AI가속기설계

Tiny Neural Network 설계

과제 #1

건국대학교

202110410

조민우

서론

본 과제는 Tiny Neural Network(TNN)를 구성하는 주요 연산 모듈들을 Verilog로 구현하고, 이를 통해 FC Layer, Normalization, ReLU 연산을 처리하는 하드웨어 가속기를 설계하는 것을 최종 목표로 한다.

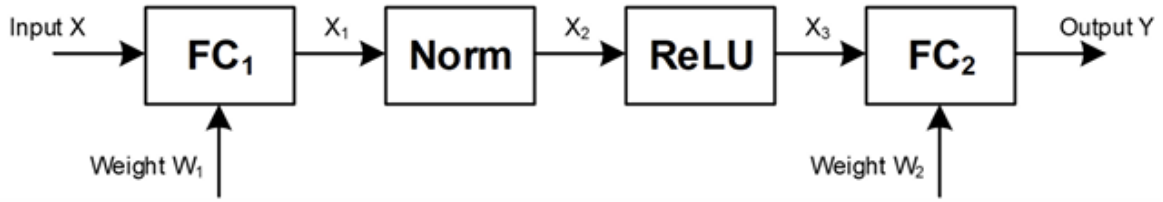


Fig. 1. Target neural network architecture.

Fig 1.1 Tiny Neural Network

TNN의 기본 동작 원리를 살펴보면, 8×8 크기의 입력 데이터와 가중치 행렬 사이의 내적 연산이 핵심이다. 이를 통해 fc Layer가 구현되는데, 결국 input 벡터와 weight 벡터의 각 원소를 곱하고 모든 결과를 더해서 하나의 출력값을 만들어내는 과정이다. 여기에 활성화 함수까지 적용하면 fc layer의 기본 연산이 완성된다.

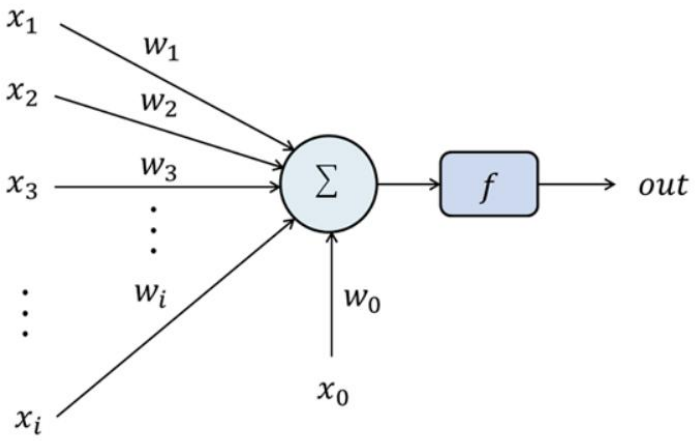


Fig 1.2 Fully Connected Layer

과제1에서는 이런 행렬 곱셈의 기초가 되는 8비트 Ripple Carry Adder와 4비트 Multiplier를 구현하는 것을 목표로 한다. 곱셈기는 shift-and-add 방식으로 구현하며, -8부터 7까지의 입력을 받아 -128부터 127까지의 출력을 낼 수 있도록 설계한다. 그리고 입력이 바뀌면 바로 출력이 나오도록 combinational logic으로 구현하고자 한다.

본론

- HA(반가산기)와 FA(전가산기)

HA는 두 비트의 입력을 받아 sum과 carry-out를 계산하는 조합 논리 회로이다. 반면 FA는 세 개의 입력(A, B, Carry-in)을 받아 sum과 carry -out을 출력한다. HA는 LSB에서 사용되며, FA는 이후 모든 비트에서 사용된다.

이 두 구성 요소는 n비트 덧셈기 설계의 핵심 블록으로 사용된다. 본 과제에서는 8비트 Ripple Carry Adder 구현을 위해 1개의 HA와 7개의 FA를 사용하여 구현하였다.

-Ripple Carry Adder8

Ripple Carry Adder는 가장 하위 비트부터 상위 비트로 carry가 전달되는 구조를 가지며, n개의 비트에 대해 n개의 가산기를 직렬로 연결하여 구성된다.

LSB는 carry-in이 필요 없으므로 HA를 사용하고, 나머지 비트는 이전 단계의 carry-out을 carry-in으로 받아 FA로 처리된다. 각 단계의 carry가 다음 단계로 전달되므로 전체 연산 시간이 비교적 길지만 구조가 단순하고 구현이 쉽다는 장점이 있다.

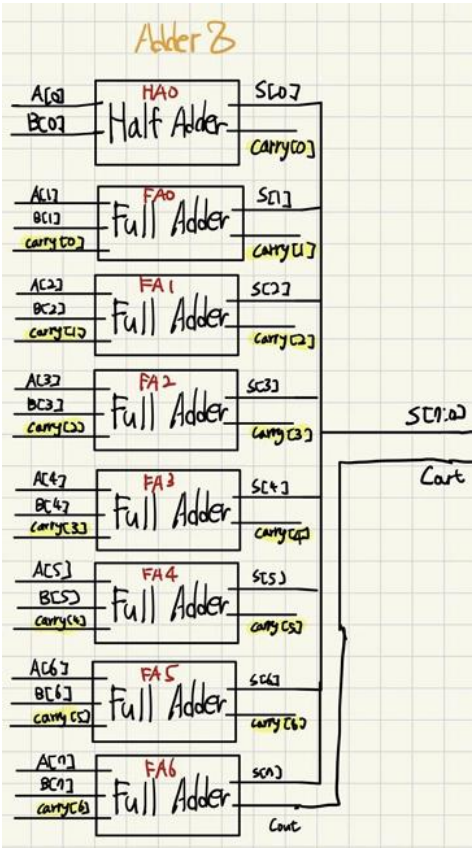


Fig 2.1 Adder8 Block Diagram

-Multiplier

4비트 곱셈기는 shift-and-add 방식을 기반으로 구현되었다. 입력 A는 부호 확장을 통해 signed 8비트로 변환되며, 입력 B의 각 비트에 대해 AND 연산과 적절한 left shift를 수행한 후, 이들을 모두 더해 최종 곱셈 결과를 생성한다.

예를 들어, 다음 연산 과정을 살펴보자. 35 x 27 은 다음과 같이 생각해볼 수 있다. 35 x 27 = (35 x 2) x 10 + (35 x 7) x 1. 이를 이진수에도 똑같이 적용할 수 있다.

MSB가 1인 경우(음수 처리)는 추가적으로 2의 보수를 이용한 보정 연산을 수행하여 signed 곱셈 결과가 정확하게 계산되도록 한다.

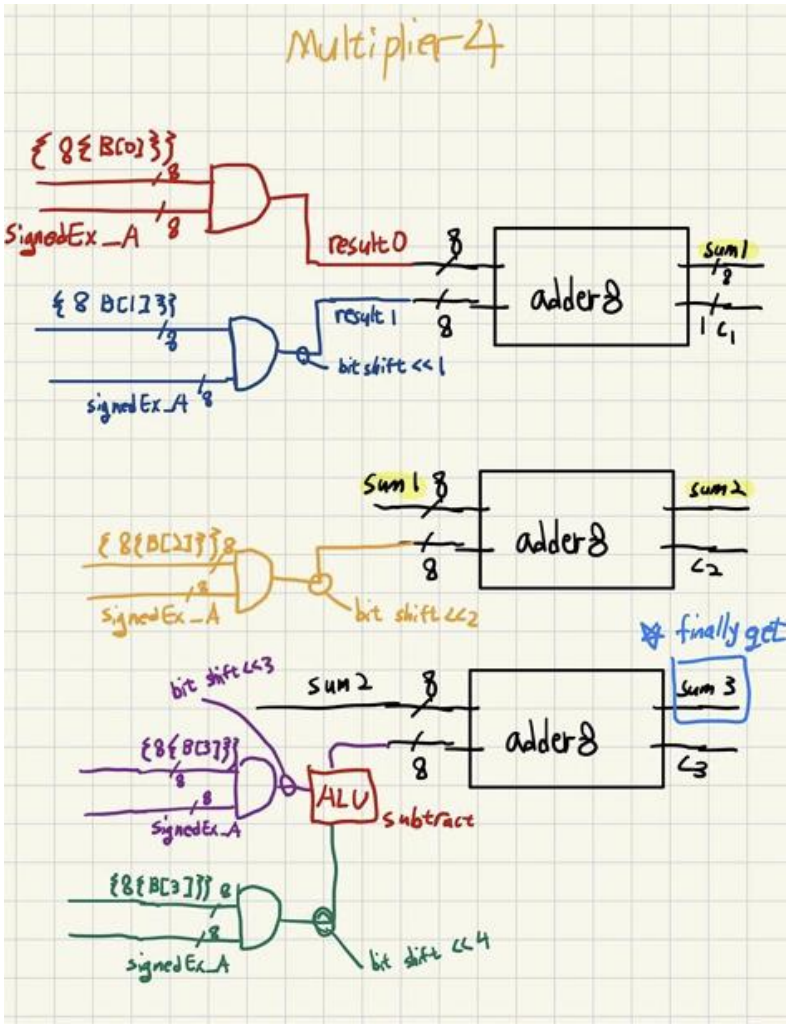


Fig 2.2 Multiplier Block Diagram

Multiplier의 Block Diagram을 살펴보면, B의 각 비트를 8비트로 replicate하고 AND 게이트를 이용하여 shift-and-add 알고리즘을 구현한 것을 확인할 수 있다. 이 설계에서는 B의 비트가 0인 경우 결과값이 0이 되고, B의 비트가 1인 경우 signedEx_A가 전달되도록 구성되어 있다.

각 AND 게이트의 출력은 B의 해당 비트 위치에 따른 A와의 부분곱을 나타내며, 이진수 체계의 자리수를 고려하여 적절한 shift가 적용된다. 또한, B[0]은 1의 자리이므로 shift 없이 그대로 사용되고, B[1]은 2의 자리로 1비트 shift가, B[2]는 4의 자리로 2비트 shift가 적용된다.

가장 중요한 부분은 MSB인 B[3]의 처리방식이다. B가 음수인 경우를 처리하기 위해 signedEx_A << 3 - signedEx_A << 4 연산을 수행한다. 이 연산은 +8 × signedEx_A에서 +16 × signedEx_A를 빼는 것으로, 결과적으로 -8 × signedEx_A가 된다. 이는 2의 보수 표현에서 MSB가 1일 때의 부호를 고려한 실제값을 반영하는 것이다. 이러한 설계를 통해 4비트 부호 있는 정수의 곱셈을 조합 논리로 구현할 수 있다.

-Various Simulation

시뮬레이션 결과를 확인하기 전에 Vivado 툴에서 제공하는 시뮬레이션 종류들을 이해해 보자.

첫번째로 Behavioral Simulation은 Verilog로 작성된 코드의 논리적 정확성을 검증하기 위한 기초적인 시뮬레이션 방법이다. 이 단계에서는 실제 하드웨어의 타이밍 특성이나 합성 과정을 고려하지 않고, 순수하게 설계된 논리 동작의 기능적 정확성만을 확인한다. 따라서 설계 초기 단계에서 알고리즘의 정확성과 기본적인 오류를 빠르게 발견할 수 있다는 장점이 있다.

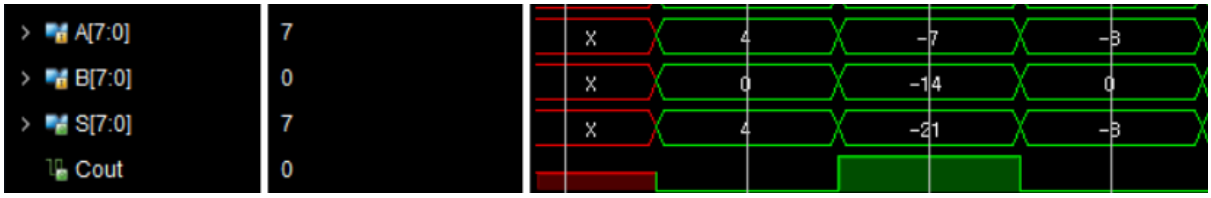
두번째로 Post-synthesis Simulation은 RTL 코드가 논리 합성 과정을 거쳐 생성된 게이트 수준의 넷리스트를 기반으로 수행되는 시뮬레이션이다. 이 과정에서는 RTL 코드가 실제 논리 게이트들로 변환된 결과를 검증하게 되어, 의도한 대로 회로를 구현했는지 확인할 수 있다. 이를 통해 합성 과정에서 발생할 수 있는 오류를 사전에 발견할 수 있다.

