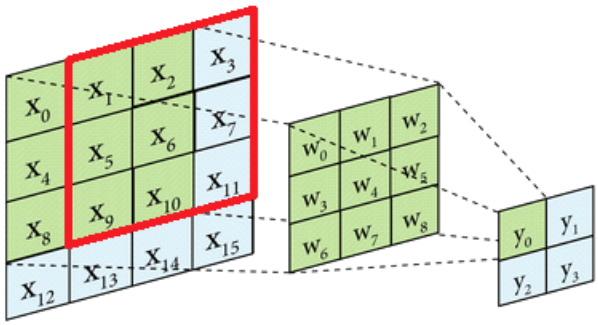
SID Project 8조

Ⅰ About convolution



< Figure 1 >

이번 프로젝트에서는 간단한 2D Convolution을 수행하는 모듈을 제작한다. 이미지를 인식할 때 자주 사용되는 2D Convolution의 경우 위의 그림과 같이 2D array의 인접한 데이터들을 각각의 weight와 곱한 후 더하여 하나의 output cell을 형성한다. 이렇게 구성된 data들은 기존의 Perceptron 방식이나 Fully connected layer와 다르게 이미지의 spatial한 정보를 담고있기 때문에 이미지 인식에서 유용하게 사용된다.

Fig.1에서 X의 값들은 입력 값으로 주어진 2D 이미지라고 생각하고 W의 값들을 필터라고 생각하면 y_0의 값은 다음과 같다.

$$y_0 = x_0w_0 + x_1w_1 + x_2w_2 + x_4w_3 + x_5w_4 + x_6w_5 + x_8w_6 + x_9w_7 + x_{10}w_8$$

동일한 방법으로 y_1의 값은 x_6을 중심으로 하는 figure 1에서 붉은색으로 표시된 부분과 필터를 곱하여 더한 것이라 생각하면 된다. 그 다음은 x_7이 중심이 아닌 x_9가 중심인 3×3 사각형과 곱해져서 y_2를 형성한다. 따라서 row가 변하는 단계에서 데이터를 skip하는 현상이 발생한다.

추가적으로 data skip을 방지하기 위해 boundary에 padding을 추가하거나 원하는 spatial 정보에 따라서 필터의 움직임을 한 칸씩이 아닌 두 칸, 세 칸으로 변화를 줄 수 있지만, 이번 프로젝트에서는 Fig.1과 원리가 동일한 가장 간단한 형태의 Convolution을 수행하는 모듈을 제작한다.

입력 데이터로는 28×28 16bit signed가 주어지며, 필터의 크기는 3×3 16bit signed이다. 따라서 출력으로는 26×26 32bit signed를 1D array 형식으로 전송하게 된다.

Ⅱ Design method

우선 가장 중요한 부분은 **매 클록마다 차례로 입력 데이터가 모듈로 들어온다**는 것이다. 따라서 사전에 세팅된 필터 값과 바이어스 값을 이용하여 **즉시** 출력 데이터를 전송하는 것이 불가능하며 **일정시간동안 입력 데이터를 받아야** 계산할 데이터가 생기게 된다.

가장 쉽게 생각할 수 있는 방법은 모든 입력 데이터를 1D array로 저장하고 그 이후 해당 데이터를 이용해 순차적으로 가공된 데이터를 출력하는 것이다. 프로젝트에서 요구하는 스펙은 28×28 크기의 16비트 입력데이터를 사용하므로 총 $28\times28\times16=12544$에 달하는 매우 큰 저장 공간을 요구하고, 따라서 Area에서 많은 손해를 입게 된다. 또한 연산시간도 길

어지게 되는데, 입력 데이터를 받는데 $28\times28=784clk$을, 또한 출력데이터를 전송하는데 $26\times26=676clk$의 시간, 총 $1460clk$을 소모하게 된다.

따라서 생각할 수 있는 방법은, 현재 만들어야 하는 출력 데이터를 위한 입력 데이터가 완성이 됐다면 즉시 출력을 하는 것이다. 예를 들어 figure 1의 경우에 y_0를 만들기 위해서는 x_{15}까지의 데이터가 전부 필요하지 않으며 x_{10}까지의 데이터만 있으면 충분하다. 우리는 이러한 점을 이용하여 입력 데이터를 저장하는 공간을 축소하였다.

