Arithmetic & Logical Computing System

2015722068 남윤창

Abstract

본 프로젝트는 Arithmetic Logical Unit을 이용하여 곱셈, 덧셈, 뺄셈, 및 논리연산을 한다. DMAC(Direct Memory Access Controller), BUS, RAM(Random Access Memory)을 이용하여 메모리 Load/Store System을 구현한다. Testbench가 Processor가 되어 전체 System의 동작을 제어한다.

목차

Introduction …… 2
Project Specification …… 3
RAM ..…… 3
BUS ….…… 3
DMAC …….. 3
ALU ....…… 4
ALU: Multiplier ...……..…… 5
Design Details ….…… 5
RAM ..…… 5
BUS ….…… 7
DMAC …….. 9
ALU ....……..…… 10
Top ……...……..…… 12
Design Verification Strategy & Result…..…….. 13
Conclusions…….…… 18
References…… 18

I. Introduction

본 시스템은 2n개 혹은 이보다 적은 수의 Operand에 대해 n개의 Instruction을 ALU에서 연산을 수행한다. 이후 n개의 결과를 Result RAM에 저장한다. 자세한 수행 순서는 다음과 같다.

시스템이 시작되면, Testbench는 BUS master가 되어 Operand를 RAM #1에 그리고 Instruction은 RAM #2에 저장한다. RAM은 최대 64개의 32bit data를 저장할 수 있다.

Testbench는 DMAC에게 Source address로부터 Destination address로 Operand, Instruction을 Copy하도록 OPERATION_START register를 이용하여 시작 명령을 내린다. DMAC가 ALU에게 전달을 해주어야 한다. 따라서 Source address는 RAM#1, RAM#2, Destination address는 ALU가 된다. ALU 내부에는 Operand를 저장하는 capacity 16의 Register File이 존재한다. Instruction을 저장하기 위한 capacity 8의 FIFO(First-in, First-out)이 존재한다.

OPERATION_START신호를 받은 DMAC은 BUS master가 되어 Source address로부터 한 번에 32bit data씩 읽어서 Destination address에 copy한다.

DMAC이 전송을 모두 마치면 DMAC내부의 INTERRUPT Register data가 1이 된다. 만약 Interrupt-driven Method라면 interrupt 신호를 1로 testbench에게 내보낸다.

DMAC의 완료 신호를 받은 Testbench는 다시 BUS master가 되어 DMAC의 INTERRUPT Register에 1값을 전송하여 interrupt를 clear한다.

연산에 필요한 모든 Operand, Instruction이 전송될 때까지 DMAC은 data이동 과정을 반복한다.

모든 Operand, Instruction 전송이 완료된 후, Testbench는 BUS master가 되어 ALU에 OPERATION_START Register를 이용하여 시작 명령을 내린다.

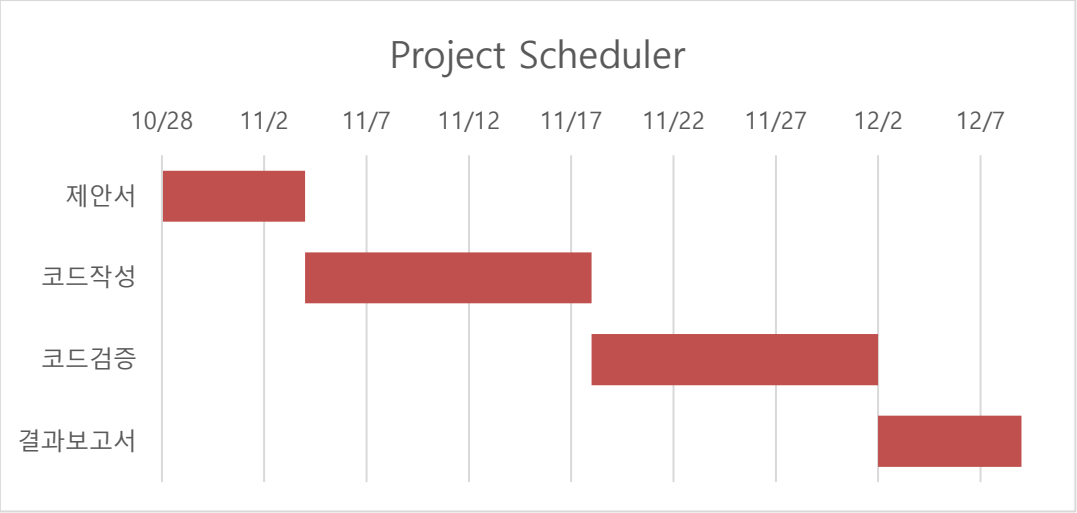
OPERATION_START신호를 받은 ALU는 내부에 있는 FIFO에서 Instruction을, Register File에서 Operand를 꺼내서 각 연산을 Result FIFO에 저장한다. 곱셈을 제외한 모든 연산은 32bit를 64bit로 sign extension하여 연산한다. 64bit 결과는 2cycle에 걸쳐 Result FIFO로 삽입된다. Result는 [63:32], [31:0]순서로 삽입된다.

Instruction FIFO가 empty가 되면, ALU는 INTERRUPT Register를 1로 만든다. Interrupt-driven Method의 경우 interrupt를 Testbench에 직접 내보낸다.

Testbench는 ALU interrupt를 clear하고, DMAC에게 ALU Result FIFO의 값을 RAM#3으로 옮기도록 Descriptor를 추가해 주고 OPERATION_START Register를 이용하여 명령을 내린다.

모든 전송이 완료되면 DMAC은 INTERRUPT Register를 1로 만들어 발생시킨다.

Testbench는 DMAC의 interrupt를 clear하고 시스템은 종료된다.



본 프로젝트의 일정은 다음과 같다. DMAC 1주, ALU 1주를 소모하였고, 이에 대한 Cycle관리를 하는 것 및 여러가지 Test Case를 검증하는 것에 대한 시간을 2주를 투자하였다.

II. Project Specification

1) RAM

RAM은 Random Access Memory이다. Random Access라는 것은 임의의 주소가 주어질 때 똑 같은 시간으로 접근이 가능하다는 의미이다. 예를 들어 본 프로젝트의 RAM #1은 0x0200 ~ 0x023F의 Address Map을 갖는다. 0x0200 ~ 0x023F 모두 1cycle만에 접근하여 write하고 1cycle만에 read하도록 구현하였다.

RAM은 본 프로젝트에서 총 3개가 쓰인다. 이는 모두 Slave Component이다. 따라서 각 RAM이 선택되었는지를 판별하기 위한 Chip Enable Signal과 R/W를 판단하기 위한 Write Enable Signal이 존재한다. For구문과 initial 구문을 사용하여 모든 Register를 0으로 초기화하고 시작한다. 이후 Rising Edge마다 Write Enable Signal과 Chip Enable Signal에 따라 값을 읽거나 쓴다.

2) BUS

BUS는 Testbench를 M0, DMAC을 M1으로 하여 총 2개의 Master Component를 갖는다. DMAC을 S0, ALU를 S1, RAM #1, #2, #3을 각각 S2, S3, S4로 하여 총 5개의 Slave Component를 갖는다. Slave 0 은 0x0000 ~ 0x001F 사이의 값을 Address로 갖는다. Slave 1은 0x0100 ~ 0x011F, Slave 2는 0x0200 ~ 0x023F, Slave 3은 0x0300 ~ 0x033F, Slave 4는 0x0400 ~ 0x043F를 Address로 갖는다.

본 프로젝트에서 구현하는 BUS는 Simple bus로 1개의 Master와 1개의 Slave만 동시에 소통이 가능하다. 즉 DMAC이 Master권한을 갖고 있다면 Testbench는 Master권한을 갖지 못한다. 또한 DMAC이 RAM #2를 Access중이라면 다른 Slave Component는 Select되지 않은 상태를 뜻한다.

Master Component는 BUS를 통해 data를 Transfer하고자 하는 경우, Request Signal을 BUS에 보내 grant signal이 1이 된 경우를 Master권한을 얻은 것으로 간주한다. 본 프로젝트에서 구현한 Simple BUS는 Greedy하게 작동한다. 즉 한 개의 Component가 Grant를 놓지 않는 경우 해당 Component가 계속 Grant를 유지하는 알고리즘이다. 자세한 내용은 아래 Design Details의 BUS FSM Diagram을 통해 확인할 수 있다.

BUS는 총 5개의 Slave Component와 연결된다. 이는 Master Component가 입력하는 Address를 BUS가 Decoding하여 특정 Slave를 select하게된다. Address는 총 16bits이다. 16bits중 상위 8bits를 Base Address라고 부른다. BUS는 Base Address를 구분하여 5개의 Slave중 Master가 원하는 Slave에 Select signal을 전달한다. 해당 Select Signal을 통해 Slave가 내보내는 Output data를 다시 Master에게 전달한다.

