

Factorial Computation System

(2020202054) (이종혁)

Abstract

디지털 논리 term project를 최종 보고서 작성을 위한 기본 template format입니다. Abstract는 해당 보고서의 내용을 전체적으로 요약하는 부분입니다. 이 뒤로 introduction, project specification, design details, design verification strategy and results에 대하여 각각의 해당하는 session에 작성하여 주시기 바랍니다. 마지막에 references는 자기가 참조한 서적, 논문 등에 대하여 정리하는 것입니다. References의 내용을 보고서에 적을 때는 예를 들어 ‘multiplier는 ~~~하는 장치이다. [1]’ 이런 식으로 적어주시면 됩니다.

I. Introduction

이번 프로젝트는 궁극적으로 Factorial 연산을 하는 Factorial Core와 값을 저장하고 읽을 수 있는 Memory를 Bus를 통해 연결하여 적절한 기능을 할 수 있도록 하는 것입니다. Bus를 통해 한 개의 Master인 Testbench가 2개의 Slave(Factorial Core/Memory) 중 하나에 요청을 하게 됩니다. 그 요청에 대해서 Bus가 Master에 Grant를 부여하면, 그 이후로 Master가 두 Slave에 접근할 수 있습니다. 동시에 접근은 불가능하고, Master가 접근하려는 Address에 따라서 해당 Address에 부합하는 Slave에 접근하게 됩니다.

Memory(RAM)에 경우, 64-bits의 Data를 총 256개 저장할 수 있는 구조입니다. cen과 wen 신호에 따라서 Read/Write Mode를 수행할 수 있으며, 메모리를 CLK에 동기화되어 구동이 됩니다.

Factorial Core에 경우, 총 7개의 Register를 통해 동작을 제어하게 됩니다. 각각 Write 전용인 opstart, opclear, intrEn, operand 총 4개의 Register가 있고, Read 전용인 opdone, result_h, result_l 이름을 가진 3개의 Register가 존재합니다. opclear를 통해 초기화를 하거나, intrEn을 통해 interrupt 신호 사용 여부를 결정하며, operand에 원하는 팩토리얼 시작점을 작성하고 opstart를 통해 팩토리얼 연산 시작이 가능합니다. 또한 opdone을 읽어서 팩토리얼 연산 종료 여부를 알 수 있고, result_h와 result_l에 저장된 결과도 확인이 가능합니다. 한 번의 곱셈에서 64-bits의 두 피연산자를 통해 128-bits의 결과를 도출하는 Booth Multiplier를 사용했기 때문에, 한 번의 연산이 끝나면 result를 반으로 나눠서 하위와 상위로 나누어 저장하게 됩니다. 그래서 Factorial Core의 특성 상, 큰 수의 operand가 입력이 된다면, 보통 사람들이 생각하는 Factorial의 결과가 나오지 못할 수도 있습니다.

II. Project Specification

저는 이번 프로젝트에서 Radix-4 Booth Multiplier를 사용했습니다. 그래서 Radix-4 Booth Multiplier의 작동 방식에 대해서 서술해보겠습니다.

Radix-4는 Radix-2와 다르게 총 3-Bits씩 읽어서 상황에 맞는 Operation을 수행하게 됩니다.

Xi	Xi-1	Xi-2	Operation
0	0	0	>>>2
0	0	1	ADD & >>>2
0	1	0	ADD & >>>2
0	1	1	>>>1 & ADD & >>>1
1	0	0	>>>1 & SUB & >>>1
1	0	1	SUB & >>>2

1	1	0	SUB & >>>2
1	1	1	>>>2

(i = 1, 3, 5, 7, ...)

제가 각 3-Bits의 Case별로 수행해야 하는 Operation을 정리한 표입니다. 이렇게 3-Bits씩 해석해서 Operation을 하는 행동을 총 32번 반복하면, Radix-4 Algorithm에 의한 결과가 나오게 되는 것입니다. 32번의 Clock Cycle을 측정하기 위해, MSB에만 1을 넣은 32비트 수를 선언하여 매 사이클마다 Logical Shift Right을 하게 하고, LSB에 1이 올 때 종료를 하도록 하였습니다.

Factorial Core에 경우, 이러한 Booth Multiplier를 사용하여 Operand가 1이 될 때까지 반복하도록 하였습니다. Factorial Core에 대한 설명으로 이어가보도록 하겠습니다.

Factorial Core을 구현할 때 처음으로 중요하다고 생각했던 것은, Register 용도에 따른 예외 처리였습니다. Write 전용 Register는 Master가 절대 읽지 못하게, Read 전용 Register는 Master가 절대 쓰지 못하도록 하는 것입니다. 그래서 각각 clk에 동기화 된 서로 다른 2개의 always문으로 input logic/output logic으로 나누었습니다.

또한, Address Offset에 따른 각 Register로의 접근이 중요하다고 생각했습니다.

op start 7000 → 011 000 000 0000
 |
 1007 → 011 000 000 0111
 ↙ E00

op clear 7008 → 011 000 000 1000
 |
 100F → 011 000 000 1111
 ↙ E01

op done 7010 → 011 000 000 1000
 |
 1017 → 011 000 000 0111
 ↙ E02

...

예시로 3가지의 경우를 적어보았습니다.

하위 3-Bits는 생각하지 않고 해야 알맞은 Offset 범위를 설정할 수 있다고 생각했습니다. 그래서 각 Address offset의 범위를 고려하여 하위 3-Bits는 버리고, 총 13-Bits Hexadecimal로 정리를 해보았더니 E00, E01, E02, ... , E06까지의 수가 나왔습니다. 이 수들을 parameter로 지정하여, offset에 따라 알맞은 Register에 접근할 수 있도록 하였습니다.

총 State는 7개로 이루어져 있으며, State에 대한 자세한 설명은 FSM 부분에서 이어가도록 하겠습니다.

RAM의 경우는 cen과 wen에 대해서 명확하게 선을 그어준다면, 작동하는데 있어서 문제를 일으키지 않을 것이라고 생각했습니다.

맨 처음에 제안서를 읽고 시작할 때, 과연 cen과 wen에 Master의 신호 중 어떤 것을 각각 연결해줘야 하는지에 대해서 고민이 많았습니다. 그런데 cen이 1일 때는 Read/Write 동작을 하고, 0일 때는 아무런 동작을 하지 않는다는 점을 파악하고, cen에는 s0_sel 신호를 연결해주었습니다. 또한 wen이 0일 때에는 Write, 1일 때에는 Read를 한다는 점에 대해서 인지를 하였고 s_wr 신호를 연결하도록 하였습니다.

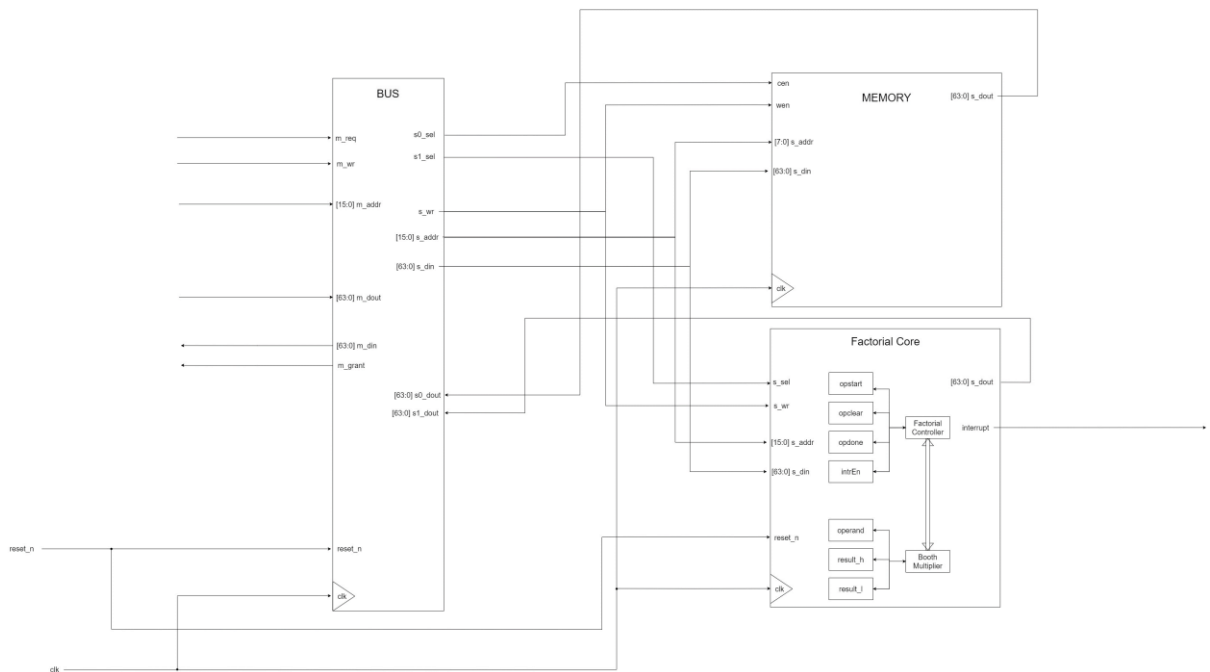
그 후, Clock에 동기화 되는 always문을 사용하여 cen과 wen의 case에 따라 RAM에 Read/Write 혹은 아무런 Operation을 하지 않도록 하였습니다.

