

# 디지털 시스템 설계

## 팀프로젝트 보고서

### [FIR\_filter]

송실대학교

디지털시스템설계(월요일3시 분반)

팀: 5조

팀장: 송O호

팀원: 김O진, 이O백, 한지윤

# 목차

## 1. 프로젝트 개요 및 목표

① 설계 목표

② 프로젝트 개요

## 2. 수행계획 및 역할분담

① 프로젝트 목표 수행을 위한 수행 계획

② 수행 일지

③ 역할 분담 및 개인별 기여도

## 3. 수행 과정

I. 개인프로젝트 진행 및 모듈 완성

II. 팀프로젝트 진행을 위한 3x3 Testbench 작성 후 검증

III. 320x320 레고 블록 이미지 처리용 Testbench 작성 후 검증

IV. FHD(1920x1080) 이미지에 대한 Testbench 작성 후 검증

V. FHD 이미지용 Testbench 성능 향상

i. 일차적으로 PE의 개수를 늘린 후, 병렬적으로 진행

ii. MAC 연산 개선

a. 3x3 이미지 필터링에서의 MAC 연산 개선

b. 320x320 이미지 필터링에서의 MAC 연산 개선

c. 1920x1080 이미지 필터링에서의 MAC 연산 개선

d. MAC 연산 구현하면서 겪은 오류

iii. 두 방법을 합쳐 최대 성능을 향상

# 1. 프로젝트 개요 및 목표

## ① 설계 목표:

FIR filter PE를 이용한 다중처리 FIR filter 시스템 구성 및 성능을 향상한 FIR filter system 구현

## ② 프로젝트 개요:

개인프로젝트를 진행해 모듈을 완성한 뒤 FIR filter의 동작 방식에 대해 알아볼 수 있었다. FIR filter의 동작을 확인하기 위해 FSM과 필터에 대한 block(Module)들을 직접 구성하여 구현한 결과, 이 모듈을 바탕으로 어떤 Testbench를 작성해야 주어진 Macro('do' 파일)와 같은 파형을 출력할지, Filtering 연산 기능을 수행하는지에 대해 생각해볼 수 있었다.

따라서 주어진 프로젝트 가이드에 따라 이를 실행할 때 쓰인 Macro(.do파일)대신 Testbench를 1개, 레고 블록 이미지 출력용 1개로 총 2개의 Testbench를 작성하였다. 그 후 만들어진 Testbench를 바탕으로 320x320 이미지에서 1920x1080이미지에 적용되는 Testbench를 제작하였다. 이때 우리의 설계 목표는 연산속도가 차이 나지 않도록 성능 향상을 한 시스템 구조를 설계하는 것이다. 따라서 처리 속도 향상 방법을 팀원들과 회의를 통해 진행하며 이때의 과정 및 시행착오를 본 보고서에 기록하며 프로젝트 진행 과정에 대해 작성하였다.

## 2. 수행계획 및 역할 분담

### ① 프로젝트 목표 달성을 위한 수행 계획:

- I. 개인프로젝트 진행 및 모듈 완성
- II. 팀프로젝트 진행을 위한 3x3 Testbench 작성 후 검증
- III. 320x320 레고 블록 이미지 처리용 Testbench 작성 후 검증
- IV. FHD(1920x1080) 이미지에 대한 Testbench 작성 후 검증
- V. FHD 이미지용 Testbench 성능 향상
- VI. 보고서 및 발표자료 작성

### ② 수행일지

2023.11.08	팀프로젝트 진행 계획 회의 및 수행계획서 초안 작성
2023.11.11	팀프로젝트 수행계획서 최종안 작성 및 제출
2023.11.15	팀프로젝트를 위한 개인프로젝트 작성 및 3x3 Testbench 검증
2023.11.19	개인프로젝트 완성 및 320x320 Testbench 작성을 위한 회의
2023.11.21	320x320 Testbench 작성 후 검증 완료
2023.11.24	FHD Testbench로의 변환을 위한 회의 및 디버깅 진행
2023.11.26	FHD Testbench 작성 후 검증, 중간발표 준비
2023.11.28	FHD 성능 향상을 위한 회의 <b>김O진, 송O호</b> : MAC 연산 변경/ <b>한O윤, 이O백</b> : PE 4개로 구동
2023.11.30	PE 4개 & 수정한 MAC을 이용해 14배의 성능 향상 구현
2023.12.01	보고서 초안 작성
2023.12.02	최종 보고서 작성
2023.12.03	발표자료 작성

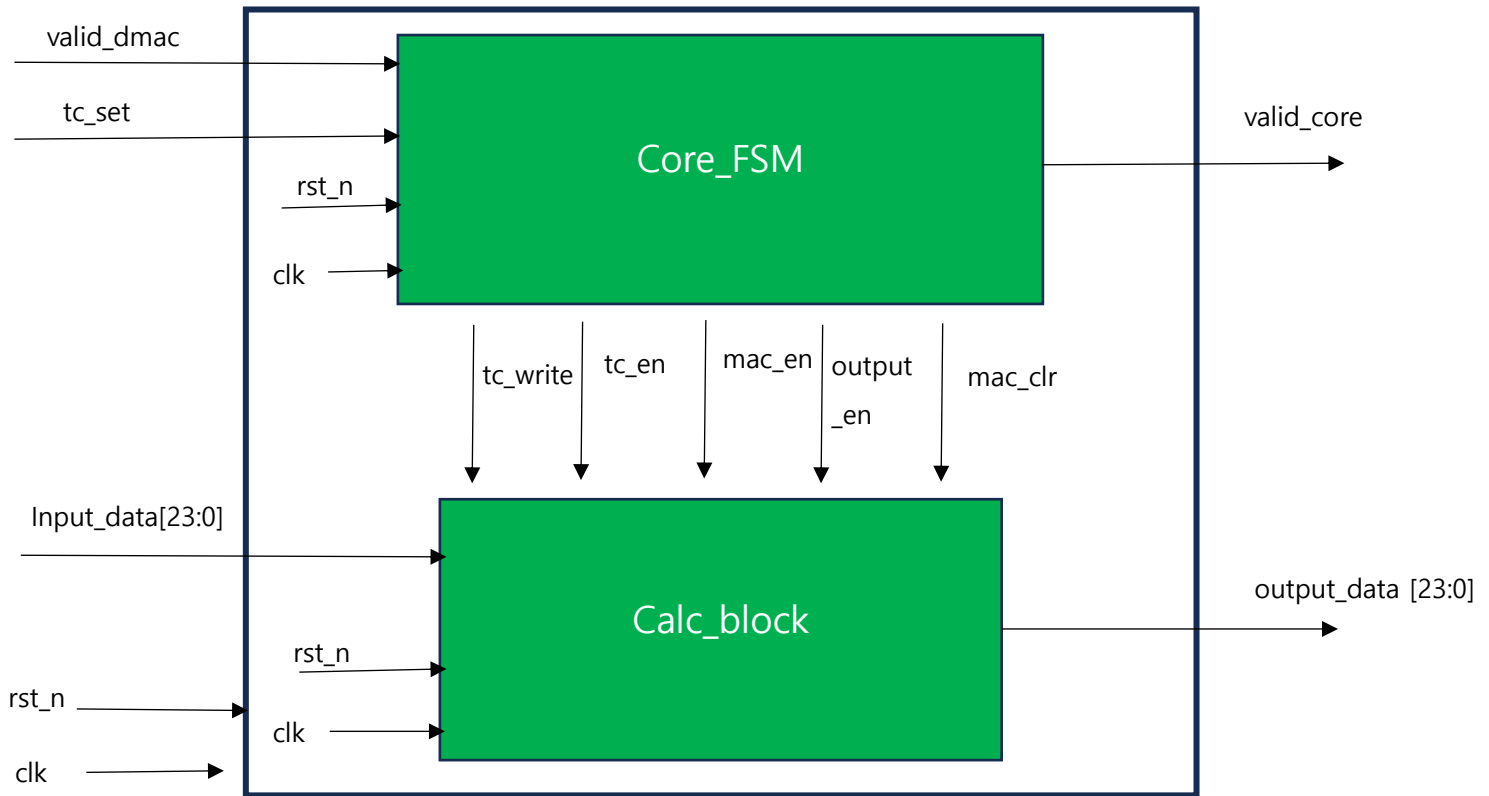
### ③ 역할 분담 및 개인별 기여도

업무	송O호	김O진	이O백	한O윤
개인 프로젝트 완성	O	O	O	O
FIR Filter test로 제공된 Macro의 결과 파형 분석	O	O		
입력 rgb, dat 파일 분석			O	O
Filtering 연산 속도 계산	O	O	O	O
I. Testbench 구현	O	O		
II. Testbench 구현			O	O
III.에서 사용될 연산 방법의 속도 계산(예측)	O	O	O	O
III. PE 재구축 -> PE 자체 늘리는 방법			O	O
III. MAC 연산 재구축 -> 9개 픽셀 일괄 계산방법	O	O		
III.으로 개선된 연산 방법들 검증 및 속도 계산	O	O	O	O
보고서 작성 및 일지 작성	O	O	O	O

