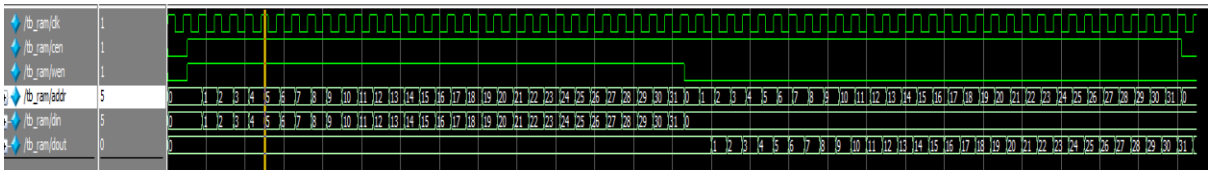
<Design Verification Strategy and Results>

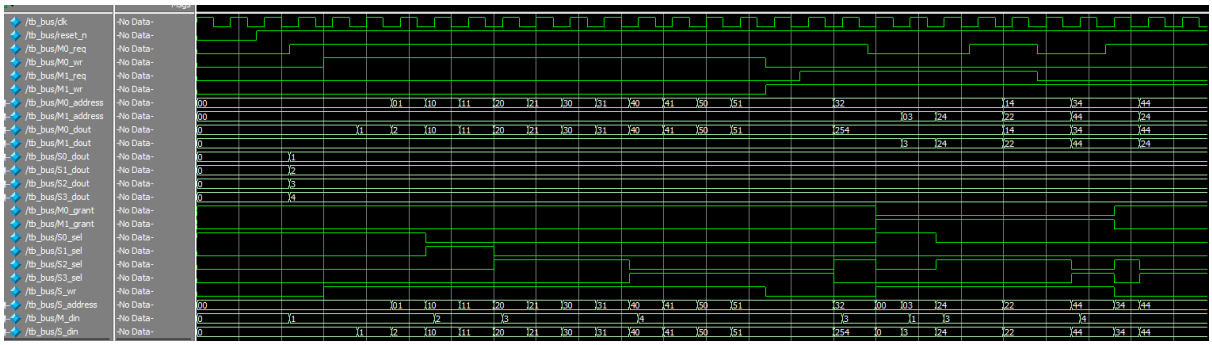
* TOP Project에 있는 RAM, BUS, DMAC_Top, Factorial_Top, multiplier를 위주로 검증을 하였다. 그러기 위해 TOP module외의 각 module의 내부 reg, wire type 변수들을 port로 설정하여 TOP Project의 외부에서 각 module별로 project를 생성하여 검증하였으니 소스 파일 확인 시 이 점 유의바랍니다. 물론 TOP module의 검증은 TOP Project 내부에서 이루어진다.

<RAM>



RAM 같은 경우는 이전에 실습 시간에 구현하였던 ram.v 파일을 그대로 쓰면 되었다. 그래도 정확성을 높이기 위해 검증을 다시 한번 하였다. 일단 cen이 1로 되면, 이는 read/write signal에 따라 ram을 사용할 수 있다는 것을 의미한다. 처음에는 cen과 wen을 둘 다 1로 함으로써, ram을 write mode로 사용할 수 있게 하였다. 1번지에는 1, 2번지에는 2, … 31번지에는 31을 저장하였다. 다 저장한 뒤에 wen을 0으로 함으로써, ram을 read mode로 사용할 수 있게 하였다. 이 때부터 주어진 address에 저장되어 있던 값들을 정확하게 dout으로 출력하는 모습을 볼 수 있다.

<BUS>



Bus도 이전 실습 시간에 구현하였던 파일을 쓰면 되지만, Design detail에서 말하였듯이 M1 grant 상태에서 M0_req가 0인데 M1_req가 0이 되었다고 해서 M0 grant로 가는 sequential logic을 수정하여 M1_req만 단순히 0이 될 때는 계속 M1 grant를 유지하는지 확인해보기 위해 검증을 자세하게 하였다.

Reset을 해주고 난 이후부터 값을 살펴보자. 일단 M_din의 출력을 제대로 확인하기 위해 S0_dout=1, S1_dout=2, S2_dout=3, S3_dout=4로 설정하였다. 그 뒤에 M0_req와 M0_wr을 차례대로 1로 올려주었다. 그렇게 되면 M0 grant로 유지가 되면서, S_wr이 1이 되어야 할 것이다. 확인해보면 S_wr의 값이 잘 변화하였다. M0 grant임을 확실히 하기 위하여 M0_dout에 값을 차례대로 써주었다. 그 와중에 Slave selection이 잘 변화하는지 확인하기 위해 M0_address의 값도 변화를

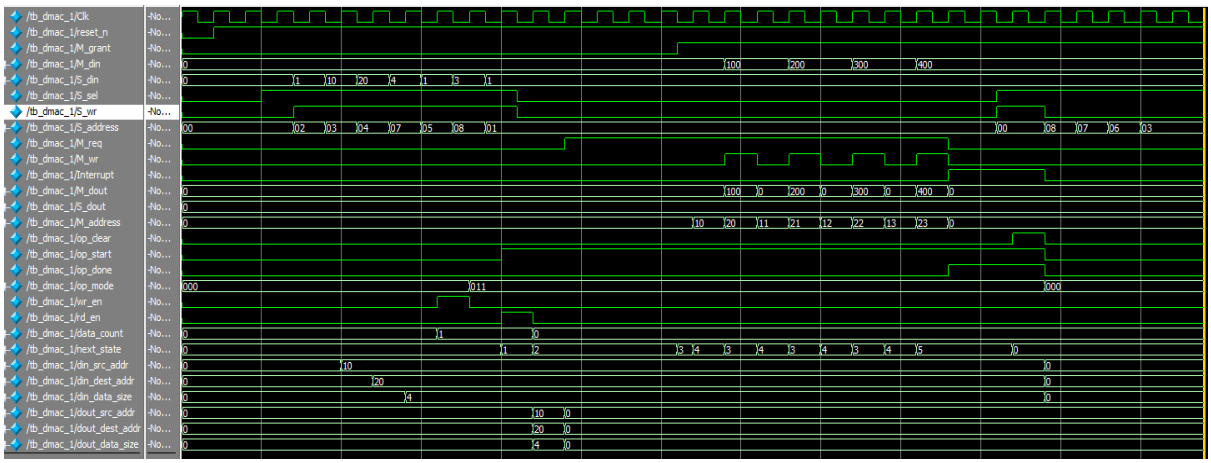
차례대로 주었다. 처음엔 계속 S0_sel이 1로 되어있었다. M0_address가 계속 0이었기 때문에 00~0F번지인 Slave 0가 select되기 때문이다. 그리고 M0_address가 10~1F번지일 때는 Slave 1이 잘 select 되고 있다. 그 뒤로도 20~3F번지일 때는 Slave 2가 select 되고 있고, 40~5F번지일 때는 Slave 3가 select되고 있는 것을 확인할 수 있다. 이로써 bus의 address decoder가 잘 구현되어있음을 확인할 수 있다. 그와 동시에 S_din에서는 M0_dout의 값이 잘 나오고 있고, M_din에서도 Slave selection에 맞게 결과값이 잘 나오고 있다.

