**Multiplier machine**

2013722095 최재은

**Abstract**

Multiplier machine은 외부에서 받아들인 data를 저장하고 이후에 이를 읽어낸 뒤 곱셈 연산을 진행하여 RAM에 저장하는 machine이다. 크게 data를 저장하는 RAM, 두 개의 값을 받아들여 이에 대한 곱셈연산을 진행하는 MULTIPLIER, 내부 데이터의 읽기/쓰기 및 연산 모듈의 동작 실행을 control하는 DMAC(Direct memory access ), 이들 간의 data / signal을 전송하는 BUS로 구성되어있다.

DMAC에게 start signal을 주면 memory에 저장된 값들을 설정된 opmode에 따라서 MULTIPLIER로 전송하는데, 이때 MULTIPLIER는 전송 받은 값들을 입력 받은 offset에 따라 해당되는 FIFO에 값들을 저장한다. 이후 DMAC의 interrupt signal이 발생되면 testbench는 BUS master가 되어 MULTIPLIER에 start signal을 준다. Start signal을 받은 MULTIPLIER는 FIFO에 저장되어있는 multiplicand와 multiplier를 pop하여 연산을 시작하는데, 이러한 동작은 FIFO가 empty가 될 때까지 진행된다. 연산을 마친 결과 값들은 MULTIPLIER 내부의 Register File에 저장하는데, 2cycle에 걸쳐서 하위 32bit, 상위 32bit 순으로 저장한다. 이때 FIFO 내부의 모든 값들에 대한 연산을 완료하면 opdone signal이 발생하며, 이 때의 offset에 따라 MULTIPLIER interrupt signal이 발생한다. MULTIPLIER interrupt signal이 발생하면 testbench는 다시 BUS master가 되어 MULTIPLIER 내부의 Register File에 저장된 값들을 RAM에 저장하기 위해 opmode를 설정하고, DMAC에게 start signal을 보내준다. DMAC는 설정된 opmode에 따라서 MULTIPLIER 내부의 Register File의 값들을 RAM으로 전송하고, 이 전송이 완료되면 DMAC interrupt signal을 발생시킨다.

BUS가 module간의 주고 받는 정보는 data와 read/write signal 뿐이므로, 미리 약속된 값들을 module 내부의 register 공간에 저장하고 읽어 들여야 한다. 그렇기 위해서 DMAC와 MULTIPLIER에 미리 약속된 register를 만들고 이들에 대한 입력은 offset address를 통해서 결정된다.

1. **Introduction**

이 프로젝트는 곱셈 연산을 하는 machine으로써 내부적으로 DMAC를 갖고 있다. 이를 통해서 대량의 data를 복사할 때, processor가 다른 동작할 수 있게 되어 system의 performance를 향상시킬 수 있다. 내부적으로 DMAC, RAM, MULTIPLIER, BUS module을 갖고 있으며, 이들에 대한 통제는 외부에서 들어오는 DMAC control signal과 DMAC를 통해서 수행된다. 외부에서 입력된 data를 ram에 저장하고 이를 DMAC를 통해 읽어내어 MULTIPLIER로 입력해주고, MULTIPLIER는 입력된 값들을 그 내부에 저장하고 있다가 start signal에 따라서 그들에 대한 연산을 시작한다. 이렇게 연산이 된 값들은 MULTIPLIER 내부의 Register File에 저장이 되며, 그 값들은 다시 DMAC의 통제에 따라서 RAM에 저장이 된다.

1. **Project Specification**
2. DMAC(Direct Memory access)

* Memory와 i/o device 간의 data 전송을 제어하는 모듈.
* BUS의 slave interface를 통해서 제어된다.
* Data의 이동/복사를 위해 필요한 정보인 source address, destination address, data size를 묶어 하나의 descriptor로 판단한다.
* 각각의 offset이 할당된 register를 통해서 제어할 수 있다.
* BUS로부터 S\_sel, S\_wr, S\_address, S\_din, DMAC\_master로부터 opdone을 받아 들여 offset에 따라 내부의 register에 값을 저장하고 그 값들을 출력하며, FIFO에 descriptor와 write enable을 넘겨주는 DMAC\_slave, DMAC\_slave로부터의 descriptor와 write enable을 저장하기 위한 FIFO, 자체적인 state를 가지고 DMAC\_slave로부터 받은 opstart, opdone\_claer signal을 통해서 FIFO에서 descriptor와 write\_enable을 받아 그 값들을 증감하여 master interface output으로 출력하는 DMAC\_master로 구성되어있다.
* DMAC\_slave는 S\_sel, S\_wr, opdone, S\_din[0], offset에 따라서 각 register에 값을 저장한다. 그에 대한 설명은 아래와 같다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| S\_sel | S\_wr | offset | Opdone | S\_din[0] | operation |
| 1 | 1 | 0x0 | 0 | 1 | OPERATION\_START[0] <= 1 |
| 1 | 1 | 0x1 | x | 1 | OPERATION\_CLEAR[0] <= 1 |
| 1 | 1 | 0x2 | x | x | INTERRUPT\_ENABLE[0] <= 1 |
| 1 | 1 | 0x3 | x | 1 | PUSH\_DESCRIPTOR[0] <= 1 |
| 1 | 1 | 0x4 | x | x | SOURCE\_ADDRESS <= S\_din[7:0] |
| 1 | 1 | 0x5 | x | x | DESTINATION\_ADDRESS <= S\_din[7:0] |
| 1 | 1 | 0x6 | x | x | DATA\_SIZE <= S\_din[7:0] |
| 1 | 1 | 0x7 | x | x | DESCRIPTOR\_SIZE[3:0] <= S\_din[3:0] |
| 1 | 1 | 0x8 | x | x | OPMODE[2:0] <= S\_din[2:0] |