	송O호	김O진	이O백	한O윤
기여도				

### 3. 수행 과정

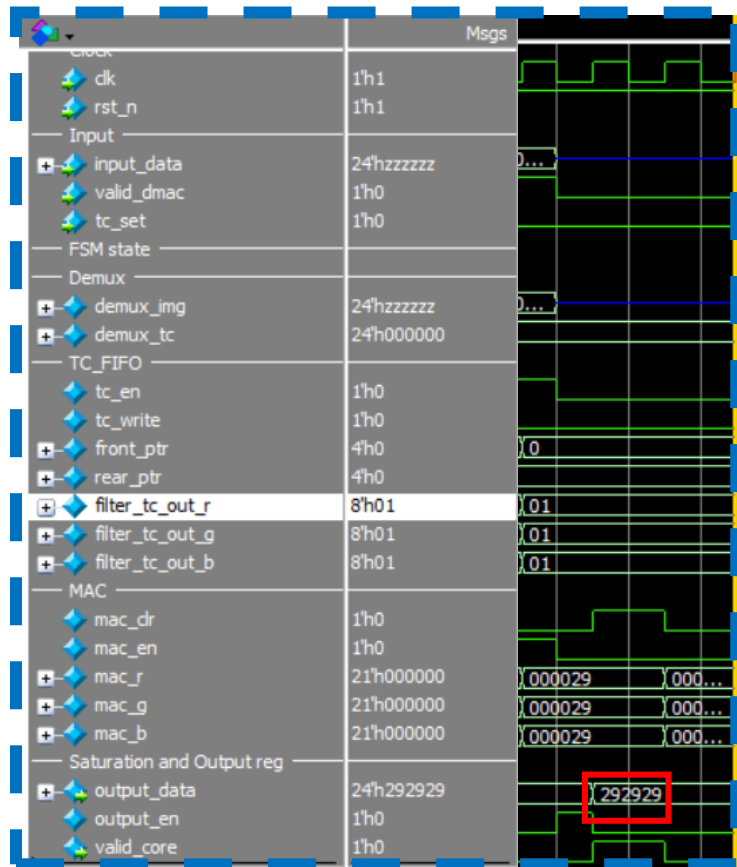
#### I. 개인프로젝트 진행 및 모듈 완성



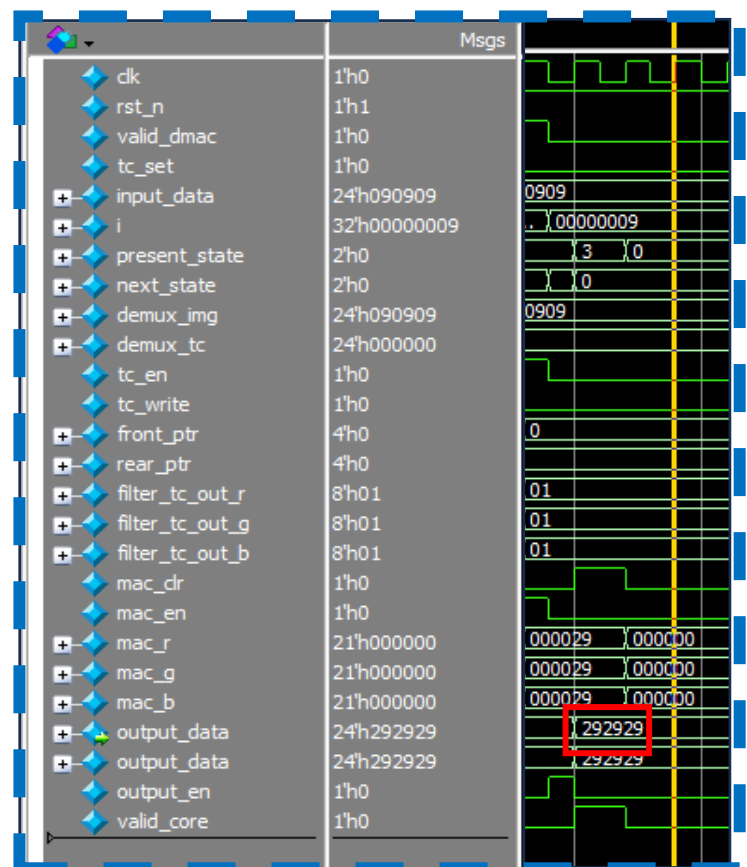
우선 팀프로젝트를 진행하기에 앞서, 각자 개인프로젝트를 진행하며 팀프로젝트를 위한 Module을 제작하였다.

## II. 팀프로젝트 진행을 위한 3x3 Testbench 작성 후 검증

Macro(".do파일)을 이용한 simulation



Testbench를 작성해 구현한 simulation



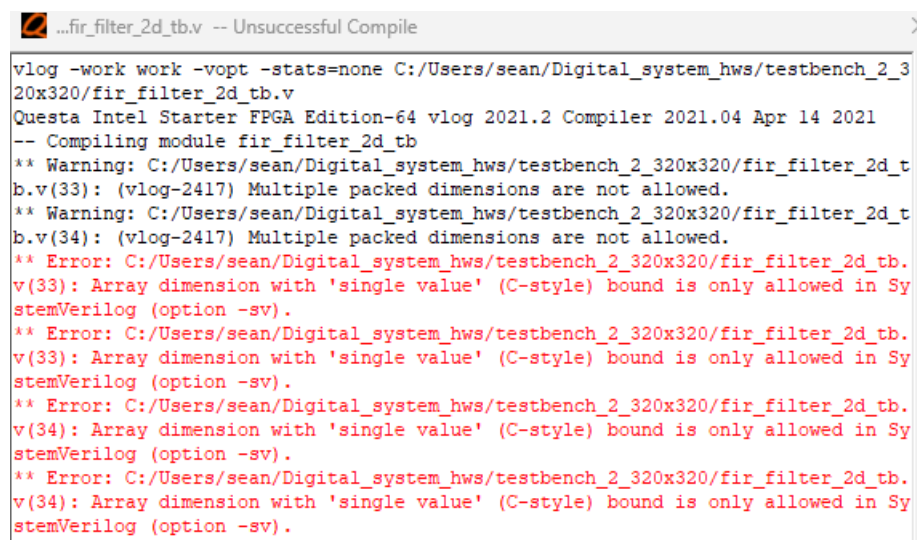
보이는 것과 같이 개인프로젝트를 진행하여 작성한 Module을 바탕으로, Module을 Macro로 돌렸을 때와 같은 output\_data가 나오도록 하는 3x3 Testbench를 성공적으로 작성하였습니다.

### III. 320x320 레고 블록 이미지 처리용 Testbench 작성 후 검증

다음으로는 본 프로젝트인 FHD image에서 실행 예정인 Source\_image로부터 이미지 data를 갖고 온 후, 이를 filtering을 하여 output\_image로 만드는 동작을 구현하기 위해 우선적으로 교수님이 첨부해주신 320x320 레고 블록 Testbench를 먼저 작성하여 보았습니다.

#### 오류 1: 2차원 배열 선언 불가

우선적으로 처음엔 C언어에서와 같이 input memory, zeropadding, output memory 등을 2차원 배열로서 만든 후 진행을 하면 보다 간편할 것 같아, 2차원 배열로 만든 후 진행을 해보았습니다.



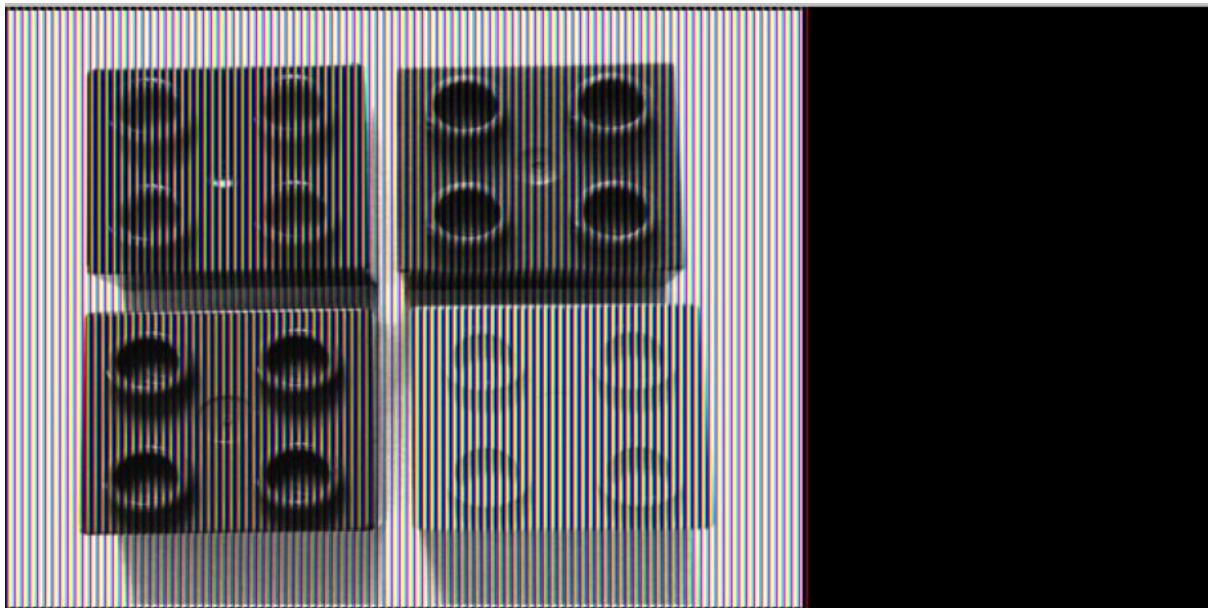
```
vlog -work work -vopt -stats=none C:/Users/sean/Digital_system_hws/testbench_2_320x320/fir_filter_2d_tb.v
Questa Intel Starter FPGA Edition-64 vlog 2021.2 Compiler 2021.04 Apr 14 2021
-- Compiling module fir_filter_2d_tb
** Warning: C:/Users/sean/Digital_system_hws/testbench_2_320x320/fir_filter_2d_tb.v(33): (vlog-2417) Multiple packed dimensions are not allowed.
** Warning: C:/Users/sean/Digital_system_hws/testbench_2_320x320/fir_filter_2d_tb.v(34): (vlog-2417) Multiple packed dimensions are not allowed.
** Error: C:/Users/sean/Digital_system_hws/testbench_2_320x320/fir_filter_2d_tb.v(33): Array dimension with 'single value' (C-style) bound is only allowed in SystemVerilog (option -sv).
** Error: C:/Users/sean/Digital_system_hws/testbench_2_320x320/fir_filter_2d_tb.v(33): Array dimension with 'single value' (C-style) bound is only allowed in SystemVerilog (option -sv).
** Error: C:/Users/sean/Digital_system_hws/testbench_2_320x320/fir_filter_2d_tb.v(34): Array dimension with 'single value' (C-style) bound is only allowed in SystemVerilog (option -sv).
** Error: C:/Users/sean/Digital_system_hws/testbench_2_320x320/fir_filter_2d_tb.v(34): Array dimension with 'single value' (C-style) bound is only allowed in SystemVerilog (option -sv).
```

하지만 이때 다음과 같이 Verilog에서는 2차원 배열 선언이 안 된다는 오류가 나타나, Width\*height의 1차원 array를 만들어 1차원 array로 진행을 하며, 하나의 행마다의 구분을 i, j indexing을 통해 다음 행의 시작 주소를 width를 통해 계산하며 넘어가는 형식으로 진행하도록 수정하였습니다.



### 오류2: \$fwrite 중 형식지정자 '%u' 사용

처음 Testbench를 작성할 때, "\$fwrite"를 이용해 이미지 파일로부터 저희가 저장하고자 하는 memory에 write를 진행하는 과정에서, 우선 강의자료에 나와있는 대로 '%u'로 형식지정자를 설정해 진행을 하였는데, 결과를 확인해보니 아래와 같이 오류로 인해 원본 이미지가 아닌 의도하지 않았던 이미지가 출력되었습니다.