BUS를 설계하는 데 있어서는 “값을 어느 시점에 다른 곳에 전달하느냐?”에 대해서 고민을 많이 했습니다.

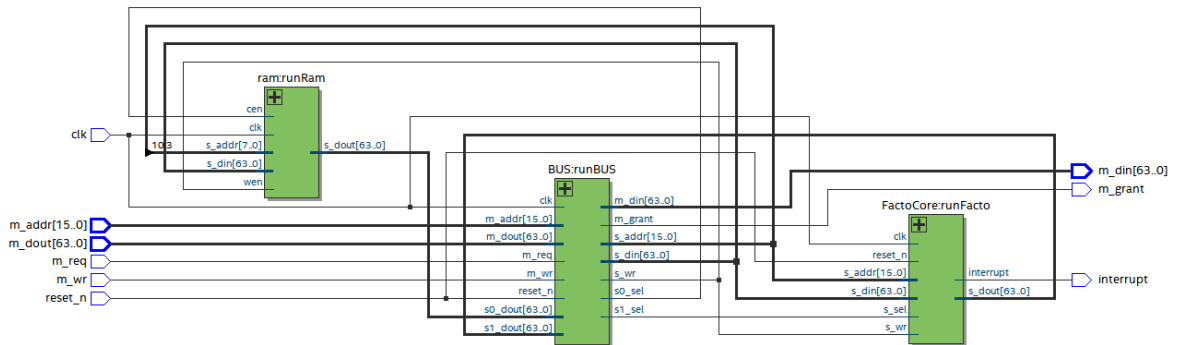
저는 bus_arbit, bus_addr, mux3_64bits, 그리고 BUS 모듈을 통해서 BUS를 구현했습니다. bus_arbit에 경우에는 m_req 신호를 받은 상태, 받지 않은 상태를 통해 Clock에 동기화 되어 m_grant를 내보내도록 하였습니다. 그렇게 synchronous한 m_grant 신호를 받게 되면, m_grant가 1일 때만 항상 Slave에 Master의 값을 전달해주도록 하였습니다.

bus_addr는 m_req의 신호에 따라 m_req가 0이면 두 Slave의 sel 신호를 모두 0, 1인 경우에는 Address를 해석하여 slave 중 어느 곳이 선택되었는지, 혹은 둘 다 선택이 안 되었는지를 판별하였습니다. 이렇게 나온 두 Slave의 sel 값을 Clock에 동기화 시켜서 mux3_64bits에 넣어주었고, 그로 인해 Clock에 synchronous하게 Slave의 값을 Master가 받을 수 있게 되었습니다.

III. Design Details

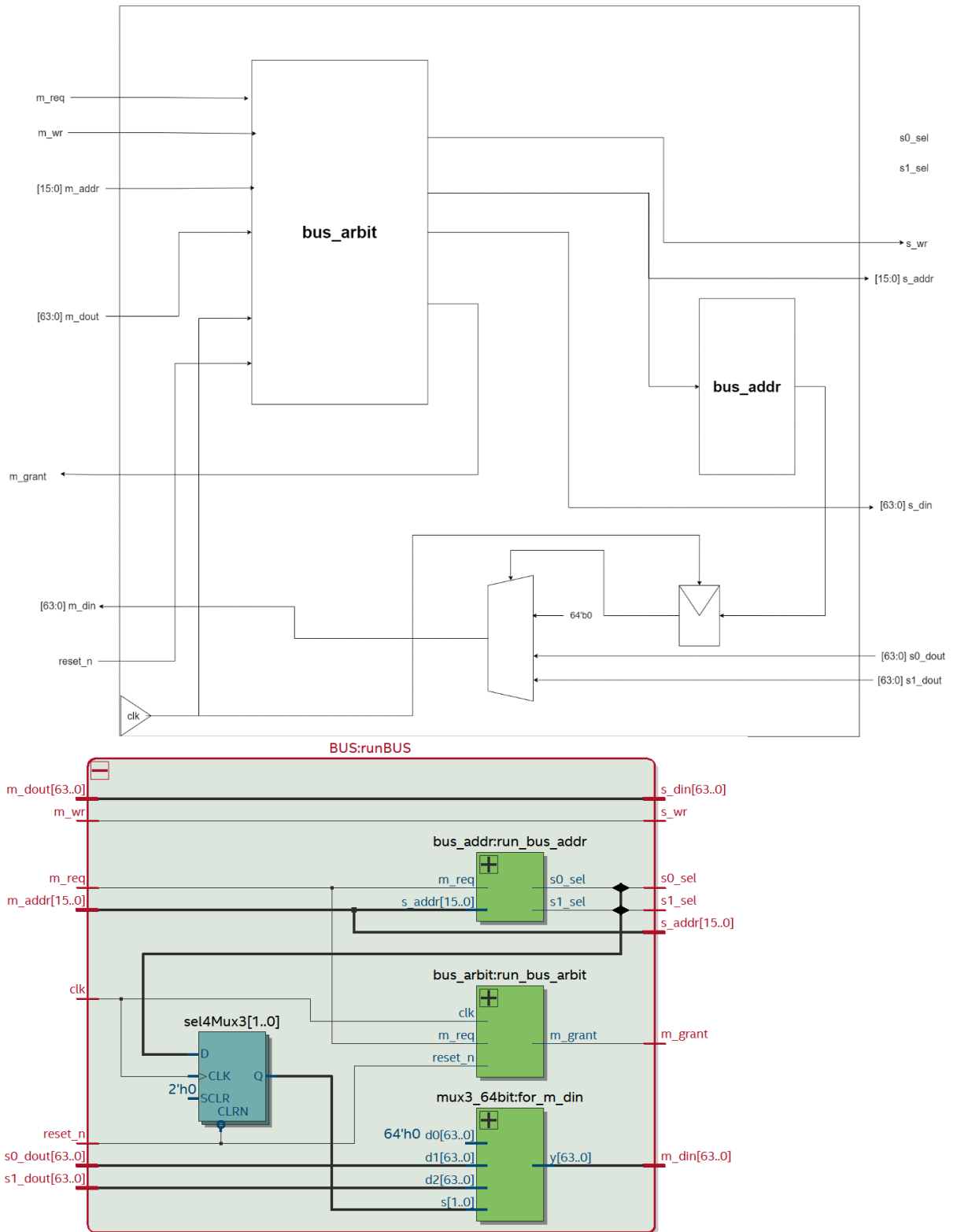


이것이 이번에 제가 설계한 Top 모듈의 전체 Block Diagram입니다.

