**TERM PROJECT**

**FACTORIAL & FIFO data transfer**

최동환

**Abstract**

이번 설계과제는 Multiplier와 Factorial 그리고 FIFOTOP을 설계하여 BUS를 통해 각각의 component를 제어하고 결과값을 검증을 받는 project이다. 각각 component가 다른 기능을 하고 있기 때문에 올바른 기능을 하도록 구현을 하고 TOP module에 연결 했을 때 사용자에 의해 BUS를 통한 작동이 가능 하여야 한다. 수많은 신호들을 통제하기 힘든 부분을 인터페이스가 간단한 BUS를 통해 하드웨어 구현하는 것이 중요한 핵심이다. 각각의 component구현을 하나의 하드웨어로 연결함으로써 하나의 system을 확인 할 수 있다. 이어서 진행 될 Introduction에서는 해당 프로젝트의 전체적인 묘사를 구성 하였으며, 2.specification에서는 각 component에 관한 간단한 설명을 하였다. 3. Design detail에 내가 구현한 코드가 어떻게 구성 되어있는지 자세하게 설명하였으며, 4. 검증에선 각 component별로 wave form을 통해 자세한 검증을 시행하였다.

|  |
| --- |
| **Ⅰ. Introduction** |
| 해당 프로젝트는 Factorial연산을 하는 장치를 하드웨어적으로 구현하는 것에 목적을 둔다. 이 장치에서 전체적인 system을 관리하는 프로세서는 test bench module이다. 전체적인 순서를 설명 하자면, test bench에서 bus grant를 받아서 FIFOTOP\_IN에 factorial연산을 할 값들을 넣어준다. 총 8개까지 가능하다. 그리고 test bench는 bus grant를 놓고, factorial에게 연산을 시작하는 op\_start 신호를 보낸다. 그러면 factorial이 bus grant를 요청한 후 bus grant를 받는다. 그리고 나서 factorial은 FIFO\_IN에게 data를 요청한다. 이 부분에서 만약 FIFO\_IN이 비었다면 연산할 데이터가 없는 것이기 때문에 bus grant를 놓고 interrupt 신호를 발생 시킨다. FIFO\_IN에서 data를 읽어왔다면 bus grant를 놓은 후 그 값을 MULTIPLIER연산의 반복을 통해 factorial값을 계산 한다. 연산이 완료된 후에 128-bit 결과 데이터를 register에 4부분으로 나누어 저장한다. 그 후 다시 factorial은 bus grant를 요청하고 4부분으로 나눠진 결과가 저장된 register를 FIFO\_OUT에 저장한다. FIFO\_IN에서 8개까지 저장이 가능 했으므로 FIFO\_OUT에는 32개까지 data가 저장 가능하다. 그 후에 다시 FIFO\_IN에 data를 요청하는 부분으로 돌아가서 위의 과정을 반복한다. 위의 설명에 FIFO\_IN이 비었다면 bus grant를 놓고 interrupt 신호를 발생 시키는 부분으로 돌아가면, interrupt가 발생 된 후에 FACTORIAL은 interrupt가 다시 0이 되기를 기다린다. interrupt가 0이 되면 다시 연산을 할 수 있는 초기 상태로 돌아가게 된다. 이것이 이 장치의 전체적인 한 cycle이다.  아래 그림은 전체적인 데이터의 흐름을 나타낸다. grant를 받고 놓는 혹은 INTERRUPT 등 세부 과정은 생략하여 그렸다.    1. data ( tb🡪BUS)  2. data(BUS 🡪 FIFO\_IN)  3. op\_start signal(tb 🡪BUS 🡪FACTORIAL)  4. data(FIFO\_IN 🡪BUS)  5. data(BUS 🡪FACTORIAL)  6. data(FACTORIAL 🡪MULTIPLIER)  7. data(EXECUTE)  8. data(MULTIPLIER 🡪FACTORIAL)  9. data(FACTORIAL 🡪BUS)  10. data(BUS 🡪FIFO\_OUT)  11. data(FIFO\_OUT🡪tb) |

|  |
| --- |
| **Ⅱ. Project Specification** |
| **1. FIFOTOP\_IN & FIFOTOP\_OUT** |
| **1) FIFO\_IN & FIFO\_OUT**  Fifo\_top에서 데이터를 저장하고 있는 기능을 하는 하위 module이다. 상태에 따라 output이 결정되고 다음상태가 결정되는 FSM이며, 총 5개의 상태를 갖는다.  **●5개의 상태**  **IDLE** : 맨 처음의 상태이며 아무 동작도 안 하는 상태이다. 따라서 어떤 한 상태에서 No operation 신호가 들어올 경우 IDLE상태로 이동하게 된다. 이 상태에서는 들어오는 input값에 따라 다음 상태가 결정 된다.  **WRITE** : FIFO에 데이터를 저장하는 상태이다. 들어오는 input에 따라 어느 상태로도 이동할 수 있지만, Data\_count가 이미 꽉 찼을 경우에 쓰기 명령이 들어올 경우 WR\_ERROR 상태로 이동하게 된다.  **READ** : FIFO에서 데이터를 읽어오는 상태이다. 들어오는 input에 따라 어느 상태로도 이동 할 수 있지만, Data\_count가 0일 경우에는 Read명령이 들어올 경우 RD\_ERROR 상태로 이동 하게 된다.  **RD\_ERROR** : 이미 empty인데 또 다시 read명령이 들어왔을 때 set되는 상태이다. WRITE 상태나 IDLE상태로 갈 수밖에 없으며, 또 다시 read명령이 들어올 경우 상태는 제자리에 머물게 된다.  **WR\_ERROR** : 이미 full인데 또 다시 write 명령이 들어왔을 때 set되는 상태이다. READ상태나 IDLE상태로 갈 수 밖에 없으며, 또 다시 write명령이 들어올 경우 상태는 제자리에 머물게 된다.  **●각 상태에서의 출력**   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  | | FULL | | | EMPTY | | | Data\_count==0 | | 0 | | | 1 | | | Data\_count==Max\_size | | 1 | | | 0 | | | Else case | | 0 | | | 0 | | |  | IDLE | | WRITE | READ | WR\_ERROR | RD\_ERROR | | WR\_ACK | 0 | | 1 | 0 | 0 | 0 | | WR\_ERR | 0 | | 0 | 0 | 1 | 0 | | RD\_ACK | 0 | | 0 | 1 | 0 | 0 | | RD\_ERR | 0 | | 0 | 0 | 0 | 1 |   FIFO\_IN과 FIFO\_OUT의 차이점 : FIFO\_IN은 32-bit데이터를 8개 까지 저장할 수 있고, FIFO\_OUT은 32-bit 데이터를 32개 까지 저장 할 수 있다.  **2) FIFOTOP\_IN & FIFOTOP\_OUT**  FIFOIN\_TOP 혹은 FIFOOUT\_TOP은 module안에 data를 쓰거나 module로부터 data를 읽을 수 있는 장치이다. 해당 프로젝트에서는 SLAVE로서만 작동을 하기 때문에, select 신호가 1일 때만 데이터를 읽고 쓸 수 있다. FIFO\_TOP안에는 FIFO의 flag값을 저장하고 있는 FIFOFLAG register가 있는데, 이 것은 0x0의 offset을 갖는다. FIFO가 출력하는 flag의 값들이 하위 6-bit에 저장 되어 있다. 그리고 내부의 FIFO의 offset은 0x1이다. FIFO\_TOP의 select 신호가 1이고, offset이 0x0일 때, wr이 0이라면 FIFO\_FLAG가 선택된 것이므로 FIFO\_FLAG 값이 output으로 출력되어야 한다. 이를 제외한 나머지 경우에는 외부에서 값이 쓰여져도 안되며, 읽을 수 없다. FIFO\_FLAG의 값은 오직 FIFO에서 출력된 output flag들만이 저장 되어야 한다. 그리고 selece신호가 1이고, offset이 0x1일 때는 FIFO\_TOP의 FIFO가 선택된 것이다. wr이 0이라면 FIFO에서 dequeue된 data가 output으로 출력 되어야 하며, wr이 1이라면 FIFO로 data가 write되어야 한다. 이 외의 경우에는 FIFO에 data가 쓰여지거나 FIFO로부터 데이터가 읽어 질 수 없다. 그리고 processor인 test bench에서 확인 할 수 있는 output인 fifo count와 fifo flag가 있는데, 이 output들은 현재 FIFO의 상태를 알기 위한 신호로서 해당 FIFO\_TOP의 해당 FIFO가 선택 되었을 때만 출력 되어야 하는 output들이다. |
| **2. BUS** |
| Bus는 compnent들 간에 data를 전송 할 수 있도록 연결해주는 장치이다. 컴퓨터 내부에 있는 하드웨어간에도 데이터 교환이 필요하기 때문에, BUS는 필수적인 장치라고 할 수 있다. 그중에서도 정보를 쓰기 or 읽기 를 요청할 수 있는 장치를 master라고하고, master에게 요청을 받는 장치는 slave라고 한다. 이번 project에서 사용할 bus는 32bit 데이터를 전송 할 수 있는 것이며, master는 2개까지 slave는 3개 까지 가능한 것이다. ‘Address의 bandwitdth는 8bits이며, 총 2^8 - 256개의 address를 선택 할 수 있으며, 그 중 상위 4-bit은 component를 선택 해주는 base address이며, 하위 4-bit은 component안의 register 혹은 구성요소등을 선택 할 수 있는 offset address이다. 따라서 2^4 - 16개의 component를 선택 할 수 있고, component안에서 또한 16개의 offset address로 register를 선택 할 수 있다. [1]’ BUS의 Grant는 arbitor에 의해 결정 되는데, 2개의 master가 보내는 request신호에 따라 어떤 master에게 Grant를 줄지가 결정 된다. Grant가 부여된 master가 주는 address에 따라 어떤 slave를 선택 할지를 결정 할 수 있으며, write/read 신호에 따라 write or read 기능을 할지 정한다. write기능을 요청 한 것이라면, master data out이 slave data in을 통해 slave로 전달되며, read기능을 요청 한 것 이라면 slave data out이 master data in을 통해 master에게 data가 전달된다. |
| **3. MULTIPLIER** |