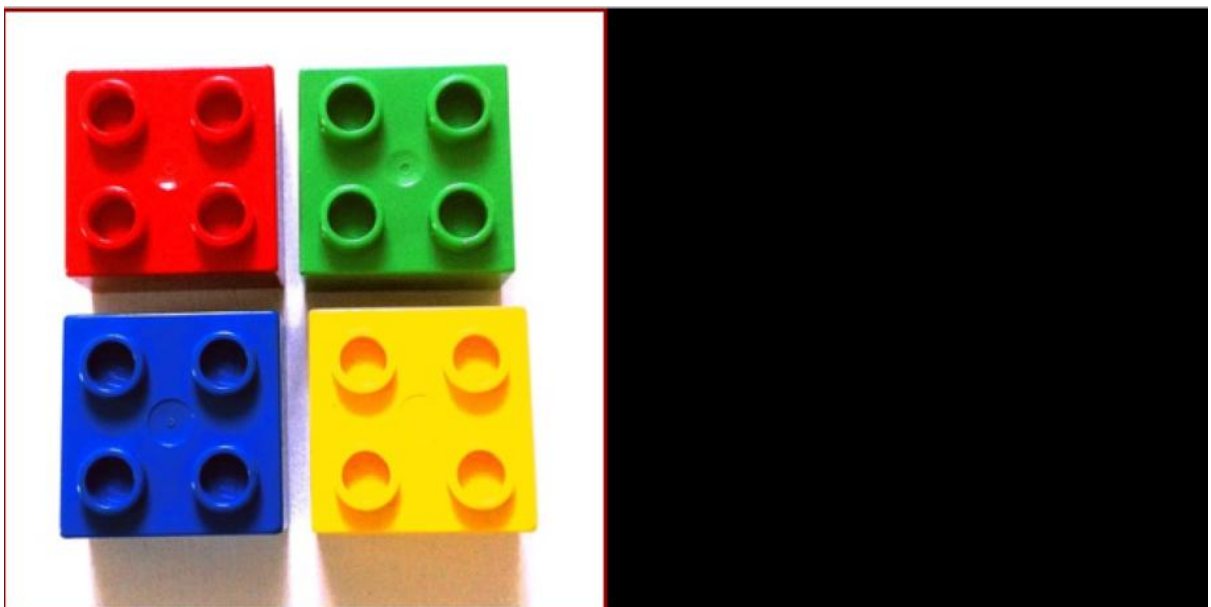
### 오류3: \$fwrite 중 형식지정자 아예 사용 X

위의 오류 이미지가 출력된 뒤, 어디서 오류가 나타난 것인지 몰라 디버깅을 하는 과정 중, 이번에는 형식지정자를 아예 사용하지 않고 바로 이미지를 읽는 방식으로 출력을 해보았습니다. 이때 아래와 같이 오류 이미지이긴 하지만, 오류 이미지의 결과물이 레고 블록이 아주 희미하게 보이는 화면으로 달라졌다는 것을 확인할 수 있었습니다. 이때 여기서 형식지정자의 변화로 인해 이미지의 생김새가 달라졌다는 것을 깨닫게 되어, 형식지정자를 집중적으로 수정하게 되었습니다.



**수정1:** 형식지정자 %c%c%c로 변경

형식지정자 오류에 대해 생각을 해보았을 때, 출력 결과물이 r g b 로 나오므로 output\_memory 각각의 r g b에 대해서 character로 지정해 출력을 하는 방식으로 해보았더니, 아래의 그림과 같이 정상적으로 320x320 -> 640x320의 image 중 왼쪽에 저희가 의도했던 대로 원본 이미지가 정상적으로 나타나는 것을 확인할 수 있었습니다.



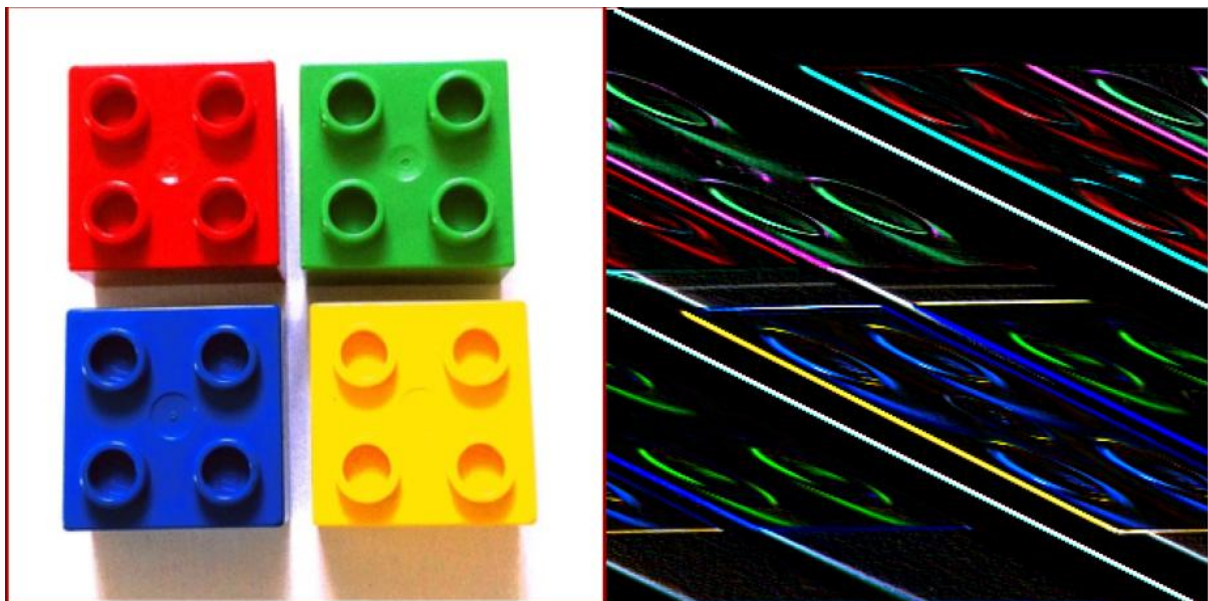
#### 오류4: output\_data [array indexing]

우선 filter\_memory에 넣어뒀던 filter값을 input\_data에 넣어주고, 그 다음으로 input\_data에 zero padding을 고려한 image\_memory를 넣어주면, module 내에서 그 data들을 가지고 동작을 해 output\_data를 내보내줄 것이고,

그것을 다시 최종 output\_image\_memory에 넣어주면 저희가 의도했던 필터를 씌운 이미지를 얻을 수 있을 것이라고 생각을 하고 진행하였습니다.

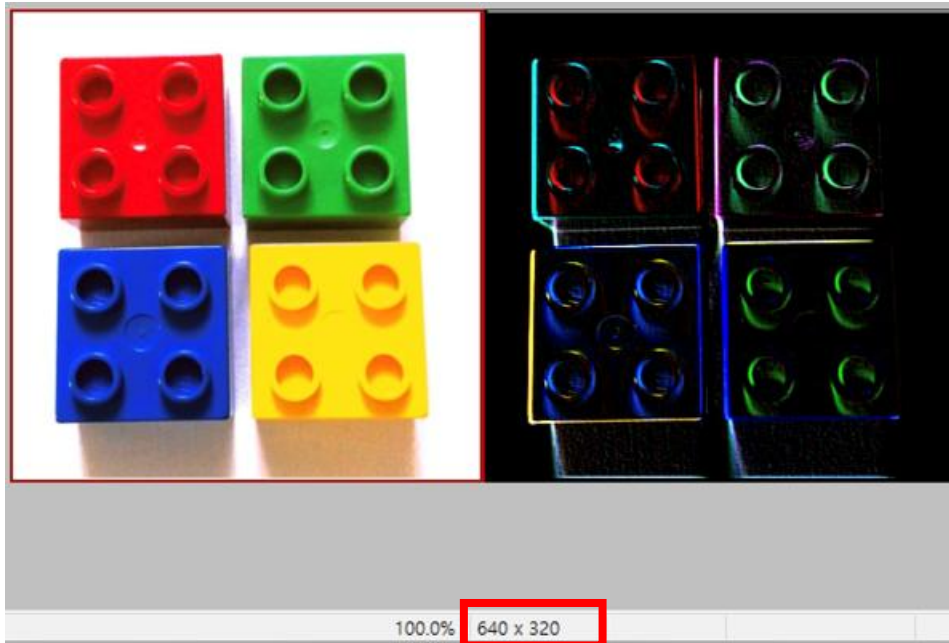
하지만 이때 output\_image\_memory 각각의 array에 넣어줄 output\_data에도 그에 맞는 array indexing을 해주어야 될 것 같다고 생각을 하여, output\_data에도 array indexing을 하는 식으로 진행을 하였습니다.

그 결과 아래와 같이 오른쪽 이미지가 오류가 생겨 오류 이미지로 나오게 되었습니다.



#### 수정2: output\_data에 array indexing 제거

역시 디버깅을 하는 과정을 통해 생각해보니 output\_data는 array indexing이 있으면 안 된다는 것을 깨닫고, 이를 제거하여 결과를 출력해보았더니, 아래와 같이 의도했던 640x320 image를 완성할 수 있었습니다.

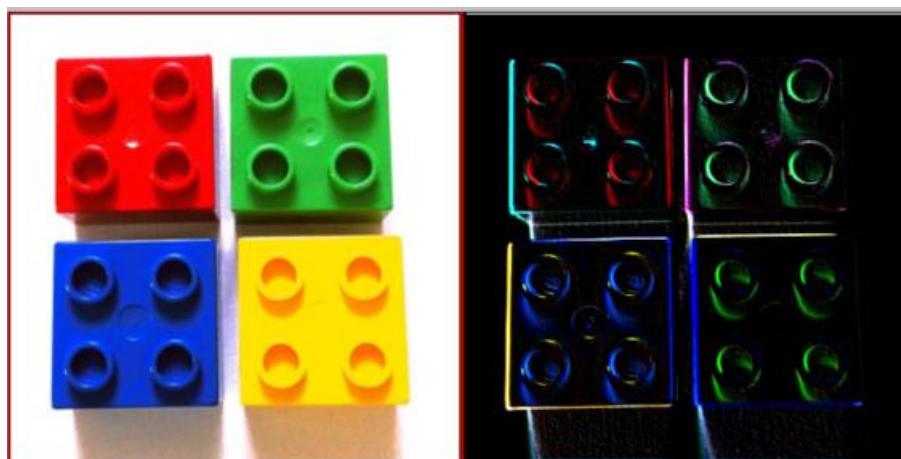


AS	Now	26624470 ns
Cursor 1	Cursor 1	0 ns

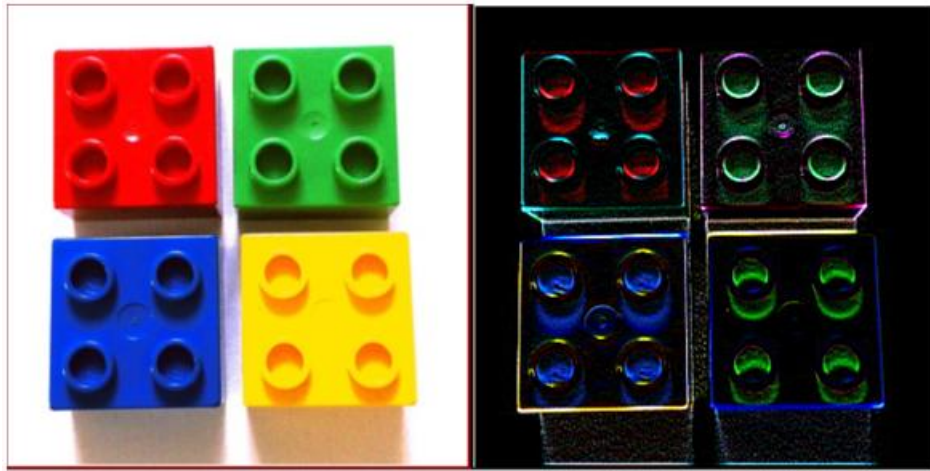
### <320x320에 대한 Simulation 결과>

또한 320x320 사이즈의 Testbench는 다음과 같이 26,624,470ns가 걸리는 것을 확인할 수 있습니다.