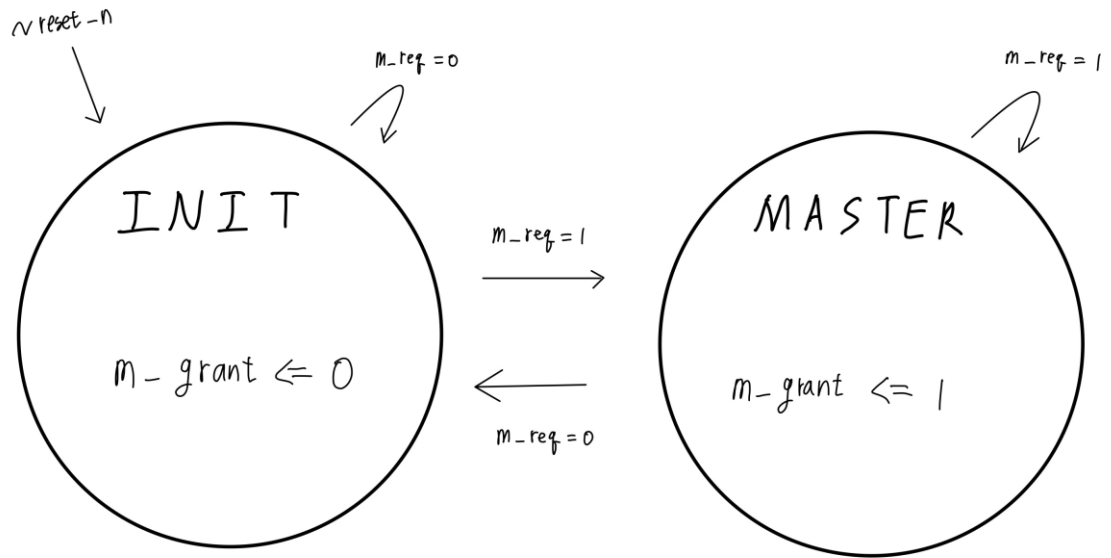


실제 나온 제 RTL Viewer와 동일한 것을 확인할 수 있습니다.

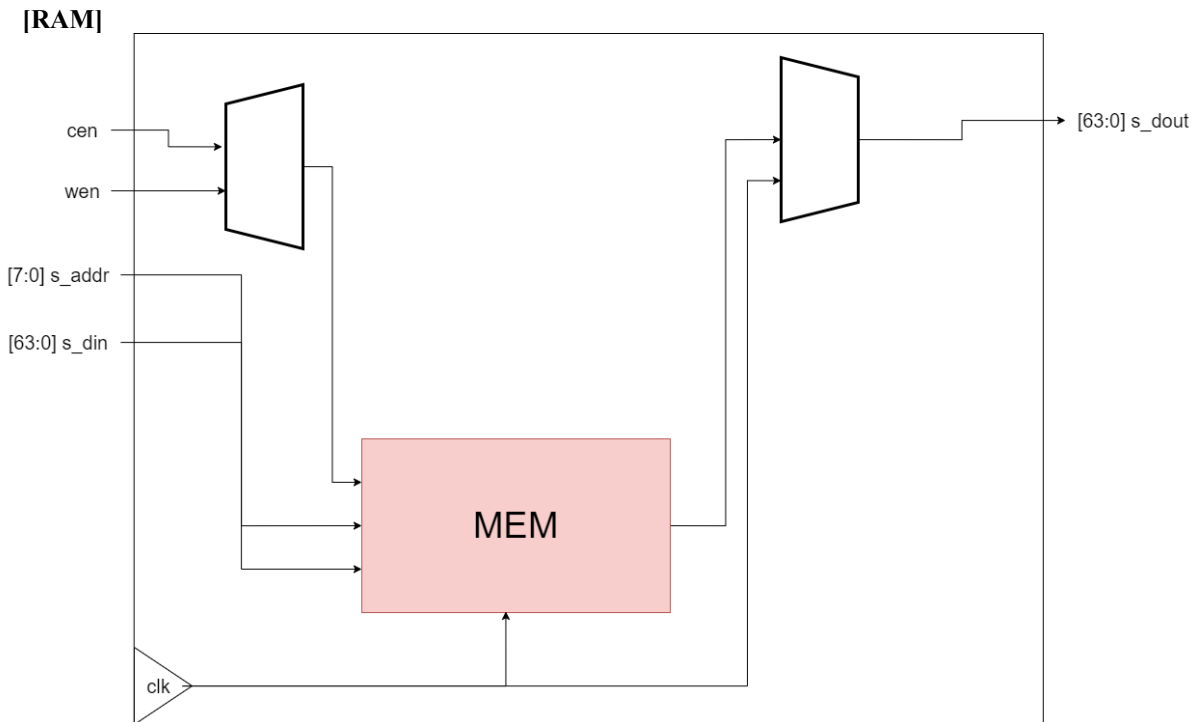
[BUS]

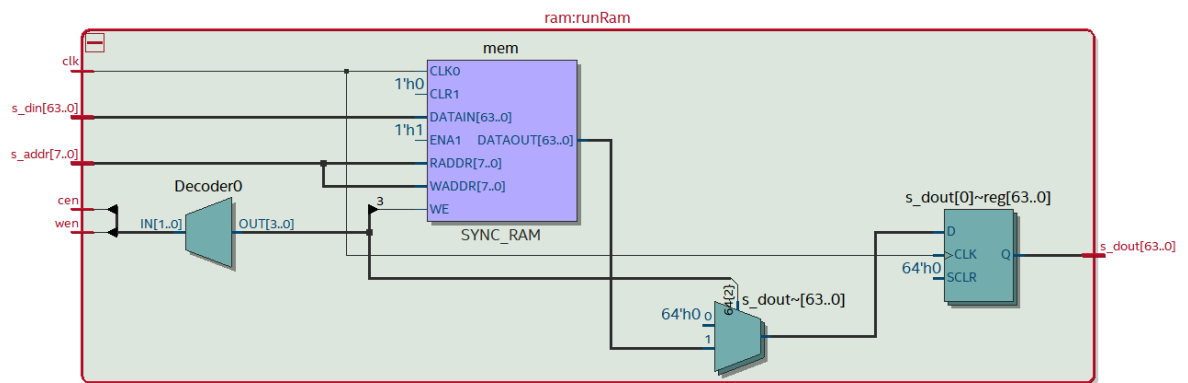


Bus에 경우 Master인 Testbench로부터 Request가 들어오면 Grant를 부여하고, 그 Grant 신호에 맞춰서 Master의 Data들을 Slave들에 전달을 하도록 설계가 되어있습니다. 또한 Master가 입력한 Address에 따라서, Slave 중 하나에 Sel 신호를 부여하며, 이러한 Sel 신호를 Clock 신호에 맞춰 MUX3에 넣어주어 Slave로부터 오는 값을 Master에 선택적으로 전달할 수 있습니다.



