		wAddr	3	write address
		rAddr	.3	read address
		wData	32	write data
	output	rData	32	read data
register32_8	input	clk	1	clock
		reset_n	1	reset which operating at active low register goes to 0
		en	1	register enable
		d_in	32	Data input
	output	d_out0~7	32	Register Data output
write _operation	input	we	1	Write enable
		Addr	3	Write Address
	output	to_reg	8	Selected register enable signal
read _operation	input	from_reg0~7	32	8 registers` data
		addr	3	Read address
	output	Data	32	Data output

e. FIFO



FIFO는 First in first out으로 데이터가 저장되고 지워지는데 일정한 규칙이 있는 구조를 뜻한다. 규칙은 먼저 들어온 data가 있다면 data가 지워지는 작업을 수행할 때 가장 먼저 지워지는 명령을 수행하게 되는 규칙을 뜻합니다. 이번 프로젝트에서 Matrix가 해당 방식을 포함하고 있다. FIFO는 Register file을 포함하고있으며 특징으로 Invalid한 read 또는 write 요청에 대해 FIFO 상태를 변화시키지 않는다. FIFO는 write와 read 요청에 따른 현재상태를 check하고 출력하며 4가지의 feedback(wr_ack, wr_err, rd_ack, rd_err)을 제공한다.

(1) Module configuration

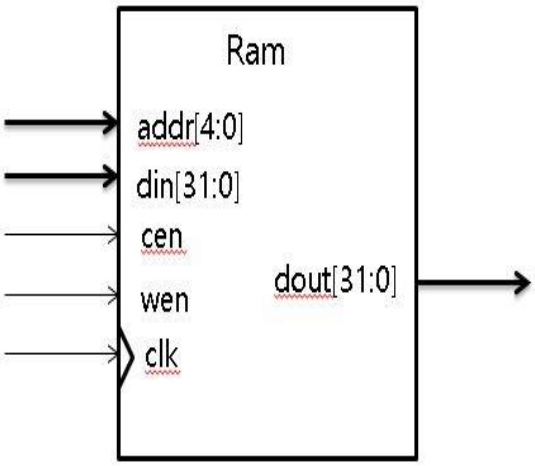
구분	이름	설명
Top module	fifo	FIFO의 top module
Sub module	fifo_ns	Next state module(top module에서 instance)
Sub module	fifo_cal	Calculate address module(top module에서 instance)
Sub module	fifo_out	Output logic module(top module에서 instance)
Sub module	Register_file	Register file module(top module에서 instance)

(2) I/O configuration

Module 이름	구분	이름	비트 수	설명
fifo	input	clk	1-bit	Clock
		reset_n	1-bit	Active low에 동작하는 reset 신호
		rd_en	1-bit	Write enable
		wr_en	1-bit	Write address
		d_in	32-bits	Read address
	output	d_out	32-bits	Read data
		full	1-bit	Data full signal
		empty	1-bit	Data empty signal
		wr_ack	1-bit	Write acknowledge
		wr_err	1-bit	Write error
		rd_ack	1-bit	Read acknowledge
		rd_err	1-bit	Read error
		data_count	4-bits	Data count vector
fifo_ns	input	wr_en	1-bit	Write enable
		rd_en	1-bit	Read enable
		state	3-bits	Current state
		data_count	4-bits	Data count vector
	output	next_state	3-bits	Next state
fifo_cal_addr	input	state	3-bits	Current state
		head	3-bits	Current head pointer
		tail	3-bits	Current tail pointer
		data_count	4-bits	Current data count vector

	output	we	1-bit	Register file write enable
		re	1-bit	Register file read enable
		next_head	3-bits	Next head pointer
		next_tail	3-bits	Next tail pointer
		next_data_count	4-bits	Next data count vector
fifo_out	input	State	3-bits	Current state
		Data_count	4-bits	Current data count vector
	output	Full	1-bit	Data full signal
		Empty	1-bit	Data empty signal
		Wr_ack	1-bit	Write acknowledge
		Wr_err	1-bit	Write error
		Rd_ack	1-bit	Read acknowledge
		Rd_err	1-bit	Read error

