Digital Systems Design

Term Project: CNN Accelerator

Final Report

Team 2

전기·정보공학부 2019-10108 이창우

전기·정보공학부 2019-16314 한동민

전기·정보공학부 2020-17400 김호준

CONTENTS

1. `scale_uart.py` Address Mapping

```
def su_conv_control(self, I, F, W, B, R, vaddr, caddr):
    ######################################################################
    # You should revise below code
    # Your apb-register and FSM will not match below control flow
    ######################################################################
    ######################################################################
    # You should revise below code
    # Your apb-register and FSM will not match below control flow
    # Address Space of VDMA0 : 0x0D00_0000(vaddr) - 0x0D0F_FFFF (2^20)
    # Address Space of APB0  : 0x0C00_0000(faddr) - 0x0C0F_FFFF (2^20)
    #
    # Address @ Python  |  Signal @ Testbench
    # ------------------+----------------------
    #       0x00        |   conv_start
    #       0x04        |   InCh
    #       0x08        |   clk_counter
    #       0x0C        |   FLength
    #       0x10        |   NaN
    #       0x14        |   NaN
    #       0x18        |   OutCh
    #       0x1C        |   conv_done
    #       0x20        |   F_writedone
    #       0x24        |   B_writedone
    #       0x28        |   rdy_to_transmit
    #       0x2c        |   transmit_done
    #       0x30        |   F_writedone_respond
    #       0x34        |   B_writedone_respond
    #       0x38        |   rdy_to_transmit_respond
    #       0x3c        |   transmit_done_respond
    #       0x40        |   COMMAND
```

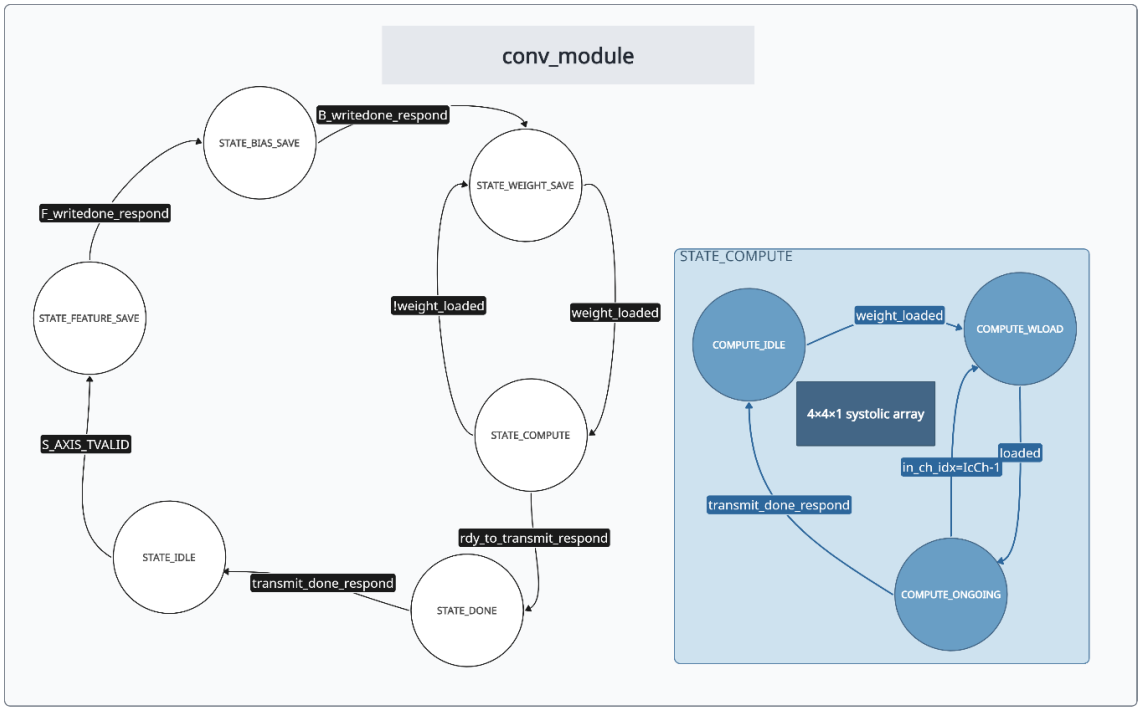
```
def su_fc_control(self, F, W, B, R, vaddr, faddr):
    ######################################################################
