REPORT

보고서 작성 서약서

1. 나는 타학생의 보고서를 복사(Copy)하지 않았습니다.
2. 나는 타학생의 보고서를 인터넷에서 다운로드 하여 대체하지 않았습니다.
3. 나는 타인에게 보고서 제출 전에 보고서를 보여주지 않았습니다.
4. 보고서 제출 기한을 준수하였습니다.

나는 보고서 작성시 위법 행위를 하지 않고,
성.균.인으로서 나의 명예를 지킬 것을 약속합니다.

과 목 : 논리회로설계실험_ICE2005_44

과 제 명 : Term-Project

담당교수 : 오 윤 호

학 과 : 전자전기공학부

학 년 : 3학년(5학기)

학 번 : 2018312959(최보열-제출자)

이 름 : 최보열, 박상현, 장재성, 홍나경

제 출 일 : 2022/06/03

조 : Group4

0. Table of Contents

1. **Report Cover** **(1pg)**
2. **Introduction** **(2pg)**
3. **Memory** **(3pg)**
4. **Computation module** **(3pg~16pg)**
    1) **Serial mode** **(3pg~6pg)**
    2) **Systolic mode** **(7pg~9pg)**
    3) **Custom mode** **(10pg~16pg)**
5. **Display module** **(17pg)**
6. **Controller** **(17pg ~ 19pg)**
7. **Simulation** **(20pg ~ 25pg)**
8. **Collaboration** **(26pg)**

1. Introduction

본 프로젝트의 목표는 DNN model에 사용되는 multiply-accumulate(Mac) 연산 가속화를 위한 효율적인 HW module(Systolic array)를 설계하는 것이다. 이에 'Group 4'는 2가지 목표를 가지고 진행하였다. 첫째는 주차별 강의를 통해 설계한 다양한 module을 최대한 많이 활용해보는 것이며, 두번째는 Custom module 설계에 있어 빠른 Latency와 적은 resource를 사용하며 빠른 latency의 HW module을 설계하는 것에 있음.

이후, 설명하는 모든 module은 (controller제외), gatelevel, structural modeling 기법만을 사용하여 설계하였음.

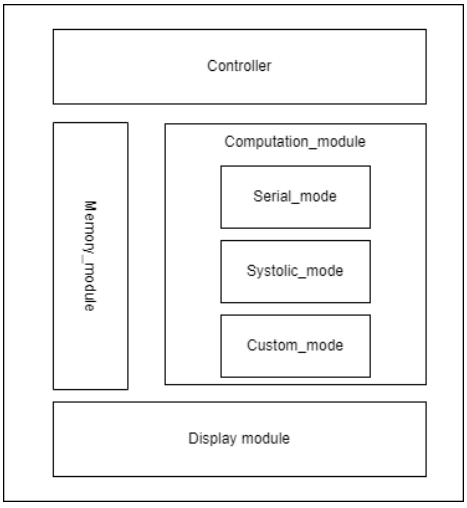


Figure 1-Top Module Block Diagram

2. Memory

2.1 Memory address map

추후 연산에 필요한 data의 순서와 data들의 저장 위치를 부분적으로 일치시켜 Memory에 대한 효율적인 data 접근이 가능하도록 위와 같은 address map을 결정하였다.

Address	Data	Address	Data	Address	Data
0x0000_0000	B11	0x0000_0009	A11	0x0000_0012	A32
0x0000_0001	B21	0x0000_000A	A12	0x0000_0013	A33
0x0000_0002	B31	0x0000_000B	A13	0x0000_0014	A34
0x0000_0003	B12	0x0000_000C	A14	0x0000_0015	A41
0x0000_0004	B22	0x0000_000D	A21	0x0000_0016	A42
0x0000_0005	B32	0x0000_000E	A22	0x0000_0017	A43
0x0000_0006	B13	0x0000_000F	A23	0x0000_0018	A44
0x0000_0007	B23	0x0000_0010	A24		
0x0000_0008	B33	0x0000_0011	A31		

Table 1-Memory Address Map

2.2 Memory Access

해당 project에서 memory 접근하는 순간은 크게 2가지인데, memory를 write initialize 하는 접근과 연산에 필요한 data를 read하는 접근이다. 각 접근에 필요한 address는 상이하며 준비된 memory는 single port이기 때문에, 어떠한 address 와 we(write enable) signal을 사용할 건지 선택해야 한다. 이를 제어하는 것이 오른쪽 그림의 'memory interface module' 이다.

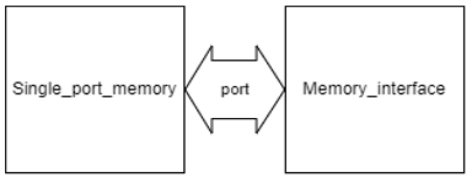


Figure 2-Memory & Memory Interface

"Memory는 오직 Memory Interface와 정보를 교환한다."

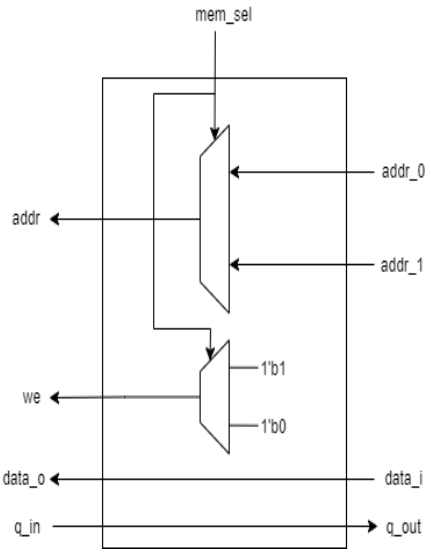


Figure 3-Memory Interface Module Block Diagram

3. Computational Module

3.1 PE(Processing Unit)

PE는 Serial, Systolic-array를 구성하는 기본 Processing unit이다. 따라서 Serial-mode와 Systolic-array에 대한 설명 전에 PE의 구성 및 동작에 대해 설명할 예정이다. PE에는 Serial-mode와 Parallel-mode를 위한 영역이 구분된다.

Serial-mode / Parallel-mode -> Systolic-array structure

1) Serial-mode operation

Serial-mode의 연산은 두 입력 $A_{i,j}$와 $B_{i,j}$을 곱하는 연산이 진행되고, 곱의 결과에 다음 순서의 입력 $A_{i,j}$와 $B_{i,j}$에 대한 곱의 결과가 더해지는 과정이다. 즉, 곱셈 연산의 결과를 계속 누적한다. Serial-mode에 대한 구조는 곱셈 연산을 하는 Multiplier와 결과값을 누적하기 위한 덧셈 연산이 포함된 Accumulator로 구성된다.

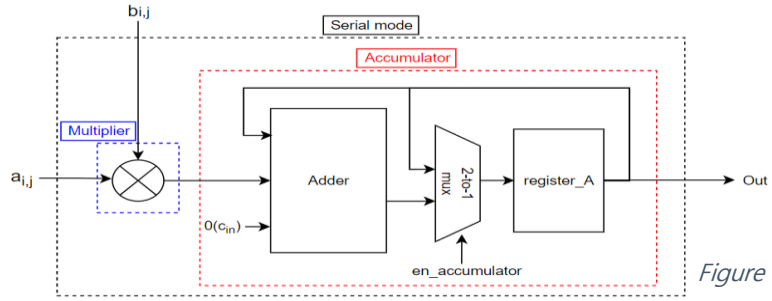


Figure 4-PE Serial-mode domain Block Diagram

2) Parallel-mode operation

Parallel-mode의 경우, 먼저 $A_{i,j}$와 $B_{i,j}$를 곱하고, 결과 값에 위쪽 PE로부터 받은 Parallel-mode 연산 결과인 y_in(맨 위쪽 PE인 경우 0)을 더하는 과정으로 이뤄진다. 이 때, $A_{i,j}$는 왼쪽 PE로 전달받고, $B_{i,j}$는 외부 module(Data loader)에 의해 연산 전에 미리 register에 저장되어 있다. CLK의 trigger가 발생하면 Parallel-mode의 연산 결과는 아래쪽 PE로 전달되며, 왼쪽 PE로 전달받은 $A_{i,j}$은 오른쪽 PE로 전달된다. 이에 대한 Block diagram은 다음과 같다.

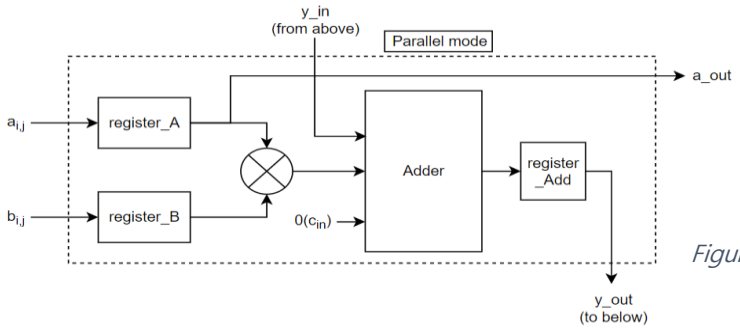


Figure 5-PE Parallel-mode domain Block Diagram

위에서 언급한 Serial-mode와 Parallel-mode구조를 합쳐 하나의 PE 안에 구성하면 다음과 같다.

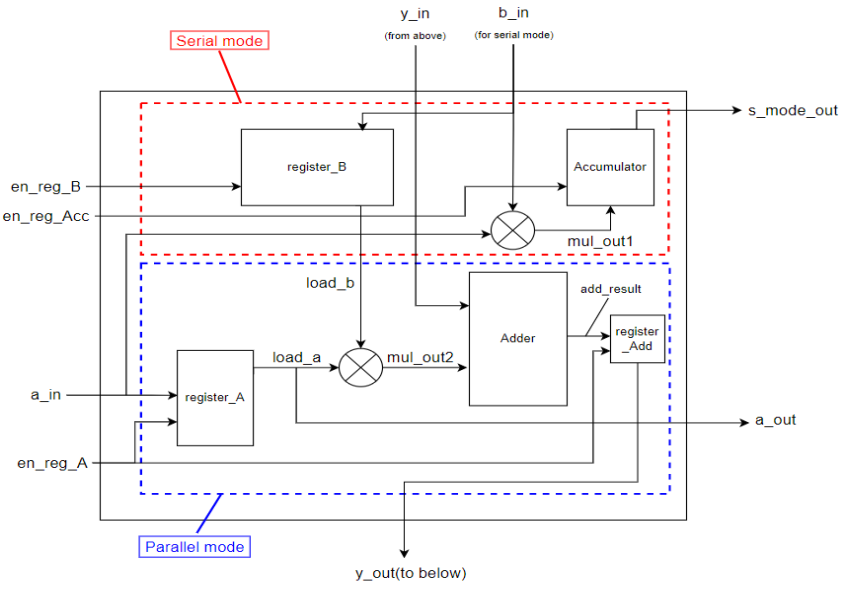


Figure 6-Overall PE Block Diagram

(각 in/out port들에 대한 자세한 설명은, 추후 Data Loader간의 interface 목차에서 보다 자세히 설명하겠다)

3.2 Serial-mode

Serial-mode는 앞서 설명한 PE 1개와 Serial-mode연산을 control하는 Serial-Data-Loader로 구성된다.

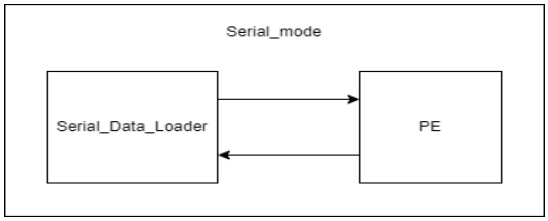


Figure 7-Serial-mode Top hierarchy

1) Serial-Data-Loader 역할

Serial-Data-Loader의 역할은 PE의 input으로 쓰일 값들에 해당하는 address를 생성/전송하고, 그 address를 통해 memory로부터 전달받은 data를 적절한 timing에 PE로 전달한다. 또한 PE Accumulator update를 control하는 신호를 생성하는 역할을 한다. 이는 다른 Systolic, Custom 연산에서 사용한 모든 Data-Loader들의 공통사항이다.

2) INTERFACE between PE and Serial-Data-Loader

name	Direction	Description
A_in	Data_Loader -> PE	Memory상에서 Data Loader에서 생성한 A의 address에 저장되어 있는 실제 Feature value
B_in	Data_Loader -> PE	Memory상에서 Data Loader에서 생성한 B의 address에 저장되어 있는 실제 Filter value
en_acc	Data_Loader -> PE	PE에 존재하는 accumulator의 enable 신호. Data Loader에서 input set(A,B)가 준비되기 전까지는 비활성화되어 있다가 준비가 되면 활성화 시킨다.

Table 2-PE & Serial-Data-Loader Interface

3) Base Address & Offset

먼저 input으로 쓰이는 $A_{i,j}$와 $B_{i,j}$의 address를 구하는 과정에 대한 설명이다. B(filter) 값들이 저장되어 있는 memory의 address값은 striding 순서와 상관없이 일정하나, A(feature)의 값들이 저장되어 있는 memory address의 값들은 striding에 따라 변화한다. 이때, A(feature)의 address를 다음과 같이 생각하면, striding마다 변하는 address값을 규칙적으로 바라볼 수 있다.

$$A's\ address = base\ address + A's\ offset$$

$$A's\ offeset = count + additional\ num$$

$$B's\ address = B's\ offeset = count$$

이를 이용하면, 매 striding마다, base address의 값만 변화시켜서 일관되게 적용할 수 있다. 이를 counter와 추가적인 logic을 통해 위 식의 count와 additional num을 생성한다.