이제 결과물을 출력하는 Testbench를 작성하였으므로, 이를 이용해 교수님께서 강의영상에서 말씀하셨던 대로 Sobel Mask와 Laplacian Mask 모두에 진행을 해 보았습니다.

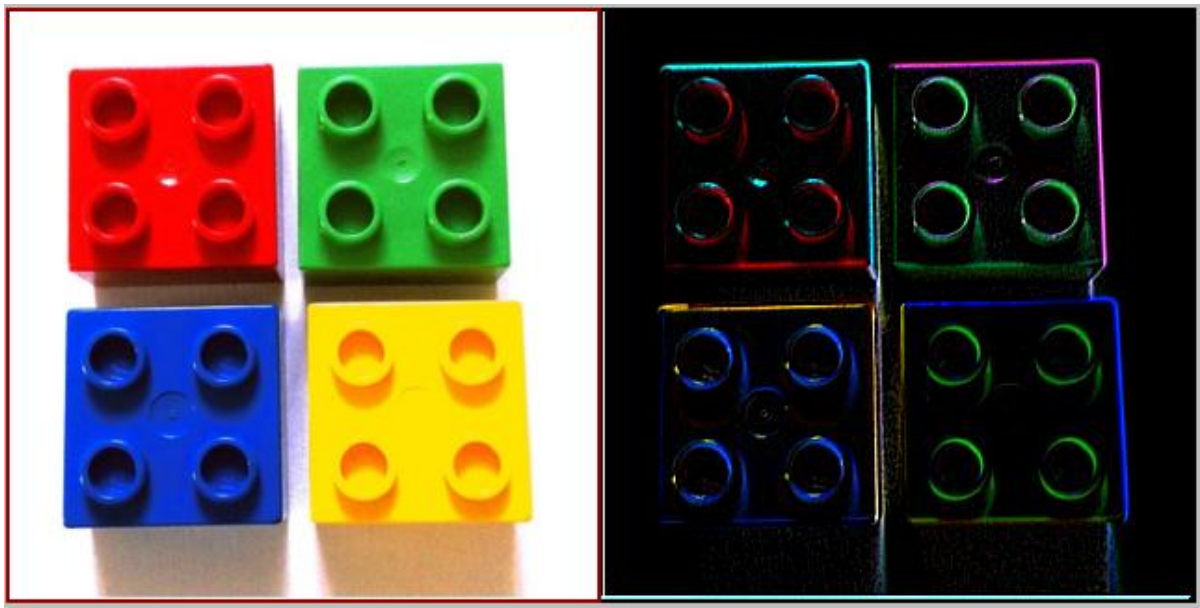


<Sobel Mask 적용 image>



<Laplacian Mask 적용 image>

또한 교수님이 제공해주신 결과물 이미지와 동일한지 비교를 해보았는데, 눈에 띄는 것 중 하나가 제일 하단에 있는 파란 실선이었습니다. 이를 보고 제대로 filtering이 안 된 것인지 의문이 들어, 해당 위치를 simulation을 통해 찾아가보았더니, 이는 교수님이 제공해주신 이미지 파일은 맨 하단 data들에 대해 zero padding이 되어 있지 않다는 것을 알아낼 수 있었습니다.

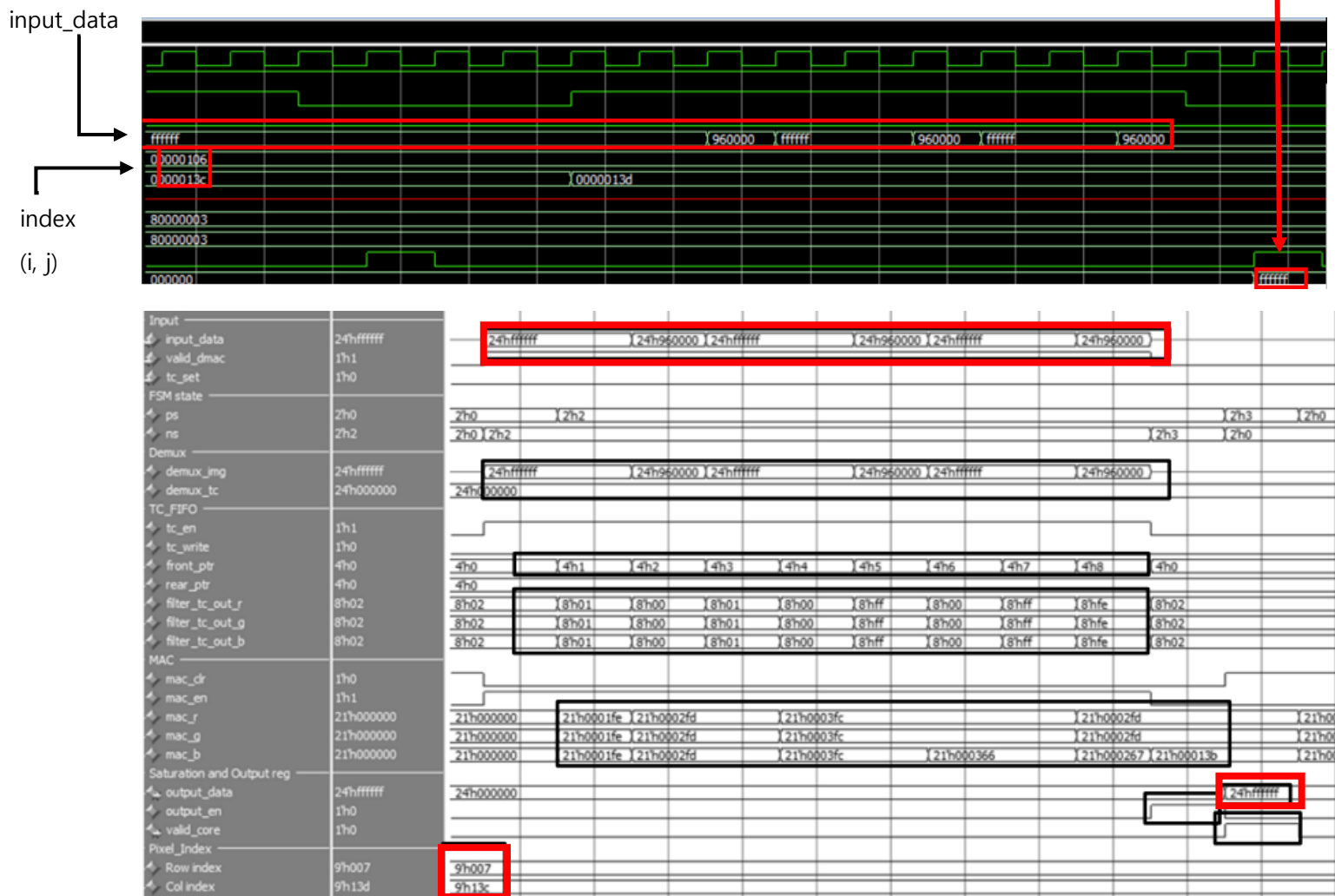


<교수님이 제공해주신 결과물 이미지>

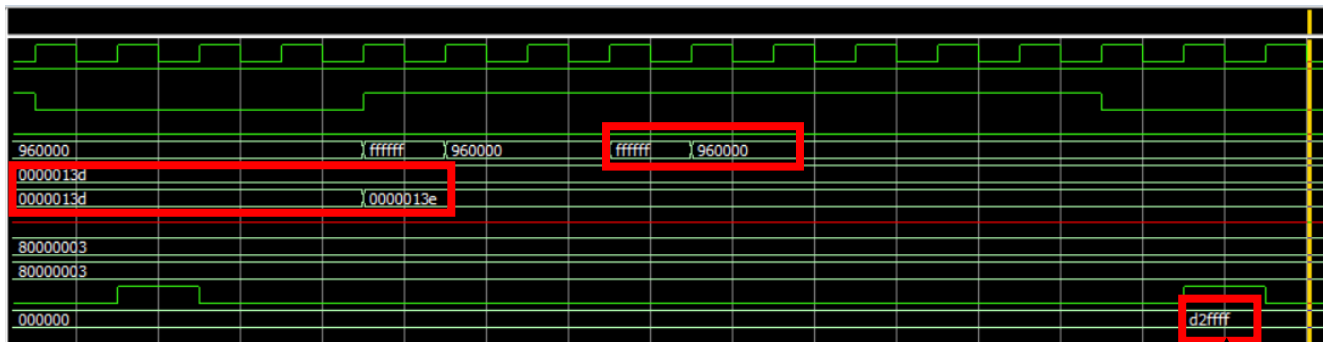


또한 교수님이 제공해주신 simulation 결과와 비교해봤을 때도, zero padding으로 인해 약간의 row, col 차이는 있지만, input이 들어왔을 때의 output이 동일한 것을 통해 성공적으로 구현했음을 또한 알 수 있었습니다.