| 1)Booth Multiplication algorithm – radix4   |  |  |  |  |  | | --- | --- | --- | --- | --- | | x(i) | x(i-1) | x(i-2) | operation | y(i) | | 0 | 0 | 0 | only 2-bit Arithmetic Shift Right the result | 0 | | 0 | 0 | 1 | Add multiplicand to result and 2-bit ASR | +1 | | 0 | 1 | 0 | Add multiplicand to result and 2-bit ASR | +1 | | 0 | 1 | 1 | Add 2\*multiplicand to result and 2-bit ASR | +2 | | 1 | 0 | 0 | Sub 2\*multiplicand to result and 2-bit ASR | -2 | | 1 | 0 | 1 | Sub multiplicand to result and 2-bit ASR | -1 | | 1 | 1 | 0 | Sub multiplicand to result and 2-bit ASR | -1 | | 1 | 1 | 1 | only 2-bit ASR | 0 |   [2]  multiplier의 최하위 bit에서부터 3-bit씩 조사해서 해당 표에 따라 result에 어떤 연산을 해줄지 결정한다.  제일 처음 multiplier의 하위 bit을 조사 할 때에는 multiplier의 2-bit + 0 으로 조사를 해준다.  2) 추가한 algorithm – only shfit 조사.  조사한 bit을 제외하고 multiplier의 나머지 bit이 모두 0 혹은 1일 때는 result에 따로 연산을 해줄 필요없이 오직 shfit만 해주면 된다. 나머지 bit수 / 2를 해준만큼 result를 shift해준 다면 빠른 multiple 연산이 가능하다.  3) MULTIPLIER의 작동원리  해당 module은 input값으로 multiplier와 multiplicand를 받아서 연산을 통해 곱셈의 결과를 output으로 내보내는 장치이다. 이의 구현을 위해 booth multiplication algorithm을 사용 하였다. 상태가 3가지로 나뉘어 작동하며, 각각 IDLE, EXECUTE, DONE 이다.IDLE상태일 땐 연산을 준비한다. EXECUTE상태일땐 곱셈연산을 수행한다. DONE상태일 땐 결과 값을 지니고 있으며, clear 신호가 나오면 IDLE갈 수 있게 준비한다. EXECUTE상태일 때는 한 cycle마다 count값이 1씩 증가하며, 만약 모든 bit이 0일 때는 cycle을 줄여서 한번에 count값이 끝까지 올라 갈 수 있게 작동한다. |
| **4.FACTORIAL** |
| 이번 프로젝트의 핵심 module이다. 이 장치는 데이터를 받아서 factorial연산을 해주는 기능을 한다. factorial이란 n! = n x n-1 x n-2 …… x 2 x 1 를 의미하며 이 연산을 하기 위해서 내부에 MULTIPLIER module이 들어있다. FACTORIAL은 master와 slave로 모두 사용 될 수 있으며, SLAVE로 선택 되었을 때는 test bench에 의해 FACTORIAL에 데이터가 써질 수 도 있고, 데이터가 읽어질 수도 있다. 반대로 FACTORIAL은 master grant를 요청 할 수도 있으며, master grant를 받았을 때, FIFOTOP을 slave로 선택하여 데이터를 읽어 올 수 도 있고, FIFOTOP에 데이터를 쓸 수도 있다. 전체적인 연산 과정은 다음과 같다.  1. testbench로부터 operation start 명령을 받는다. 이 때에는 Factorial이 slave로서 선택 된다.  2. BUS grant를 받기 위해 request 신호를 보낸다.  3. BUS grant를 받은 후, FIFOTOP\_IN을 slave로 선택 한 후에 FLAG값을 읽어온다.  4. FLAG값을 조사해서 EMPTY상태일 경우에는 14.로 이동한다.  5. FIFOTOP\_IN에게 N\_value(factorial연산할 대상)을 요청한다.  6. FIFOTOP\_IN에게 N\_value를 받은 후에 N\_value를 조사한다. BUS grant를 놓는다.  7.N\_value가 0, 1, 2이면 MULTIPLIER를 사용하여 연산할 필요 없이 바로 10번으로 간다.  8. N! 이므로 처음에 N과 N-1의 곱을 MULTIPLIER를 통해 계산한다. 이 단계에서 op\_done이 1로 set되면 연산이 끝난 것이므로, 다음 state로 넘어간다.  9. 계산된 N x N-1 을 result에 저장하고, N이 2인지 조사해준다. N이 2일 경우 다음 단계로 넘어간다. 2가 아닐 경우엔 남은 factorial연산을 해주어야 한다. N을 -1해주고, 다음 곱셈연산을 할 multiplicand는 result로 multiplier는 N-1로 설정 해준다. 그리고 8번으로 돌아가 이 과정을 반복한다.  10. factorial연산이 완료 되었으므로, FACTORIAL내부 register에 결과값을 써주어야 한다. result는 128-bit 이고, 내부 register는 32-bit이기 때문에 총 4개의 register에 나눠서 저장한다.  11. 다시 BUS grant를 요청 한다.  12. BUS grant를 받은 후에 FIFO\_TOP에 결과가 저장 된 총 4개의 register를 모두 써준다.  13. 위의 3번으로 돌아간다.  14. FIFOTOP\_IN이 empty인 경우엔 연산할 대상이 없는 것이므로, bus grant를 놓는다.  15. 그리고 Interrupt register를 1로 만들어서 interrupt신호를 발생 시킨다.  16. test bench는 이것을 확인하고 Factorial을 slave로서 선택해서 그 interrupt를 없애 준다. 그럼 FACTORIAL은 초기화과정을 거쳐서 처음 상태인 IDLE로 돌아가게 된다. |
| **5. TOP** |
| FACTORIAL, BUS, FIFOTOP\_IN, FIFOTOP\_OUT을 모두 합쳐놓은 module로서 전체적인 기능을 수행 하는 module이며, 이것의 input은 사용자가 조정 할 수 있는 master data들과 clk reset\_n이다. 따라서 TOP의 test bench를 통하여 clock을 만들어 내고, reset신호도 줄 수 있다. 그리고 사용자가 직접 master0의 data를 조정해서 전체적인 데이터의 흐름을 만들어 낼 수 있다. |

각 component 별 작동원리 기재 (ex) FIFO 작동원리, Multiplier 알고리즘 등…

|  |
| --- |
| **Ⅲ. Design Details** |
| **1. FIFO\_IN & FIFO\_OUT** |
| 1) Pin description   |  |  |  | | --- | --- | --- | | **Input** | clk | clock | | reset\_n | reset signal | | wr\_en | write enable | | rd\_en | read enable | | din[31:0] | 31-bit input data | | **Output** | dout[31:0] | 31-bit output data | | data\_count[3:0] or data\_count[5:0] | data count [3:0] 🡪 FIFO\_IN, [5:0] 🡪 FIFO\_OUT | | full | flag this mean full-data | | empty | flag this means empty-data | | wr\_ack | flag this means write acknowledge | | wr\_err | flag this means write error | | rd\_ack | flag this means read acknowledge | | rd\_err | flag this means read error |   2) Block diagram    위 그림은 FIFO\_IN& FIFO\_OUT의 schemetic symbol을 나타낸다. 왼쪽은 input data들을 나타내고 오른쪽은 output data들을 나타낸다. rd\_en, wr\_en에 따라 read 또는 write 기능을 수행하며, write기능을 할 때 din에 32-bit data가 들어가고 read 기능을 할 때 dout으로 32-bit data가 출력된다. data\_cnt는 현재 FIFO에 저장된 data의 개수를 의미하며, full, empty, wr\_ack, wr\_err, rd\_ack, rd\_err 은 현재 FIFO의 상태를 나타내는 flag들이다.  3) FSM    **4) state description**   |  |  | | --- | --- | | state | Description | | IDLE | 초기상태 or wr\_en, rd\_en 이 둘 다 0일 때 아무것도 안 하는 상태를 나타낸다 | | **WRITE** | FIFO 에 data를 write하는 상태이다. | | **READ** | FIFO에서 data를 read하는 상태이다. | | **RD\_ERROR** | FIFO에서 data를 read했지만 저장 된 것이 없어서 실패한 상태이다. | | **WR\_ERROR** | FIFO에 data를 write했지만 가득 차서 write할 수 없는 상태이다. |   RD\_ERROR과 WR\_ERROR을 제외한 상태에서는 위의 5개 condition에 따라 모든 상태로 이동 할 수 있으며, RD\_ERROR에서는 rd\_con을 제외한 모든 경우에서 다른 state로 이동이 가능하다. 반대로 WR\_ERROR상태에서는 wr\_con을 제외한 모든 경우에서 다른 state로 이동이 가능하다.  5) register description   |  |  | | --- | --- | | **Name** | **Description** | | head[2:0] or [4:0] | Read할 address가 저장 된 값 | | tail[2:0] or [4:0] | write 할 address가 저장 된 값 | | data\_count[3:0] or [5:0] | 현재 data 개수가 저장 된 값 | | \*next logic of head, tail, data\_count  state따라 next logic이 결정 된다.  IDLE : write read 둘 다 안 하므로 next\_head, next\_tail, next\_data\_count모두 이전 값을 갖는다.  WRITE : write를 하는 상태이므로 next\_head는 이전 값을 갖고 , next\_tail은 tail+1 , next\_data\_count는 data\_count+1 값을 갖는다.  READ : read를 하는 상태이므로 next\_head는 head+1의 값을 갖고 next\_tail은 이전 값을 갖고, next\_data\_count는 data\_count-1값을 갖는다.  WR\_ERROR & RD\_ERROR : 두 상태에서 모두 write read 모두 하지 않기 때문에 next\_head, next\_tail, next\_data\_count는 모두 이전 값을 갖는다. | | |
