Term Project (Factorial Computation System)

2022202061 양서은

Abstract

2023년 디지털논리회로2 Term Project는 Factorial computation system을 제작하는 것이다. 사용자로부터 operand 값을 입력 받아 펙토리얼 연산을 수행하는 것이 주 기능이다. Quartus를 사용해 Verilog 언어로 제작했으며 testbench를 통해 시스템의 동작을 검증한다. Factorial core, Bus, Ram으로 구성되어 있으며 testbench로부터 address 값과 input 값을 받아 알맞은 명령을 수행하고 값을 testbench에 전달하며 소통한다.

이 보고서에서 시스템을 이루는 각 요소들이 어떻게 동작하는지 어떤 값을 주고받는지 설명할 것이다.

I. Introduction

본 시스템은 Factorial core, Bus, Memory로 구성되어 있으며 testbench를 통해 동작을 통제한다. Factorial core가 본 시스템에서 가장 중요한 요소인데 Bus를 통해 접근할 수 있으며 복수 개의 register을 가지고 있다. 각각 opstart, opclear, opdone, intrEn, operand, result_h, result_l인데 이 값들을 통해 Factorial core의 동작을 제어하고 다른 요소(Bus, Ram)과 소통할 수 있다.

testbench에서 address 값과 din 값으로 정보를 전달한다. Base address를 기반으로 slave를 선택하고 Factorial core의 경우 address offset 값을 통해 어떤 register에 접근할지 결정한다. testbench로부터 operand, opstart 값을 받아와 저장되면 펙토리얼 연산을 시작한다. 이 때 interrupt 신호를 사용하고 싶다면 opstart 값을 받기 전에 intrEn register에 1을 저장해야 한다. 연산이 종료된 후 초기화를 원한다면 opclear register에 1을 저장해 Factorial core를 초기화할 수 있다. operand register에는 음수가 들어오지 않으며 2^{31}-1을 초과하는 값도 들어오지 않는다. intrEn register에 1을 저장하지 않는다면 interrupt 신호를 사용할 수 없고 opdone register의 값을 계속 polling 해야 연산이 종료됨을 알 수 있다. opdone 신호를 통해 현재 Factorial core가 어떤 상태인지 확인할 수 있는데 opdone[1:0]==2'b00일 땐 대기, opdone[1:0]==2'b01일 땐 연산 시작 및 진행 중, opdone[1:0]==2'b11일 땐 연산이 완료되었음을 나타낸다. operand 값으로 0이나 1이 저장되면 multiplier를 수행하지 않고 바로 결과값 1을 저장하며 연산이 완료된다.

펙토리얼 연산의 곱셈은 Assignment9로 구현했던 Booth Multiplier를 사용한다. 64bit끼리의 곱셉을 연산하고 128bit의 결과값을 내보낸다. result_h, result_l로 나눠 저장하고 다음 연산에선 result_l의 값과 operand를 곱한다. 연산 도중, result_l가 0이 된다면 result_h에 저장된 값을 불러와 곱셈을 진행한다.

따라서 이 시스템은 정확한 펙토리얼 결과값을 반환하지 않을 수 있다. operand의 값이 충분히 커져 중간 결과값이 64bit보다 커질 수 있는 상위 몇 bit는 result_h에 저장되어 다음 곱셈 연산에 영향을 끼치지 않기 때문이다.

연산이 종료된 후 testbench에서 결과값을 읽어올 때 최종 결과값은 result_h, result_l 2개의 register에 나눠 저장되어 있기 때문에 2회에 걸쳐 값을 읽어와야 한다. 값을 읽어와 Memory에 저장하고자 한다면 address 값을 기준으로 Memory에 저장된다. 따라서 overwrite하지 않도록 유념하며 testbench를 작성해야 한다. Factorial core의 초기화를 위해 opclear register에 1이 저장된다면 opclear, intrEn, operand를 제외한 register가 초기화된다. 따라서 testbench에서 opclear register 값에 다시 0을 써주며 해제해야 한다.

	~11/16	11/17~11/22	11/23~11/29	11/30~12/06
제안서 작성				
코드 작성				
코드 검증				
결과 보고서 작성				

프로젝트에 수정사항이 생기면서 마지막 주에도 코드 수정 및 검증이 이루어졌다.

II. Project Specification

1. Factorial core

입력으로 들어온 address에 대해 offset을 판단하는 모듈, register file에 din 값을 저장하는 모듈, controller, output을 결정하는 모듈로 이루어져 있다.

A. address_offset

slave로 Factorial core가 선택되고 입력 받은 address를 기준으로 offset을 판단한다. 해당된 offset은 register file의 enable 신호로 입력되어 해당하는 register에 값이 저장되도록 한다. register file에 flip flop이 7개이기 때문에 7bit offset을 선언하여 one hot encoding 방식으로 값을 할당했다.

B. register_file

7개의 flip flop으로 이루어져 있으며 각각 opstart, opclear, opdone, intrEn, operand, result_h, result_l 값을 저장한다. 이 때 주의해야 할 것은 opclear 신호에 의해 초기화되는 register가 있기 때문에 그에 해당하는 register의 reset 신호로는 reset_n & (~opclear)신호가 들어간다. 또한 opdone, result_h, result_l는 controller에서 나온 값들이 들어가기 때문에 enable 신호로 항상 1이 들어간다.

C. controller

FSM 형식으로 동작하며 state, Booth Multiplier의 start, clear 신호를 저장하는 flip flop과 next state logic, output state logic으로 이루어져 있다.

state	ns condition	output
INIT	opstart, 1 저장되면 EXCEP으로 이동	register 값 초기화
EXCEP	operand 값이 0 혹은 1이면 DONE으로 이동, 아니라면 START로 이동	register 값 이전 값으로 유지 opdone[1]=1
START	OPERATION으로 이동	register 값 이전 값으로 유지 op_start=1 (multiplier start signal)
OPERATION	op_done 신호가 1이 되면 CLEAR로 이동 (op_done은 Booth Multiplier 연산 완료 신호)	register 값 이전 값으로 유지
CLEAR	MINUS로 이동	operand 값 1 감소 결과값 저장 op_start=0 op_clear=1 (multiplier clear signal)
MINUS	operand 값 1이면 DONE으로 이동, 아니면 OPERATION으로 이동	register 값 이전 값으로 유지 operand 값 1이면, op_start=0 아니면 op_start=1 op_clear=0
DONE	opclear, 1 저장되면 INIT으로 이동	opdone[0]=1 register 값 이전 값으로 유지

위 표에 나와 있는 FSM을 기반으로 연산이 진행된다. Mutliplier는 controller 내부에 위치하고 있으며 Radix-4로 구현했다. 자세한 설명은 Assignment9 보고서와 소스코드를 참고하길 바란다.