    # You should revise below code
    # Your apb-register and FSM will not match below control flow
    # Address Space of VDMA0 : 0x0C00_0000(vaddr) - 0x0C0F_FFFF (2^20)
    # Address Space of APB0  : 0x0D00_0000(faddr) - 0x0D0F_FFFF (2^20)
    #
    # Address @ Python  |  Signal @ Testbench
    # ------------------+----------------------
    #       0x00        |   COMMAND
    #       0x04        |   NONE
    #       0x08        |   clk_counter         // from fc_apb.v skeleton code
    #       0x0C        |   fc_done
    #       0x10        |   max_index           // from fc_apb.v skeleton code
    #       0x14        |   F_writedone
    #       0x18        |   B_writedone
    #       0x1C        |   NONE
    #       0x20        |   start_response      // ADDED
    #       0x24        |   done_response       // ADDED
    #       0x28        |   fc_start            // ADDED
    #       0x2c        |   receive_size_f      // ADDED
    #       0x30        |   receive_size_b      // ADDED
    #       0x34        |   receive_size_w      // ADDED
```

```
def su_pool_control(self, I, F, R, vaddr, paddr):
    ######################################################################
    # You should revise below code
    # Your apb-register and FSM will not match below control flow
    # Address Space of VDMA2 : 0x0C20_0000(vaddr) - 0x0C2F_FFFF (2^20)
    # Address Space of APB2  : 0x0D20_0000(paddr) - 0x0D2F_FFFF (2^20)
    #
    # Address @ Python  |  Signal @ Testbench
    # ------------------+----------------------
    #       0x00        |   pool_start, init
    #       0x04        |   pool_done
    #       0x08        |   clk_counter
    #       0x0C        |   IN_CH // num of channel
    #       0x10        |   FLEN  // input size
```