| **2. FIFOTOP\_IN & FIFOTOP\_OUT** |
| 1) pin description   |  |  |  | | --- | --- | --- | | **input** | clk | clock | | reset\_n | ~(reset) | | sel | select signal (1이면 작동, 0이면 작동안함) | | wr | wr이 1이면 write, 0이면 read | | address[7:0] | 선택될 fifo들의 주소값을 나타낼수 있는 address | | din[31:0] | input data | | **output** | dout[31:0] | output data | | fifo\_cnt[3:0]  or fifo\_cnt[5:0] | fifo의 데이터 수( [3:0] 🡪 FIFOTOP\_IN , [5:0] 🡪FIFOTOP\_OUT) | | fifo\_flag[5:0] | 현재 fifo의 상태  fifo\_flag[5] = full , fifo\_flag[4]=empty, fifo\_flag[3]=wr\_ack  fifo\_flag[2] = wr\_err, fifo\_flag[1] = rd\_ack, fifo\_flag[0]=rd\_err |   2)Block diagram    위의 그림은 FIFOTOP의 schematic symbol을 나타낸다. address에 따라 FIFO를 선택 할 수도 있고, FIFOFLAG register를 선택 할 수 있다. FIFOTOP이 선택된 상태에서 wr값에 따라 FIFOTOP에 값을 쓰거나 읽을 수 있다. output fifo\_flag는 내부 FIFO의 flag이며, fifo\_cnt는 내부 FIFO의 저장된 데이터 개수다.  3) register description   |  |  |  |  | | --- | --- | --- | --- | | **offset** | **Type** | **Name** | **Description** | | 0x0 | READ | FIFOFLAG | 내부에 있는 FIFO의 flag가 하위 6-bit에 저장된다.  나머지 [31:6]bits은 reserved value이다. | | \*next logic of FIFOFLAG  next\_FIFOFLAG는 단지 내부 FIFO의 output fifo\_flag가 저장된다. | | | |   4) output description   |  | | --- | | * **output logic of dout** | | FIFOTOP이 slave로서 선택 되었을 때 next\_sel, next\_address, next\_wr에 의해 결정 된다. 왜 현재 sel이 아닌 next sel인 이유는 해당 sel, address, wr이 입력 되고 나서 FIFO에 의해 한 cycle뒤에 그에 상응하는 결과가 나오기 때문이다.  next\_address가 0x0, next\_sel이 1, next\_wr이 0일 때만 dout은 FIFOFLAG가 된다.  next\_address가 0x1, next\_sel이 1 일 때 dout은 fifo\_dout이 된다.  그 외의 경우에 dout은 기본 값인 0이 출력 되어야 한다. | | * **output logic of fifo\_cnt, fifo\_flag** | | FIFOTOP의 output이다. next\_sel이 1, next\_address가 0x1 즉 fifo가 선택 되었을 때만 fifo\_flag, fifo\_cnt는 fifo의 output은 w\_fifo\_flag, w\_fifo\_cnt가 출력 되어야 한다. 그 외의 경우에는 기본값인 0이 출력 되어야 한다. | |
| **3. BUS** |
| 1) pin description   |  |  |  | | --- | --- | --- | | **input** | clk | clock signal | | reset\_n | reset signal | | m0\_req | master0’s request | | m0\_wr | master0’s wirte or read signal | | m\_address[7:0] | master0’s address | | m0\_dout[31:0] | master0’s output data | | m1\_req | master1’s request | | m1\_wr | master1’s write or read signal | | m1\_address | master1’s address | | m1\_dout | master1’s output data | | s0\_dout | slave0’s output data | | s1\_dout | slave1’s output data | | s2\_dout | slave2’s output data | | **output** | m0\_grant | master0’s grant | | m1\_grant | master1’s grant | | m\_din[31:0] | master’s input data | | s0\_sel | slave0 select | | s1\_sel | slave1 select | | s2\_sel | slave2 select | | s\_address | slave address | | s\_wr | slave write or read | | s\_din[31:0] | slave input data | | **reg** | next\_s0\_sel  next\_s1\_sel  next\_s2\_sel | next s0 sel  next s1 sel  next s2 sel |   ‘위의 표를 확인해보면 input port에 m0\_dout, s0\_dout 등, output port에 보면 m\_din, s\_din 이런 식으로 거꾸로 되어 있는 것을 확인 할 수 있는데, 이는 BUS가 data의 전달 역할을 하기 때문이다. master 또는 slave의 data\_out이 bus에게는 input data이고, master 또는 slave의 input data는 bus에게는 output data이다.  2)block diagram    BUS의 schematic symbol이다. 안으로 들어가는 화살표는 input을 나타내고 밖으로 나오는 화살표는 output을 나타낸다. M자로 붙어있는 data는 master interface이고 , S자로 붙어있는 data는 slave interface를 나타낸다.  memory map은 다음과 같다.   |  |  | | --- | --- | | MEMORY MAP | | | slave0 | 0x00 ~ 0x0F | | slave1 | 0x10 ~ 0x1F | | slave2 | 0x20 ~ 0x2F |   3)FSM  BUS module안의 arbiter의 state의해 grant가 결정 된다. arbiter의 state transition diagram은 다음과 같다.    M0grant : master 0의 input data들이 Bus에서 기능을 한다.  M1grant : master 1의 input data들이 Bus에서 기능을 한다.  초기 상태는 M0grant이다.  M0grant일 때 M0\_req==0 && M1\_req==1인 상태에서 M1\_grant로 상태가 변화하고, 나머지 경우일 땐 원래 상태를 유지한다.  M1grant일 때 M0\_req==1 && M1\_req==0인 상태에서 M0\_grant로 상태가 변화하고, 나머지 경우일 땐 원래 상태를 유지한다. |
| **4. MULTIPLIER** |
| 1) pin description   |  |  |  | | --- | --- | --- | | input | clk | clock | | reset\_n | reset signal | | op\_start | 연산의 시작을 알리는 신호 | | op\_clear | 초기화를 시켜주는 신호 | | multiplier | 32-bit input data | | multiplicand | 32-bit input data | | output | op\_done | 연산이 끝났다는 것을 알려주는 신호 | | result | 128-bit 곱셈 연산 결과 data | | reg | count | 연산이 몇 번 수행되었는지 count하는 신호 | | next\_count | next\_count | | next\_result | next result value | | r\_multiplier | input으로 들어온 multiplier의 값이 cycle마다 2-bit씩 shift되는 값, 최하위 3bit이 x(i) x(i-1) x(i-2)를 나타낸다. | | next\_r\_multiplier | next r\_multiplier | | shift\_scale | result를 몇 bit shift해줄지 결정해주는 신호 | | next\_shfit\_scale | next\_shfit\_scale | | wire | w\_count | count+2 | | w\_multiplicand | operate value of multiplicand | | w\_result | result operation value of multiplicand | | w\_shift\_scale | shift-scale에 연결될 cla 연산결과 | | w\_ci | cla64의 carry-in, 보수를 더해줄 때 필요한 ci |   2)block diagram    MULTIPLIER의 schematic symbol 이다. 왼쪽은 input data를 나타내고 오른쪽은 output data를 나타낸다.  64-bit multiplier와 64-bit multiplicand가 곱해져서 128-bit result로 나온다. op 신호들은 state를 조정하는 input output data이다.  3)FSM    4)State description   |  |  | | --- | --- | | IDLE | 처음 초기 상태이다. op\_start가 1이 되면 연산을 하는 EXECUTE로 상태가 천이 된다. 처음 reset이 활성화 되었을 때 의 state이다. | | EXECUTE | 연산을 수행하는 상태이다. 