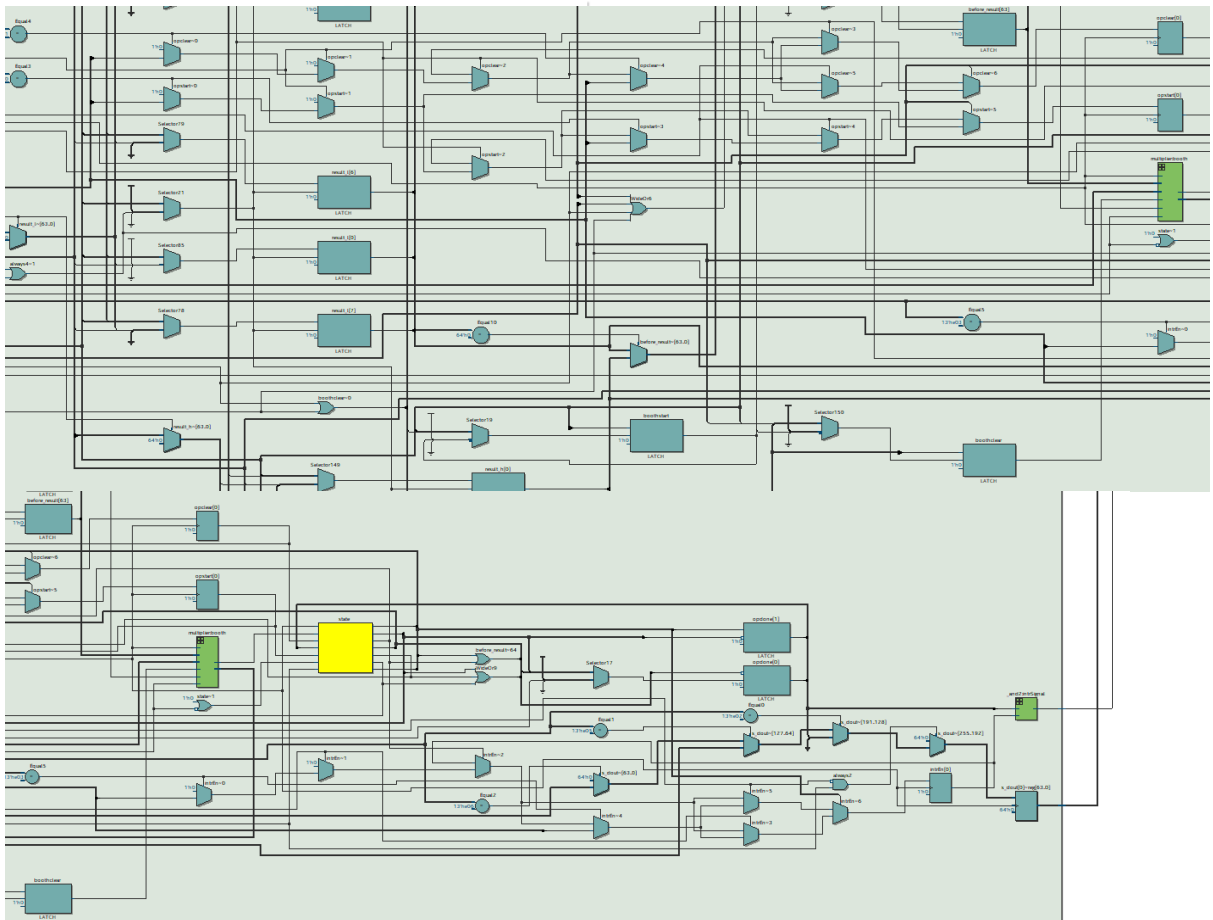
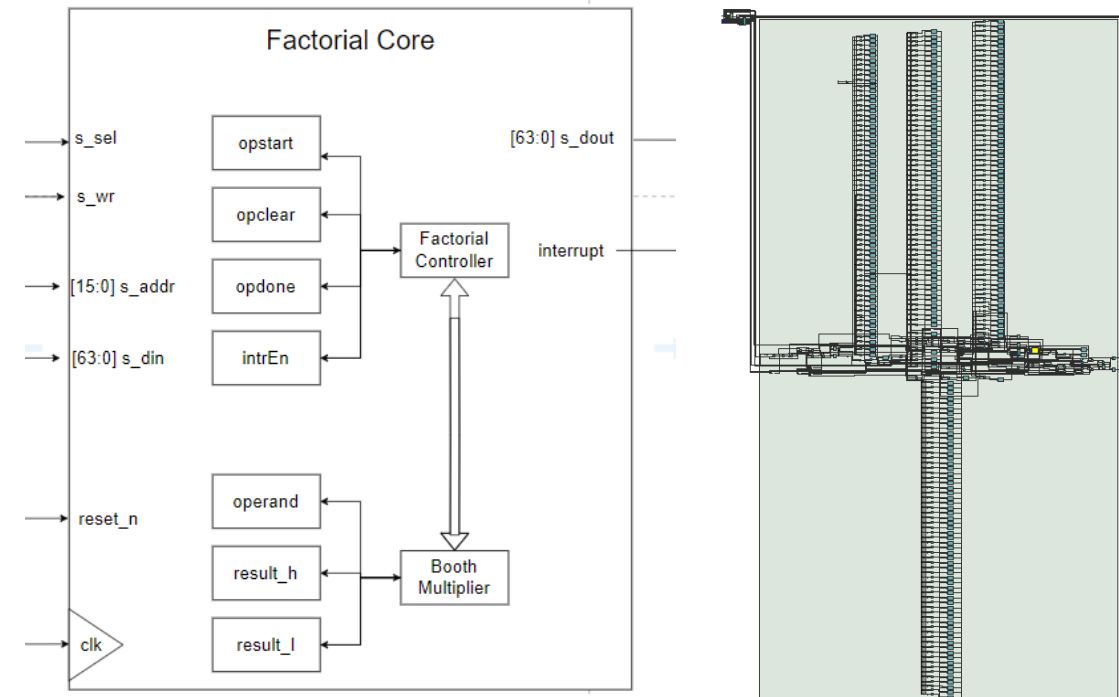
앞서 명시한 것처럼, BUS에서는 bus_arbit 모듈에서만 State를 정의하였습니다. 그러한 State Transition Diagram은 위와 같습니다. m_req 신호에 따라서 Master에 grant를 부여하는 방식의 사실상 간단한 구조로 이루어져 있습니다.