`scale_uart.py`에서 모듈 별 data들에 대한 address는 위와 같이 설정하였다. 기본적인 command들 뿐 아니라 input matrix의 size와 channel에 관한 정보들을 받아 적절한 연산을 수행할 수 있도록 구현하였다.

2. Modules

2.1. conv_module



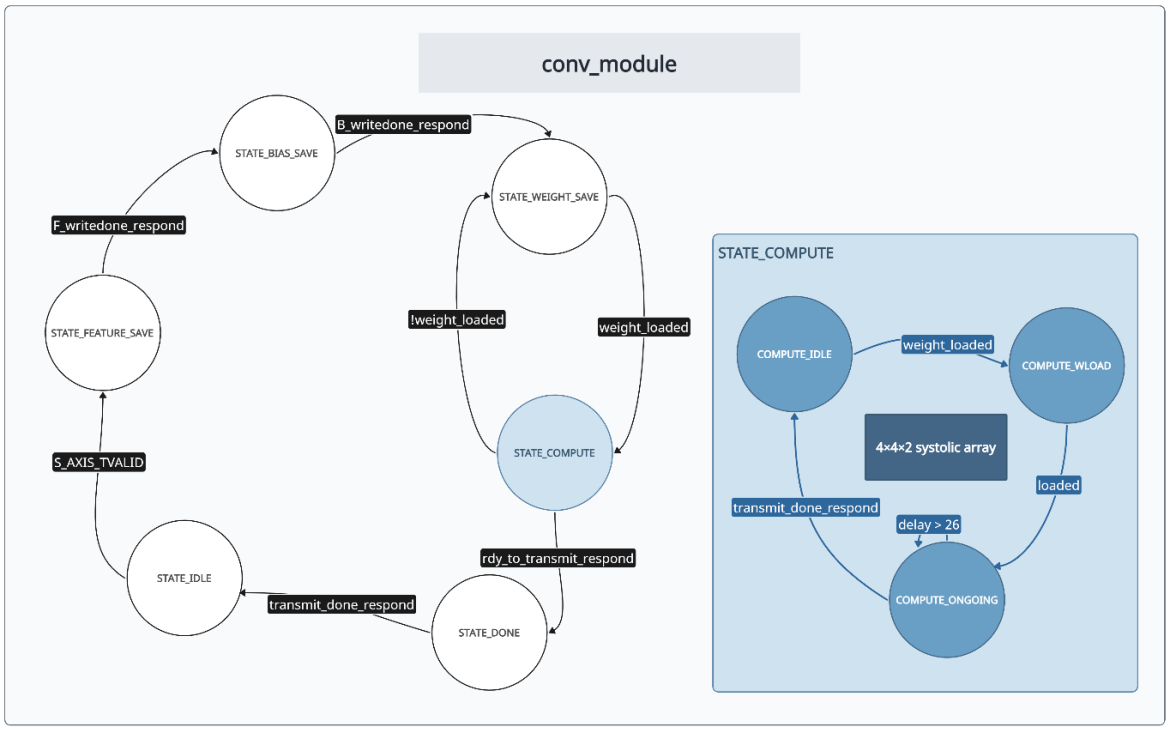
2.1.1. Data Saving

총 4개의 BLOCK RAM을 사용하여 FEATURE, BIAS, WEIGHT, OUTPUT을 따로 저장해 주었다. Weight의 경우 parameter의 개수가 많아 한 번에 BLOCK RAM에 담을 수 없어 한번의 `STATE_WEIGHT_SAVE` state에서는 4개의 `out_ch` 만큼의 weight를 저장하도록 하였다. 4x4 systolic array를 사용하여 conv 연산을 구현하였기 때문에 바로 data를 전송할 수 없어 연산 결과 값을 저장하는 BLOCK RAM도 사용하였다. Feature와 bias의 경우 한 번에 다 전송을 받지만 weight는 해당 weight가 필요한 연산이 끝날 때마다 `S_AXIS_TDATA`로부터 전송 받아, BLOCK RAM에 address 0부터 새로 값을 적어준다. 이때 `S_AXIS_TREADY`값을 적절하게 0으로 내려주도록 구현하였다.

2.1.2. Computing

4x4 systolic array를 사용하였다. 이미 BLOCK RAM에 저장된 데이터를 받아오는 `COMPUTE_WLOAD` state와 받아온 데이터를 바탕으로 연산을 수행하는 `COMPUTE_ONGOING` state로 나누어서 구현하였다. 이때 `S_AXIS_TDATA`로부터 저장한 weight들을 모두 사용하고 나면 다시 `STATE_WEIGHT_SAVE` state로 넘어가서 필요한 데이터들을 새로 받아온다. `COMPUTE_WLOAD` state에서는 4x8x9 bit의 feature와 weight를 받아오는데, 이는 weight가 3x3을 기준으로 연산이 진행되기 때문이다. 즉 9개의 값들을 받아와 한 output matrix entry에 대한 연산을 한번의 `COMPUTE_ONGOING` state에서 끝내기 위해 4x8x9 bit을 받도록 구현한 것이다.