CLK	1	2	3	4	5	6	7	8	9
$A_{i,j}$	A_{11}	A_{12}	A_{13}	A_{21}	A_{22}	A_{23}	A_{31}	A_{32}	A_{33}
$B_{i,j}$	B_{11}	B_{21}	B_{31}	B_{12}	B_{22}	B_{32}	B_{13}	B_{23}	B_{33}
baseaddr	1st striding = 6'b00_1001								
A's offset	0	1	2	4	5	6	8	9	10
B's offset	0	1	2	3	4	5	6	7	8
count	0	1	2	3	4	5	6	7	8
add num	0	0	0	1	1	1	2	2	2

Table 3-Base Address & Offset in Serial-mode

4) Serial-mode dataflow control

Serial-mode 연산을 위한 input들은 각 striding에 맞춰 Figure8의 왼쪽과 같은 순서로 PE에 load되어야 한다. 그러나 Memory의 1개의 data를 load하기 위한 read latency가 1 cycle이 소모되기 때문에, x-2와 같이 하나의 input set을 준비하고 다음 input set을 준비하는데 까지의 delay가 존재한다. **이 delay의 시간동안, PE의 accumulator는 동작을 멈춰야 한다.** 이를 control하기 위해서 PE의 accumulator의 Enable신호를 해당 data loader에서 생성/전송하도록 한다.

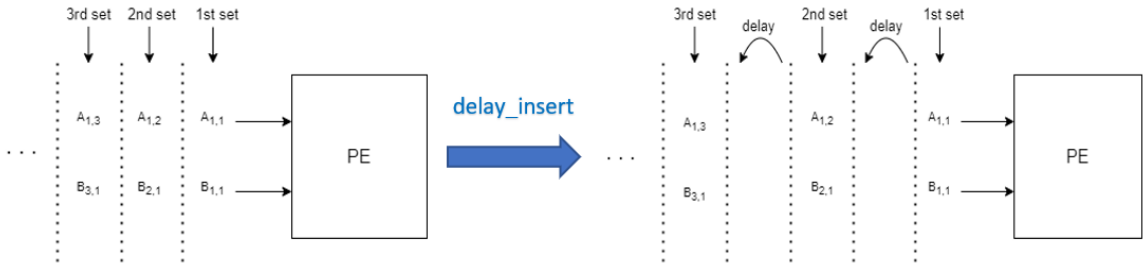
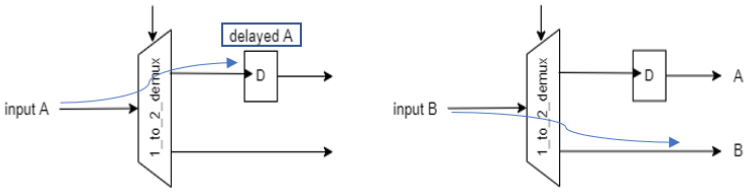


Figure 8- Serial-mode Input Set Loading Delay

Input set들간의 delay뿐 아니라 하나의 input set을 준비하기 위해서도 delay가 필요하다. A값이 먼저 준비되면 이를 D flip-flop을 통해 1 cycle delay 시키고, 다음 B값이 준비되는 타이밍에 동시에 전달하도록 설계하였다.

Figure 9-Serial-mode Input Set Delay for Sync



5) Serial-Data-Loader Block Diagram

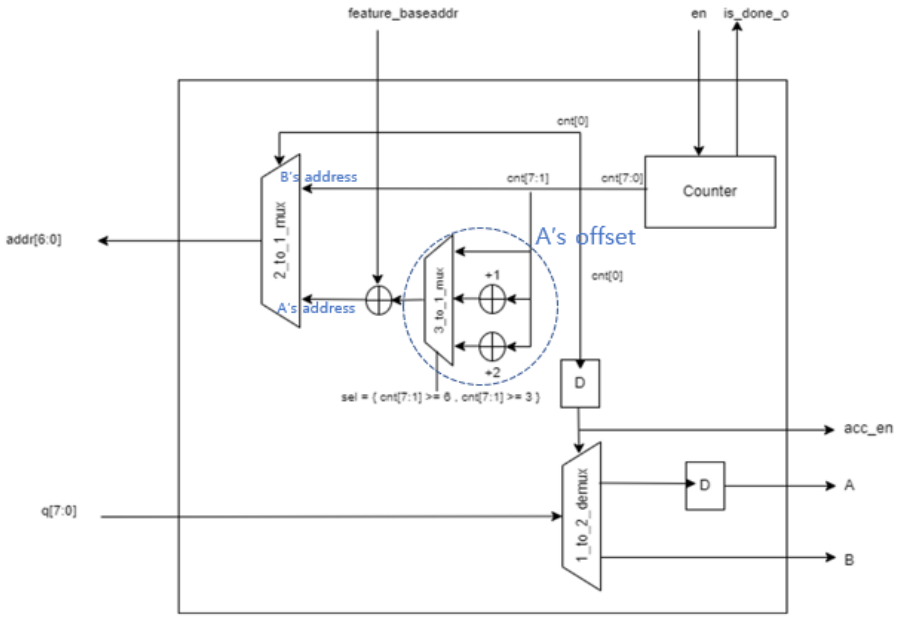
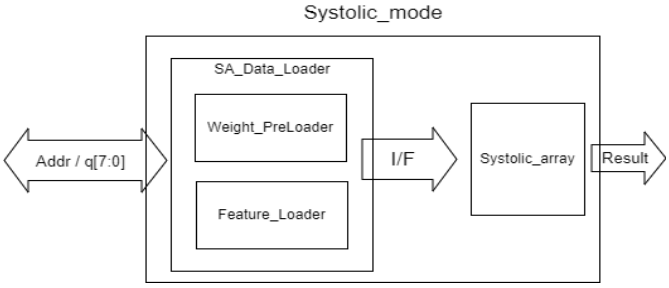


Figure 10-Serial-Data-Loader Block Diagram

3.3 Systolic mode

Systolic mode는 Systolic-array와 SA(Systolic Array)-Data-Loader로 구성된다.



1) Systolic Array

9개의 PE가 3x3 matrix array형태로 연결되어 systolic structure를 구성한다. 앞선 PE의 parallel-mode가 지원되도록 적절한 porting이 필요하고, weight값이 preload되기 위한 data-path가 존재한다.

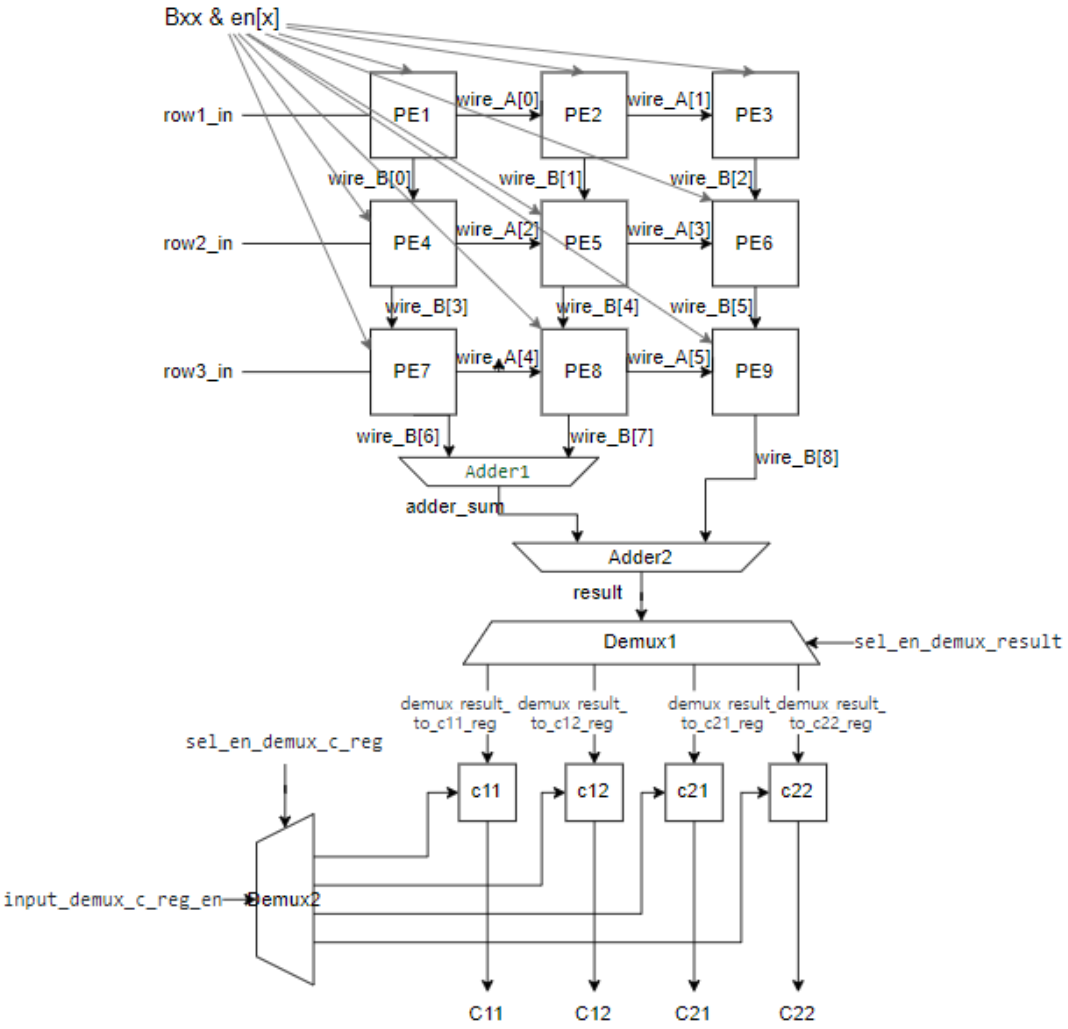


Figure 11-Systolic-array Structure

2) Interface between Systolic array and SA-Data-Loader

Name	Direction	Description
[7:0] B_{xx} [8:0]	SA_Data_Loader -> Systolic array	Systolic array에 PreLoad하기 위한 Weight value들
[8:0] Preload_ens	SA_Data_Loader -> Systolic array	Systolic array에 Weight value를 PreLoad하기 위한 Register의 enable 신호. 각 1bit 씩 하나의 register's en에 연결
sa_en	SA_Data_Loader -> Systolic array	Data Loader에서 input set이 준비되기 전까지 Systolic array의 모든 동작을 잠시 멈추기 위해 사용되는 signal, Systolic array 내에서 reg_A와 register_Add의 en로 사용된다.
row1_in, row2_in, row3_in	SA_Data_Loader -> Systolic array	Systolic array의 input set이자 feature($A_{i,j}$)에 해당하는 value들

3) SA-Data-Loader dataflow control

Systolic-mode는 연산 이전에 9개의 PE 각각에 존재하는 register_B에 weight 값들을 미리 저장시켜 두는 과정이 필요하다. 이를 weight-preload 라고 지칭한다. weight-preload는 외부 module인 'SA_Data_Loader'의 'Weight_PreLoader'에 의해 진행된다.

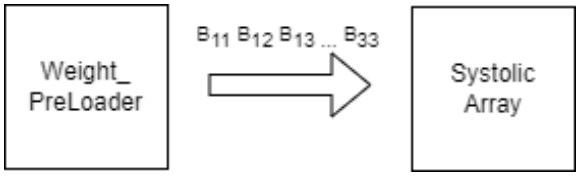


Figure 12-Weight-preload

모든 weight 값들이 저장되면, 'Feature_Loader'를 통해 Systolic-array의 input으로 사용될 row1_in, row2_in, row3_in를 준비/전송된다. 그 순서는 다음과 같다. 그림x-x에서 마지막 input set(0,0,A_{13})이 들어오고(5^{th} cycle) 난 후 다음 cycle인 6^{th}에서 각 striding의 결과값(c_{xx})이 도출된다.

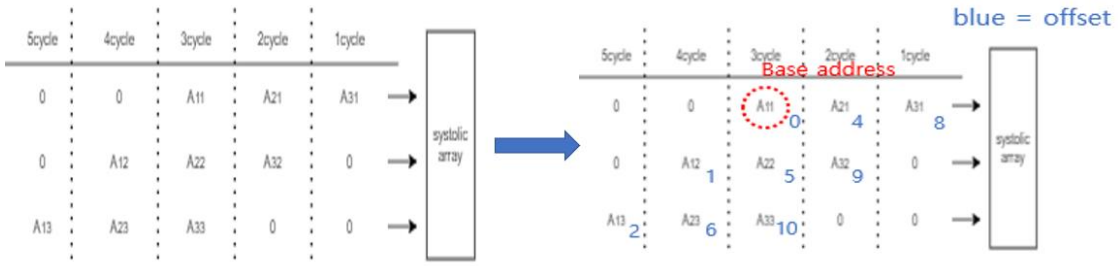


Figure 13-Systolic-mode Input set

A(feature), B(Weight)에 해당하는 address 역시 PE에서 사용했던 방식을 이용한다.

$$A's\ address = base\ address + offset\ ,\ A's\ offeset = Decoder's\ output\quad B's\ addres = Decoder's\ output$$

Offset을 생성하는 방식은 PE와 달리 내부 counter에서 출력하는 'cnt(count)'값을 input으로 받아서 순차적으로 필요한 offset(figure13-오)을 출력해주는 Decoder를 사용하여 처리한다.