마지막으로 Post-implementation Simulation은 배치와 실제 배선까지 모두 완료된 최종 회로에 대해 실제 타이밍을 고려하여 수행하는 시뮬레이션이다. 이 단계에서는 사용할 실제 FPGA 보드의 물리적 특성 등이 모두 반영되어 실제 칩의 동작과 가장 유사한 결과를 얻을 수 있다. 특히 이 시뮬레이션에서는 setup time과 hold time을 이용한 slack 값들이 계산되며, 이러한 타이밍 정보를 통해 설계가 안정적으로 동작할 수 있는지 최종 검증할 수 있다.

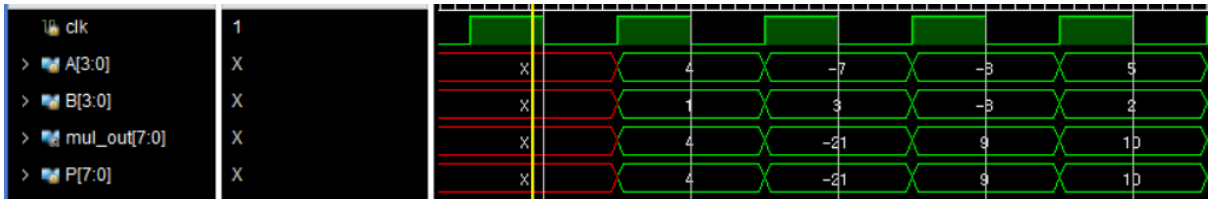
Behavioral Simulation 결과 분석

과제에서는 실제 FPGA 보드까지 사용하지는 않으니까, Behavioral Simulation으로 설계한 회로들이 논리적으로 올바르게 동작하는지 확인하도록 하였다.

Ripple Carry Adder



Multiplier



두 모듈의 input, output을 simulation에서 확인해본 결과 signed 정수에 대해서도 잘 동작하고 있음을 확인할 수 있었다.

결론

본 과제를 통해 기본적인 연산 소자인 HA와 FA의 동작 원리를 이해하고, 이를 활용한 8bit Ripple Carry Adder를 직접 구현해 보았다. Ripple Carry Adder는 구조가 단순하고, 구현이 용이하다는 장점이 있지만, carry가 각 단계마다 순차적으로 전달되기에 계산 속도에 한계가 있다는 단점도 존재한여 실제 고성능 프로세서에서는 다른 방식(Carry Look Ahead Adder) 사용하는 것으로 알고 있다.

이러한 설계 경험들을 통해, verilog의 기초문법, 연산기들의 동작 원리에 대해 기초 지식을 쌓을 수 있었고, 하위 모듈들이 상위 모듈에서 어떻게 주요한 역할을 하고 있음을 이해하였다. 이번 과제의 상위 모듈인 Ripple Carry Adder와 Multiplier 역시 추후, 다른 모듈에서 유용하게 사용될 것이라고 생각된다.

AI가속기설계

Tiny Neural Network 설계

과제 #2

건국대학교

202110410

조민우

서론

과제2에서는 Tiny Neural Network에서 추후 사용될 fully connected layer의 연산을 위해 multiply-and-accumulate(이하, MAC) unit을 설계하려한다.

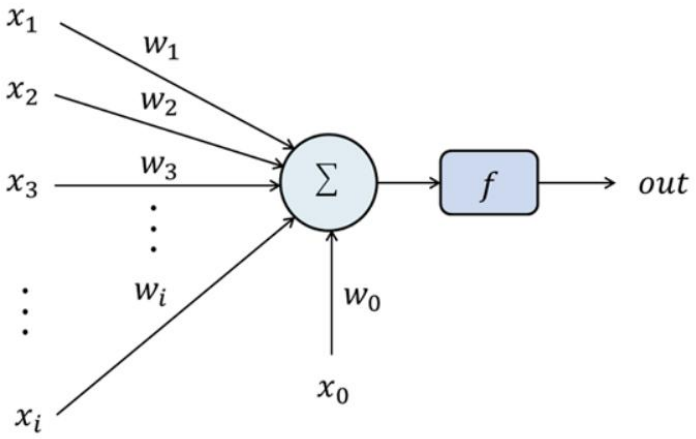


Fig 1.1 Fully Connected Layer

FC Layer는 입력 벡터 x와 가중치 벡터 w 사이의 MAC 연산 결과에 Activation Function을 적용하여 출력을 생성하는 구조이다.
이 연산은 수학적으로는 두 벡터의 내적으로 표현되며, 각 input vector와 weight vector를 곱한 뒤, 그 결과를 모두 더해 하나의 스칼라 값을 구한다.
즉, FC Layer의 연산은 다음과 같이 이해할 수 있다:

output = Activation($x_1 \cdot w_1 + x_2 \cdot w_2 + \ldots + x_n \cdot w_n$)

이러한 연산의 핵심은 곱셈과 덧셈을 반복적으로 수행하는 MAC 연산에 있으며, 이를 하드웨어로 구현하기 위해서는 효율적인 MAC 연산 모듈이 반드시 필요하다.

Cycle 1	Multiplication (c x e)	Previous accumulation if exists
Cycle 2	Multiplication (c x 1)	Accumulation M + (c x e) x 1
Cycle 3	Multiplication (4 x e)	Accumulation M + (c x 1) x 10
Cycle 4	Multiplication (4 x 1)	Accumulation M + (4 x e) x 10
Cycle 5	Next multiplication if exists	Accumulation M + (4 x 1) x 100

Fig 1.2 MAC Unit calculation flow

MAC unit은 앞서 제작한 multiplier와 adder 모듈을 이용하여 제작하고 Fig 1.2와 같은 flow로 4비트씩 쪼개어 연산을 진행한다(본론에서 자세히 언급). 또한, en이 high인 시점부터 5 cycles 뒤에 결과가 출력되며, 연산 중에는 busy 신호가 high로 유지된다.

본론

-signed, unsigned 여부를 고려한 multiplier

Mac8 unit은 8bit의 A, B를 input으로 받아 16bit의 output을 내보낸다. 그러나, 앞서 제작한 multiplier는 4bit의 A,B를 input으로 받아 8bit의 output을 내보냈으므로 이를 재사용하려면 다음 그림과 같이 8bits 곱셈을 4bit 곱셈으로 나누어서 진행하여야한다.

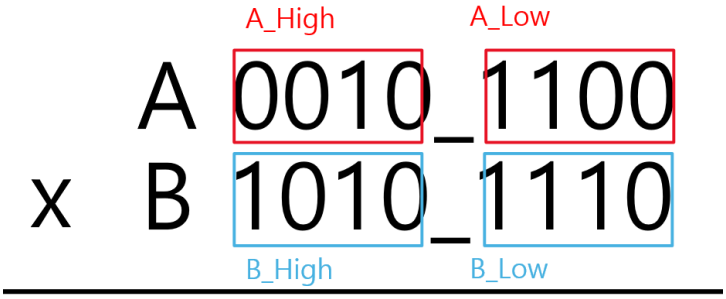


Fig 2.1 Multiplication

i) A_Low x A_Low (unsigned x unsigned)

ii) A_High x B_Low(signed x unsigned)

iii) A_Low x B_High(unsigned x signed)

iV) A_High x B_High (signed x signed) (기존 multiplier 기능)

또한, 각 case 별로 부호 고려 여부가 다르므로 여러 모듈을 나누어 선언할 필요가 있다.

코드에서는 concatenate와 삼항 연산자를 이용하여 MSB를 확인하여 부호처리를 진행하였다.

-Mac8

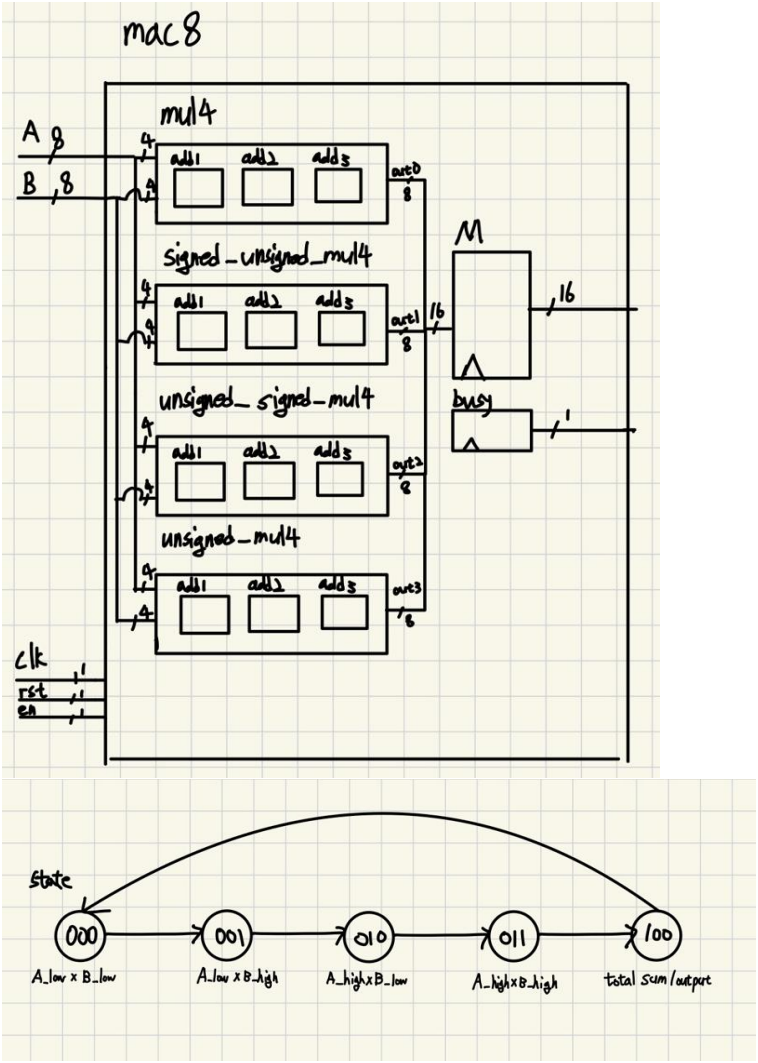


Fig 2.2 Mac8 block diagram & finite state machine

MAC unit 설계의 핵심은 FSM(Finite State Machine)이다. 앞서 만든 서로 다른 4개의 곱셈기를 이용하기 위해 과제의 조건대로 5개 상태로 나누어서 단계적으로 처리하도록 설계했다.

State 000 (IDLE): 평상시 대기 상태다. enable 신호가 들어오면 busy를 켜고 첫 번째 부분곱(A_low×B_low)을 저장한 다음 state 001로 넘어간다.

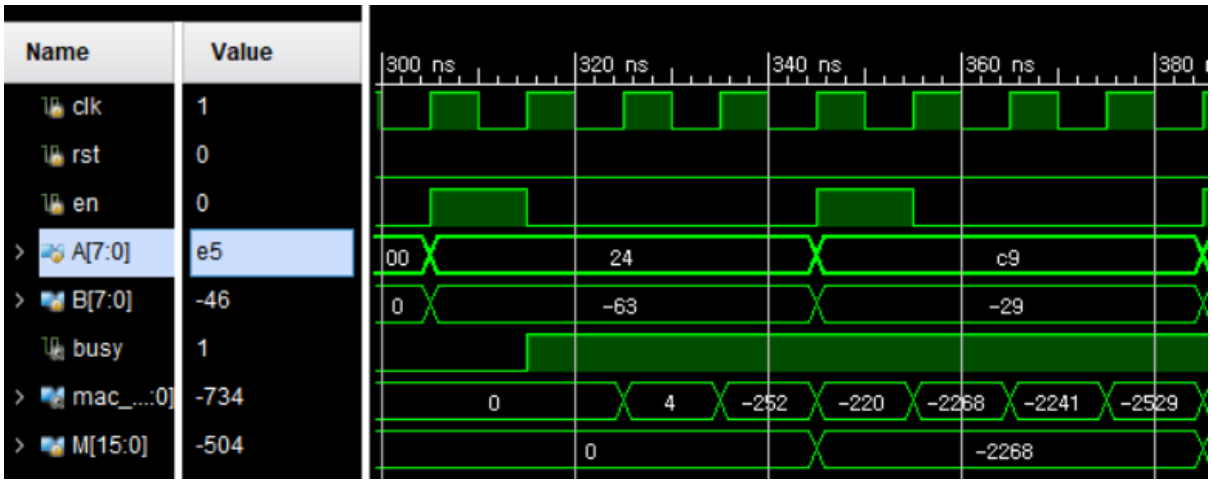
State 001: 두 번째 부분곱(A_low×B_high)을 계산해서 저장하고, 동시에 이전 누적값과 첫 번째 부분곱을 더한 결과를 출력한다.

