Factorial Computation System

2022202055 윤재석

Abstract

디지털논리회로2 term project로 설계한 Factorial Computation System이다. Top module에서 bus, ram, factorial core를 instance하여 factorial 연산을 수행하는 module이다. Bus를 이용하여 ram과 factorial core를 slave로 두어 두 slave에 서로 값을 주고받을 수 있게 연결하여 연산을 하도록 설계 및 구현한다. 사용자로부터 값을 입력 받아 해당하는 주소에 맞게 slave로 선택한 후 연산을 진행한다.

I. Introduction

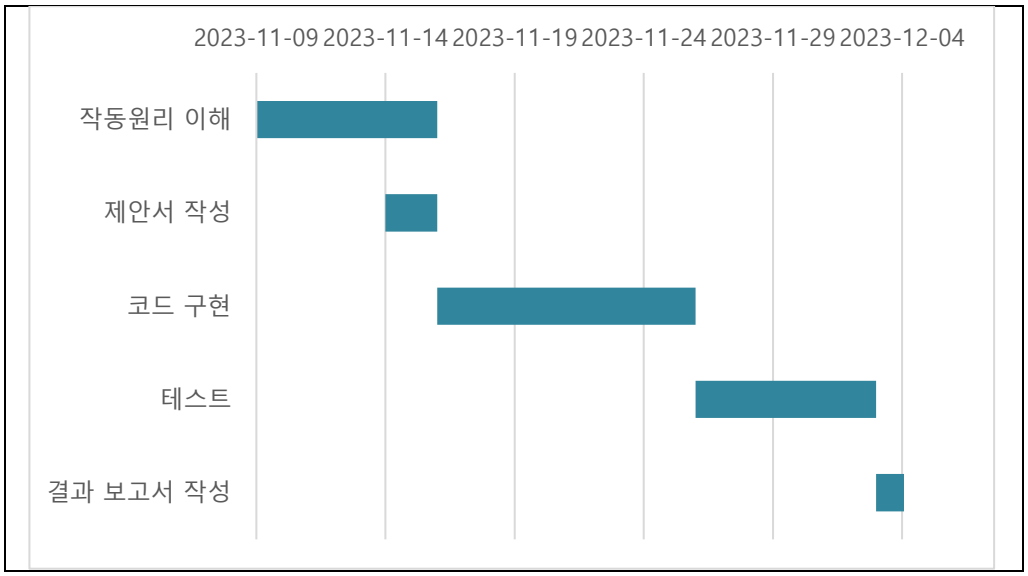
Factorial 연산으로는 testbench에서 주소 및 값을 입력 받아 ram 또는 factorial core로 접근한다. 이후 입력 받은 값을 통해 연산을 진행한다. Operand인 n은 1부터 $2^{31} - 1$의 값을 입력 받을 수 있다. Multiplier를 통하여 factorial 연산을 수행하며 operand를 하나씩 감산하며 이를 진행한다. 이때 발생하는 result는 128비트로 상위 64비트는 result_h, 하위 64비트는 result_l에 저장한다. Factorial 연산을 진행하면서 곱하는 승수와 피승수는 operand와 result_l를 기본으로 연산을 진행한다. 이때 result_l가 0일 경우 result_h의 값을 입력 받아 multiplier에 연산을 수행할 수 있도록 한다. Operand는 2까지만 연산을 수행할 수 있도록 하며 만약 0 또는 1의 값이 들어올 경우 곱셈에 대한 연산을 진행하지 않고 바로 값을 출력할 수 있도록 예외 처리를 하도록 한다. Factorial core에 있는 register는 offset에 따라 write하거나 read할 수 있다. 이때 write하는 register의 순서로는 operand, intrEn, opstart를 통해 연산을 시작한다. 만약 opclear가 입력될 경우 opstart, opdone, result_h, result_l를 초기화할 수 있도록 구현한다. 입력 받은 intrEn이 1인 경우 interrupt를 발생시키도록 하며 interrupt를 사용하지 않는 경우 polling의 방식으로 opdone에 따라서 연산의 상황을 알 수 있도록 한다. 연산이 완료된 후 result_h와 result_l를 2회에 걸쳐 offset의 주소 값에 따라 read하며 그 결과를 memory에 write한다. 주소는 testbench에서 임의로 작성할 수 있다. 이후 factorial core를 초기화하여 다음 연산을 수행할 수 있도록 한다. Factorial core 내부에는 factorial controller와 booth multiplier가 있다. Multiplier는 operand와 result에서의 조건에 따라 64비트를 입력 받아 2개의 값을 곱한다. 이때 factorial core와 multiplier끼리 hand-shking하여 register의 값과 신호를 주고받아 factorial 연산을 완료하도록 한다.

Bus는 1개의 master와 2개의 slave를 가진다. 이때 master로는 Top 모듈이 있으며 slave는 각각 ram과 factorial core이다. Ram의 주소는 0x0000부터 0x07FF까지의 범위로 값을 입력 받을 수 있다. Factorial core의 주소는 0x7000부터 0x71FF까지의 범위로 값을 입력 받을 수 있다. Slave 0는 ram으로 memory에 접근할 수 있도록 한다. Slave 1은 factorial core로 입력 offset에 따라 factorial 연산을 수행할 수 있도록 한다. Bus에서 입력 받은 operand, intrEn, opstart를 통해 factorial core의 연산을 수행한 뒤 address를 memory에 접근할 수 있도록 입력을 받으면 bus로 전달된 값을 ram에 저장할 수 있도록 구현한다.

Ram은 64비트의 데이터를 총 256개 저장할 수 있도록 하는 memory이다. Bus로부터 factorial 연산이 완료된 결과를 주소 값을 입력 받았을 때 해당하는 주소에 그 값을 저장할 수 있는 역할을 수행한다.

Top은 bus의 master interface로 해당 모듈에서 주소와 bus에 request를 하여 값을 주고받을 수 있는 모듈이다. Testbench에서 입력 받은 값을 Top모듈에 주어 이를 bus에 전달할 수 있도록 설계한다.

FSM을 통하여 각 모듈에 대한 logic이 어떻게 되는 지 알 수 있다. 이에 대한 설계를 완료한 뒤 각 모듈에 대해서 testbench를 작성하여 결과 값이 제대로 출력되는 지 확인하고 설계 과정에서 잘못된 점을 바로잡는다.



프로젝트를 진행하는 일정 및 계획을 나타낸 간트 차트이다. 11월 9일부터 16일까지 프로젝트 명세서를 보며 factorial 연산에 대해서 각 모듈이 어떻게 작동해야 하는 지에 대해서 이해하도록 한다. 이후 14일부터 16일까지 이해한 내용을 바탕으로 프로젝트를 어떻게 진행할 지에 대해서 제안서를 작성한다. 제안서를 작성한 후 코드를 직접 설계한대로 구현한다. 이때 코드를 작성해보며 오류가 생기는 점을 수정하고 완성한다. 11월 26일 까지의 과정으로 코드를 완성하였을 때 testbench를 작성하도록 한다. 작성한 testbench를 바탕으로 코드에 문제가 발생되었을 경우 오류를 다잡기 위하여 기간을 12월 3일까지로 정한다. 12월 4일까지 보고서를 작성하도록 하며 기타 오류나 수정 사항이 발생할 경우 마감 기한인 12월 6일까지 코드 및 보고서를 수정하여 작성하고 제출하도록 하였다.

II. Project Specification

Top module은 input으로 들어온 값과 wire를 통하여 bus, ram, factorial core를 instance하여 상호작용을 할 수 있도록 구현한다. 주로 bus를 통하여 master interface로써 값을 입력하고 그 값을 해당하는 address에 맞게 입력되었다면 slave interface에 접근하여 값을 주고받을 수 있다.