D. _mux3

3-to-1 mux를 통해 Factorial core의 dout 값을 결정한다. address_offset 모듈에서 결정된 offset 값에 따라 해당하는 register의 결과를 가져온다.

2. Bus
   Master는 testbench, Slave는 Ram과 Factorial core이다. 따라서 address, wr, din 신호는 모든 slave와 wire로 연결되어 있고 address에 따라 select signal의 값만 변한다. address의 base address를 확인하여 0x0000~0x07ff이면 slave0인 Ram에게 명령하는 것이고 0x7000~0x71ff라면 slave1인 Factorial core에 명령하는 것이다. address와 wr 신호를 사용하여 slave에 값을 전달하거나 slave로부터 값을 받아온다. Bus를 통해 사용자와 시스템이 상호작용할 수 있는 것이다.

3. Ram (Memory)
   이차원 배열을 선언하여 해당하는 행에 값이 저장될 수 있다. s0_sel 신호가 1일 때만 동작하며 wr 신호가 1일 때 write, 0일 때 read 동작을 수행한다. 입력으로 들어온 address에 해당하는 배열 행에 din 값을 저장하거나 해당 행에 저장되어 있던 값을 불러온다.

4. Top
   Bus, Ram, Factorial core 모듈이 모두 instance되어 있다. Ram에 s0 신호, Factorial core에 s1 신호를 넣어주어 완성한다.

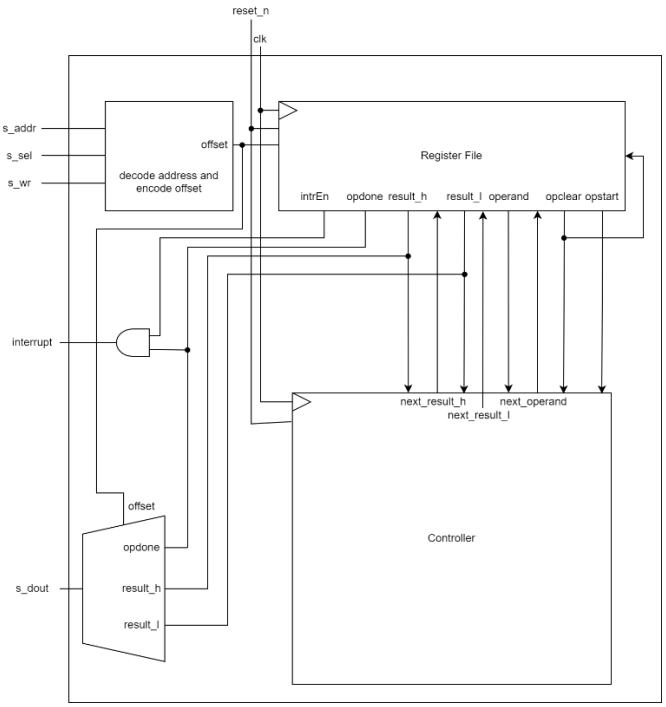
III. Design Details

각 component 별 pin description, block diagram, FSM 기재

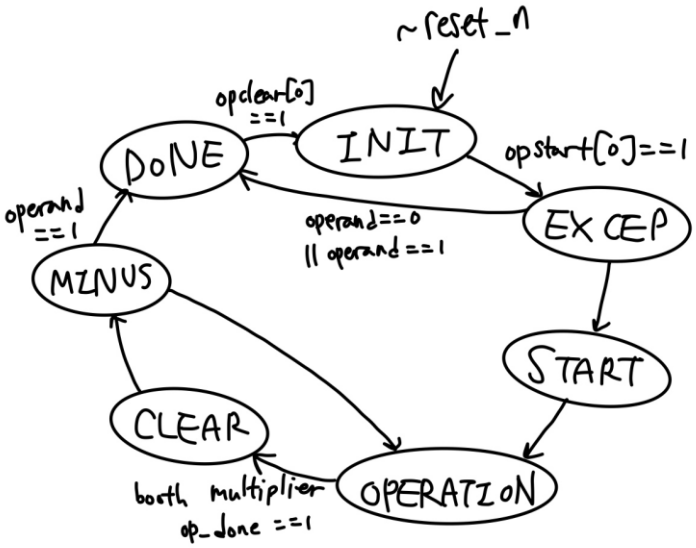
1. Factorial core
   1) pin description

Direction	Port name	Bit width	Description
Input	clk	1	clock
	reset_n	1	active low reset
	s_sel	1	slave1 select (s1_sel)
	s_wr	1	write/read
	s_addr	16	address
	s_din	64	data input
Output	s_dout	64	data output
	interrupt	1	interrupt out

   2) block diagram



3) state transition diagram

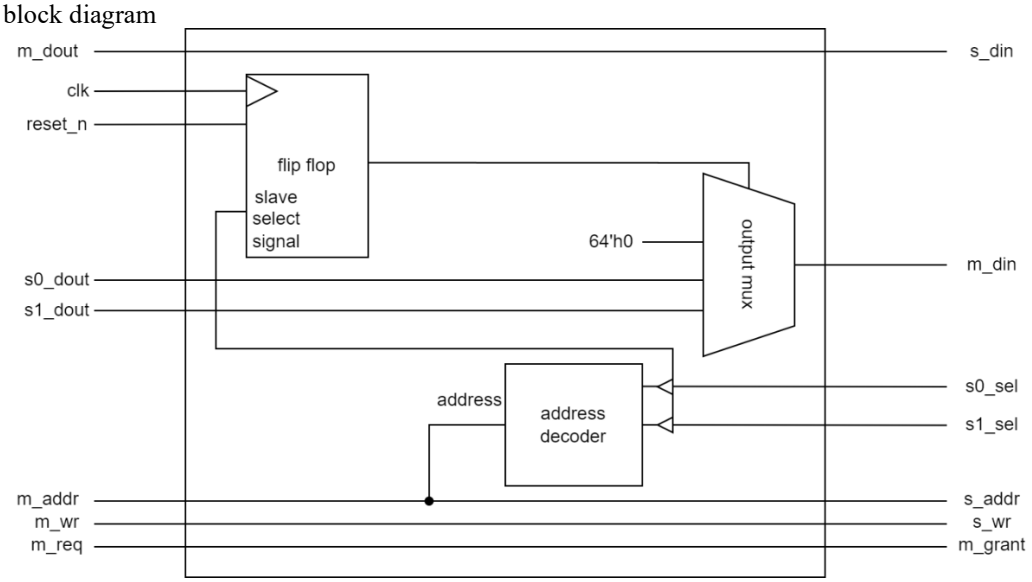


2. Bus

1) pin description

Direction	Port name	Bit width	Description
Input	clk	1	clock
	reset_n	1	active low reset
	m_req	1	master request
	m_wr	1	master write/read
	m_addr	16	master address
	m_dout	64	master data output
	s0_dout	64	slave0 data out
	s1_dout	64	slave1 data out
Output	m_grant	1	master grant
	m_din	64	master data input
	s0_sel	1	slave0 select
	s1_sel	1	slave1 select
	s_addr	16	slave address
	s_wr	1	slave write/read
	s_din	64	slave data input