State 002: 세 번째 부분곱(A_high×B_low)을 처리하면서 두 번째 부분곱까지 반영된 결과를 출력한다.

State 003: 네 번째 부분곱(A_high×B_high)을 처리하고 세 번째 부분곱까지의 누적 결과를 출력한다.

State 100: 모든 부분곱이 합쳐진 최종 결과가 나온다. 이 값을 total에 저장하고 출력으로 내보낸다. 만약 다음 enable이 또 들어오면 바로 다음 연산을 시작하고, 그렇지 않으면 busy를 끄고 IDLE로 돌아가게 된다.

Behavioral Simulation 결과 분석



가장 초기 연산인 0x24(36) * 0xc1(-63)을 예시로 계산과정을 확인해보자.

1) A_Low * B_Low 4*1 = 4 total: 4

2) A_Low * B_High 4*c(-4)*16(shift left 4) = -256 total: -252

3) A_High * B_Low 2*16(shift left 4)*1 = 32 total: -220

4) A_High * B_High c(-4)*2*16^2(shift left 8) = -2048 total: -2268

앞서 언급한 자리수에 따른 부호 연산 계산과정을 모두 만족함을 볼 수 있다. MAC unit은 이러한 연산을 이어나가며 값을 축적해나가는 과정을 반복하게 된다.

결론

-연속 연산 처리

실제 신경망에서는 MAC 연산이 연속으로 일어나게 된다. 그래서 하나 끝나고 다음 연산의 시작을 기다리면 너무 비효율적이다.

우리 설계에서는 state 100에 있을 때 새로운 en이 들어오면 바로 다음 연산을 시작하도록 했다. busy 신호도 연산이 진행되는 동안 계속 high로 유지해서 외부에서 상태를 확인할 수 있게 하였다. 이와 같이 파이프라인 방식으로 구현함으로써 throughput을 향상시킬 수 있었다.

-16비트 가산기의 필요성

누적 연산을 위해서는 16비트 가산기가 필요했다. 이는 8비트 가산기의 원리를 그대로 확장하면 된다. generate 문을 써서 15개의 full adder를 반복적으로 생성하는 방식으로 깔끔하게 구현할 수 있었다.

-MAC unit의 활용도

CNN에서 하는 convolution 연산도 결국 필터(weight)와 입력 데이터(input) 간의 MAC 연산의 연속이고, transformer 역시 자세히 공부해보지는 못했지만 교수님께서 말씀하신 바에 따르면 MAC 연산으로 이루어진다. 또, 딥러닝을 공부하다보니 backpropagation에서 gradient를 계산해서 loss를 구할 때 역시 MAC 연산이 많이 일어난다는 것을 알게 되었다. 결국 우리가 지금 만든 이 작은 MAC unit이 ChatGPT나 이미지 인식 모델 같은 AI 시스템의 가장 기본이 되는 구성요소임을 깨달았고, MAC unit의 최적화가 AI 알고리즘의 처리속도를 결정하는 주요 유닛이라는 것을 이해할 수 있었다.

AI가속기설계

Tiny Neural Network 설계

과제 #3

건국대학교

202110410

조민우

서론

과제 3에서는 지금까지 만든 MAC unit들을 16개씩 연결해서 4×4 Systolic Array를 output stationary 방식으로 만드는 것을 목표로 한다. 4×4 행렬 A와 B를 곱해서 C = A × B의 행렬곱을 계산하는 것이다.

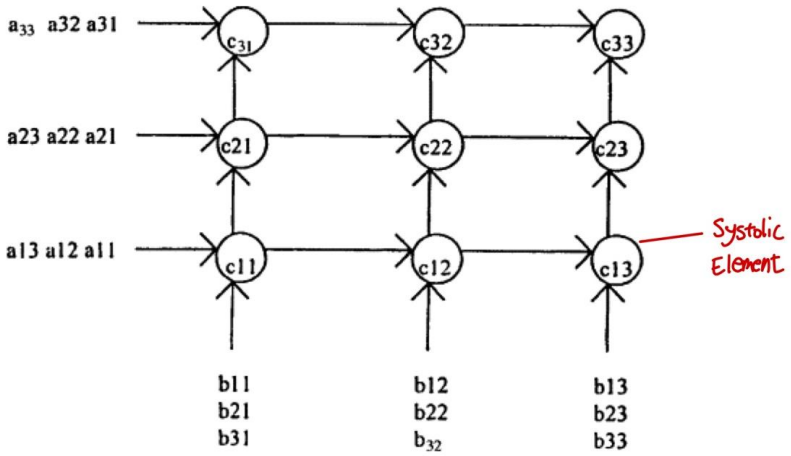


Fig 1.1 Systolic Array 구조

Systolic Array는 행렬 곱에 특화된 하드웨어 구조 Google사에서 제작한 TPU에서 대표적으로 사용되고 있다. Systolic Array는 각 연산 유닛(systolic element)이 데이터를 받아서 계산하고 다음 유닛으로 전달하여 data reuse와 pipelining을 통해 효율적인 연산을 진행하도록 도와주는 하드웨어 구조이다.

Systolic Array의 종류는 output stationary, weight stationary 두 종류가 존재하는데, 우리가 채택할 방식인 output stationary 방식부터 살펴보자.

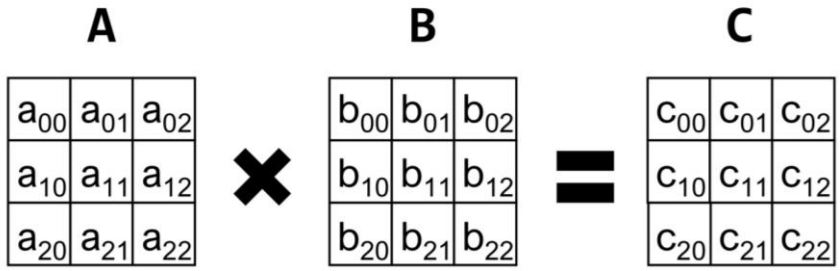


Fig 1.2 A x B = C 행렬곱 예시

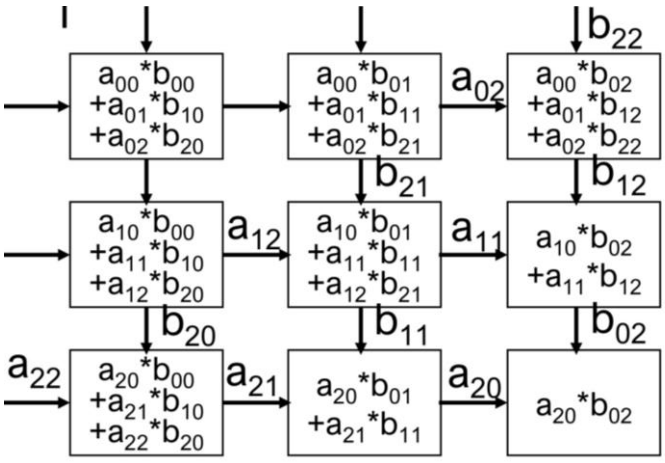


Fig 1.3 Systolic Array (Output Stationary) at cycle 5

Output Stationary는 출력 결과(output)가 PE 내부에서 정적으로 유지되며 계속 업데이트 되는 구조이다. 즉, 하나의 PE가 하나의 출력 값을 담당하며, 그 값을 누적합 형태로 계산한다. input A와 B의 데이터는 PE 사이를 지나가며 계산에 쓰이고 사라지지만, 출력 값은 해당 PE 안에 남아있고, 해당 원소의 연산이 끝나면 출력 값으로 쓰이게 된다.
이 구조는 output과 관련된 메모리 접근을 줄일 수 있어 MAC연산이 많은 구조에서 효율적이다.

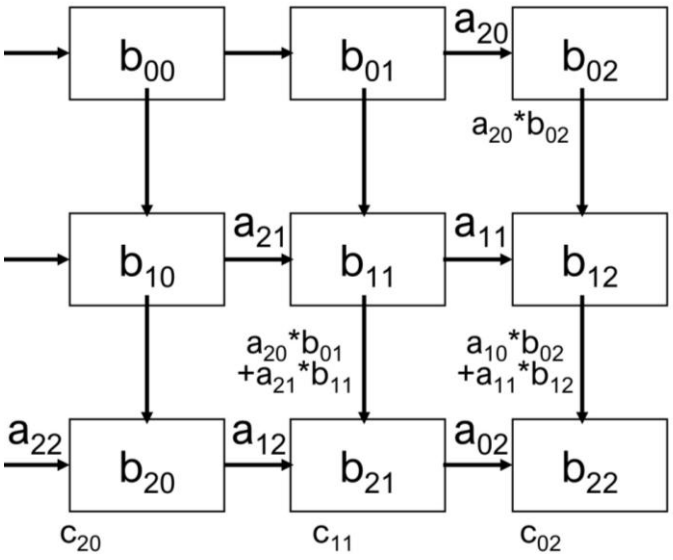


Fig 1.3 Systolic Array (Weight Stationary) at cycle 5

Weight Stationary는 Weight가 PE에서 내부에서 정적으로 유지되는 구조이다. 입력 A와 중간 결과인 부분합은 PE를 지나며 옆 PE로 전달되고, 고정된 Weight와 곱셈을 수행하게 된다. 이 방식은 Weight를 여러 번 재사용하는 경우 memory bandwidth를 줄일 수 있어 효율적이고, 특히 딥러닝에서 Weight가 일정 시간 동안 반복 사용될 때 많이 사용하게 된다.

본론

Systolic Element

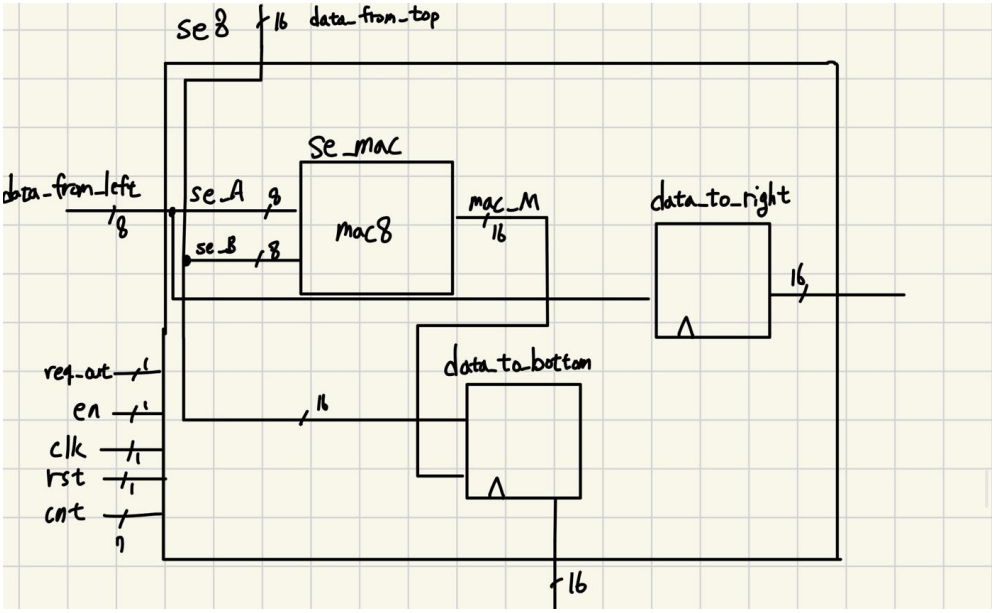


Fig 2.1 Systolic Element Block Diagram

4*4 행렬 곱셈을 하기위해 A,B 데이터를 7번 반복하여 testbench에서 각 element에 해당하는 값을 보내는데, 이를 se에서 mac을 통해 연산, 축적하게 된다. 4 x 4행렬이므로 각 element는 4번의 mac연산을 진행하게 된다. 왼쪽에서 들어오는 data_from_left는 행렬 A의 원소를 나타내며, 이 데이터는 내부 MAC 연산에 사용된 후 오른쪽으로 그대로 전달된다. 위쪽에서 들어오는 data_from_top은 행렬 B의 원소와 이전 단계의 partial sum을 포함하고 있으며, 하위 8비트는 MAC 연산의 B 입력으로 사용되고 전체 16비트 데이터는 아래쪽으로 전달된다.