연산 중 op\_clear가 1로 set되면 IDLE로 상태가 천이되고, 연산이 완료되어 op\_done이 1로 set되면 DONE상태로 천이된다. | | DONE | 연산이 끝난 상태이다. op\_clear가 1이되면 다시 연산을 준비하는 상태인 IDLE로 상태가 천이된다. |   5) register description     |  |  | | --- | --- | | **Name** | **Description** | | count[5:0] | 연산의 횟수를 저장하고 있는 register이다. radix4에선 multipler를 3-bit씩 읽어 result값에 알맞은 연산 후 2-bit씩 shift하는데 2-bit씩 shfit하는 것을 한 번의 연산이라고 본다. | | \*next logic of count  next\_count는 현재 상태에 따라서 어떤 값이 저장 될지 결정 된다.  IDLE : counting 할 필요가 없으므로 next\_count= 0이다.  EXECUTE : counting을 해야 한다. next\_count는 w\_count가 된다. w\_count는 현재 count의 값에 shift\_scale값이 더해진 값이다.  DONE : 연산이 끝난 상태이므로 counting해줄 필요가 없다. op\_clear가 1이 되기 전 까진 이전 값을 유지하고 있다가 op\_clear가 1이 되면 count는 0으로 reset된다. | | | **Name** | **Description** | | r\_multiplier[64:0] | radix4에서 x(i) x(i-1) x(i-2)의 값이 [2:0]에 저장 되어있는 register이다. r\_multiplier의 하위 5-bit을 보고 result에 어떤 연산을 해줄지 결정한다. | | \*next logic of r\_multiplier  next\_r\_multiplier도 현재 상태에 따라서 어떤 값이 저장 될지 결정된다. r\_multiplier의 최하위 3-bit에는 radix4의 x(i) x(i-1) x(i-2)가 저장 되어 있다.  IDLE : 연산을 준비하는 상태이므로, input으로 들어온 multiplier의 값이 상위 64-bit에 저장되고 최하위 bit에는 0이 저장된다.  EXECUTE: 연산을 한 cycle진행 할 때마다 next\_r\_multiplier에 이전의 값이 2-bit left shift된 값이 저장 된다.  DONE : 연산이 종료 되었기 때문에 변할 필요가 없다. 따라서 next\_r\_multiplier는 이전값 r\_multiplier로 설정 해준다. | | | **Name** | **Description** | | shift\_scale | 한 cycle에서 몇 bit를 shift할지를 결정해주는 값이 저장 되어있는 register이다. | | \*next logic of shift\_scale  next\_shift\_scale은 현재상태와 r\_multiplier의 값에 따라 어떤 값이 저장 될지 결정된다.  IDLE : shift\_scale의 보통 값은 count에 1이 더해져야 하기 때문에 1이다. 따라서 IDLE상태에선 보통값인 1을 갖는다.    EXECUTE : 연산이 실행중인 이 상태에서는 r\_multiplier의 값에 따라 next\_shift\_scale이 정해진다. r\_multiplier의 모든 bit이 0일 때에는 원래대로라면 1cycle당 계속해서 2-bit씩만 결과값을 Arithmetic Shift Right 해야 하지만 이것을 한번에 해주는 logic이다. r\_multiplier의 모든 bit이 0일 때 next\_shift\_scale은 w\_shift\_scale이 된다. w\_shift\_scale은 31에서 현재 count를 뺀 값이다. 원래 count가 32가 되면 연산이 끝나는데, 앞으로의 남은 연산 count를 계산해서 그만큼 shift해야 할 값을 next\_shift\_scale에 저장 하는 것이다. r\_multiplier의 모든 bit이 0이 아닐 때에는 next\_shift\_scale에 원래의 shift scale인 1의 값을 저장 해준다.    DONE : 더 이상 shift될 필요가 없고, count도 더해질 필요가 없으므로 next\_shift\_scale은 0이 저장된다. | | | **Name** | **Description** | | result | output으로도 쓰이며 곱셈 결과가 저장 되어있는 register이다. execute에서는 곱셈연산이 완료되기 전까지 logic에 따라 계속해서 값이 바뀌어 저장된다. | | \*next logic of result  next\_result또한 상태에 따라 값이 다르게 저장 된다.  IDLE : 곱셈연산의 준비 단계이므로 아직 결과물이 없으므로 next\_result에 0을 저장해준다.  EXECUTE : 만약 count가 32보다 크거나 같다면 연산이 끝난 것이므로 next\_result는 이전값이 저장된다.  그렇지 않고 만약 shift\_scale이 1이 아니라면 shift\_scale 크기만큼 result를 ASR 해주어야 하는 조건이다. next\_result에 result의 shift\_scale의 2배크기만큼 ASR 된 값이 저장 된다. 이 때 shift\_scale을 2배해주는 이유는 radix4이기 때문이다. radix2에서는 shift\_scale을 2배해줄 필요 없지만, radix4에서는 count와 result의 bit 수가 맞지 않기 때문에 2배해주어 저장 해준값을 ASR한 것을 저장 해야 한다.  이 외의 경우에는 일반적인 radix4 연산값을 next\_result에 저장 해주면 된다. 아래에 표로 나타 내었다.   |  |  |  | | --- | --- | --- | | **r\_multiplier [2] [1] [0]** | **r\_multiplier [4] [3] [2]** | **next\_result** | | +2A or -2A | +2A or -2A | 2-bit Arithmetic Shift Right of w\_result | | +2A or -2A | +A or –A or 0 | 1-bit Arithmetic Shift Right of w\_result | | 0 | +2A or -2A | 3-bit Arithmetic Shift Right of result(previous value) | | 0 | +A or –A or 0 | 2-bit Arithmetic Shift Right of result(previous value) | | +A or –A | +2A or -2A | 3-bit Arithmetic Shift Right of w\_result | | +A or -A | +2A or -2A | 2-bit Arithmetic Shift Right of w\_result |   w\_result 는 w\_multiplicand와 현재 result의 상위 64-bit과 w\_ci 값을 더한 결과 값이다.  w\_multiplicand – r\_multiplier[2][1][0] 이 -2A, -A일 경우 ~multiplicand의 값이 들어간다. 그 이외에는 multiplicand값이 들어간다.  w\_ci - r\_multiplier[2][1][0] 이 -2A, -A일 경우 1의 값이 들어간다. 그 이외에는 0이 들어간다. | |   6)output description |
| **5. FACTORIAL** |
| 1) pin description   |  |  |  | | --- | --- | --- | | input | clk | clock | | reset\_n | reset signal | | S\_sel | (slave) select | | S\_wr | (slave) write or read | | S\_address[7:0] | (slave) address | | S\_din[31:0] | (slave)input data | | M\_grant | (master) BUS grant를 받았는지의 여부. M\_grant가 1일 경우에 master로서 slave들을 조정 할 수 있다. M\_req를 통하여 요청 할 수 있다. | | M\_din[31:0] | (master) input data | | output | S\_dout[31:0] | (slave)data out | | M\_req | (master)BUS grant를 요청 하는 신호이다. | | M\_wr | (master) write or read –slave의 input wr로 BUS를 통해 이동된다. | | M\_address | (master) address – slave의 input address로 BUS를 통해 이동된다. | | M\_dout[31:0] | (master) output data – slave의 input data로 BUS를 통해 이동된다. | | interrupt | factorial이 기능을 다한 후 processor로 발생시키는 신호 | | wire | op\_done | output of MULTIPLIER – 연산 종료를 알린다. | | mul\_result | output of MULTIPLIER – 곱셈 결과이다. |   2) block diagram    FACTORIAL의 schematic symbol 이다. 왼쪽편은 input을 나타내며, 오른쪽은 output을 나타낸다.  3)FSM  **4) state description**   |  |  | | --- | --- | | **IDLE** | 초기상태 이다. Interrupt\_Enable 신호가1이 되면 INTERRUPT\_ENABLE 상태로 이동한다. 아닐 경우엔 IDLE상태에 머무른다 | | **INTERRUPT\_ENABLE** | Interrupt\_Enable신호가 1이된 상태이다. Op\_start가 1이 되면 BUS\_REQUEST상태로 이동한다. | | **BUS\_REQUEST** | BUS grant를 요청 하는 상태이다. output인 M\_req를 1로 만들어 준다. M\_grant가 1이 되면 BUS\_GRANT상태로 이동하고, M\_grant가 0일 경우엔 BUSY상태로 이동한다. | | **BUSY** | BUS grant를 요청 하는 상태이지만, processor의 다른 할 일에 의해 아직 grant가 주어지지 않은 상태이다. 계속해서 M\_req를 1로 유지하면서 M\_grant가 1이 될 때까지 상태를 유지한다. M\_grant가 1이 되면 BUS\_GRNAT상태로 이동한다. | | **BUS\_GRANT** | BUS grant를 받은 상태이다. 이 때 M\_address를 FIFOTOP\_IN의 FIFOFLAG를 선택하는 0x10으로 주고, M\_wr을 0으로 주어 FIFOTOP\_IN에게 data를 요청한다. 그러면 한 cycle뒤에 data가 BUS를 통해 도착한다. 