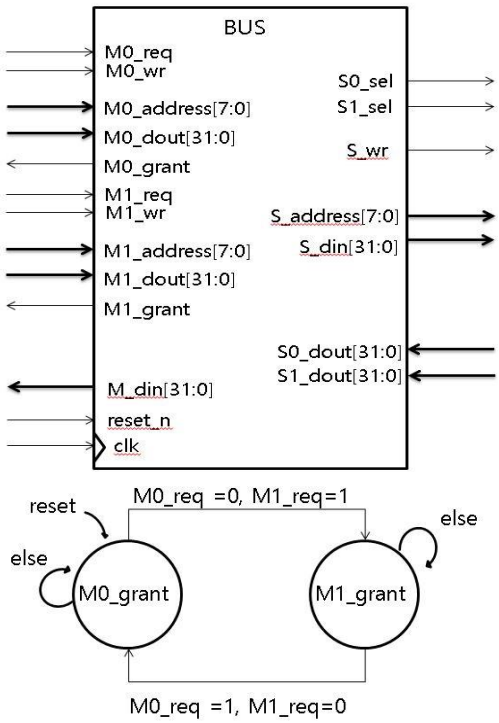
f. Memory



Memory는 address를 기반으로 data를 저장하는 hardware로써 이번 프로젝트에서는 random access memory를 구현하도록 한다. RAM이란 기억된 정보를 읽기도 하고 다른 정보를 기억시킬 수도 있는 메모리로서, 컴퓨터의 주기억장치, 응용 프로그램의 일시적 로딩, 데이터의 일시적 저장 등에 사용되며 해당 프로젝트에서도 데이터를 읽고 쓰기위한 hardware이다.

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

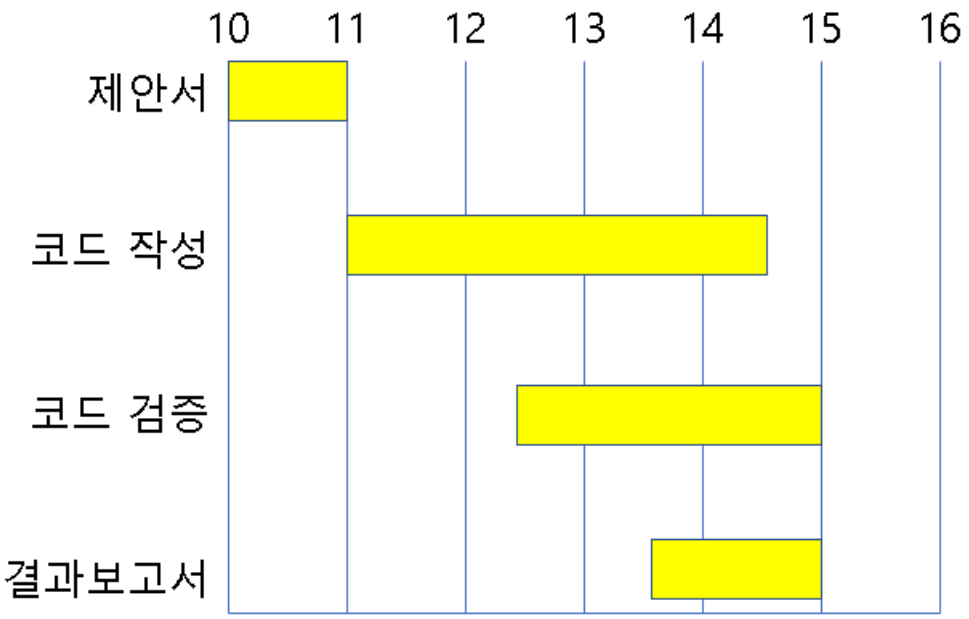
g. BUS



BUS는 값을 변화시키고 특정한 계산을 하는 hardware가 아니라 연결된 hardware간의 data전송이 가능할 수 있도록 도와주는 hardware이다. Bus의 특징으로는 새로운 hardware가 추가되기 쉬우며 가격이 저렴하다는 특징을 가지고 있다. 해당 프로젝트에서 가지는 구조의 특징으로는 2개의 master와 4개의 slave를 가지고 있으며 address는 8bits를 가진다. Data의 bits는 32bits이고 slave들의 가지는 address가 다른데 slave 0 (0x00~0x0F), slave1 (0x10~0x1F), slave2 (0x20~0x3F), slave3 (0x40~0x5F) 사이의 address 를 memory map region으로 가진다.