Systolic Array

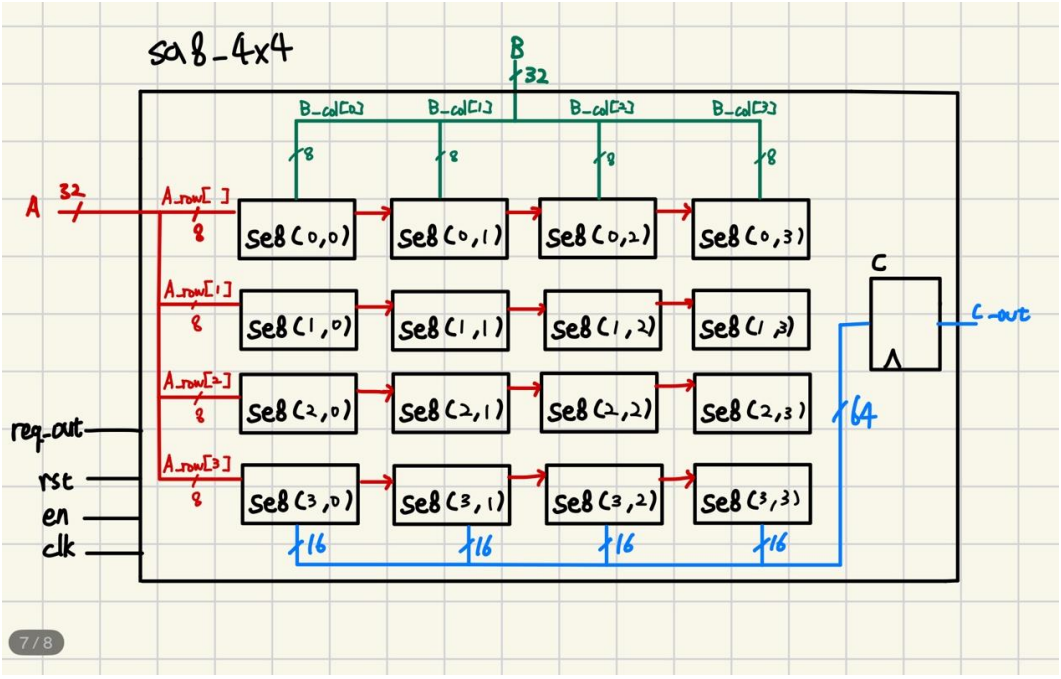


Fig 2.2 Systolic Array Block Diagram

sa8_4x4 모듈은 16개의 se8 모듈을 4×4로 배치하여 만든 systolic array이다. Input 행렬 A와 B는 각각 32비트로 표현되며, 8비트씩 4개의 원소로 분할되어 A_row와 B_col 배열에 저장된다. Systolic array의 systolic element는 cnt가 (i+j)*4 + 1부터 (i+j)*4 + 16 사이에 있을 때 enable된다. 이렇게 clock count에 맞추어 각 elements들의 연산을 조절할 수 있다.

Data flow를 살펴보면, 첫번째 열의 element들은 A_row에서 직접 데이터를 받고, 나머지 열의 element들은 왼쪽 element의 출력을 입력으로 받는다. 마찬가지로 첫 번째 행의 element들은 B_col에서 데이터를 받고, 나머지 행의 element들은 위쪽 element의 출력을 입력으로 받는다.

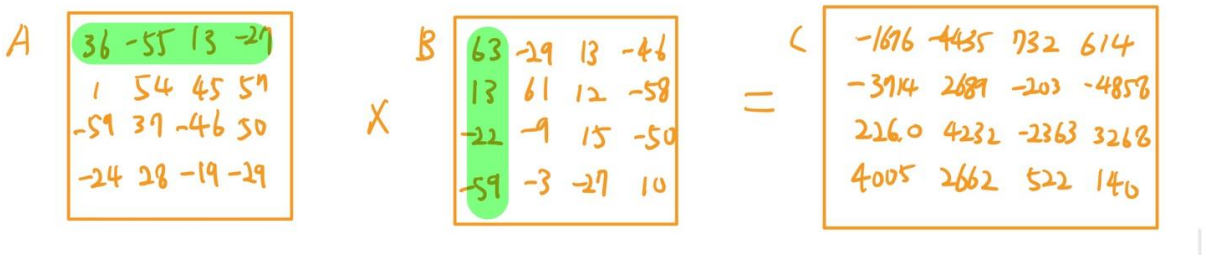
결과 출력은 cnt가 43~46 사이에서 순차적으로 이루어진다. cnt가 43일 때는 마지막 행인 3번째 행의 결과가, 44일 때는 2번째 행의 결과가, 45일 때는 1번째 행의 결과가, 46일 때는 0번째 행의 결과가 출력된다. 이러한 역순 출력은 systolic array의 데이터 흐름 특성상 아래쪽 행부터 연산이 완료되기 때문이다.

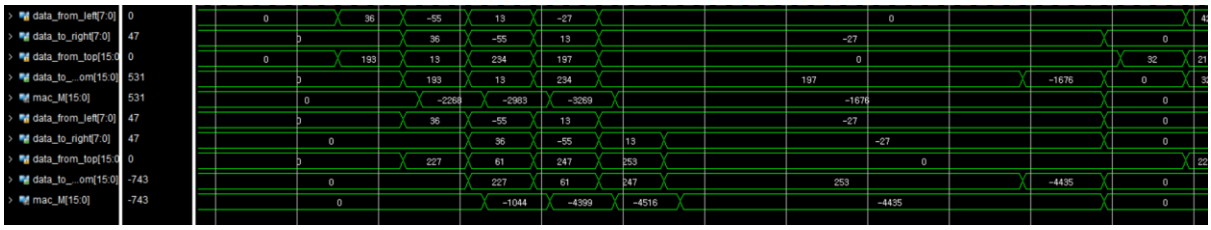
Behavioral Simulation 결과 분석

(파형을 분석할 때 유의할 점이 C_col이 64비트인데, B는 32비트이므로 B 행렬을 전달 받을 때는 unsigned 표현으로 읽히게 된다. 따라서, 파형에서 197은 -59, 253은 -3에 대응된다.)

예시

가장 먼저 연산되는 A, B 행렬의 연산 wire, reg 등을 확인해보자.





A_row와 B_col의 곱을 mac에서 진행하기 위해 output stationary 구조를 이용해 cycle 마다 값을 넘겨주는 것을 waveform에서 확인할 수 있다. 그리고, 최종 결과 값을 data_to_bottom을 재사용해 C행렬에 넘겨주게 된다.

조금 더 구체적으로 살펴보면 data_to_right reg는 data_to_left로부터 A(36,-55,13,-27)의 0행에 해당하는 정보들을 넘겨 받아 이를 바탕으로 mac연산을 진행하고 있다. data_from_top 역시 data_to_top으로 B의 열을 전달하는 모습이다. 이러한 식으로 값을 넘겨주고 se 모듈 내에 있는 mac에서 연산 및 축적을 진행하고 sa에서 se 모듈을 이용해 req_out 신호를 기점으로 C행렬을 완성하게 된다.

결론

Systolic array는 특정 연산에 특화된 하드웨어 구조로, 데이터가 규칙적으로 흘러가면서 병렬 처리가 이루어진다. TPU에 주로 사용된다고 알려져 있지만, 다른 GPU도 마찬가지로 특정 연산에 특화된 구조를 이용하여 연산 처리 속도를 증가시키고 있을 것이라 생각 된다.

AI가속기설계

Tiny Neural Network 설계

과제 #4

건국대학교

202110410

조민우

서론

과제 4는 과제 3에서 만든 4x4 output stationary systolic array를 이용하여 8x8 행렬 곱을 4x4 Tile로 연산하는 것을 목표로 한다. 이전까지의 과제와는 달리 input 행렬인 A, B는 memory에 저장되어있고 controller 모듈을 이용하여 read해야한다.

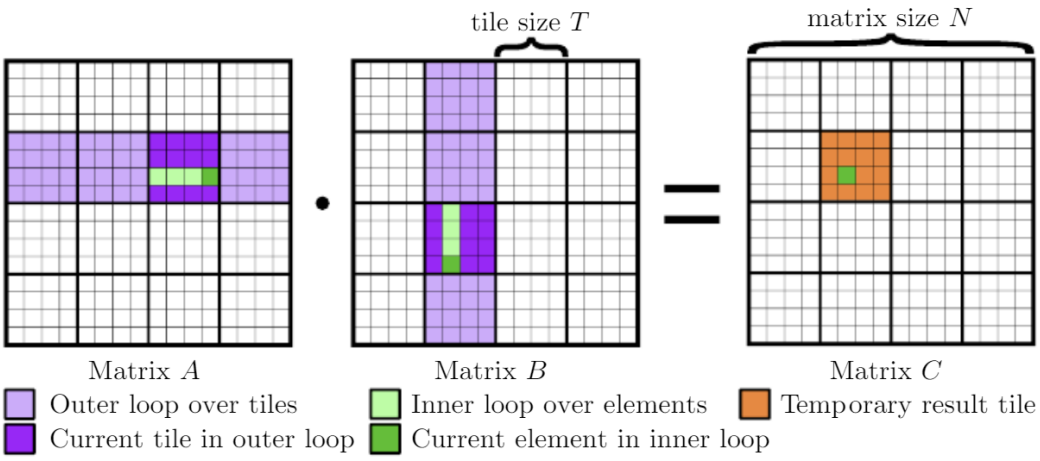


Fig 1.1 Tiled Matrix Multiplication

Tiled Matrix Multiplication은 큰 행렬을 작은 타일로 나누어 처리함으로써 메모리 대역폭을 효율적으로 활용하고, 캐시의 locality를 향상 시켜 가속기의 성능을 향상시킬 수 있는 중요한 기법이다. 8×8 행렬 곱셈 C = A × B를 4×4 Tile로 쪼개어 연산한다고 생각해 보자.

C[0,0] = A[0,0] × B[0,0] + A[0,1] × B[1,0]

C[0,1] = A[0,0] × B[0,1] + A[0,1] × B[1,1]

C[1,0] = A[1,0] × B[0,0] + A[1,1] × B[1,0]

C[1,1] = A[1,0] × B[0,1] + A[1,1] × B[1,1]

여기서 각 타일은 4×4 크기이며, 총 8번의 4×4 Matrix Multiplication과 4번의 MAC 연산이 필요하다. 과제 4에서는 이러한 타일링 연산을 지원하기 위해 Controller, Memory Interface, Top 모듈을 구현하려한다.

본론

Mem_behavior

Memory Behavior 모듈은 실제 하드웨어의 메모리 동작을 시뮬레이션하는 behavioral 모

델이다. 이 mem_behavior 모듈은 외부 파일에서 초기 데이터를 읽어와 메모리 배열에 저장하고, read/write을 수행할 수 있다.

Controller

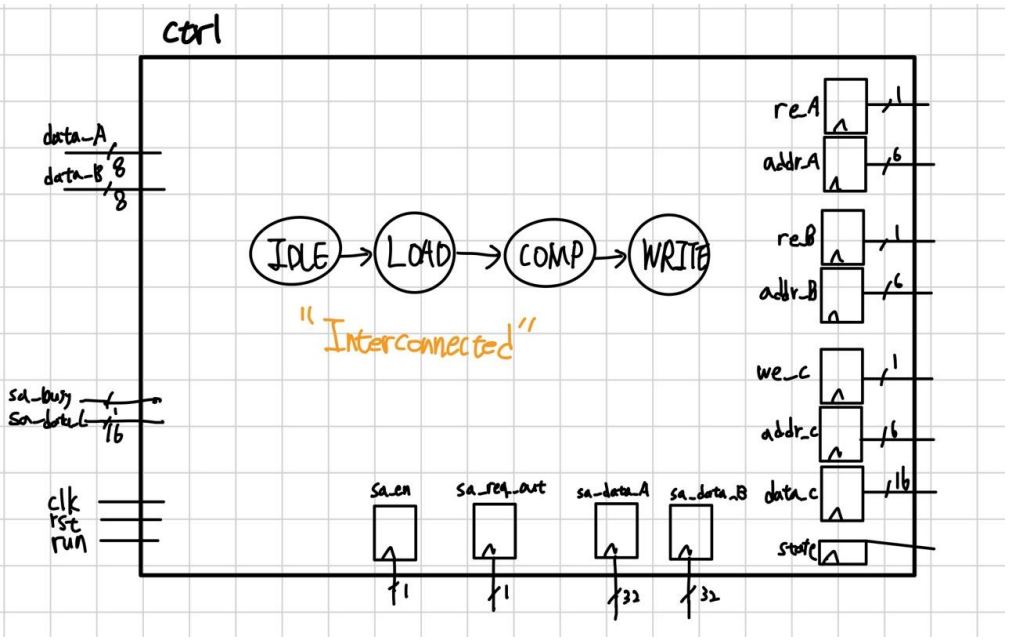


Fig 2.1 Controller Block Diagram

Controller는 8×8 Tiled Matrix Multiplication을 제어하는 핵심 모듈로 4개의 STATE(IDLE, LOAD, COMP, WRITE)를 가진 FSM으로 구현되었다.