이제 bus arbiter가 잘 작동하는지 확인하기 위해, 중간에 M0 grant상태에서 M1_req의 값을 1로 주었다. 하지만 이 때 M0_req가 1인 상태이므로 M1 grant로 이동하지 않는 것을 확인할 수 있고, 예시로 M0_address와 M0_dout의 값을 변경하였을 때 여전히 S_address와 S_din에서는 Master 0의 값을 출력하는 것을 확인할 수 있다. 그 다음에는 M0_req를 0으로 내림으로써, M1 grant가 되게 하였다. 예상대로 M1 grant로 잘 변화하였고 이제부터는 M1_address와 M1_dout에 따라 S_address와 S_din의 결과값이 출력되는 것을 확인할 수 있다.

마지막으로 M1_grant상태에서 M1_req가 0으로 내려갔을 때를 확인해보자. Waveform을 보면 이 때 M0_req도 0인데, 이 때는 bus arbiter에서 그대로 M1 grant를 유지하도록 해야 한다. 예상대로 M1 grant가 잘 유지되고 있고, 그 다음에 다시 M0_req를 1로 올렸는데 이 때는 정상적으로 M0 grant가 잘 인가되어 Master 0의 address와 dout이 S_address와 S_din으로 잘 출력되는 것을 볼 수 있다. 따라서 bus arbiter도 잘 구현되어 있는 것을 확인할 수 있다.

<DMAC>

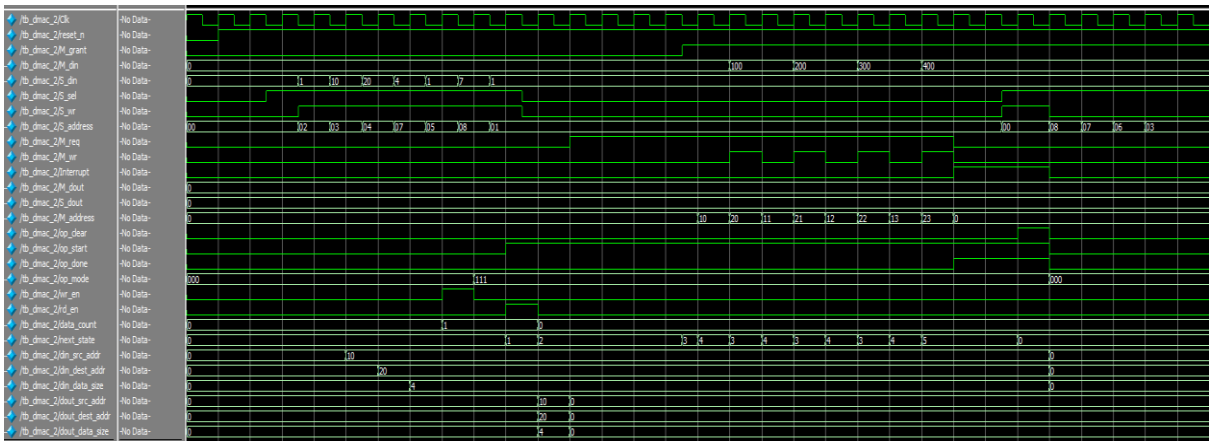
1) tb_dmac_1(op_mode=3'b011, single data input)



다음 testbench는 op_mode를 3'b011로 작동시키고 data push를 한번만 했을 때이다. Op_mode를 011로 한 것은 Source increment mode와 Destination increment mode의 작동이 잘 되는지 한번에 관찰하기 위함이다. S_sel과 S_wr이 모두 1이 되는 순간 dmac의 내부 register의 값을 쓰기 시작한다. Interrupt_en에 1을 넣고, source address에는 10, destination address에는 20, data size에는 4를 넣고 push descriptor에 1을 넣음으로써 descriptor를 fifo에 push하였다. 아래의 data_count가

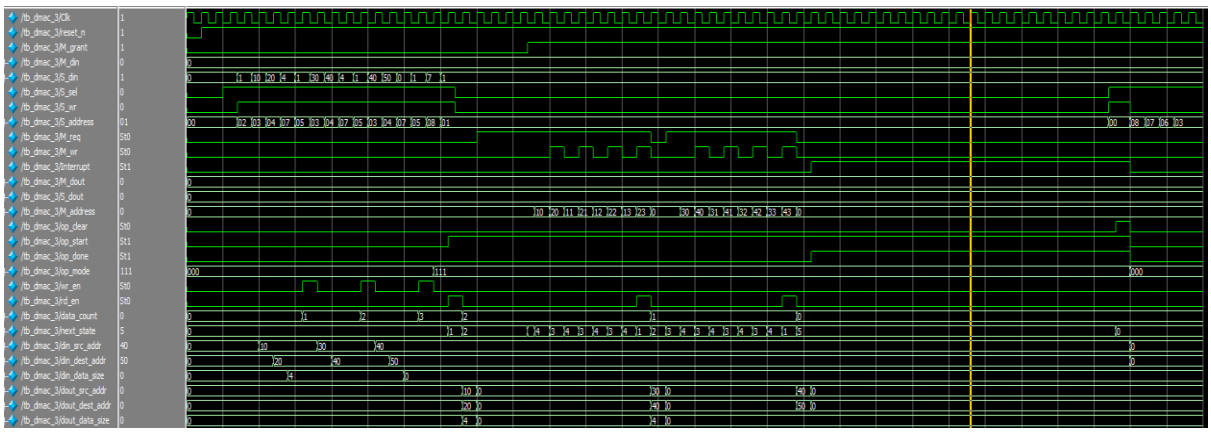
1이 된 것이 그 결과이다. 그 뒤로 op_mode를 011로 선언하고 op_start를 1로 하여 dmac의 작동을 시작하였다. Op_start가 되고 난 후, state는 FIFO_POP이 되어 descriptor가 fifo로부터 pop 된 것을 waveform에서 확인할 수 있고, 한 cycle 뒤에 BUS_REQ state가 되면서, M_req가 켜진 것을 볼 수 있다. Testbench상으로 M_grant를 1로 해준 후, MEM_READ state가 되면서 M_address로 source address를 잘 출력하고 있다. 그 다음에는 MEM_WRITE state로 가면서 M_wr이 켜지고 들어오는 M_din의 값을 M_dout으로 destination addres와 함께 잘 출력해준다. 이 과정을 data_size의 크기만큼 반복하고 DONE state로 넘어가면서 interrupt가 잘 나오는 것을 확인할 수 있다. Interrupt가 뜬 다음 S_wr을 1로 켜주고 op_clear register(0x00)에 1을 넣었다. 따라서 IDLE state로 넘어가면서 interrupt가 0으로 내려가고 op_mode, data size, descriptor size, source address와 같은 register의 값을 읽어도 전부 0이 나오는 것을 확인할 수 있다.

2) tb_dmac_2(zero initialize)



다음 testbench는 tb_dmac_1과 거의 비슷한데, op_mode[2]의 값을 1로 선언하여 zero initialize 모드로 켰을 때이다. 위와 같이 M_din의 값이 M_wr일 때 들어옴에도 불구하고, tb_dmac_1과 달리 M_dout의 값은 0으로 나오면서 destination address가 증가하는 것을 확인할 수 있다. 한마디로 destination address의 값을 0으로 초기화 하는 것이다.