2) block diagram



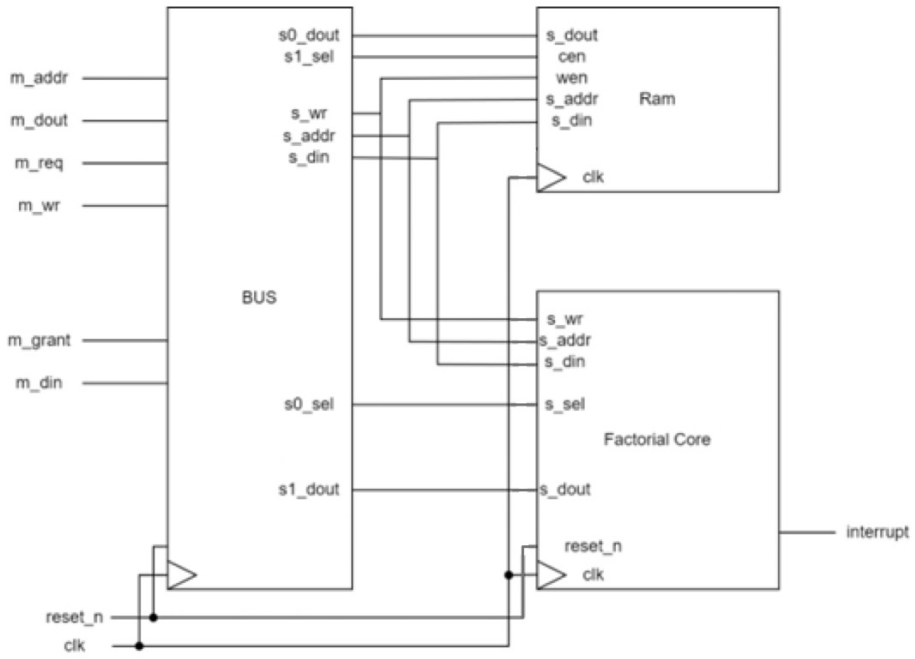
3. Ram
   1) pin description

Direction	Port name	Bit width	Description
Input	clk	1	clock
	cen	1	chip enable
	wen	1	write enable
	s_addr	8	address
	s_din	64	data in
Output	s_dout	64	data out

4. Top
   1) pin description

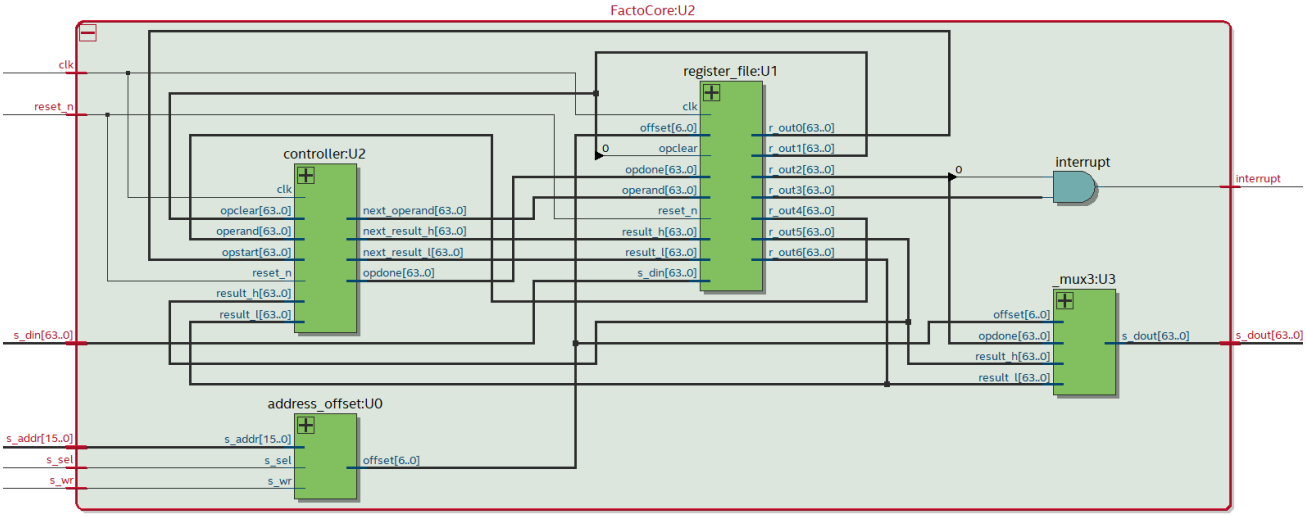
Direction	Port name	Bit width	Description
Input	clk	1	clock
	reset_n	1	active low reset
	m_req	1	master request
	m_wr	1	master write/read
	m_addr	16	master address
	m_dout	64	master data output
Output	m_grant	1	master grant
	m_din	64	master data in
	interrupt	1	Factorial core interrupt