Direction	Port name	Description
Input	clk	Clock
	reset_n	Active low reset
	M0_req	Master 0 request
	M0_wr	Master 0 write/read signal
	M0_address[7:0]	Master 0 address
	M0_dout[31:0]	Master 1 data output
	M1_req	Master 1 request
	M1_wr	Master 1 write/read signal
	M1_address[7:0]	Master 1 address
	M1_dout[31:0]	Master 1 data output
	S0_dout[31:0]	Slave 0 data out
	S0_dout[31:0]	Slave 1 data out
Output	M0_grant	Master 0 grant
	M1_grant	Master 1 grant
	M1_din[31:0]	Master data input
	S0_sel	Slave 0 select
	S1_sel	Slave 1 select
	S_address[7:0]	Slave address
	S-wr	Slave write and read signal
	S_din[31:0]	Slave data input

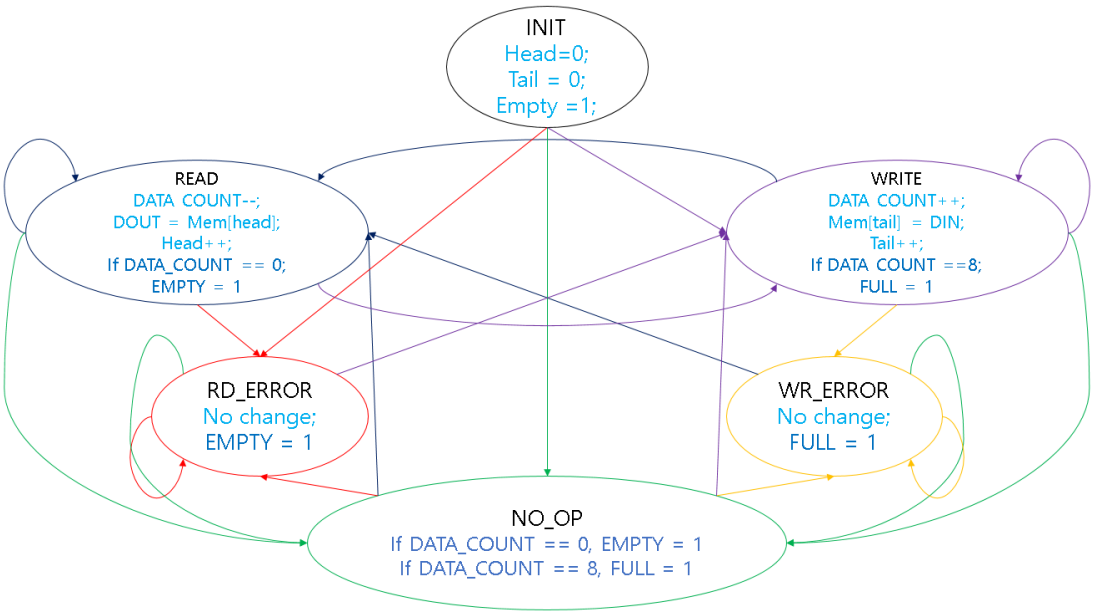
3. Schedule



코드 작성은 필요할 때 마다 계속 수행해야 하므로 기간을 충분히 두고 꾸준히 진행할 생각입니다.