\* 이외의 case에 대하여 offset이 0x0~0x3의 경우는 0을 입력하고, 0x4~0x8의 경우는

기존의 값을 유지하도록 한다.

* S\_sel = 1, S\_wr = 0이 되면 offset에 따라서 각 Register에 저장된 값들을 출력해준다.
* DMAC\_master State의 천이에 관한 부분은 Design Details에서 설명하겠다.
* State와 opmode에 따라서 source, destination address를 증감하는데 이에 대한 설명은 아래와 같다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| State | Opmode | source\_address | destination\_address | Data\_size |
| FIFO\_POP | X | Retain | Retain | Retain |
| BUS\_REQUEST | X | Retain | Retain | Retain |
| MEMORY\_READ | X | Retain | Retain | -1 |
| MEMORY\_WRITE | 1x1 | +1 | 0 | Retain |
| 1x0 | Retain | 0 | Retain |
| 000 | Retain | retain | Retain |
| 001 | +1 | Retain | Retain |
| 010 | Retain | +1 | Retain |
| 011 | +1 | +1 | Retain |

\* default case는 현재의 값들을 그대로 유지한다.

* Read enable은 FIFO\_POP state일 때만 1이 된다.
* State에 따라서 master interface output, opdone signal을 출력하는데 이는 아래와 같다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| State | M\_req | M\_wr | M\_address | M\_dout | opdone |
| IDLE | 0 | 0 | 0 | 0 | 0 |
| FIFO\_POP | 0 | 0 | 0 | 0 | 0 |
| BUS\_REQUEST | 1 | 0 | 0 | 0 | 0 |
| MEMORY\_READ | 1 | 0 | Source\_address | 0 | 0 |
| MEMORY\_WRITE | 1 | 1 | Destination\_address | M\_din | 0 |
| DONE | 0 | 0 | 0 | 0 | 1 |

\* default case는 모든 output을 0으로 출력한다.

1. MULTIPLIER

* 입력 받은 multiplicand, multiplier를 저장하는 2개의 FIFO, 결과를 저장할 Register File, interrupt signal과 offset에 따른 값들을 저장하는 register를 통해서 내부 signal을 조절하며 S\_dout의 값, interrupt signal을 출력하는 mul\_slave, 입력 값들을 실제로 곱셈 연산하는 mul\_master로 구성되어있다.
* 입력으로 S\_sel, S\_wr, S\_din, S\_din을 받으며, 출력 값으로 interrupt와 S\_dout을 갖는다.
* S\_sel = 1, S\_wr = 1일 때에 FIFO, Register File에 쓰고 읽는 동작을 수행한다.
* S\_wr = 1, S\_din[0] = 1이고 S\_address의 하위 4bit인 offset 값이 0x4이면 2개의 FIFO에서 POP하여 mul\_master에게 입력으로 주고, 동시에 opstart signal을 1로 입력한다.
* Mul\_master는 IDLE, POP, EXEC, WRITE\_B, WRITE\_F, DONE의 state를 갖는다.

EXEC state에서 연산이 완료되어 WRITE\_B state가 되면 wr\_en을 1로 출력하고 결과 값의 하위 32bit, write address를 출력한 뒤 WRITE\_F state로 넘어간다. WRITE\_F state 또한 wr\_en을 1로 출력하고 결과 값의 상위 32bit, write address를 출력한 뒤 POP state로 넘어간다. POP state에서는 입력 받은 fifo empty signal을 통해서 DONE state로 넘어가거나 EXEC state로 넘어간다. State가 DONE이 되면 opdone signal을 출력한다.