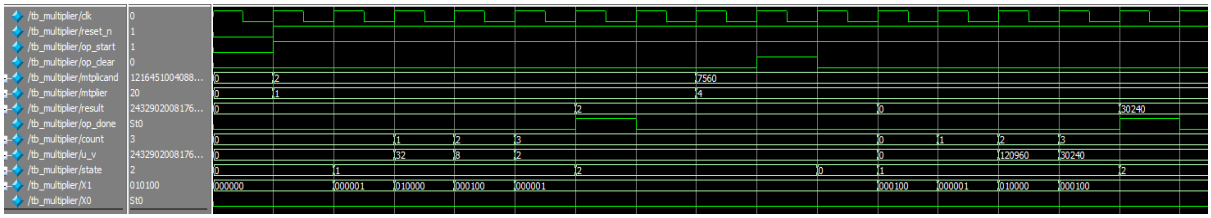
3) tb_dmac_3(multiple data input)



다음 testbench는 descriptor를 여러 개 넣었을 때이다. 처음에 Interrupt_en register에 1을 넣어주고 source address를 10, destination address를 20, data size를 4로 넣고 fifo에 push하였다. Tb_dmac_1에서는 이후 op_start를 해주었지만, 여기서는 그 다음 source address를 30, destination address를 40, data size를 4로 넣고 또 push하였다. 마지막으로 source address를 40, destination address를 50, data size를 0으로도 넣고 push하여 총 3개의 descriptor를 fifo에 push 하였다. Op_mode는 3'b111(zero initialize mode)로 설정하고 op_start signal을 주었다. 처음에는 tb_dmac_1과 같이 op_mode가 잘 작동하면서 MEM_READ와 MEM_WRITE가 잘 진행된다. 주의깊게 봐야하는 것은 그 다음이다. Data count가 0이 아니므로 첫번째 descriptor를 전부 write 한 후 FIFO_POP state로 넘어가는 것을 확인할 수 있다. 따라서 data count가 1로 되고 다음 descriptor를 pop 해오는 것을 내부 wire인 dout_src_addr, dout_dest_addr, dout_data_size에서 확인할 수 있다. 그 뒤로는 전과 같이 BUS_REQ state를 거쳐 다음 descriptor의 source address인 30, destination address인 40에 op_mode에 따라 MEM_READ와 MEM_WRITE를 반복한다. 이 동작이 다 끝나면 fifo로부터 마지막 남은 descriptor를 pop해오는 것을 볼 수 있는데 data count가 0이고 data size가 0이므로 FIFO_POP state를 거쳐 바로 DONE state로 가는 것을 확인할 수 있다. 마지막으로 op_clear 동작이 잘 동작하는 것도 확인할 수 있다.

<Multiplier>

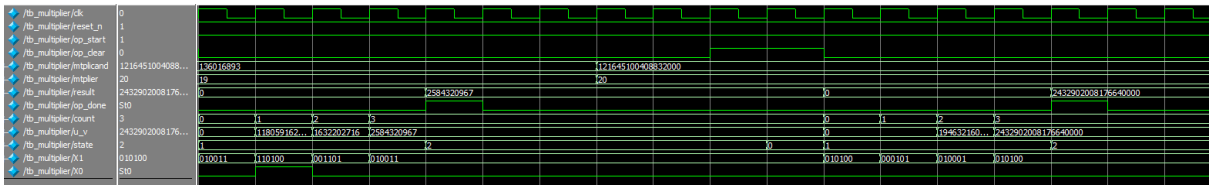
1) tb_multiplier (little number execution)



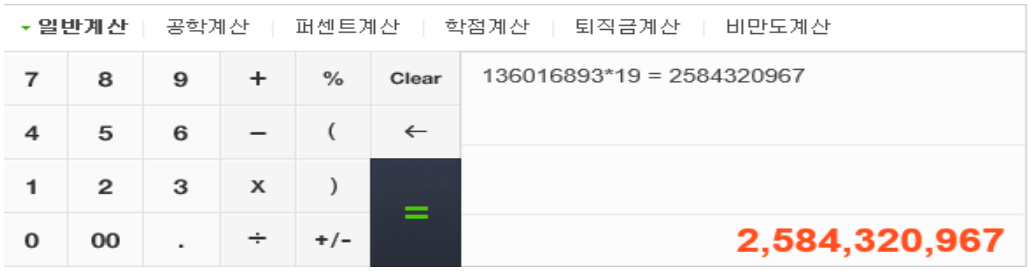
다음 waveform은 tb_multiplier에서 크기가 작은 숫자들의 곱셈을 검증해본 것이다. 내가 구현한 multiplier는 project용 multiplier로, multiplier는 무조건 1~20의 값이어야 한다. Multiplicand의 크기는 상관이 없다. 먼저 2X1을 보자. Multiplier가 1이므로, X1은 000001로 잘 초기화 되고 X0가 0이므로 radix-4에서의 pattern인 010에 따라 +Multiplicand(2)후 shift right twice가 되어 u_v의 값이 32가 되는 것을 알 수 있다. 그 뒤로는 pattern이 계속 000이므로 shift right twice만 2번 연속된다. 마지막으로 op_done이 1로 켜지면서 result값이 잘 나오는 것을 볼 수 있다. Operation 동작 횟수인 count도 3이 되었을 때 곱셈이 잘 마무리 되는 것을 알 수 있다.

다음 곱셈인 7560X4의 경우를 보자. 처음에 X1이 000100으로 잘 초기화 되고 X0도 0으로 되어 있다. Pattern이 000이므로 처음에는 shift left twice만 한다. 그 뒤에는 X1이 000001이고 X0가 0이므로 pattern이 010이다. 따라서 +Multiplicand(7560)후 shift right twice를 하게 되는데 이는 u_v에서 120960의 값으로 표현된다. 그 다음 pattern은 000이므로 shift right twice를 하게 되면 결과값인 30240이 나오게 된다. 두 번의 계산 모두 결과 값이 제대로 나오는 것을 확인할 수 있다.

2) tb_multiplier (high number execution)



Little number execution을 검증할 때 X1과 X0, u_v의 변화를 관찰 했으므로 이번 high number execution에서는 결과 값을 위주로 체크해보자. 이번 팩토리얼에서는 20!까지 가능해야 하므로 multiplier에 20까지의 숫자가 최대로 들어갈 수 있다. 136016893X19를 했을 때 결과는 25843209670이 계산기 상으로 맞는 값이다.



(출처: 네이버 계산기)

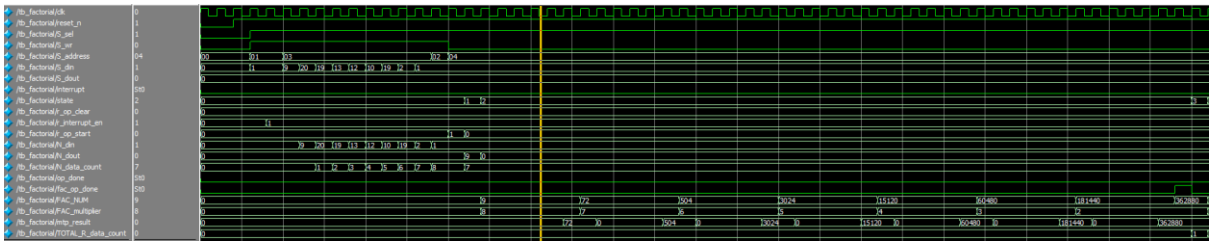
위와 같은 결과 값이 잘 나오는 것을 확인할 수 있다. 다음으로, 121645100408832000X20인데, 이 121645100408832000는 19!의 값이다. 한마디로 이 수에 20을 곱해봄으로써 20!의 결과가 잘 나올 수 있는지 확인해보기 위함이다.