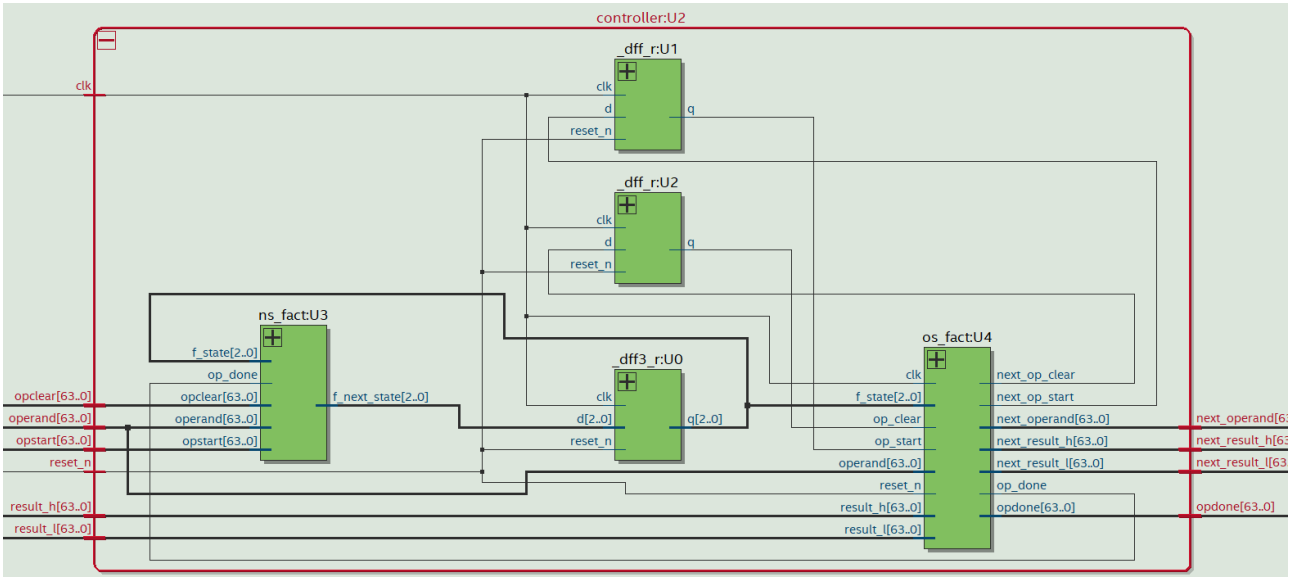
   2) block diagram

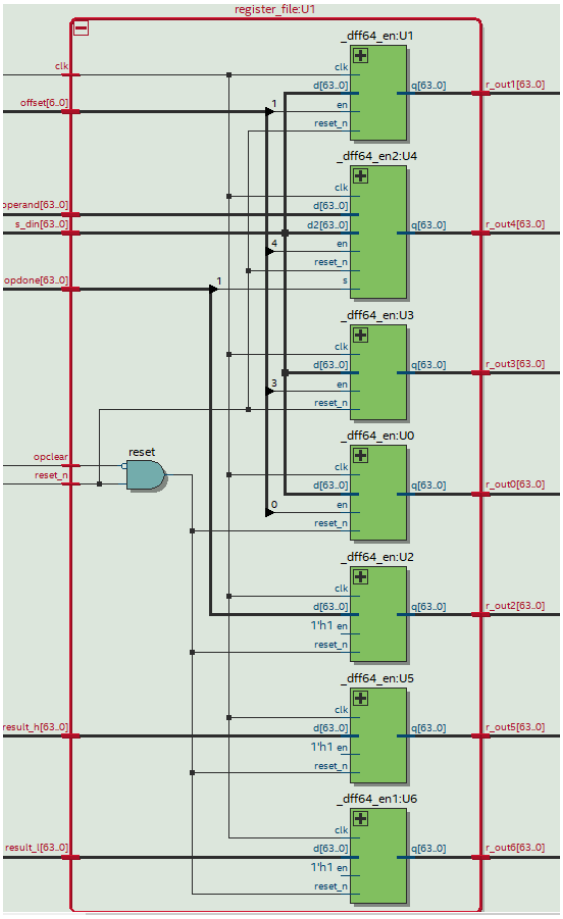


IV. Design Verifiaction Strategy and Results

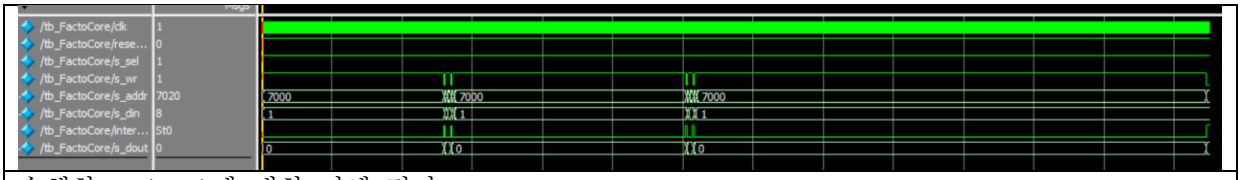
1. Factorial core
   A. RTL Viewer



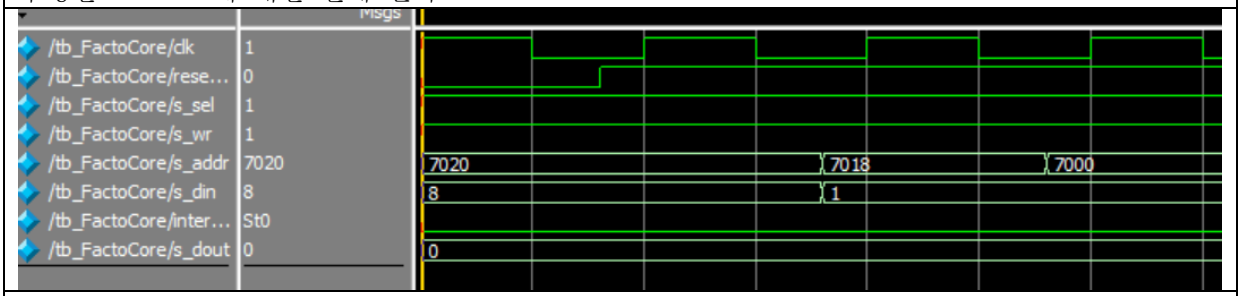




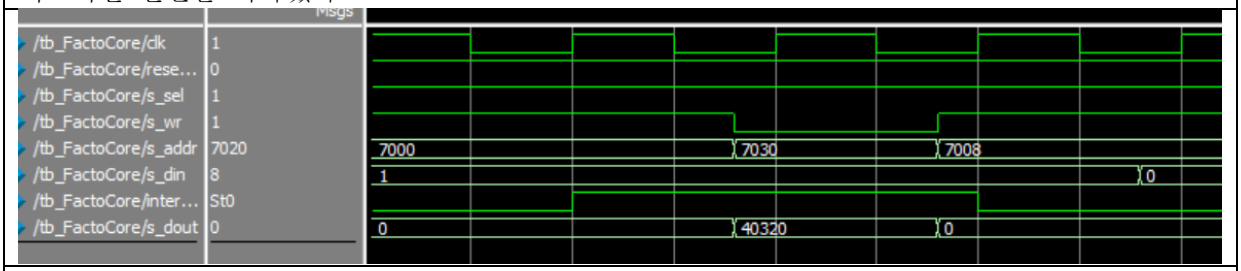
B. testbench



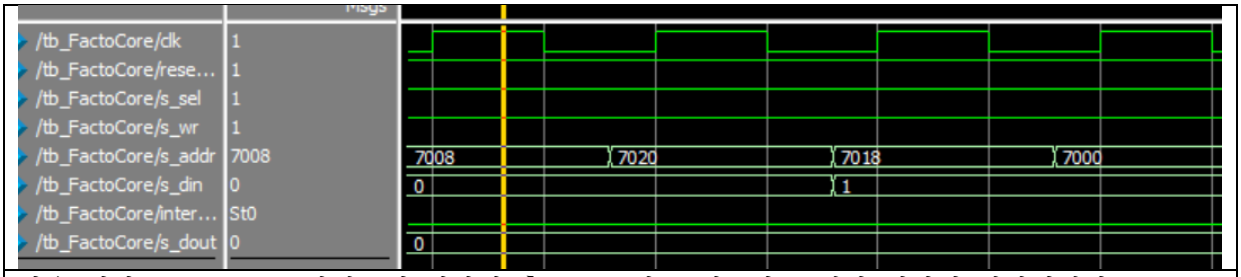
수행한 testbench에 대한 전체 결과



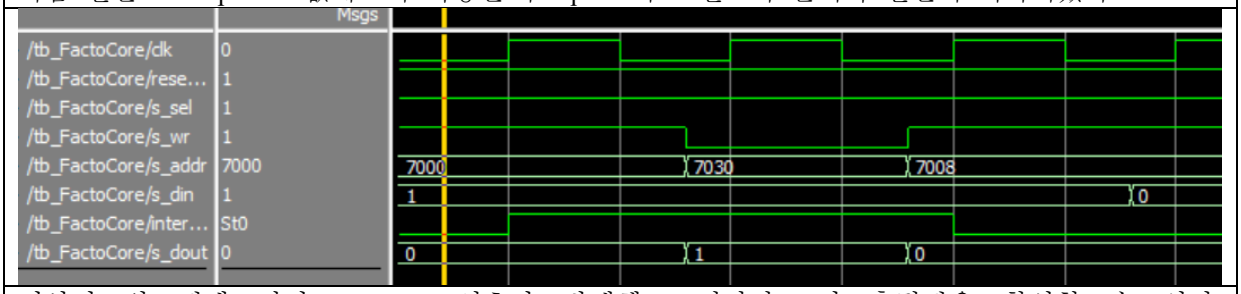
operand 값으로 8이 입력되었고 interrupt 신호를 사용하도록 하고 address 7000에 1을 저장하면서 펙토리얼 연산을 시작했다.



