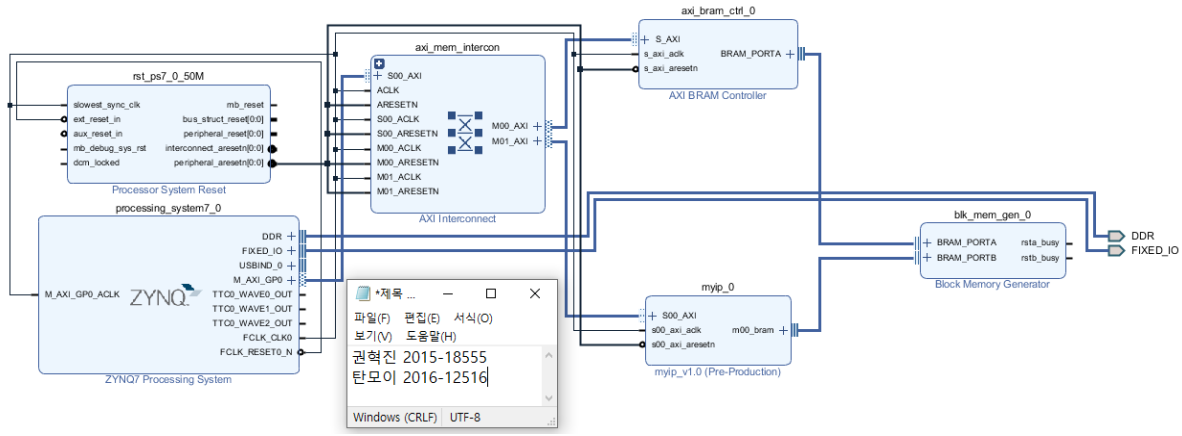
Report : HSD - term project v0

권혁진 2015-18555

탄모이 2016-12516

HW 구현



LAB 10에서 제공되었던 Block Design을 기반으로 IP중 하나인 myip_v1.0을 수정하여 이전까지 구현했던 pe_controller를 추가하고 이를 BRAM과 연결하여 Hardware를 구현하였습니다.

pe_controller는 다음과 같이 동작하는 moore machine입니다.

- State = {PE_IDLE, PE_LOAD, PE_CALC, PE_DONE}
  - PE_IDLE
    - input signal인 start의 신호를 기다립니다.
    - start signal 발생 시 PE_LOAD로 state를 바꿉니다.
  - PE_LOAD
    - BRAM으로부터 두 matrix A,B를 load하여 global buffer에 저장합니다.
    - 이 때 matrix B 또한 A와 같이 row를 우선으로 load하기 때문에 이후에 계산시 결과 matrix는 $A \cdot B$가 아닌 $A \cdot B^T$가 됩니다.
      - 따라서 software 작성 시 이를 유의하여야 합니다.
    - 모든 data를 load한 후 PE_CALC로 state를 바꿉니다.
  - PE_CALC
    - 각 연결된 모든 Processing element에 input으로 global buffer에 있는 data를 제공하며 MAC연산을 반복하여 matrix multiplication을 시행합니다.
      - 앞서 언급하였듯이 그 결과는 $A \cdot B^T$입니다.
    - 연산이 끝나면 PE_DONE으로 state를 바꿉니다.
  - PE_DONE
    - PE_CALC에서의 연산으로 얻은 data를 BRAM에 write합니다.
    - BRAM에 write하는 작업이 끝나면 각 Processing element에 reset signal을 주어 초기화 시켜 다음 작업을 준비합니다.

        - 동시에 output port인 done port에 신호를 주고 이를 5-cycle 동안만 유지한 후 PE_IDLE로 state를 바꿉니다.
- Reset
    - Synchronous negative reset입니다.
    - State를 PE_IDLE로 바꿉니다.
- BRAM communication
    - BRAM의 특성 상 pe_controller와 위상인 180도 차이나는 clock을 BRAM의 clock으로 주었습니다.
        - 이를 위해 clock wizard IP를 사용하였습니다.

Error : [place 30-487]

Implementation을 실행 중 다음의 에러가 발생하였습니다. SLICE 부족으로 LUT를 배치할 수 없어 발생한 에러였습니다. 해결을 위해 LUT 사용량을 최소화 하는 동시에 SLICE를 확보를 해야했습니다.

1. if-else 블록 나누기

pe_controller을 moore machine으로 구현하기 위해 state에 따른 logic 부분을 큰 if-else block으로 구현하였으나 이는 LUT 사용량을 증가시키는 원인이 되었습니다. 따라서 기존의 코드에서 큰 block들을 최대한 나누었습니다. 각각의 state별로 `always @(posedge CLK)` block을 할당하였습니다.