(출처: Google 계산기)

수가 너무 커서 상위 숫자만 Google 계산기상으로 나오지만 일단 이 상위 비트들은 전부 계산 결과와 일치하는 것을 알 수 있다. 인터넷에서 조금만 찾아봐도 나오는 20!의 값과 세부적으로도 전부 일치하는 값이 나오는 것을 알 수 있다. 결론적으로 20!의 값까지 계산이 가능하고 출력까지 가능하다는 것을 알 수 있다.

<Factorial_Top>

1) Data input, first execution (9!)

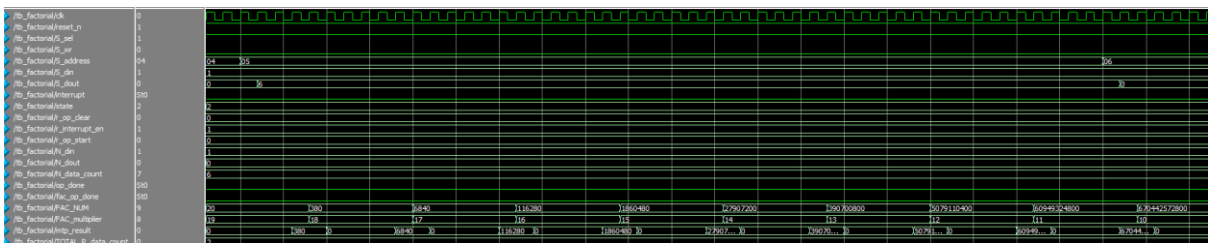


처음에 Interrupt_en register에 1을 넣고, N_FIFO에 순서대로 9, 20, 19, 13, 12, 10, 19, 2, 1을 넣어보았다. 그러나 아래의 N_data_count에서 8이 되고 난 후 더 이상 올라가지 않았기 때문에 마지막에 들어간 1은 포함되지 않았다는 것을 알 수 있다. 그 뒤에는 op_start signal을 1로 하여 factorial 연산을 시작한다. 처음에 op_start가 켜지면서 N_FIFO 로부터 9가 pop되고 N_data_count가 7로 감소하는 것을 확인할 수 있다. 9가 pop되고 나서 내부 register인 FAC_NUM과 FAC_multiplier가 각각 9와 8로 update되고 계산이 이루어져 72의 결과가 처음에 나온다. 72의 결과가 나온 뒤에는 이 값이 FAC_NUM으로 update되고, FAC_multiplier는 7로 update가 되는 것을 확인할 수 있다. 이 과정을 FAC_multiplier의 값이 2가 될 때까지 잘 반복하는 것을 알 수 있다. 마지막 연산이 끝나게 되면서 factorial 하나의 연산이 끝났음을 알려주는 fac_op_done signal이 1로 켜지며 최종 결과 값으로 FAC_NUM이 update된다. 이제 최종 FAC_NUM의 값과 정확한 연산 결과를 비교해보자.

(출처: Google 계산기)

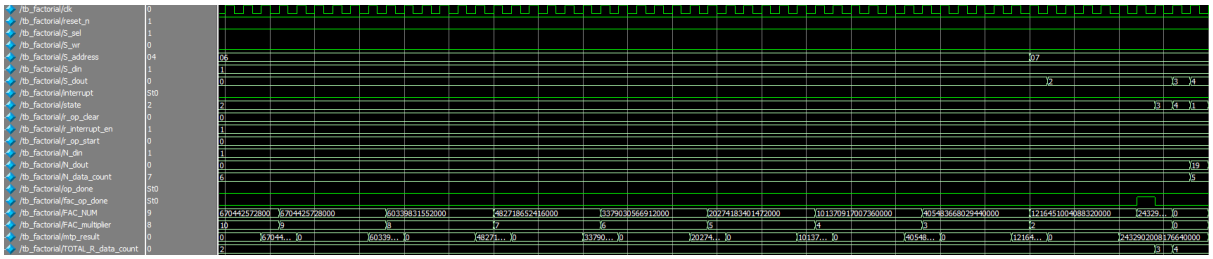
팩토리얼의 연산결과가 정확하다는 것을 확인할 수 있다. 저 Waveform 상으로는 잘 보이지 않지만 결과가 나온 뒤에 R_FIFO에 두 cycle에 걸쳐 결과값을 저장한다. 이에 관해서는 뒤에서 보이겠다.

2) Second execution (20!)



이번엔 20!의 연산을 확인해볼 것이다. 연산 과정은 위에서 확인해 봤으므로 다시 하지는 않겠다.

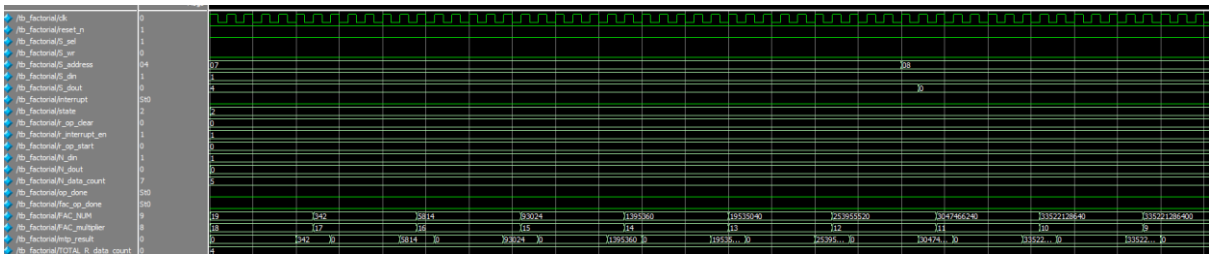
처음에 N_FIFO 로부터 20을 pop하여 FAC_NUM과 FAC_multiplier를 20과 19로 update하고 연산을 시작한다. 중간중간 Register의 값을 읽어보는 부분에 주의해보자. 초반에 S_address를 05로 바꿔주고 이 register의 값을 read 하였다. 05번지의 register는 N_FIFO의 data count이다. 지금 상태는 9!의 연산이 끝나고 20을 pop해온 상태이므로 N_FIFO의 data count는 6인 상태이다. 그에 맞게 S_dout 값이 잘 출력되는 것을 알 수 있다. 후반부에서는 06번지를 읽고있다. 06번지는 N_FIFO의 flag의 값을 표현한다. 지금 N_FIFO는 empty, full, rd_err, wr_err 그 무엇도 만족하지 않으므로 전부 0이다. 따라서 S_dout에서 0의 값이 잘 나오고 있다.



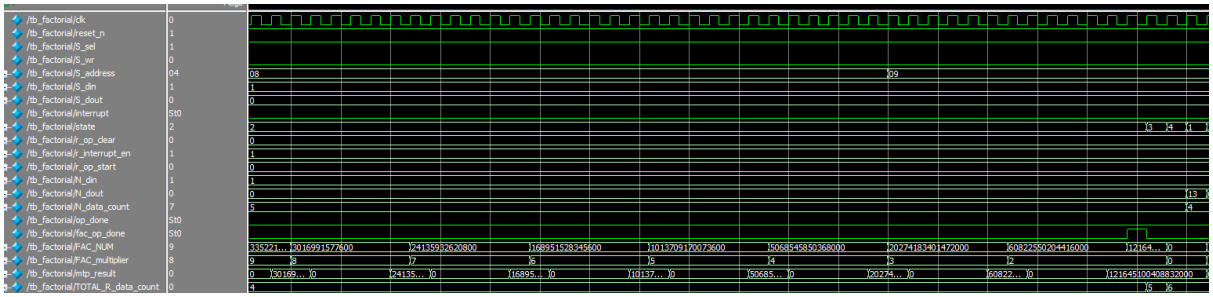
20!의 연산이 끝나기 전, 07번지의 register를 read하였다. 07번지는 R_FIFO의 data count이다. 따라서 앞에서 연산을 마친 9!의 값이 두 개의 data로 나뉘어서 R_FIFO에 결과값이 2개 저장이 되어 있다는 것을 확인할 수 있다. 마지막으로 연산의 결과가 2432902008176640000이 나오는데, 이 값은 20!의 값과 일치하는 것을 위의 multiplier 검증에서 확인하였기 때문에 정확한 값이 나오는 것을 알 수 있다.