Time	Front										Back
T=9	I[0]	I[1]	I[2]	I[3]	I[4]	I[5]	I[6]	I[7]	I[8]	I[9]	I[10]
T=10	I[11]	I[1]	I[2]	I[3]	I[4]	I[5]	I[6]	I[7]	I[8]	I[9]	I[10]
T=11	I[11]	I[12]	I[2]	I[3]	I[4]	I[5]	I[6]	I[7]	I[8]	I[9]	I[10]
T=12	I[11]	I[12]	I[13]	I[3]	I[4]	I[5]	I[6]	I[7]	I[8]	I[9]	I[10]
T=13	I[11]	I[12]	I[13]	I[14]	I[4]	I[5]	I[6]	I[7]	I[8]	I[9]	I[10]
T=14	I[11]	I[12]	I[13]	I[14]	I[15]	I[5]	I[6]	I[7]	I[8]	I[9]	I[10]

< Figure 2 >

Fig.2의 예시는 3×3 filter가 4×4 input에 convolution 연산을 진행하는 time flow를 나타낸 것이다. 1D buffer는 총 11개의 데이터를 담을 수 있으며 T=0에 첫 데이터가 buffer에 입력된다고 생각하면 T=9가 되는 순간부터 convolution 연산을 수행할 수 있다. 따라서 T=9~10까지는 결과를 출력할 수 있으나, boundary에서 filter가 적용되는 지점이 row가 변화하며 data를 건너뛰는 현상이 발생하여 2 clk 동안 출력이 불가능하며, 이후 T=13, 14에 결과를 출력함으로써 convolution 연산을 마무리 할 수 있다.

1D buffer는 circular queue 방식으로 데이터를 저장하며 fig.2를 보면 알 수 있듯 filter의 각 row 별로 circular queue를 1칸씩 순환하며 출력을 위한 연산을 수행한다. 따라서 전체 데이터를 저장하지 않고도 연산을 수행할 수 있다는 장점이 있다.

그렇다면 이번 프로젝트에서 필요한 1D buffer의 크기는 filter의 크기가 3×3이므로 $28\times2+3=59$의 크기가 될 것이다. 하지만 우리 조는 크기를 64로 사용하였다. 그 이유는 다음과 같다.

만일 buffer의 크기가 59라면 circular queue를 구성함에 있어서 추가적은 conditional branch를 구성하여 59가 넘어가는 순간에 0으로 초기화를 시켜주어야 한다. 하지만 만일 64로 구성한다면 6-bit으로 표현이 가능하고 unsigned로 사용하면 자동으로 overflow가 발생함에 따라 0으로 돌아오기 때문에 implementation이 용이하다. 약간의 area penalty가 존재하지만 MUX와 wire interconnect area의 이득을 생각하여 line buffer의 size를 64로 설정하였다.

만일 이렇게 구성한다면 $T=1clk$일 때 첫 데이터가 준비된다고 생각하면 $T=59clk$부터 출력이 가능하며 $28-3+1=26$개의 output을 출력하는데 $+26clk$ 그리고 다음 row로 이동하는 delay가 $+2clk$이다. 추가적으로 총 row도 26 row가 존재하므로 이를 계산식에 추가하면, 총 걸리는 시간은 $59+26\times(26+2)=787clk$으로 naive하게 구성했을 때의 $1460clk$보다 throughput이 약 2배 증가했다는 것을 알 수 있다.

하지만 이후 설계과정에서 buffer에 data를 입력하는 부분을 nonblocking assignment로 진행함에 따라 59clk 직후에 output data를 출력하는 것에 문제가 발생하였고 따라서 실제 설계에서는 1clk이 추가되어 총 788clk 동안 수행되는 모듈을 설계하였다.