Bus module은 Top module로부터 입력 받은 master interface 값을 통하여 address decoder에서 주소를 분류하여 0x0000~0x07FF이면 ram으로 0x7000~0x71FF이면 factorial core로 접근할 수 있도록 설계한다. Arbiter를 통하여 master interface의 값을 m_req에 따라서 state를 2개로 분류하여 m_grant를 0 또는 1로 출력하여 master가 접근할 수 있는 지를 결정한다. 2개의 submodule에서 입력 받은 값을 통해서 mux를 이용해 m_wr, m_addr, m_dout를 m_grant에 따라서 select하여 각각 s_wr, s_addr, s_din으로 값을 전달할 수 있도록 한다. 또한, s0_dout, s1_dout을 입력 받은 주소 값에 따라서 s0_sel 또는 s1_sel에 값이 입력되었을 때 해당하는 slave로 접근할 수 있도록 mux를 통해 m_din으로 master가 값을 받을 수 있도록 설계한다. 이때 m_grant와 s0_sel, s1_sel이 0일 경우 해당하는 mux에서 받는 값은 0으로 고정한다.

Ram은 bus로부터 받은 s0_sel 값을 통해 내부의 input으로 cen을 설정한다. 입력 받은 s_wr은 wen으로 받고 s_addr을 통하여 해당하는 주소 값에 cen과 wen의 조건에 따라서 값을 write하거나 read할 수 있도록 한다. 만약 cen과 wen이 모두 1일 경우 입력 받은 s_din을 write하고 cen만 1일 경우 해당 주소 값에 저장된 값을 s_dout으로 read하여 출력할 수 있도록 한다. 모두 0일 경우 s_dout은 0을 출력하도록 한다.

Factorial을 연산하기 위해서 booth multiplier를 이용하는데 이때 multiplier module은 next state logic과 output logic으로 나누어 구현한다. Next state logic을 통하여 multiplier의 input에 따라 state를 결정하도록 한다. 이때 내부의 register는 state 및 count로하여 radix-2이기 때문에 총 64번의 cycle을 연산할 수 있도록 구현한다. Count를 더하는 방법은 CLA를 통하여 값을 한 번씩 증가시키도록 한다. 초기화하는 reset_n과 opclear에 따라 count를 초기화할 수 있도록 하며 IDLE state와 MUL state로 나누어 연산을 진행한다. 이때 입력되는 신호로 op_start가 1인 경우에 연산을 진행할 수 있도록 하며 op_clear가 active-high할 경우 IDLE state로 이동하여

값을 초기화할 수 있도록 설계한다. 만약 연산이 종료된 후 count가 64가 되면 op_done에 대한 신호를 1로 변경하여 연산이 종료되었음을 알린다. 이때 신호에 맞는 state를 매 cycle마다 output으로 출력하여 output logic으로 신호를 전달할 수 있도록 한다. Multiplier의 output을 만들기 위해서 설계한 logic으로 result를 출력하도록 하기 위해서 radix-2에서는 곱할 2개의 비트를 읽어 00, 01, 10, 11일 경우에 따라서 각각 다음 값을 출력하도록 한다. IDLE state일 경우에는 연산을 진행하지 않고 MUL state일 경우에만 비트를 읽어 00과 11일 경우에는 값을 sign 비트에 따라서 arithmetic shift right만 하여 값을 출력한다. 01일 경우에는 result의 상위 64비트를 CLA로 더한 값으로 임시로 저장한 뒤에 sign 비트를 통하여 arithmetic shift right를 진행하여 next result를 출력하도록 값을 지정한다. 10일 경우에는 result의 상위 64비트를 CLA로 감산한 값으로 임시로 저장한 뒤에 sign 비트를 통하여 arithmetic shift right를 진행하여 next result를 출력하도록 값을 지정한다. 이때 연산을 진행하다가 op_clear나 op_done에 따라서 값을 초기화하거나 연산을 종료하여 값을 출력한다.

Factorial core module은 Top module로부터 입력 받은 주소 값과 설계로 지정된 offset에 따라서 값을 register에 입력하도록 구현한다. Factorial core는 내부의 factorial controller와 multiplier를 hand-shaking하도록 구현한다. Multiplier에서 사용하는 mul_start, mul_clear, mul_done와 result에 대한 신호를 controller에서 입력 및 출력하여 연산을 할 수 있도록 한다. Factorial controller에서 mul_done, result를 입력으로 받고 mul_start와 mul_clear를 출력한다. Multiplier에서 연산이 완료될 경우 mul_done을 받아 다음 state로 이동하기 위해서 받는다. 이때 연산을 반복해야 하므로 mul_start와 mul_clear를 출력하여 연산을 반복할 수 있게 설계한다. Factorial controller에서 next state logic과 output logic을 통하여 총 6개의 state로 작동한다. 각각 offset에 맞는 register의 값을 decoder로부터 전달받거나 전달할 수 있도록 한다. 이때 전달받은 opstart, opclear, intrEn, operand를 통해 연산을 진행한다. State는 IDLE, START, EXEC, LAST, DONE, CLEAR로 구현한다. IDLE은 초기 상태의 state로 multiplier의 연산을 진행하기 전에 상태이며 opstart이 1일 경우에 START로 이동하면서 multiplier의 연산을 시작하도록 구현한다. START는 factorial의 연산의 첫번째 곱으로 입력 받은 operand와 result_l의 1을 곱하여 연산을 진행한다. 이때 multiplier의 mul_done이 1으로 입력되는 경우 EXEC state로 이동할 수 있도록 하며 next operand를 operand에서 1을 감산한 결과로 입력한다 이때 result_h를 multiplier의 result의 상위 64비트로 result_l를 result의 하위 64비트로 설정한다. 단, operand가 0또는 1로 입력되었을 경우 예외처리를 하여 state를 바로 DONE으로 이동할 수 있도록 하며 multiplier의 연산을 수행하지 않도록 mul_start를 0으로 주며 mul_clear를 1로 준다. 이때 result_h는 0으로 result_l는 1로 출력하도록 한다. 이후 multiplier에서 연산할 값으로 result_h 또는 result_l를 입력할 때 result_l가 0일 경우 result_h가 multiplier로 입력되어 연산을 수행할 수 있도록 한다. EXEC에서는 factorial의 연산이 끝나기 전까지 mul_done이 1일 때 operand를 감산하여 next operand에 저장한다. 또한, result_h와 result_l를 각각 multiplier의 상위 64비트, 하위 64비트로 저장한다. Multiplier에서 연산할 값으로 result_l가 0일 경우 result_h가 입력되어 연산할 수 있도록 값을 전달한다. Operand가 2일 경우 곱셈 연산을 종료할 수 있도록 LAST로 이동한다. LAST에서는 마지막으로 multiplier의 연산을 수행하며 result_h와 result_l에 result를 저장할 수 있도록 한다. 이후 LAST에 들어온 경우 clear가 되지 않는 이상 DONE으로 이동한다. DONE에서는 factorial의 연산이 종료되었다는 것을 알리기 위하여 opdone의 값을 3으로 설정한다. 즉, opdone을 3으로 설정한다. 이후 result_h와 result_l의 값을 연산 결과에 맞게 값을 저장할 수 있도록 한다. CLEAR는 모든 state에서 opclear이 1이 되었을 경우에 이동할 수 있으며 이때는 opstart, opdone, result_h, result_l를 초기 값으로 초기화할 수 있도록 구현한다. Factorial 연산이 종료된 후 intrEn이 1이고 opdone의 값이 3인 경우에 연산이 종료되었음을 알 수 있으며 interrupt를 발생시킬 수 있도록 AND연산으로 구현한다.

III. Design Details

각 모듈에 대해서 pin을 나타내면 다음과 같다.

Direction	Port name	Bit width	Description
Input	clk	1	Clock
	reset_n	1	Reset when active low
	m_req	1	Master request signal

	m_wr	1	Master write and read signal
	m_addr	16	Master address signal
	m_dout	64	Master data output
Output	m_grant	1	Master grant signal
	m_din	64	Master data input
	interrupt	1	Factorial core interrupt signal

Top module의 pin description이다.

Direction	Port name	Bit width	Description
Input	clk	1	Clock
	cen	1	Chip enable signal
	wen	1	Write enable signal
	s_addr	16	Slave address signal
	s_din	64	Slave data input
Output	s_dout	64	Slave data output

Ram의 pin descriptioon이다.

Direction	Port name	Bit width	Description
Input	clk	1	Clock
	reset_n	1	Reset when active low
	m_req	1	Master request signal
	m_wr	1	Master write and read signal
	m_addr	16	Master address signal
	m_dout	64	Master data output
	s0_dout	64	Slave 0 data input
	s1_dout	64	Slave 1 data input
Output	m_grant	1	Master grant signal
	m_din	64	Master data input
	s0_sel	1	Slave 0 select signal
	s1_sel	1	Slave 1 select signal
	s_addr	16	Slave address signal
	s_wr	1	Slave write and read signal
	s_din	64	Slave data input

BUS의 pin description이다.