2.1.3. Pipelining, Parallel(3D) Systolic Array

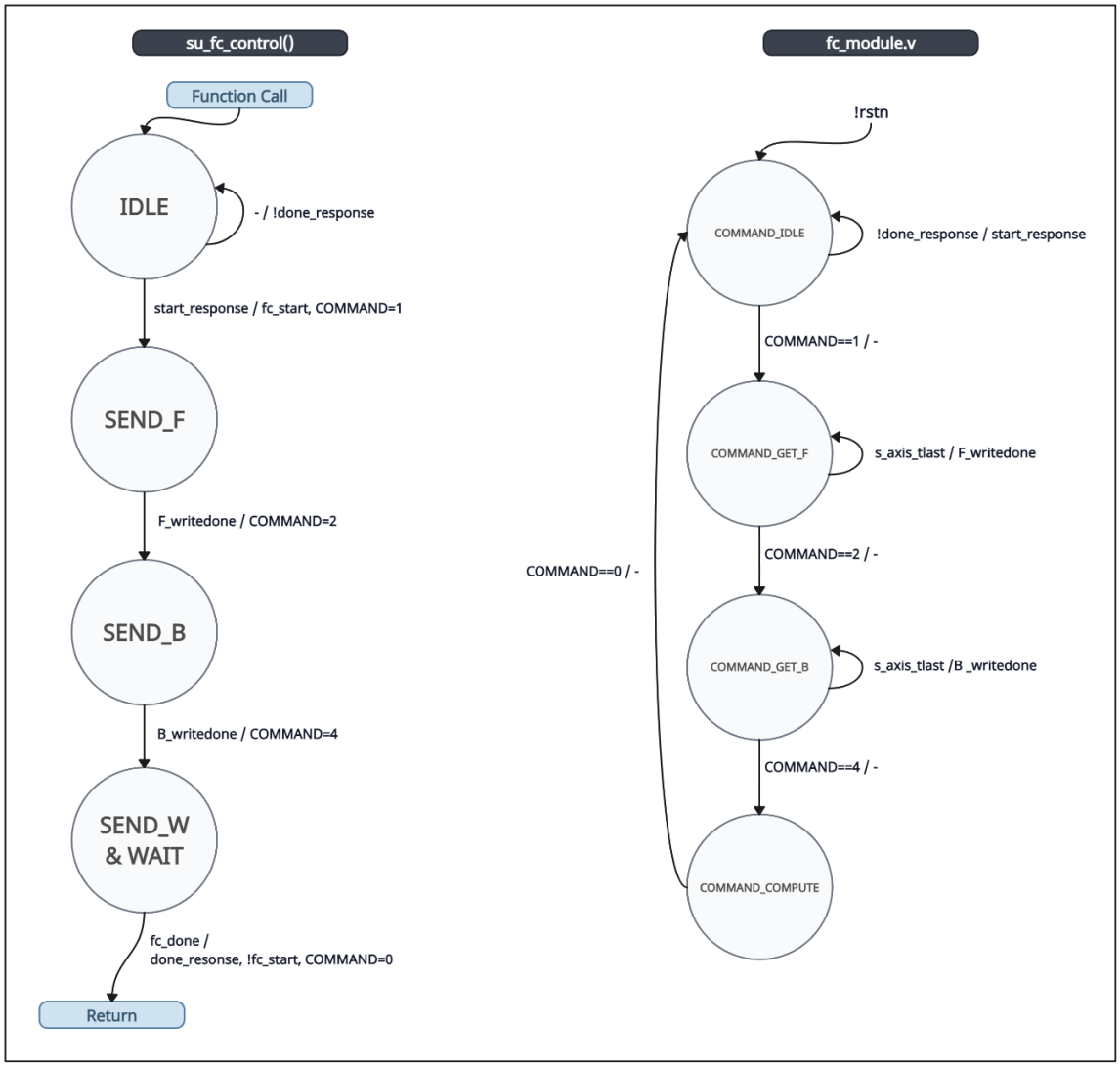


성능을 높이기 위해 고민을 하였고, `COMPUTE_ONGOING` state와 `COMPUTE_WLOAD` state를 구분할 필요가 없다는 생각이 들어 두 state를 동시에 진행하는 방식으로 pipelining을 구현하였다. 결과적으로 시간은 50%정도 줄어드는 것을 확인했다.

또, systolic array를 32x4, 8x8 과 같이 확장하는 방법도 생각해 보았으나 그만큼 기다려야 하는 delay가 늘어나기 때문에 적합하지 않다고 판단했고, genvar를 활용하여 4x4x2 parallel systolic array를 적용하여 pipelining을 적용한 모델과 비교했을 때 시간을 20%가량 단축시켰다.