Ⅲ Convolution module

```
1  module conv1(data_in, data_out, clk, rst_n);
2  input signed [15:0] data_in;
3  input clk, rst_n;
4  output reg signed [31:0] data_out;
5
6  reg signed [15:0]buffer[63:0]; // line buffer
7  reg [5:0] ptr;                 // used for circular queue
8  reg [5:0] offset[8:0];         // used for locating input data
9  reg state;                     // checks whether data is ready
10 always @(posedge clk) begin
11     if(rst_n==0) begin // initialzation
12         ptr<=0;
13         offset[0]<=0;
14         offset[1]<=1;
15         offset[2]<=2;
16         offset[3]<=28;
17         offset[4]<=29;
18         offset[5]<=30;
19         offset[6]<=56;
20         offset[7]<=57;
21         offset[8]<=58;
22         state<=1'b0;
23     end
24     else begin
25         buffer[ptr]<=data_in;
26         if(state==1'b0) begin // data waiting state
27             if(ptr==6'd58) begin
28                 state<=1'b1;
29                 ptr<=ptr+1;
30             end
31             else begin
32                 state<=1'b0;
33                 ptr<=ptr+1;
34             end
35         end
36         else begin // output generating state
37             ptr<=ptr+1;
38             data_out<=(-4972*buffer[offset[0]]
39                        -622*buffer[offset[1]]
40                        +2988*buffer[offset[2]])
41                       +(-2478*buffer[offset[3]]
42                        +1703*buffer[offset[4]]
43                        +2519*buffer[offset[5]])
44                       +(2008*buffer[offset[6]]
45                        +1748*buffer[offset[7]]
46                        +79*buffer[offset[8]])-58730196;
47             offset[0]<=offset[0]+1;
48             offset[1]<=offset[1]+1;
49             offset[2]<=offset[2]+1;
50             offset[3]<=offset[3]+1;
51             offset[4]<=offset[4]+1;
52             offset[5]<=offset[5]+1;
53             offset[6]<=offset[6]+1;
54             offset[7]<=offset[7]+1;
55             offset[8]<=offset[8]+1;
56         end
57     end
58 end
59 endmodule
```

< Figure 3 >

Fig.3가 위에서 설명한 design 방식에 따라 제작한 모듈이다. 16-bit data를 data_in으로 전달받으며 clk와 rst_n 또한 탑 모듈로부터 입력받는다. data_out wire를 32-bit 크기로 제작한다. 모듈 내의 register로는 buffer, ptr, offset, state가 존재한다.

buffer	1D buffer, line buffer이며 총 64개의 16-bit 데이터를 저장할 수 있다.
ptr	circular queue의 front를 추적하는 용도로 사용한다.
offset	input data가 circular queue의 어느 index에 위치하는지 추적하기 위한 register.
state	데이터 준비 단계와 데이터 출력 단계를 구분하기 위한 state register.

line 11 : rst_n이 0이면 데이터가 초기화되어야 하므로 모든 ptr, offset, state register를 초기화한다. offset의 경우 28×28 크기에 맞추어 첫 output neuron을 생성하기 위한 위치로 설정된다.

line 26 : 만일 아직 data가 59개가 아니라면 state를 0으로 설정하여 output을 출력하지 않고 data를 circular queue에 저장한다.

line 27 : 데이터가 모두 준비되었다면 state를 변경하여 output 출력 단계로 넘어간다.

line 36 : 데이터 출력이 가능하므로 offset을 이용해 buffer에서 input data를 추출하고 미리 hard coding된 weight값과 곱하고 bias를 제하여 output data를 형성한다.

line 47 : 이후 모든 offset의 값을 1씩 더함으로써 다음 output data를 위한 input data의 위치를 수정한다.