Input set(Feature_1, Feature_2, Feature_3)을 준비하는 것 또한 PE와 유사한 timing delay Issue를 가진다. Data Loader로부터 input set을 준비되어 입력되기 전까지 Systolic array는 동작을 멈춰야 하는데, 이를 control하는 signal이 'sa_en'이다. 또한 하나의 input set이 준비되기까지 data를 잠시 저장하고 출력되는 timing을 맞춰야 하는데, 다음 logic을 통해 구현할 수 있다.

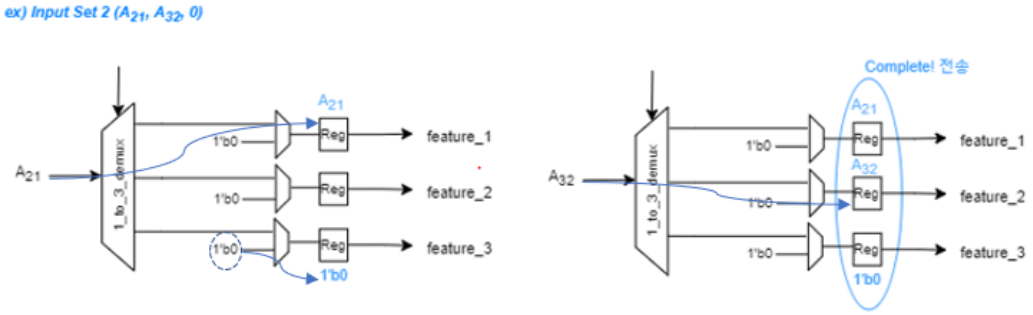


Figure 14-Systolic Input Set Delay for Sync

4) SA-Data-Loader Block Diagram

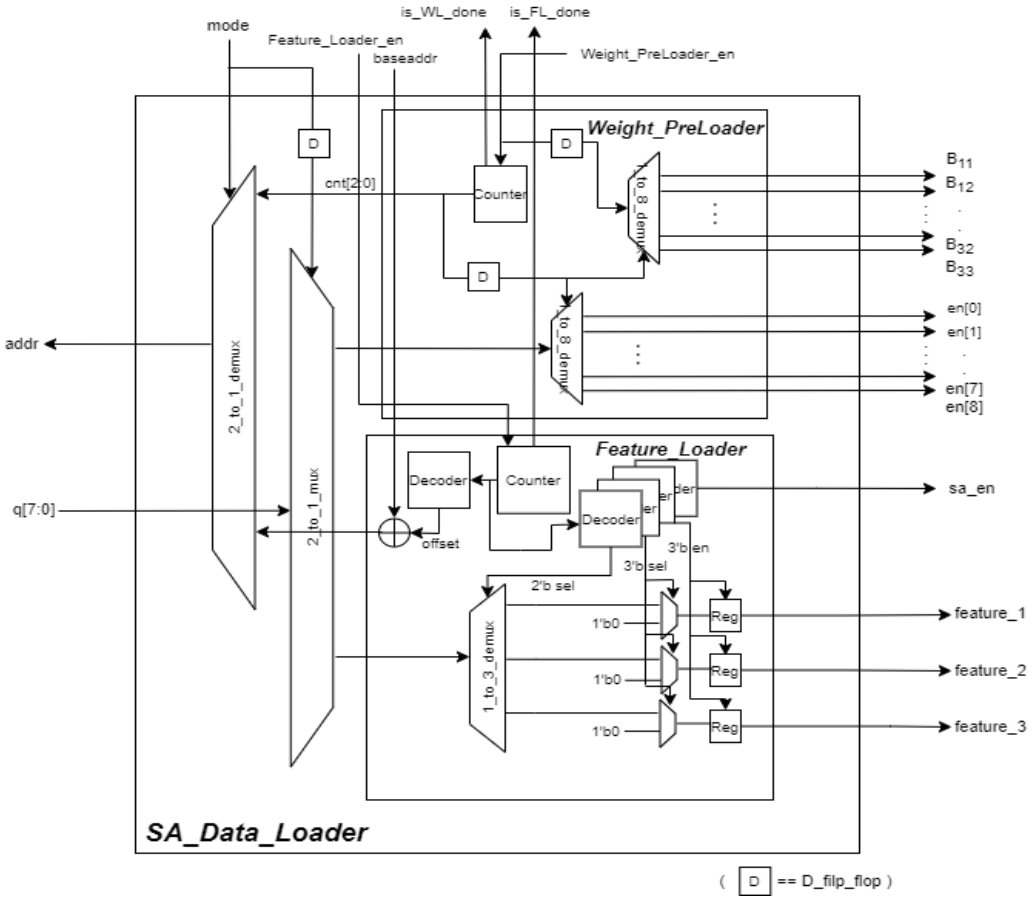


Figure 15-SA-Data-Loader Block Diagram

3.4 Custom mode

1) Introduction

Term project에서 공통적으로 사용되는 SPRAM(single-port-ram)은 1cycle에 8bit data 1개밖에 가져오지 못하고 이는 Serial, Systolic array의 성능에 주요 bottleneck이다. 따라서 SPRAM을 사용하는 HW에서 CNN연산의 속도를 높이기 위해, memory access를 낮추는 것이 중요하다고 판단된다. 따라서 해당 목차에서는 feature, weight buffer와 추가적인 control signal을 이용해서 memory access를 최대한 줄이고, 연산의 결과인 2x2 output feature 4개를 동시에 구하는 Custom module을 소개한다.

2) Design approach

2-1) Backgroud

Custom module의 아이디어는 크게 Data reuse, Parallelism, Operation participation 3가지 측면에서 접근하여 고안되었다. 첫번째로, Systolic array의 data reuse에서 착안하였다. Systolic array 또한 memory access를 중요 bottleneck으로 보았고, 이를 줄이기 위한 노력으로 weight값들을 PE에 저장하는 방식을 통해서 사용하고 있다. Custom module에서는 더나아가, weight와 feature를 모두 저장하는 방식으로 memory access를 줄여나갈 것이다. 두번째로, Systolic array의 parallelism에서 착안하였다. Systolic array는 2D matrix구조의 PE array로 feature와 weight의 multiplication연산이 동시에 여러 PE에서 진행되어 serial module에 비해 많은 cycle을 overlapping할 수 있었다. Custom module에서는 첫번째 발상에 더불어 준비되어있는 weight와 feature들을 이용해서 stride기준으로 연산을 분할해서 진행하는 것이 아니라 4번의 stride연산이 동시에 진행되도록 설계해서 cycle time을 줄여나갈 것이다. 세번째로, CNN연산의 각 feature와 weight연산 참여횟수에 집중하였다. Figure16은 4x4 input feature와 3x3 weight의 연산 참여횟수를 나타나는데 이를 확인해보면 하나의 feature가 weight와 연산하는 횟수가 위치에 따라서 차이가 있는 것을 확인할 수 있다.

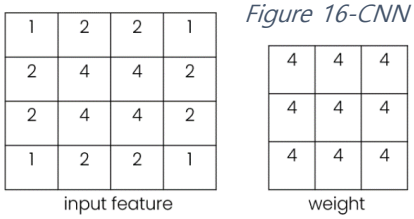
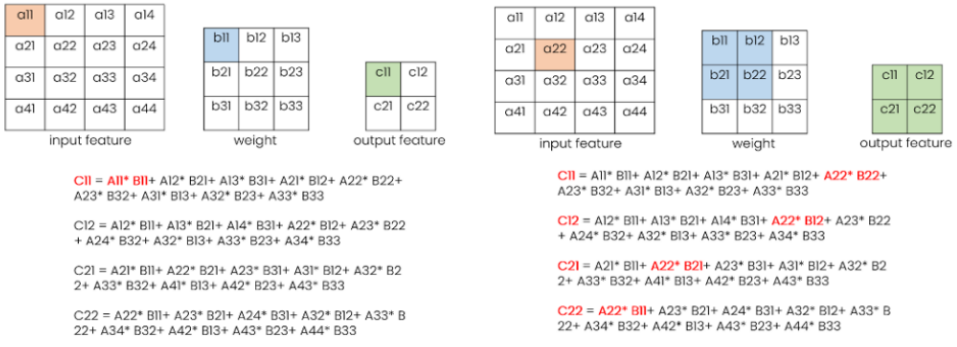


Figure 16-CNN 연산참여 횟수

이를 통해서 효율적인 buffer사용, feature, weight buffer 개수 짐작이라는 2가지 통찰을 얻을 수 있다. 먼저 weight 최대 4개가 미리 준비되어 있을 경우에 해당 feature가 들어간다면 output_feature map 4개에 필요한 모든 값을 연산할 수 있다. 예를들어, Figure17(오)과 같이 a22는 2x2 feature_out에 4번의 weight와 연산을 통해서 4개 b11, b12, b21, b22의 feature_out의 모든 값에 필요하다. 반면에 Figure17(왼)의 a11과 같은 경우에는 겨우 b11과의 1번의 연산이 끝이다. 따라서 feature를 기준으로, a22가 memory에서 들어오는 경우와 같이 최대 4개의 weight가 준비되었을 경우에 해당 feature는 더 이상 CNN연산에서 필요가 없어진다. Weight기준으로 볼 경우 b11과 연산되는 a11, a12, a21, a22가 모두 들어와서 연산이 마무리된 경우 weight의 필요가 없어진다. 이를통해 효과적으로 buffer를 사용해서 resource를 줄일 수 있다. 그렇다면 feature, weight buffer를 몇 개 사용하는 것이 좋을 것인지를 판단해야되는데 설계자는 참여횟수가 최대 4회인 것을 보고 4개의 weight와 feature buffer를 기준으로 잡았다. 여유 buffer로 각각 5개의 buffer를 사용하는 것을 초기설계에 기준으로 잡았다. 이후에 추가적인 설계 과정에서 연구에 의해 feature, weight buffer는 4개만 있어도 충분한 것으로 판단내릴 수 있었다.

Figure17-CNN Operation



2-2) Weight Ready Operation

Figure18은 Custom_Top에서 사용되는 Buff_ALU모듈의 block digram 이다. Feature_out 2x2인 4개를 동시에 구하기 위해서 다음과 같은 Buff_ALU가 4개가 존재한다. 먼저 weight가 weight_buff에 저장되고, feature가 feature_buff로 들어오면feature와 weight가 8_bit_multiplier 를 통과해서 결과값을 출력한다.

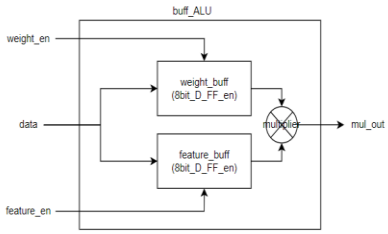


Figure 18-Buff-ALU Block Diagram

앞서 말한 내용처럼 weight가 free되기 위해서는 4개의feature값들과 연산이 마무리되어야 한다. 예를들어 Figure19와 같이 b11은 a11, a12, a21, a22와 연산되어야 한다. weight b11, b21, b31, b12가 각각의 ALU에 있는 weight_buff 4개에 저장이 되었고, a11 – a12 – a13 … - a22까지 순차적으로 들어온다고 가정하자. a22가 들어온 순간을 살표보면, b11과 연산되는 a11, a12, a21, a22의 연산이 모두 진행되고, 이는 accumulator에 적절히 저장되었을 것이다. 이후에 b11은 CNN연산에서 더 이상 사용되지 않아서, b11이 있던 weight_buff에 다음 weight인 b22를 update할 수 있게되고 다음으로 feature input들이 a22다음인 a23부터 feature들이 다시 들어오기 시작한다. 즉 weight가 들어오고, feature값들이 들어오면서 해당 weight의 4번의 연산이 끝나게 되면 다음 weight가 update되는 구조이다.

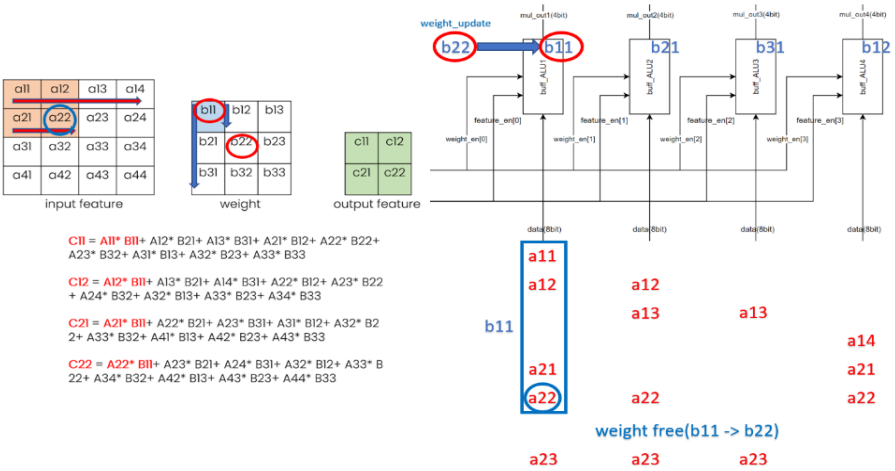


Figure 19-Weight-Free example(b11->b22)

2-3) Operation flow

Figure21은 전체적으로 memory의 access에 따라서 load되는 data가 feature와 weight buff의 어디에 update 되는지를 나타내는 dataflow이다. w는 weight를 나타내고 f는 feature를 나타낸다. w, f뒤의 숫자는 buffer_ALU의 번호를 나타낸다. 예를들어 그림20과 그림21을 참고해서 b11->w1는 weight값 b11이 buff_ALU1의 weight_buff에 update된다는 의미이고, 이와 동일하게 a11 -> f1는 buff_ALU1의 feature_buff에 update된다는 의미이다. 따라서 stage6의 a12와 같은 경우에는 f1, f2에 들어감으로 해당 stage에 weight_buff에 들어가있는 w1의 b11, w2의 b21과 연산해서 a12 * b11, a12 * b21이 연산되어 출력되는 것이다.