IDLE	말그래도 IDLE 상태이다. run 신호가 들어오면 모든 변수를 초기화하고 LOAD 상태로 천이하게 된다. Tile idx(tile_i, tile_j, tile_k), 카운터, result_C, SA input 배열 A, B 등을 모두 초기화한다.

LOAD	메모리로부터 A, B 타일 데이터를 읽어오는 상태이다. 18 사이클 동안 진행되며, 첫 16 사이클에서는 4×4 타일의 16개 원소를 읽는다.
	주소 계산이 핵심인데, A 행렬은 (row-wise)로 읽고, B 행렬은 (column-wise)로 읽어야 한다:
	addr_A <= (tile_i * 4 + ctrl_cnt / 4) * 8 + (tile_k * 4 + ctrl_cnt % 4);
	addr_B <= (tile_k * 4 + ctrl_cnt % 4) * 8 + (tile_j * 4 + ctrl_cnt / 4);
	메모리 읽기에 delay가 있어 2 사이클 후부터 데이터를 저장한다.

	Systolic Array에서 4×4 행렬 곱셈을 수행하는 상태이다. 48사이클 동안 진행

COMP	되며, 첫 16 사이클에서는 4번의 en signal을 이용해 SA를 구동하게 된다. 41 사이클에서 req_out이 1이 되어 결과를 출력하고, 42~46 사이클에서 4×4 result_C 행렬을 받아 누적한다.
WRITE	계산 완료된 결과를 메모리에 저장하는 상태이다. 16사이클 동안 4×4 결과 타일을 메모리에 쓴다. 주소 계산은 Output 타일 C의 주소를 고려하여 연산 된다: addr_C <= (tile_i * 4 + ctrl_cnt / 4) * 8 + tile_j * 4 + (ctrl_cnt % 4);

Top

Top 모듈은 Controller와 Systolic Array를 연결하는 최상위 모듈이다. 외부 memory 인터페이스와 내부 SA 인터페이스 사이의 신호 변환과 제어를 담당한다. 주요 기능으로는 Controller로부터의 메모리 제어 신호를 top 모듈의 output 포트로 연결하여 외부 메모리 인터페이스를 제공하고, Systolic Array의 리셋 신호를 생성한다.

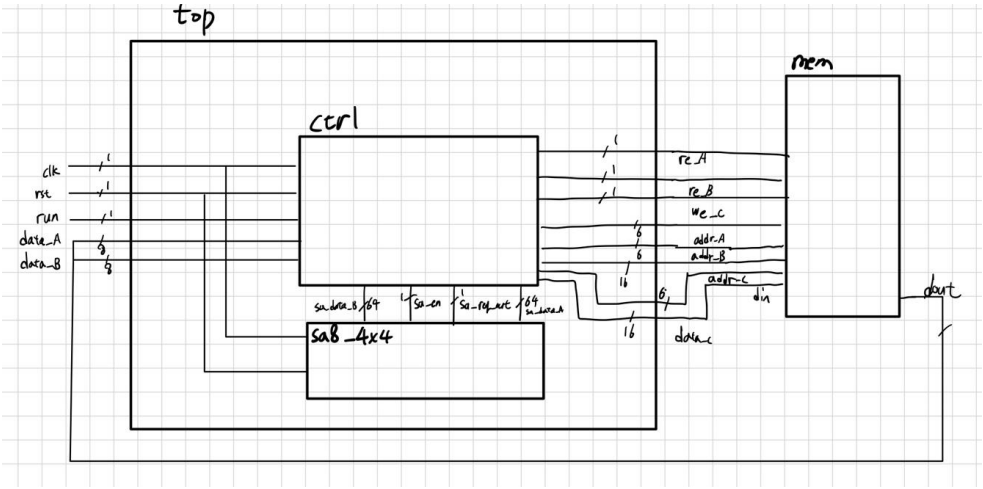
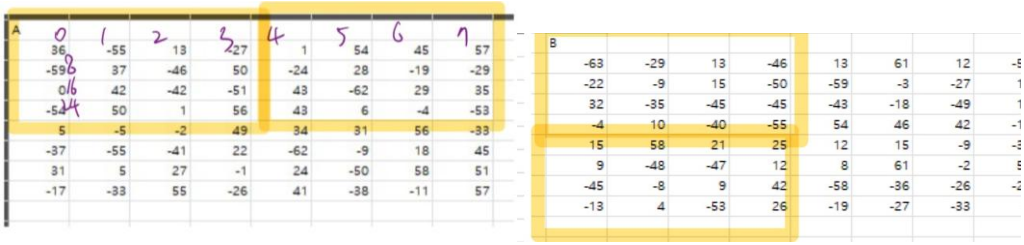


Fig 2.2 TOP module Block Diagram

Behavior Simulation Result

LOAD

앞선 과제와 달리 사용할 data들은 memory에서 가지고 오는 것으로 A는 row-wise, B는 col_wise로 memory address index를 설정해주어야한다.



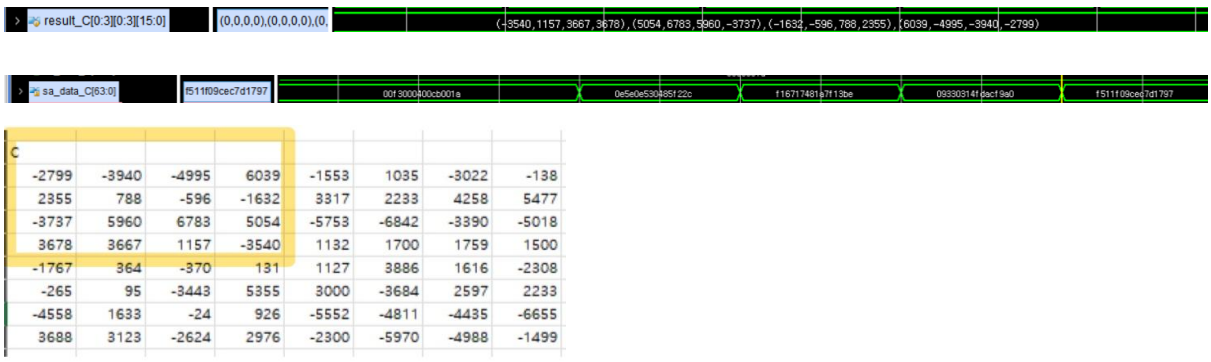
Reference와 비교해보면 적절한 mem index와 data값이 선택되었음을 확인해볼 수 있다.

LOAD state에서는 이 data_A, data_B 값들을 load하고 sa_A, sa_B reg에 저장해두어 COMP 단계에서의 연산을 준비한다.

COMP

앞서 load한 data들을 Systolic array에 전달하여 40 cycles 동안 4x4 matrix 연산을 진행하고 output을 result_C reg에 저장한다.

sa_data_A,B port에 연결하여 SA로 값을 넘겨줄 때, 32비트로 확장하여 값을 넘겨주어야 하며 각 element에 적절한 값을 넘겨주도록 7cycles이 필요하다.



2번의 LOAD->COMPUTE이 완료되었을 때의 sa_data_c(HEX)와 result_C에 저장된 값이다. 이를 reference_C와 비교하면 적절한 값이 선택되었음을 확인할 수 있다. 또, 주의할 점이 우리가 sa를 만들 때, 하위 row부터 읽어오도록 구현하였었기에 WRITE시 이를 고려하여 역순으로 넣어주어야함을 기억해야한다.(코드로 구현)

WRITE

WRITE state에서는 앞서 LOAD->COMP->LOAD->COMP 과정을 거치며 result_C에 저장해둔 결과를 data_C port에 연결하고 addr_C를 설정해주어 mem에 저장한다.

addr_C는 c[0][0]에 해당하는 값에 저장될 수 있도록 addr_A와 같이 row-wise로 idx가 증

가한다. (하지만 A와 달리 tile_i,j 이용)

C

-2799	-3940	-4995	6039	-1553	1035	-3022	-138
2355	788	-596	-1632	3317	2233	4258	5477
-3737	5960	6783	5054	-5753	-6842	-3390	-5018
3678	3667	1157	-3540	1132	1700	1759	1500
-1767	364	-370	131	1127	3886	1616	-2308
-265	95	-3443	5355	3000	-3684	2597	2233
-4558	1633	-24	926	-5552	-4811	-4435	-6655
3688	3123	-2624	2976	-2300	-5970	-4988	-1499

data_C를 통해 전달되는 값들을 reference_C와 비교해보면 4x4 tile의 data들이 적절히 전달되고 있음을 확인해 볼 수 있다.

결론

이번 과제를 통해 AI 가속기 설계에서 메모리 구조에 대한 이해가 중요함을 확연하게 체감할 수 있었다. 데이터를 효율적으로 읽고 쓰는 것이 전체 성능에 미치는 영향을 이해했으며, 특히 주소 계산과 clock에 동기화하는 것이 쉽지 않음을 깨달았다.

또한, 타일링은 단순히 큰 행렬을 작은 조각으로 나누는 것이 아니라, 메모리 대역폭과 하드웨어 자원의 제약을 극복하는 핵심 기법이며, 실제 GPU나 TPU에서도 이와 유사한 타일링 전략을 사용하여 대규모 행렬 연산을 처리하기에 이러한 시스템을 만들어본 것이 큰 도움이 되었을 것이라 생각한다.

이번 과제는 지금까지 만든 모듈들을 통합하여 행렬 곱셈기를 구현하는 과정이었다. 이를 통해 AI 가속기의 전체적인 아키텍처와 각 구성요소 간의 상호작용을 Low-level 언어로 이해할 수 있었으며, 실제 하드웨어 설계에서 고려해야 할 다양한 요소들을 경험할 수 있었다.

AI가속기설계

Tiny Neural Network 설계

Final Project

건국대학교

202110410

조민우

서론

이번 프로젝트는 앞서 과제1~4에서 만든 모듈들을 이용하여 합성 신경망 시스템(Tiny Neural Network)을 만드는 것이다.

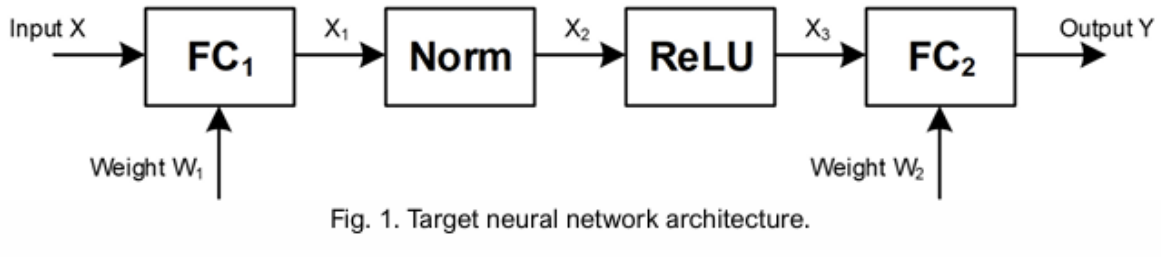


Fig. 1. Target neural network architecture.

Fig 1.1 Tiny Neural Network

구현하려는 네트워크는 FC layer 1, normalization, activation(ReLU), FC layer 2로 구성되어 있다. (바이어스 항은 무시) 일반적으로 FC layer는 하나의 벡터를 input으로 사용하지만, 이 프로젝트에서는 배치를 처리하는 것으로 가정하며, W-first (WX = Y) 방식을 이용하여 연산을 진행한다. Norm layer는 element-wise로 연산하며, 각 element를 32로 나누는 것으로 대체한다. ReLU 역시 element-wise로 연산한다. 그리고, 각 layer에서의 input/output의 비트 수는 아래의 block diagram을 참고하면 된다.

본 프로젝트에서는 batch가 8인 경우와 batch가 16인 경우를 하나의 하드웨어에서 구동하도록 설계해야하며, 모델을 검증할 테스트 벤치도 직접 작성하여야한다.

본론

우선적으로 하드웨어 구조와 기능 등을 살펴본 후, 구체적인 dataflow는 waveform과 함께 확인해보자.