모든 Master가 Request Signal을 보내지 않는다면 M0, Testbench가 Master권한을 얻도록 설계하였다.

BUS는 오로지 각 Peripheral Component들에게 16bits Address와 32bits data Transfer 통로로만 통신한다. 따라서 System의 모든 요청은 Address와 data로 이루어진다.

3) DMAC

DMAC이란 Direct Memory Access Controller이다. Processor 이외의 Device가 Memory와 I/O Device간의 data전송을 제어하는 역할을 한다. 즉 Processor가 data transfer권한을 DMAC에게 부여하여 processor는 그 시간동안 다른 일처리를 할 수 있도록 하여 효율성을 높이는 hardware이다.

DMAC은 1개의 Slave Component와 Master Component를 갖는다. Testbench와 소통할 경우는 Slave Component가 활성화되고, DMAC이 data transfer를 하는 경우는 Master Component가 활성화되도록 하였다.

DMAC은 data transfer를 하기 위한 Address를 capacity 16 FIFO를 통해 저장하고 있다. Source Address, Destination Address, Data Size를 각각의 FIFO에 저장한다. 즉 DMAC은

1개의 Slave Component, 1개의 Master Component, 3개의 capacity 16 FIFO로 구성된다. 자세한 내용은 아래의 Block Diagram에서 확인할 수 있다.

Testbench는 DMAC에게 Descriptor를 전송한다. BUS를 통해 Base Address로 DMAC을 선택하고 Offset Address를 통해 DMAC내부의 Register에 접근한다. 0x03, 0x04, 0x05를 통해 Descriptor정보를 전송하고 이를 FIFO에 Enqueue하기 위해 0x06을 Offset Address로 갖는 DESCRIPTOR_PUSH에 data 1을 쓴다.

Testbench가 DMAC의 OPERATION_START Register에 0x00000001값을 입력하여 시작 신호를 준다. DMAC은 3개의 FIFO에서 REQ = {Source Address, Destination Address, Data Size}를 POP한다. DMAC은 BUS에게 Request Signal을 보낸다. Request Signal을 보낸 후 Grant를 받는다면 Source Address에서 Data를 읽는다. BUS를 통해 받아온 data는 다시 Destination Address로 보내진다. 이를 Data Size만큼 반복한다. FIFO에 남은 Descriptor가 없다면 DMAC은 INTERRUPT Register값을 1로 만든다. DMAC의 INTERRUPT_ENABLE Register가 1이라면 interrupt-driven method로 testbench에 직접 interrupt를 전송한다. INTERRUPT_ENABLE Register가 0이라면 polling method로 testbench가 INTERRUPT Register를 직접 읽어 1인지 아닌지 확인하여 transfer완료 여부를 판별한다. 본 프로젝트에서는 두 가지 모두 구현한다.

본 프로젝트에서는 Data Size는 1로 고정이다. Addressing Mode를 구현하지 않아 모든 Descriptor의 data전송은 한번에 32bits씩 이루어진다.

DMAC Slave의 PUSH_DESCRIPTOR에 신호가 들어온 경우, 만약 FIFO가 full이라면 wr_err신호가 발생한다. DMAC은 이 신호를 읽어 FAULT state로 전환한다. 이 정보는 0x08을 offset address로 갖는 DMA_STATUS Register를 읽어 확인할 수 있다. 또한 OPERATION_START 신호가 들어온 경우 FIFO가 empty인 경우도 동일하게 FAULT state가 된다. 자세한 내용은 아래의 FSM Diagram에서 다룬다.

4) ALU

ALU는 본 프로젝트의 메인 component이다. ALU는 두 개의 입력을 이용해 Opcode에 해당하는 연산을 이용해 결과를 도출하는 hardware이다. 본 프로젝트의 곱셈을 제외한 64bits 연산을 한다. 곱셈은 32bits연산을 한다. 곱셈은 아래에서 자세하게 설명 되어있다.
ALU는 DMAC에게서 전달받은 Instruction을 ALU_STATUS가 IDLE상태일 경우에만 INSTRUCTION Register와, Instruction을 저장하는 capacity 8 FIFO에 저장한다. ALU는 DMAC에게서 전달받은 Operand를 offset address를 통해 ALU 내부의 Register File에 저장한다.

DMAC이 모든 정보들을 전달 후 Testbench는 ALU의 OPERATION_START에 1을 Write하여 시작 신호를 준다. ALU는 Instruction FIFO에서 값을 Pop한후 이를 Opcode, Operand A address, Operand B address, Shift amount로 나눈다. Operand A, B값을 Register File에서 가져온 후 이를 Opcode에 알맞게 연산을 수행한다. 만약 Opcode가 0 이라면 NOP이므로 아무런 행동을 하지 않은 채 다음 Instruction을 수행한다. Opcode가 F라면 곱셈이므로 Booth Multiplication Module로 값을 전달해 준 후 곱셈이 끝나기를 기다린다. 이외의 산술, 논리 연산은 모두 1cycle만에 결과값이 나온다. NOP를 제외한 모든 연산이 끝난 경우 Result FIFO에 [63:32] 부터 Enqueue를 진행한다. 이후 [31:0]를 Push하여 총 2cycle만에 Data를 result FIFO에 저장한다. 모든 Instruction을 수행하여 Instruction FIFO가 empty상태가 되면 ALU INTERRUPT Register를 1로 바꾼다. Interrupt-driven method의 경우 DMAC과 마찬가지로 직접 testbench로 interrupt신호를 전달하고, polling method의 경우 INTERRUPT Register를 읽어 연산이 종료되었음을 확인한다.

ALU는 모두 16가지의 연산을 제공한다. LSL(Logical Shift Left), LSR(Logical Shift Right), ASR(Arithmetic Shift Right), ADD(Carry Look-ahead Adder), SUB(Carry Look-ahead Adder), MUL(radix-2 Booth Multiplier), 그리고 AND, OR, XOR, XNOR등 논리연산을 지원한다.

5) ALU: Multiplier

본 프로젝트에는 Radix-2 Booth Multiplication을 구현하였다. Radix-2 Booth Multiplication은 Binary Multiplier와 동일하게 N bit의 경우 N cycle을 소모한다. 따라서 32bits x 32bits이므로 총 32cycle을 소모하여 64bits의 결과값을 도출할 수 있다.

Xi	Xi-1	Operation	Description	Yi
0	0	Shift Only	String of zeros	0
0	1	Add and Shift	End of a string of ones	1
1	0	Sub and Shift	Beginning of a string of ones	-1
1	1	Shift Only	String of ones	0

Radix-2 Booth Multiplication의 Multiplier의 bit pattern을 통해 Operation을 수행한다. Bit pattern에 대한 연산 수행은 위 표에 따른다. 자세한 내용의 FSM Diagram 및 Pin Description은 아래에 추가 설명한다.

본 프로젝트에서 Multiplier는 ALU Calculation block에서 생성한 mul_start signal을 받아서 연산을 시작하고 연산이 끝남과 동시에 mul_done signal을 ALU Calculation block으로 전달하여 연산이 끝났음을 알린다. Multiplier가 진행되는 동안 ALU는 아무것도 하지 않고 기다린다. Multiplier는 64bits Result를 ALU Calculation에게 전달하여 Result FIFO로 Push하도록 한다.