* 내부적으로 MULTIPLICAND, MULTIPLIER register를 두어 입력 받은 피연산자, 연산자를 저장할 수 있도록 설계하였다.
* Mul\_master에서 입력 받은 multiplicand와 multiplier를 곱셈 연산할 때에 radix-2 booth’s multiplication algorithm을 사용하였다. 그에 대해 Design Details에서 설명할 예정이다.
* FIFO에 저장되어있던 모든 multiplicand, multiplier에 대한 곱셈 연산이 완료되면 mul\_master는 opdone signal을 출력한다.
* Mul\_slave는 S\_sel, S\_wr이 1일 때 offset에 내부의 register에 알맞은 값을 저장하며,

S\_wr = 0일 때 mul\_master의 state를 입력으로 받아서 FIFO에 write enable signal과 저장할 값을 넘겨준다. 또한 mul\_master로부터 read address를 받아 Register File에 저장된 값을 RESULT register에 저장한다.

* Offset address가 0x5이면 OPERATION\_CLEAR register의 0번째 비트에 1이 기록되며, mul\_master로 opclear signal이 들어가게 된다.

1. BUS

* 각 sub module 간의 data와 signal을 전송하는 역할을 수행한다.
* 주고 받는 Data의 bandwidth는 32bit이며, address의 bandwidth는 8bit이다.
* 내부의 arbiter module이 request에 따라서 두 개의 master중 하나에게 우선권을 준다. 이 우선권에 대한 사항은 Design Details에서 설명한다.
* Master의 역할을 모두 수행하는 DMAC, MUTIPLIER와 slave로 3개의 RAM을 갖는다.
* Slave에 따라서 각각 다른 memory map region address를 갖는다. 각각의 영역은 다음과 같다.

|  |  |
| --- | --- |
| Name | Memory map region address |
| Slave 0 | 0x00 ~ 0x0F |
| Slave 1 | 0x10 ~ 0x1F |
| Slave 2 | 0x20 ~ 0x3F |
| Slave 3 | 0x40 ~ 0x5F |
| Slave 4 | 0x60 ~ 0x7F |

* Address decoder를 통해서 입력 받은 address를 5bit의 selector로 변환하는데 이는 one-hot encoding 방식을 이용하여 5개의 selector를 control할 수 있게 한다. 이때의 selector는 5개의 submodule(dmac, multiplier, ram(3))에 사용된다.

1. RAM(Random Access Memory)

* 초기에 내부의 모든 값들은 0으로 초기화되어 있다.
* Input value로 cen, wen, data, address를 가지며 output value로 입력된 address에 저장된 값을 출력한다.
* Chip enable과 Write enable이 모두 1인 경우에만 입력된 address에 data를 저장하며,

Chip enable이 1, Write enable이 0인 경우에는 입력된 address에 저장되어 있는 data를 output으로 출력한다.

1. **Design Details**

각 component 별 pin description, block diagram, FSM 기재

1. DMAC

* **Pin description**

|  |  |  |
| --- | --- | --- |
| Direction | Port name | Description |
| Input | clk | Clock |
| reset\_n | Active low reset |
| M\_grant | Grant -> bus에서 data를 주고 받는 것을 허락해주는 signal. 해당 signal이 1이면 bus를 통해 다른 slave component를 제어할 수 있다. |
| M\_din[31:0] | Data input |
| S\_sel | Select |
| S\_wr | Write / Read |
| S\_address[7:0] | Address->하위 4bit를 offset으로 사용하여 register를 구분함. |
| S\_din[31:0] | Data input |
| output | M\_req | Request -> bus를 통해 data를 주고 받을 수 있도록 요청하는 signal |
| M\_wr | Write / Read |
| M\_address[7:0] | Address |
| M\_dout[31:0] | Data output |
| S\_dout[31:0] | Data output |
| interrupt | Interrupt -> data의 이동/복사가 끝나면 내부 register값에 따라서 발생함. |

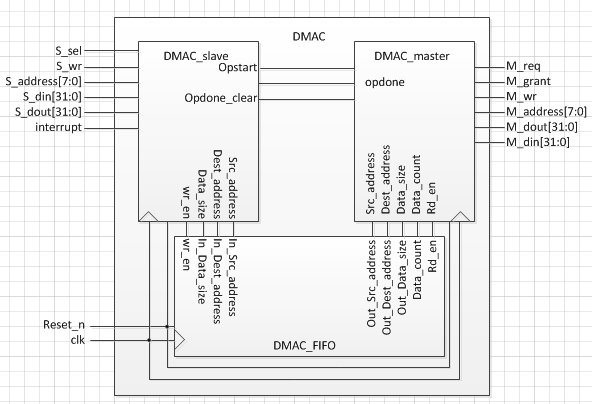
\* M -> master interface / S -> slave interface