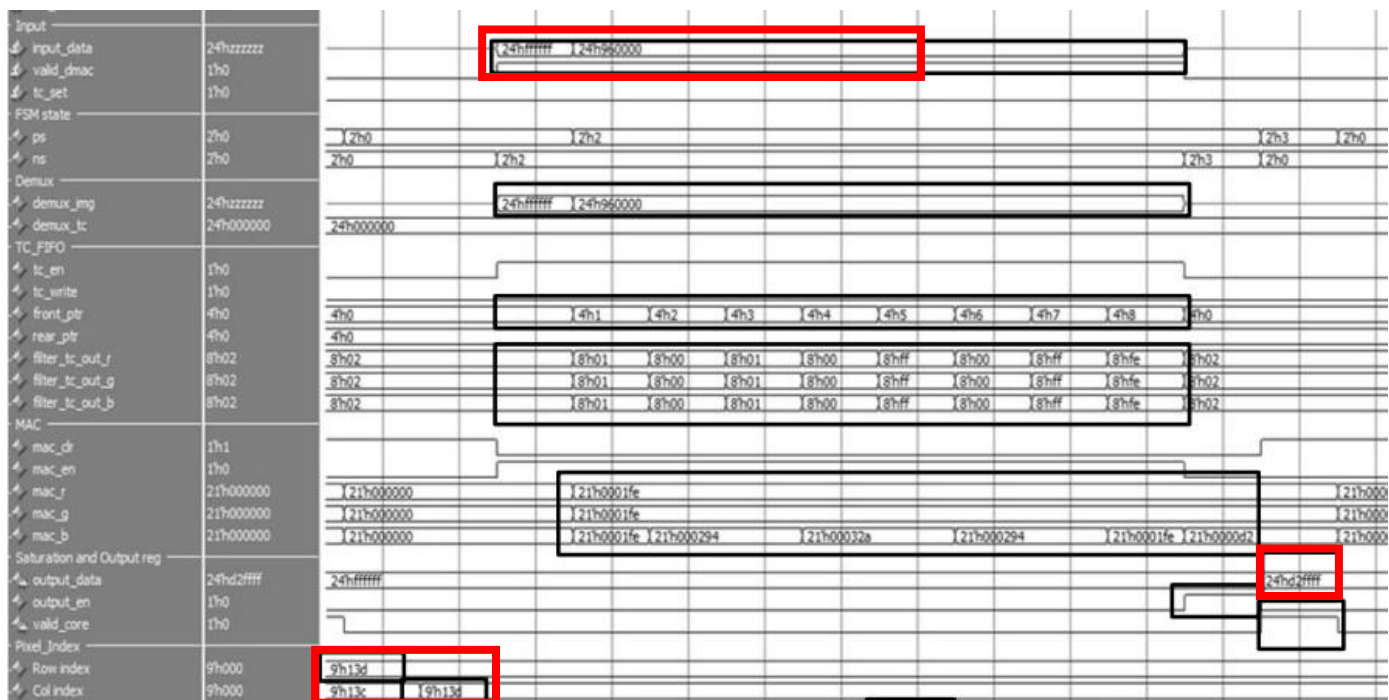
### <아래는 Sobel Mask에 대한 Simulation>



The diagram shows the input\_data and index (i, j) for the matrix element access. The input\_data is a 2D array, and the index (i, j) is a pair of integers. The input\_data is represented by a grid of cells, and the index (i, j) is represented by a pair of arrows pointing to the corresponding cell in the grid.



output\_data



#### IV. FHD(1920x1080) 이미지에 대한 Testbench 작성 후 검증

**오류1:** IMAGE\_HEIGHT, IMAGE\_WIDTH 반대로 설정

320X320 Testbench를 parameter를 이용해 제작하였으므로, 만들어 둔 Testbench에서 parameter 값만 바꾸어주면 쉽게 FHD에 대한 Testbench를 작성할 수 있을 것이라 예상하였다.

따라서 IMAGE에 대한 WIDTH, HEIGHT 값을 1920, 1080으로 수정한 뒤 결과를 출력해보았는데, 아래와 같은 의도하지 않았던 정사각형의 이미지(1080X1080)가 출력되었다.



**수정1:** 2차원 배열 indexing 수정

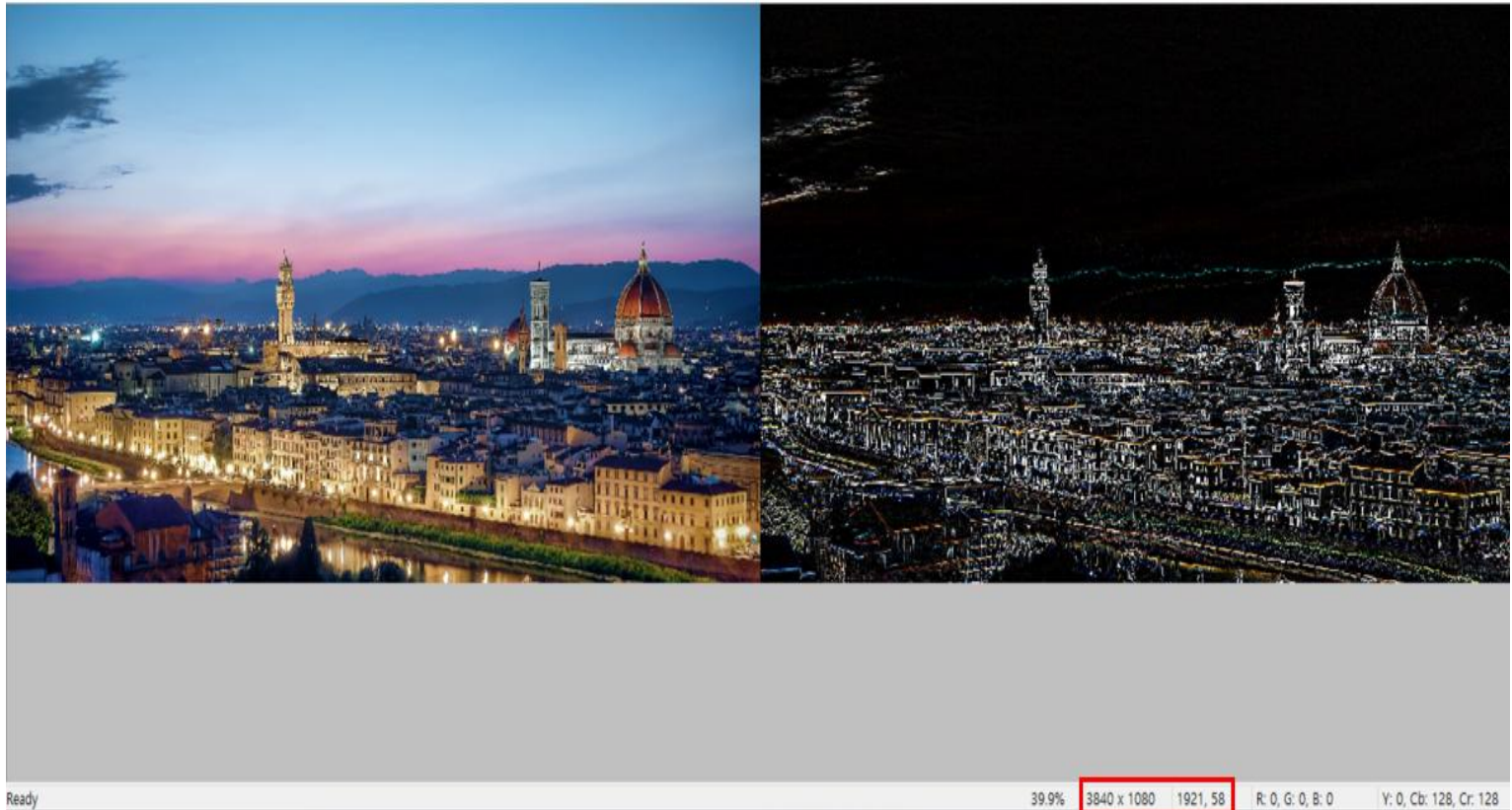
오류에 대한 원인을 디버깅을 해보았더니 Testbench 진행 과정 중, WIDTH와 HEIGHT가 뒤바뀐 곳이 있어 이러한 오류가 나타나는 것을 알 수 있었다.

이전에 진행하였던 Testbench에서는 WIDTH와 HEIGHT가 320으로 같아 오류가 발생하지 않았지만, FHD이미지에서는 뒤바뀐 WIDTH와 HEIGHT로 인해 원본 및 필터링한 이미지가 모두 1080x1080로 나타났다.

Testbench 중 왼쪽 원본 이미지를 output\_image\_memory에 넣는 과정과 오른쪽 필터링 이미지를 위해 필터링을 한 output\_data를 output\_image\_memory에 넣는 과정 속에서 2차원 배열 계산을 output\_image\_memory 자체는 1차원 배열로 만든 후, i, j indexing을 이용해 j를 하나씩 늘려 한 행이 지나가면, i를 통해 다음 행으로 넘어가도록 만들었다.



이때 output\_image는 3840x1080의 사이즈를 가지므로 다음 행으로 넘어가기 위해서는 2\*WIDTH를 지나야 하는데, 이 부분이 뒤바뀌어져 있어 오류가 발생하였다는 것을 알 수 있었다. 따라서 2차원 배열의 한 행마다의 시작 주소를 계산해주는 indexing을 수정하여 아래와 같은 의도했던 이미지를 출력하였다.



Now	539136470 ns
Cursor 1	599234225 ns

#### <FHD에 대한 Simulation 결과>

아무 성능 향상을 하기 전 1920x1080 size에 걸리는 시간은 539,136,480ns가 걸리는 것을 확인할 수 있다. 따라서 320x320 Testbench와 비교하였을 때 20.25배 정도의 차이가 나타나는 것을 확인할 수 있었다.

## V. FHD 이미지용 Testbench 성능 향상

Testbench 성능 향상 방법에 있어 아래와 같은 계획을 세워 진행하였다.