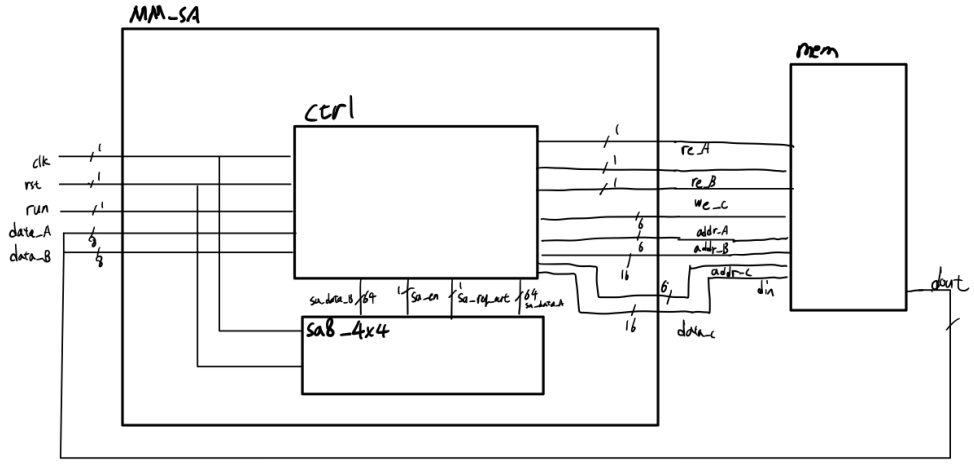


Fig 2.1 MM_SA(FC_layer) Block Diagram

MM_SA

MM_SA는 앞서 만든 4x4 Systolic Array를 이용하여 FC layer를 구현한 상위 모듈이다.

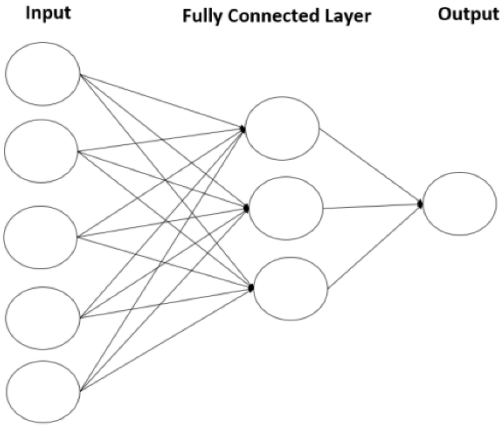


Fig 2.2 FC layer graph representation

FC layer는 input x와 가중치 w의 곱의 누적합을 output y로 이용한다. Fig 2.2에 표현된 node와 edge는 input과 weight의 곱으로 표현한다. 이러한 곱 연산의 반복은 행렬 곱으로 대체할 수 있다. 그렇기에 8x8 행렬 곱이 필요한 본 과제에서는 앞서 제작한 4x4 systolic array를 이용하여 MAC연산을 구현할 수 있다. 다만, 우리가 앞서 제작한 systolic array는 4x4이고 실제로 구현해야할 행렬 곱은 8x8이기에 앞선 과제4에서 활용한 Tiling 기법을 이용하여 8x8 행렬곱을 진행한다.

Normalization & ReLU

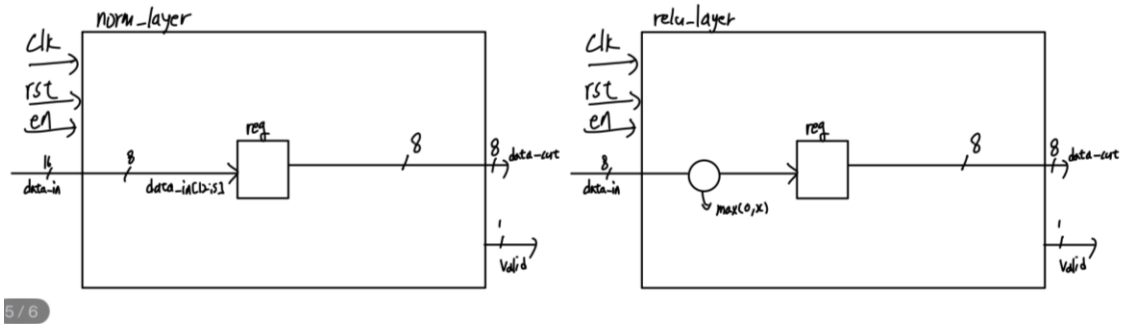


Fig 2.3 Norm_layer & ReLU_layer Block Diagram

Normalization은 추론 시 입력 데이터나 중간 특성 맵의 값 범위를 일정하게 유지하여 연산의 안정성을 높인다. ReLU 함수는 추론 과정에서 학습된 특성을 유지하면서 불필요한 연산을 제거한다. 더구나, Sparsity 메모리 구조에서는 0을 저장하지 않는 방식으로 구

현될 수 있기에 ReLU는 Tiny Neural Network의 장점을 살릴 수 있는 활성화함수이다.

Normalization 모듈(Norm_layer)부터 살펴보면 입력 데이터의 분포를 정규화하는 과정을 진행한다. 과제에서는 데이터들을 32로 나누는 것으로 정규화를 진행하였으며, 나는 LSB 5비트를 잘라내는 방식으로 구현하였다.

ReLU는 비선형성을 도입하는 레이어이다. 입력값이 양수일 때는 그대로 출력하고 음수일 때는 0으로 출력하는 ReLU 함수의 특성을 하드웨어적으로 구현하였다. 이러한 연산은 기울기 소실 현상 해결과 연산 효율성을 높이는데 도움을 준다.

Tiny Neural Network

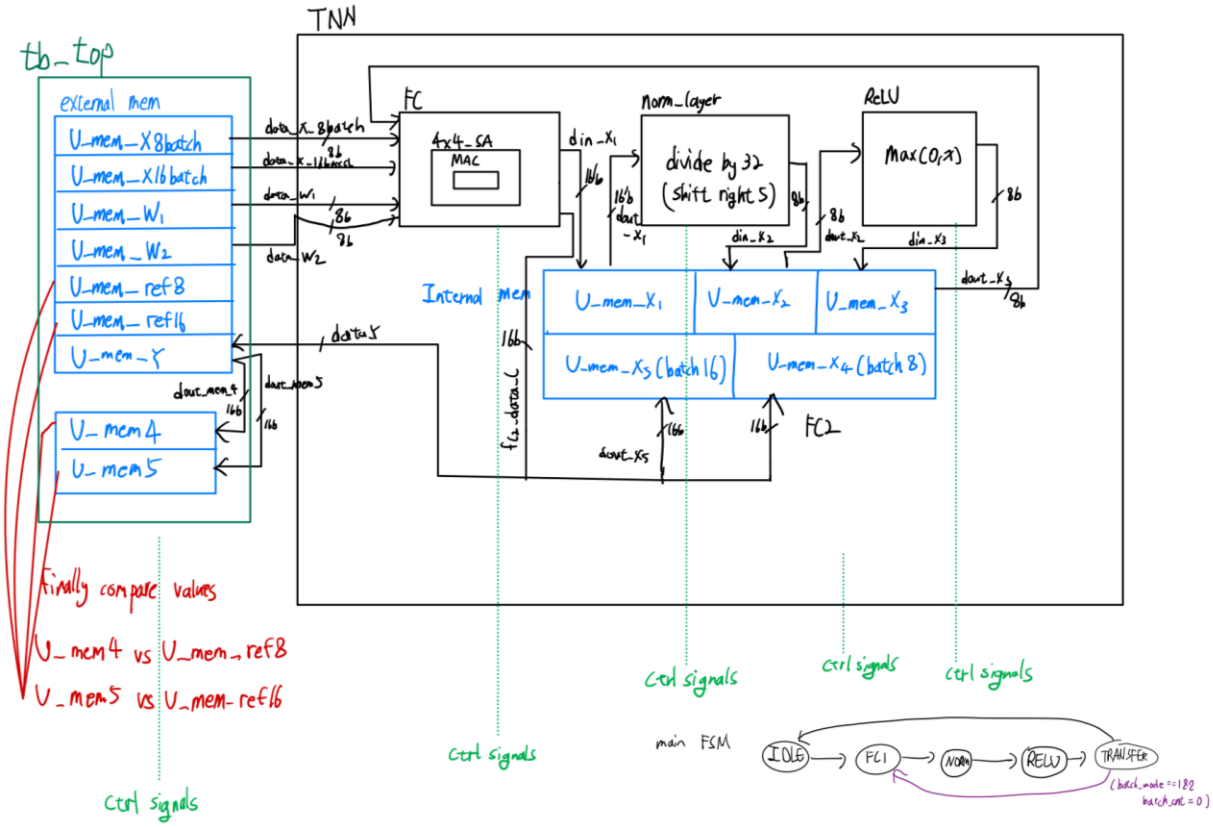


Fig 2.3 Tiny Neural Network Block Diagram

이제 최종적으로 지금까지 만든 모듈들을 이용하여 하나의 시스템을 모듈화하여 구성해야한다. 서론에서 언급한 Fig1.1의 dataflow를 다시 짚어보면 input –FC1–Norm–ReLU-FC2–output의 흐름을 가지고 있다.

Waveforms & Analysis

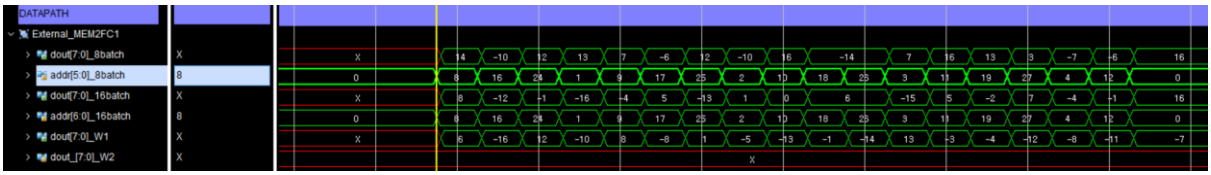
<Tiny Neural Network Dataflow in 8 batch>

(dataflow 이해와 가독성을 위해 각 state 진행의 **가장 앞 16 elements waveform**을 첨부하였습니다.)

2.2.1 U_mem_X, W1 -> fc1

14	-10	12	13	-10	4	10	-1
7	-6	12	-10	15	9	8	-16
16	-14	-14	7	-15	12	-6	6
16	13	3	-7	5	2	-16	8
-6	16	-2	6	5	-9	4	12
7	15	-4	-9	9	7	3	-3
-6	6	5	12	14	-2	16	13
-11	16	9	15	7	5	5	6

6	8	-13	-3	-11	6	6	13
-16	-8	-1	-4	-7	15	-2	-4
12	1	-14	-12	6	-16	-12	13
-10	-5	13	-8	-3	5	-2	8
-6	-2	4	3	12	15	0	-14
-8	-9	11	8	-6	16	-12	6
-14	-14	0	6	16	15	-1	-9
7	2	-1	9	14	-4	8	-8

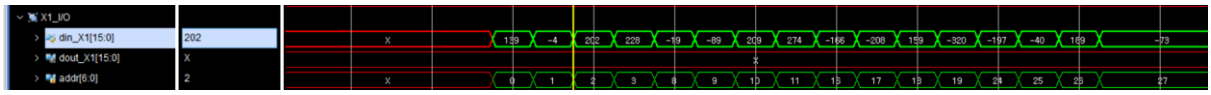


(4x4 Tile addressing for Matrix Multiplication)

2.2.2 fc1 -> U_mem_X1 -> NORM

■ fc1 -> mem

139	-4	202	228	-306	-329	-393	-474
-19	-89	209	274	-169	-152	-359	-399
-166	-208	159	-320	77	-51	179	273
-197	-40	189	-73	5	113	77	-114
-49	97	-447	-324	270	-107	636	236
-125	263	-2	43	209	590	400	341
-132	-314	188	104	144	-33	-20	-253
564	91	426	161	-376	-180	-397	-131



(4x4 Tile addressing for Matrix Multiplication)

■ mem -> NORM

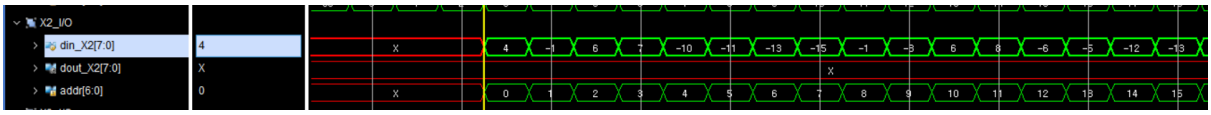
139	-4	202	228	-306	-329	-393	-474
-19	-89	209	274	-169	-152	-359	-399
-166	-208	159	-320	77	-51	179	273
-197	-40	189	-73	5	113	77	-114
-49	97	-447	-324	270	-107	636	236
-125	263	-2	43	209	590	400	341
-132	-314	188	104	144	-33	-20	-253
564	91	426	161	-376	-180	-397	-131

(ascending addressing)