Direction	Port name	Bit width	Description
Input	s_address	16	Slave address signal
	m_req	1	Master request signal
Output	s0_sel	1	Slave 0 select signal
	s1_sel	1	Slave 1 select signal

BUS 내부 submodule인 bus_addr의 pin description이다.

Direction	Port name	Bit width	Description
Input	clk	1	Clock
	reset_n	1	Reset when active low
Output	m_req	1	Master request signal
	m_grant	1	Master grant signal

BUS 내부 submodule인 bus_arbit의 pin description이다.

Direction	Port name	Bit width	Description
Input	clk	1	Clock

	reset_n	1	Reset when active low
	s_sel	1	Slave select signal
	s_wr	1	Slave write and read signal
	s_addr	16	Slave address signal
	s_din	64	Slave data output
Output	s_dout	64	Slave grant signal
	interrupt	1	Factorial core interrupt signal

FactoCore의 pin description이다.

Direction	Port name	Bit width	Description
Input	clk	1	Clock
	reset_n	1	Reset when active low
	mul_done	1	Multiplier done signal
	s_sel	1	Slave select signal
	s_wr	1	Slave write and read signal
	s_addr	16	Slave address signal
	s_din	64	Slave data input
	result	128	Multiplier result
Output	mul_start	1	Multiplier start signal
	mul_clear	1	Multiplier clear signal
	s_dout	64	Slave data output
	opstart	64	Opstart register
	opclear	64	Opclear register
	opdone	64	Opdone register
	intrEn	64	intrEn register
	operand	64	Operand register
	result_h	64	Result_h register
	result_l	64	Result_l register
	multiplier	64	Multiplier data in multiplier

FactoCore의 내부 submodule인 FactoCtrl의 pin description이다.

Direction	Port name	Bit width	Description
Input	clk	1	Clock
	reset_n	1	Reset when active low
	s_sel	1	Slave select signal
	s_wr	1	Slave write and read signal
	s_addr	16	Slave address signal
	s_din	64	Slave data input
	opdone	64	Opdone register
	result	128	Multiplier result
Output	opstart	64	Opstart register
	opclear	64	Opclear register
	intrEn	64	intrEn register
	operand	64	Operand register
	s_dout	64	Slave data output

FactoCore의 내부 submodule인 FactoDecoder의 pin description이다.

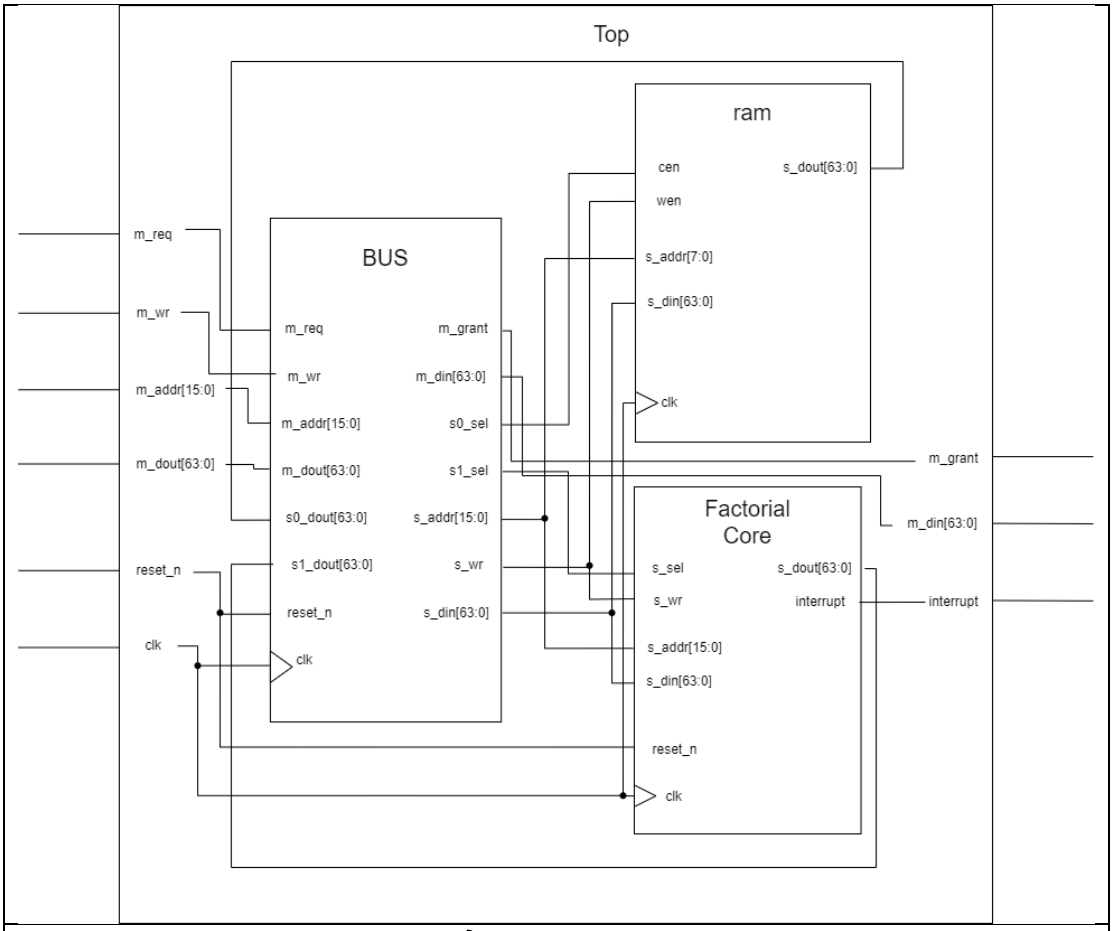


그림 1. Block Diagram

각 모듈을 Top에 instance한 block diagram이다. Top이 testbench에서 m_req, m_wr, m_addr, m_dout, reset_n, clk를 받아 master interface로써 bus에 값을 전달한다. 이후 bus에서 select 값에 따라서 ram과 factorial core에 접근하여 값을 주고받을 수 있도록 한다. 이때 m_grant와 m_din는 bus에서 top으로 값을 출력하고 interrupt는 factorial core에서 값을 testbench로 바로 전달할 수 있도록 구현한다.

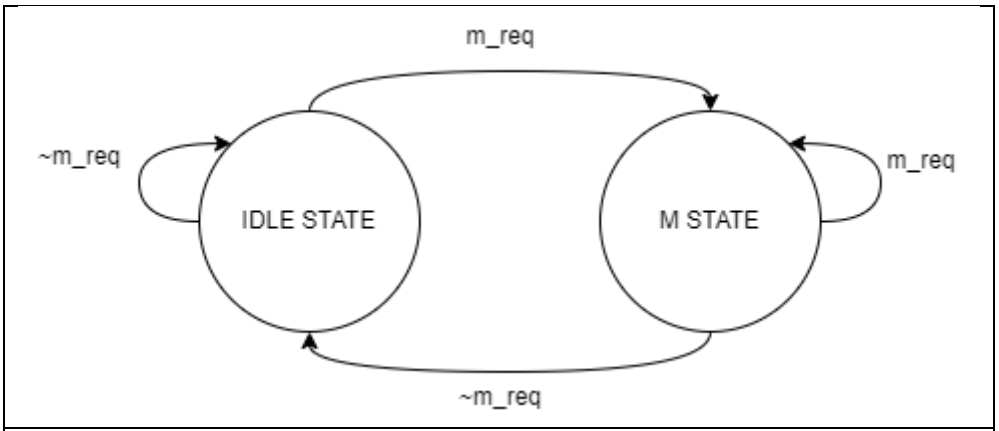


그림 2. Bus Arbiter FSM

Bus의 arbiter에서 사용된 FSM을 나타낸 것이다. IDLE state와 M state로 나뉘어 m_req에 따라서 다음 state로 이동한다.