- i. 일차적으로 PE의 개수를 늘린 후, 병렬적으로 진행
- ii. MAC 연산 개선
- iii. 두 방법을 합쳐 최대로 성능을 향상
- iv. 결과 분석

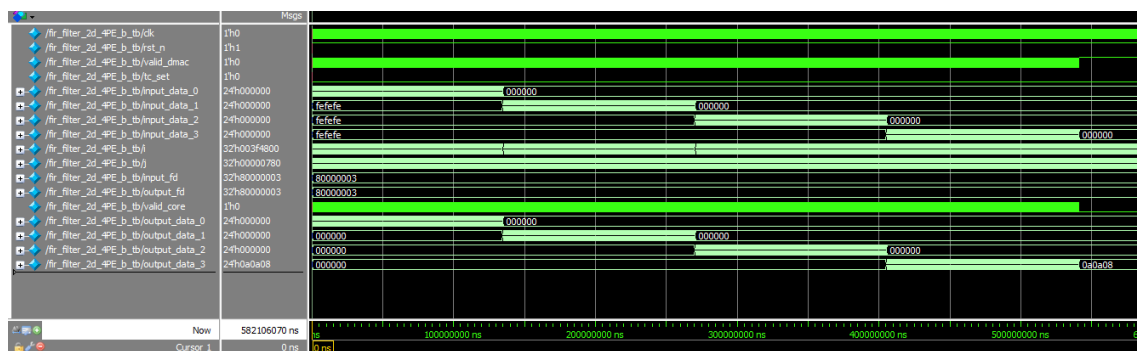
### i. 일차적으로 PE의 개수를 늘린 후, 병렬적으로 진행

**예상1:** 우선 PE 자체를 늘리면 그 늘리는 개수만큼 처리 시간이 감소할 것이라고 예상했다.

#### 오류1

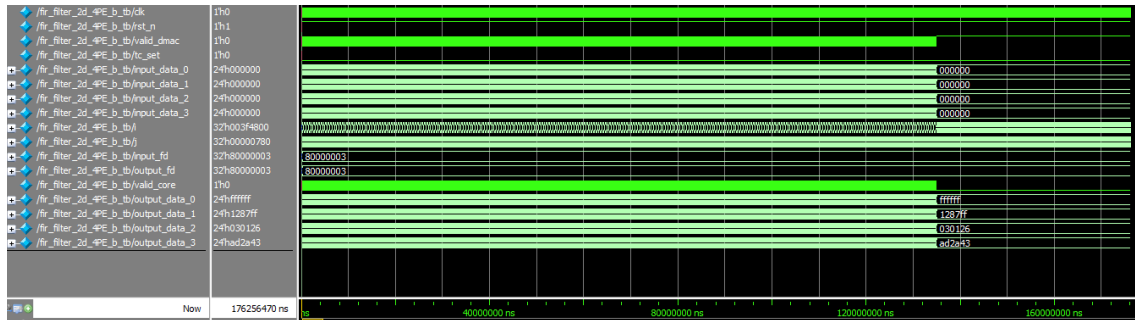
이때 우리는 FHD(1920x1080) 이미지를 1920x270으로 4등분을 하여 4개의 PE로 각 한 부분 씩 필터링을 하도록 하였다.

처음 진행할 때에는 각 PE에 대해 별도의 for loop를 사용하게 했으며, 또한 중간에 output\_data를 저장할 임시 'output\_image\_pe' reg를 4개 생성해 진행하였다. 각 reg는 필터링이 끝나면 병합이 되어 최종 이미지와 합치는, 후처리 방식으로 진행하였다. 하지만 이런 식으로 각 PE를 별도의 for loop를 사용해 진행하였더니, 아래의 simulation 그림에서 볼 수 있듯이, 1개의 PE가 끝나고나서 다음 PE가 진행되는, 즉 의도했던 병렬적 진행이 되지 않는 것을 알 수 있었다.



## 수정2

우리가 의도했던 구현은 4개의 PE가 병렬적으로 진행하는 것이었으므로, PE마다 각각의 loop를 통해 진행하는 것이 아닌, 하나의 단일 loop 내에서 4개의 PE가 동시에 동작하도록 수정을 하였더니, 아래의 simulation과 같이 병렬적으로 처리가 되는 것을 확인할 수 있었다.



## 수정3

하지만 simulation 진행 과정을 분석해 보았을 때, 병렬적으로 PE가 진행함에도 필터링이 완료된 후에 임시 reg 4개를 이용해 저장해 두었다가, 마지막에 병합하는 후처리 방식으로 인해 simulation 마지막 부분에서 시간이 예상보다 오래 걸린다는 것을 확인할 수 있었다.

원인을 분석해본 결과, 이 병합하는 과정에서 시간이 오래 소모되는 것으로 생각되어 중간 이미지를 저장하는 레지스터를 건너뛰고 바로 최종 이미지로 출력되도록 수정을 하였다.



따라서 PE 4개를 사용했을 때 114,048,360ns가 걸리는 것을 확인할 수 있다.

따라서 기존의  $539,136,480 / 114,048,360 = 4.727$ 로 예상했던 것과 비슷하게 4배의 성능 향상을 이루었다는 것을 알 수 있다.

## 예상1에 대한 검증

a) PE 1개 사용(1개당 1920\*1080개 pixel) - 456,192,360ns

00:04:17.72

Now	456192360 ns
Cursor 1	4225779 ns

b) PE 2개 사용(1개당 1920\*540개 pixel) - 228,096,360ns (2배 감소)

00:03:37.78

Now	228096360 ns
Cursor 1	4225780 ns

c) PE 3개 사용(1개당 1920\*360개 pixel) - 152,064,360ns (3배 감소)

00:03:00.94

Now	152064360 ns
Cursor 1	4225779 ns

d) PE 4개 사용(1개당 1920\*270개 pixel) - 114,048,360ns (4배 감소)

00:02:50.49

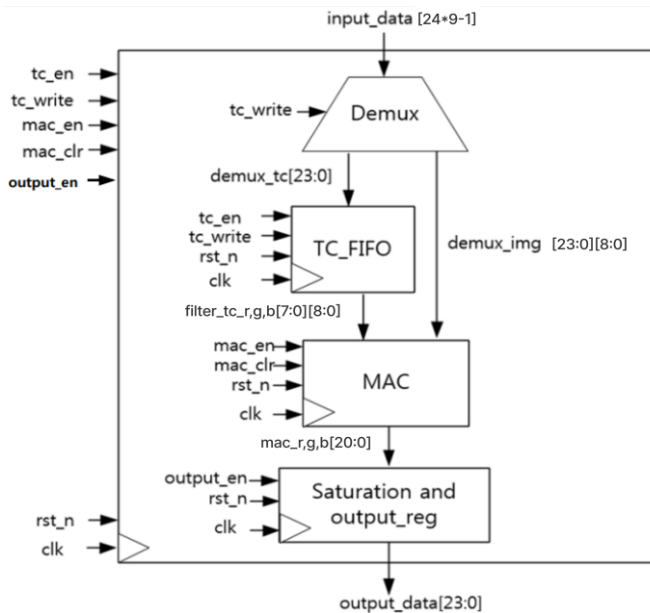
Now	114048360 ns
Cursor 1	4225779 ns

## ii. MAC 연산 개선

### a. 3x3 이미지 필터링에서의 MAC 연산 개선

기존의 MAC 연산은 아래의 기존 3x3 이미지 필터링 파형에서 살펴볼 수 있다 시피, 9개의 이미지 픽셀 정보를 1클럭 당 불러오면서, 9번의 곱셈과 8번의 덧셈을 통해 MAC연산이 총 190ns가 소모됨을 확인할 수 있다.

그러나 이러한 MAC연산은 input\_data의 throughput을 늘려 한 번에 9개 픽셀의 이미지를 입력시켜, input\_data를 1클럭 당 한 픽셀씩 불러오지 않고, 9개의 픽셀에 대한 MAC연산 진행을 9개의 픽셀 전체에 대해 한 번에 진행하여 그 연산 속도를 단축시키고자 하였다.



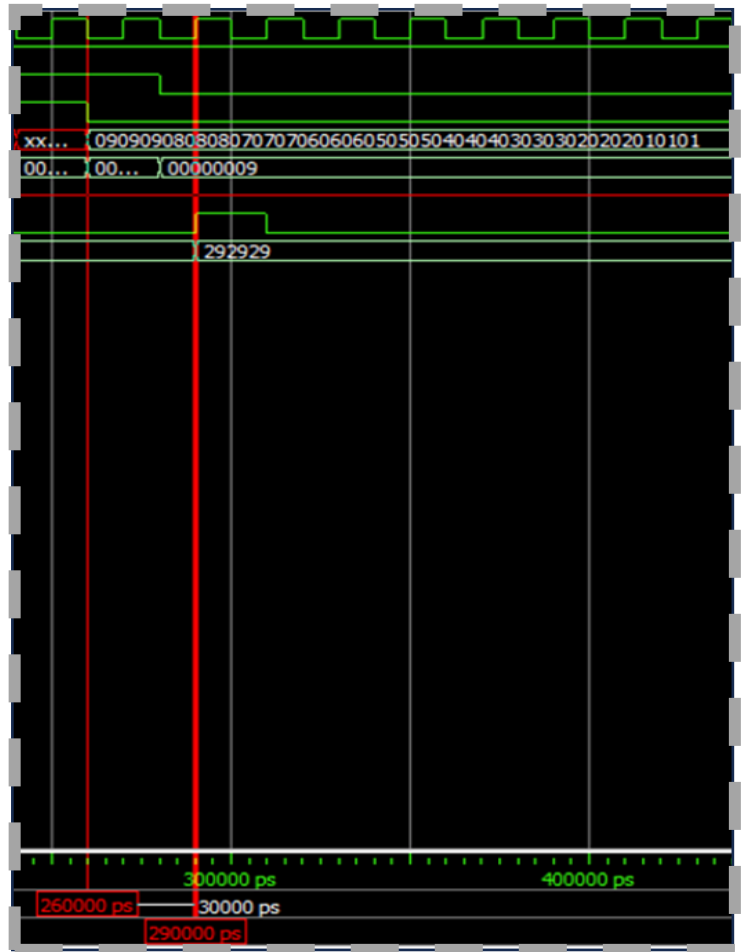
그 결과 아래의 '3x3 이미지 필터링 MAC 수정 시 파형'을 보면, 30ns로 현저히 속도가 빨라진 것을 알 수 있다. 물론 3x3 이미지 필터링 수행 시의 수행 결과이므로 그 차이는 당장에 중요하지 않고 속도가 개선된다는 점만 염두에 두고 다음으로 320x320이미지에 대한 MAC연산 변경을 적용하여 보았다.

물론 연산 속도 개선은 testbench에서 MAC 연산을 위해 9개의 픽셀을 9번의 cycle에 나누어 입력시키던 기존의 이미지 필터링 과정과 달리 한 번에 9개의 픽셀에 대한 MAC 연산을 시키므로 testbench에서 9개의 픽셀을 필터링하는데 걸리는 시간에서 clk 주기\*8만큼 줄어든다는 것을 예상할 수 있었다.