다음 cycle에 바로 CHECK\_FLAG 상태로 이동한다. | | **CHECK\_FLAG** | BUS로부터 받은 M\_din을 조사한다. M\_din[4]가 FIFOTOP\_IN의 empty인지의 여부를 판단하는데 1일 경우 empty이므로 FACTORIAL연산을 종료 하는 staged의 처음인 NOT\_BUS\_GRANT2로 이동한다. 0일 경우 empty가 아니므로 REQUEST\_N\_VALUE 상태로 이동한다. | | **REQUEST\_N\_VALUE** | M\_address를 FIFOTOP\_IN의 FIFO를 선택하는 0x11, M\_wr을 0으로 주어 FIFOTOP\_IN에게 data를 요청한다. 그러면 한 cycle뒤에 data가 BUS를 통해 도착한다. 다음 cycle에 바로 N\_VALUE\_READ상태로 이동한다. | | **N\_VALUE\_READ** | BUS로부터 받은 M\_din을 FACTORIAL의 register에 저장한다. 바로 다음 상태인 CHECK\_N\_VALUE로 이동한다. | | **CHECK\_N\_VALUE** | : N\_VALUE를 조사하는 상태이다. 이 때 result에 알맞은 값을 써준다. 2 일 경우엔 2로 0또는 1일 경우에는 1로 그 외의 경우에는 N\_value를 result에 그대로 써준다. N\_VALUE가 0 또는 1 또는 2이면 곱셈 연산을 해줄 필요가 없으므로 바로 WRITE\_RESULT\_REGISTER상태로 이동한다. 0, 1, 2가 아닐 경우 MULTIPLE\_EXECUTE상태로 이동한다. | | **MULTIPLE\_EXECUTE** | 곱셈연산을 하는 상태이다. 이 때 multiplicand는 result가 된다. multiplier의 값은 N\_value-1값이 된다. MULTIPLIER의 input인 op\_start에 1값을 넣어주면 연산이 시작된다. MULTIPLIER의 output인 op\_done이 1이 될 때까지 해당 상태를 유지한다. op\_done이 1이 되었을 경우 SUB\_MULTIPLIE\_VALUE상태로 이동한다. | | **SUB\_MULTIPLE**  **\_VALUE** | 곱셈연산후의 결과값이 result에 저장된다. 만약 이 state일 때 N\_value가 2라면, n부터 2까지의 곱셈연산이 진행 된 것이므로, factorial연산이 끝난것이다. N\_value가 2일 경우엔 WRITE\_RESULT\_REGISTER로 이동한다. 아닐 경우엔 다음 곱셈연산을 준비한다. N\_value를 -1 해주고, MULTIPLIER의 input인 op\_clear신호를 1로 set해준다. 그러면 MULTIPLIER는 다음 cycle에서 초기상태인 IDLE상태로 돌아가게 된다. N\_value가 2가 아닐 경우엔 그대로 MULTIPLE\_EXECUTE상태로 되돌아가서 남은 factorial연산을 진행해주어야 한다. | | **WRITE\_RESULT**  **\_REGISTER** | factorial연산이 끝난 후 result에 저장된 값이 FACTORIAL내부 register값인 32-bit register 4개에 나눠서 저장이 되는 state이다. result의 상위 32-bit부터 차례대로 register Result3, Result2, Reuslt1, Result0에 저장이 된다. 바로 다음 state인 BUS\_REQUEST2로 이동한다. | | **BUS\_REQUEST2** | BUS grant를 요청하는 상태이다. output인 M\_req를 1로 만들어 준다. M\_grant가 1이 되면 BUS\_GRANT상태로 이동하고, M\_grant가 0일 경우엔 BUSY상태로 이동한다. | | **BUSY2** | BUS grant를 요청 하는 상태이지만, processor의 다른 할 일에 의해 아직 grant가 주어지지 않은 상태이다. 계속해서 M\_req를 1로 유지하면서 M\_grant가 1이 될 때까지 상태를 유지한다. | | **BUS\_GRANT2** | BUS grant를 받은 상태이다. 이 때 M\_address를 FIFOTOP\_OUT의 FIFOFLAG를 선택하는 0x20으로 주고, M\_wr을 0으로 주어 FIFOTOP\_OUT에게 data를 요청한다. 그러면 한 cycle뒤에 data가 BUS를 통해 도착한다. 바로 CHECK\_FLAG 상태로 이동한다. | | **CHECK\_FLAG2** | BUS로부터 받은 M\_din을 조사한다. M\_din[5]가 FIFOTOP\_OUT의 full인지의 여부를 판단하는데 1일 경우 full이므로 FIFOTOP\_OUT에 더 이상 데이터를 쓸 수가 없다. 따라서 NOT\_BUS\_GRANT2로 이동한다. 0일 경우 FULL이 아니므로 WRITE\_RESULT2 상태로 이동한다. 이동하기 전에 현재 상태에서 M\_address를 FIFOTOP\_OUT의 FIFO를 선택하는 0x21로 바꿔주고 M\_wr을 1로 주어 FIFOTOP\_OUT에게 M\_dout을 보낸다. Result3가 M\_dout이다. | | **WRITE\_RESULT2** | 마찬가지로 Result2를 FIFO에 써야하는 상태이다. M\_address는 0x21이고 M\_wr는 1이다. M\_dout을 Result2로 설정해준다. 바로 WRITE\_RESULT1 상태로이동한다. | | **WRITE\_RESULT1** | WRITE\_RESULT2와 같다. M\_dout을 Result1으로 설정해준다. 바로 WRITE\_RESULT0 상태로 이동한다. | | **WRITE\_RESULT0** | WRITE\_RESULT1과 같다. M\_dout을 Result0로 설정 해준다. 다시 BUS\_GRANT상태로 이동해서 cycle을 반복한다. | | **NOT\_BUS\_GRANT2** | FACTORIAL이 FIFOTOP\_OUT이 가득 차서 연산을 할 필요가 없거나, FIFOTOP\_IN이 비어서 더 이상 연산을 할 데이터가 없을 때 이동 되는 state이다. BUS grant를 놓는 state로서 M-req를 0으로 만들어주면 된다. 이 때 Interrupt register에 1을 써주어 다음 cycle에 interrupt output이 발생하도록 만들어 준다. 바로 INTERRUPT\_WAIT 상태로 이동한다. | | **INTERRUPT\_WAIT** | interrupt가 0이 되길 기다리는 state이다. test bench에서 factorial을 slave로 선택 한 후 Interrupt register에 0을 써주면 해당 state에서 다음 state로 넘어가게 된다. 다음 state는 INITIALIZATION이다. | | **INITIALIZATION** | 모든 register들을 초기화 시켜주어 FACTORIAL이 다시 초기 상태로 돌아갈 수 있게끔 만들어주는 상태이다. 바로 IDLE상태로 이동한다. |   5) register description   |  |  |  |  | | --- | --- | --- | --- | | **offset** | **Type** | **Name** | **Description** | | 0x0 | READ | N\_value | factorial연산을 할 N값이 저장되는 register이다. | | \*next logic of N\_value  Factorial의 많은 상태 중 N\_value가 처음에 저장되는 state는 N\_VALUE\_READ이다. 이 전 state인 REQUEST\_N\_VALUE 상태에서 FIFOTOP\_IN에게 FIFO의 dequeue data를 보내달라고 요청한다. 그러면 N\_VALUE\_READ일 때 그 data가 도착을 하게 되고 이 상태에서 next\_N\_value에 M\_din의 값을 넣어주게 되면 factorial 연산을 할 N값이 정상적으로 read된다.  그리고 다음 상태인 CHECK\_N\_VALUE에서는 next\_N\_value가 이전 값을 갖게 되고, 연산을 하는 MULTIPLIE\_VALUE에서도 이전 값을 그대로 갖게 된다. SUB\_MULTIPLE\_VALUE state에서 next\_N\_value의 값은 N\_value-1이 되어 저장 된다. multiplier에게 -1된 값을 주기 위함이다. 이 N\_value는 WRITE\_RESULT\_REGISTER 상태에서도 이전값을 유지하지만,  나머지 상태에서는 N\_value가 쓸모가 없으므로 0으로 default value를 갖게 된다. | | | | | **offset** | **Type** | **Name** | **Description** | | 0x1 | R/W | Interrupt\_Enable | 이 register의 [0] bit이 1일 경우엔, FACTORIAL의 계산할 값이 없을 때 output interrupt가 1이 되는 것이 가능하다.  해당 register의 [31:1]bits는 reserved로 사용되지 않는다. | | \*next logic of Interrupt\_Enable  FACTORIAL의 초기 상태인 IDLE에서 FACTORIAL이 slave로서 선택되고, S\_wr이 1이고 해당 offset인 0x1이 선택 되었을 때 next\_Interrupt\_Enable은 S\_din값을 취한다. 나머지 상태에서는 next\_Interrupt\_Enable은 이전 값을 갖는다. 예외적으로 INITIALIZATION에서는 FACTORIAL을 초기상태로 돌려 놔야 하기 때문에 next\_Interrupt\_Enable값이 0을 갖는다. | | | | | **offset** | **Type** | **Name** | **Description** | | 0x2 | R/W | Interrupt | FACTORIAL이 기능을 다하면 해당 register의 [0]bit에 1을 써주어 output인 interrupt를 발생시킨다. interrupt가 발생하면 testbench가 해당 register의 [0]bit에 다시 0을 써줌으로 써 interrupt를 해소시킨다. | | \*next logic of Interrupt  FACTORIAL이 기능을 다 끝낸 상태는 NOT\_BUT\_GRANT2상태이다. 이 상태로 오기 위한 조건은 FIFOTOP\_IN이 empty일 때나, FIFOTOP\_OUT이 full일 때이다. 더 이상 연산을 할 수 없거나 할 필요가 없을 때이다. 이 상태에서 next\_Interrupt를 1로 set해준다.  그리고 나서 다음 state인 INTERRUPT\_WAIT 상태에서 해당 FACTORIAL이 선택되고, S\_wr이 1값을 갖고 해당 register의 offset인 0x2가 선택 되었을 때 next\_Interrupt에 S\_din을 취한다.  나머지 경우에는 next\_Interrupt는 이전 값을 가지면 된다. | | | | | **offset** | **Type** | **Name** | **Description** | | 0x3 | W | OperationStart | 해당 register에 [0]bit에 1이 써질 경우에 factorial의 연산이 시작된다. 