III. Design Details

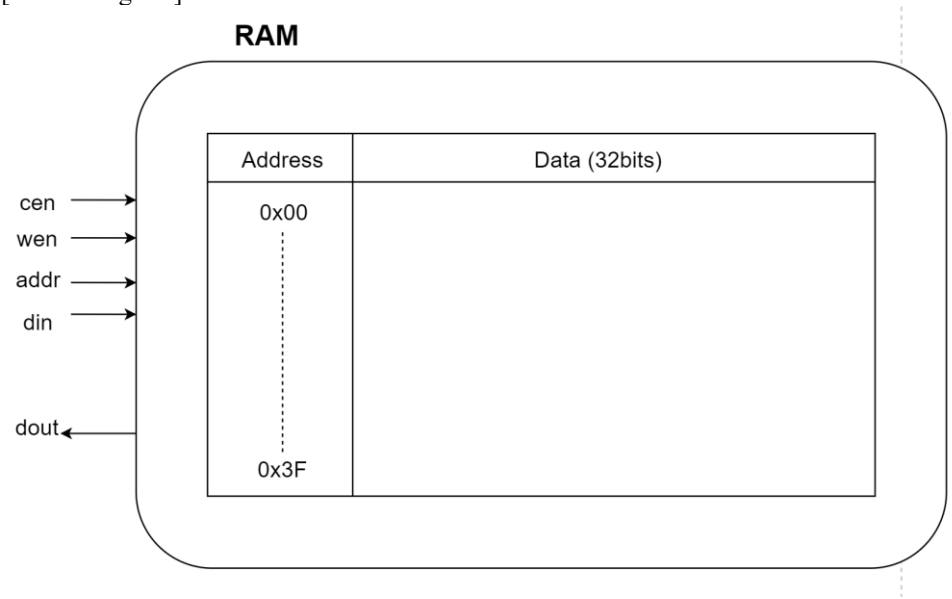
1) RAM

[Pin Description - RAM]

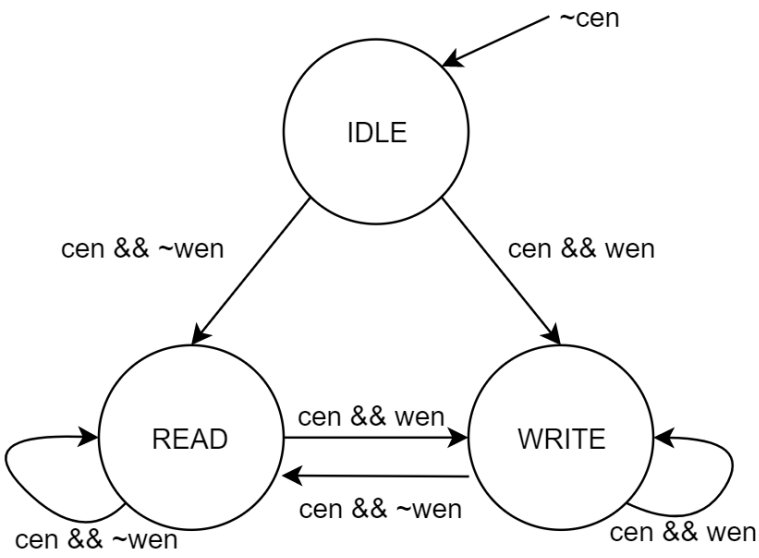
Direction	Port name	Description
Input	Clk	Clock
	Cen	Chip Enable
	Wen	Write Enable
	Addr[15:0]	Address
	Din[31:0]	Data in
Output	Dout[31:0]	Data out

RAM의 pin description은 다음과 같다. Cen은 BUS에서 전달해주는 slave select signal과 연결된다. Wen은 BUS에서 전달하는 m_wr과 연결된다. Addr과 Din은 BUS의 m_addr, m_dout과 연결된다. Output dout은 BUS의 s_dout과 연결된다.

[Block Diagram]



[FSM Diagram - RAM]



State Diagram에 영향을 주는 pin은 cen, wen뿐이다. 따라서 cen이 1이라면 R/W상태가 활성화되고 wen이 1이라면 W, wen이 0이라면 R상태가 되는 원리이다.

2) BUS

[Pin Description - BUS]

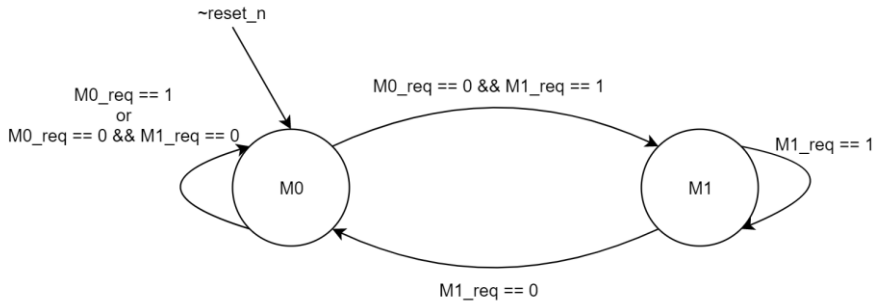
Direction	Port name	Description
Input	Clk	Clock
	Reset_n	Asynchronous active low reset
	M0_req	Master 0 request
	M0_wr	Master 0 R/W
	M0_addr[15:0]	Master 0 address
	M0_dout[31:0]	Master 0 data output
	M1_req	Master 1 request
	M1_wr	Master 1 R/W
	M1_addr[15:0]	Master 1 address
	M1_dout[31:0]	Master 1 data output
	S0_dout[31:0]	Slave 0 data output
	S1_dout[31:0]	Slave 1 data output
	S2_dout[31:0]	Slave 2 data output
	S3_dout[31:0]	Slave 3 data output
	S4_dout[31:0]	Slave 4 data output
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
	S4_sel	Slave 4 select
	S_addr[15:0]	Slave R/W
	S_din[31:0]	Slave data input

총 2개의 Master Component에 해당하는 M0, M1 Pin과 5개의 Slave에 해당하는 s0~s5 pin들이 존재한다. Block Diagram은 아래에 자세하게 나와있다.

[Base Address Map - BUS]

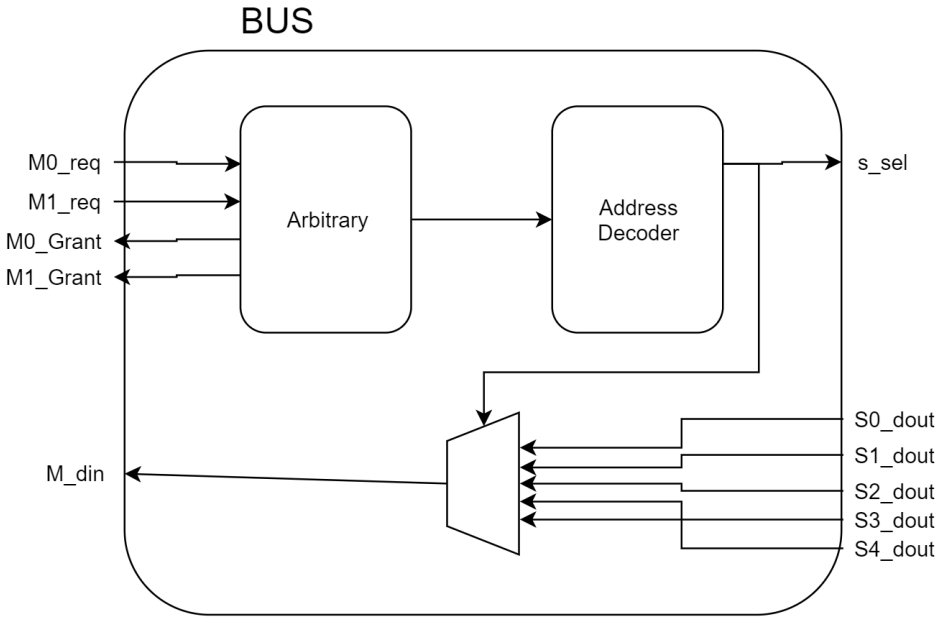
Component	**Address Region**
0x0000 ~ 0x001F	DMAC
0x0100 ~ 0x011F	ALU
0x0200 ~ 0x023F	RAM (Instruction)
0x0300 ~ 0x033F	RAM (Operand)
0x0400 ~ 0x043F	RAM (Result)

[FSM Diagram - BUS]



다음은 BUS의 Master Component를 선택하기 위한 Arbitrary의 FSM Diagram이다. Greedy하게 Master권한을 받기 때문에 한쪽의 request가 계속 1이라면 계속 Master를 쥐고 있도록 설계하였다. 두 Master모두 request를 보내지 않는다면 testbench에 해당하는 Master 0 가 grant를 받도록 설계하였다.

[Block diagram - BUS]



Output s_sel은 총 5개의 slave에 대한 select bit를 집약적으로 표현하였다.