Ⅳ Testbench

```
1   `timescale 1ns/1ns
2   module do_conv;
3
4   reg clk;
5   reg rst_n=1'b1;
6   reg signed [15:0] data_in;
7   wire signed [31:0] data_out1;
8   wire signed [31:0] data_out2;
9   wire signed [31:0] data_out3;
10
11  integer file_input, file_output, work_sequence,i;
12  reg [120:0]input_txt;
13  reg [120:0]output_txt;
14  conv1 example1(data_in, data_out1, clk, rst_n);
15  conv2 example2(data_in, data_out2, clk, rst_n);
16  conv3 example3(data_in, data_out3, clk, rst_n);
17
18  initial
19  begin
20      clk = 1'b0;
21      forever #50 clk = ~clk;
22  end
23
24  initial
25  begin
26      work_sequence = $fopen("work_sequence.txt","r");
27      for(i=0;i<20;i=i+1) begin
28         $fscanf(work_sequence,"%s",input_txt);
29         $fscanf(work_sequence,"%s",output_txt);
30         file_output = $fopen(output_txt, "w");
31         file_input = $fopen(input_txt, "r");
32         rst_n=1'b0;
33         #100 rst_n=1'b1;
34         repeat (60) begin
35               $fscanf(file_input, "%6d", data_in); #100;
36         end
37         repeat (26) begin
38               repeat (26) begin
39                     $fscanf(file_input, "%6d", data_in);
40                     $fdisplay(file_output, "%12d", data_out1);#100;
41               end
42               $fscanf(file_input, "%6d", data_in);#100;
43               $fscanf(file_input, "%6d", data_in);#100;
44         end
45         $fclose(file_input);
46         file_input = $fopen(input_txt, "r");
47         rst_n=1'b0;
48         #100 rst_n=1'b1;
49         repeat (60) begin
50               $fscanf(file_input, "%6d", data_in); #100;
51         end
52         repeat (26) begin
53               repeat (26) begin
54                     $fscanf(file_input, "%6d", data_in);
55                     $fdisplay(file_output, "%12d", data_out2);#100;
56               end
57               $fscanf(file_input, "%6d", data_in);#100;
58               $fscanf(file_input, "%6d", data_in);#100;
59         end
60         $fclose(file_input);
61         file_input = $fopen(input_txt, "r");
62         rst_n=1'b0;
63         #100 rst_n=1'b1;
64         repeat (60) begin
65               $fscanf(file_input, "%6d", data_in); #100;
66         end
67         repeat (26) begin
68               repeat (26) begin
69                     $fscanf(file_input, "%6d", data_in);
70                     $fdisplay(file_output, "%12d", data_out3);#100;
71               end
72               $fscanf(file_input, "%6d", data_in);#100;
73               $fscanf(file_input, "%6d", data_in);#100;
74         end
75         $fclose(file_input);
76         $fclose(file_output);
77      end
78      $finish;
79  end
80
81  endmodule
```

< Figure 4 >

Fig. 4는 위에서 작성한 convolution 연산 모듈 3개를 엮어서 simulation을 진행하는 testbench이다.

clk	주기가 100ns signal을 저장하는 register
rst_n	rst_n이 0이면 모든 module reset
data_in	file_input으로 입력되는 data를 전달하기 위한 register
data_out1	conv1.v 로부터 전달되는 데이터를 연결하기 위한 wire
data_out2	conv2.v 로부터 전달되는 데이터를 연결하기 위한 wire
data_out3	conv3.v 로부터 전달되는 데이터를 연결하기 위한 wire
input_txt	입력 데이터 저장 위치
output_txt	출력 데이터 저장 위치
work_sequence.txt	input data가 저장된 txt파일의 이름과 output data를 저장할 txt파일의 이름을 저장해놓은 파일

line 14~16 : 모듈 3개를 연결하는 부분

line 18~22 : Clock generation with T=100ns

line 28~29 : work_sequence.txt로부터 입력 txt 파일과 출력 txt파일의 이름을 저장.