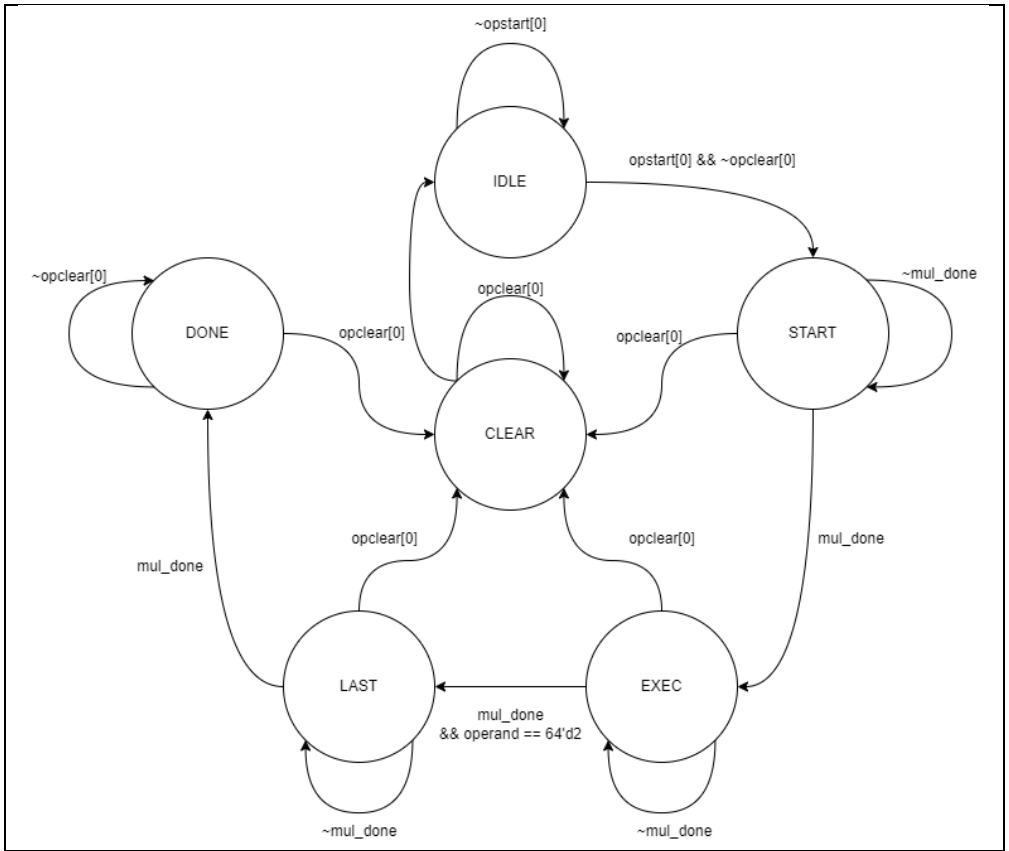


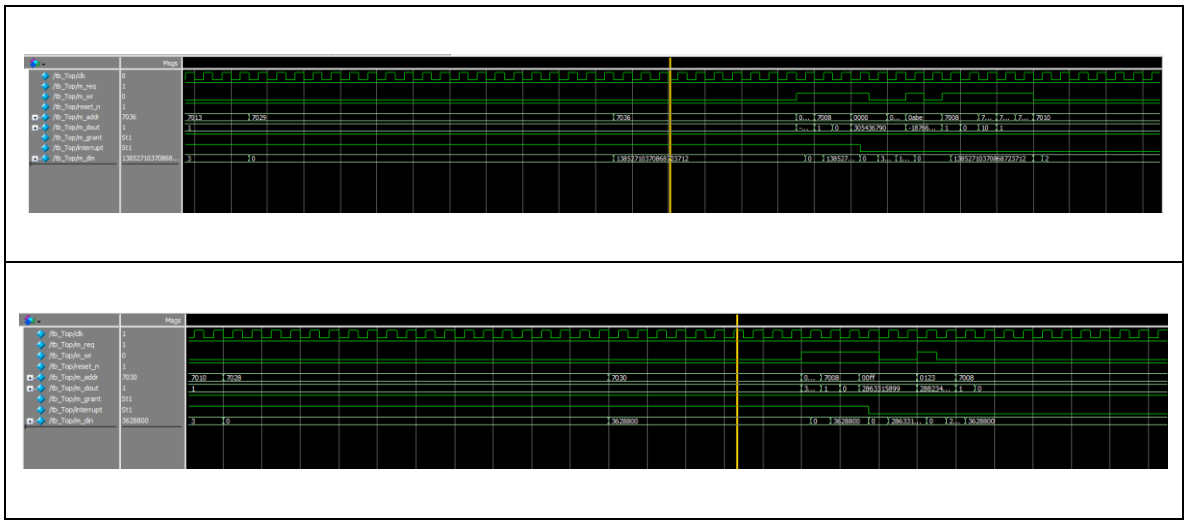
그림 3. Factorial controller FSM

Factorial controller에서 사용된 FSM을 나타낸 것이다. 총 6개의 state로 IDLE, START, EXEC, LAST, DONE, CLEAR로 나타낼 수 있다. IDLE state는 초기 상태를 나타내는 state이다. 이때 factorial core에서 opstart에 1이 들어오면 START state로 이동한다. START, EXEC, LAST state는 모두 multiplier에서 mul_done 신호가 들어왔을 때 다음 state로 이동한다. START는 EXEC, EXEC는 LAST로 이동한다. LAST는 DONE으로 이동하지만 이때 operand가 2가 되었을 때 EXEC에서 LAST로 이동하고 mul_done이 되었을 때 LAST는 DONE으로 이동한다. DONE은 opclear이 1일 때까지 DONE을 유지하도록 구현한다. IDLE state를 제외한 모든 state는 opclear에 1이 들어오면 CLEAR state로 이동하도록 한다. CLEAR state는 초기화한 후 state를 IDLE로 이동하여 초기 값으로 되돌린다.

IV. Design Verification Strategy and Results

Top module은 값을 reset으로 모든 신호를 초기화한다. 이후 offset으로 opclear를 1로 바꾸어 factorial core의 모든 register를 초기화한다. 이후 operand에 값을 입력하여 factorial 연산을 수행할 수 있도록 한다. intrEn에 1을 주어 interrupt가 발생하는 지를 확인한다. Opstart에 값을 1로 입력하여 factorial 연산을 시작하도록 한다. 이후 operand에 맞게 multiplier 및 factorial core가 연산을 마칠 수 있는 충분한 시간을 입력한 뒤 결과 값을 확인한다. 이후 주소를 변경하여 factorial core에서 ram으로 이동하도록 한 후 입력된 결과 값을 사용자가 원하는 주소에 입력되는 지 확인한다. 이후 다른 주소에도 정상적으로 값이 입력되는 지 계속 주소를 바꾸며 값을 확인한다. 이때 64비트의 값을 주고받으므로 metastability [1]가 발생할 수 있기 때문에 clk에 따라 값을 모두 변화시키지 않도록 작성한다.

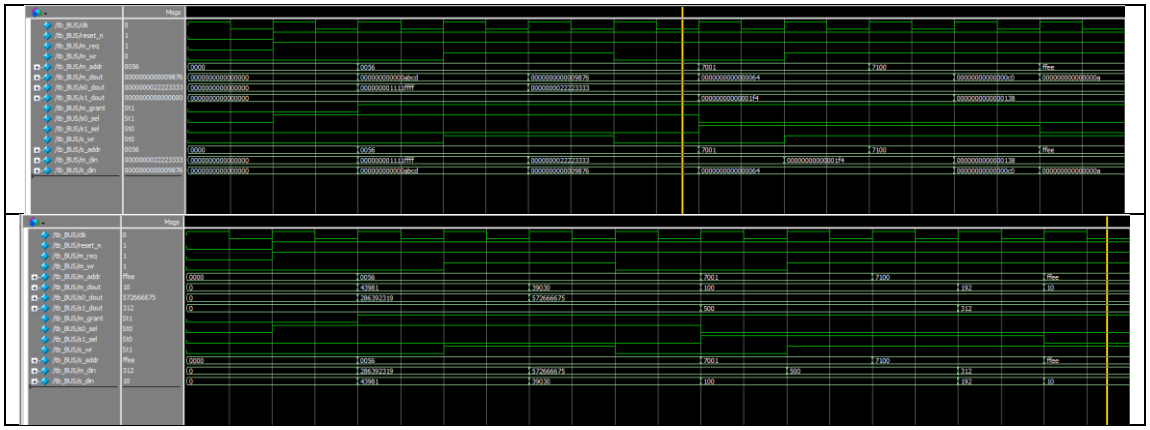
Top module을 검증하는 방법이다. Operand를 616과 10으로 입력하여 연산을 진행하고 이를 메모리에 저장하였을 때의 결과를 확인한다. 또한 ram이 정상적으로 작동하는 지 확인하기 위해서 그 값을 bus에서 입력 받아 저장한다.