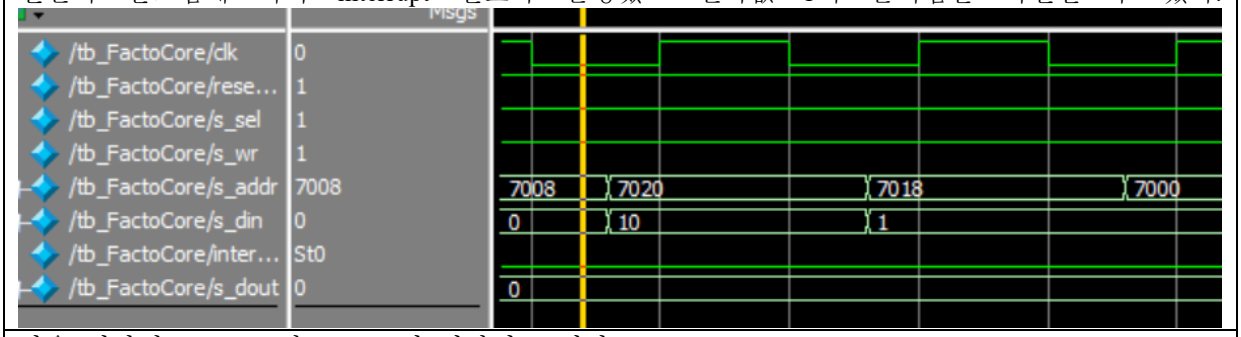
8! 연산이 완료되어 interrupt 신호가 발생했고 8! 의 결과값인 40320이 출력되고 있음을 확인할 수 있다. address 7008을 사용해 Factorial core를 초기화하는 것을 알 수 있다.



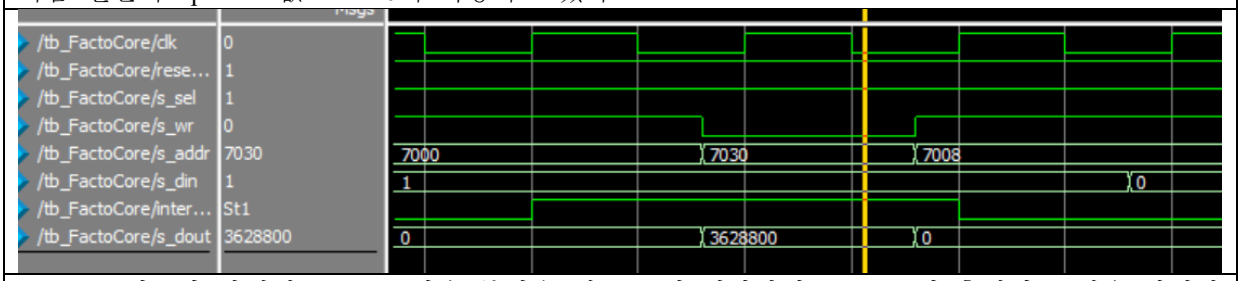
다음 연산으로 operand 값에 1이 저장된 후 opstart의 1 신호가 들어와 연산이 시작되었다.



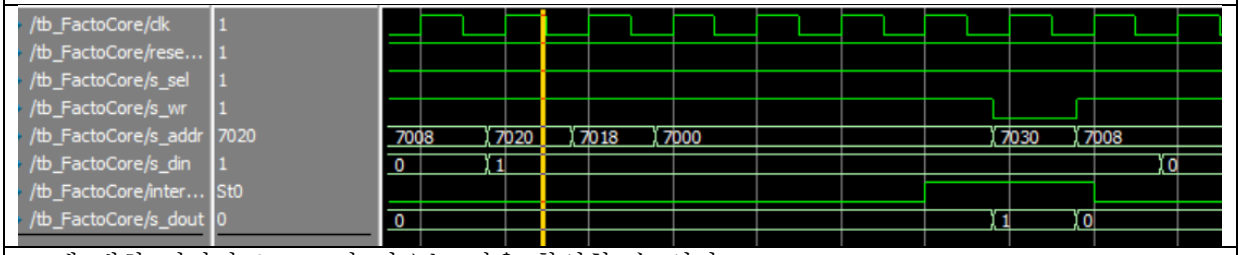
연산이 완료됨에 따라 interrupt 신호가 발생했고 결과값 1이 출력됨을 확인할 수 있다.



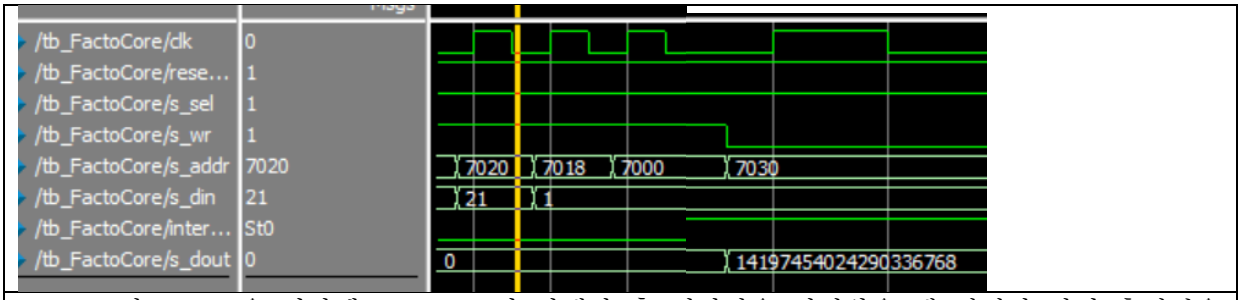
다음 연산의 operand 값으로 10이 저장되고 있다.



interrupt 신호가 발생했고 result 값을 불러올 때 10! 의 결과값인 3628800이 출력되는 것을 확인할 수 있다.