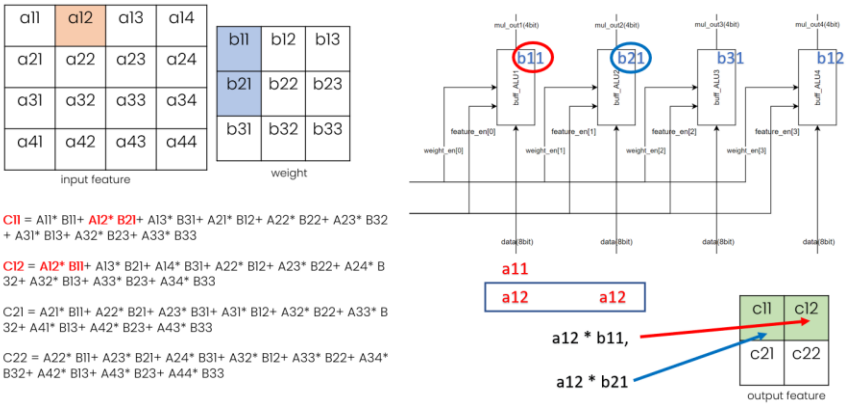


Figure 20-Weight-Feature-Update example(load a12)

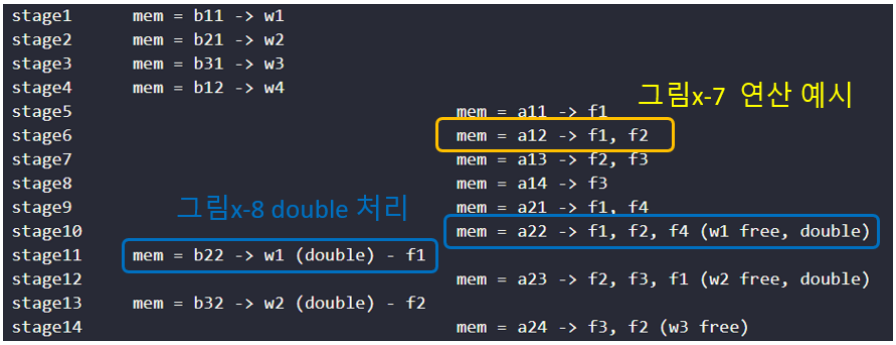


Figure 21-Weight, Feature Update Flow

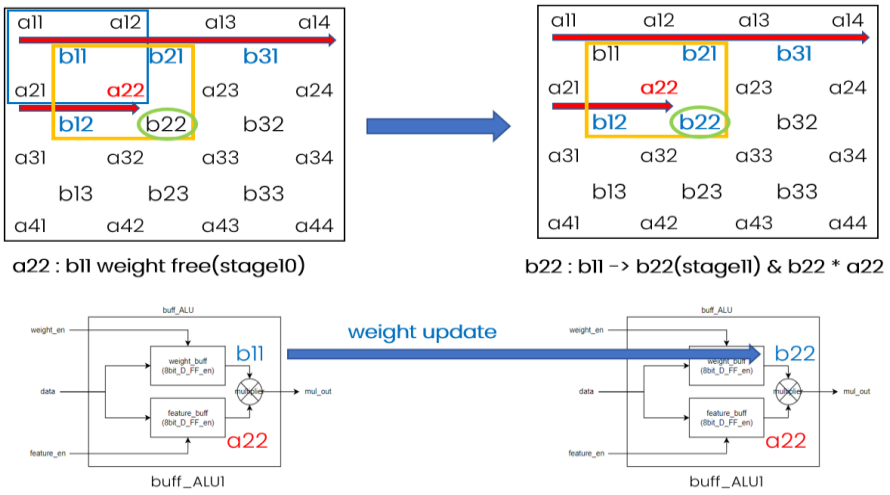
2-4) Double

해당 Operation에서 주의를 요하는 부분이 있다. Figure22를 살펴보자. 왼쪽은 a22가 들어오는 stage10을 나타내는데 앞서 설명한 weight ready operation에 따르면 a22가 들어올 때 주황색 사각형으로 둘러쌓인 weight_buff 4곳에 b11, b21, b12, b22개가 준비되어 있어야 한다(푸른색 글씨-> weight_buff에 저장되어있는 weight). 그런데 weight는 순차적으로 b11, b21, b31, b12를 가지고 있음으로 b22가 준비되지 않았다. 따라서 buffer가 5개 필요할 것으로 예상되지만 feature 또한 buffer에 저장되고 연산된다는 점에서 double의 개념을 도입하면, weight buffer는 4개면 충분하다는 관찰을 얻어낼 수 있다.

Figure22의 왼쪽과 같이 b22가 준비되지 않았지만 a22 feature를 먼저 update시켜 연산을 진행한다. 그러면 동시에 b11이 4개의 feature와 연산이 종료되었음으로 b11을 free하고 해당 weight_buff에 a22와 연산해야 되는 b22를 update시켜줄 수 있게되고 해당 ALU의 output을 accumulator로 보내주면된다. 그림22의 아래 buff_ALU1이 stage10과 stage11의 해당 과정을 보여주고 있다. 이로써 a22를 먼저 update시켰지만 b11, b21, b12, b22와의 연산 결과 모두를 얻어낼 수 있다.

Figure 22-Double example

*(b11 * a22, b22 * a22)*



2-5) Accumulate mul_out Value dataflow

앞선 Weight Ready Operation을 통해 2x2 feature_out을 구하기 위한 weight x feature의 모든 mul_out값들을 구할 수 있다. 그렇다면 이제 c11, c12, c21, c22를 각각 구하기 위해서 어떻게 weight x feature의 결과인 mul_out값들을 c11, c12, c21, c22로 재배치해서 accumulate하는지를 논의해야된다. 이는 복잡해 보이지만 상당히 단순하게 해결할 수 있다. Figure19의 b11이 순차적으로 연산되는 결과를 왼쪽 c11~c22 연산식과 함께 살펴보자. b11 * a11, b11 * a12, b11 * a21, b11 * a22이렇게 4개가 나오는데, 이는 c11, c12, c21, c22순으로 사용되는 것을 해당 그림의 왼쪽에 식에서 관찰할 수 있다. 이는 feature와 weight를 load하는 순서와 관련이 있다. 붉은 화살표와 같이 a11–a12–a13–a14–a21 - ..., 순서로 feature들을 load하기 때문에 해당 weight에서 나오는 output은 ACC1 – ACC2 – ACC3 – ACC4로 순차적으로 보내면된다.

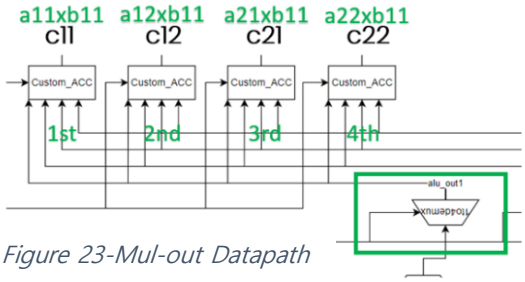


Figure 23-Mul-out Datapath

이 사실을 이용해서 Figure23과 같이 복잡한 control signal없이 1to4 mux와 2bit_upper라는 추가적인module을 이용해서 없이 mul_out을 적합한 accumulator로 전송할 수 있다.

2-6) Additional Discussion

해당 목차는 해당 custom module을 설계하기 위해서는 Custom_ACC, mul_out muxing, demux signal, control logic등에 대한 토의가 추가적으로 필요하다. 첫번째로, Accumulator에 4개의 buff_ALU로부터 mul_out을 받기 때문에 기존의 PE에서 사용되던 accumulator에 변경이 필요하고, 이를 해결하기 위해서 Figure24와 같이 8bit_adder 2layer를 추가한 custom accumulator를 사용하였다.

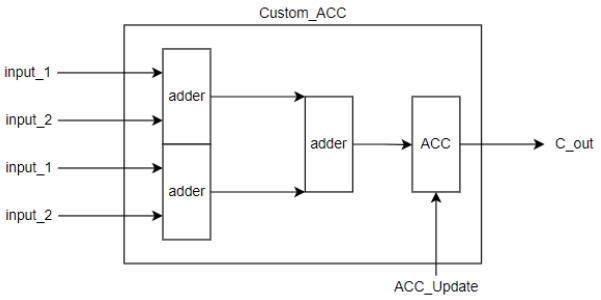


Figure 24-Custom Accumulator Block Diagram

두번째로, Accumulator는 Figure24, 25에서 볼 수 있듯이 buff_ALU 4개로부터 mul_out 4개를 input으로 받는데 잘못된 mul_out은 accumulator에서 연산이되지 않아야 함으로 추가적으로 mul_out값을 사용할 것인지 구분하는 muxing과정이 요구된다. 이를 위해서 buff_ALU부터 Acc까지 path에는 2 to 1mux를 추가하였다.

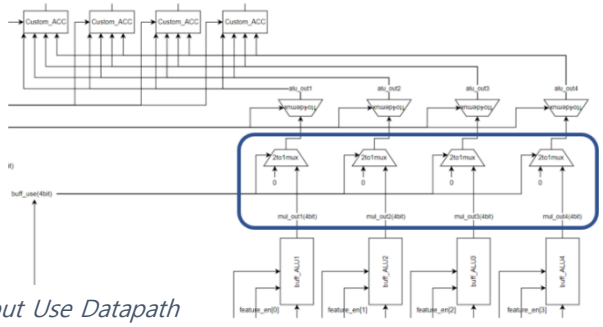


Figure 25-Mul-out Use Datapath

세번째로 Figure26에서 accumulator 4개에 c11부터 c22까지의 순차적으로 mul_out을 보내기 위해서 1 to 4 demux에서 00 – 01 – 10 – 11이라는 demux select signal이 필요한데 이와 같은 data를 client의 설정에 따라 순차적으로 내뱉는 2bit_upper설계가 필요하다. 이는 그림 x-11처럼 Adder, DFF, mux등을 이용해서 쉽게 구현할 수 있다. (2bit-up-counter와 동작 동일)

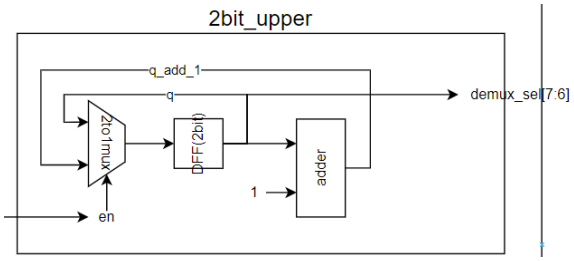


Figure 26-2bit-upper Block Diagram

마지막으로 해당 Custom module의 stage별로 동작을 control하는 signal들이 필요한데 이를위한 dataloader의 설계가 필요하다. data loader에서 생성해줘야 하는 control signal는 다음과 같다.

feature_en	buff_ALU 4개에 존재하는 feature_buff(DFF)의 enable신호로 들어오는 data가 적합한 feature_buff에 update되도록 control하는 신호.
weight_en	buff_ALU 4개에 존재하는 weight_buff(DFF)의 enable신호로 들어오는 data가 적합한 weight_buff에 update되도록 control하는 신호.
buff_use	weight x feature값인 mul_out을 8'b0와 muxing하는데 사용되는 mux select신호
demux_sel	muxing된 mul_out값인 alu_out 4개가 적합한 accumulator로 전달되록 demuxing하는데 사용되는 demux select신호
acc_en	ACC 4개의 enable신호로 알맞은 mul_out값들이 전달되었을 경우에 accumulator를 활성화시켜서 output feature 값을 누산한다.

Table 4-Custom Control Signal

3) Module Specification

Custom-mode는 Computational module에 의해서 control되는 Custom CNN연산의 Top hierachy에 있는 module이다. Custom_mode는 Data_loader와 Custom_Top으로 구성된다. Computational module에서 custom enable신호에 의해 Custom_mode가 active되고, Custom_mode는 내부의 data_loader에 의해 적절한 address를 출력한다. 해당 address는 computational module을 거쳐서 memory에서 적절한 data를 data_loader로 보내주고, data_loader는 해당 data와 함께 control신호를 Custom_Top으로 전송해준다. Custom_Top의 동작에 의해서 출력된 feature out값(c11, c12, c21, c22)과 data_loader의 is_done신호는 Computational module로 전달된다.