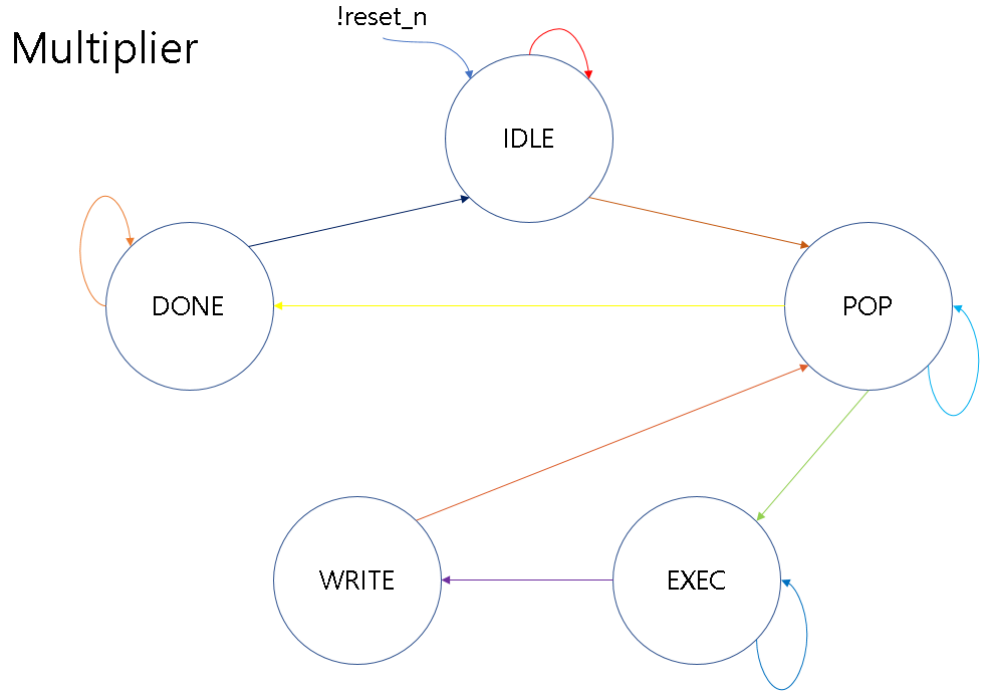
4. State transition diagram

FIFO



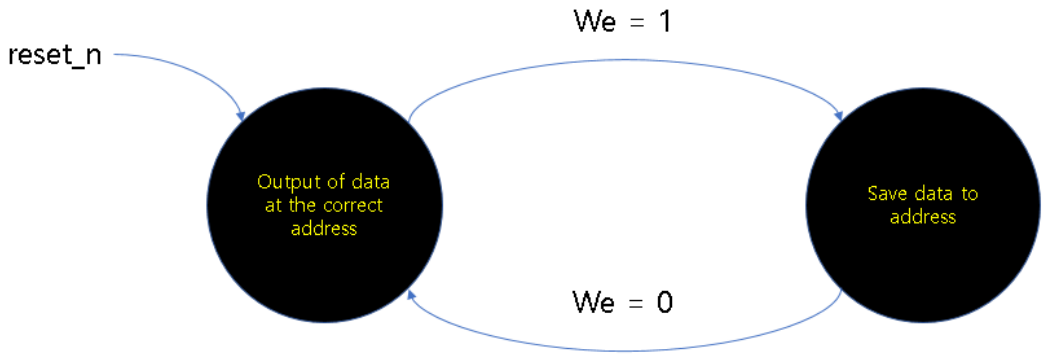
FIFO의 State Diagram입니다. Project에서도 FIFO가 쓰일 예정이며 동작 방식은 미리 설계한 FIFO와 동일합니다.