실제 LUT 사용량이 감소했느나, 그 감소량은 적었습니다.

2. Distributed ram 사용하기

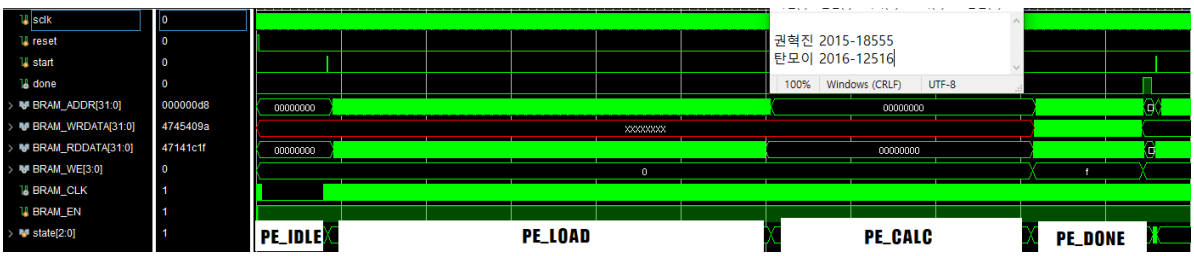
기존의 코드에서 buffer를 구현하기 위해서 register file을 사용했습니다. 그러나 SLICE의 부족으로 최대한 SLICE를 확보할 필요가 있었습니다. 따라서 큰 register file를 LUT 전체적으로 분산할 수 있는 Distributed ram을 사용하였습니다.

3d ram은 vivado에서 지원되지 않아 register file로 구현되기 때문에, 기존에 3d ram 형식으로 선언하였던 global buffer를 2d ram 형식으로 선언하는 동시에 Distributed ram을 사용하도록 지정하였습니다.

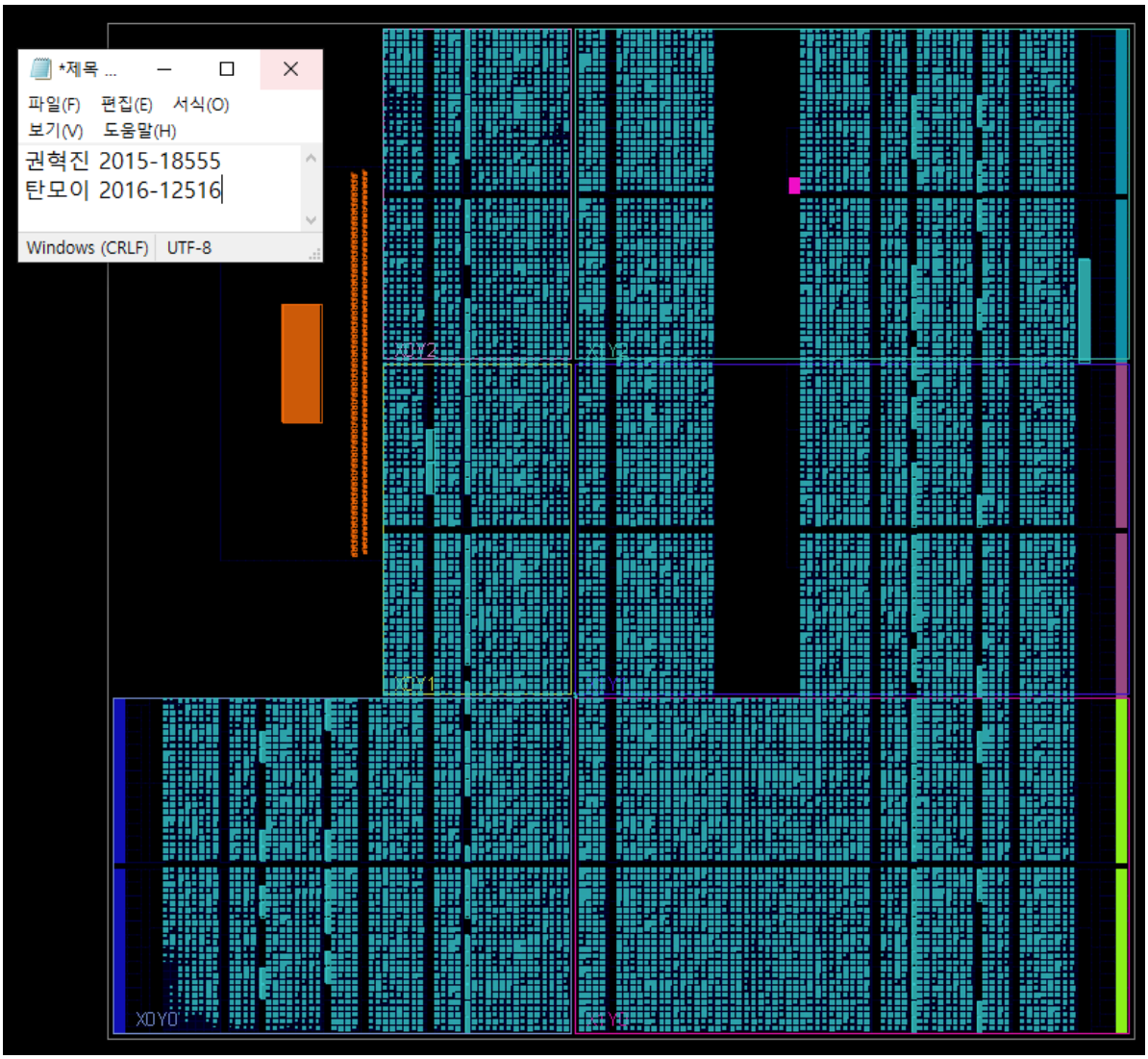
```
(* ram_style = "distributed" *) reg [BITWIDTH-1:0] ain[L_RAM_SIZE-1:0];
(* ram_style = "distributed" *) reg [BITWIDTH-1:0] bin[L_RAM_SIZE-1:0];
(* ram_style = "distributed" *) reg [BITWIDTH-1:0] gb1[L_RAM_SIZE*L_RAM_SIZE-
1:0];
(* ram_style = "distributed" *) reg [BITWIDTH-1:0] gb2[L_RAM_SIZE*L_RAM_SIZE-
1:0];
```