나머지 [31:1]bit은 reserved이다. | | \*next logic of OperationStart  FACTORIAL이 slave로서 선택되고, S\_wr이 1값을 갖고, S\_address가 해당 register의 offset인 0x3일 때 만 next\_OperationStart는 S\_din값을 취한다. 그 외의 경우에는 next\_OperationStart는 0값을 갖는다. | | | | | **offset** | **Type** | **Name** | **Description** | | 0x4 | R | Result0 | FACTORIAL의 128-bit 연산결과가 32-bit씩 나누어 저장하고 있는 register이다. Result0부터 하위 32-bit부터 차례대로 저장되어 있는 register이다. | | 0x5 | R | Result1 | | 0x6 | R | Result2 | | 0x7 | R | Result3 | | \*next logic of Result0~Result3  FACTORIAL 연산이 끝난 직후의 상태는 WRITE\_RESULT\_REGISTER이다. 이 상태로 오기 위한 조건은 MULTIPLIER로부터의 factorial연산 후 N\_value의 값이 2가 되어 더 이상 곱셈을 진행할 필요가 없을 때 혹은 FIFOTOP\_IN에서 읽어온 값을 check했을 때 2이하의 수일 때이다.  WRITE\_RESULT\_REGISTER 상태에서 FACTORIAL연산의 결과가 저장 되어있는 result[127:0]이 32-bit씩 나누어 next\_Result3~0 까지에 저장된다.  그리고 INITIALIZATION상태에서 next\_Result3~0는 모두 0으로 초기화된다.  나머지 상태에서는 next-Result3~0은 모두 이전 값을 갖는다. | | | |   6) output description     |  | | --- | | * **output logic of master elements** | | master elements에는 M\_req, M\_address, M\_wr, M\_dout이 있다. M\_req는 BUS grant를 요청하는 신호이다. M\_address와 M\_wr과 M\_dout은 FACTORIAL이 BUS grant를 가질 때 slave로 버스를 통해 보내는 data들이다. M\_req같은 경우엔 FACTORIAL이 BUS grant를 가지고 있더라도 안전한 동작을 하기 위해 grant를 가져야 하는 기간 동안에는 M\_req를 1로 유지한다. 그리고 FACTORIAL의 slave로 선택 될 수 있는 component는 FIFOTOP\_IN과 FIFOTOP\_OUT이다. FIFOTOP에 write를 할 때에는 FIFOTOP에서 몇 cycle이 걸리든 상관 없지만, read동작을 할 때에 올바른 결과값을 얻기 위해서는 data를 받아들일 state보다 1 cycle전에 M\_address와 M\_wr을 통해 값을 요청해야 한다. 이는 FIFOTOP에서 data를 dequeue하는데 1 cycle이 소모되기 때문이다. FIFOTOP이 address와 wr, sel 신호를 받으면 해당하는 주소값의 데이터가 1 cycle후에 FIFOTOP의 dout으로 나오게 되는 것이다.  1. FACTORIAL이 BUS\_REQUEST상태일 때 BUS grant를 요청한다. 따라서 M\_req=1이 된다. BUS\_REQUEST2, BUSY, BUSY2 상태일 때도 마찬가지로 M\_req는 1이 된다. 이 때에 address, wr값은 default가 된다. 왜냐하면 data를 읽고 쓰려는 목적이 아니고 grant를 받으려는 목적이기 때문이다.  2. FACTORIAL이 BUS\_GRANT 또는 BUS\_GRANT2 상태일 때 M\_address는 0x10 또는 0x20, M\_wr은 0 이다. 이 address는 FIFOTOP의 FIFOFLAG를 나타내는 주소값이다. BUS\_GRANT의 다음상태가 FLAG를 check하는 상태이므로, 1cycle미리 주소값과 wr을 보내주는 것이다.  3. REQUEST\_N\_VALUE는 FIFOTOP\_IN의 FLAG를 check한 뒤 empty가 아닐 경우 이동 되는 state이다. 따라서 이 state에서 M\_address 0x11, M\_wr를 0 으로 보내면 FIFO의 dequeue값을 다음 state인 N\_VALUE\_READ state에서 받을 수 있다.  4. CHECK\_FLAG2는 BUS\_GRANT2의 다음 상태이다. 이 상태에서는 FIFOTOP\_OUT이 비었는지 확인하고 동시에 M\_address 0x21 M\_wr을 1, Mdout은 Result3로 주어 FIFOTOP\_OUT에 Result3를 쓴다.  5. WRITE\_RESULT2 상태에서는 M\_address : 0x21 M\_wr :1 Mdout을 Result2로 주어 버스를 통해 FIFOTOP\_OUT에 Result2를 쓴다.  6. WRITE\_RESULT1 상태에서는 M\_address : 0x21 M\_wr :1 Mdout을 Result1로 주어 버스를 통해 FIFOTOP\_OUT에 Result1를 쓴다.  7. WRITE\_RESULT0 상태에서는 M\_address : 0x21 M\_wr :1 Mdout을 Result0로 주어 버스를 통해 FIFOTOP\_OUT에 Result0를 쓴다.  이 외의 경우에 master elements들은 default value를 갖는다.  default value : M\_req = 0 , M\_address = 0xff , M\_wr=0 , M\_dout=0 이다. | | * **output logic of S\_dout** | | S\_dout은 FACTORIAL이 slave로서 선택 되었을 때 wr이 0으로 들어왔을 때만  해당 하는 주소값에 따라 register의 값이 출력되는 output이다.  S\_address가 0x00 일 때 S\_dout은 N\_value이다..  S\_address가 0x01 일 때 S\_dout은 Interrupt\_Enable이다.  S\_address가 0x02 일 때 S\_dout은 Interrupt이다..  S\_address가 0x04 일 때 S\_dout은 Result0이다.  S\_address가 0x05 일 때 S\_dout은 Result1이다..  S\_address가 0x06 일 때 S\_dout은 Result2이다.  S\_address가 0x07 일 때 S\_dout은 Result3이다. | | * **output logic of interrupt** | | output interrupt는 Interrupt[0]과 Interrupt\_Enable[0]값이 1일 때만 1이다. | |
| **6. TOP** |
| 1) pin description   |  |  |  | | --- | --- | --- | | input | clk | clock | | reset\_n | reset signal | | M0\_req | master0 request | | M0\_wr | master0 read or write signal | | M0\_address[7:0] | master0 address | | M0\_dout[31:0] | master0 output data | | output | M0\_grant | master 0 grant | | interrupt | FACTORIAL interrupt signal | | M\_din[31:0] | master input data | | fifo\_cnt\_in[3:0] | FIFOTOP\_IN data 수 | | fifo\_cnt\_out[5:0] | FIFOTOP\_OUT data 수 | | fifo\_flag\_in[5:0] | FIFOTOP\_IN 의 flag | | fifo\_flog\_out[5:0] | FIFOTOP\_OUT 의 flag |   위의 표는 TOP의 input, output signal이다.  memory map은 다음과 같다.   |  |  | | --- | --- | | address | component | | 0x00~0x0F | FACTORIAL | | 0x10~0x1F | FIFOTOP\_IN | | 0x20~0x2F | FIFOTOP\_OUT |   2) block diagram |

|  |
| --- |
| **Ⅳ. Conclusion Design Verification Strategy and Results** |
| **1. FIFOIN\_TOP** |
| factorial의 연산을 할 데이터들이 저장되는 장치이다.  검증전략은 다음과 같다.  1. 8개의 data들이 잘 write되고, read 되는가? FULL일 때 data가 쓰여지지 않아야 한다. EMPTY일 때 데이터가 읽어지지 않아야한다.  2. handshake signal인 flag값들이 잘 출력 되는가?  3. 해당 module이 select 되지 않거나, FIFO가 주소값을 통하여 select되지 않았을 때 올바른 결과가 출력되는가?    reset\_n이 1이 된 직후부터 sel 1, wr 1, address를 0x11 즉 FIFO를 선택하고, write기능을 한다. 00000000부터 차례대로 FIFO에 write될 때마다 count가 1씩 증가하는 것을 볼 수 있다. 이 때 fifo\_flag는 001000 rd\_ack만 1이 된 것을 확인 할 수 있다. din에 77777777이 입력을 마지막으로 8에서 더 이상 cnt는 증가하지 않는 것을 볼 수 있다. 그리고 이 때 fifo\_flag는 101000 즉 rd\_ack와 full이 1로 set된 것을 확인 할 수 있다. 그 후에 88888888이 입력 되었을 때 더 이상 cnt는 증가 하지 않으며, 이 때 fifo\_flag는 100100이다. full상태와 wr\_err이 1로 set된 것을 확인 할 수 있다. 그 후에 address가 0x10, wr 0 으로 입력을 주었다. 이 것은 FIFO\_FLAG register의 값을 read하겠다는 뜻이다. 이전의 fifo\_flag가 dout으로 출력 되는 것을 확인 할 수 있다.    190000ps 부근에서 wr이 0으로 바뀌고 address가 11이다. 이 때에는 FIFO에 대하여 read기능이 수행되어야 한다. cnt가 8부터 1씩 줄며 0000000 부터 7777777까지 dout으로 출력되는 것을 확인 할 수 있다. 이 때 fifo\_flag는 000010 즉 rd\_ack 1로 set되 있는 것을 확인 할 수 있다. 그리고 77777777이 출력 되었을 때 fifo\_flag는 010010 즉 rd\_ack와 empty가 1로 set되어 있다. 그리고 그 후에 한번 더 read명령이 수행되자 fifo\_flag는 010001로 바뀌었다. 즉 rd\_err과 empty가 1로 set되어 있는 것이다. 270000ps 이후 sel이 0으로 내려갔다. 그 후로 fifo\_flag가 출력되지 않는 것을 확인 할 수 있다. sel이 1이되고 주소값이 이상 한 곳을 가르켜도 fifo\_flag의 값은 나오지 않는다. 올바르게 작동하는 것을 확인 할 수 있다.  위의 검증으로 검증전략 1. 2. 3. 을 모두 만족하는 것을 확인 하였다. |