연산이 끝나고 R_FIFO의 data count를 나타내는 TOTAL_R_data_count의 값이 두 cycle에 걸쳐서 1씩 증가하는 것을 알 수 있다. 나온 연산 결과값이 두 개의 data로 나뉘어서 R_FIFO에 한 cycle에 한 개씩 저장이 되는 것이다.

3) Third execution (19!)

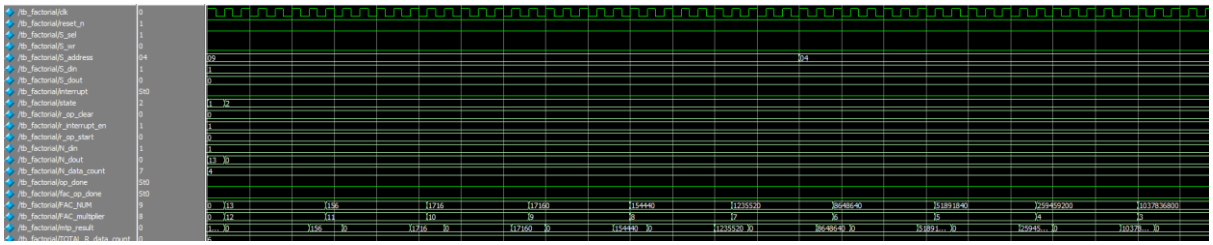


처음에 N_FIFO 로부터 19를 pop 하여 FAC_NUM과 FAC_multiplier를 19와 18로 update하고 연산을 시작한다. 여전히 S_address가 07번지일 때 값을 read하고 있다. 07번지는 R_FIFO의 data count를 의미하므로 9!, 20!의 결과 값인 총 4개의 data가 있어야 하고, S_dout에서 그에 맞는 값을 적절히 출력하고 있다. 중간에 08번지의 register의 값을 read한다. 08번지는 R_FIFO의 flag register인데, 지금 R_FIFO는 empty, full, rd_err, wr_err 모두 충족하지 않으므로 전부 0이다. S_dout에서 그에 맞는 값을 잘 출력하고 있다. 연산 과정 분석은 생략한다.

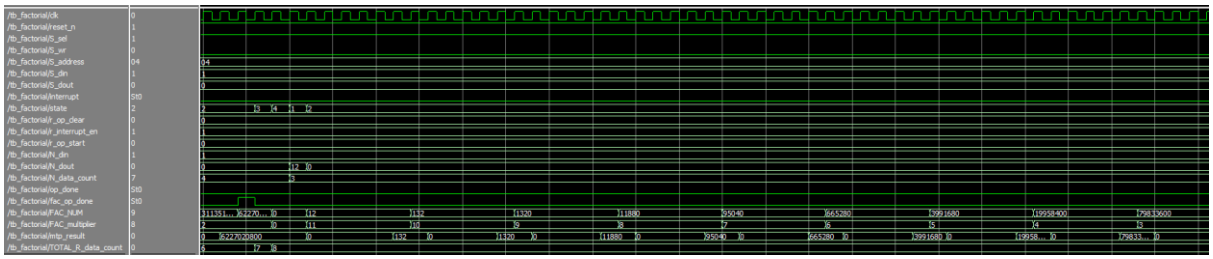


연산이 끝나기 전에 09번지의 값을 read하는데 09번지는 op_done register이다. 아직 남은 factorial 연산이 많기 때문에 절대 op_done은 1이 될 수가 없다. 따라서 0이 나와야하고, S_dout에서도 0이 제대로 출력되고 있다. 연산이 끝나면서 Fac_op_done이 1로 켜지고, 결과값은 121645100408832000이 나온다. 이는 19!의 결과 값과 일치하는 것을 위의 multiplier 검증에서 확인하였다. 그러므로 정확한 연산 값이 나온 것을 알 수 있다. 연산결과가 나온 후, R_FIFO에 2 cycle에 걸쳐 결과 값이 저장되어 TOTAL_R_data_count가 6이 되는 것을 확인할 수 있다.

4) Fourth execution (13!, 12!)



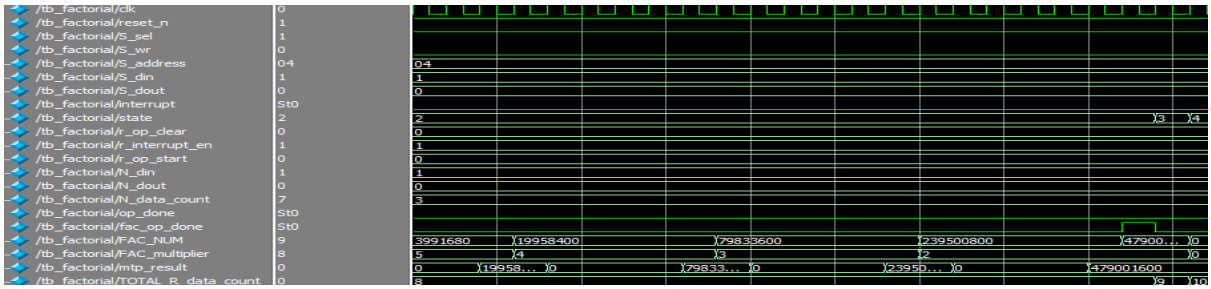
13!, 12!을 연산하는 모습이다. 먼저 13을 N_FIFO로부터 pop해온 후, FAC_NUM과 FAC_multiplier를 13과 12로 update하고 연산을 시작한다. 중간에 04번지를 read하도록 하였는데, 04번지는 R_FIFO를 가르킨다. R_FIFO는 MEM_WRITE state에서만 읽을 수 있으므로 지금은 아무것도 읽지 않아야 한다. 그에 맞게 S_dout은 0으로 출력되는 것을 확인할 수 있다.



13!의 연산이 끝나면서 fac_op_done이 1로 켜지고 결과값이 6227020800으로 나온다.

(출처: Google 계산기)

위와 같이 결과 값이 정확히 나오는 것을 확인할 수 있고, 2 cycle에 걸쳐서 R_FIFO에 결과값이 저장되는 것을 확인할 수 있다. 결과값을 저장한 직후 바로 N_FIFO로부터 12를 pop해서 FAC_NUM, FAC_multiplier를 12와 11로 update한 후 연산을 시작하는 것을 볼 수 있다.



12!의 연산이 끝나고 fac_op_done이 1로 켜지고, 결과값이 479001600으로 나오는 것을 알 수 있다.