* **Register Description**

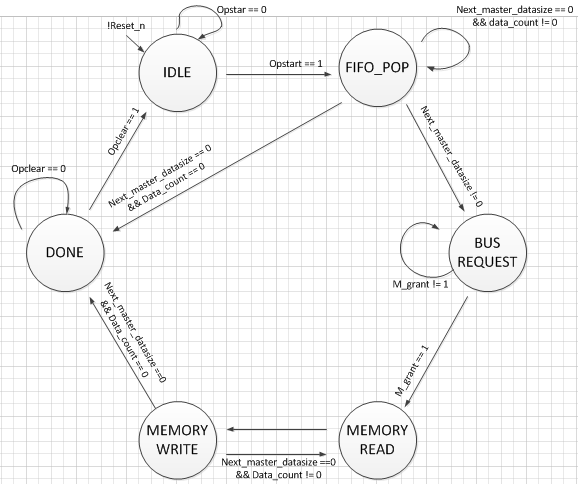
|  |  |  |  |
| --- | --- | --- | --- |
| Offset | Type | Name | Description |
| 0x0 | W | OPERATION START | [0]에 1이 써지면 DATA의 이동/복사를 시작함. DMAC가 동작 중이거나 OPERATION DONE register의 [0]이 1이면 해당 register의 [0]에 1을 쓰지 않는다. |
| 0x1 | W | OPERATION CLEAR | [0]에 1이 써지면 모든 register의 값이 default value가 된다. |
| 0x2 | R/W | INTERRUPT ENABLE | [0]가 1이면 OPERATION DONE register가 1이 되었을 때 interrupt port에서 1이 출력.  0이 되면 OPERATION DONE register가 1이 되었을 때 interrupt port에서 0이 출력. |
| 0x3 | W | PUSH DESCRIPTOR | [0]에 1이 써지면 SOURCE ADDRESS, DESTINATION ADDRESS , DATA SIZE register의 값을 내부의 FIFO에 저장한다. 최대 8개의 descriptor를 저장할 수 있다. |
| 0x4 | R/W | SOURCE ADDRESS | Data의 이동/복사를 할 때 read를 시작할 address를 저장함. |
| 0x5 | R/W | DESTINATION ADDRESS | Data의 이동/복사를 할 때 write를 시작할 address를 저장함. |
| 0x6 | R/W | DATA SIZE | Data의 이동/복사를 할 때 전송할 data의 크기를 저장한다. 단위는 byte. |
| 0x7 | R | DESCRIPTOR SIZE | FIFO에 저장되어 있는 descriptor의 수를 나타냄. 최대 8개를 저장하므로 하위 4bit만 사용한다. |
| 0x8 | R/W | OPERATION MODE | Zero initial기능  Source address, destination address의 increment mode가 있음.  Opmode[0] : source address increment  Opmode[1] : destination address increment  Opmode[2] : zero initialize |

\* 위의 register들은 값이 유동적인 bit를 제외하고는 모두 reseved임.

* **Block diagram**



* **FSM**( DMAC\_master )



2. MULTIPLIER

* **Pin description**

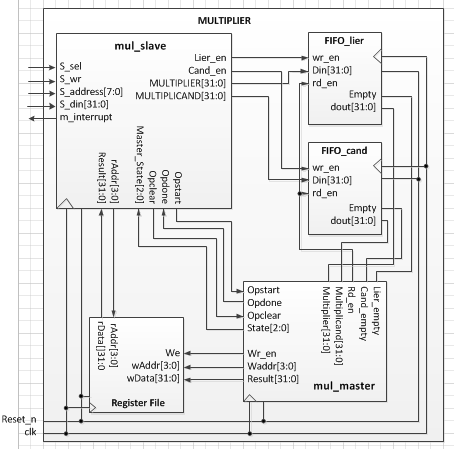
|  |  |  |
| --- | --- | --- |
| Direction | Port name | Description |
| Input | clk | Clock |
| reset\_n | Active low reset |
| S\_sel | Select |
| S\_wr | Write / Read |
| S\_address[7:0] | Address ->하위 4bit을 offset으로 사용하여 register를 구분함. |
| S\_din[31:0] | Data input |
| Output | S\_dout[31:0] | Data output |
| m\_interrupt | Interrupt -> 연산이 완료되면 내부 register의 값에 따라서 발생함. |