| **2. FIFOOUT\_TOP** |
| FIFOIN\_TOP과 저장할 수 있는 개수만 다르고 동일한 알고리즘이다. 따라서 32개가 잘 저장 되는지만 검증 해보면 된다.      fifo\_cnt가 32가 되자 fifo\_flag가 101000이 되었다. 즉 full flag가 올라 간 것을 확인 할 수 있다.      그 후 fifo\_cnt가 0이 될 때 까지 data가 잘 출력 되는 것을 확인 할 수 있다. 0이 되었을 때 flag는 010010인 것을 확인 할 수 있다. empty와 rd\_ack가 1이다. |
| **3. MULTIPLIER** |
| 검증 전략 :  1.양수와 음수, 음수와 음수 모든 경우에 cover되는지 확인한다.  2. radix4 알고리즘을 사용 하였으므로 x(i) x(i-1) (i-2) 의 총 8가지의 경우에 모두 잘 되는지 확인한다.  3. radix4 알고리즘에 multiplier의 상위 bit을 보고 다 0일 경우 그대로 shift만하고 연산을 끝내는 logic을 추가하였으므로 연산이 예상보다 빠른 cycle에 끝나는지 확인한다.  4. 연산중에 op\_clear신호가 들어오면 연산을 멈추고 초기상태로 돌아가는지 확인하다.    multiplier는 6 multiplicand는 -6이다. 따라서 multiplier가 하위 110을 제외하고 모든 bit이 0이기 때문에 4cycle에 연산이 끝난 것을 확인 할 수 있다. 결과값도 tb\_result에 제대로 써진 것을 확인 할 수 있다.  검증 전략 3.을 확인 하였다.    multiplier는 -6 multiplicand는 6이다. 32cycle후에 -36 결과값이 나온 것을 확인 할 수 있다.  검증전략 1.의 일부를 확인 할 수 있다.    multiplier, multiplicand 모두 음수이다. 32cycle후에 225제대로된 결과가 나오는 것을 확인 할 수 있다. 검증전략 1.의 일부를 확인 할 수 있다.    multiplier는 42962 multiplicand는 5이다. 42962는 10100111110100010이다. LSB부터 x(i) x(i-1) x(x-2)로 나타내면 100, 001, 010, 110, 111, 011, 100, 101, 001, 000 으로 나눌 수 있다. 따라서 곱셈 결과가 제대로 나온다면 radix-4의 8개 operation을 모두 검증 할 수 있다. multiplier를 10진수로 나타내면 42962이다. 42962 \* 3 = 128886이므로 결과가 제대로 나온 것을 확인 할 수 있다.  따라서 검증전략 1. 2. 를 확인 하였다.    tb\_op\_start가 0에서 1이 된 후 연산이 시작된다. 중간에 op\_clear가 1이 되자 연산을 멈추고 result가 0으로 돌아간 것을 확인 할 수 있다. 검증전략 4.를 확인 하였다. |
| **4. BUS** |
| 검증전략  1. Master의 address에 따라 알맞은 slave가 선택 되는가, wr에 따라 알맞은 기능을 수행하는가?  2.Grnat가 잘 옮겨 가는가?    wave 처음은 M0\_grant이다. 그리고 처음에 M0\_address는 00이고, M0\_wr는 0이다. 따라서 M\_din은 S\_din이 되어야 한다. 알맞게 수행 되는 것을 볼 수 있다. 그 후에 M0\_wr이 1로 올라 갈 때 S\_wr도 따라서 올라가는 것을 볼 수 있으며, 이 때에는 write 기능을 하게 된다. 해당 하는 M0\_address의 M0\_dout이 S\_din으로 가는 것을 볼 수 있다. 그리고 끝부분에 M1\_req가 1로 되고, M0\_req가 0이 되는데 이 때 grant가 M0에서 M1으로 넘어가야 옳다. M1\_grant로 넘어가는 것을 확인 할 수 있다.  검증전략 1. 2. 를 모두 검증하였다. |
| **5. FACTORIAL** |
| factorial 연산이 제대로 되는지의 검증은 TOP부분에서 다루었다. 따라서 FACTORIAL에서는 내가 FSM을 설계한대로 상태가 잘 이동되는지를 검증한다.    1. 처음에 reset에 의해 초기화가 되고나서 S\_address 0x01, S\_wr 1, S\_din 1 로 준다.  IDLE 🡪 INTERRUPT\_ENABLE  2. S\_address 0x03  INTERRUPT\_ENABLE 🡪 BUS\_REQUEST  3. M\_grant를 받는다. – FACTORIAL이 master가 되었을 때이다.  BUS\_REQUEST 🡪 BUS\_GRANT  4. 바로 M\_address가 10이 되는 것을 볼 수 있다. 이것은 FIFOTOP\_IN의 FIFOFLAG를 읽겠다고 요청하는 것이다.  BUS\_GRANT 🡪 CHECKFLAG  5. M\_din을 100000으로 주었기 때문에 N\_value를 요청하는 state로 넘어간다.  CHECKFLAG 🡪REQUEST\_N\_VALUE  6. M\_address가 11 이 된 것을 볼 수 있다. 이것은 FIFOTOP\_IN의 FIFO에게 data read를 요청하는 것이다.  REQUEST\_N\_VALUE 🡪 N\_VALUE\_READ  7. M\_din을 10으로 주었다. 즉 연산할 값이다. FACTORIAL은 이 값을 N\_value register에 저장할 것이다.  N\_VALUE\_READ 🡪CHECK\_N\_VALUE 🡪 MULTIPLER\_EXECUTE    8. FACTORIAL이 MULTIPLIER를 통해 연산하는 중이다. tb\_FACTORIAL에서 while문을 통해 연산이 끝나고 연산값을 register에 저장한 후에 M\_req가 1이 될 때까지 반복문을 실행했다.  MULTIPLER\_EXECUTE🡪 SUB\_MULTIPLE\_VALUE 이 과정이 계속 반복되다가 factorial 연산이 끝나면,  SUB\_MULTIPLE\_VALUE 🡪 WRITE\_RESULT\_REGISTER로 넘어간다.  WRITE\_RESULT\_REGISTER 🡪 BUS\_REQUEST2 상태가 되면 M\_req가 1이 된다.    9. 처음 부분이 M\_req가 1이 된 순간이다. 그 후에 바로 M\_grant를 1로 준다.  BUS\_REQUEST2 🡪 BUS\_GRANT2🡪CHECK\_FLAG2  10. FACTORIAL의 M\_address가 0x20이 된 것을 확인 할 수 있다. 이것은 FIFO\_TOP\_OUT의 FIFOFLAG을 요청 하는 것이다. 따라서 empty값인 16을 M\_din으로 준다.  CHECK\_FLAG2 🡪 WRITE\_RESULT2  11. M\_address가 21이 된 것은 FIFOTOP\_OUT의 FIFO주소값이다. 이곳에 데이터를 wirte하겠다는 뜻이다.  4cycle동안 M\_dout으로 data가 출력 되는 것을 볼 수 있다. 그리고 노랑선 부분에 10! 의 결과인 3628800이 출력되는 것을 볼 수 있다.  WRITE\_RESULT2 🡪 WRITE\_RESULT1 🡪WRITE\_RESULT 0  12. FACTORIAL M\_add가 다시 10이 되었다.  WRITE\_RESULT0 🡪 CHECK\_FLAG  13. FIFOTOP\_IN에게 다시 FIFOFLAG를 요청 하는 것이다. 이 때 M\_din의 값이 empty를 의미하는 16이므로 interrupt를 발생시키는 부분으로 넘어가게 된다.  CHECK\_FLAG 🡪 NOT\_BUS\_GRANT2 🡪 INTERRUPT\_WAIT  14. FACTORIAL가 interrupt를 발생시키는 것을 확인 할 수 있다. 그 후에 SLAVE input을 통하여 interrupt를 0으로 만들어주면 FACTORIAL은 초기 상태로 돌아가게된다.  INTERRUPT\_WAIT 🡪INITIALIZATION 🡪IDLE |
| **6. TOP** |
| 가장 상위 module인 TOP의 검증이다. testbench\_TOP을 통하여 검증을 하였다.  검증전략은 다음과 같다.  1. 0~20의 수에 대해서 factorial 연산결과가 올바르게 나오는가? 특히 0, 1, 2는 multiplier연산 없이 바로 나오는가?  2. 8개의 data가 입력 되었을 때 8개의 data에 대한 factorial 결과가 모두 출력되는가?  3. 1~7개의 data가 입력 되었을 때 7개의 data에 대한 factorial 결과가 모두 출력되는가?  4. 0개의 data가 입력 되었을 때 factorial은 바로 interrupt를 1로 set되어 testbench에 출력되는가?  아래는 위의 4가지 case에 대한 검증 결과이다.   |  | | --- | | \*TESTBENCH CASE 1 🡪 입력 값 : 0, 1, 2, 3, 4, 5, 6, 7 총 8개의 수 | | 분석 : 모든 과정을 분석 해보면, 20000ps 부분에서 M0\_address가 11이 되고 M0\_dout값에 0,1,2,3,4,5,6,7,8을 순차적으로 값을 넣어주었다. 따라서 FIFOTOP\_IN의 FIFO가 선택 된 것이며, 그것에 write를 한다는 뜻이다. fifo\_cnt\_in과 fifo\_flag\_in에서 값이 출력되는 것을 볼 수 있다. 그리고 0부터 7까지 입력 될 때 마다 fifo\_cnt\_in 값이 1,2,3…차례대로 증가 하는 것을 볼 수 있다. 그리고 나서 M0\_address의 값을 0x01, 0x03으로 주고 M0\_dout을 1로 준 것을 볼 수 있다. 이것은 각각 FACTORIAL의 register에 ‘1’ 값을 써준 다는 의미이다. 그리고 나서 M0\_req를 0으로주면 2 cycle뒤에 M0\_grant가 0이 되는 것을 볼 수 있다. FACTORIAL이 grant를 가져 간 것을 의미한다. 그리고 나서 150000ps 부근에서 M\_din이 32인데, 이것은 FACTORIAL이 FIFOTOP\_IN의 FIFOFLAG register의 값을 요청해서 출력된 값이다. 32면 FULL을 의미하는데 현재 FIFOTOP\_IN은 8개 가득 찬 상태이므로 올바르게 나온 것을 확인 할 수 있다. 그리고 나서 잠시후 fifo\_cnt\_in이 7로 떨어진 것을 볼 수 있는데, 이것은 FACTORIAL이 FIFOTOP\_IN의 FIFO로부터 1개의 값을 read한 것을 의미한다. 