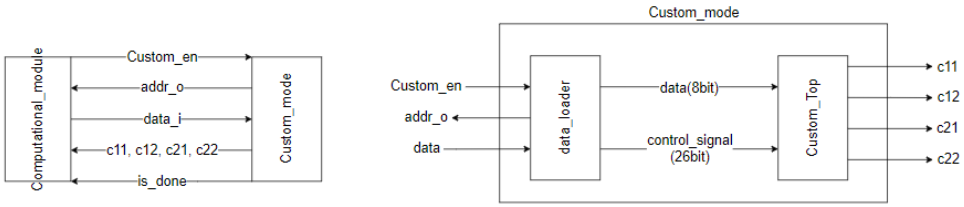


Figure 27-Custom-mode Hierarchy

3-1) Interface between Custom-mode and Computational module

Name	direction	description
Custom_en	Computation -> Custom	custom 동작을 시작하는 enable신호(active high), data_loader로 들어가서 data_loader의 counter를 작동시킨다.
addr_o	Custom -> Computation	memory에 data를 요청하는 address신호
data_i	Computation -> Custom	addr_o에 의해서 요청된 data신호, addr_o이 들어가고 1cycle뒤에 해당 data가 나온다.
is_done	Custom -> Computation	custom 동작이 끝났다고 computational module에 알려주는 신호로 Custom 동작이 끝났을 경우1cycle동안 high가 유지된다.
c11, c12, c21, c22	Custom -> Computation	CNN연산의 output feature값, is_done이 high됐을 경우 capture된다.

Figure 28-Custom-mode & Computational module Interface

3-2) Custom-Top

3-2-1) Custom Top Block Diagram

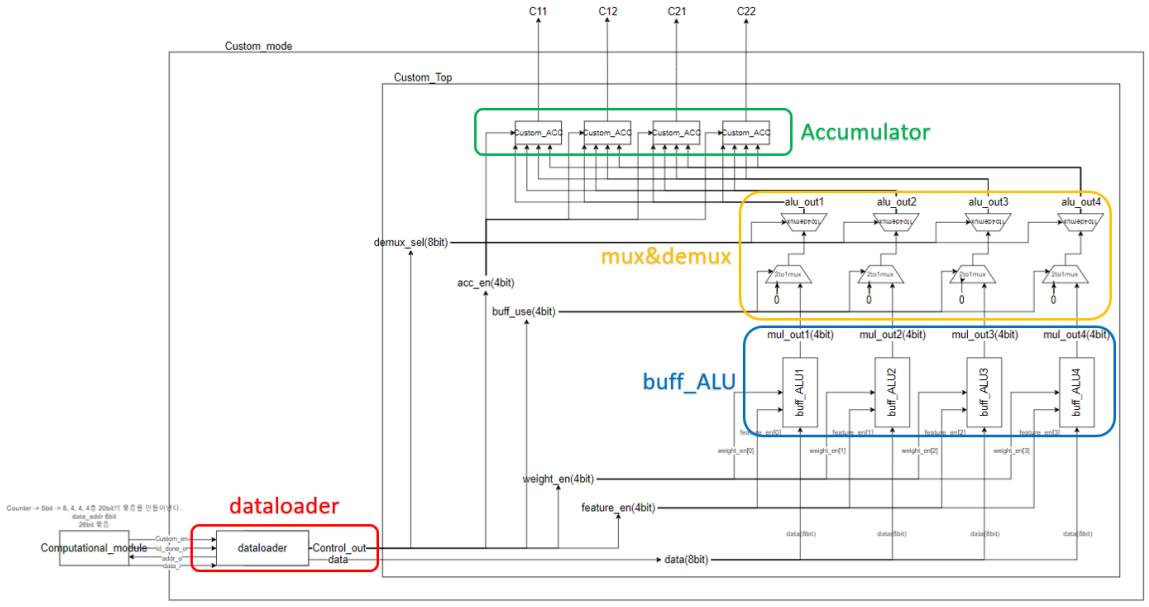
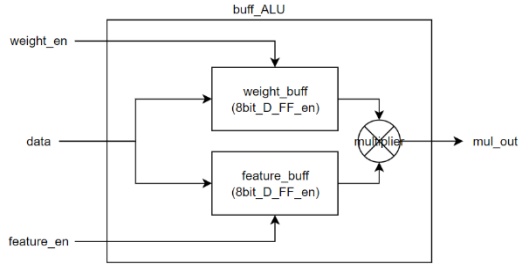


Figure 29-Custom Top Block Diagram

3-2-2) Custom-Top Dataflow

Data를 받으면 buff_ALU에 존재하는 feature_buff, weight_buff에 feature_en, weight_en신호를 이용해서 update시키고, feature x weight값인 mul_out을 출력한다. mul_out은 buff_use에 의해서 8'b0와 muxing되어 1to4 demux로 전달된다. 1to4 demux는 demux_sel신호를 이용해서 Custom_ACC 4개중에 적합한 accumulator로 값을 전달한다. 해당 Custom_ACC는 acc_en신호를 이용해서 알맞은 feature x weight 값일 경우에 값을 누산시켜서 output feature값을 만들어낸다.

Figure 30-Custom-Buff-ALU Block Diagram



3-2-3) buff_ALU(Custom_Top의 submodule)

Enable_8bit_DFF 2개와 multiplier로 구현된 간단한 ALU buffer 모듈이다. feature와 weight값을 받아서 posedge & enable high일 경우에 값을 update해서 weight x feature을 출력한다.

3-2-4) Custom_ACC(Custom_Top의 submodule)

8bit_adder 3개와 accumulator로 구성된 Custom_Accumulator이다. demux를 통해서 mul_out값들을 4개(사용되지 않는 path는 8'b0를 전달해옴) 받아오기 때문에 동시에 모두 더한 값을 accumulator에 update하기 위해서 adder layer가 2개 추가되었다. acc_en에 의해서 적합한 mul_out값일 경우에 accmulator가 누산시켜 output feature값을 구한다.

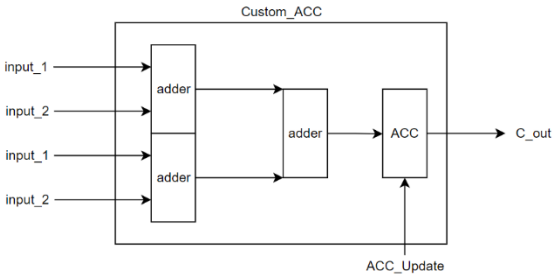


Figure 31-Custom-Accumulator Block Diagram

3-3) Custom-Data-Loader

Custom-Data-Loader는 기존의 Serial, Systolic에서 사용되는 data-loader들과 유사한 동작을 한다. Counter를 통해서 stage가 clk과 sync되며, data address와 dataflow control위한 signal들을 생성해서 Custom-Top의 연산을 관리한다.

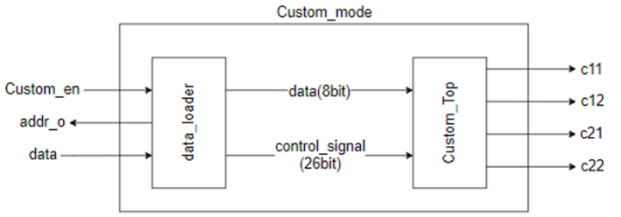


Figure 32-Custom-Top & Custom-Data-Loader Hierarchy

3-3-1) Interface between Custom-Data-Loader and Custom-Top

Name	Direction	description
data	Data-Loader -> Custom-Top	mem -> mem_IF -> Computation -> Data_loader를 통해서 전달받은 연산에 사용되는 data값
feature_en	Data-Loader -> Custom-Top	buff_ALU 4개에 존재하는 feature_buff(DFF)의 enable신호로 들어오는 data가 적합한 feature_buff에 update되도록 control하는 신호. (ex. buff_ALU1, buff_ALU2의 feature_buff를 update하는 경우 4'b1100)
weight_en	Data-Loader -> Custom-Top	buff_ALU 4개에 존재하는 weight_buff(DFF)의 enable신호로 들어오는 data가 적합한 weight_buff에 update되도록 control하는 신호. (ex. buff_ALU3의 weight_buff를 update하는 경우 4'b0010)
buff_use	Data-Loader -> Custom-Top	weight x feature값인 mul_out을 8'b0와 muxing하는데 사용되는 mux select신호 (ex. 해당 stage에서 buff_ALU1, buff_ALU3가 알맞는 mul_out값일 경우 4'b1010으로 나머지 buff_ALU2와 buff_ALU4는 muxing되어서 0이 되도록함)
demux_sel	Data-Loader -> Custom-Top	muxing된 mul_out값인 alu_out 4개가 적합한 accumulator로 전달되록 demuxing하는데 사용되는 demux select신호 (ex. alu_out1이 ACC3로 가야되는 경우 demux_sel[7:6] = 2'b10으로 demux하고 나머지 3개의 ACC는 0으로 전달) (*mul_out값이 사용되지 않을경우에는 buff_use에 의해 0으로 출력됨으로 아무값이 선택되어도 tolerance하다*)
acc_en	Data-Loader -> Custom-Top	ACC 4개의 enable신호로 알맞은 mul_out값들이 전달되었을 경우에 accumulator를 활성화시켜서 output feature 값을 누산한다.

Table 5-Custom-Data-Loader & Custom-Top Interface

3-3-2) Module Activation

Dataloader는 Serial, Systolic에서 사용되는 dataloader와 같은 개념이다. Scalalbe Counter에 설정된 cnt값을 이용해서 control과 address signal을 만들어내고 이를 buffer를 통해서 memory와 cycle sync를 맞춘다음 출력하는 동작을 한다. 다만, 2가지 주의깊게 살펴야 되는 부분이 존재한다.

먼저 Figure21을 보면 feature와 weight는 동시에 호출되지 않는다. 따라서 feature_en과 weight_en은 각각의 4bit짜리 decoder대신에 몇번째 ALU에 update를 해야되는지를 의미하는 4bit짜리 buff_en신호와 해당 cnt stage일 때 feature인지 buffer인지 구분짓는 1bit짜리 decoder로 resource를 줄일 수 있다.

두번째로 buff_use라는 값의 재사용이다. 앞서 Figure23에서 논의했듯이 하나의 ALU에서 출력은 ACC1부터 ACC4까지 순차적으로 전달되는데 이를 위한 demux_sel신호를 buff_use를 이용해서 구현할 수 있다. buff_use가 high라는 것은 해당 ALU값이 이용되었다는 것이고, 그렇다면 demux_sel은 다음 ACC를 가르키면 된다. 이를 앞에서 설명한 2bit_upper의 en값으로 사용한다면 추가적인 decoder없이 demux_sel신호를 만들어 낼 수 있다.

3-3-3) Custom-Data-Loader Block Diagram

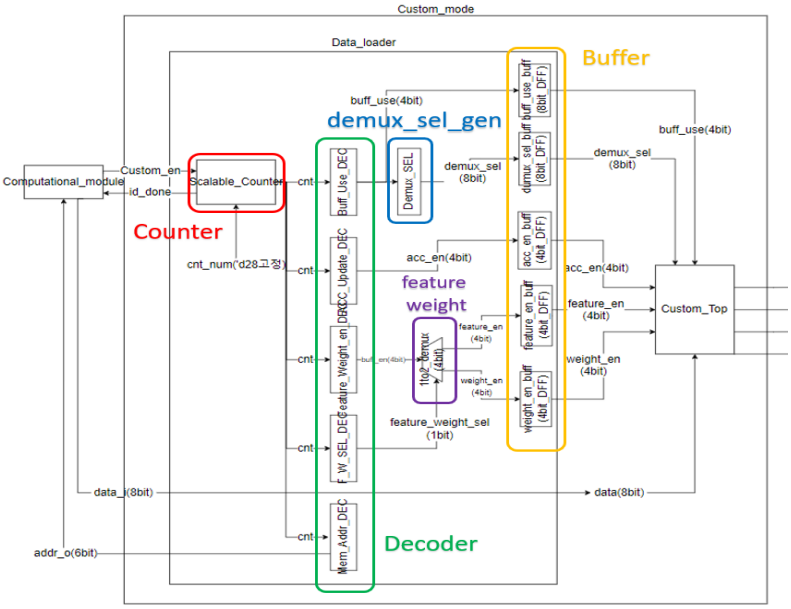


Figure 34-Custom-Data-Loader Block Diagram

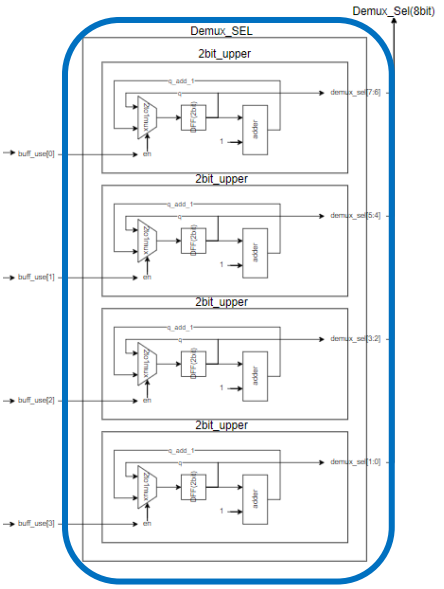


Figure 33-DEMUX-SEL-Generator Block Diagram

3-4) Custom Insight

(1) weight가 먼저 저장되고, 연산되기 위해서 들어오는 1개의 input feature가 update될 때 필요한 모든 weight가 준비되어 있음으로 feature와 weight를 위한 memory access는 중복되지 않고 unique하게 진행된다. 따라서 불필요한 memory access를 대폭 감소시킬 수 있다.

(2) Buffer를 효과적으로 사용할 수 있다. Feature와 weight를 buffer에 저장해서 reuse하는 HW logic이지만 적절한 data control통해 적은 수의 buffer로 data reuse를 수행할 수 있다.