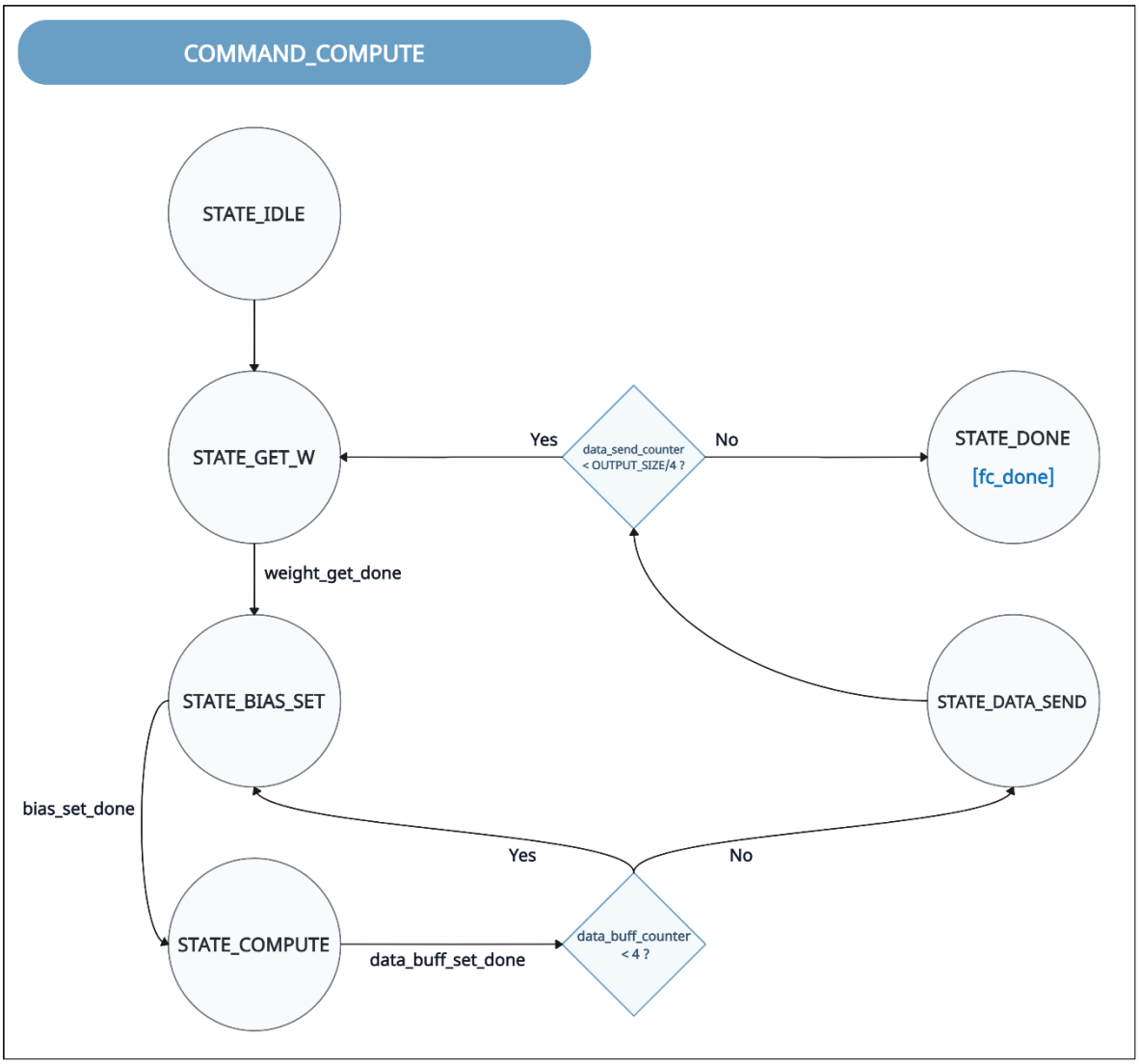
2.2. fc_module

2.2.1. High-Level FSMs: `su_fc_control()` & `fc_module.v`



위 그림과 같이 `scale_uart.py`에 있는 `su_fc_control()`함수의 `COMMAND` signal에 따라 Feature, Bias, Weight data를 모듈 내부로 받도록 구현하였다. 이때 Feature 및 Bias는 필요한 모든 data를 한 번에 받아오는 반면, Weight는 `COMMAND_COMPUTE` 내부에서 32-bit output 하나를 내보내기 위해 필요한 개수인 `INPUT_SIZE*4`씩 총 `OUTPUT_SIZE/4`번에 걸쳐 받아오도록 하였다.

2.2.2. Intra-Module FSM: `fc_module.v`



- STATE_IDLE

`COMMAND_COMPUTE`의 최초 STATE이다. Internal FSM에 필요한 여러 가지 signal을 초기화한다. 1 cycle 이후 `STATE_GET_W`로 transition한다.

- STATE_GET_W

`INPUT_SIZE*4` 만큼의 8-bit Weight를 받는 state이다. 하나의 output을 계산하기 위해 필요한 모든 data를 다 받으면 `weight_get_done`을 assert하여 `STATE_BIAS_SET`으로 transition한다.