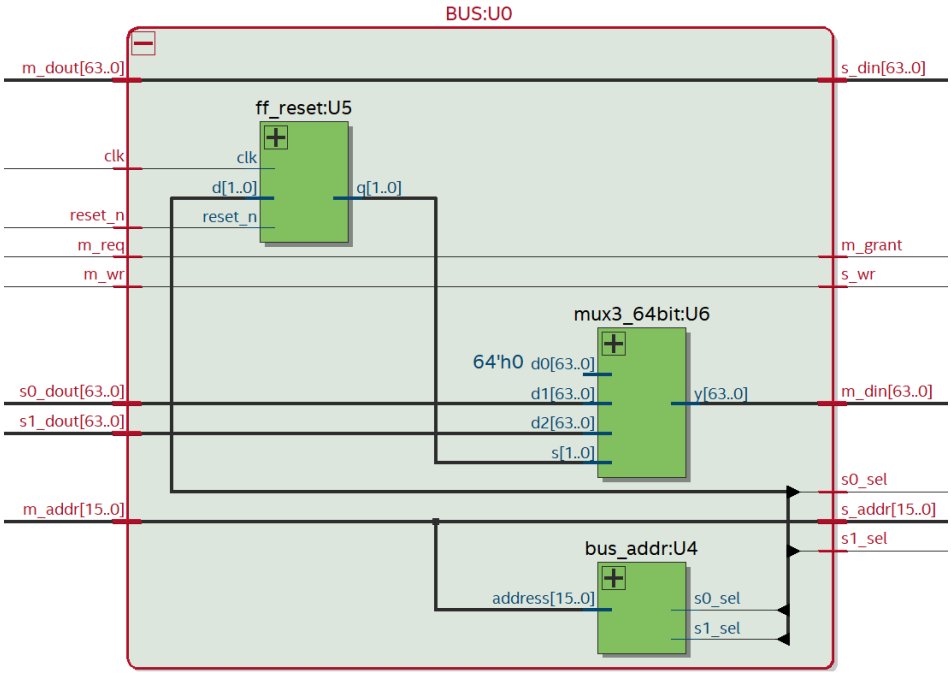
1! 에 대한 결과값도 1로 잘 나오는 것을 확인할 수 있다.



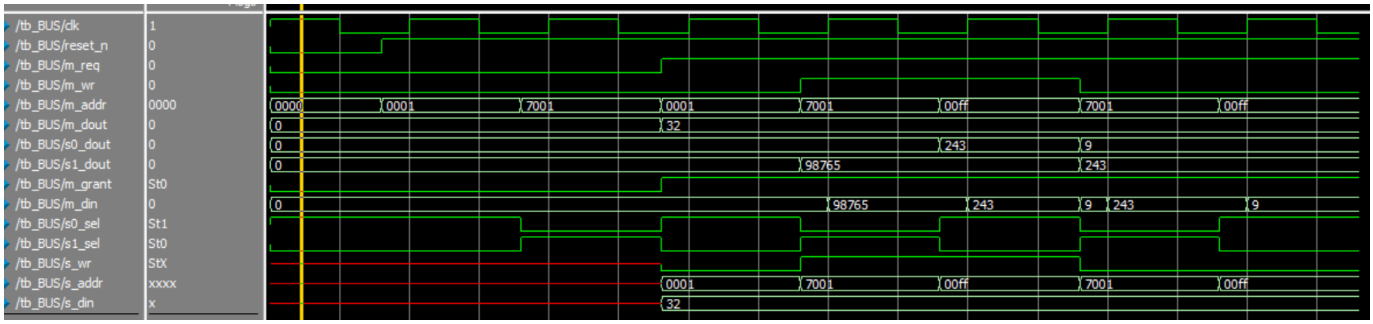
operand 값으로 21을 저장했고 interrupt가 발생한 후 결과값을 읽어왔을 때 적절한 값이 출력됨을 확인할 수 있다.

2. Bus

A. RTL viewer

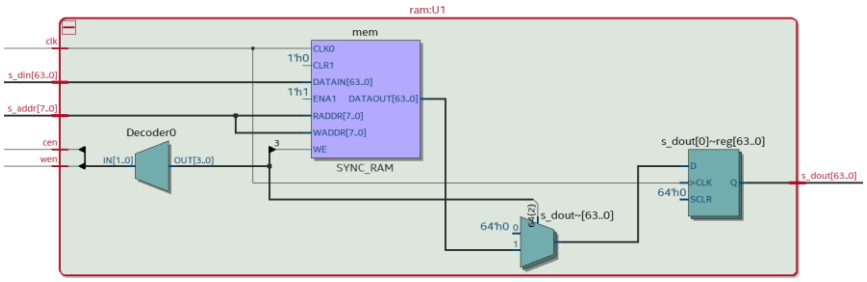


B. testbench

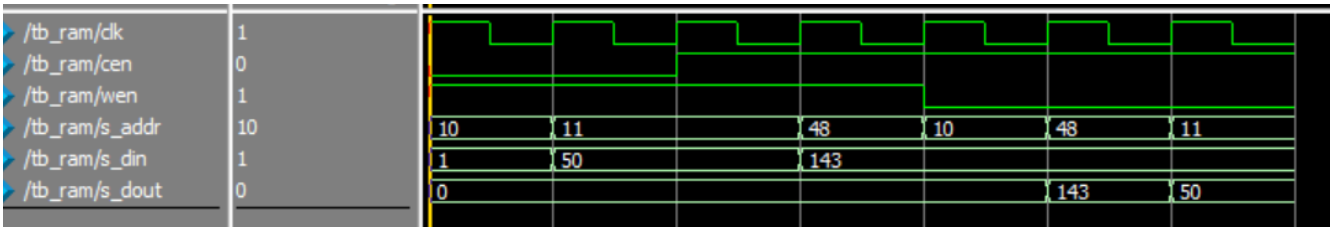


입력값에 따라 알맞은 결과값이 출력되는 것을 확인할 수 있다. m_req 신호가 0일 땐 입력 받은 address 값에 따라 slave 선택은 이루어지지만 어떤 값도 전달하지 않는다. m_req 신호가 1이 되면 m_grant 신호가 1이 되고 입력 받은 address 값을 기준으로 slave를 선택하고 m_dout 값을 전달한다. m_wr 신호가 0일 땐 read 동작을 수행하기 때문에 해당 address에 저장된 값을 불러와 출력한다.