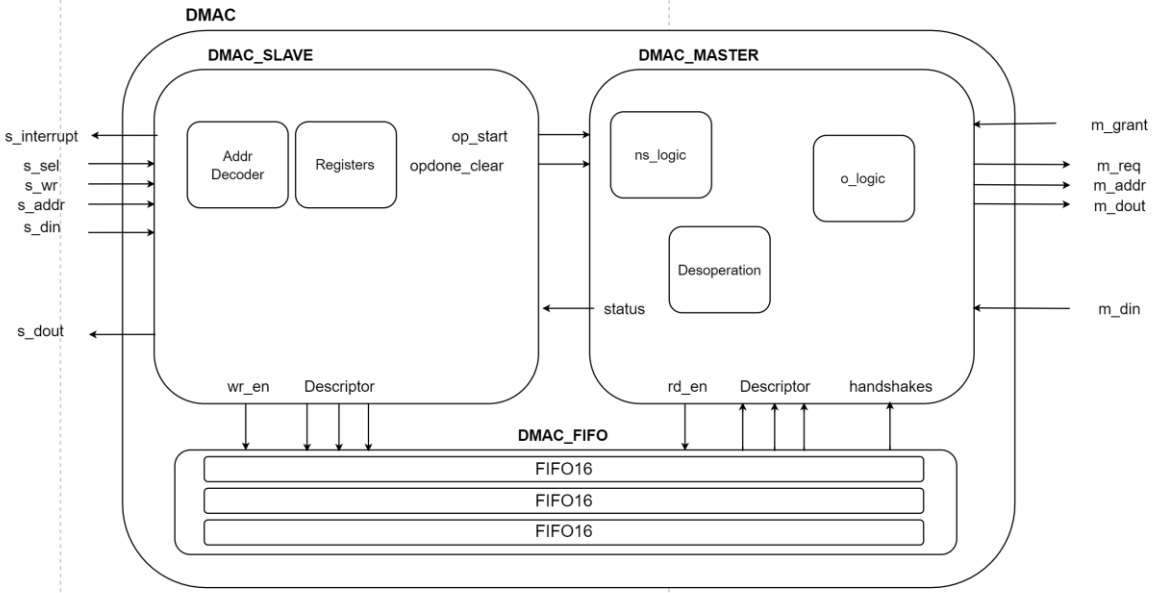
3) DMAC

[Pin Description - DMAC]

Direction	Port name	Description
Input	Clk	Clock
	Reset_n	Asynchronous active low reset
	M_grant	If grant is high, it means it can use BUS as Master Component
	M_din[31:0]	Data input (Master)
	S_sel	Select (Slave)
	S_wr	R/W (Slave)
	S_addr[15:0]	Address (Slave)
	S_din	Data input (Slave)
Output	M_req	Grant Request
	M_wr	R/W (Master)
	M_addr[15:0]	Address (Master)
	M_dout[31:0]	Data output (Master)
	S_dout[31:0]	Data output (Slave)
	S_interrupt	Done signal to testbench

DMAC은 Slave와 Master Interface를 모두 갖는다. 따라서 이에 해당하는 BUS Interface가 모두 존재하므로 위와 같은 Pin description을 갖는다.

[Block Diagram - DMAC]

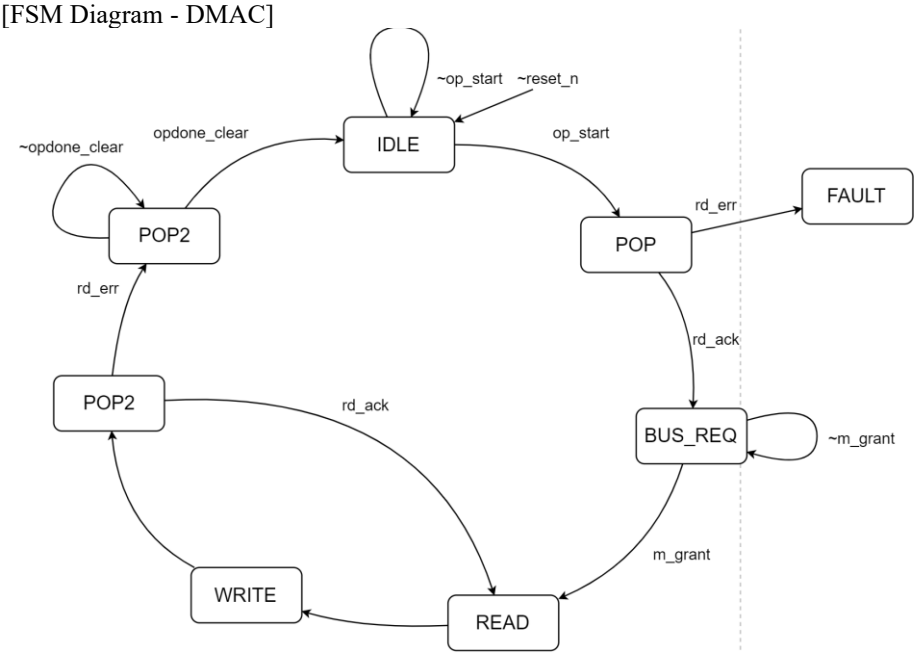


DMAC 내부의 Block Diagram은 다음과 같다. 크게 DMAC_SLAVE, DMAC_MASTER, DMAC_FIFO로 이루어진다. DMAC_SLAVE에서는 BUS로부터 받아온 정보를 Address Decoder를 이용하여 Register로 정리하여 R/W를 실행하고 이를 FIFO에 저장하는 행동을 한다.

DMAC_FIFO는 3개의 capacity 16 FIFO를 갖는다. 이를 하나로 Descriptor라고 표현하며 이는 각각 Source address, Destination Address, Data Size를 갖는다. 이는 Slave를 통해 들어온 정보를 저장하며 Master에게 Pop한다.

DMAC_Master는 DMAC이 주체가 되어 data transfer하는 Module이다. 이는 FSM Diagram을 내포하고 있는 실질적인 Module이다. 타 Slave에서 Data를 read하여 또 다른 Slave에 Data를 Write하는 역할을 한다.

[FSM Diagram - DMAC]



DMAC의 FSM Diagram은 다음과 같다. Operation Mode를 구현하지 않았기 때문에 READ Write는 각각 1cycle만 소모하게 된다. POP은 Grant가 0인경우의 POP, POP2는 Grant가 1인경우의 POP을 뜻한다. 이는 각각 다르게 행동한다. POP2에서 m_addr은 system에 포함되지 않는 주소를 access하여 의도에 맞지 않는 행동을 하지 않도록 설계했다.

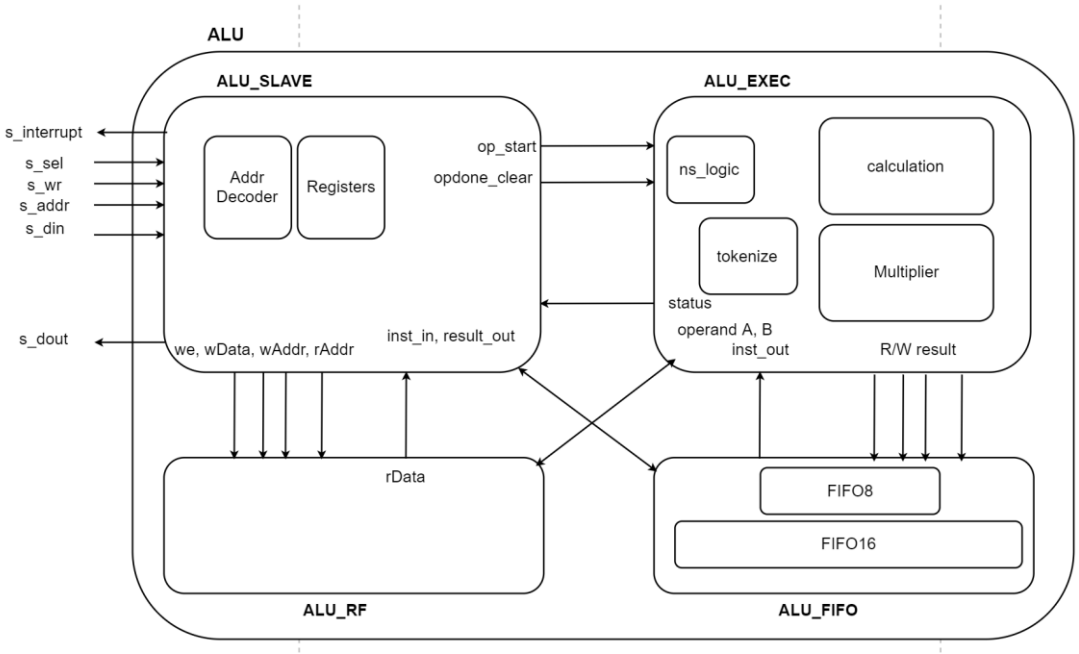
4) ALU

[Pin Description - ALU]

Direction	Port name	Description
Input	Clk	Clock
	Reset_n	Asynchronous active low reset
	S_sel	Select (Slave)
	S_wr	R/W (Slave)
	S_addr[15:0]	Address (Slave)
	S_din	Data input (Slave)
Output	S_dout[31:0]	Data output (Slave)
	S_interrupt	Done signal to testbench

ALU는 Slave Interface만 포함한다. 따라서 Pin description이 위와 같다.