(3) Multiplier의 개수를 대폭 낮출 수 있다. Systolic array와 같은 경우에 2D을 위해 weight의 개수와 동일한 9개의 multiplier를 사용하고 있지만, Custom module에서는 4개의 multiplier밖에 사용되지 않는다. Term-Project에서 CNN연산의 feature, weight들이 8bit int로 고정되어 있지만 해당 data들이 integer가 아닌 float가 되거나, data의 size가 8bit가 아닌 32bit가 되는 경우처럼 커진다면 Custom module의 resource(or area) 측면에서의 장점은 극대화될 것이다.

(4) Bottleneck으로는 CNN의 Scale이 커진다면 control signal logic의 complexity로 구현이 어려울 것으로 예상된다.

4. Display Module

Display module의 input으로 Systolic-mode 와 Custom-mode의 결과값이 들어오며, 이를 displayed_ number라는 하나의 배열에 저장한다. 총 8개의 result data를 출력하기 위해, 1초마다 배열의 index를 이동시켜야 하는데, 이때 필요한 signal이 'sel signal' 이다.

```
reg [15:0] displayed_number [0:7];
else if(reg_en == 1'b1) begin
    displayed_number[0] <= {8'b0,c11_sa};
    displayed_number[1] <= {8'b0,c12_sa};
    displayed_number[2] <= {8'b0,c21_sa};
    displayed_number[3] <= {8'b0,c22_sa};
    displayed_number[4] <= {8'b0,c11_custom};
    displayed_number[5] <= {8'b0,c12_custom};
    displayed_number[6] <= {8'b0,c21_custom};
    displayed_number[7] <= {8'b0,c22_custom};
end
```

```
case(LED_activating_counter)
2'b00: begin
    Anode_Activate = 4'b1000;
    // activate LED1 and Deactivate LED2, LED3, LED4
    LED_BCD = displayed_number[sel[2:0]]/1000;
    // the first digit of the 16-bit number
    end
```

(sel[3:0] signal은 1초마다 1씩 증가하여 배열의 index로 사용되는데, 이때 하위 3bit만 index로 사용된다.)

모든data를 출력하게 되면(sel = 4'b1000) is_done_o이라는 또다른 signal의 값이 1이 되며, 외부의 controller는 이를 판독하여 display가 끝났음을 인지한다.

```
always @(posedge clock_100Mhz or posedge reset)
begin
    if(reset==1)
        sel <= 4'b0;
    else if(one_second_enable==1)
        sel <= sel + 1;
end
assign is_done_o = (sel == 4'b1000);
```

Figure 35-Display Module Code

5. Controller

Controller의 전체적인 state flow는 위와 같다. Memory initialize에 25개의 state, Serial-mode에 6개의 state, Systolic-mode에 9개의 state, Custom-mode에 2개의 state, Display mode에 2개의 state로 총 44개의 state로 진행된다. Memory initialize를 하는 과정을 제외하면 Controller에서 특정 data의 address를 출력하지 않는다. **이는 각 연산 mode의 Data-Loader에서 처리하며, Controller는 각 stride에 맞는 적절한 base address와 stride의 완료/진행을 제어한다. (해당 HW에서는 Moore Machine을 설계에 채택하였다.)**

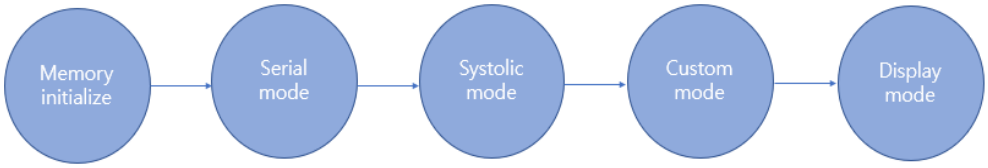


Figure 36-Controller Overall State Flow

5-1) Memory Initialize

Memory address map에서 정한 순서에 맞게 data를 initialize한다. 이 값들은 memory interface로 전해져 memory에 data write을 진행한다.

	S_MEM_INIT0	S_MEM_INIT1	S_MEM_INIT2	...	S_MEM_INIT9	S_MEM_INIT10	S_MEM_INIT11	...	S_MEM_INIT24
addr	6'b00_0000	6'b00_0001	6'b00_0010	...	6'b00_1001	6'b00_1010	6'b00_1011	...	6'b01_1000
data	B_{11}	B_{21}	B_{31}	...	A_{11}	A_{12}	A_{13}	...	A_{33}

Table 6-Memory Initialize State

5-2) Serial-mode

Serial-mode의 경우, striding 단계별로 해당 base-address를 Serial-mode에 전달한다. 이때 각 striding이 완료됨을 인지하고 다음 striding 단계로 넘어가기 위한 제어 신호는 Serial-mode-done 신호로, Serial-data-Loader의 내부 counter에서 출력하는 is_done_o을 사용한다.

	Serial_mode_stride_1	Serial_mode_stride_2	Serial_mode_stride_3	Serial_mode_stride_4	Serial_mode_wait	Serial_mode_done
rst_computation_module	1'b0	--	--	--	--	--
Mem_sel	1'b1	--	--	--	--	--
Serial_mode_en	1'b1	--	--	--	--	--
Computation_mode_sel	2'b00	--	--	--	--	--
Serial_mode_feature_baseaddr	6'b00_1001	6'b00_1010	6'b00_1101	6'b00_1110	--	--

Table 7-Serial-mode State Table

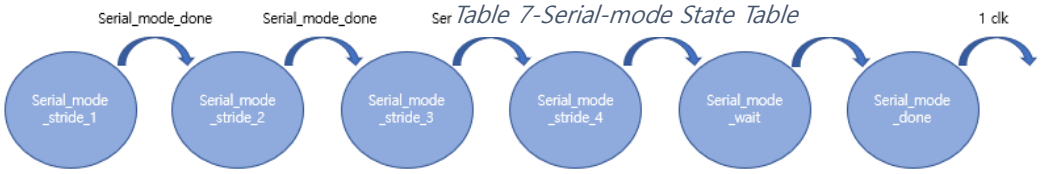


Figure 37-Serial-mode Finite State Machine

5-3) Systolic-mode

Systolic-mode의 경우 Serial-mode와 유사하나 Weight를 preload하기 위한 Weight-Preload 단계가 추가되었다. Systolic-mode 역시 마찬가지로 각 striding이 완료되었음을 인지하고 다음 striding 단계로 넘어가기 위한 제어 신호는 Weight-Preloader/Feature-loader_done 신호로, SA-Data-Loader의 내부의 각 counter에 의해 결정되는 'is_WL/FL_done_o' signal을 이용했다.

	Systolic_mode_weight_preload	Systolic_mode_stride_1	Systolic_mode_stride_2	Systolic_mode_stride_3	Systolic_mode_stride_4	Systolic_mode_wait_1	Systolic_mode_wait_2	Systolic_mode_wait_3	Serial_mode_done
rst_computation_module	1'b0	--	--	--	--	--	--	--	--
Mem_sel	1'b1	--	--	--	--	--	--	-	--
Weight_Preloader_en	1'b1	1'b0	--	--	--	--	--	--	--
Feature_loader_en	1'b0	1'b1	1'b1	1'b1	1'b1	1'b0	--	--	--
c_reg_sel	2'b00	2'b00	2'b01	2'b10	2'b11	--	--	--	--
Systolic_mode	1'b0	1'b1	--	--	--	--	--	--	--
Computation_mode_sel	2'b01	--	--	--	--	--	--	--	--
Systolic_mode_feature_baseaddr	--	6'b00_1001	6'b00_1010	6'b00_1101	6'b00_1110	--	--	--	--

Table 8-Systolic-mode State Table

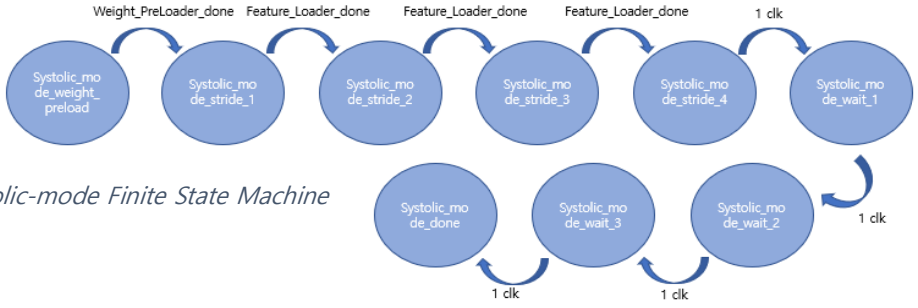


Figure 38-Systolic-mode Finite State Machine

5-4)Custom-mode

Custom-mode는 내부의 Custom-Data-Loader에서 한번의 실행으로 모든 stride를 한꺼번에 계산하므로, Con troller는 그저 동작의 시작과 종료만을 제어하면 된다.

	Custom_mode_en	Custom_mode_done
rst_computation_module	1'b0	1'b1
Mem_sel	1'b1	--
Custom_mode_en	1'b1	1'b0
Computation_mode_sel	2'b10	--

Table 9-Custom-mode State Table

5-5)Display_mode

Display_mode 역시 Controller는 동작의 시작과 종료만을 제어하면 된다.

	Display_mode_en	Display_mode_done
rst_display_module	1'b0	1'b1
Mem_sel	1'b1	--
Display_mode_reg_en	1'b1	1'b0
Computation_mode_sel	2'b10	2'b00

Table 10-Display-mode State Table

6. Simulation

Simulation을 위한 Input matrix와 Weight matrix는 다음과 같이 설정한다. 해당 값들은 simulation에 한해서 memory module의 내부에서 initial statement를 사용하여 초기화한다.

1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4

1	2	3
1	2	3
1	2	3

또한 performance 측정을 위해 speedup을 다음과 같이 정의한다.

$$X = Speedup_{BA} = \frac{Executiontime_B}{Executiontime_A} = \frac{DataLoadTime_B + CompuTationTime_B}{DataLoadTime_A + CompuTationTime_A}$$

Data를 memory로부터 Load하는 시간을 성능평가에 포함하기에, 각 연산 mode를 simulation 함에 있어 각 **연산 module과 memory를 하나의 top module로 선언한 후** testbench의 DUT로써 사용한다. 위에서 언급했듯이 memory는 별도의 test plan 없이 내부적으로 initial statement를 사용하여 적절히 초기화한다.

6-1) Memory

모든 Filter와 Feature value에 대한 memory initialize는 top_module simulation에서 진행하도록 한다. 본 항목에서는 임의의 5개의 data를 write/read 함으로써 **write/read latency**가 얼마나 걸리는지에 중점을 두어 측정하였다.

```
// read

#10 // read addr = 6'b00_0000
    we      = 1'b0;
    addr    = 6'b00_0000;
#10 // read addr = 6'b00_0001
    addr    = 6'b00_0001;
#10 // read addr = 6'b00_0010
    addr    = 6'b00_0010;
#10 // read addr = 6'b00_0011
    addr    = 6'b00_0011;
#10 // read addr = 6'b00_0100
    addr    = 6'b00_0100;
```

Figure39-Single-Port-Ram TB(read)

```
// write

#10 // write num 0 in addr = 6'b00_0000
    we      = 1'b1;
    addr    = 6'b00_0000;
    data    = 8'd0;
#10 // write num 1 in addr = 6'b00_0001
    addr    = 6'b00_0001;
    data    = 8'd1;
#10 // write num 2 in addr = 6'b00_0010
    addr    = 6'b00_0010;
    data    = 8'd2
#10 // write num 3 in addr = 6'b00_0011
    addr    = 6'b00_0011;
    data    = 8'd3
#10 // write num 4 in addr = 6'b00_0100
    addr    = 6'b00_0100;
    data    = 8'd4
```

Figure 40-Single-Port-Ram TB(write)

(write/read 작업 모두 1 clock cycle 간격으로 진행하였다)

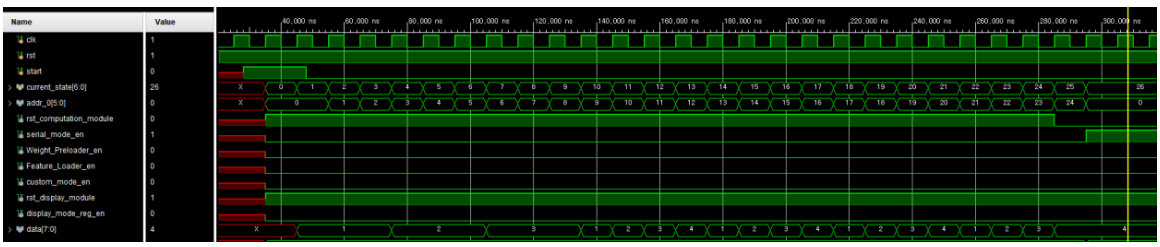
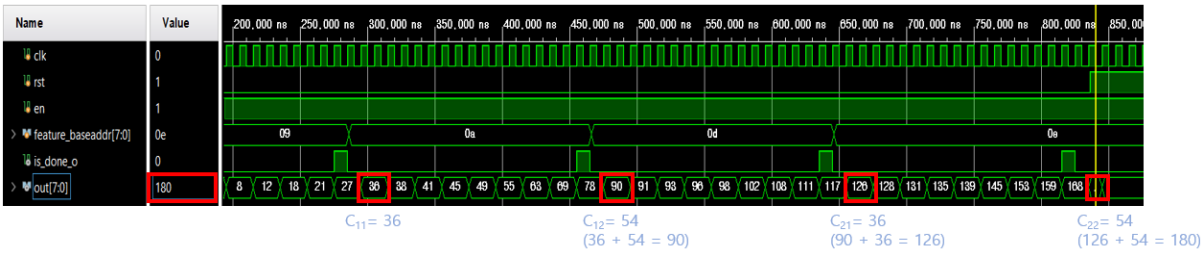


Figure 41-Single-Port-Ram TB Waveform in vivado21.2 simulator

6-2) Serial-mode

Figure 42-Serial-mode TB Hierarchy & TB code

Serial-mode의 경우 enable 신호를 인가함으로 active된다. striding이 끝나면 내부에 있는 Serial-Data-Loader가 'is_done_o' 신호를 high로 출력하여 끝났음을 외부에 알려준다. 이를 wait()문을 사용하여 catch 한 다음, 다음 striding에 필요한 base address를 인가해주는 방식으로 4번의 striding을 진행하였다.