3. Ram (Memory)
   A. RTL viewer

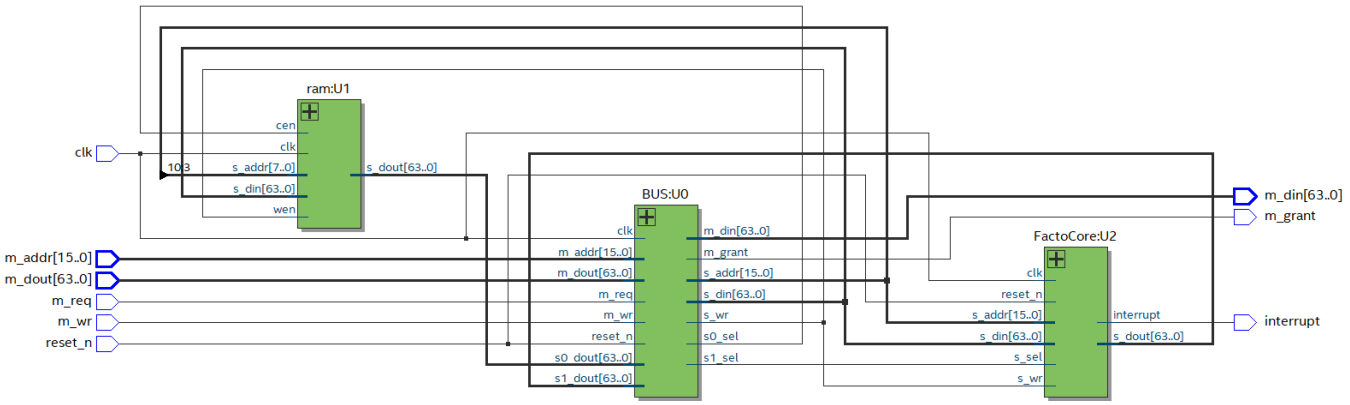


   B. testbench



cen 신호가 1일 때 ram에 정보를 읽고 쓰는 동작이 수행됨을 확인할 수 있다. wen 신호가 1일 때 ram에 din 값을 저장하고, 0일 때 ram에서 값을 불러와 dout에 저장한다.

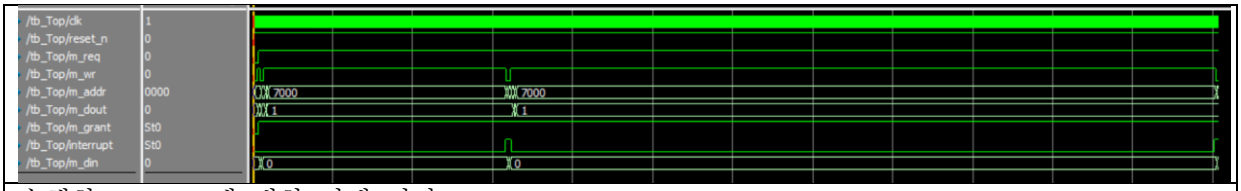
4. Top
   A. RTL viewer



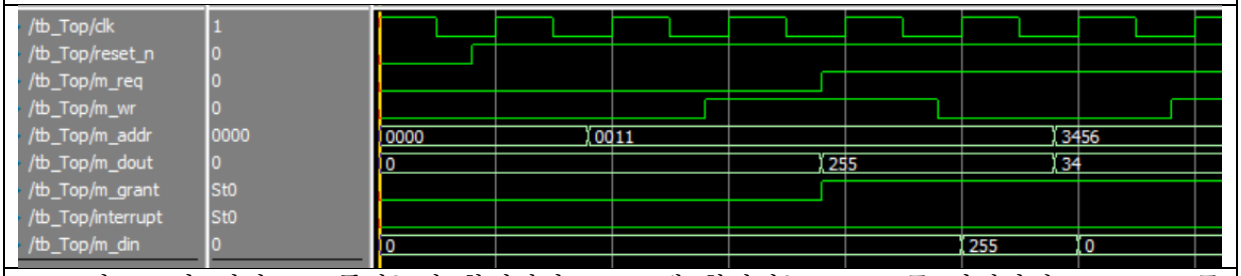
   B. Synthesis Result

Flow Status	Successful - Tue Dec 05 19:29:11 2023
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	Top
Top-level Entity Name	Top
Family	Cyclone V
Device	5CGXFC7D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	9,154 / 56,480 (16 %)
Total registers	16854
Total pins	150 / 522 (29 %)
Total virtual pins	0
Total block memory bits	0 / 7,024,640 (0 %)
Total DSP Blocks	0 / 156 (0 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 16 (0 %)
Total DLLs	0 / 4 (0 %)

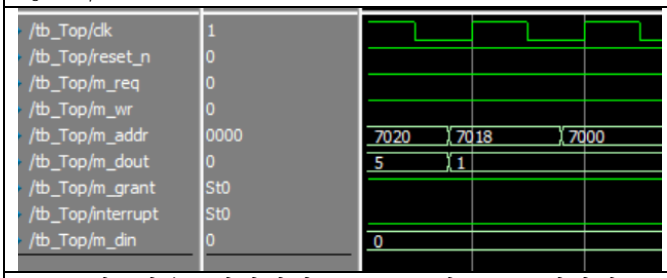
C. testbench



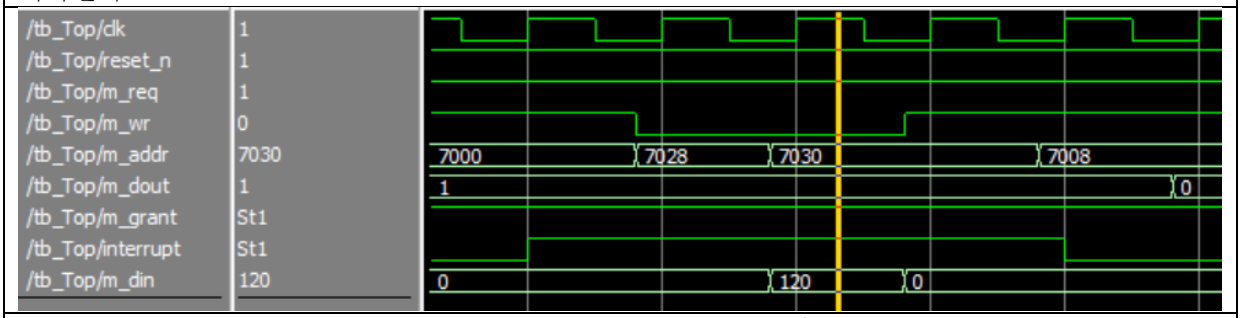
수행한 testbench에 대한 전체 결과



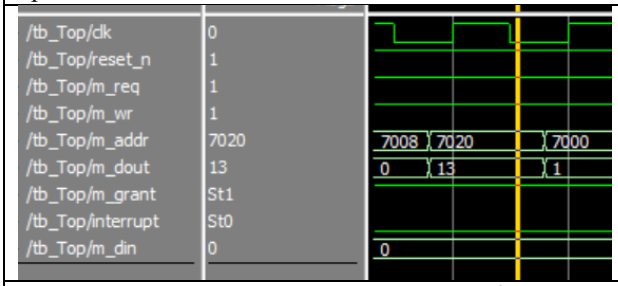
slave0와 Bus가 제대로 소통하는지 확인한다. slave0에 합당하는 address를 가진다면 read, write를 제대로 수행하지만 0x3456과 같이 합당하지 않은 address를 가진다면 값을 slave에게 전달하지 않는다.