- STATE_BIAS_SET

BRAM_B의 한 address에 저장된 32-bit data 중 하나의 8-bit bias data를 선택한 후 27-bit으로 extension하여 `psum`에 저장한다. 이후 `bias_set_done`을 assert하여 `STATE_COMPUTE`로 transition한다.

- STATE_COMPUTE

매 cycle마다 BRAM_F, BRAM_W에서 32-bit data를 읽고 이로부터 4개의 $I*W$ term을 combinational logic으로 계산하여 `psum`에 더한다. 이때 accumulate의 latency는 `INPUT_SIZE/4` cycle이다.

MAC 연산의 결과는 8-bit인 반면 data send granularity는 32-bit이기 때문에, 4개의 output을 `data_buff`에 저장해야 한다. 따라서, MAC 연산이 끝난 이후 (`data_buff_set_done`) buffer를 확인하여 buffer가 다 채워진 경우 `STATE_DATA_SEND`, 그렇지 않은 경우 `STATE_BIAS_SET`으로 transition한다.

- STATE_DATA_SEND

AXI-Stream protocol을 통해 `data_buff`의 data를 내보낸다. 앞서 언급했듯이 granularity가 32-bit (4 outputs)이기 때문에, 총 data send 횟수는 `OUTPUT_SIZE/4`이다. 따라서, `data_send_counter` signal을 이용하여 data send 횟수를 세고, 모든 data를 보냈다면 `STATE_DONE`으로 transition한다.

- STATE_DONE

`fc_done` signal을 assert하여 `su_fc_control()`이 `COMMAND`에 0을 assign하도록 한다. 이 때 `STATE_IDLE` 및 `COMMAND_IDLE`로 transition하여 모듈을 초기화한다. 단, `max_index` register는 초기화하지 않는다.

2.3. pool_module

2.3.1. Overall Operation

scale_uart.py에서 su_pool_control() 함수의 pool_start가 1이 되면 pool_module이 작동을 시작한다. pool_module 내에서는 S_AXIS_TREADY와 S_AXIS_TVALID가 모두 1이면 S_AXIS_TDATA의 데이터를 받기 시작하고, input matrix의 size에 따라 총 두 개의 row까지 유동적으로 저장할 수 있도록 temp reg를 사용하여 구현하였다. 데이터를 모두 받아 저장한 다음 pooling을 수행하는 것이 아니라, pooling이 가능한 만큼의 데이터가 모이면 데이터를 받음과 동시에 pooling result를 내보내는 real-time 계산 방식을 채택하였다. pooling_module 내에 (input_size*input_size*input_channel_size)/4에 해당하는 num_input을 정의하였고, counter가 이보다 커지면 모든 데이터를 받고 pooling 연산을 끝낸 것이므로 pool_done과 m_axis_tlast를 1로 올려주도록 구현하였다.

2.3.2. `find_max` task & `input_size` Handling

Pooling의 filter size는 2x2로 고정되어 있기 때문에, 4개의 8bit 원소를 받아 max 연산을 수행하는 find_max라는 task를 정의하였다. Input의 row 하나가 모두 들어온 후 그 다음 row의 원소가 네 개씩 들어올 때마다 find_max 연산의 결과 값을 m_axis_tdata reg의 알맞은 index에 넣어주도록 구현하였다. 데이터는 32bit씩 들어오기 때문에 S_AXIS_TDATA를 한 번 받는 것이 matrix의 원소 네 개를 받는 것에 해당한다.

pooling_module의 input으로 들어오는 matrix들은 size가 4x4, 8x8, 16x16, 32x32로 한정되어 있기 때문에, 각 경우에 따라 find_max를 수행해주는 원소의 index들을 다르게 설정하였다. Matrix의 input_size는 scale_uart.py에서의 FLEN에 해당하고, input_channel_size는 IN_CH에 해당한다. 이 값들을 pool_apb.v에서 받고, 이를 pool_module의 input으로 넘겨줄 수 있도록 구현하였다.