6-3) Systolic-mode

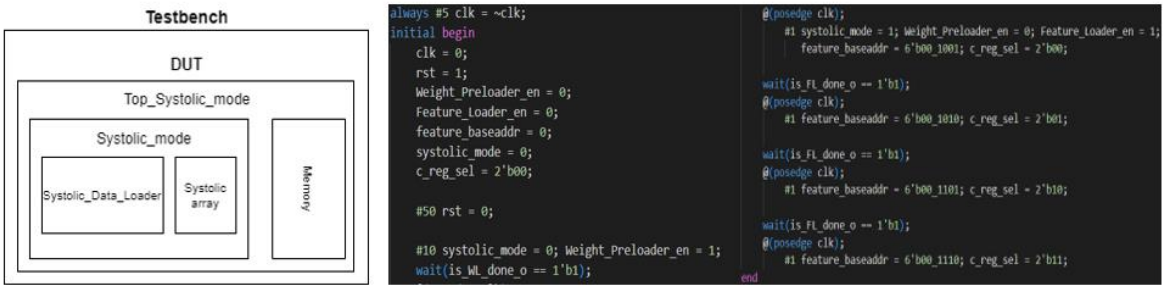


Figure 43-Systolic-mode TB Hierarchy & TB code

연산을 하기 앞서 filter의 값들을 미리 불러오는 'Preload'의 과정이 선행되어야 한다. 따라서 먼저 'Systolic-Data-Loader'내부의 'Weight-Preloader'를 작동시키는 Weight_Preloader_en을 활성화시켜 Preload가 되기까지 기다리며, 마찬가지로 wait()문을 사용하여 완료됨을 확인하면 Feature-Loader를 활성화시켜 연산을 진행한다. 이 역시 Serial-mode와 마찬가지로 wait()문을 통해 striding 단계에 맞는 base-address를 인가한다.

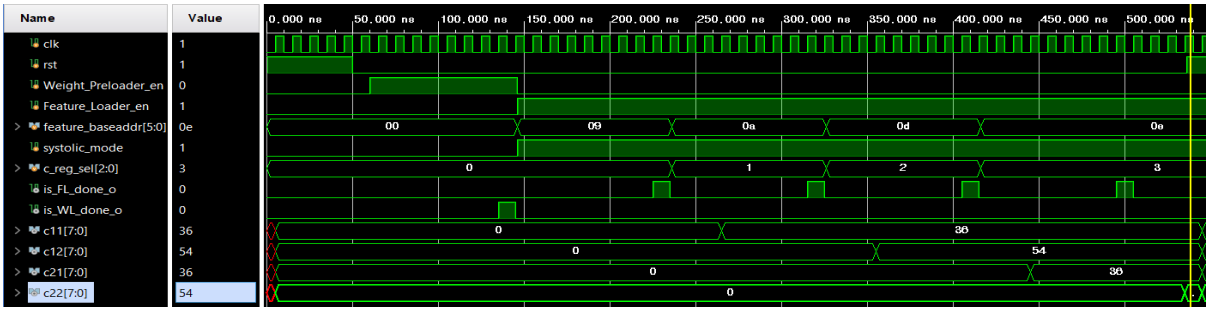
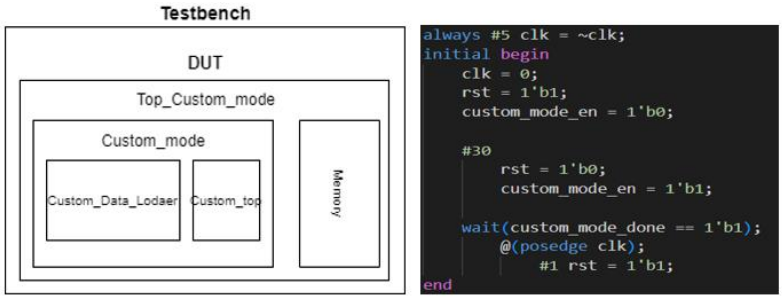


Figure 44-Systolic-mode TB Waveform in vivado21.2 Simulator

6-4) Custom-mode

Custom mode는 한번의 동작으로 모든 Striding연산을 동시에 진행하므로 위 2가지 연산 mode와는 달리 제어해야 할 부분이 적다. 단지 Custom_mode의 연산동작의 시작을 custom_mode_en을 통해 진행시키며, 'Custom_Data_Loader'에서 출력하는 , 동작이 완료됨을 알리는 custom_mode_done 신호를 wait()문을 통해 catch 하여 값을 살펴본다.

Figure 45-Custom-mode TB hierarchy & TB code



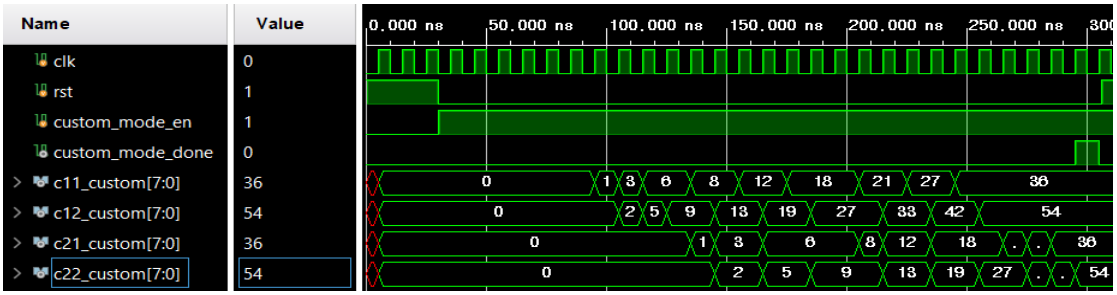


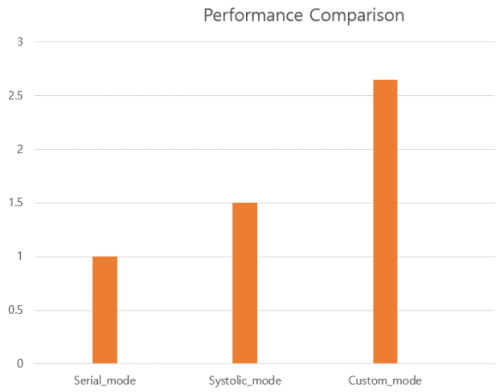
Figure 46-Custom-mode TB Waveform in Vivado21.2 Simulator

6-5) Total Execution time 비교

1) **Serial-mode: 1(74cycle)**
2) **Systolic-mode: 1.50(74/49cycle)**
3) **Custom-mode: 2.65(74/28cycle)**

Table11-Computation-mode

Performance Comparison Graph



6-6) Conclusion

➔ Custom-mode는 memory access를 줄이고, parallel하게 feature들을 동시에 구하는 연산 방식을 채택하여, Serial-mode에 비해서 2.65배, Systolic-mode에 비해서 1.50배 latency가 감소하였다.

6-7) Display module

```
always #5 clk = ~clk;
initial begin
    clk = 0;
    rst_display_module = 1'b1;
    display_mode_reg_en = 1'b0;
    c11_sa = 8'b0;
    c12_sa = 8'b0;
    c21_sa = 8'b0;
    c22_sa = 8'b0;
    c11_custom = 8'b0;
    c12_custom = 8'b0;
    c21_custom = 8'b0;
    c22_custom = 8'b0;

    #30
        rst_display_module = 1'b0;
```

```
    #30
        rst_display_module = 1'b0;
    #30
        @(posedge clk);
            #1  display_mode_reg_en = 1'b1;
                c11_sa = 8'd36;
                c12_sa = 8'd54;
                c21_sa = 8'd36;
                c22_sa = 8'd54;
                c11_custom = 8'd36;
                c12_custom = 8'd54;
                c21_custom = 8'd36;
                c22_custom = 8'd54;
        wait(display_done == 1'b1);
            @(posedge clk);
                #1 rst_display_module = 1'b1;
                   display_mode_reg_en = 1'b0;
end
```

먼저 simulation 결과를 간편하게 관측하기 위해, simulation 상에서만 **Display module의 'one_second_counter'의 주기를 12cycle, 'refresh-counter'의 주기를 4cycle로 조정**하여 측정하였다.

Figure 47-Display-module TB code

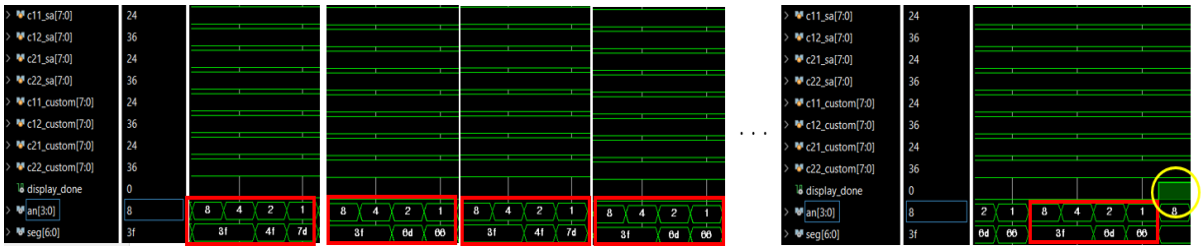


Figure 48-Display-module TB Waveform in Vivado21.2 Simulator

8개의 input에 대한 올바른 an/seg값이 출력됨을 볼 수 있으며, 8초(해당 simulation에서는 12 * 8 = 96 clock cycle)이 지난 후 display_done signal이 1'b1로 활성화됨을 볼 수 있다.

6-8) Controller

```
#3   start = 1'b1;    // give start signal
@(posedge clk);
@(posedge clk);
#3   start = 1'b0;

// wait initialize memory(>25)
repeat(26) begin
    @(posedge clk);
end
```

1. 동작의 시작을 알리는 start 신호를 인가함으로써 Controller의 state는 'Memory Initialize'(25 states) 를 진행한다.

Figure 50-Controller TB code(Initialize)

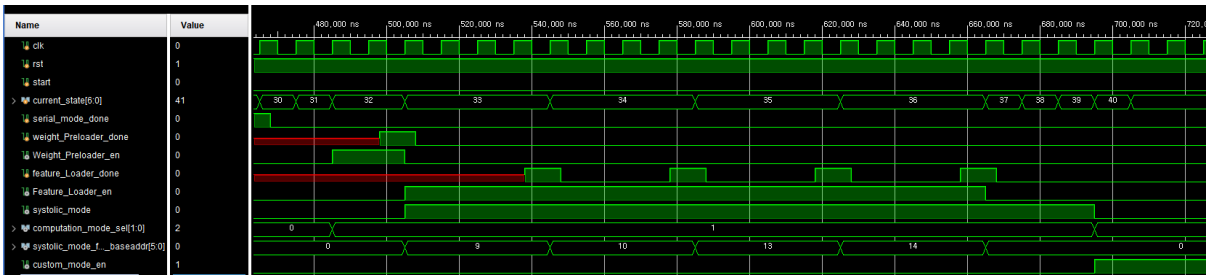


Figure 49-Controller TB Waveform(Initialize) in Vivado21.2 Simulator

```
// wait serial_mode 1st stride
repeat(3) begin
    @(posedge clk);
end
// Here control must be wait for serial 1st stride done signal
#3  serial_mode_done = 1'b1;    // give done signal
@(posedge clk);
#3  serial_mode_done = 1'b0;    // done signal off


// wait serial_mode 2nd stride
repeat(3) begin
    @(posedge clk);
end
```

Figure 52-Controller TB code(Serial-mode)

2. Memory initialize 가 완료되면 Controller 는 자동으로 Serial-mode state(6 states) 로 넘어간다. Serial-mode는 4번의 Striding을 진행하는데, 각 Striding이 종료될 때마다 이를 알리는 serial_mode_done 신호를 인가해야 정확하게 동작한다.