[Block Diagram - ALU]

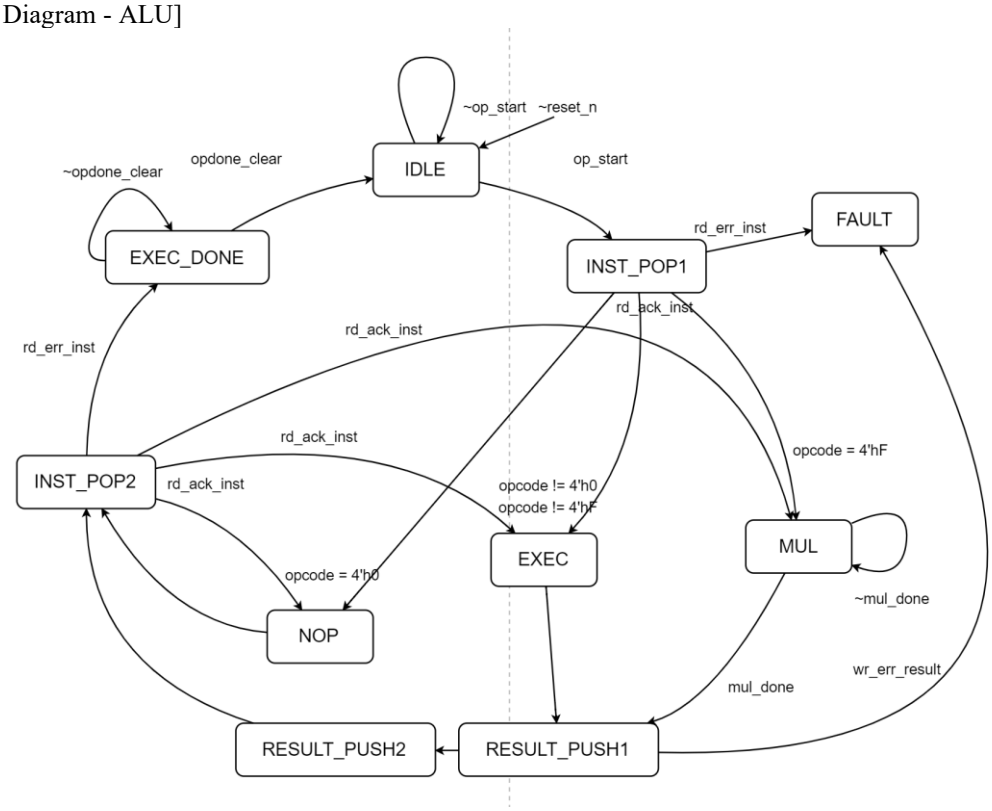


ALU의 Block Diagram은 위와 같다. 크게 4가지로 구성된다. ALU_SLAVE는 Slave Interface로 BUS와 통신한다. DMAC이 전달하는 Instruction을 받고 연산이 종료된다면 interrupt signal을 출력하는 역할을 한다.

ALU_EXEC는 실제로 연산을 하는 Module이다. ALU_EXEC안의 tokenize는 opcode를 분리하여 operand를 불러오고 그에 알맞은 값을 calculation으로 넘기는 역할을 한다. Calculation은 해당 opcode에 따라 알맞은 연산을 수행한다. 다만 opcode가 4'hF인 경우 곱셈을 수행하기위해 mul_start, multiplicand, multiplier값을 Multiplier Module로 전달한다. Multiplier는 Radix-2 Booth Multiplier이다. 따라서 총 32cycle이 소모되며 연산이 완료되면 Mul_done signal을 Calculation Module에 전달한다.

ALU_RF는 DMAC으로부터 전달받은 Operand들을 저장하고 있다. Capacity는 16이다. ALU_FIFO는 capacity 8 instruction FIFO와 capacity 16 Result FIFO를 내포한다. 이는 각각 독립적으로 작동한다. ALU Result FIFO는 63:32 – 31:0 순으로 enqueue, dequeue된다.

[FSM Diagram - ALU]



INST_POP1은 FAULT state를 판별하기 위한 장치이다. Empty인 경우 FAULT state로 넘어가도록 설계하였다. INST_POP2는 empty라면 EXEC_DONE으로 state를 변화시킨다. 이를 통해 POP signal을 생성하지 못하도록 설계하였다. NOP는 연산을 하지 않으므로 RESULT_PUSH1, RESULT_PUSH2로 가지 않고 바로 다음 Instruction을 POP하도록 설계하였다. RESULT_PUSH1, 2는 각각 64bits result를 2cycle에 나누어 32bits씩 저장하는 state이다. MUL은 Multiplier이다. 이는 따로 연산을 하도록 진행하였다. EXEC는 모든 연산이 1cycle만에 종료되므로 1cycle후에 바로 RESULT_PUSH state로 넘어가도록 설계하였다. RESULT_PUSH1에서 또 push를 하려고 하는 경우 wr_err신호를 발생시켜 FAULT state로 넘어간다.

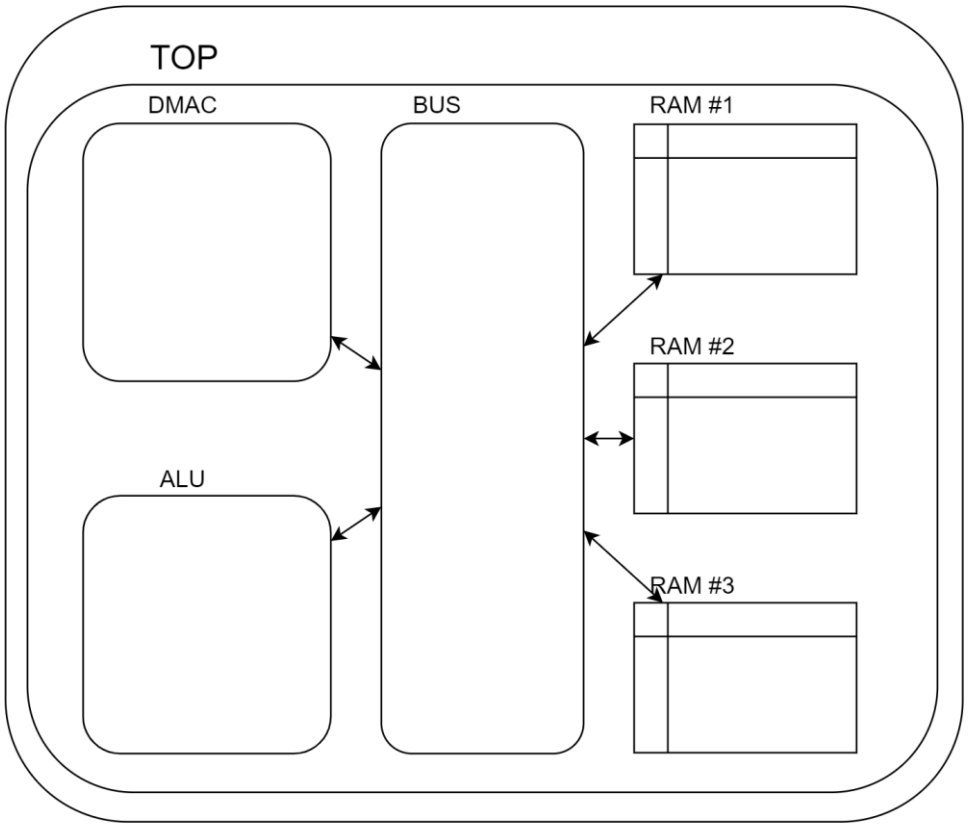
5) Top

[Pin Description - Top]

Direction	Port name	Description
Input	Clk	Clock
	Reset_n	Asynchronous active low reset
	M0_req	BUS request
	M0_wr	R/W
	M0_addr	Address
	M0_dout	Data input
Output	M_grant	Master 0 grant
	A_interrupt	ALU interrupt
	D_interrupt	DMAC interrupt
	M_din[31:0]	Data input

앞서 구현한 각 Component들을 합친 전체 System의 Pin Description이다. Testbench에 해당하는 Pin들이 주를 이룬다. ALU와 DMAC의 종료 신호를 알리는 interrupt wire는 각각 component와 직접 연결되는 특징을 갖는다.