실제 LUT 사용량은 큰 변화가 없었으나 SLICE 사용량을 제한치 이하로 낮출 수 있었습니다.

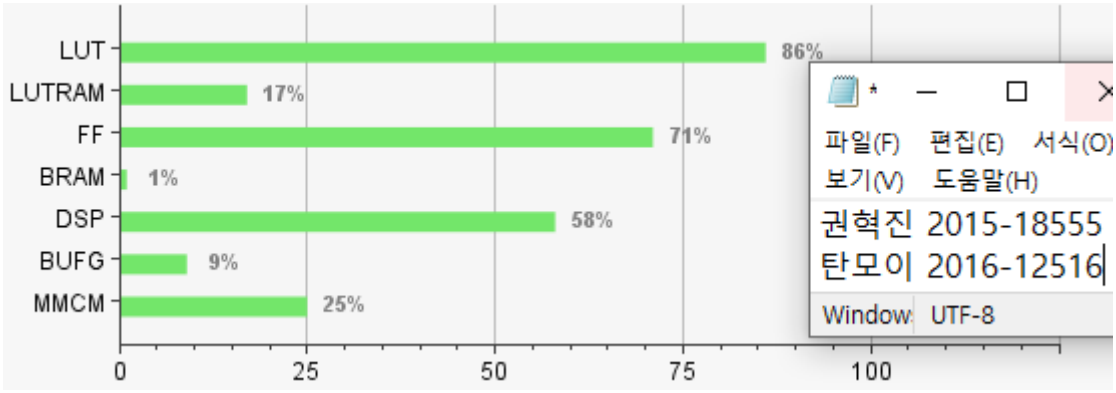
Simulation



Implementation report



- Floorplanning



- Resource Usage

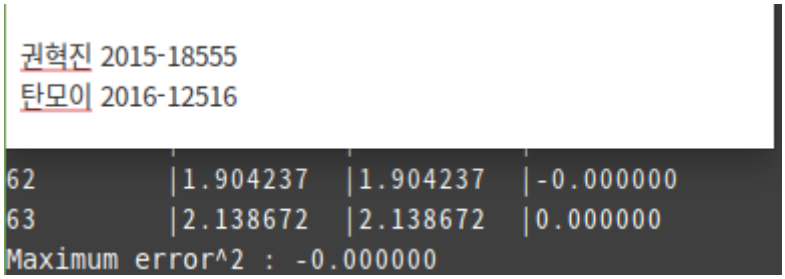
SW 구현

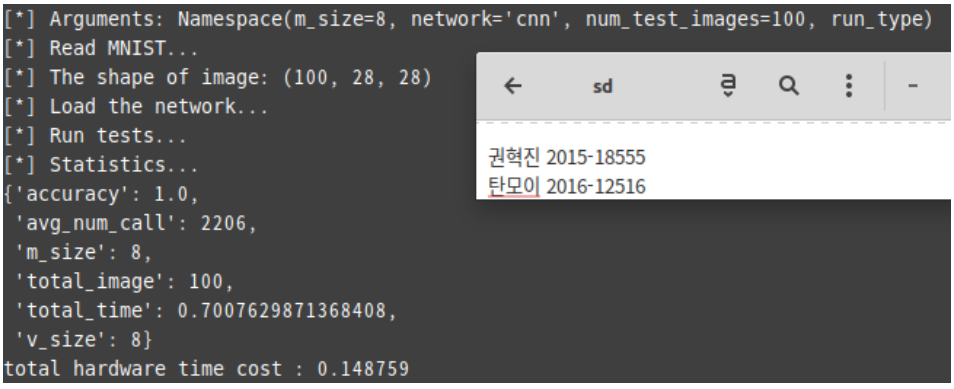
이전 LAB에서 작성한 코드를 재사용하였습니다.

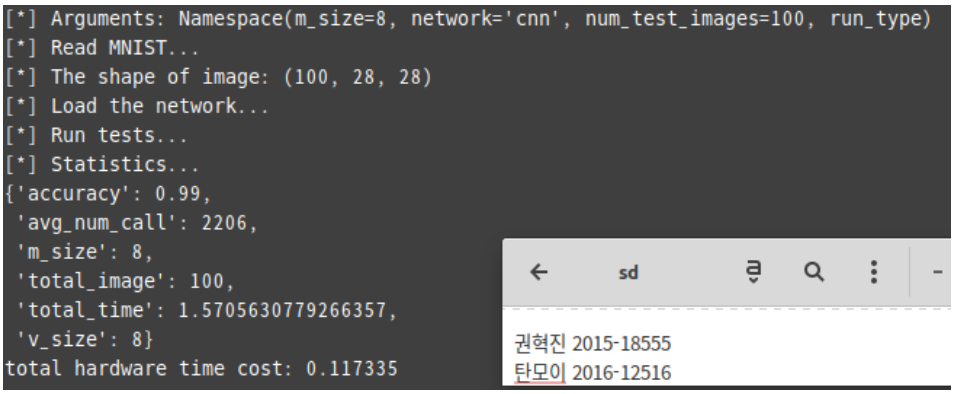
Matrix-Vector, Matrix-Matrix 곱 모두 tiling을 통해 작게 분할하고 분할된 것들끼리의 곱을 반복하여 결과를 생성해냅니다.

Matrix-Matrix 곱의 경우 하드웨어를 사용하는 경우 tiling 시 input_matrix의 경우 tiling된 부분을 transpose한 형태로 blockMM method에 넘겨집니다. 이는 hardware 구현상 두 번째 matrix가 transpose된 상태로 연산되기 때문입니다.

검증 및 testbench

- 검증
  - 제공된 verify.cpp를 수정하여 작성하였습니다.
  - 8x8 크기의 random한 두 matrix의 곱을 cpu와 hardware 상에서 수행한 후 둘의 오차 matrix에서 제곱의 크기가 최대인 element를 통해 검증하였습니다.
  - 

- Testbench
  - Result
    - CPU 연산
      - 

      - blockMM : 총 수행 시간 : 0.148759 sec