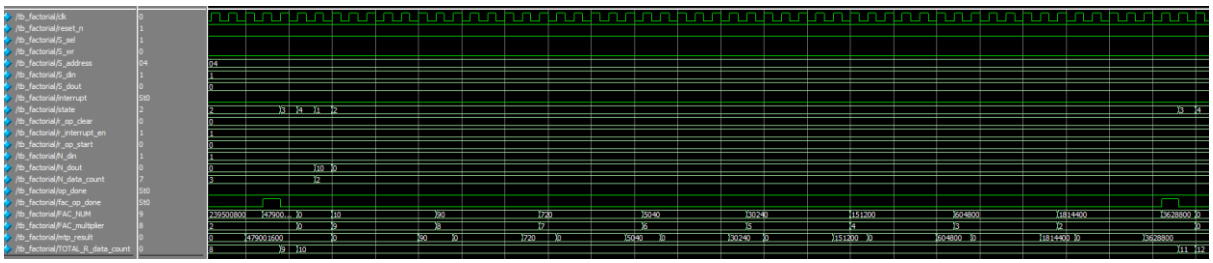
12 팩토리얼 =

479001600

(출처: Google 계산기)

위와 같이 결과 값이 정확히 나오는 것을 확인할 수 있고, 2 cycle에 걸쳐서 R_FIFO에 결과값이 저장되는 것을 확인할 수 있다.

4) Fifth execution (10!, 2!)

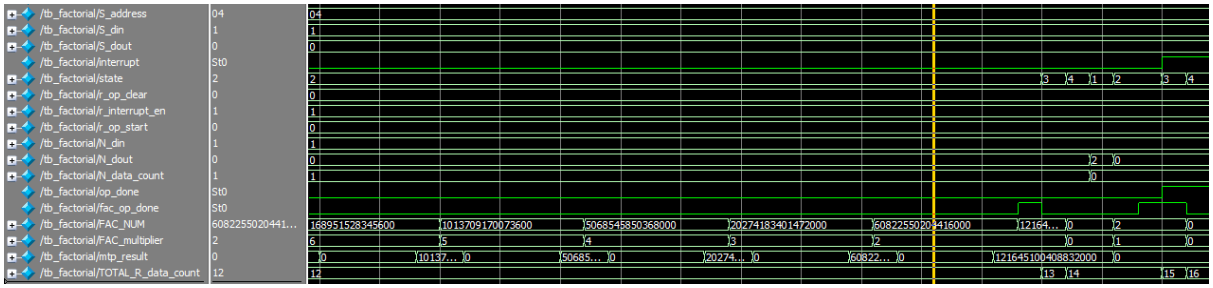


12!의 연산이 끝난 후 N_FIFO로부터 10을 pop하고 FAC_NUM, FAC_multiplier를 10과 9로 update하고 연산을 시작한다.

10 팩토리얼 =

3628800

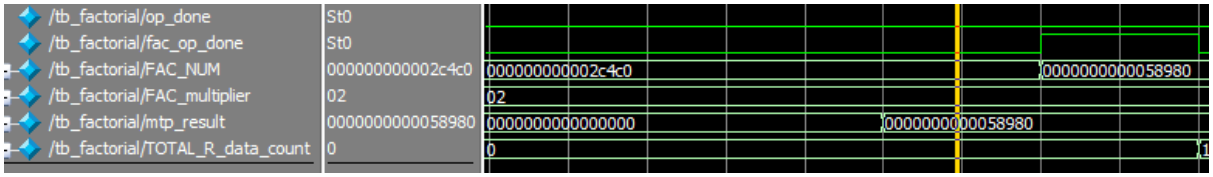
10!의 결과로 3628800이 나오는데 이는 정확한 결과값이다. 결과가 나온 후, 2 cycle에 걸쳐 결과값을 R_FIFO에 저장하여 R_FIFO의 data count가 12가 되는 것을 볼 수 있다.



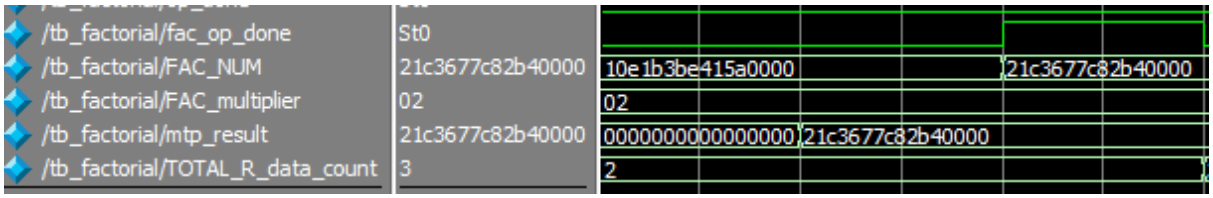
남은 N_FIFO의 값들 중 19는 아까 검증했기 때문에 생략하고, 2!일 때를 보자. 2를 pop한 후 FAC_NUM과 FAC_multiplier를 2와 1로 update한다. 그러나 FAC_NUM이 2일 때는 연산을 아예 하지 않고 바로 2의 결과를 도출하도록 설계하였기 때문에 바로 fac_op_done이 1로 켜지는 것을 볼 수 있다. 이 연산이 끝나면 N_FIFO의 data count가 0이므로 더 이상 pop 해올 수가 없기 때문에 더 이상 연산을 하지 않는다. 마지막까지 결과를 R_FIFO에 저장하면, 모든 연산이 끝나게 되면서 op_done 신호가 1로 바뀌게 되는 것을 확인할 수 있다.

5) 각 FACTORIAL의 결과값을 16진수로 계산.