* **Register Description**

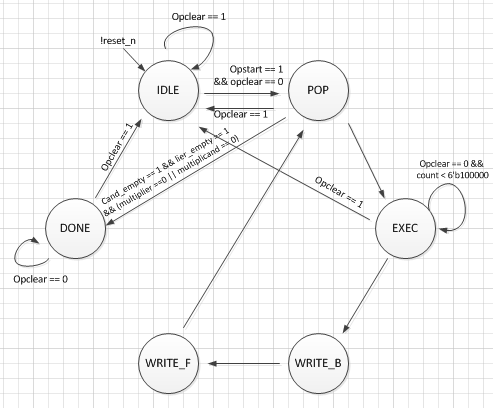
|  |  |  |  |
| --- | --- | --- | --- |
| Offset | Type | Name | Description |
| 0x0 | R/W | MULTIPLICAND | DMAC로부터 전달받은 multiplicand를 저장 |
| 0x1 | R/W | MULTIPLIER | DMAC로부터 전달받은 multiplier를 저장 |
| 0x2 | R | RESULT | RF에 저장 되어 있는 곱셈연산의 결과 값을 DMAC를 통해 2cycle에 걸쳐 RAM에 저장할 때 사용 |
| 0x3 | W | INTERRUPT  ENABLE | [0]이 1이면 MULTIPLIER의 OPERATION DONE이 1일 때, interrupt port에서 1이 출력  [0]이 0이면 0이 출력됨 |
| 0x4 | W | OPEARTION  START | [0]에 1이 써지면 FIFO에서 multiplicnad와 multiplier를 POP하고 opstart를 1로 출력하여 mul\_master에 넘겨준다. 곱셈연산중에 [0]에 1이 써지면 해당 값은 무시됨. |
| 0x5 | W | OPERATIOL CLEAR | [0]에 1이 써지면 모든 register의 값이 default value가 됨. |
| 0x6 | R | OPERATION  DONE | 해당 register의 값을 read할 때 [0]이 1이면 MULTIPLIER가 곱셈연산을 완료하였음을 나타낸다. |

\* 위의 register들은 값이 유동적인 bit를 제외하고는 모두 reseved임.

* **Block diagram**



* **FSM (mul\_master)**

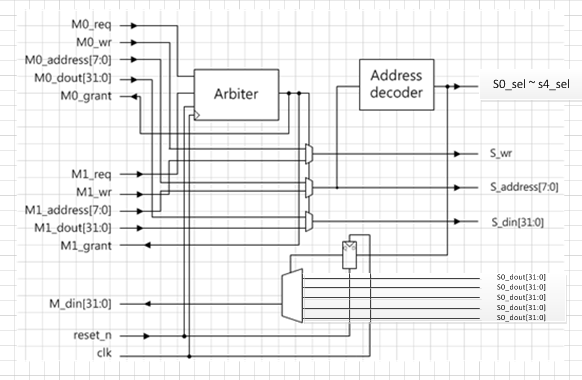


3. BUS

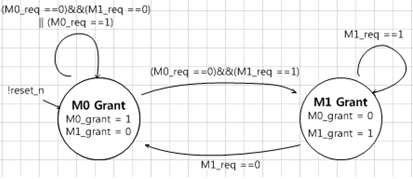
* **Pin description**

|  |  |  |
| --- | --- | --- |
| Direction | Port name | Description |
| Input | clk | Clock |
| Reset\_n | Active low reset |
| M0\_req | Master 0 request |
| M0\_wr | Master 0 write/read |
| M0\_address[7:0] | Master 0 address |
| M0\_dout[31:0] | Master 0 data output |
| M1\_req | Master 1 request |
| M1\_wr | Master 1 write/read |
| M1\_address[7:0] | Master 1 address |
| M1\_dout[31:0] | Master 1 data output |
| S0\_dout[31:0] | Slave 0 data out |
| S1\_dout[31:0] | Slave 1 data out |
| S2\_dout[31:0] | Slave 2 data out |
| S3\_dout[31:0] | Slave 3 data out |
| S4\_dout[31:0] | Slave 4 data out |
| Output | M0\_grant | Master 0 grant |
| M1\_grant | Master 1 grant |
| M\_din[31:0] | Master data input |
| S0\_sel | Slave 0 select |
| S1\_sel | Slave 1 select |
| S2\_sel | Slave 2 select |
| S3\_sel | Slave 3 select |
| S4\_sel | Slave 4 select |
| S\_address[7:0] | Slave address |
| S\_wr | Slave write/read |
| S\_din[31:0] | Slave data input |

* **Block diagram**



* **FSM**



4. RAM

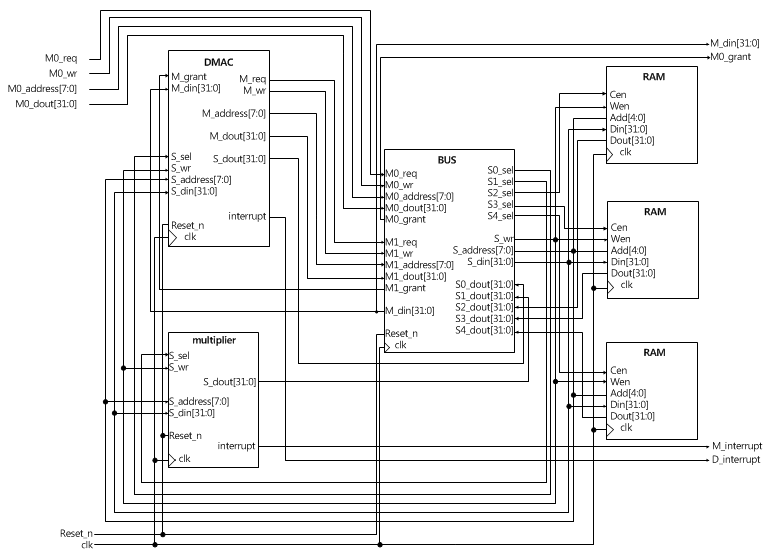
* **Pin description**

|  |  |  |
| --- | --- | --- |
| Direction | Port name | Description |
| Input | clk | Clock |
| Cen | Chip enable |
| Wen | Write enable |
| addr[4:0] | address |
| din[31:0] | Data in |
| Output | dout[31:0] | Data out |