<기존 3x3 이미지 필터링 파형>



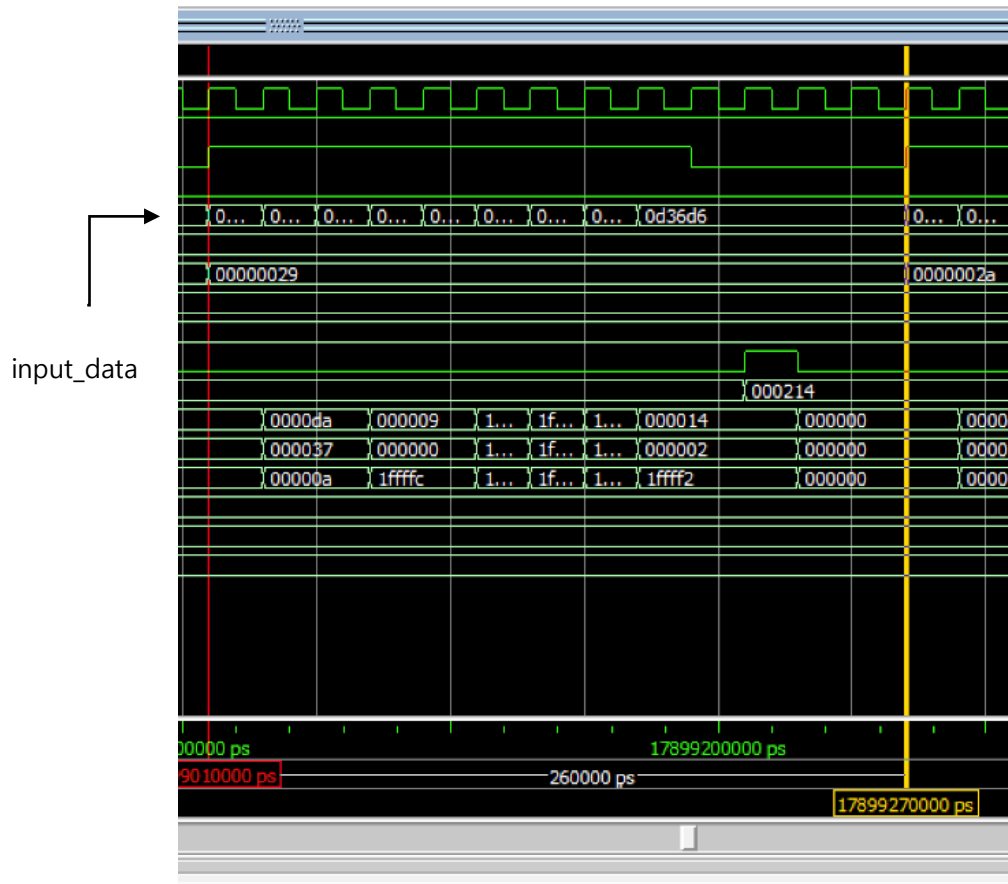
<3x3 이미지 필터링 MAC 수정 시 파형>



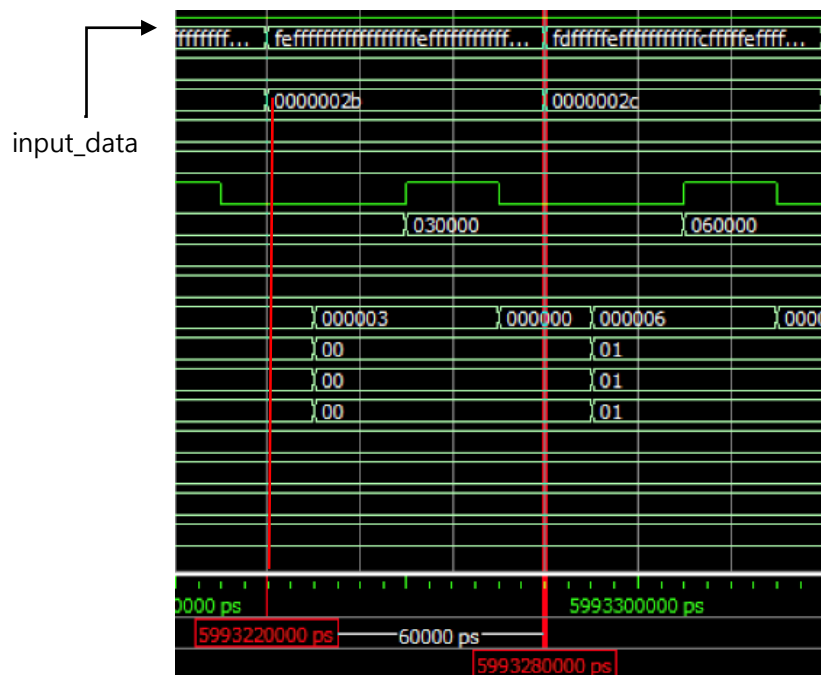
## b. 320x320 이미지 필터링에서의 MAC 연산 개선

3x3 이미지 필터링에 대한 MAC 연산 개선 이후에 320x320 이미지 필터링에 대한 MAC 연산 개선도 실행해보았다. 아래의 파형을 보면 기존의 260ns, 개선된 파형은 60ns로 약 4배 이상 속도 개선이 되었음을 확인할 수 있다.

320x320 이미지 필터링에서도 그 차이는 당장에 중요하지 않고 속도가 개선된다는 점만 염두에 두고 다음으로 1920x1080이미지에 대한 MAC연산 변경을 적용하여 보았다.



<320x320 이미지 필터링 파형>



<320x320 이미지 필터링 MAC 수정 시 파형>

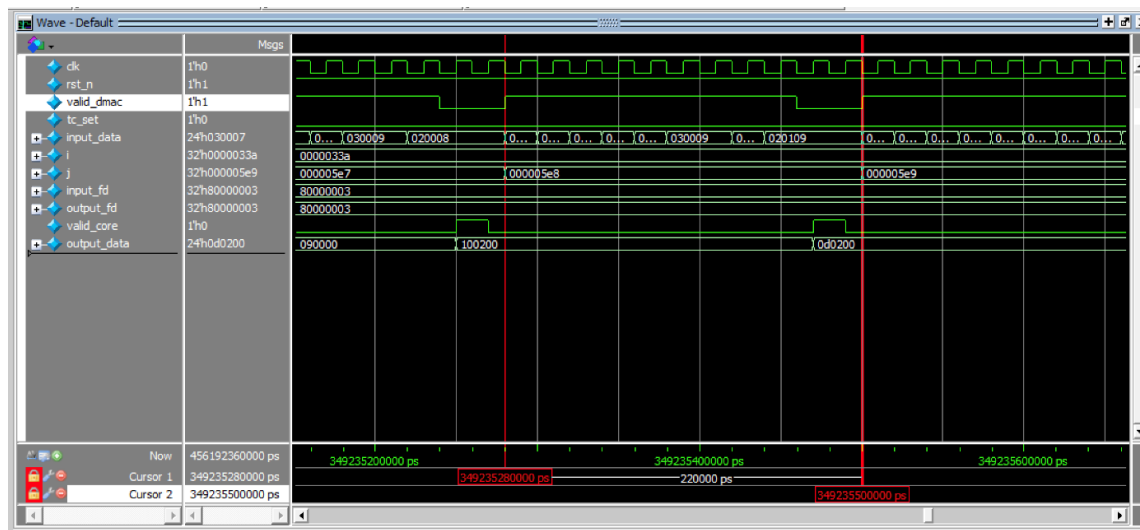
### c. 1920x1080 이미지 필터링에서의 MAC 연산 개선

동일하게 수정된 MAC연산 기능을 1920x1080 이미지 필터링을 할 때 사용해 보았다. 기존 1920x1080 이미지 필터링 파형을 보면 기존 이미지를 처리하기 위해 한 픽셀의 output을 얻으려면 220ns가 필요함을 알 수 있었다.

또한 수정된 파형을 확인해보면, input 이미지 중 9픽셀을 처리하여 한 픽셀의 output을 얻기 위해 60ns의 시간이 필요함을 알 수 있었다. 구체적으로, 기존의 testbench에서는 time delay를 총 160ns 더 사용하였고,

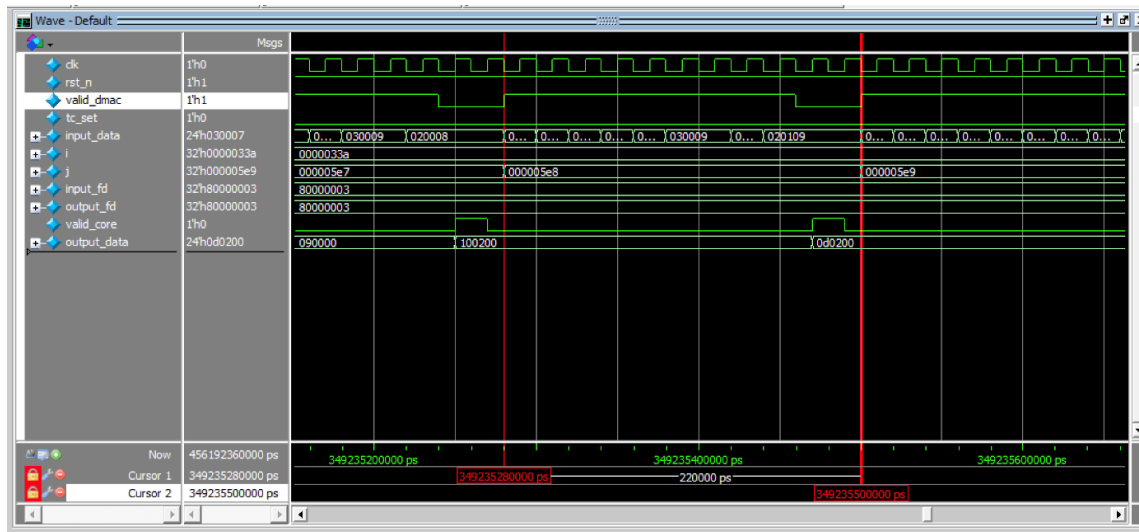
픽셀 9개를 한 번에 9개의 픽셀\*3개의 rgb(24bit) 크기로 확장된 reg로 입력 받아 fir\_filter\_2d 모듈에서 한번에 MAC연산을 하여 그 출력 값을 한번에 출력하므로 한개의 output 픽셀을 얻기 위해 걸리는 시간이 총 160ns가 감소되어  $220\text{ns}/60\text{ns} = 3.66\text{배}$  더 향상된 속도를 확인할 수 있었다.

최종 이미지를 얻기 위해 걸린 시간에 대해서도 기존은 456192360ns, 수정된 것은 124416360ns로  $456192360/124416360=3.66\text{배}$ 로 총 시간도 3.66배 감소됨을 확인할 수 있었다.



<기존 1920x1080 이미지 필터링 파형>





## <1920x1080 이미지 필터링 MAC 수정 시 파형>

### d. MAC 연산 구현하면서 겪은 오류

#### 오류1

MAC 연산을 구현하면서 기존의 input\_data의 throughput을 24비트에서 9픽셀에 대한 데이터를 입력시키기 위한 24\*9비트를 만들어야 하였다. 그렇기에 array 구조로 만들어 unit under test 모듈에 전달하는 것을 시도해보았다. 당연히 정상적으로 모듈의 input데이터로 전달되고, array 구조이므로 접근 및 사용이 원활할 것으로 생각되었다.