Top에서 입력 받은 대로 값을 출력하는 것을 알 수 있다. 616!에 대한 연산 결과가 정상적으로 출력되며 bus에서 신호를 받아 memory에 저장하여 읽는 것도 제대로 작동하는 것을 알 수 있다. 이후 clear를 통해 값을 초기화한 후에 10!에 대한 연산 결과도 올바르게 나오는 것을 알 수 있다.

BUS module은 값을 reset을 통해 모든 신호를 초기화한다. 이후 m_req에 1을 입력하여 master interface가 접근해도 되는 지에 대하여 확인한다. 이후 slave 0에 대해서 접근하기 위해 주소 값을 설정한 뒤 값을 입력하여 제대로 값을 write하고 read할 수 있는 지 확인한다. Slave 1에 대해서도 접근하여 제대로 값을 write하고 read할 수 있는 지 확인한다. 이후 slave에 대한 주소가 올바르지 않은 경우에 slave에 접근을 하지 않는 지 확인한다.

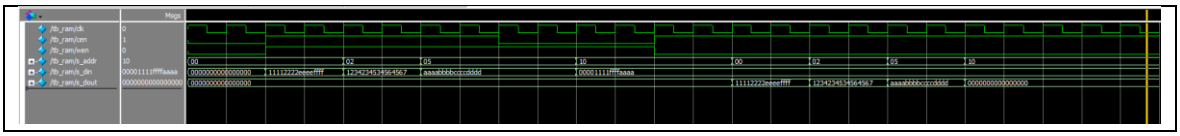
BUS module을 검증하는 방법이다. 값을 초기화한 후 master에서 접근을 할 수 있도록 신호를 준다. 이후 각 slave에 접근하도록 주소를 정한 뒤 임의의 값을 입력하여 제대로 출력하는 지 확인한다. 주소가 올바르지 않을 경우에 slave에 접근하지 않았는 지 확인한다.



BUS에서 입력 받은 대로 m_req가 1일 때 주소 값에 따라 slave에 접근하여 값을 write하거나 read하는 것을 알 수 있다. 이때 slave에 올바른 주소가 입력되지 않았을 경우 slave에는 접근하지 않으나 bus로써 값이 입력되는 것은 알 수 있다.

Ram module은 reset이 존재하지 않고 내부에서 for문으로 초기화하기 때문에 memory를 초기화할 필요성은 없다. 각 신호를 입력으로 초기화한 뒤 chip enable과 write enable에 입력을 하여 ram을 작동하도록 한다. Ram이 활성화될 때 write를 확인하고 read를 확인한다.

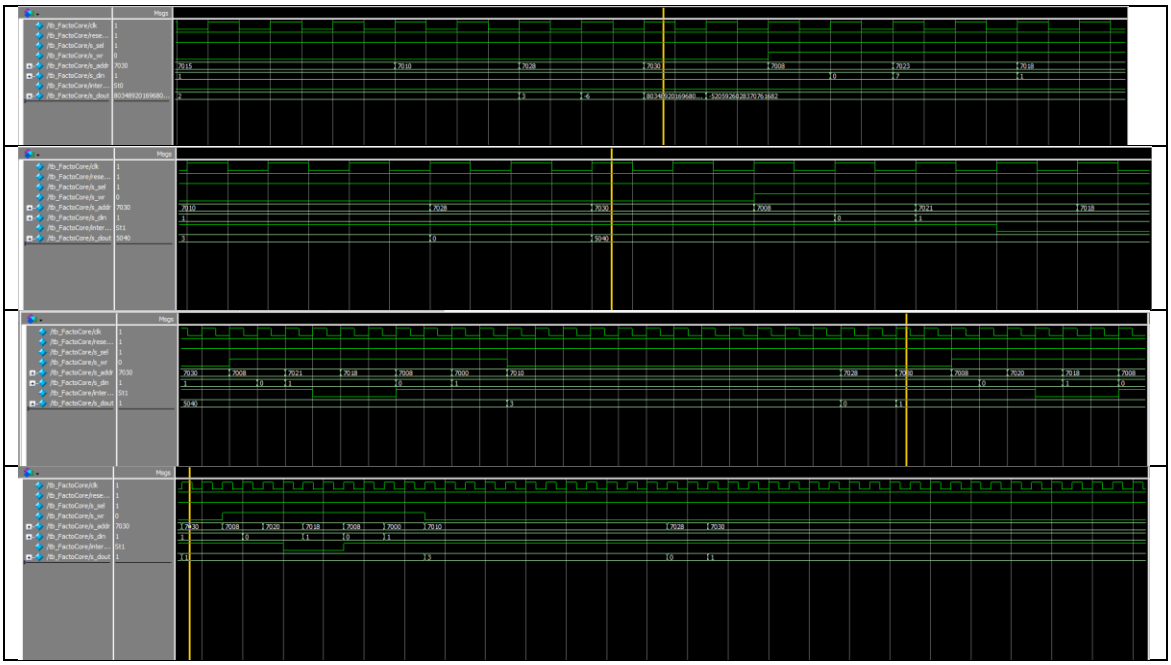
Ram을 검증하는 방법이다. 값을 초기화한 후 memory에 cen과 wen을 1로 주어 특정 주소 값에 맞게 입력한 값을 제대로 write를 할 수 있는 지 확인한다. 이후 ram을 비활성화 하여 값을 입력해본 후 read를 통해 제대로 값을 write하고 cen이 0일 때 ram이 비활성화한 지 확인한다.



Ram에서 입력 받은 대로 cen과 wen이 1일 때 memory에 write하도록 입력한다. 입력 받은 주소 값에 따라서 memory에 저장하고 cen이 0일 때 값을 write한다. 이후 cen을 1로 wen을 0으로 바꾸어 read하였을 때 read를 제대로 수행하는 지 기존에 write했던 주소 값에 따라서 값을 출력할 때 cen이 0일 때 write한 주소 값은 write하지 않았던 것을 알 수 있다.

FactoCore module은 reset을 통해 값을 초기화한 다음 factorial 연산을 수행하도록 한다. Factorial core에 맞는 주소 값을 입력 받으면 해당하는 offset에 따라서 register에 값을 입력하고 opstart가 1이 되면 연산을 시작하도록 한다. 입력 순서는 operand, intrEn, opstart로 연산할 operand와 interrupt 발생 여부, 연산 시작을 기준으로 한다. 이후 multiplier에서 연산을 진행하고 factorial 연산이 종료될 때까지 cycle을 기다리다가 result를 각각 result_h와 result_l에 저장한다. 다음 factorial 연산을 진행하기 위해서 offset에 opclear에 맞는 주소 값을 주고 factorial core를 초기화한 뒤 새로운 factorial 연산을 수행한다. 이를 반복하여 모든 연산이 제대로 수행되는 지 확인하고 0!과 1!에서도 제대로 예외처리가 되는 지 확인한다.

FactoCore를 검증하는 방법이다. 값을 초기화한 후 factorial 연산을 진행하기 위해서 operand를 입력 받고 intrEn을 입력 받는다. 이후 opclear를 0으로 설정한 뒤에 opstart를 1로 주어 연산을 시작하도록 한다. 이때 21!까지의 수는 64비트를 모두 사용하는 값이므로 21! 이후의 연산을 알기 위해서 더 큰 수인 483을 준다. 연산이 완료될 때까지 충분한 delay를 주어 cycle에 따라서 값을 연산을 하고 완료된 결과를 저장한다. 이후 factorial core를 opclear를 통해 초기화한 후 새로운 factorial 연산을 시작할 수 있도록 하여 7!, 1!, 0!을 계산할 수 있도록 한다. Radix-2로 구현하여 cycle을 계산한 결과는 operand에 clock 주기와 비트 수를 곱하였다. 따라서 이를 정리하면 operand * clock cycle * bit width로 cycle을 계산할 수 있다.