5. TOP

* **Pin description**

|  |  |  |
| --- | --- | --- |
| Direction | Port name | Description |
| Input | clk | Clock |
| Reset\_n | Active low reset |
| M0\_req | Master request |
| M0\_wr | Master write/read |
| M0\_address[7:0] | Master address |
| M0\_dout[31:0] | Master data output |
| Output | M0\_grant | Master 0 grant |
| m\_interrupt | Multiplier interrupt |
| d\_interrupt | Dmac interrupt |
| M\_din[31:0] | Master data input |

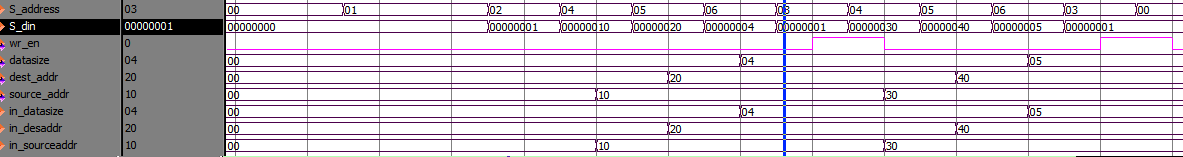
* **Block diagram**

1. **Design Verification Strategy and Results**

1. DMAC

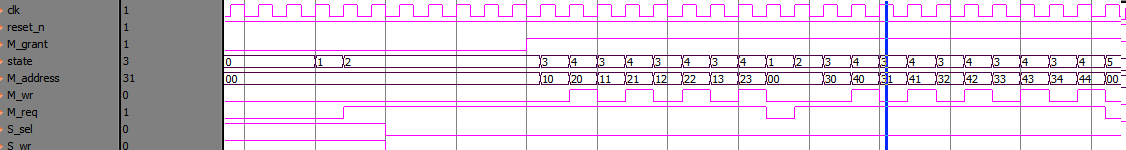
- M\_grant, S\_sel, S\_wr에 따라 S\_din이 src\_addr, dest\_addr, datasize에 잘 들어가는지, 또 이것이

fifo에 잘 넘어가는지를 확인



-> 아래의 in\_으로 시작하는 값들을 보면 slave에서 출력된 값들이 fifo에 잘 들어감을 확인할 수 있음.

-S\_sel, S\_wr, M\_grant와 state에 따른 M\_address의 변화를 확인.



->S\_sel, S\_wr = 0이고 M\_grant가 1이 되면서 state가 3, 4를 반복하며 해당되는 M\_address가 바뀌는 것을 확인 할 수 있음.

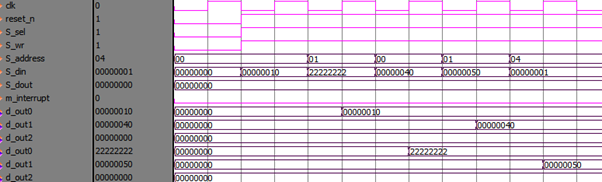
- state, INTERRUPT\_ENABLE[0]을 통해 interrupt가 발생하는지를 확인



-> INTERRUPT\_ENABLE[0]이 1이고, state가 5로 넘어가면서 interrupt output이 1로 rising하는 것을 확인할 수 있음.

2. MULTIPLIER

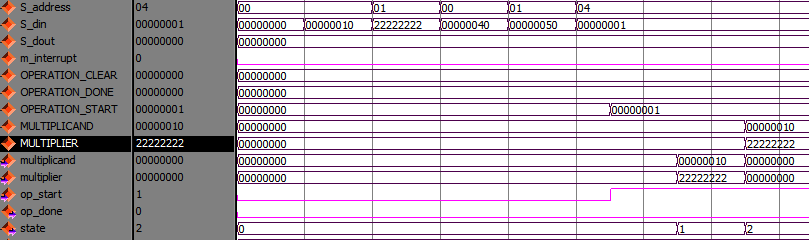
- 여러 개의 multiplicand, multiplier를 주고 이들이 각각의 FIFO에 잘 저장이 되는지 확인



-> offset이 0일때의 S\_din이 FIFO\_multiplicand에 저장되는 것을 확인할 수 있음.