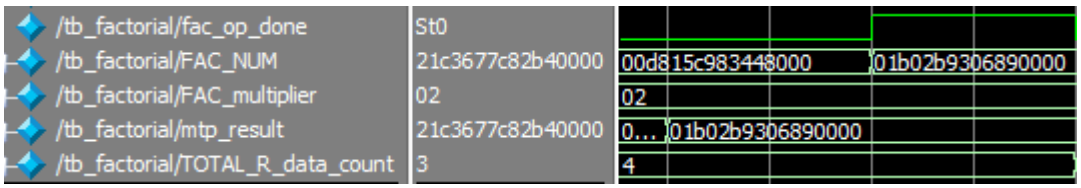
1. 9! = 00000000 | 00058980



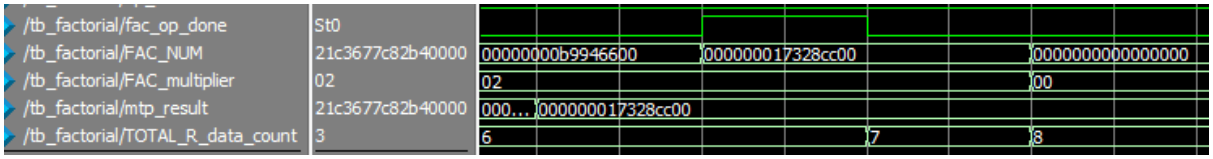
2. 20! = 21c3677c | 82b40000



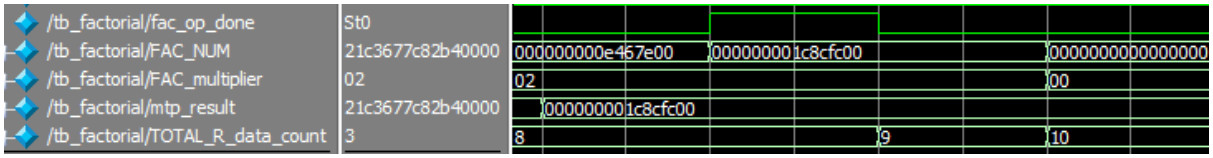
3. 19! = 01b02b93 | 06890000



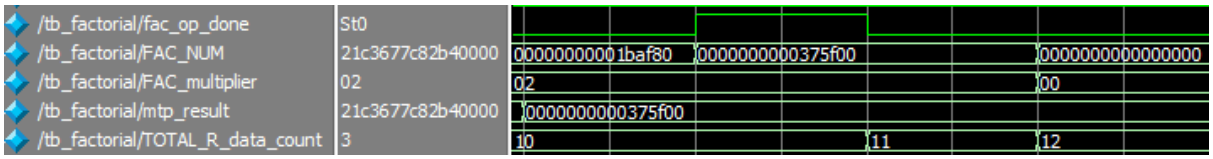
4. 13! = 00000001 | 7328cc00



5. 12! = 00000000 | 1c8cfc00

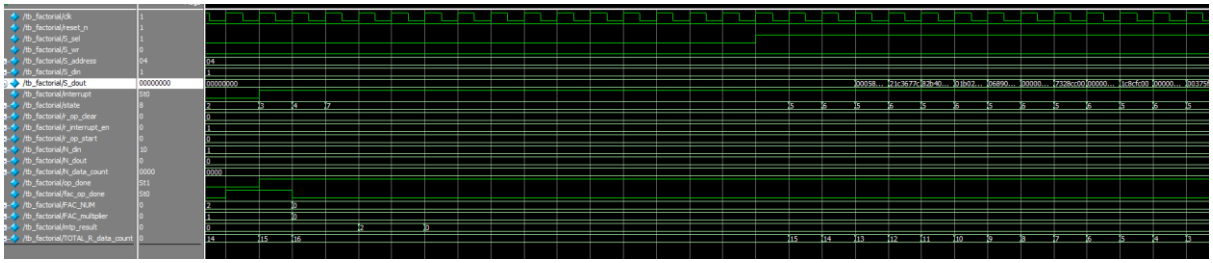


6. 10! = 00000000 | 00375f00



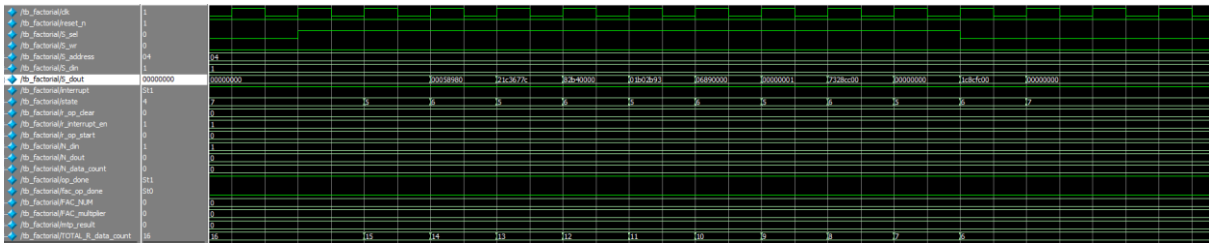
6) S_dout 출력 확인

1. Read signal이 들어오면 Memory에 write하는지 확인

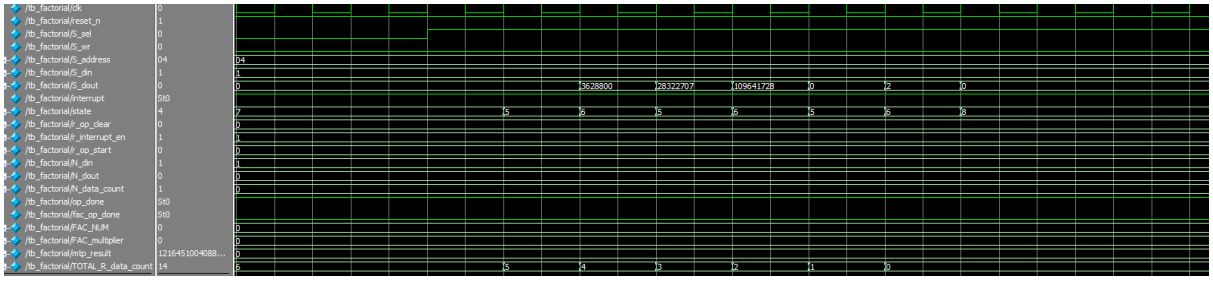


Op_done이 1이어도, read signal인 (S_sel==1)&&(S_wr==0)&&(S_address[3:0]==4'h4)이 충족되어야만 MEM_WRITE state로 넘어가야 한다. Waveform 상으로는 read 조건이 충족이 안될 때 R_FIFO로부터 값을 읽지 않고 STOP_1 state에 있다가 조건이 충족되면서 MEM_WRITE state로 잘 넘어가는 것을 확인할 수 있다.

2. 위에서 얻은 16진수의 결과 값이 분리되어 잘 나오는지 확인

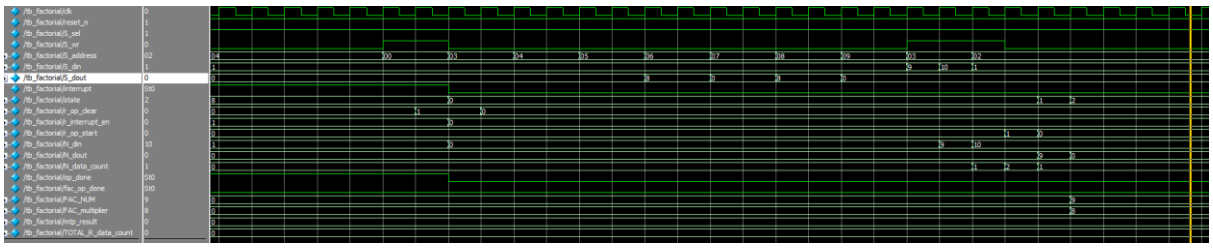


처음엔 Read 조건이 계속 충족되고 있으므로 R_FIFO로부터 연속적으로 값을 POP하여 상위32비트, 하위 32비트 순서로 S_dout에서 출력한다. 위에서 확인한 각 결과값들을 16진수로 변환했을 때의 분리된 두 개의 결과 값들이 순서대로 잘 출력되는 것을 확인할 수 있다. 그러나 중간에 S_sel을 0으로 바꿔서 read 조건을 충족시키지 않게 해봤다. 이렇게 되었을 때는 중간에 MEM_WRITE를 멈추고 STOP state로 가서 멈추는 것을 확인할 수 있다.



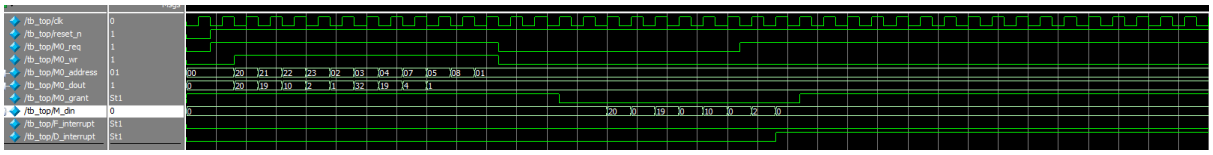
다시 S_sel을 1로 하여 read 조건을 충족시키면 남은 R_FIFO로부터 값을 잘 pop해오는 것을 볼 수 있다.

7) op_clear를 통한 IDLE 상태 변화 확인



Op_clear에 1을 썼을 때의 모습이다. 다음 cycle에서 interrupt가 0으로 내려가고 state는 IDLE, 각 register들은 전부 0으로 초기화가 되는 것을 확인할 수 있다. 그 뒤로 각 register들을 확인해보면 03(N_FIFO), 04(R_FIFO)번지에는 값이 없으므로 0을 출력한다. 05(N_FIFO_COUNT)도 0이 나오고, 06(N_FIFO_FLAGS)번지는 N_FIFO가 empty이므로 1000, 즉 8을 출력한다. 07(R_FIFO_COUNT)도 0이 나오고, 08(R_FIFO_FLAGS)번지는 R_FIFO가 empty이므로 06번지와 마찬가지로 8을 출력한다. 09(op_done)번지도 연산이 초기화 되었으므로 0으로 출력된다. 결론적으로, 모든 register들이 잘 초기화 되었고 그 뒤로 새로 값을 넣고 다시 연산을 시작하여도 잘 이루어지는 모습이다.