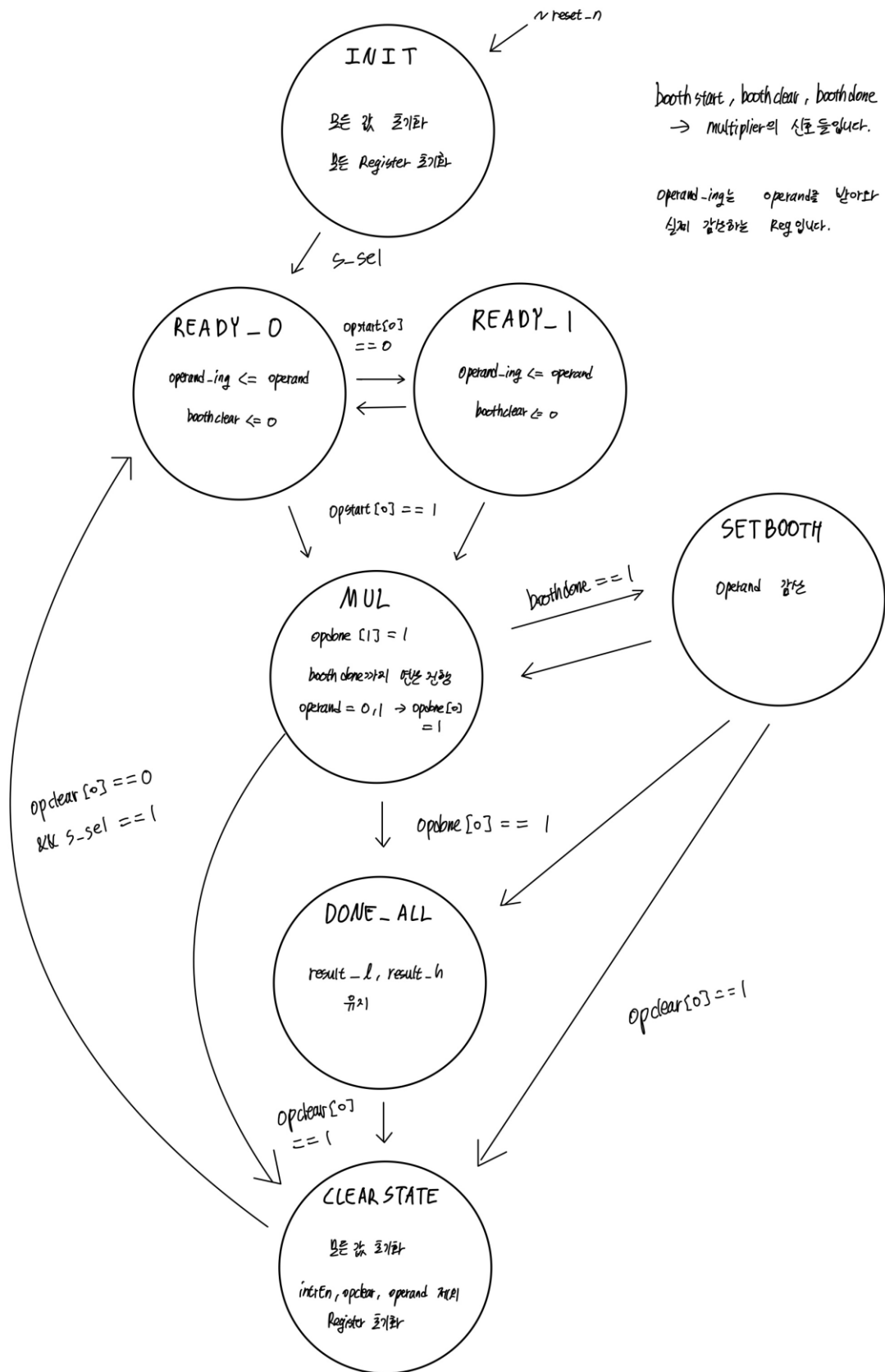


RAM에 경우, cen과 wen에 대해서 Case문으로 동작하기 때문에, Decoder가 사용되었습니다. 각각의 address와 input data가 들어오면, cen/wen에 따라서 Read/Write 동작을 하게 됩니다.

[Factorial Core]



RTL Viewer는 위와 같습니다. 여러 개의 Case문과 조건문, always문으로 인해 상당히 복잡한 것을 확인할 수 있습니다.



Factorial Core의 FSM입니다. INIT 상태는 초기나 reset이 들어오면 모든 값을 0으로 초기화 해주는 상태입니다. 한 사이클 내에 초기화를 시키고, 그 이후에는 READY State로 가서, Factorial 연산 준비과정을 거칩니다. READY State을 두 개로 나눈 이유는, READY State를

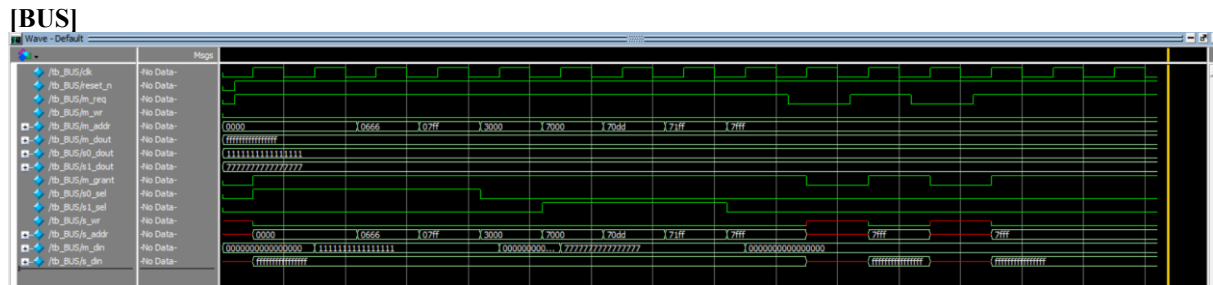
머무는 동안 operand가 바뀌는 것을 반영하여 operand_ing에 넣어주기 위해서 입니다. operand_ing는 operand 신호를 받아와 실제로 감산을 진행하게 되는 reg입니다. boothdone, boothstart, boothclear는 Factorial Core에서 instance한 Multiplier의 opdone, opstart, opclear 신호들입니다. 매번 boothdone이 나오면, SETBOOTH State로 가서 다음 multiplier 실행을 위해 clear 등의 값을 넣어주고, result 값도 update합니다. 또한 가장 중요한 operand_ing 감산도 진행합니다. 그러다가 operand가 1이 되었다면, DONE_ALL State로 넘어가기 위해 opdone[0]에 1을 작성합니다.

operand가 1이나 0인 상황을 처리하기 위해서, flag를 세워서 READY State에서 받아온 operand가 1이나 0일 때, flag를 1로 만들고, MUL 상태로 들어가자마자 예외처리를 하게 됩니다.

DONE_ALL State는 result_h와 result_l 값을 계속해서 유지합니다.

CLEARSTATE는 그동안 쓰인 값들을 초기화해주고, 하지만 INIT 상태와는 다르게 operand와 opclear와 intrEn 값은 초기화해주지 않습니다.

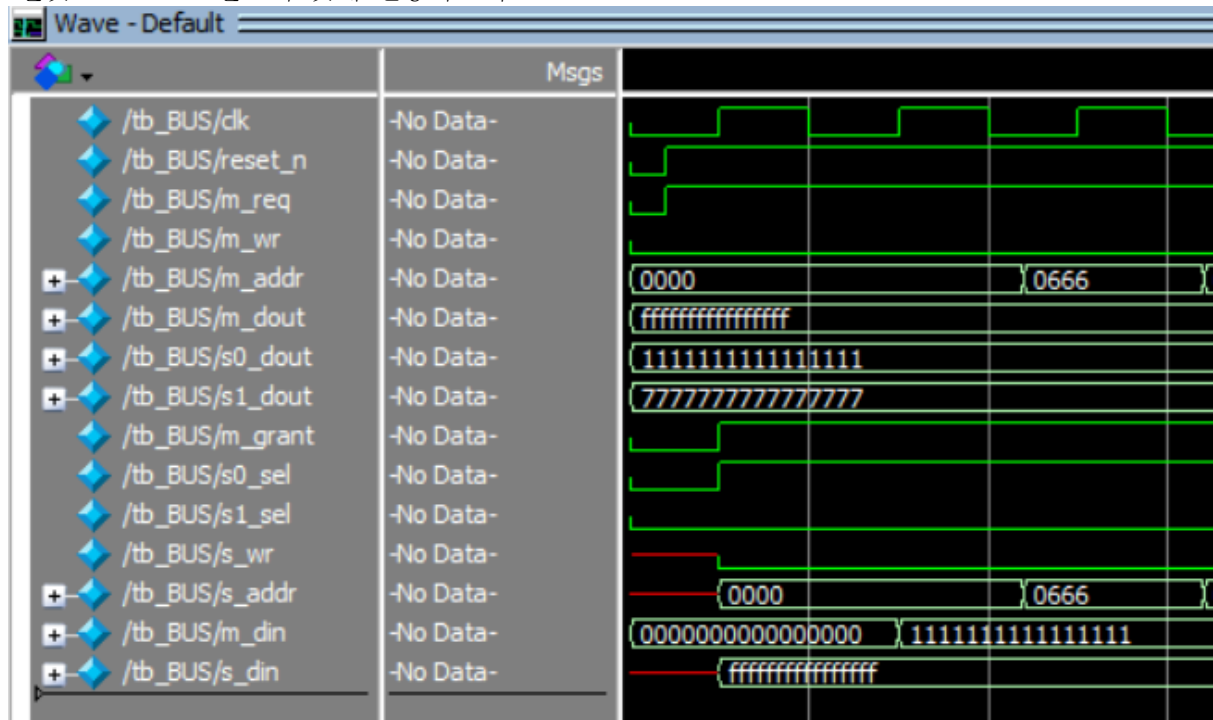
IV. Design Verifiacion Strategy and Results



BUS 모듈의 테스트벤치 Waveform 결과입니다.

검증 전략은 다음과 같습니다.

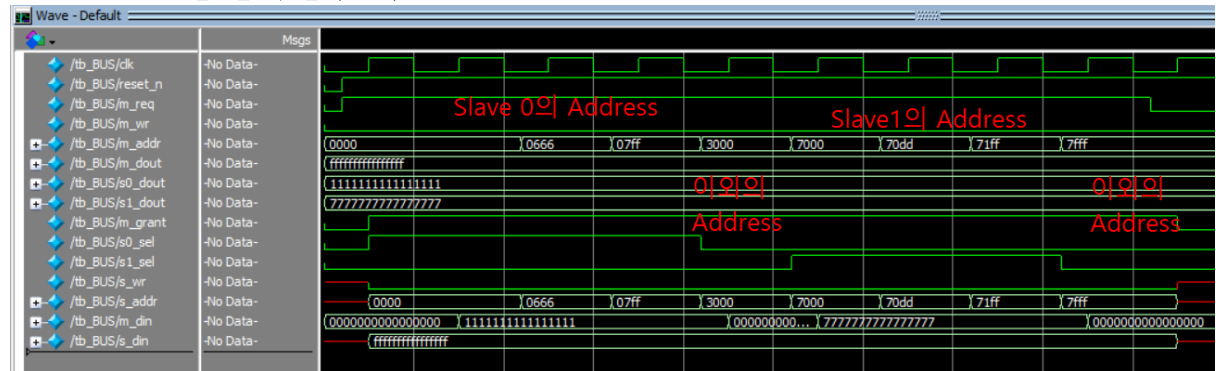
-알맞은 CLOCK 신호에 맞게 반응하는지?