FactoCore에서 입력 받은 operand로 483, 7, 1, 0을 입력한다. 483과 7에 대해서는 연산을 진행하면서 factorial에서 올바르게 연산을 한다. 이후 1과 0을 입력했을 때 1과 0은 multiplier 연산을 하지 않기 때문에 cycle을 크게 기다리지 않고 바로 값을 출력하게 작성하였다. 이때 결과값이 올바르게 예외 처리된 것을 알 수 있다.

Flow Status	Successful - Tue Dec 05 22:19:09 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	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	8,879 / 41,910 (21 %)
Total registers	16863
Total pins	150 / 499 (30 %)
Total virtual pins	0
Total block memory bits	0 / 5,662,720 (0 %)
Total DSP Blocks	0 / 112 (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 / 15 (0 %)
Total DLLs	0 / 4 (0 %)

그림 4. Flow summary

Top module을 compile 했을 때의 flow summary이다. Logic utilization은 8879개이며 register는 16863개를 사용했다. Ram에서 64비트의 register를 총 256개를 사용하여 16384개를 사용하며 이외 다른 모듈에서 사용하는 64비트의 다른 register 및 일부 register를 사용한 결과이다. 사용된 pin은 150개로 Top에서 사용된 pin의 총 비트 수이다.

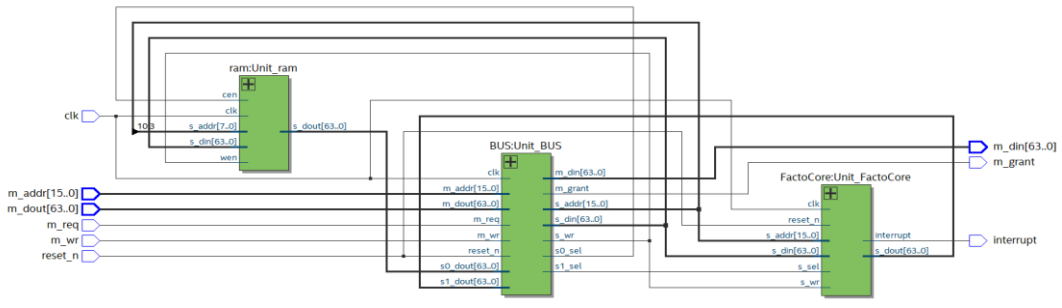


그림 5. Top module

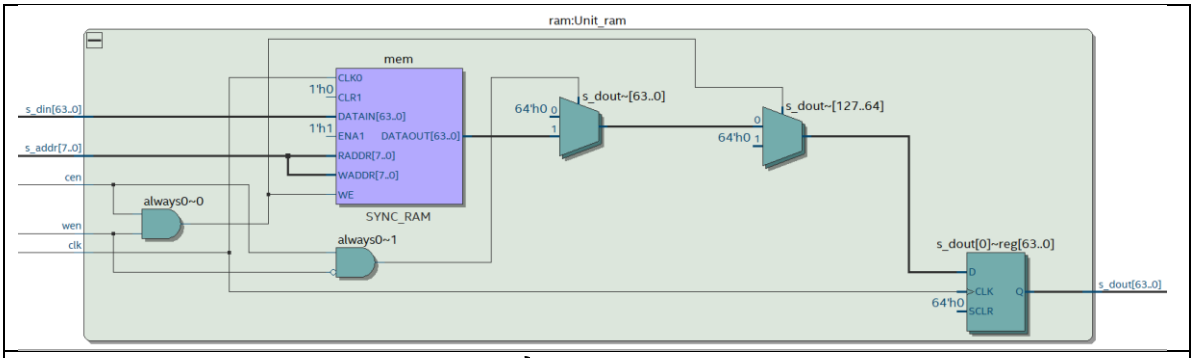


그림 6. ram module

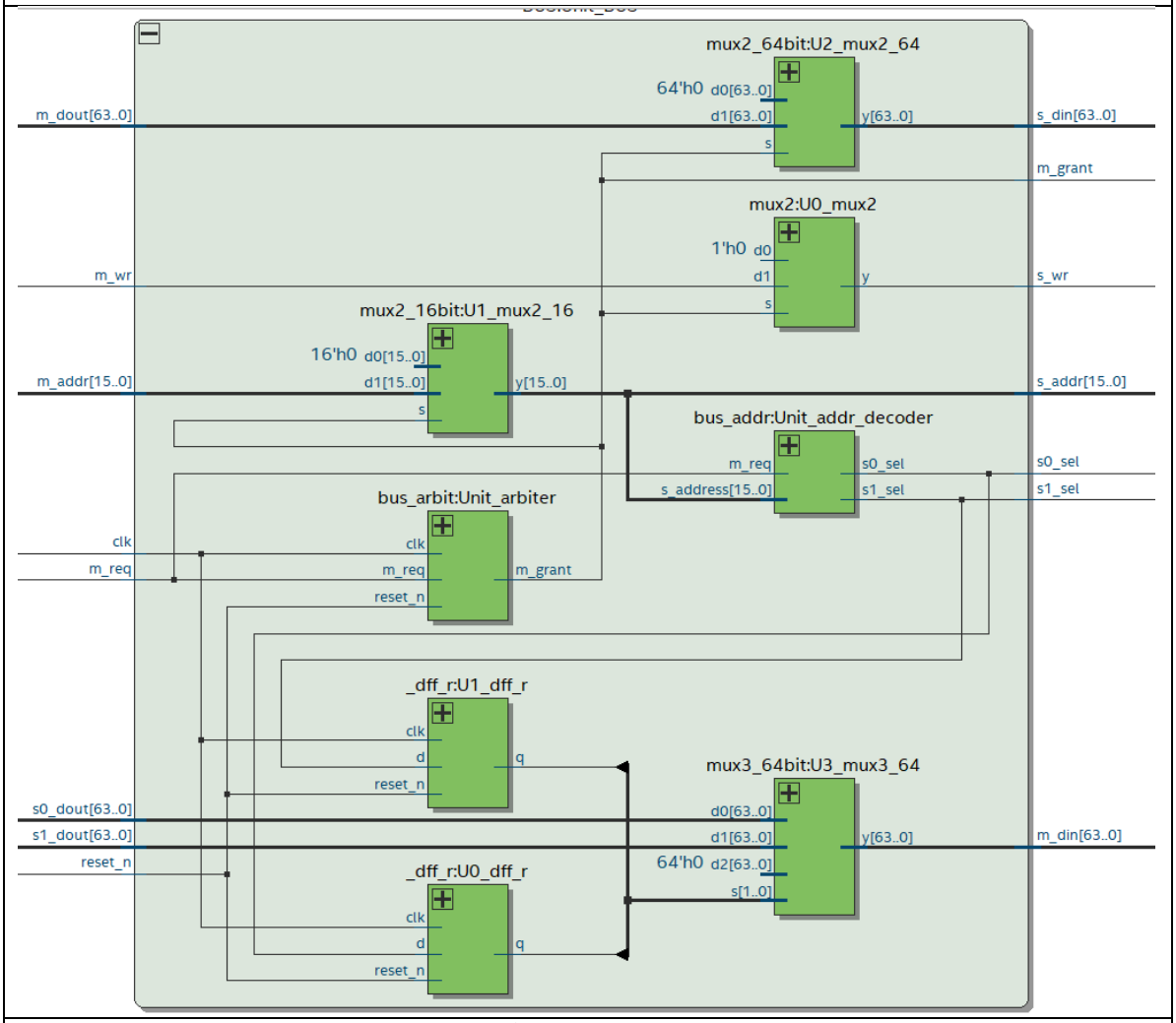


그림 7. BUS module

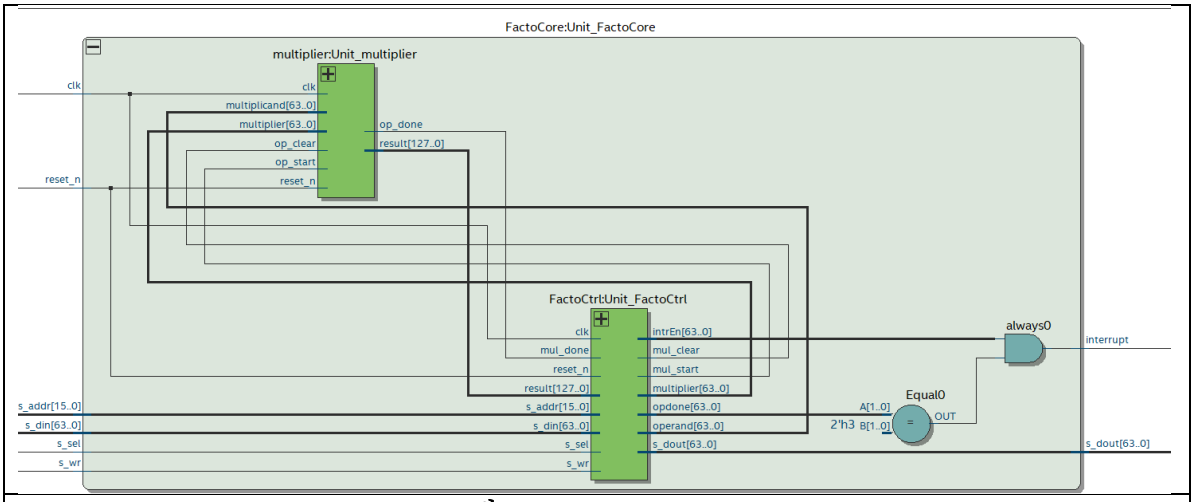


그림 8. FactoCore module

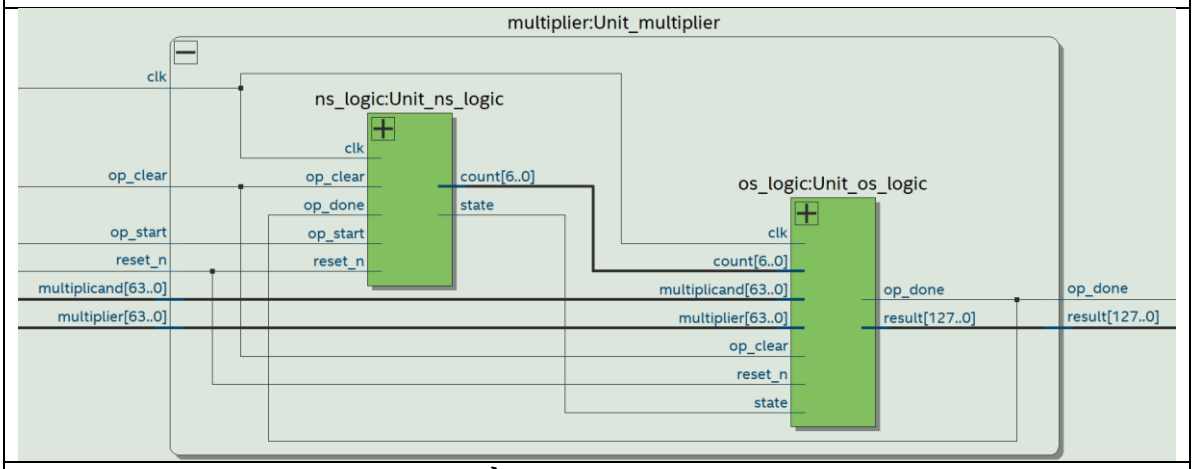


그림 9. multiplier module

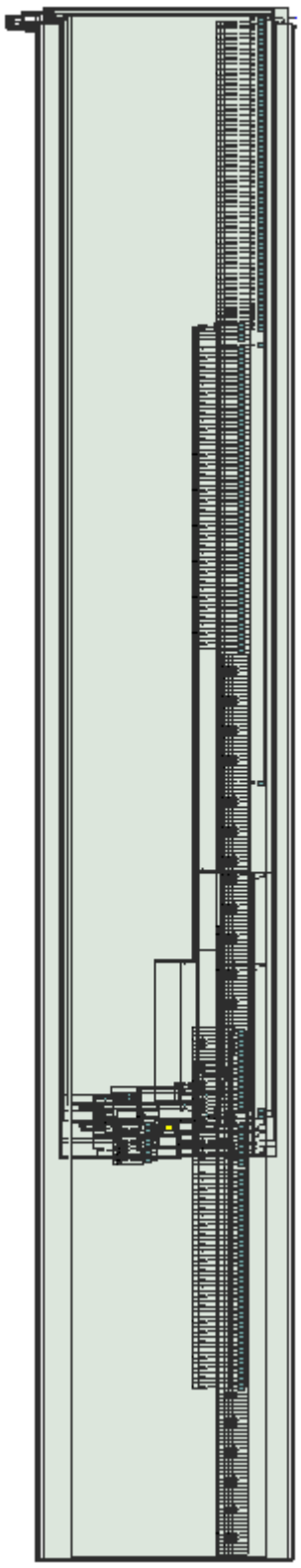


그림 10. FactoCtrl module

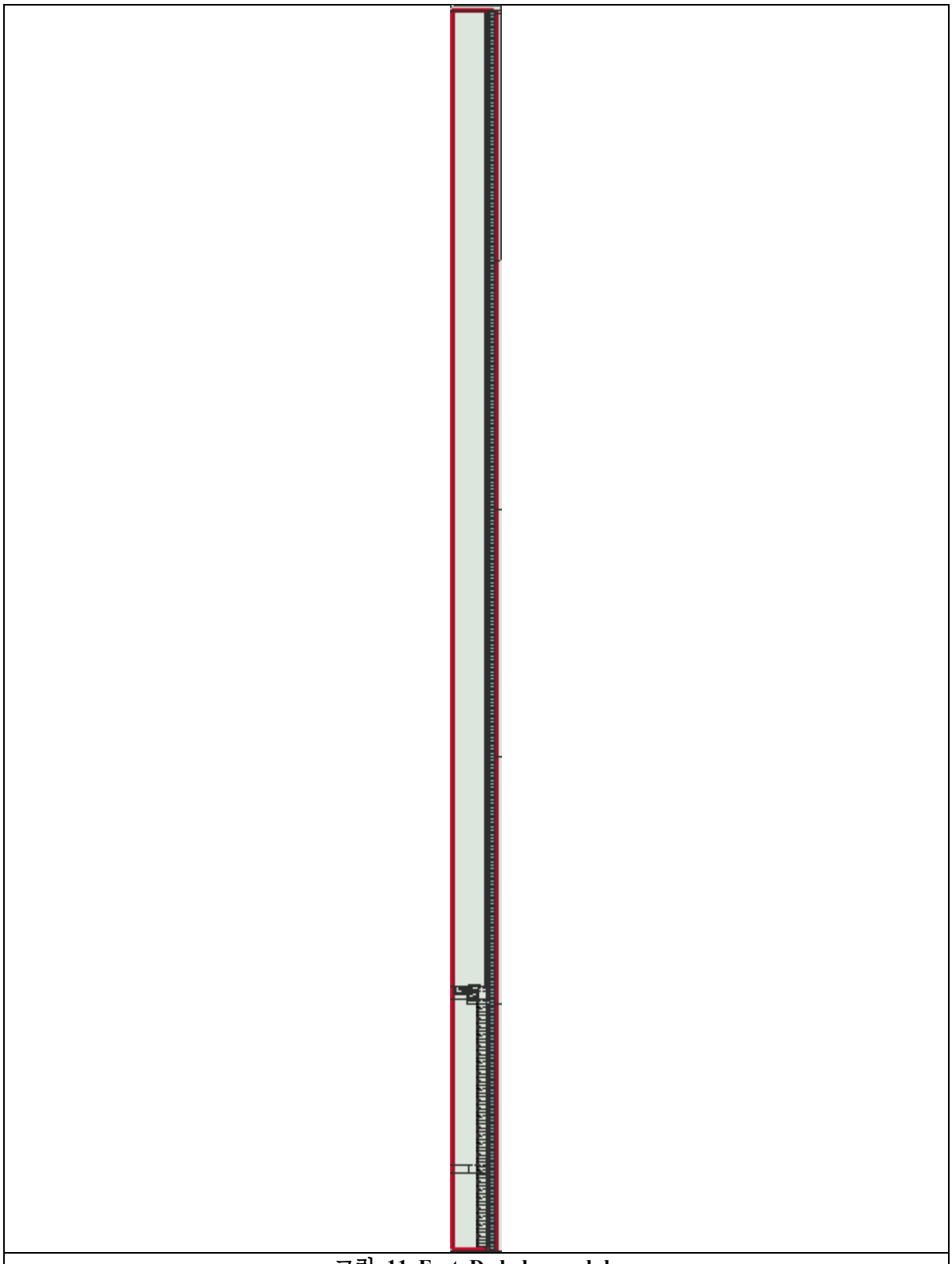


그림 11. FactoDedoder module

각 모듈에 대한 RTL Viewer이다. Top module은 BUS, ram, FactoCore를 instance한 모듈이며 Top에서 받은 input을 BUS로 전달하는 것을 알 수 있다. BUS는 arbiter와 decoder로 주소에 맞게 값을 전달하고 flip flop과 mux를 통하여 slave로 값을 주고받는 것을 알 수 있다. Ram은 전달받은 데이터를 내부 register에 저장하는 모습을 알 수 있다. FactoCore는 전달받은 주소 값을 통하여 FactoCtrl과 multiplier간의 상호작용을 통하여 연산을 진행한다는 것을 알 수 있다.

Factorial controller와 factorial decoder에서의 RTL Viewer는 next state logic, output logic으로 상당히 많은 게이트가 mux와 게이트를 사용한 것을 알 수 있다. Factorial controller 내부에 instance한 factorial decoder도 마찬가지로 offset 조건에 맞게 register에 값을 주고받을 수 있도록 설계되어 많은 logic gate를 사용하는 것을 알 수 있다.