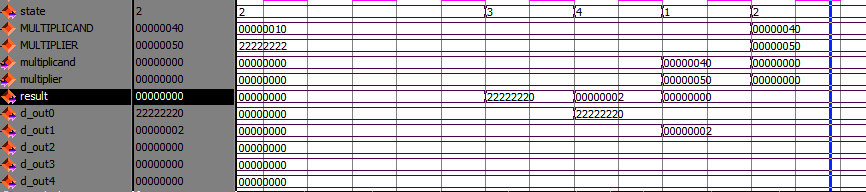
-> offset이 1일때의 S\_din이 FIFO\_multiplier에 저장되는 것을 확인 할 수 있음.

- offset에 따라 동작이 잘 수행되는지 확인



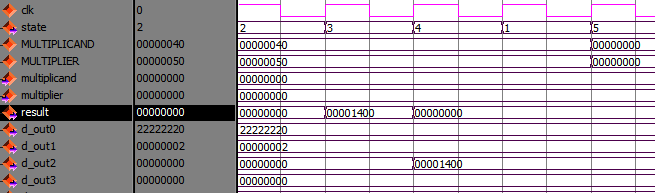
-> offset이 4가 되면서 OPERATION\_START register의 [0]에 1이 써짐

-> master의 state가 0(IDLE)에서 1(POP)으로 넘어가 multiplicand와 multiplier를 받아오고, 2(EXEC)가 되면서 연산을 시작하는 것을 확인할 수 있음.

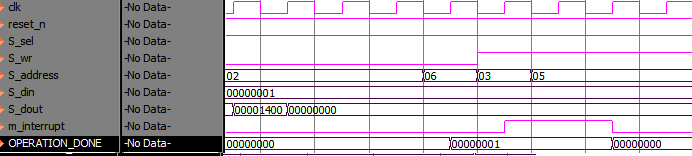


-> 연산이 완료되자 state가 3(WRITE\_B)이 되어 result로 하위 32bit을 내보내고 그것을 RF에 저장하는 것을 확인할 수 있음. 그 다음 state가 4(WRITE\_F)가 되어 result로 상위 32bit을 내보내고 그것을 RF에 저장하는 것을 확인할 수 있음.

-> 그 다음 state가 1(POP)이 되어 값들을 받아오고 2(EXEC)로 넘어가 연산을 시작함을 확인할 수 있음.



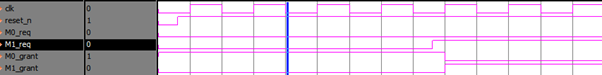
-> 새로 받은 값에 대한 연산이 끝나고 state가 넘어가면서 위에서와 동일하게 RF에 값을 저장하고 POP state로 넘어가지만 empty이므로 5(DONE)으로 넘어감.



-> offset 이 0x6이 되면서 OPERATION\_DONE[0]에 1이 쓰이고 offset이 3으로 넘어가면서 m\_interrupt이 발생하는데 이후 offset이 5가 되면서 register가 0으로 초기화 된다.

3. BUS

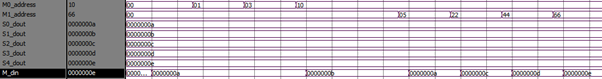
- request에 따른 grant의 변화를 확인한다.



- master의 address에 따른 selector의 변화와 S\_address, S\_din의 출력을 확인한다.

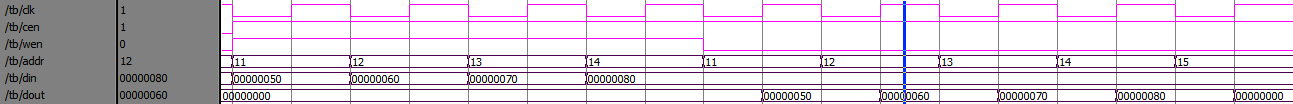


- master의 address에 따른 M\_din output의 출력을 확인한다.



4. RAM

- cen, wen에 따라서 값이 제대로 저장이 되는지, 저장된 값이 제대로 출력이 되는지를 확인.

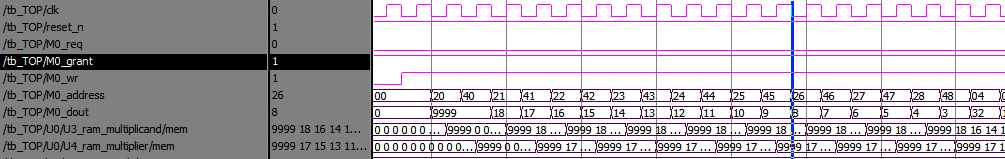


- 저장한 값들이 실제 메모리 공간에 저장이 됐는지를 확인.



5. TOP

- grant와 wr signal을 통해서 m\_dout이 잘 저장되는가.



-> 위에서 확인하는 바와 같이 해당 address region에 따라서 값이 fifo에 저장되고 있다. 아래의 사진은 multiplicand와 multiplier의 최종적인 저장 값



- testbench에서 입력해준 multiplicand, multiplier 값이 ram에 실제로 저장이 되는가



-> ram\_multiplicand



->ram\_multiplier

\* MULTIPLIER의 Register File에 연산 결과는 잘 저장이 되었으나 이를 불러오는 과정에서 DMAC가 req를 올리지 못해 값을 불러오지 못하여 ram\_result에 대한 결과는 작성하지 않았습니다.