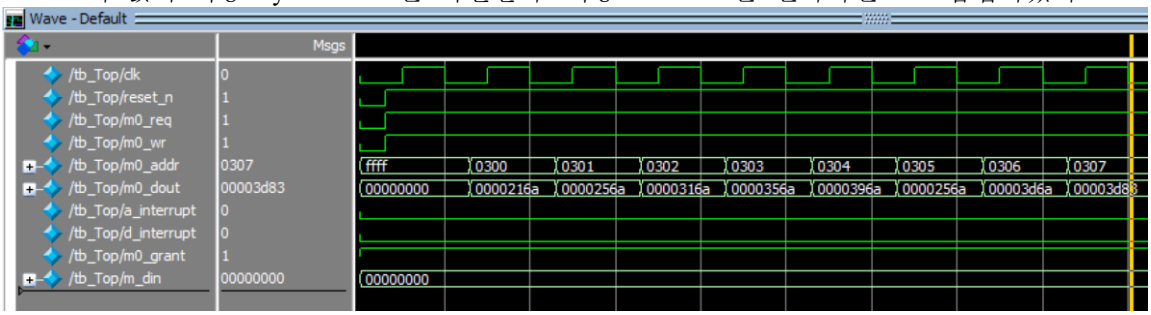
[Block Diagram - Top]



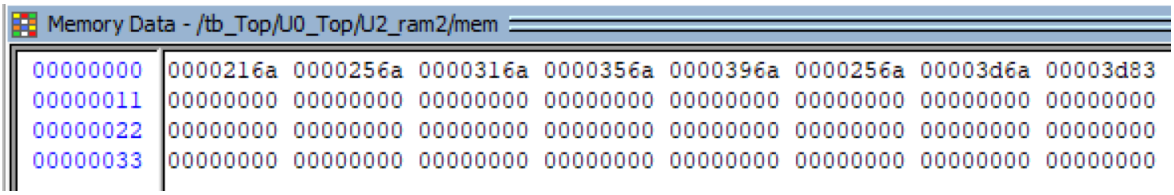
위 Diagram의 Top은 Testbench에서 모든 동작을 control한다.

IV. Design Verification Strategy and Results

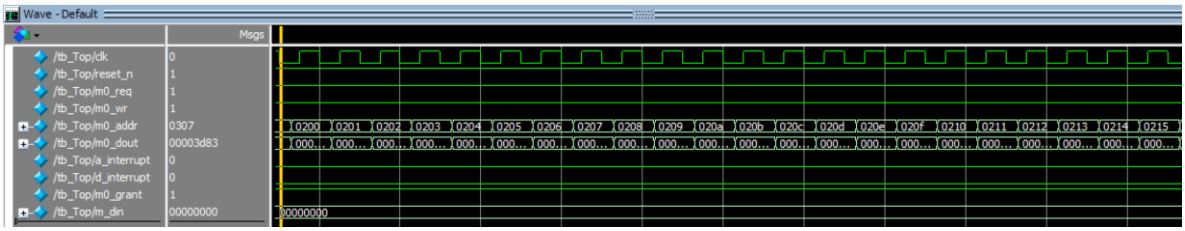
각 모듈을 구현하여 각각의 Simple Testbench를 작성하여 결과를 확인한 후 Top Module testbench에 얹어 최종 System Flow를 확인한다. 최종 testbench를 결과화면으로 삽입하였다.



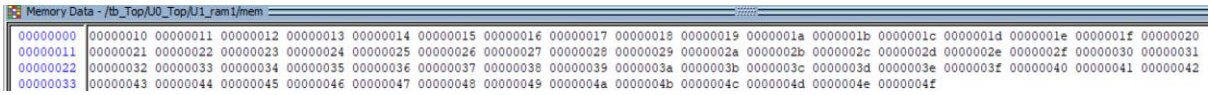
위 waveform은 Testbench가 Master권한을 갖고 RAM #2에 Instruction 값을 넣어준 모습이다.



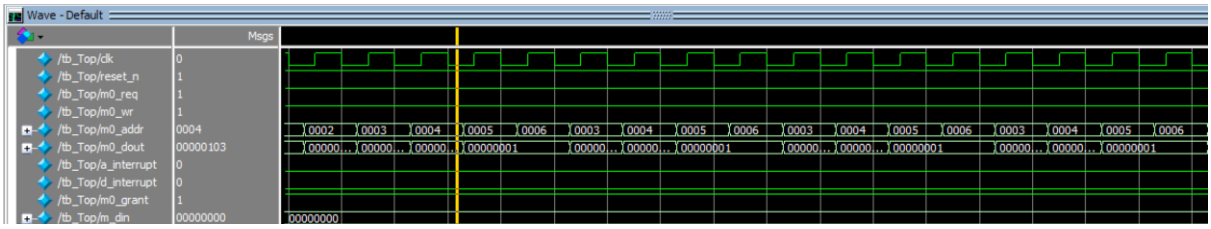
다음과 같이 결과가 제대로 RAM #2에 저장되는 것을 확인했다.



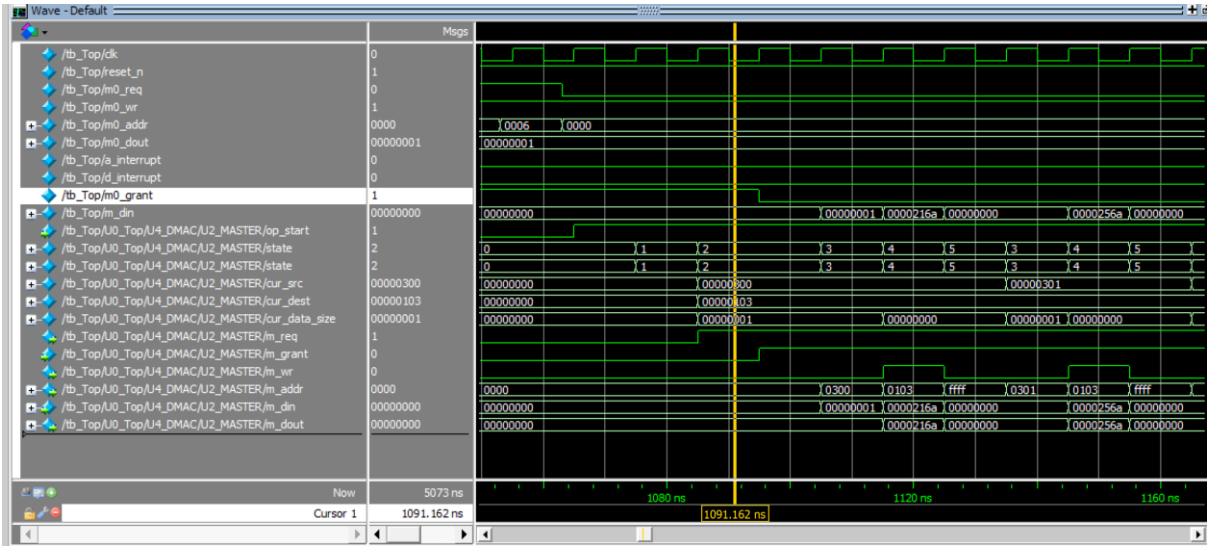
위 waveform은 Testbench가 Master권한을 갖고 RAM #1에 Operand값을 대입하였다.



RAM #1에 제대로 10~4F의 값이 모두 대입되는 것을 확인하였다.

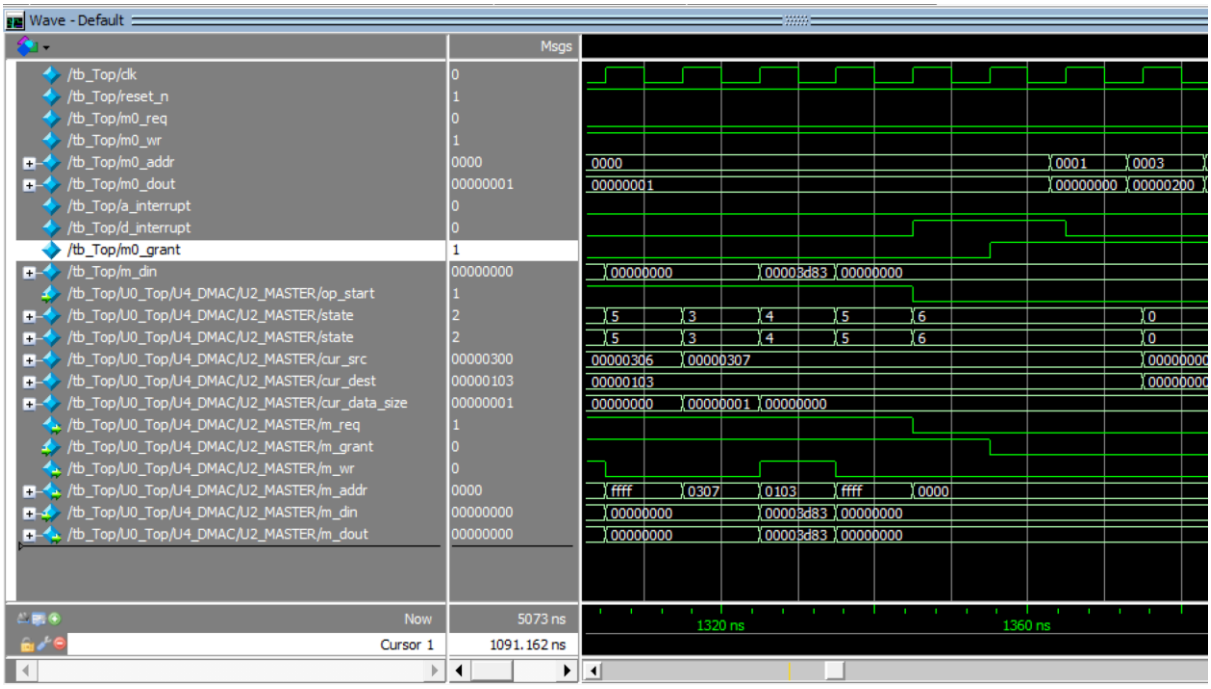


DMAC에 Descriptor들을 넣어주는 모습이다. 우선 Interrupt-driven Method를 검증하기 위해 INTERRUPT_ENABLE Register에 1을 넣어주었다.