그런데 아래의 Illegal reference to memory 오류를 접하고, 모듈의 input 데이터로 배열을 전달하는 것은 불가능한 것이라는 것을 확인할 수 있었다. 때문에 전체적인 구조를 input\_data[24\*9-1:0] 길이의 input\_data를 사용하는 것으로 변경하여 오류를 해결하였다.

```

...fir_filter_2d_tb.v -- Unsuccessful Compile

vlog -work work -vopt -stats=none C:/Users/sean/Digital_system_hws/MAC_change_testbench_1_3x3/fir_filter_2d_tb.v
Questa Intel Starter FPGA Edition-64 vlog 2021.2 Compiler 2021.04 Apr 14 2021
-- Compiling module fir_filter_2d_tb
** Error: C:/Users/sean/Digital_system_hws/MAC_change_testbench_1_3x3/fir_filter_2d_tb.v(35): (vlog-2110) Illegal reference to memory "input_data".
** Error: C:/Users/sean/Digital_system_hws/MAC_change_testbench_1_3x3/fir_filter_2d_tb.v(35): (vlog-2110) Illegal reference to net array "#implicit-wire#2".
** Error: C:/Users/sean/Digital_system_hws/MAC_change_testbench_1_3x3/fir_filter_2d_tb.v(35): (vlog-2110) Illegal reference to memory "input_data".
  
```

## 오류2

아래의 Range width must be constant 오류 화면을 보면, 범위를 지정하는 part-select에 constant 값이 아니면 오류가 나는 것을 확인할 수 있었다. 처음 구현할 경우에는 일반적인 c언어 프로그래밍을 하듯이 변수를 넣어 범위를 지정할 수 있을 것이라 예상했으나,

Verilog에서는 그 범위를 상수 값으로 지정해 놓아야 한다는 것을 배울 수 있었고, 이를 9개의 input\_data를 상수 값으로 일일이 직접 범위 지정하여 오류를 해결할 수 있었다.

```
...fir_filter_2d_tb.v -- Unsuccessful Compile
vlog -work work -vopt -stats=none {C:\Users\sean\Digital_system_hws\MAC_change_testbench_1_3x3\fir_filter_2d_tb.v}
Questa Intel Starter FPGA Edition-64 vlog 2021.2 Compiler 2021.04 Apr 14 2021
-- Compiling module fir_filter_2d_tb
** Error: C:\Users\sean\Digital_system_hws\MAC_change_testbench_1_3x3\fir_filter_2d_tb.v(61): Range width must be constant expression.
```

### iii. 두 방법을 합쳐 최대한 성능을 향상

MAC연산 처리 방법 변경 + PE개수 증가에 따른 성능 향상 비율은 아래와 같다.

a) MAC연산 변경 \* PE 1개 사용 - 124,416,360ns (3.66배 감소)

00:03:12.59

Now	124416360 ns
Cursor 1	1152660 ns

b) MAC연산 변경 \* PE 2개 사용 - 62,208,360ns (약 7.33배(3.66\*2) 감소)

00:02:40.33

Now	62208360 ns
Cursor 1	1152660 ns

c) MAC연산 변경 \* PE 3개 사용 - 41,472,360ns (약 11배(3.66\*3) 감소)

00:02:29.98

Now	41472360 ns
Cursor 1	1152660 ns

d) MAC연산 변경 \* PE 4개 사용 - 31,104,360ns (약 14.64배(3.66\*4) 감소)

00:02:18.69

Now	31104360 ns
Cursor 1	1152660 ns

걸리는 시간을 통해 알 수 있듯이, MAC 연산 개선으로 인한 3.66배와 PE 개수에 따른 속도 증가가 중복되어 곱 연산으로 빨라지는 것을 확인할 수 있다.

#### iv. 결과 분석

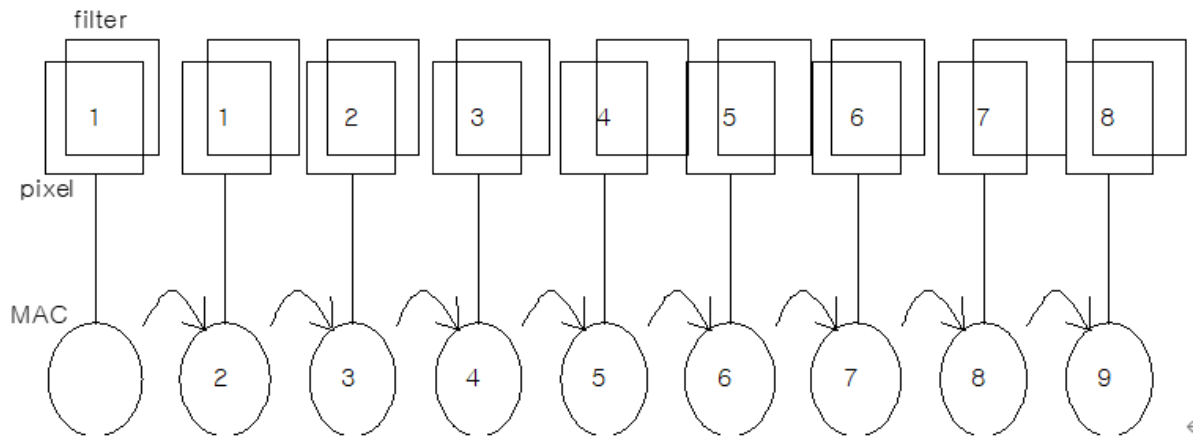
- PE 개수를 4개로 설정한 이유

PE개수를 늘리면 늘릴수록 시뮬레이션 속도는 그만큼 배수적으로 빨라지지만 실제 시간(현실세계에서의 시간)을 측정해 봤을 때는 시간 감소 폭이 점점 줄어드는 것을 확인할 수 있다. 이러한 현상이 발생하는 이유는 이미지를 불러와서 zero padding을 하는 과정, 결과 이미지의 왼쪽을 완성시키는 과정, 결과 이미지를 생성하는 과정을 한 cycle로 설정하여 시뮬레이션에서는 delay없이 표현하기 위해 사실상 MAC연산과정만 측정한 것이기 때문에 이런 결과가 나옴을 알 수 있다. 또한, simulation에서 걸리는 시간만을 생각해보았을 때는 당연히 PE의 개수를 계속해서 늘려 나가 27개까지도 늘려 최선의 성능을 구현할 수 있지만, 위에서 확인해보았듯이 PE가 늘어남에 따른 실제 시간은 그만큼 배수적으로 늘지 않았고, 시간 감소 폭이 점차 줄어드는 것 + PE를 늘림에 따라 현실 회로에서 생기는 reg/wire의 개수의 증가를 고려해보았을 때 4개 이상의 PE에 대해서는 실제 시간의 감소보다 증가하는 reg/wire에 대한 비효율성이 더 크다고 생각해 PE를 4개로 설정하였다.

- MAC 연산에서 실제 시간이 예상한 결과처럼 줄어들지 않은 이유

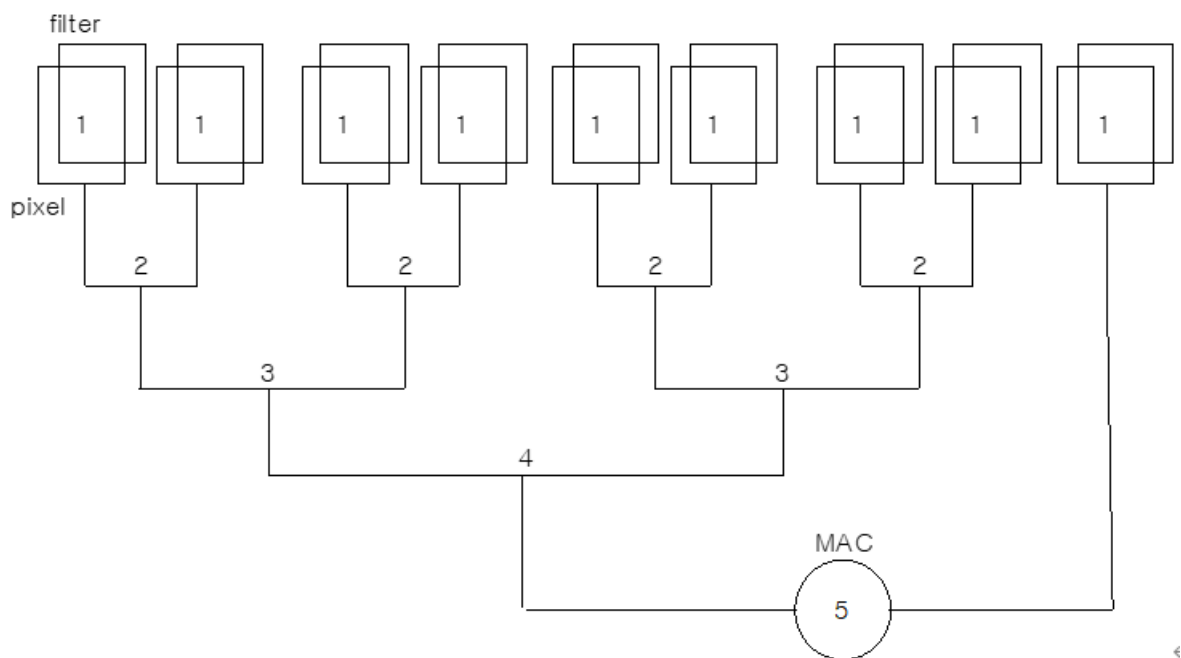
MAC연산의 연산 시간은 9cycle로 돌렸을 때보다 1cycle로 돌렸을 때 예상으로는 9배, 쉬는 2번의 cycle을 포함한다 해도 3.66배정도 차이가 날 것이라 생각했지만 타이머로 시간을 측정해 보았을 때는 약 1.3배정도 줄어드는 것을 확인할 수 있다.

그 이유는 곱 연산 또는 합 연산을 한 번 하는 것을 한 타임이라 생각하면 수정 전 MAC에서는 아래 그림과 같이 곱하고 더하는 연산을 한 픽셀 씩 총 아홉 타임 실행(9번의 곱 연산, 8번의 합 연산)하지만,



<수정 전 MAC 연산>

MAC연산을 수정한 후에는 실제 하드웨어 상에서 전체 픽셀을 필터와 곱하는데 한 타임, 그 후로 연산자를 통해 두 픽셀씩 연산하는데 네 타임을 실행하여 총 다섯 타임(9번의 곱 연산, 8번의 합 연산)이 걸리는 것을 알 수 있다. 시뮬레이션 상으로는 한 번의 cycle에 연산을 하게 설계하여 MAC연산만 생각했을 때 9배의 차이가 생긴다고 예상했지만 실제로는  $9/5 = 1.8$ 배의 차이가 생긴다는 것을 알 수 있다. 여기에 PE를 늘렸을 때처럼 여러 과정을 한 사이클에 담아 두었기 때문에 대략 1.2~1.4배 정도 차이나는 것을 알 수 있다.



<수정 후 MAC 연산>

따라서 최종적으로 PE 4개의 병렬적 처리 + MAC 연산 성능 개선을 통해 31,104,360ns, 약 14.64배의 성능향상을 이루어 냈다.

최종적인 시스템은 아래 그림과 같다.