Multiplier



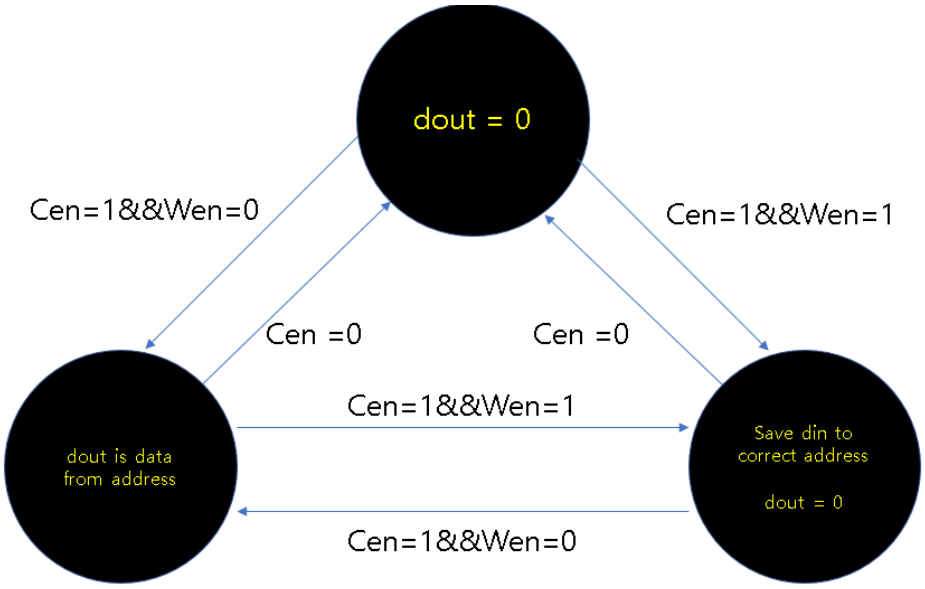
이전에 Multiplier와는 다른 점으로 FIFO의 특징이 합쳐진 Diagram입니다. 상태를 결정하는 요소로는 Multiplier를 다른 좀 더 높은 radix로 설계할 계획이기 때문에 count에 대한 변동 사항이 예상되며 Pop과 Write상태에 관련하여 조금 더 숙지할 필요가 있습니다.

Register File



Write enable signal에 따른 간단한 Register file의 동작을 그린 Diagram입니다. We신호는 And gate와 mux로 연결되어 있어 we신호가 없으면 mux가 정상작동 하지 않아 데이터를 저장할 수 없습니다.

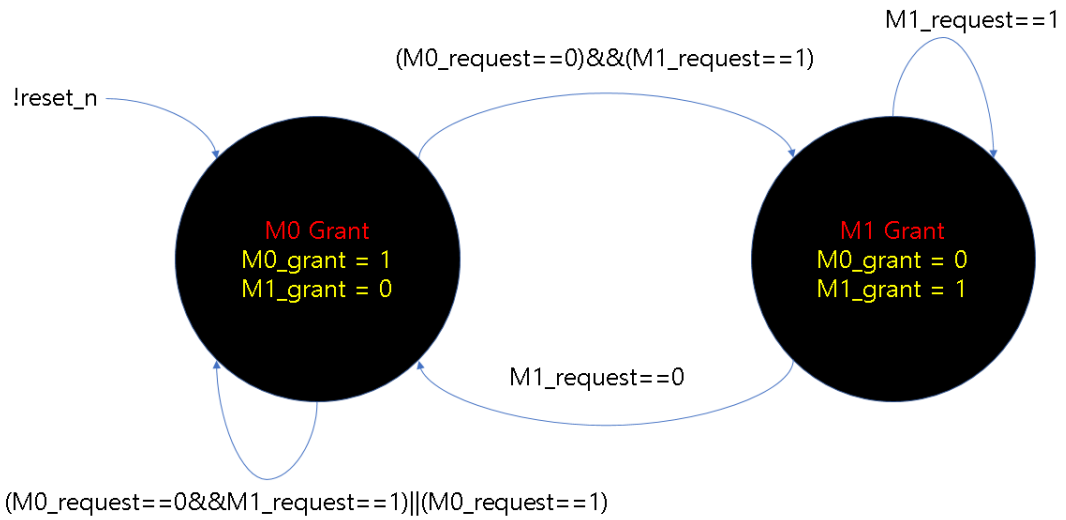
Memory



Cen, Wen신호에 따라 Memory의 동작을 나타낸 diagram입니다. Clk이 반응할 때 마다 cen의 값과 wen의 값을 인식하여 알맞은 동작을 합니다.