2.2.3 NORM -> U_mem_X2 -> RELU

■ **NORM -> mem**

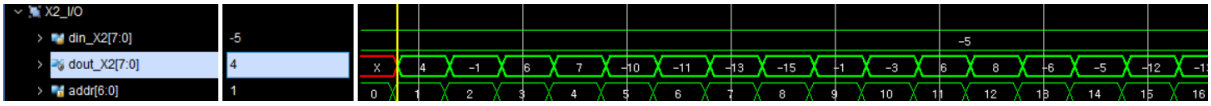
4	-1	-6	-7	-10	-11	-13	-15
-1	-3	6	8	-6	-5	-12	-13
-6	-7	4	-10	2	-2	5	8
-7	-2	5	-3	0	3	2	-4
-2	3	-14	-11	8	-4	19	7
-4	8	-1	1	6	18	12	10
-5	-10	5	3	4	-2	-1	-8
17	2	13	5	-12	-6	-13	-5



(ascending addressing)

■ **mem -> RELU**

4	-1	-6	-7	-10	-11	-13	-15
-1	-3	6	8	-6	-5	-12	-13
-6	-7	4	-10	2	-2	5	8
-7	-2	5	-3	0	3	2	-4
-2	3	-14	-11	8	-4	19	7
-4	8	-1	1	6	18	12	10
-5	-10	5	3	4	-2	-1	-8
17	2	13	5	-12	-6	-13	-5

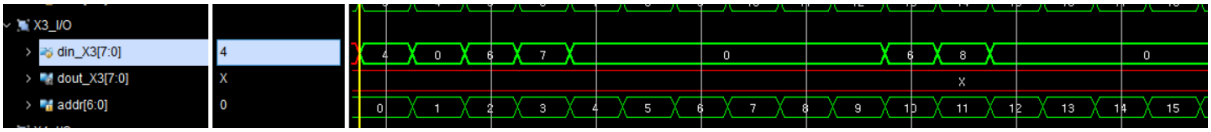


(ascending addressing)

2.2.4 RELU -> U_mem_X3 -> fc2

■ **RELU -> mem**

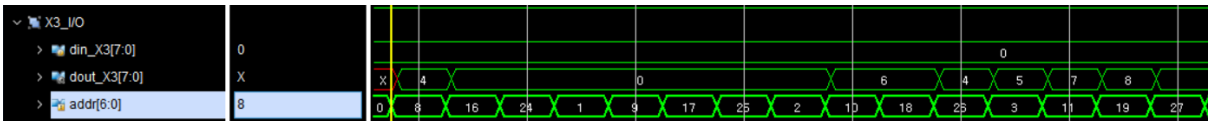
4	0	6	7	0	0	0	0
0	0	6	8	0	0	0	0
0	0	4	0	2	0	5	8
0	0	5	0	0	3	2	0
0	3	0	0	8	0	19	7
0	8	0	1	6	18	12	10
0	0	5	3	4	0	0	0
17	2	13	5	0	0	0	0



(ascending addressing)

■ **mem -> fc2**

4	0	6	7	0	0	0	0
0	0	6	8	0	0	0	0
0	0	4	0	2	0	5	8
0	0	5	0	0	3	2	0
0	3	0	0	8	0	19	7
0	8	0	1	6	18	12	10
0	0	5	3	4	0	0	0
17	2	13	5	0	0	0	0

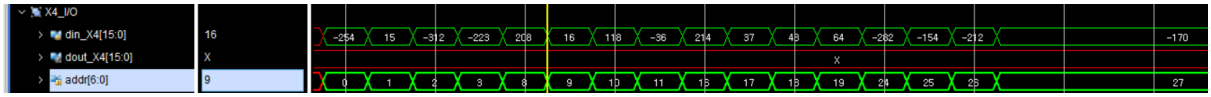


(4x4 Tile addressing for Matrix Multiplication)

2.2.5 fc2 -> U_mem_X4 -> tb_top

■ **fc2 -> mem**

-254	15	-312	-223	-118	222	-137	5
208	16	118	-36	146	-156	270	154
214	37	43	64	-50	-27	-26	-83
-282	-154	-212	-170	-38	-276	-212	-148
-119	-69	-195	-127	-136	-75	-180	-141
-128	70	-106	29	80	96	309	228
275	6	217	159	-148	30	-239	-10
186	111	172	85	30	198	136	83



(4x4 Tile addressing for Matrix Multiplication)

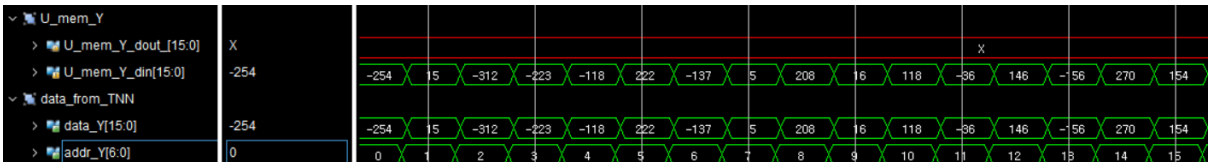
■ **mem -> wire(data_Y)**

-254	15	-312	-223	-118	222	-137	5
208	16	118	-36	146	-156	270	154
214	37	43	64	-50	-27	-26	-83
-282	-154	-212	-170	-38	-276	-212	-148
-119	-69	-195	-127	-136	-75	-180	-141
-128	70	-106	29	80	96	309	228
275	6	217	159	-148	30	-239	-10
186	111	172	85	30	198	136	83

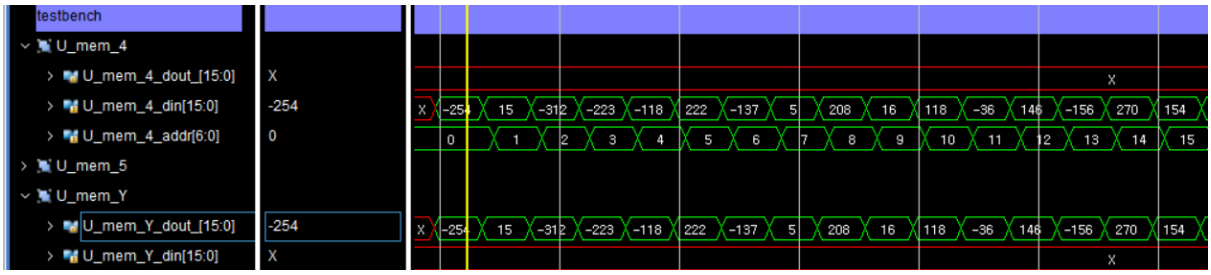
(mem_x4 values are connected to data_Y(output wire) and stored in U_mem_Y in the testbench.)

<Testbench Dataflow in 8 batch>

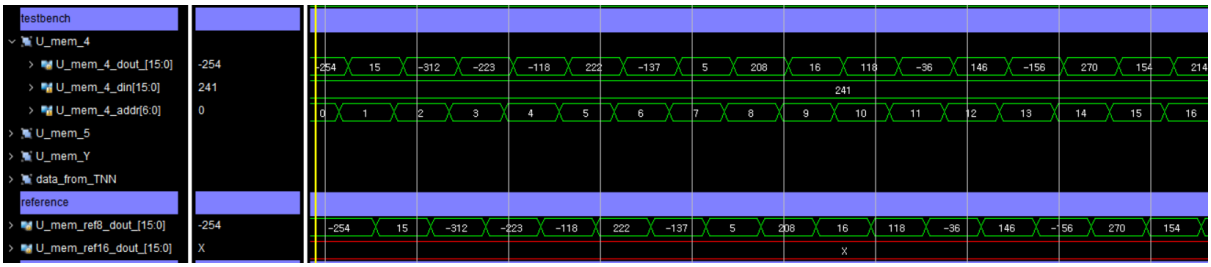
■ TNN -> tb_top (U_mem_Y)



■ U_mem_Y -> U_mem_4 (copy data mem_Y to mem_4)



■ Finally compare results with reference



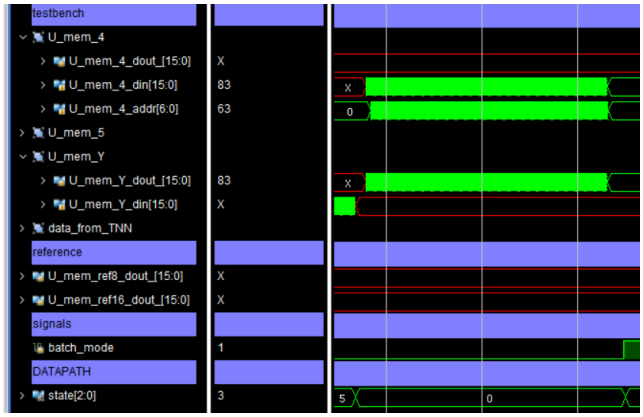
<Tiny Neural Network Dataflow in 16 batch>

1.2.1 IDLE(batch0~7)

(dataflow 이해와 가독성을 위해 16-batch는 state별로 waveform을 첨부하였습니다.)

8	-1	-16	-5	-2	-8	-14	-8
-4	-13	6	12	9	-8	-7	-9
0	6	-1	-10	0	14	9	-10
5	7	6	-2	16	-13	-11	-7
-1	-13	9	-14	-9	13	-16	6
-16	-15	7	-5	-13	1	-12	-4
5	1	-13	-7	10	14	-9	13
14	2	8	-8	14	2	9	1
-12	-16	6	2	-15	-8	12	14
5	1	-7	-11	10	8	-5	15
6	-15	-11	-16	2	10	-6	0
-2	-4	-13	7	-14	16	11	-16
16	-1	0	-3	-1	-10	5	-10
12	-6	-15	3	0	16	6	-16
-12	9	9	10	3	-8	-13	0
7	7	-4	12	-13	-11	-10	-14

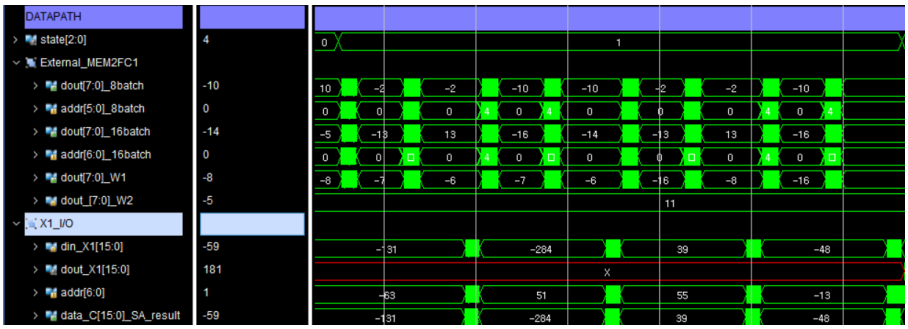
■ **IDLE**



(Wait until the values(batch8) are copied from U_mem_Y to U_mem_4, wait for batch_mode signal)

2.2.2 FC1(batch0~7)

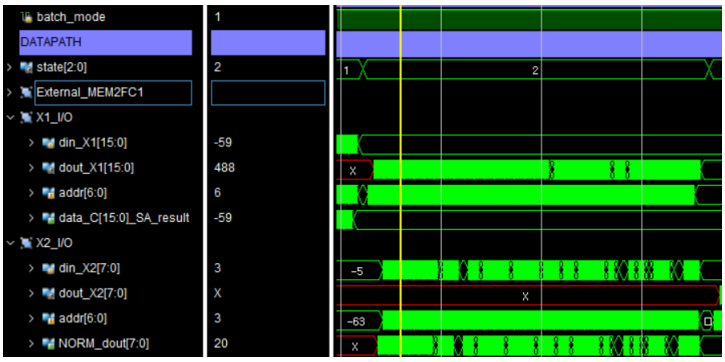
■ **external mem -> FC1 -> U_mem_X1**



(In the FC1 state, we can observe that the input is received from external memory, the results are stored in U_mem_X1, and the SA output is being computed.)

2.2.3 NORM(batch0~7)

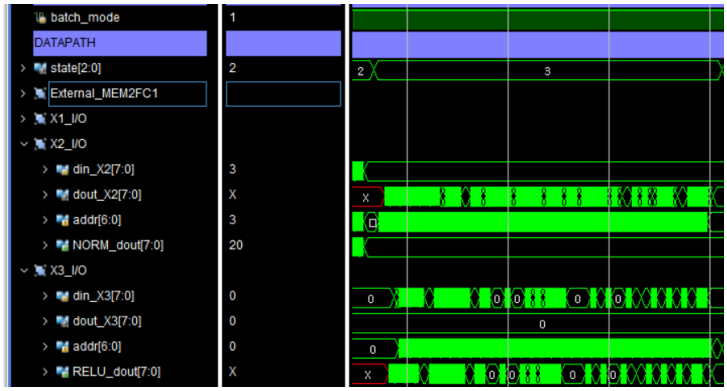
■ **U_mem_X1 -> NORM -> U_mem_X2**



(In the NORM state, we can observe that the input is received from U_mem_X1, the results are stored in U_mem_X2, and the NORM output is being computed.)