    - Hardware 연산
      - 

      - blockMM : 총 수행 시간 : 0.117335 sec
  - 결과 분석
    - blockMM에서의 총 수행 시간은 Hardware 연산에서 cpu대비 21% 감소했습니다.
    - 그러나 전체 수행 시간은 증가하였는데
      - 이는 Input matrix의 tiling 시 각 tile을 transpose된 형태로 blockMM에 제공하기 위해 memcpy를 통해 한 행을 복사하지 못하고 tile의 각 element를 하나씩 nested for-loop를 통해 대입하여 발생한 overhead로 판단됩니다.
    - Tiling의 방식 및 크기는 동일하므로 두 경우 모두 blockMM의 호출 횟수는 동일합니다.
    - 다만 accuracy에서 작은 감소가 발생했습니다.
      - hardware에서의 작은 오차가 중첩되어 인해 발생한 것으로 추정합니다.

- 개선방안
  - Simulation에서 확인할 수 있듯, 한 번의 hardware 작업 시, 전체 소비 cycle 중 많은 비율이 PE_LOAD에서 사용됩니다.
    - 따라서 PE_LOAD에서의 소비 cycle을 줄일 경우 성능 개선이 가능할 것으로 보입니다.
    - 구현하지 못한 quantization, zero-skipping이 이 경우에 속하는 것으로 판단됩니다.
  - floating point 연산을 담당하는 모듈에서도 많은 수의 cycle이 소비가 됩니다.
    - 따라서 연산의 delay를 줄여 성능 개선이 가능할 것으로 보입니다.
    - 구현하지 못한 quantization이 이 경우에 속하는 것으로 판단됩니다.