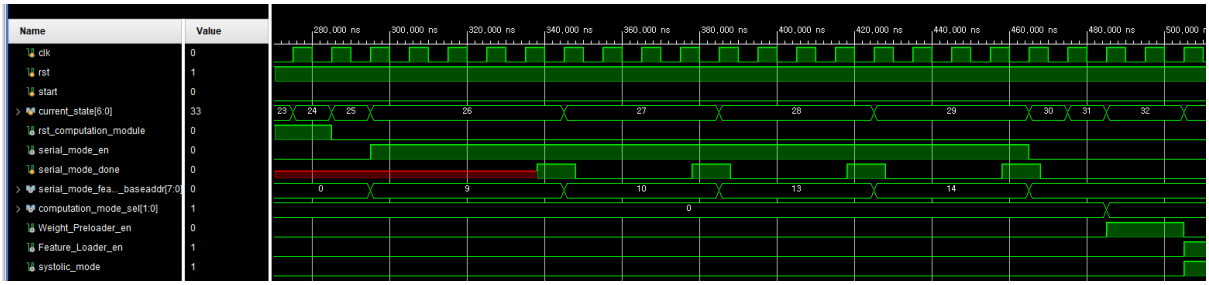


Figure 51-Controller TB Waveform(Serial-mode) in Vivddo21.2 Simulator

```
// serial finished -> systolic weight preload start //

// wait weight preload
repeat(3) begin
    @(posedge clk);
end
// Here control wait for weight_preloader_done signal
#3  weight_Preloader_done = 1'b1;
@(posedge clk);
#3  weight_Preloader_done = 1'b0;


// weight preload finished -> systolic stirde start //

// wait stride1 feature preload
repeat(3) begin
    @(posedge clk);
end
// Here control wait for feature_preloader_done signal
#3  feature_Loader_done = 1'b1;
@(posedge clk);
#3  feature_Loader_done = 1'b0;


// wait stride2 feature preload
repeat(3) begin
    @(posedge clk);
```

Figure 53-Controller TB code(Systolic-mode)

3. Serial-mode가 완료되면 Controller는 자동으로 Systolic mode state(9 states)로 넘어간다. Systolic-mode는 앞서 설명한대로 Weight_ preload를 마쳐야 striding을 진행할 수 있기에 먼저 Weight_Preload_en신호를 통해 먼저 Preload 동작을 완료한다. 그후 Systolic_-mode는 4번의 Striding을 진행하는데, 각 Striding이 종료될 때마다 이를 알리는 feature_loader_done 신호를 인가해야 정확하게 동작한다.

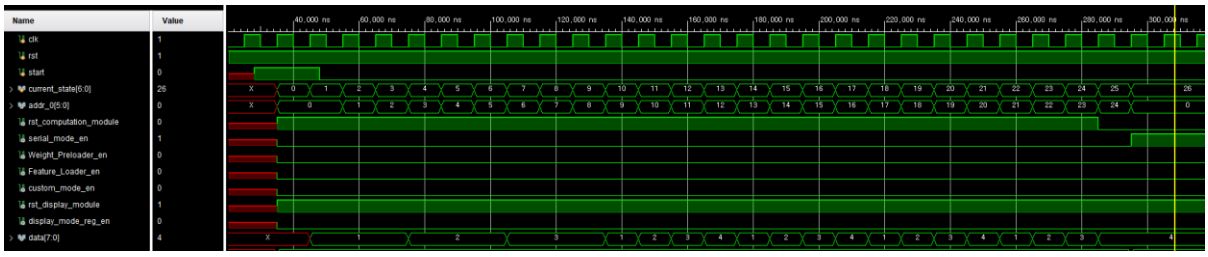


Figure 54-Controller TB Waveform(Systolic-mode) in Vivado21.2 Simulator

```
//! systolic operation need 3cycle delay for getting output feature
repeat(4) begin
    @(posedge clk);
end


// systolic finisehd -> custom start //

// wait custom mode operation
repeat(5) begin
    @(posedge clk);
end
// give custom mode done signal
#3  custom_mode_done = 1'b1;
```

4. Systolic_mode가 끝나면 Controller는 자동으로 custom_mode state로 넘어간다. Custom_mode_ state는 별도의 조작없이 모든 striding을 병렬적으로 진행함으로 본 simulation에서는 5 clock cycle에 custom_mode가 종료된다고 가정하겠다. 따라서 5 clock cycle 이후 다음 state(display state)로 이동할 수 있도록 custom_mode_done 신호를 인가한다.

Figure 56-Controller TB code(Custom-mode)

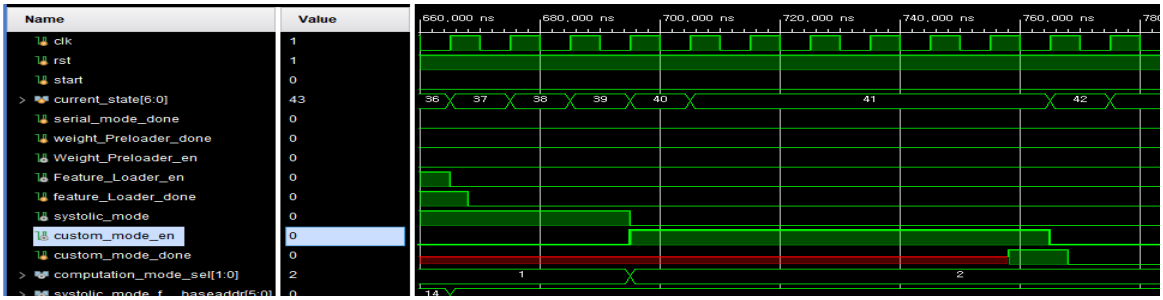


Figure 55-Controller TB Waveform(Custom-mode) in Vivado21.2 Simulator

```
// custom mode finishsed -> Display mode start //
// wait display module activation
repeat(3) begin
    @(posedge clk);
end
// give display mode done signal
#3  display_done = 1'b1;
@(posedge clk);
#3  display_done = 1'b0;
```

5. Display state로 진입한 Controller는 일정 시간(실제 동작은 8초, simulation에서는 1 clock cycle로 가정)이 지나면 display_done signal을 인가받아 전체 Controller의 동작을 마무리한다.

Figure 57-Controller TB code(Display-mode)

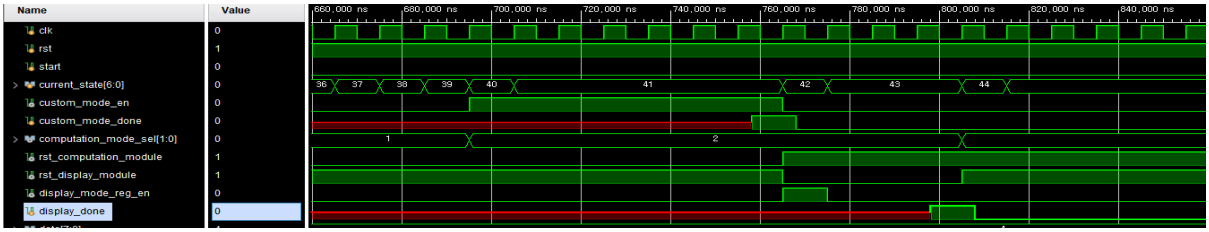


Figure 58-Controller TB Waveform(Display-mode) in Vivado21.2 Simulator

6-9 Top-module

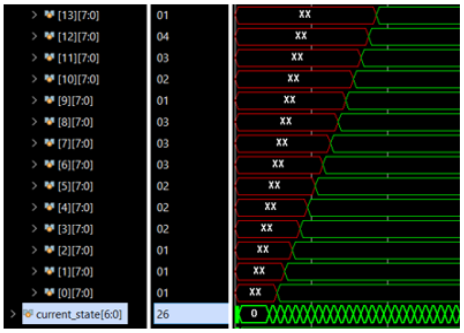
Display-mode simulation에서와 유사하게 waveform을 간편하게 관측하기 위해, simulation 상에서만 **Display module의 'one_second_counter'의 주기를 4cycle, 'refresh-counter'의 주기를 1cycle로 조정**하여 측정하였다. Top module의 test plane에서 확인하고 하는 것은 총 4가지로, **1) memory에 정확히 Initialize되는가, 2) Systolic_mode와 3) custom_mode의 output이 정확히 출력되는가, 4) 앞의 2 가지 연산 mode에서 나오는 output이 정확히 display module의 입력으로 들어가 내부 배열에 저장이 되는가를 확인할 것이다.** 따라서 Top module의 in/out을 제외한 추가적인 내부 instance module들의 in/out을 같이 관찰한다.

```
always #5 clk = ~clk;
initial begin
    clk = 0; sw[0] = 0; sw[1] = 0;
    #30 sw[0] = 1;
    #10 sw[1] = 1'b1;
end
```

해당 testbench에서 sw[0]는 controller의 reset과 연결되며, sw[1]는 controller의 'input start'와 연결된다. 일정 시간 후 sw[0]을 1로 인가하여 controller의 reset 상태를 해제하고(controller reset = active-low) sw[1]을 인가하여 controller의 동작을 시작한다.

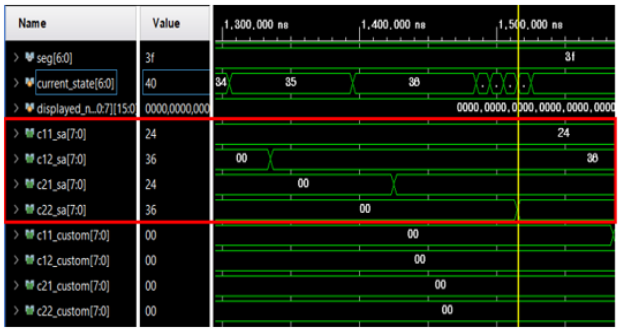
Figure 60-Top-module TB code

- 1st Memory Initialize



- Memory address map에 알맞게 data가 고 있음을 볼 수 있다.

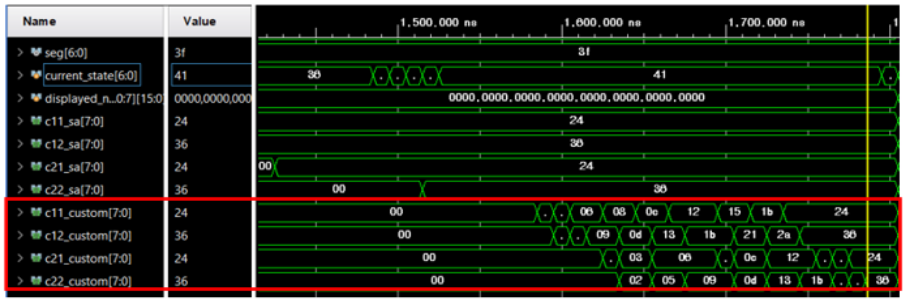
- 2nd Systolic_mode



- 40번째 state에서 4번의 striding이 완료되며 저장되 Systolic_mode가 완료됨을 볼 수 있다.(36,54,36,54)

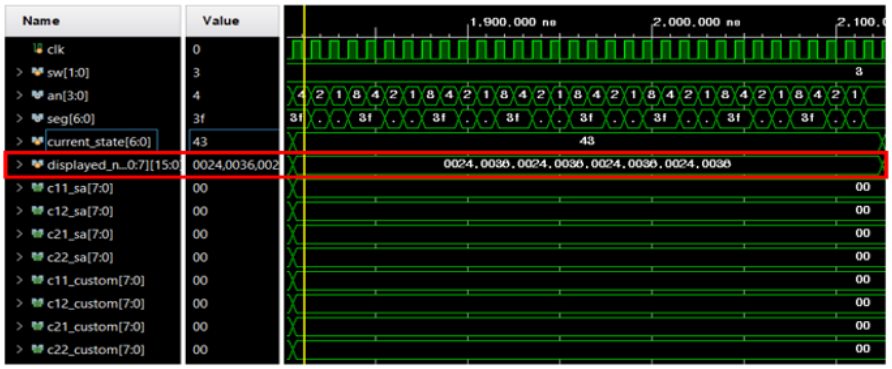
Figure 59-Top-module TB Waveform in Vivado21.2 Simulator(Mem init, Systolic)

- 3rd Custom_mode



- 41번째 state에서 custom_mode의 연산이 완료됨을 볼 수 있다(36,54,36,54)

-4th Display mode



- 43번째 state에서 2가지 연산 mode에서 출력된 8개의 output이 display module 내부의 배열('displayed_number')에 알맞게 저장되었음을 볼 수 있으며 동시에 an/seg가 규칙적으로 displayed_number에 맞게 변화함을 볼 수 있다.

Figure 61-Top-module TB Waveform in Vivado21.2 Simulator

(Custom, Display)

7.Collaboration

1) 역할분담

- PE, Systolic array : 장재성, 홍나경
- Serial/Systolic Data Loader : 박상현
- Controller : 박상현
- Custom mode : 최보열
- Display module : 장재성, 홍나경

2) 회의

1. 2022/04/08, 오프라인, 해동학술정보관 2세미나실, 19:30~21:00

- 전체적인 일정 계획과 방향성 토의, 개발환경 일원화(Git, Github사용법)
- 기초적인 ALU(ex. ADD, MUL ...) 설계 역할 분담

2. 2022/05/06, 오프라인, 해동학술정보관 2세미나실, 13:00~16:00

- PE 설계 관련 토의
- Systolic 구조 관련해서 오인사항 토의
- Custom 구조 관련 토의

3. 32022/05/12, 오프라인, 해동학술정보관 2세미나실, 13:00~16:00

- data loader 개념 도입 및 상호 모듈 간의 interface 논의
- output feature map형성 과정 및 systolic에 추가되는 모듈 토의
- custom block diagram 관련 추가적인 토의

4. 2022/05/19, 오프라인, 수업 실험실, 18:00 ~ 19:30

- FPGA 포팅 실습 및 작동 확인, 보고서 형식에 대한 토의

5. 2022/05/26, 오프라인, 수업 실험실, 14:00 ~ 15:00

- Verified live at the class time -> 검증 완료, 보고서 현황 확인 및 각 코드 주석 검토
- 해당날짜의 14시 00분에 담당 교수님께 직접 FPGA 포팅 검증완료했음.

3) 개발환경

Term Project를 진행함에 있어서 code-conflict 방지, version management, 업무분담 및 개발 진행사항 확인을 위해서 GitHub 공용 repositiory, Git Issue, project기능을 활용하였음.

Github Repo URL: https://github.com/sg05060/SKKU_TermProject