1. **Conclusion**

이번 프로젝트를 진행하면서 최고의 난관은 역시나 MULTIPLIER를 짜면서 내부에서 이동하는 data의 클럭을 맞추는 것이 아니었나 싶습니다. RESULT 같은 경우에는 offset을 변경하면서 바로 두 클럭만에 결과들이 나와야 하는데 Register file에 enable signal을 주고 다시 값을 받아오면서 한 클럭을 대기하고 두 번째 클럭부터 값이 나오기 시작해서 이를 해결하기 위해서 next\_ 변수를 입력으로 주어 해결했습니다. 그리고 실질적인 연산을 하는 mul\_master에 multiplicand와 multiplier가 들어가고 나서 이것이 저장되지 않았었지만, 이것은 내부적으로 새로운 register(MULTIPLICAND, MULTIPLIER)를 선언하여 값을 넘겨줌으로써 해결하였습니다. 처음에는 master의 opdone signal에 대하여 일전의 과제에서처럼 하나의 쌍에 대한 연산이 끝나면 opdone이 발생하도록 하여 fifo에서 값을 받아오도록 하였으나, fifo에 있는 모든 쌍에 대한 연산이 끝나고 나야 opdone이 발생하고 이것을 OPERATION\_DONE[0]에 저장해야 한다는 부분을 읽고 나서 slave의 큰 부분을 수정해야 했습니다. 또한 offset 별로 값이 저장돼야 하는데 값을 저장하고 있지 않고 초기화하는 등 여러 가지 문제점이 발생했지만 하나하나 control signal과 state를 확인해 가면서 풀어나갔습니다. 가장 어이 없었던 것은 master의 state가 넘어가지 않아서 한 시간 정도 코드를 들여다보다 if / else if로 인해 if에서의 조건에 해당하지 않는 것으로 인해 else if case가 돌아가지 않았던 것이었습니다. 그 이외에도 각각의 submodule에서 testbench를 돌렸을 때는 잘 되는 듯싶더니 이를 TOP에서 instance하여 연결하니까 완전하게 작동되지 않는 경우도 있었습니다. 아직도 DMAC에서 request가 앞부분에서는 잘 되면서 RF에 저장된 값을 읽어오는 때는 req가 발생하지 않는 것이 해결되지 않았습니다. 그로 인해 연산 결과를 불러와 ram\_result에 저장하지 못하였습니다. 도대체 무엇이 문제인지 의문입니다. 그래서 인지 DMAC 부분에서는 점수가 그다지 좋지 못했습니다. 이외의 ram\_multiplicand, ram\_multiplier에 대한 값의 저장은 잘 동작하는 것으로 확인되었습니다. 12-01에 있었던 실제 검증에서는 Multiplier\_Top의 interrupt를 m\_interrupt로 하지 않아 처음에 error가 났었고 이를 수정하고 나서는 정상적으로 작동하였습니다.

이번 프로젝트는 지금까지 했던 과제들을 총 집합하여 이들을 instance하는 방식으로 진행되었습니다. 지금까지 구현했었던 과제들은 warning에 대하여 중요한 사항이 아니면 염두에 두지 않아도 되었으나, 이번에 대부분의 warning을 잡아내면서 그 동안 작성해왔던 코드들이 얼마나 불안정한 방식이었는가를 다시 한번 깨닫는 순간이었습니다. 그 수 많던 latch들이 실제로 칩을 만드는 단계가 되면 다 추가적인 비용이 되는 것이므로 이들을 모두 없애는 것이 중요하다고 생각되었습니다. if 문에 제대로 된 조건을 넣지 않아 생겼던 경고들 또한 불안정한 동작을 야기할 수 있기에 이들을 잡아내는 것 역시 중요한 작업 중 하나였습니다. 또한 always 문을 사용하는 것에 대해서, 특히나 clock을 염두에 두면서 각 value들의 값들을 조정하는 것에 대해서 정말 많은 공부가 되었다고 생각합니다. 이전에는 그냥 ‘이렇게 하는 거랬지…’하면서 사용했던 것이라면, 지금은 마치 visual studio에서 f10을 눌러가며 한 줄씩 디버깅을 하듯이 클럭을 보면서 ‘이때는 이 값이 이렇게 뛰고, 이 값이 뛰었으니까 다음 클럭에서는 이 값이 뛰어야 한다’고 생각할 수 있게 되었습니다. 이러한 점을 통해 프로그램의 진행을 sequential하게 생각할 수 있는 능력을 배양하는 좋은 기회가 되었다고 생각합니다.

1. **Reference**

[1] Wikipedia / https://ko.wikipedia.org

[2] 2016\_ProjectSpecification\_v11

[3] google / https:// www.google.com