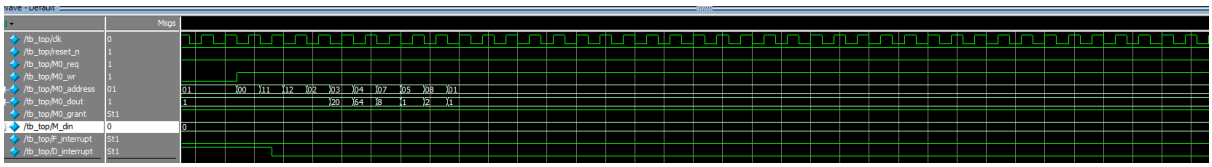
<TOP>



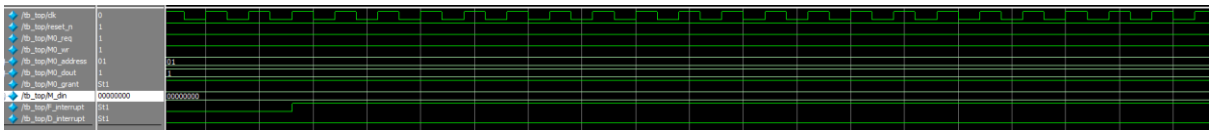
처음에 M0_grant를 제대로 확보하기 위해 M0_req를 1로 올리고 testbench master로 결정해주었다. N-value를 저장하는 RAM의 address인 0x20~0x3F에다가 testbench를 통해 N-value를 저장하는 모습이다. 0x20번지에는 20, 0x21번지에는 19, 0x22번지에는 10, 23번지에는 2를 저장하였다. 그 다음에는 DMAC의 address인 0x00~0x0F에다가 내부 register의 값들을 저장한다. Interrupt_en인 02번지에 1을 넣고, source address인 03번지에 0x20(10진수로 32), destination address인 04번지에 0x13(10진수로 19), data size인 07번지에 4를 넣어주고 push descriptor에 1을 넣어서 data를 push하도록 setting해주었다. Op_mode는 001로 해주었고, 마지막으로 op_start를 하도록 하였다. 먼저 source address를 0x20번지로 하고, destination address를 0x13번지로 한 이유는 DMAC을 통해 RAM으로부터 Factorial의 N_FIFO에 N-value의 값을 전달해줘야 하기 때문이다. N-value는

0x20~0x23번지에 저장되어 있으므로 source address increment가 필요하지만, 이 값들이 저장되는 곳은 factorial의 N_FIFO 한곳이므로 destination address는 계속 유지되어야 하는 op_mode=001로 선언하였다.

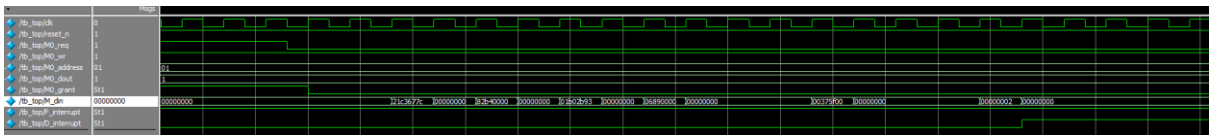
DMAC의 op_start를 1로 해주었기 때문에 DMAC의 master request가 들어올 것이다. 따라서 M0_req를 0으로 내림으로써 M1 grant를 허용하였다. M1_grant가 되면서 DMAC이 RAM으로부터 값을 차례대로 읽어서 M_din의 값이 20, 0, 19, 0, 10, 0, 2, 0으로 나오는 것을 확인할 수 있다. 중간에 나오는 0은 dmac이 read를 하는 state이기 때문에 그런 것이다. 이렇게 전부 factorial의 N_FIFO에 값을 써주게 되면 DMAC의 interrupt를 의미하는 D_interrupt가 1로 되고 Master 1의 request가 0으로 될 것이다. 이제 factorial의 작동 및 결과값 전송을 위한 register setting을 해야 하므로 testbench의 master권한을 얻기 위해 M0_req를 1로 올리고 M0_grant를 확보한다.



M0_grant 확보 후, M0_wr을 1로 올린 후 DMAC의 초기화를 위해 00번지(op_clear register)에 1을 넣어주었다. 이렇게 해서 D_interrupt가 0으로 내려간 것을 확인할 수 있다. 다음으로 Factorial의 interrupt signal check를 위해 11번지(interrupt_en)에 1을 넣고, factorial의 op_start에 1을 넣어서 연산을 시작하게 하였다. 그 다음은 dmac을 setting해주어야 한다. R_FIFO로부터 값을 read해서 RAM(0x40~0x5F)에 write 해야 하므로 source address는 R_FIFO를 가르키는 0x14(10진수로 20)로 하고 destination address는 RAM을 가르키는 0x40(10진수로 64)로 하였다. Data size는 결과 값이 총 8개이므로 8로 설정하였다. Op_mode는 R_FIFO로부터 계속 값을 read하여 RAM의 address를 증가시키면서 write해야 하므로 destination address increment mode인 010으로 설정하였다. 마지막으로 dmac의 op_start에 1을 넣어서 시작 대기 상태로 setting을 완료하였다.

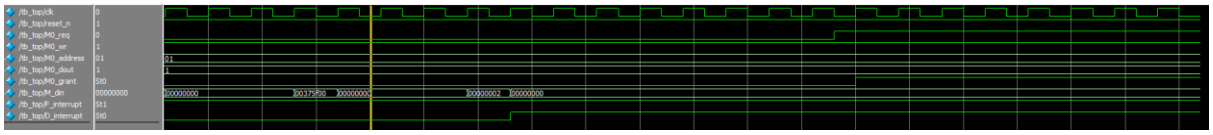


다음과 같이 대략 1000ns 정도가 지난 후에 F_interrupt가 1로 켜진 것을 확인할 수 있다. 이는 factorial의 모든 연산을 끝낸다는 의미이다.



지금 DMAC은 op_start가 된 상태이기 때문에 계속 M1_req를 1로 보내고 있다. 마침 factorial의 연산도 전부 완료하였으니 이제 DMAC에게 master 권한을 주기 위해 M0_req를 0으로 내렸다. 따라서 M1 grant로 바뀌고 factorial 검증에서 검증한 결과 값들이 [63:32], [31:0] 순서로 나오는 것

을 확인할 수 있다. 중간에 한번씩 0이 나오는 것은 dmac이 값을 read하는 state이기 때문이다.



이렇게 마지막까지 값을 전부 ram에 write 하면 DMAC의 interrupt 신호인 D_interrupt가 1로 켜지는 것을 확인할 수 있다. 따라서 M1_req가 0으로 나오고 있을 것이다. 그러나 bus의 구조상 아직 M1_grant이다. M1_req가 0으로 나오는 것을 확실히 확인하기 위해 마지막 쯤에 M0_req를 1로 해보았다. 예상대로 M0 grant가 인가되었다. 이는 M1_req가 0으로 나오고 있다는 사실을 증명한다.