2.2.4 RELU(batch0~7)

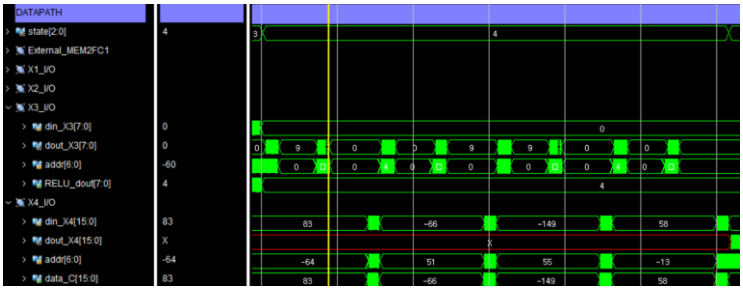
■ **U_mem_X2 -> RELU -> U_mem_X3**



(In the RELU state, we can observe that the input is received from U_mem_X2, the results are stored in U_mem_X3, and the RELU output is being computed.)

2.2.5 FC2(batch0~7)

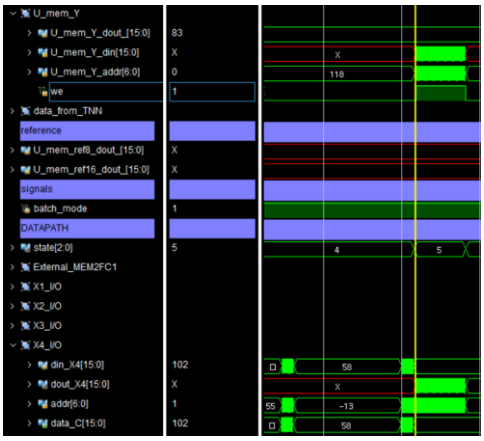
■ **U_mem_X3 -> FC2 -> U_mem_X4 & U_mem_Y(testbench)**



(In the FC2 state, we can observe that the input is received from U_mem_X3, the results are stored in U_mem_X4, and the SA output is being computed.)

2.2.6 TRANSFER(batch0~7)

■ **U_mem_X4 -> U_mem_Y(testbench)**

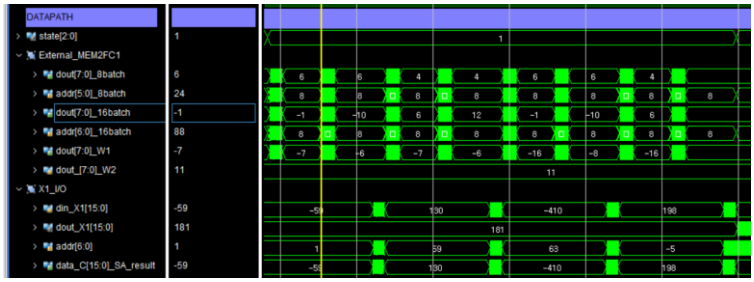


(mem_x4 values are connected to data_Y(output wire) and stored in U_mem_Y in the testbench.)

2.2.7 FC1(batch8~15)

8	-1	-16	-5	-2	-8	-14	-8
-4	-13	6	12	9	-8	-7	-9
0	6	-1	-10	0	14	9	-10
5	7	6	-2	16	-13	-11	-7
-1	-13	9	-14	-9	13	-16	6
-16	-15	7	-5	-13	1	-12	-4
5	1	-13	-7	10	14	-9	13
14	2	8	-8	14	2	9	1
-12	-16	6	2	-15	-8	12	14
5	1	-7	-11	10	8	-5	15
6	-15	-11	-16	2	10	-6	0
-2	-4	-13	7	-14	16	11	-16
16	-1	0	-3	-1	-10	5	-10
12	-6	-15	3	0	16	6	-16
-12	9	9	10	3	-8	-13	0
7	7	-4	12	-13	-11	-10	-14

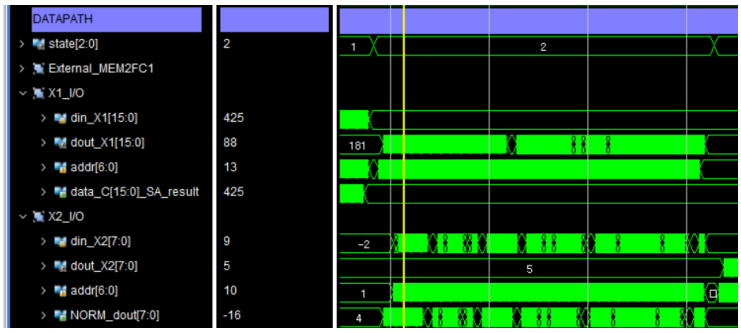
■ **external mem -> FC1 -> U_mem_X1**



(In the FC1 state, we can observe that the input is received from external memory, the results are stored in U_mem_X1, and the SA output is being computed.)

2.2.8 NORM (batch8~15)

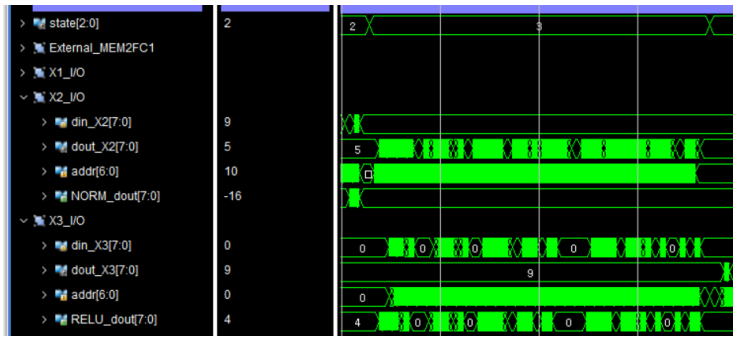
■ **U_mem_X1 -> NORM -> U_mem_X2**



(In the NORM state, we can observe that the input is received from U_mem_X1, the results are stored in U_mem_X2, and the NORM output is being computed.)

2.2.9 RELU(batch8~15)

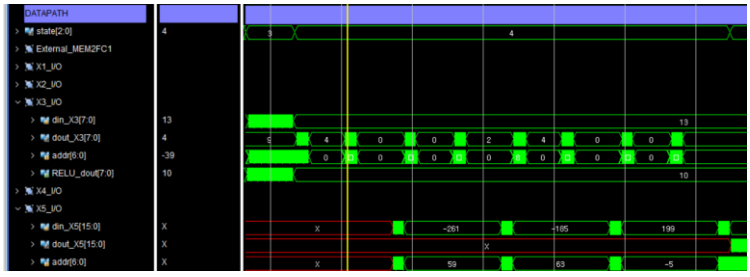
■ **U_mem_X2 -> RELU -> U_mem_X3**



(In the RELU state, we can observe that the input is received from U_mem_X2, the results are stored in U_mem_X3, and the RELU output is being computed.)

2.2.10 FC2(batch8~15)

■ **U_mem_X3 -> FC2 & U_mem_Y(testbench)**



(In the FC2 state, we can observe that the input is received from U_mem_X3, the results are stored in U_mem_X5, and the SA output is being computed.)

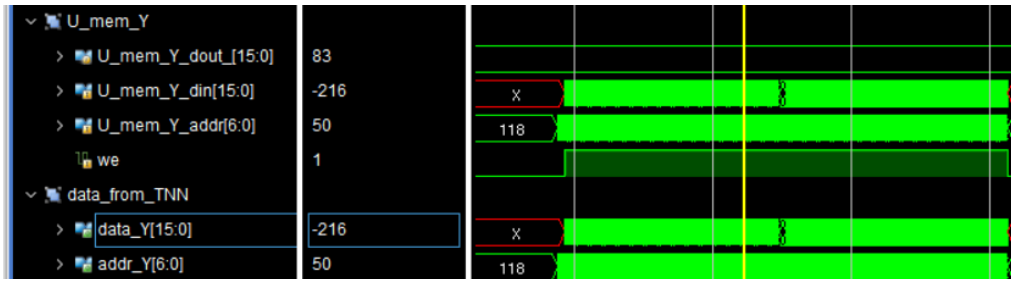
2.2.11 TRANSFER(batch8~15)

■ **U_mem_X5 -> U_mem_Y**

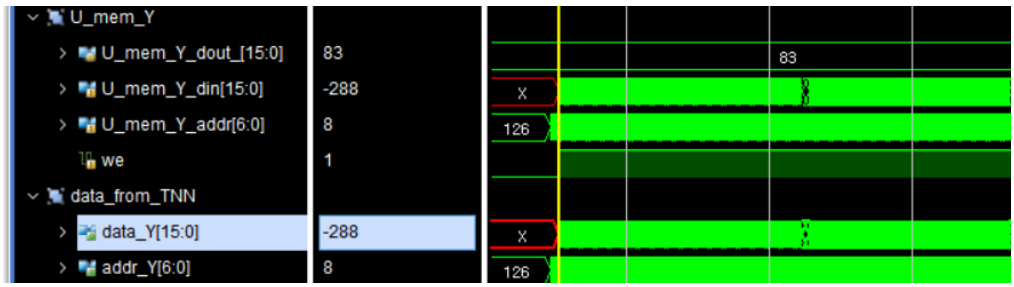
(mem_x5 values are connected to data_Y(output wire) and stored in U_mem_Y in the testbench.)

<Testbench Dataflow in 16 batch>

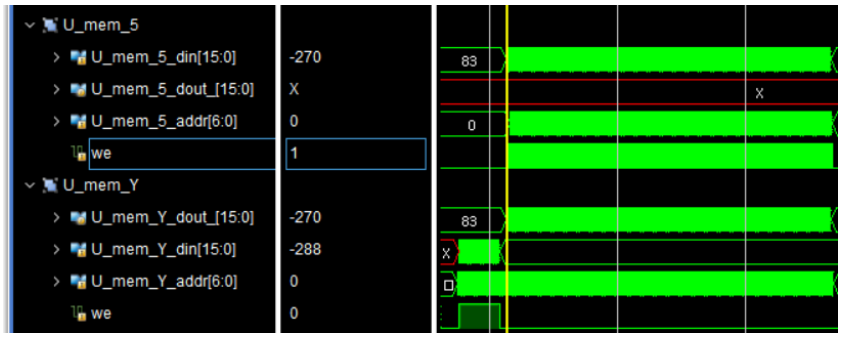
■ **TNN_batch0~7 -> tb_top (U_mem_Y)**



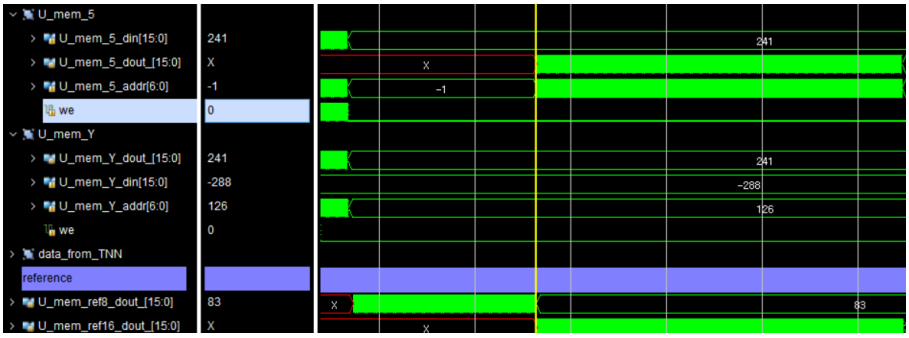
■ **TNN_batch8~15 -> tb_top (U_mem_Y)**



■ U_mem_Y -> U_mem_5 (copy data mem_Y to mem_5)



■ Finally compare results with reference



<Overall Waveforms>

결론

이 프로젝트를 통해 AI 가속기 설계의 핵심 원리와 실제 구현의 복잡성을 깊이 이해할 수 있었다. 특히, Systolic Array의 타일링 기법이 제한된 하드웨어 리소스로 큰 행렬 연산을 효율적으로 처리하는 핵심임을 깨달았다.

AI 가속기 설계는 정확성, 성능, 전력 효율성, 면적 효율성 사이의 복잡한 균형을 맞추는 작업이다. 간단해 보이는 MAC 연산 하나에도 수많은 최적화 고려사항이 숨어있으며, 모바일 기기와 IoT 환경에서의 AI 구현을 위해서는 이러한 트레이드오프가 더욱 중요해질 것으로 보인다.