DMAC의 OPERATION_START값을 1로 변경한 결과, op_start signal을 1로 획득하는 것을 확인했다. 위 FSM에 따라 2(BUS_REQ)에서 m_req signal이 1이 되어 다음 cycle에 m1_grant를 획득하는 것을 확인했다. 3(READ), 4(WRITE)를 통한 data transfer를 m_wr, m_din, m_dout을

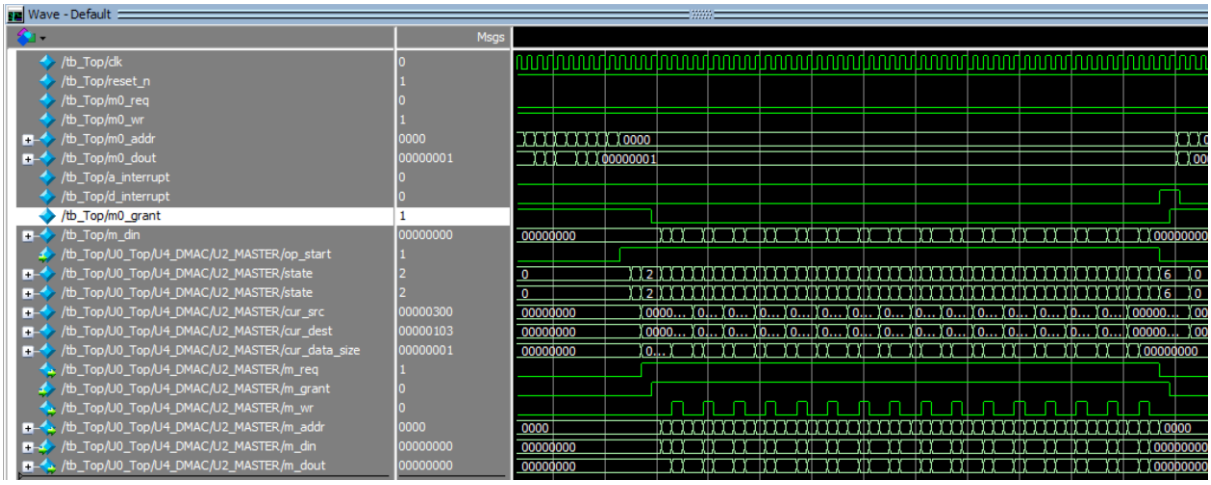
통하여 확인했다. 5(POP2)를 통하여 새로운 descriptor를 pop하는 동안 address가 이상한 곳을 접근하지 않도록 ffff값을 넣어주어 NOP의 효과를 보는 것을 확인했다.



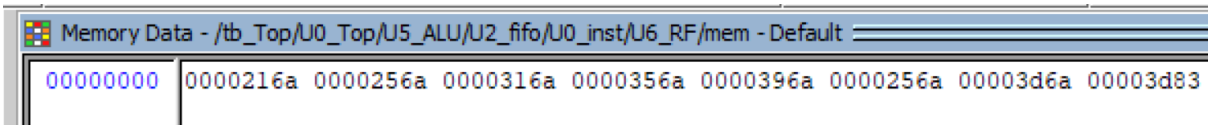
DMAC의 data transfer가 종료되면 d_interrupt signal이 1이 되고, 1cycle후에 testbench가 m0_grant를 획득하여 DMAC의 interrupt signal을 0로 만들어 주는 것을 확인하였다.

```
while(d_interrupt != 1'b1) begin                    //interrupt-driven method
#10;
end
```

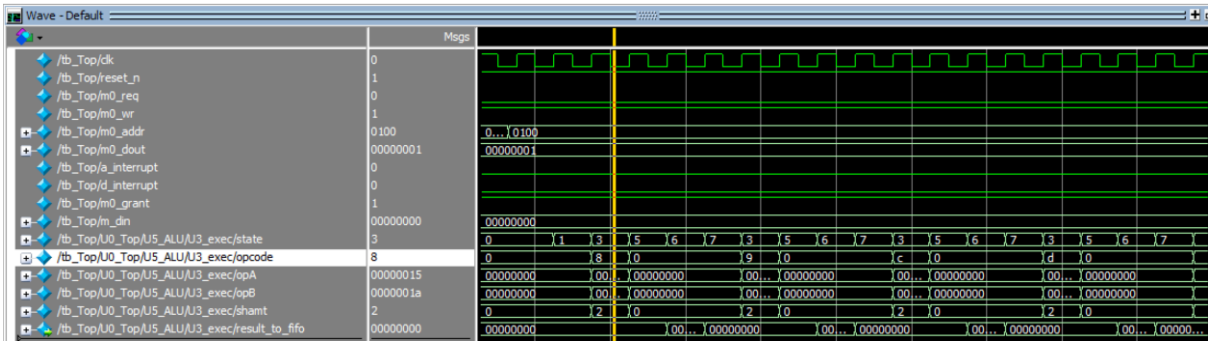
다음과 같은 코드를 작성하여 d_interrupt값을 testbench가 기다리도록 설계 하였다.



DMAC이 모든 Instruction과 Operand를 옮길 때까지 반복한다. D_interrupt signal이 위의 waveform과 동일한 형태로 발생하는 것을 확인하였다.