첫 부분을 확인해보면, 처음에 m_req가 들어오고, Clock 신호에 맞춰서 m_grant가 올라가게 됩니다. 그에 맞춰서 s0_sel, s1_sel 신호도 올라가지만, s_dout이 m_din에 전달되는 것은 다음

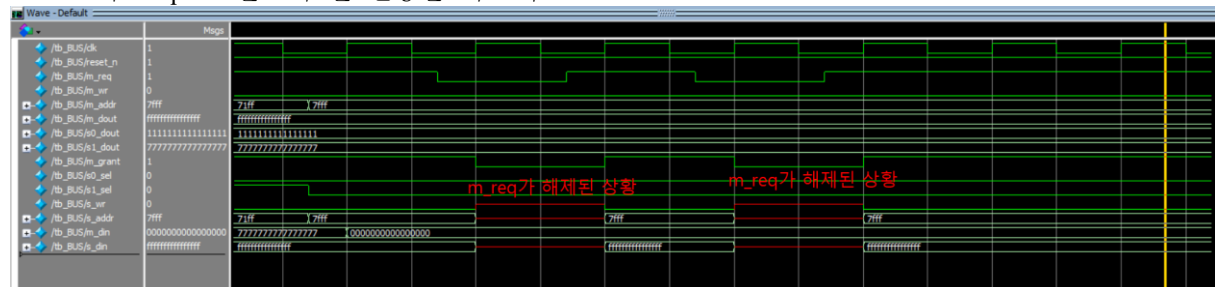
Clock Cycle인 것을 확인할 수 있습니다. MUX3에 들어가는 s 신호가 Clock에 synchronous 하기 때문입니다.

- Address Offset을 잘 구별하는지?



Slave 0 또는 Slave 1에 맞는 Address에 접근할 때에는, 그에 맞는 s_sel 신호들이 적용이 되고 m_din의 값도 알맞게 잘 나오게 됩니다. 하지만 0x3000이나 0x7fff 같은 경우에는 m_din은 MUX3에 맞게 0이 나오는 것을 확인할 수 있습니다. 또한 s_sel 신호들이 모두 0이 됩니다.

-Master의 Request 신호에 잘 반응을 하는지?



의도적으로 m_req의 값을 반복적으로 내렸다가 올려봤습니다. 그 결과 m_grant도 0이 되며, Master가 Slave에 전달하는 값들도 X값으로 끊긴 것을 확인할 수 있습니다.

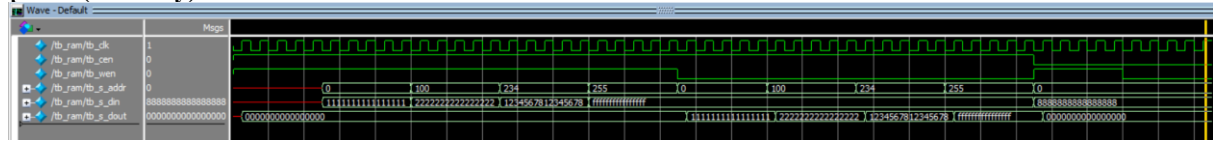
Flow Summary

<<Filter>>

Flow Status	Flow Failed - Sun Dec 03 17:31:28 2023
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	Top
Top-level Entity Name	BUS
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	35 / 41,910 (< 1 %)
Total registers	3
Total pins	360 / 499 (72 %)
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 %)

BUS의 총 Register는 3개, Logic 사용량은 35, Pin은 총 360개가 사용되었습니다.

[RAM(Memory)]



RAM에 대한 테스트벤치 Waveform 결과입니다.

- 정해진 곳에 잘 입력이 되고, 출력이 되는지?

각 주소에 대응하여 넣은 s_din이 잘 입력되어서 s_dout으로 확인이 가능합니다.

- cen/wen 값에 맞게 구동을 하는지?

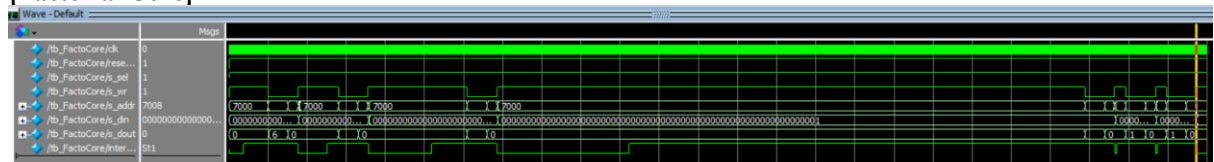
cen = 1/wen = 1인 경우, 원하는 Memory에 Write를 할 수 있습니다. 그 후, cen = 1/wen = 0인 상태에서는 Memory를 Read할 수 있기 때문에 확인해보면 앞서 입력한대로 잘 Memory에 쓰였고 그 내용을 읽을 수 있습니다.

cen = 0이 되면 아무리 다른 값을 조작하더라도 아무런 결과를 확인할 수 없습니다.

Flow Status	Successful - Sun Dec 03 19:12:11 2023
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	Top
Top-level Entity Name	ram
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	7,838 / 41,910 (19 %)
Total registers	16448
Total pins	139 / 499 (28 %)
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 %)

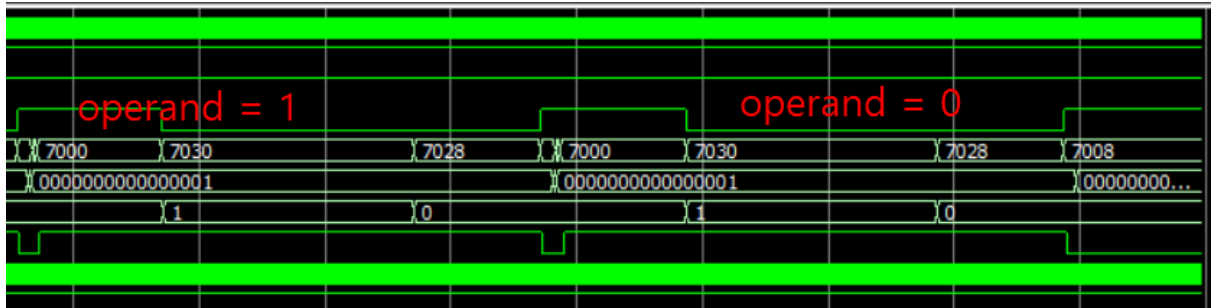
Logic 사용량은 78838, Register 사용량은 16448, 총 Pins는 139개가 쓰였습니다.

[Factorial Core]



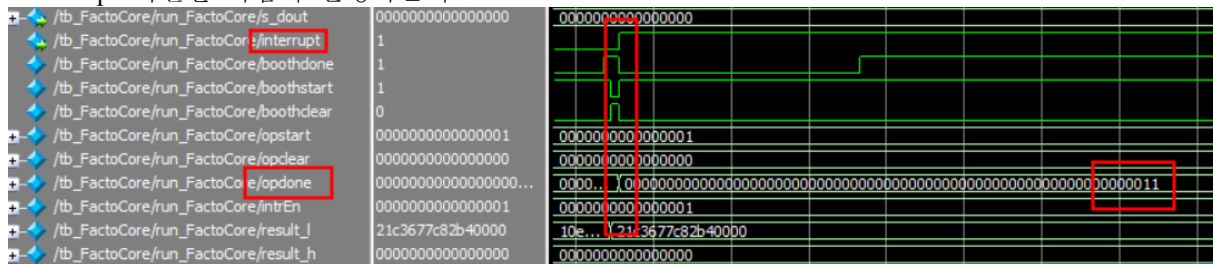
Factorial Core의 Testbench Waveform 결과입니다. 검증 전략은 다음과 같습니다.