3. Contribution

- 이창우 (fc)

→ merge two controllers in Lab7 into one, simplify MAC without using module

Lab7에서 Compute Unit Controller (CUC) 및 Memory Unit Controller (MUC)를 이용하여 fc module을 구현하였지만, 프로젝트에서는 좀더 직관적으로 구현하기 위해 하나의 FSM으로 나타냈다. 또한 8-bit multiplication이 timing violation을 일으키지 않는 것을 확인한 후, MAC module을 사용하지 않고 '*' operator를 이용해 매 cycle마다 4개의 $I*W$ term을 parallel하게 계산함으로써 연산 latency를 줄일 수 있었다.

testbench로 주어진 data가 3개의 fc layer 중 2개밖에 없었기 때문에 `.npy` 파일로부터 FC layer 1에 해당하는 data를 text file로 변환하여 simulation에 사용하였다. 이를 통해 `STRIDE_SIZE`마다 `s_axis_t_last` signal이 assert되는 것을 확인하는 등 디버깅에 큰 도움이 되었다.

최초에는 32x65536의 BRAM을 사용하여 모든 weight을 한 번에 받는 형태로 구현했지만 해당 구현을 implement 시 `fc_module.v`에서만 약 57%의 BRAM을 사용하는 것을 확인했다. 따라서 resource 사용량을 줄일 필요성을 느꼈고, 이에 따라 weight는 4개의 output (32-bit)을 계산할 양만큼씩 나누어서 받도록 구현을 수정했다. 이로써 `fc_module.v` 내에서 모듈 사용량을 약 40%p 줄일 수 있었다.

- 한동민 (pool)

→ `find_max` task design, minimal usage of reg without bram, real-time max pooling

pooling layer는 input data들이 row 별로 들어오고, 이에 대해 2x2 pooling 연산을 수행하기 때문에 2개의 row만 있으면 연산을 수행할 수 있다. 즉 모든 data를 bram에 저장하고 그 후 연산을 수행하는 것이 아니라, 최대 두 개의 row만 reg에 저장하고 연산을 수행한 후 `m_axis_tdata`를 통해 결과를 내보내는 real-time max pooling 방식을 채택하였다. 이에 `pool_module`에서의 bram 사용량을 획기적으로 줄임으로써 다른 module에서 부담 없이 resource를 사용할 수 있도록 하였다. 또한 data를 저장하고 다시 내보내는 과정이 없으므로, cycle 수를 최소화할 수 있었다.

최초에는 `s_axis_tready`를 wire로 구현하여, simulation tb는 모두 통과하는 데 반해 실제 hardware test에서는 input이 한 cycle 씩 밀리는 현상이 발생하였다. 이에 reg로 구현된 `sync_s_axis_tready`를 추가하여, `s_axis_tready` signal이 모듈 내에서 synchronous하게 올라올 수 있도록 함으로써 문제를 해결하였다.

기본 제공된 8x8 matrix인 `pool_input_32bits.txt` 뿐 아니라, `new_cifar10_random_data` directory에 있는 txt file들을 32bit마다 줄바꿈하여 simulation으로 실행시켜 보았다. 이로써 board에 올리기 전 simulation을 통해 모든 input size, 즉 32x32, 16x16, 8x8, 4x4 matrix에 대한 pooling 연산의 정확성을 확인할 수 있었다.

- 김호준 (conv)

→ im2col, systolic array, weight load-compute pipelining, 3D systolic array using genvar

상기 4가지 모두 어려움을 겪었으나 waveform을 활용하여 틀린 값 – `row`, `column`, `inch`, `outch idx` value들을 수정해 나갔다. state 별로, 연산 종류 별로 나눠서 테스트를 진행하였다.

testbench는 통과했지만 single layer test에서 작동하지 않는 문제가 있었다. 포트 연결을 통해 그 이유가 `OutCh`로 할당해 놓은 address의 value의 값을 python skeleton 코드에서 0으로 재할당해주는 것이 원인임을 발견했고, 해당 줄을 삭제함으로써 해결하였다.

원래 weight를 한 번에 저장하고 그 후에 연산을 진행하는 방식으로 코드를 짰었는데, 이 방식은 BRAM의 width가 150000정도 필요했다. 이는 FPGA에 올릴 수 없었고, `STATE_WEIGHT_SAVE`와 `STATE_COMPUTE` 간에 상황에 따라 state가 변하는 logic을 추가하여 weight를 분할해서 받는 방식으로 코드를 수정했다. 왜 tb에 `W_writedone`, `W_writedone_respond` signal은 없었는지 알게되었다.