Bus

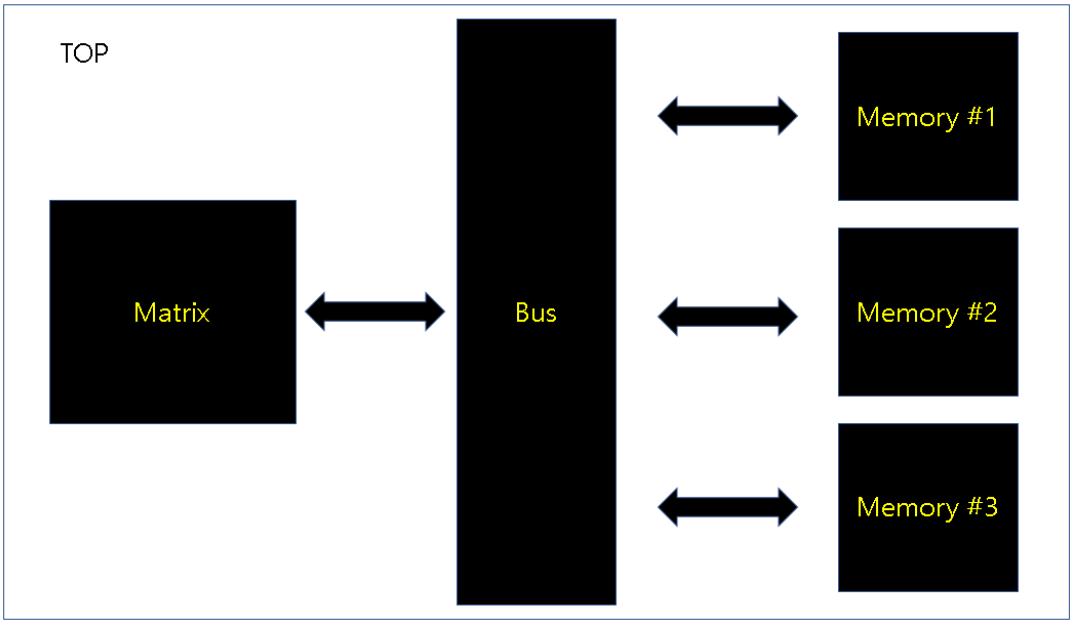
Bus의 master와 slave의 관계는 신호를 처리하는 Arbiter에서 처리하며 결정된다. Arbiter의 diagram은 다음과 같이 나타낼 수 있다.



request신호에 따라 상태가 결정되며 특별한 신호가 없는 경우 M0의 상태에 머물러 있는다. Project에서는 2개의 Master와 4개의 Slave로 나뉘어 있어 수정이 필요하다.

5. Module instance design

제가 생각했을 때 모듈은 가장 크게 Matrix, Bus, Memory로 나눌 수 있다고 생각됩니다. 해당 모듈들을 보기 쉽게 연결하면 아래 그림처럼 표현할 수 있습니다.



동작 방법은 Memory에서 값을 읽어와 Matrix에서 값을 처리 후 다시 Memory에 값을 넣어주는 방식입니다. 데이터가 이동할 때는 항상 Bus가 관여합니다.

6. an expected problem

가장 문제가 되는 부분은 Matrix를 구현하는데 있는 것 같습니다. 다른 logic들은 원래 실습했던 부분에서 port에 개수에 따라 조금만 변형해주면 사용가능 할 것으로 예상되지만 Matrix같은 경우는 내부 구조도 복잡하고 multiplier를 실습하면서 애매한 부분이 많았기 때문에 프로젝트를 진행하는데 있어서 연산장치가 관건이라고 생각됩니다. Matrix만 수월하게 만들면 전체적으로 수월할 것이라고 생각됩니다.

7. Design verification strategy

(1) request값에 따른 Master와 Slave의 정확한 관계 확인

(2) 메모리에서 Matrix로 불러오는 값이 정확하게 넘어오는지 확인

(3) 불러온 값이 Matrix연산을 통하여 정확하게 값이 나오는지 확인

(4) 정확히 연산 된 값이 알맞은 Memory에 저장되는지 확인

(5) 단순 계산과 정확한 저장이 잘되는지 확인된다면 여러가지 Signal(reset 등등...)에 대한 동작이 잘 예외처리 되었는지 확인

(6) 여러가지 test bench를 작성 후 검증