Slow 1100mV 85C Model

	Fmax	Restricted Fmax	Clock Name	Note
1	43.84 MHz	43.84 MHz	m_addr[10]	
2	110.79 MHz	110.79 MHz	clk	
3	139.86 MHz	139.86 MHz	FactoCore:Unit_FactoCore|FactoCtrl:Unit_FactoCtrl|state.CLEAR	
4	144.09 MHz	144.09 MHz	FactoCore:Unit_FactoCore|FactoCtrl:Unit_FactoCtrl|opclear[0]	
5	151.47 MHz	151.47 MHz	FactoCore:Unit_FactoCore|FactoCtrl:Unit_FactoCtrl|state.EXEC	
6	191.2 MHz	170.65 MHz	FactoCore:Unit_FactoCore|FactoCtrl:Unit_FactoCtrl|operand[10]	lim...eck

그림 12. Maximum clock frequency

```
#  P          *        *
#  A        * *      * *
#  S       *   *    *   *
#  S
#  E          *      *
#  D          *      *
#  !          *******
# Total calculation cycle =      277769 cycles
# ** Note: $stop    : C:/intelFPGA_lite/18.1/Project/FactorialTest/tb_FactorialCoreTop.v(20)
#    Time: 5568831 ps  Iteration: 0  Instance: /tb_FactorialCoreTop
```

그림 13. Calculation cycle

Throughput을 계산하기 위해서 timing analysis를 통해서 분석한 결과이다. Clk에 따른 Fmax 결과는 110.79 MHz이며 연산 중 일어난 cycle은 277769이다. Throughput을 계산하는 방법은 testbench에서 사용한 case 20개를 cycle의 수로 나눈 뒤 clk의 주기를 곱하는 것이다. 이를 식으로 정리하면 $\frac{20}{cycle} * maximum\ clk\ frequency$ 이다. 이를 바탕으로 계산하면 다음과 같은 결과를 얻을 수 있다. Throughput = $\frac{20}{277769} * 110.79 * 10^6$ Hz = $7.98 * 10^3$이다.

V. Conclusion

Factorial computation system을 구현한 결과로 testbench 결과 연산에 문제가 없는 것을 알 수 있다. Multiplier를 구현할 때 radix-2를 이용하여 radix-4와 radix-16과 비교하여 상당히 많은 cycle을 소모하는 것을 알 수 있지만 적은 area를 사용하는 것을 알 수 있다.

Memory에서 256개의 data를 각각 64비트만큼 저장할 수 있어야 하기에 16384개의 register가 필요하다. 각 주소별로 register를 사용해야 하는데 모든 주소 값에 맞게 data를 저장할 수 있고 필요할 때 불러올 수 있기 때문에 사용한다는 점을 알 수 있다. BUS에서 memory와 factorial core 간의 상호작용으로 data를 주고받을 수 있어 이를 사용자가 제대로 입력만 해준다면 여러 파일을 불러올 필요 없이 한 모듈, 즉 Top 모듈만으로도 충분히 여러 연산을 할 수 있다는 장점이 있다. Factorial연산을 흉내내는 모듈로 21!까지만 쉽게 결과 값을 알 수 있지만 그 이후 연산에 대해서는 곱셈기로써의 역할을 수행하는 것으로 알 수 있다.