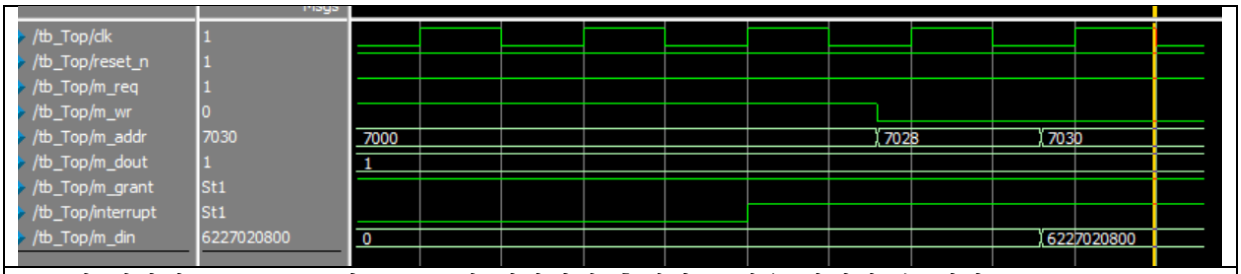
slave1에 값을 전달한다. operand에 5를 저장하고 interrupt 신호를 사용함을 전달하고 연산을 시작한다.



operand 5에 대한 결과값 5! = 120이 출력된다. 결과값을 확인한 후 Factorial core를 초기화해준다.



다음 연산으로 operand에 13을 저장한 후 펙토리얼 연산을 시작한다.



13! 의 결과인 6227020800이 result_1에 저장되어 출력되는 것을 확인할 수 있다.

```
# ================================
#  P           *         *
#  A         *  *      *  *
#  S       *    *    *    *
#  S
#  E           *      *
#  D           *      *
#  !           *******
# Total calculation cycle =      151586 cycles
# ** Note: $stop    : C:/intelFPGA_lite/18.1/20
#    Time: 3045191 ps  Iteration: 0  Instance:
# Break in Module tb_FactorialCoreTop in file C
```

이를 통해 Factorial computation system이 정상적으로 동작하고 있음을 확인할 수 있다.

V. Conclusion

프로젝트 스펙이 처음 공개되었을 땐 Multiplier도 구현하지 않았던 때라 어떻게 구현해야 될지 감조차 오지 않았다. 따라서 Multiplier와 Ram, Bus를 컴퓨터공학기초실험 실습 시간에 직접 구현할 때 프로젝트에 sub module로 사용할 것을 염두해두며 구현했었다. 세 모듈 모두 구현했을 땐 대강 머리 속에 어떤 식으로 시스템의 회로도가 이루어질지 그려졌고 어렵지 않게 구현할 수 있을 줄 알았다.

초반에 구현할 땐 Factorial core의 state 수가 많지 않았다. 하지만 매 곱셈마다 이루어져야 하는 multiplier의 초기화, operand 감산, operand의 예외 처리 등 고려해야 할 경우의 수가 존재했고 state 수가 점점 많아져 최종적으로 7개의 state를 가지는 모듈이 만들어졌다. 모든 검증이 끝나고 다시 모듈을 점검했을 때 서로 비슷한 output logic을 가지고 특정 1개의 값만 변하는 state가 있는 것을 깨달았고 state 수를 줄일 수 있다고 판단했다. state 수를 줄이고 싶었지만 시간 관계 상 그럴 수 없었다. 다음에 기회가 된다면 state 수를 줄여 더 효율적인 Factorial core를 만들고 싶다.

output logic을 만든 후 검증할 때 제대로 동작하지 않는 것을 확인했다. 코드를 다시 체크했을 때 always문의 sensitivity list에 state만 추가되어 있었다. 매 사이클마다 state가 변하는 FSM의 경우 오류가 발생하지 않지만 본 시스템의 Factorial core와 같은 경우에는 특정 신호가 들어오기 전까지 현재 state를 유지하는 state가 있기 때문에 sensitivity list에 state만 있을 경우 신호가 들어와도 state가 변하지 않는 오류가 발생했다. sensitivity list에 기타 신호를 넣어준 후 다시 시뮬레이션을 돌려보니 정상적으로 동작함을 확인할 수 있었다.

최종적으론 Factorial core의 state수가 7개지만 본래 구현할 땐 EXCEP state가 존재하지 않았다. Factorial core를 검증하던 도중 operand로 1이 들어왔을 때 곱셈 연산을 수행하고 감산해 0이 되어 결과값이 0으로 초기화되는 오류가 발생했다. 또한 다른 예외를 처리하는 과정에서 state 꼬임이 발생해 곱셈을 수행하면 안 되는 상황에서 multiplier가 동작해 올바르지 않은 결과값이 출력되었다. 따라서 INIT state에서 opstart[0]==1이 되면 무조건 EXCEP state를 거쳐 예외처리를

해주고 START state로 이동하도록 수정했다. 그 후 0, 1, 기타 수에 대해 Factorial core 연산이 정상적으로 수행되는 것을 확인할 수 있었다.

Top 모듈까지 정상적으로 동작하는 것을 확인한 후, 조교님께서 제공해주신 testbench를 사용해 시뮬레이션을 돌렸을 때 틀린 result를 받는 경우가 발생했다. 스스로 만든 testbench는 모두 통과했기 때문에 그 오류를 찾기가 힘들었는데, 그 이유는 multiplier에서 발생하고 있었다. multiplier에는 multiplier, multiplicand 두 피연산자가 입력되어 곱셈을 수행하는데 본 시스템에서 multiplier에 저장된 result 값을 할당하여 곱셈을 진행할 경우 multiplier의 sign bit까지 1이 되어 오류가 발생할 수 있다. 3bit씩 보면서 덧셈과 shift 연산을 수행하기 때문에 최상위 bit에 대한 연산에서 오류가 발생하는 경우가 생겼다. 따라서 operand가 multiplier로, result가 multiplicand로 입력되게 바꿔 최상위 bit에서 발생할 수 있는 오류를 방지했다.

VI. Reference

[1] 이준환교수님/디지털논리회로2/광운대학교/2023
[2] 이준환교수님/컴퓨터공학기초실험2/광운대학교/2023