- operand = 0이나 operand = 1일 때 잘 처리가 되는지?



TestCase 중 operand가 0이나 1인지를 확인했습니다. 그 결과 result_1 = 1, result_h = 0으로 잘 나오는 것을 확인할 수 있습니다.

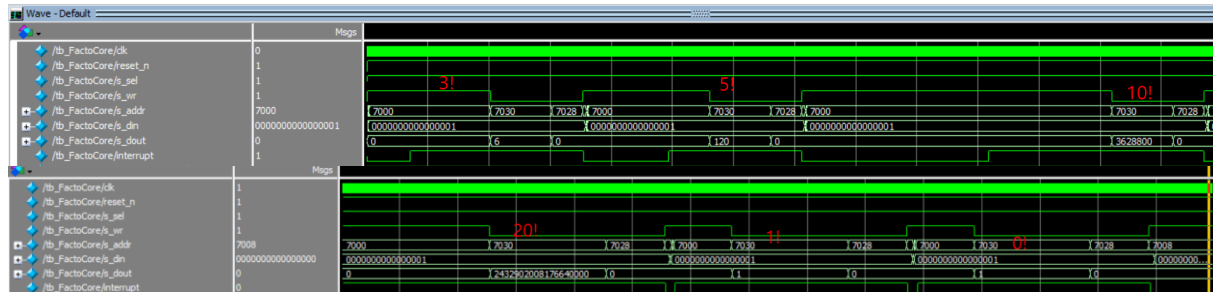
- Interrupt 적절한 시점에 발생하는지?



opdone[1:0] = 2'b11이 되자마자, 알맞은 타이밍에 interrupt 신호가 올라간 것을 확인할 수 있습니다.

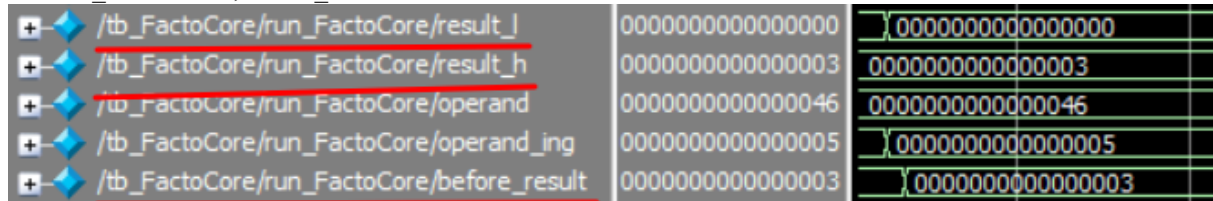
이는 opdone[0]과 intrEn[0]이 AND 게이트로 즉시 출력할 수 있도록 되어있기 때문입니다.

- 값이 맞는지?



이로써 알맞은 값이 출력되고 있음을 확인할 수 있습니다.

- Result_1이 0일 때, Result_h를 사용하는지?



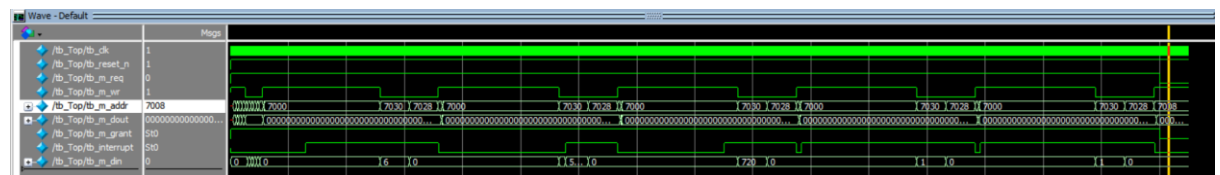
이러한 케이스를 테스트해보기 위해서, 위에 전체 테스트 Waveform에는 안 나와있는 추가 테스트를 진행하였습니다. Operand = 70입니다. 70을 계산하다보면, result_1이 0이 되는 시점이 나오는데, 그 상황에서 Factorial Core에서 Multiplicand로 사용되는 before_result라는 변수에 result_h가 들어감으로써 예외처리가 잘 되는 것을 확인할 수 있습니다.

Flow Summary	
<<Filter>>	
Flow Status	Successful - Wed Dec 06 21:27:59 2023
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	Top
Top-level Entity Name	FactoCore
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	398 / 41,910 (< 1 %)
Total registers	107
Total pins	149 / 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 %)

총 Logic 사용량은 398, Total Register 개수는 107, 총 Pin의 개수는 149개입니다.

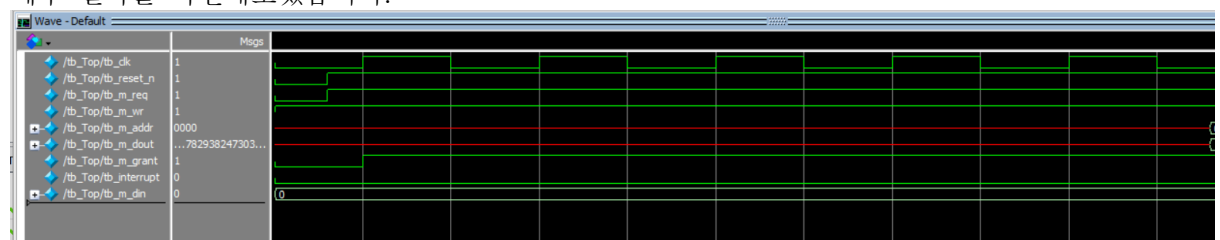
[Top]

지금까지 앞서 밝힌 검증 방법들을 한번에 테스트한 Top 모듈의 테스트벤치입니다.



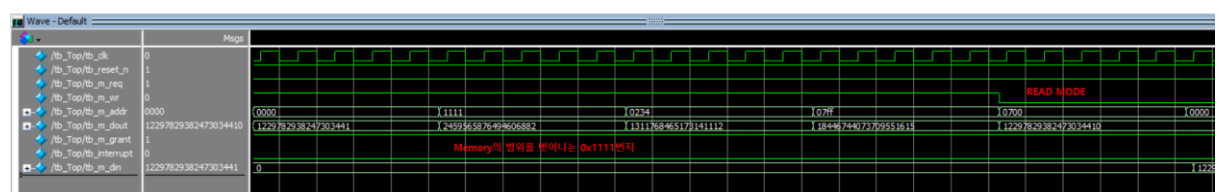
Top 모듈의 전체 Waveform은 위와 같습니다.

세부 결과를 확인해보겠습니다.



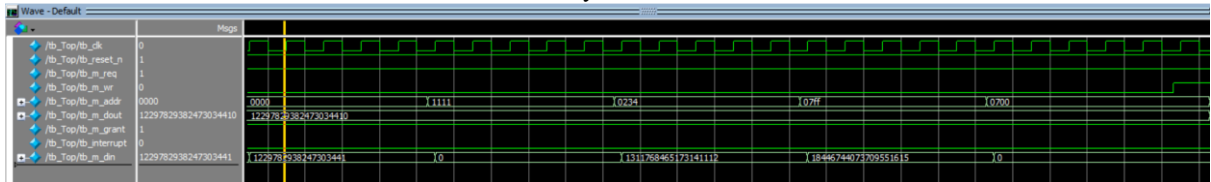
m_req 신호가 올라가면, m_grant는 그 다음 positive edge of Clock 시점에 생기게 됩니다.

아직은 m_addr나 m_dout에 대한 입력을 주지 않아 X인 상태입니다. 연산도 진행/종료되지 않아 interrupt도 0입니다.