Top module에서 각 모듈을 instance함으로써 이에 대한 연산을 모두 사용자가 관리할 수 있다는 점이 bus를 이용한 장점이라고도 할 수 있다. Top module 이외에 다른 master interface가 있다면 이를 통한 새로운 연산도 구현할 수 있기 때문에 bus를 통해서 arbiter를 내부에 구현하고 이를 m_req에 따라 구현한 것이 장점이라고 할 수 있다. 그에 맞게 input을 수정하겠지만 유동적으로 구현할 수 있다는 것에서 Top에서 즉시 ram과 factorial core로 가지 않았을 때의 장점이다. Top module에서 즉시 ram과 factorial core에 접근하였다면 사용자가 모든

신호에 대해서 주기적으로 주소 값을 수정하고 연산에 대한 cycle을 고려하여 수정해야 했을 것이다.

Top module을 구현하면서 발생한 문제점의 대부분은 모두 factorial core에서 발생한 문제이다. FactoCore module에서 register를 offset에 맞게 값을 입력하려고 할 때 여러 문제점이 발생하였다. Next state logic과 output logic으로 구현하려는 결과 각 state에 따라서 발생하는 output에 조건들을 설정하는 데에 문제가 발생하였다. 이 문제들을 FSM에 맞게 설계하고 구현하는 과정에서 기존에 구현하려던 state는 4개였지만 multiplier의 연산을 진행하면서 시작할 때와 operand가 2일 때 끝이 나므로 종료할 때의 state를 추가하여 따로 연산을 진행하였다. 이와 같은 이유 때문에 state가 2개 늘어나게 되어 총 6개로 사용하게 되었다. BUS와 ram에서는 state를 각각 4개와 3개로 작성하였지만 이들을 실제로 구현할 때 이를 FSM으로 표현하는 방법과 실제로 state를 사용한 것이 다르기 때문에 실제로 사용한 state에 대해서만 FSM을 작성하였다. 또한 multiplier와 factorial controller 간의 상호작용으로 값을 주고받을 때 multiplier 내부의 opstart, opclear, opdone 신호와 factorial core 자체의 신호가 다르기 때문에 이를 다르게 선언하여 주고받을 수 있도록 구현하였다. 이를 통해 multiplier의 연산이 종료되었을 때 다음 연산을 시작할 수 있도록 설계하고 올바르게 factorial 연산이 되는 것을 알 수 있었다. 기존에 이를 고려하지 않았을 때는 factorial 연산을 진행하는 도중 state에서 계속 operand를 즉시 빼는 결과가 발생하여 multiplier의 1 cycle이 지날 때 factorial controller도 1 cycle이 지나 operand와 multiplier의 연산이 맞지 않는 결과를 볼 수 있었다. 이후 제대로 작동하도록 수정하고 operand가 0 또는 1일 경우에 발생하는 예외 처리를 해주었다. 또한 factorial decoder에서 s_dout을 출력할 때 clock에 상관없이 asynchronous하도록 구현하기 위해서 clk에 대한 always 신호를 받지 않도록 구현하였다. 기존에는 clock에 synchronous하게 구현하여 clock에 따라서 신호가 변하였는데 이를 바꾼 것이다.

VI. Reference

[1] 유지현, 디지털논리회로1, Metastability, 광운대학교, 2023