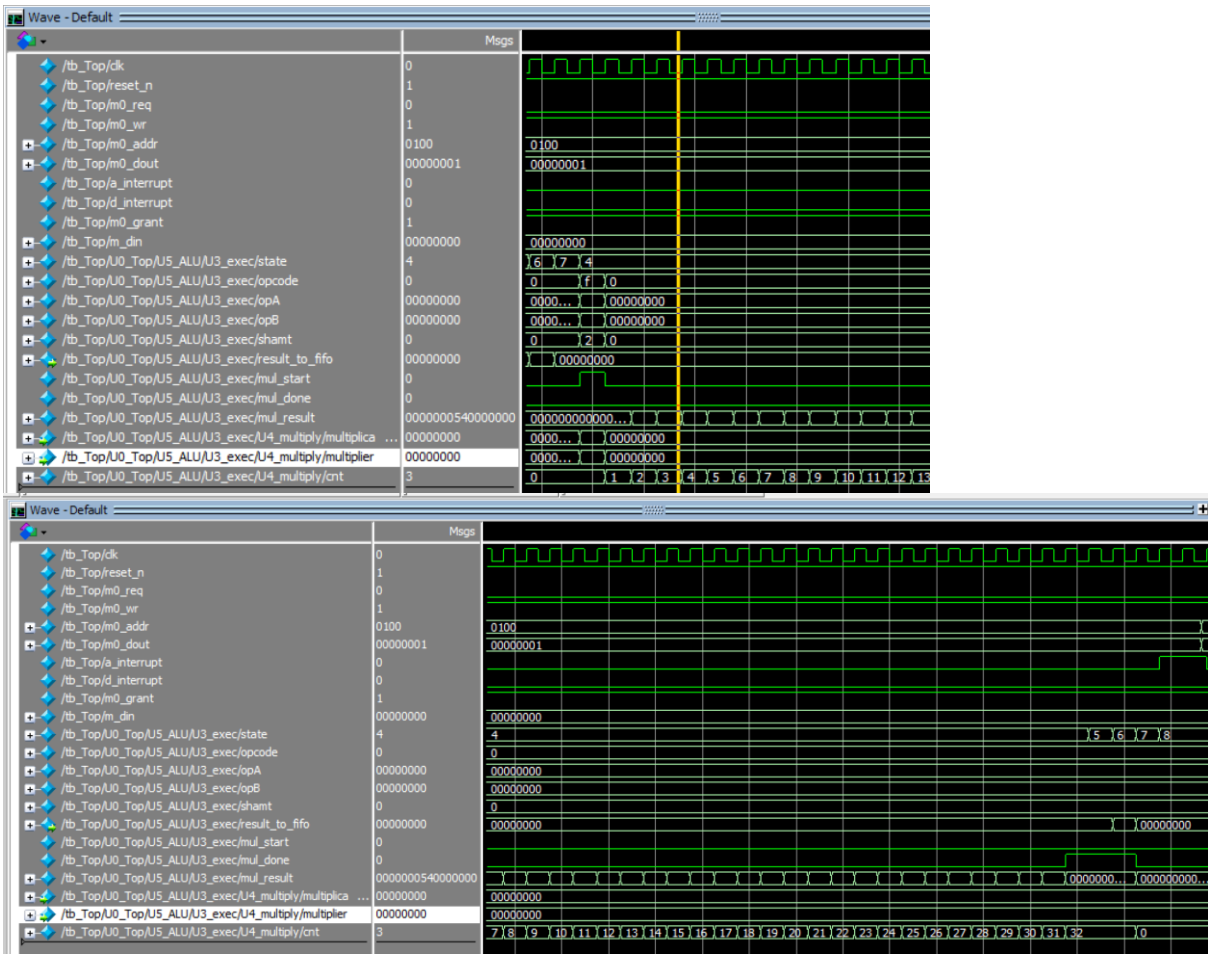


ALU의 Instruction FIFO에 초기에 입력한 값을 DMAC이 제대로 전달했음을 확인하였다.

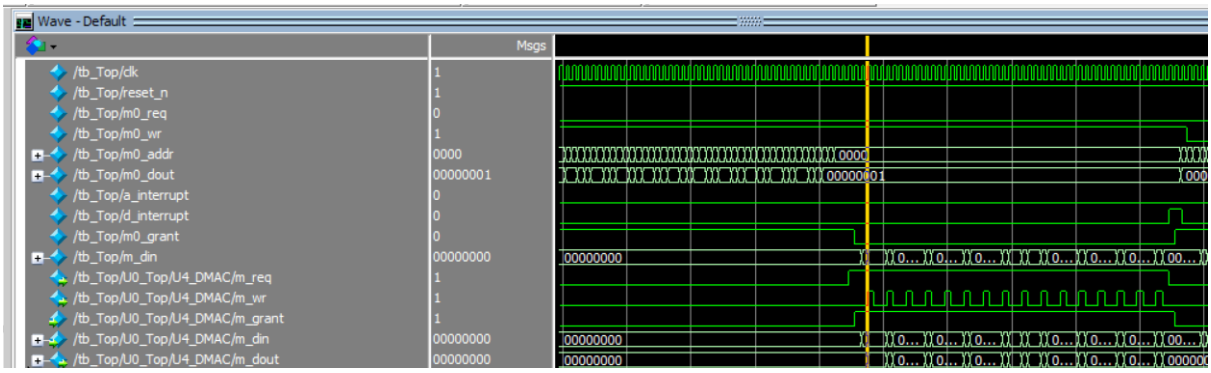


위의 instruction중 0000216a를 직접 분해한다면 opcode는 1000, opAaddr은 0101, opBaddr은 1010이 나오게 된다. Shift amount는 10이다. 이를 tokenize하여 3(EXEC) state로 전달한다. 따라서 해당 opcode에 맞는 결과를 도출하여 FIFO에 2cycle에 걸쳐서 Push하는 것을 확인하였다.

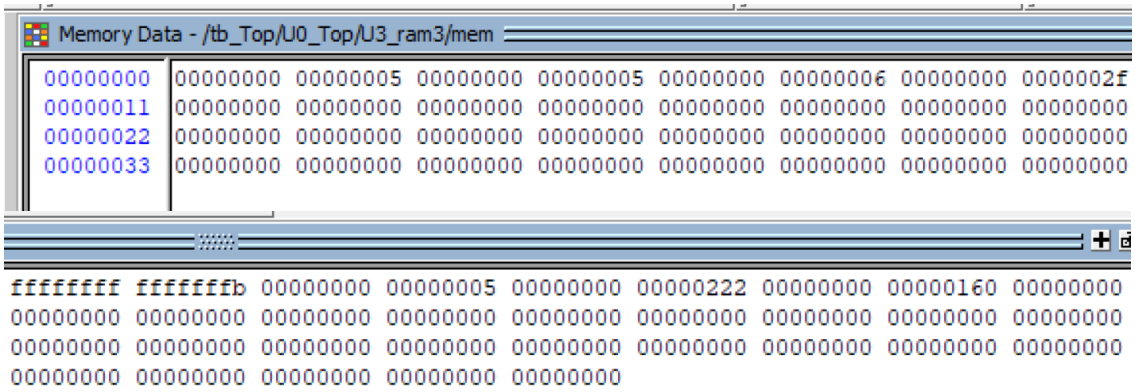
Opcode 1000의 경우 LSR A이므로 0x00000015를 >> 2bit 하여 0x00000005라는 결과를 도출할 수 있다. 다른 값들도 모두 연산이 제대로 진행되었음을 확인했다.



이는 Instruction 00003d6a를 분해한 결과 opcode는 MUL, operand 는 각각 0x15, 0x1a인 상황이다. 따라서 위 waveform에서 Mul_start신호를 Multiplier에게 전달하여 연산이 시작되는 것을 확인하였고 아래 waveform을 통하여 32cycle소모함과 동시에 mul_done signal이 calculation으로 전달되는 것을 확인하였다. 결과는 0x222가 나오게 된다. 자세한 것은 최종 RAM #3 Image를 통해 확인하였다. 모든 곱셈 연산이 완료가 된 후 a_interrupt signal이 1이 되고 ALU가 연산을 마쳤음을 알린다.



Testbench는 ALU의 interrupt를 clear해준 후 DMAC에게 ALU Result FIFO의 결과를 RAM #3으로 옮기도록 하는 Descriptor를 넣어준다. 이후 OPERATION_START 신호를 받은 DMAC은 위 DMAC FSM Diagram에 따라 R/W를 반복하며 ALU Result FIFO의 값을 읽어 RAM #3으로 전달한다. 모든 이동이 완료되면 DMAC은 Testbench에게 interrupt signal을 1로 보낸 후 모든 프로그램은 종료된다.



앞서 입력한 Instruction에 각각 해당하는 결과들이 32bits에서 64bits로 늘어나 RAM #3에 저장되어 있는 모습을 확인하였다. 63:32 – 31:0 순으로 저장되었다는 것을 확인할 수 있었다.

V. Conclusion

본 프로젝트는 Arithmetic & Logical Computing System을 구현함에 따라 hardware System Flow에 대한 깊은 이해를 할 수 있다. DMAC가 알아서 Data Transfer를 함에 따라 Processor의 효율성에 대한 이해를 높일 수 있었다.

FSM Diagram을 직접 설계하는 것을 통한 hardware computing에 대한 높은 이해도를 얻을 수 있다. Software Programming과 다르게 값에 대한 저장장치 및 전달을 이해할 수 있다. 같은 행동을 하는 state여도 다른 조건이라면 두 개의 State를 사용하여 처리한다는 내용도 알 수 있었다.

State가 적어 Chip Size가 작아진다고 무조건 좋은 것이 아니라 일 처리를 세부적으로 나누어 Cycle주기와 Chip Size에 대한 Trade Off를 이해하고 Overhead를 줄이는 방향으로 hardware를 설계할 수 있다.

단순히 Component로써의 ALU가 아닌 System 상에서의 ALU가 어떻게 작동하는지 이해할 수 있다. Register File의 rData값이 2개로 나오는 이유에 대해 이해하였다.

본 프로젝트에 대한 내용을 응용하여 DMAC에 Addressing mode를 추가한다면 DMAC에 FIFO를 추가하여 READ를 Data size만큼 반복하고 WRITE를 FIFO에서 POP하여 Performance를 개선할 수 있다.

본 프로젝트의 ALU에는 Barrel Shifter가 구현되지 않았다. Barrel Shifter를 구현하는 것을 통하여 성능이 개선된, 심화학습을 할 수 있다. 실제 CPU와 ALU의 상호작용에 대한 깊은 이해를 하였고 이를 바탕으로 더욱더 심화된 Hardware및 Software Programming 실력을 증진할 수 있다.

VI. Reference

[1] ALU / https://www.geeksforgeeks.org/introduction-of-alu-and-data-path/
[2] DMAC / https://www.elprocus.com/direct-memory-access-dma-in-computer-architecture/
[3] Simple BUS / https://en.wikipedia.org/wiki/Simple_Bus_Architecture
[4] RAM / https://namu.wiki/w/RAM
[5] D. M. Harris and S. L. Harris, Digital design and computer architecture, Morgan Kaufmann, 2007