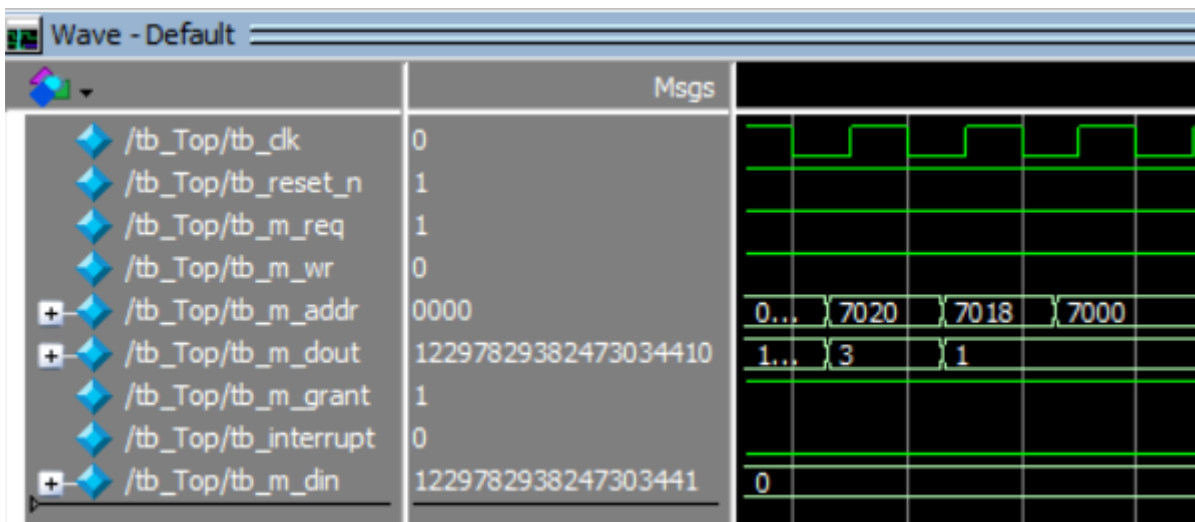


m_wr = 1 이므로 Write Mode입니다.

이론대로라면, 0x1111은 Memory나 Factorial Core의 주소도 아니기 때문에 그 어느 곳에도 쓰이면 안 됩니다. 또한 0x7000번지에 경우, RAM의 Address 범위 내에는 들어오지만, m_wr = 0으로 내렸기 때문에 READ MODE임으로 Memory에 값이 쓰이면 안 됩니다.

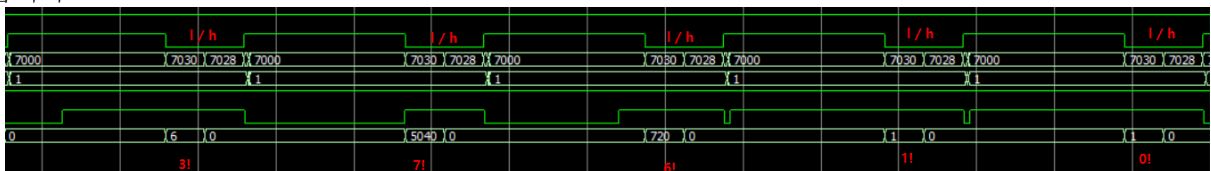


다른 값은 원래 의도한대로 잘 입력이 되어서, 출력으로 잘 나오게 됩니다. 또한 0x1111과 0x7000은 입력 자체가 되지 않아, 초기값인 0으로 나오는 것을 확인할 수 있습니다.

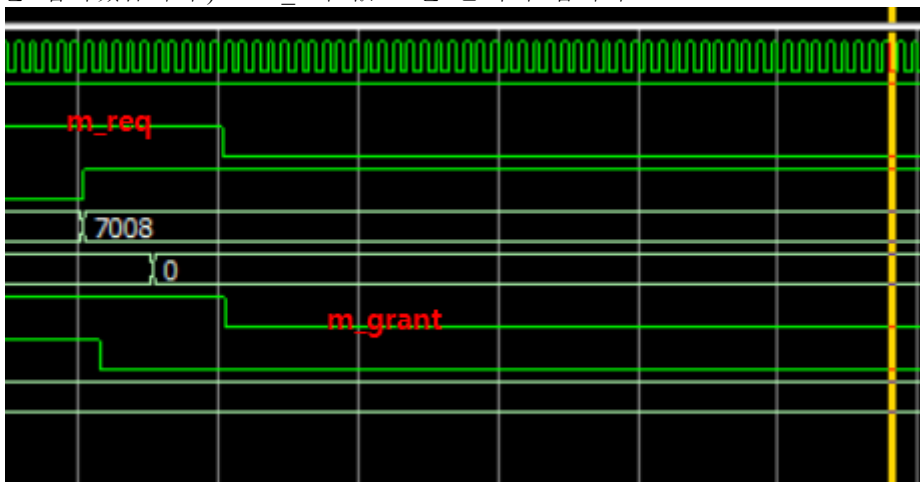


이후로는 Factorial Core에 접근하여 연산 테스트를 하였습니다.


BUS에 문제가 없다면, operand나 opstart에 대한 값에 대하여 Factorial Core에 잘 전달을 해야 합니다.



매번 어느 상황에서도 값을 잘 전달하고, 연산이 종료되면 interrupt 신호도 발생하여(intrEn에 1을 입력했습니다.) result h의 값도 잘 출력이 됩니다.



m_req에 대해서도 BUS를 통해 m_grant 값을 잘 내보내는 것을 확인할 수 있습니다.

Flow Summary	
 <<Filter>>	
Flow Status	Successful - Wed Dec 06 21:36:17 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)	8,148 / 41,910 (19 %)
Total registers	16558
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 %)

BUS, Factorial Core, RAM을 합친 Top 모듈의 총 Register 개수는 16558개입니다. Logic 사용량은 8,148이며, Pin은 150개가 사용되었습니다.

V. Conclusion

이번 프로젝트에서는 Factorial Core를 설계하는 데 있어서 상당히 많은 어려움이 있었습니다. State를 처음에는 적게 했는데, 그렇게 하다 보니 operand 감소에 있어서 한 번에 여러 번 줄어 들고, 이러한 결과가 있었습니다. 그래서 always 문 안에 state만 존재하게 한 후, state에서 한 번씩 operand를 감소하게 함으로써 원하는 결과를 얻을 수 있었습니다.

또한 이번에는 제 나름대로 고민을 많이 하다 보니, 엉뚱한 실수를 하게 되었습니다. 저는 Master가 한 Clock Cycle 내에 같은 Register에 input이 서로 다른 것이 들어오더라도, 즉 언제든지 자신의 input을 Update를 하는 것이 맞다고 생각하여 그렇게 만들었습니다. 하지만 Test를 위해 조교님께서 제공해주신 Testbench에서 BUS에 대한 오류가 나타나서 Test를 진행할 수 없다는 오류문이 나오게 되었습니다.

그래서 다시 프로젝트 Spec을 읽으며, 제가 어쩌면 과하거나 오히려 잘못된 기능을 넣었구나 라는 생각이 들어, input logic이나 output logic을 Clock에 의존하여 나오게 만들었습니다. 그랬더니 테스트가 진행되었고, 부끄럽지만 20개 중 8개가 통과하는 결과를 얻게 되었습니다.

그래서 결론적으로 정리해보자면, always의 Sensitivity List에 들어가는 요소가 값이 변화한다면 그 always문 안에 들어가게 되기 때문에, 조건이나 wire라고 무조건 넣으면 안 된다는

것입니다. 또한, 제가 이번 프로젝트를 겪으면서 느낀 것은 웬만하면 Clock에 동기화해서 always문을 실행되게 하면, 외부 값들의 변화에 영향을 받지 않고 오히려 더 차근차근 진행하게 되어 더 안전한 결과를 얻을 수 있는 것 같다고 생각합니다.

VI. Reference

- [1] 디지털논리회로2 강의자료 (Simple Bus, Memory, Booth Multiplier)
- [2] 컴퓨터공학기초실험2 실습자료 (Week 12, Week 10)

감사합니다.