그 후 FACTORIAL은 MULTIPLIER를 통하여 factorial 연산 후에 FIFOTOP\_OUT의 FIFOFLAG의 값을 요청하게 된다. 그것이 225000ps에서 M\_din으로 읽어진 것을 볼 수 있다. 16은 empty상태를 의미하므로, FIFOTOP\_OUT에 값을 써줄 수 있다. 그 후 fifo\_cnt\_out이 1, 2, 3, 4 차례대로 늘어난 것을 볼 수 있는데, 이것은 FACTORIAL이 FIFOTOP\_OUT의 FIFO에 4개의 값을 write했다는 것을 의미한다. 그 다음 300000ps 바로전에 M\_din값이 ‘1’인 것을 확인 할 수 있는데, 이것은 FIFOTOP\_IN의 FIFO에서 read된 값이 M\_din으로 들어온 것을 의미한다. 그리고 fifo\_cnt\_in은 또다시 6으로 1 떨어진 것을 볼 수 있다. 그럼 FACTORIAL은 ‘1’ 값에 대한 factorial 연산을 하게 된다. 이 과정이 6번 더 반복된다.    6번 더 반복 된 후의 testbench 캡쳐이다. 2450000ps 부근에서 fifo\_cnt\_out이 32까지 올라 가는 것을 볼 수 있으며, 이것은 총 8번의 factorial연산이 되어 FIFOTOP\_OUT의 FIFO에 32개의 data가 쓰여진 것을 의미한다. 그 후 바로 FACTORIAL은 interrupt 신호를 1로 set한다. 2485000ps 부근에서 interrupt가 ‘1’이 되는 것을 확인 할 수 있으며, 그 때 M0\_req가 1로 올라가서 testbench는 BUSgrant를 요구한다. 곧 바로 M0\_grant가 1로 set되는 것을 확인 할 수 있다. 그 후에 M0\_address를 0x02 즉 FACTORIAL의 Interrupt register에 0값을 써준 다는 뜻이다. 그 뒤 바로 M0\_address는 0x21이 입력되고 M0\_wr는 0이므로 FIFOTOP\_OUT의 FIFO에게 data read를 요청 하는 것이다. 바로 fifo\_cnt\_out이 1씩 감소하며 M\_din으로 data가 read되는 것을 확인 할 수 있다. fifo\_cnt\_out이 28일 때 ‘1’ 즉, 128’h1을 의미한다. 0의 연산 결과이므로 올바르게 나온 것을 확인 할 수 있다. 또 fifo\_cnt\_out이 24일 때 ‘1’ 이 출력된다. ‘1’의 factorial연산결과가 올바르게 나왔다. 그리고 fifo\_cnt\_out이 20일 때 ‘2’ 가 출력되었다. 2의 연산 결과가 올바르게 나왔다.  위 의 검증 전략 중 1.에서 0,1,2 의 결과가 올바르게 나온 것을 확인 하였고, 2.에서 8개의 data에 대한 결과가 모두 옳으므로 해당 검증전략이 성공하였다.    그 후로도 계속 보면 6, 24, 120 , 720 맨 마지막에는 5040이다. 즉 3!, 4!, 5!, 6!, 7!이 제대로 결과가 나온 것을 확인 할 수 있다. | | \*TESTBENCH CASE 2 🡪 입력 값 : 20, 19, 18, 17, 16, 15, 14, 13 총 8개의 큰 수 | | 위의 CASE1에서와 같이 8개의 data를 입력으로 넣어 주었다. fifo\_cnt\_in이 8까지 올라가면서 20, 19, 18, 17, 16, 15, 14, 13이 저장 되는 것을 확인 할 수 있다.    FIFO\_TOP\_OUT에서 저장된 data를 읽어오는 부분이다. 수가 너무 커서 M\_din을 16진수로 표현 하였다.  31, 30 , 29 ,28 부분에는 20!의 연산 결과가 M\_din으로 읽어진다.  20!= 2432902008176640000 이것을 계산기로 통하여 16진수로 바꾸어 보면 21C3677C\_82B40000이다. 위의 cnt가 31 30 29 28부분에서 읽어진 M\_din과 일치한다.  19!= 121645100408832000 이것을 계산기로 통하여 16진수로 바꾸어 보면 1B0\_2B93\_0689\_0000  위의 cnt가 27 26 25 24부분에서 읽어진 M\_din과 일치한다.  18!= 6402373705728000(10) 🡪 16BEECCA730000(16) cnt 21 20 부분의 결과랑 일치한다.    17! = 355687428096000(10) 🡪 1437EEECD8000(16) cnt가 17 16 부분과 일치한다.  16! = 20922789888000(10) 🡪130777758000(16) cnt가 13 12 부분과 일치한다.  15! = 1307674368000(10) 🡪 13077775800(16) cnt가 9 8 부분과 일치한다.    14! = 87178291200(10) 🡪 144C3B2800(16) cnt가 5 4 부분과 일치한다.  13! = 6227020800(10) 🡪 17328CC009(16) cnt가 1 0 부분과 일치한다.  따라서 검증전략 1.해당하는 0~20의 모든 수를 검증 하였고, 2. 또한 다시한번 검증 되었다. | | \*TESTBENCH CASE 3 🡪 testbench가 처음에 어떠한 데이터도 FIFOTOP\_IN에 써주지 않았을 경우. | | 노랑선 에서부터 보면, FIFOTOP\_IN에 어떠한 data도 보내지 않았는데, M0address를 0x01, M0\_din을 1, M0\_wr을 1, 로 주어 FACTORIAL의 Interrupt\_Enable에 1을 써주고 바로 M0address가 0x03이고 되어서 OperationStart register에 1을 써준 것을 확인 할 수 있다. 따라서 FACTORIAL의 연산이 시작된다. 2cycle뒤에 M0\_grant가 0으로 내려가는 것을 확인 할 수 있으며, 이는 FACTORIAL에게 grant가 주어졌다는 것을 의미한다. FACTORIAL은 원래대로 FIFOIN\_TOP의 FIFOFLAG를 check할 것이며, 그에 대한 결과가 M\_din에 16이 출력 되었다는 것으로 확인 할 수 있다. 이것은 FIFOFLAG가 010000이므로 EMPTY라는 것을 의미한다. 따라서 FACTORIAL은 바로 Interrupt를 발생 시킨다. 그 후로 testbench가 다시 grant를 받고 FACTORIAL의 Interrupt를 0으로 만들어준다. 그리고 나서 testbench가 끝난다.  이 testbench를 통해 검증 전략 4. 를 확인 할 수 있다. | | \*TESTBENCH CASE4 🡪 입력 값 : 4, 8, 6, 2 총 4개의 임의의 수 | | 4, 8, 6, 2 총 4개의 data가 FIFOIN\_TOP에 write된 것을 확인 할 수 있다.    fifo\_cnt\_out이 12인 지점에서 M\_din으로 24 🡪4!  fifo\_cnt\_out이 8인 지점에서 M\_din으로 40320 🡪8!  fifo\_cnt\_out이 4인 지점에서 M\_din으로 720 🡪6!  fifo\_cnt\_out이 0인 지점에서 M\_din으로 2🡪2! 이 출력된 것을 확인 할 수 있다.  위의 결과를 통해 검증전략 3. 을 확인 할 수 있다. |   위의 4가지 CASE를 통하여 검증전략을 모두 확인 하였다. |

|  |
| --- |
| **Ⅴ. 고찰** |
| **1. 고찰** |
| 이번 프로젝트를 진행하면서 개인적으로 FIFO\_TOP은 문젯거리가 아닐 거라고 생각했다. 왜냐하면 FACTORIAL module이 강조되고 모두들 FACTORIAL을 어려워 한 것처럼 나 또한 그랬다. 하지만, TOP module을 완성 한 후, 검증을 하였는데 문제점은 다른 module에서보다 FIFO\_TOP에서 많이 발생 하게 되었다. 그래서 예전 강의자료들을 다시 뒤져보면서 FIFO\_TOP에 대한 개념을 다시 검토했다. 그랬더니, 잘못된 부분이 한 두 군데가 아니었다. FACTORIAL module과 MULTIPLIER만을 중점적으로 생각하다 보니 FIFOTOP을 구현 할 때 신경 써야 할 점을 하나도 신경 쓰지 않은 것이다. 예를 들면 FIFOTOP안에 FIFO안에서 data를 내보내는 것이기 때문에 FIFOTOP으로 들어온 sel 혹은 address는 FIFO에서 나온 data보다 한 cycle 늦게 들어온 input이다. 그런데 그 한 cycle 늦게 들어온 input들로 FIFOTOP의 output을 결정하려고 하다 보니 당연히 문제점 이 생길 수 밖에 없었다. 그래서 예비 검증 때 FIFOTOP이 이렇게 잘못된 상태에서 FACTORIAL을 더 올바르게 고쳤는데도 오류가 늘어났던 것을 FIFOTOP을 수정 하고 나서야 알게 되었다. 비록 모든 오류를 전부 수정하진 못하였지만, 하드웨어가 어떻게 동작하는지 이번 프로젝트를 통해서 조금이나마 깨닫게 된 것 같고, SLAVE MASTER interface를 통해 data를 주고받는 방식을 알게 되었다. 그리고 Interrupt란 신호가 굉장히 중요한 것이란 것도 알게 되었다. 모든 하드웨어는 Interrupt 신호를 통해 대화가 가능 하다는 것을 알게 되었다. 이번 프로젝트에서 만족스러웠던 점은, MULTIPLIER를 구현 하는데 있어서 기존 보다 속도를 높이게 된 점이다. 처음에는 radix2로 하였지만, 속도가 좀 느린 거 같아서 radix4를 좀 연구해서 radix4를 구현했다. 그리고 FACTORIAL을 구현 한 뒤 시간이 좀 남아서 어떻게 하면 좀더 빠르게 작동하는 MULTIPLIER를 만들 수 있을 까 생각하다가, radix8같은 경우는 너무 area가 커질 것 같고 알고리즘도 굉장히 복잡할 것 같아서 너무 시간낭비라고 생각했다. 그러다 문득 든 생각이 factorial연산은 1~20 이기 때문에 multiplier를 1~20으로 고정 시킬 수 가 있다는 점 이었다. 그러면 multiplier가 20이 되더라고 64bit중 10100 이 5bit을 제외하고는 앞의 bit은 모두 0이라는 점이다. 따라서 이 앞의 부분을 먼저 읽고 모두 0이라는 것만 알면 shift 연산만 해주면 속도를 기존보다 엄청나게 빠르게 늘릴 수 있다는 점을 생각해 내게 되었다. 실제로 구현 해보니 그렇게 어렵지도 않았고, 곱셈 속도가 빨라지니 wave form을 확인할 때도 수월해서 좋았다. 구현 하는데 있어서 가장 어려웠던 점은 FACTORIAL이 FIFOTOP에게 data read를 요청 할 때 였다. 아무리 address를 보내도 FIFOTOP에서 데이터가 올바르게 오지 않았는데, 그것은 FIFO에서의 한 cycle 연산 때문이었다. 그것을 알게 되고, FACTORIAL에서 원래 데이터를 받아야할 state보다 한 state 이전에 address를 주었더니 올바르게 data가 전송된 것을 확인 할 수 있었다. |

**Ⅵ. Reference**

[1] 강의자료 – Memory mapped IO.pdf – 이준환 교수님

[2] 강의자료 - 08\_Multiplier.pdf – 최호석 조교