line 30~31 : work_sequence.txt로 입력받은 이름으로 파일 열람

line 32~33 : 모듈 reset

line 34~36 : data preparation을 위해 60회 반복

line 37~44 : output generation with input data feed

line 48~77 : 나머지 두 모듈에 대해서도 반복

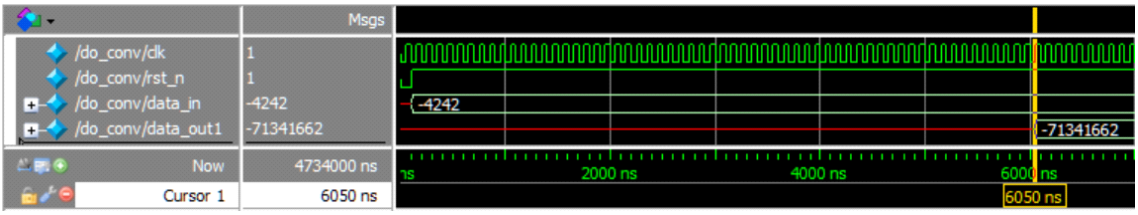
line 26~78 : 3개의 모듈을 각각의 input에 대해 반복

data를 입력하는 부분에 대해 부연설명을 하자면, line 34~36에서는 data를 준비하는 부분이며, 60개의 데이터가 모두 입력이 완료되면 output data가 출력되므로 이를 받기 위해서 line 37~44 부분으로 이동한다. 이때 26 clock cycle동안 데이터 입력과 출력 데이터 저장을 병렬로 진행하며, 이후 2 clock cycle 동안 row를 변경하기 위해 output data 출력을 멈추고 이후 다시 진행한다.

이렇게 3개의 모듈을 모두 독립적으로 실행했다. output data를 하나의 text file에 저장하기 위해 이렇게 진행한 것뿐이며, 3개 모듈을 병렬적으로 처리하는 방식도 충분히 가능하다. 하나 모듈의 진행 상황을 체크하면 $78800ns$가 걸린다는 것을 알 수 있고 주기가 $100ns$이므로 $788clk$으로 design 한 모듈이 정확히 동작한다는 것을 알 수 있다.

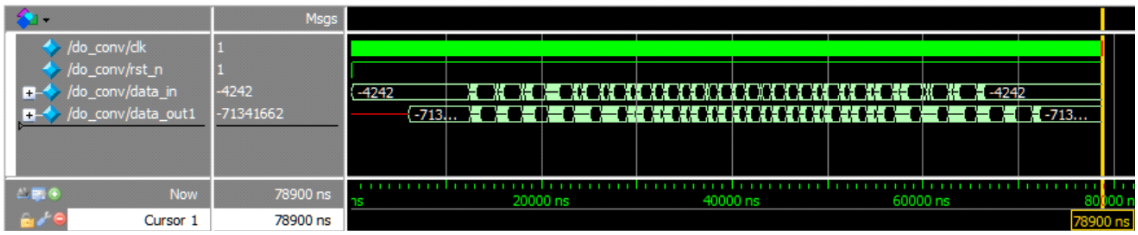
V Waveform verification

우선 testbench 결과의 waveform을 확인하면 다음과 같다. timescale은 1ns이며 클럭의 주기는 100ns이다.



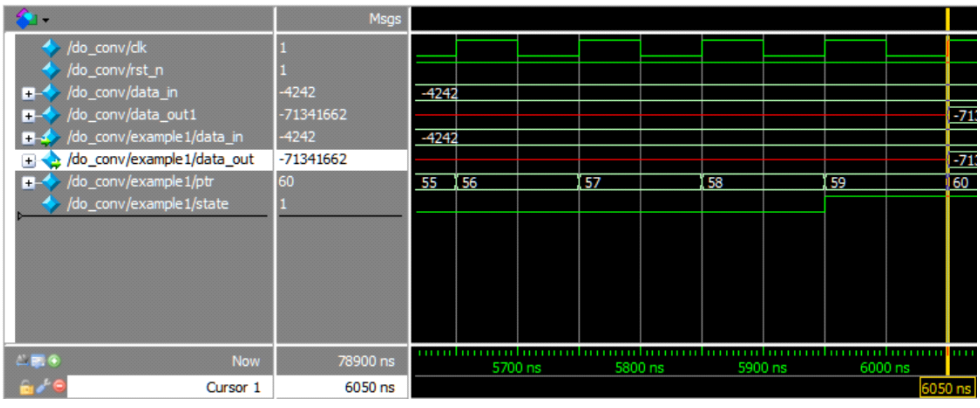
< Waveform 1>

6050ns부터 output data가 형성되는 것을 확인할 수 있으며



< Waveform 2 >

정확히 78900ns에 종료가 되는 것을 확인할 수 있다. 여기서 100ns가 추가된 이유는 reset signal에 의한 데이터 초기화 과정이 추가되기 때문이다. 따라서 Waveform을 분석한 결과 우리가 design한 대로 788 clock cycle동안 작동한다는 것을 확인할 수 있다.

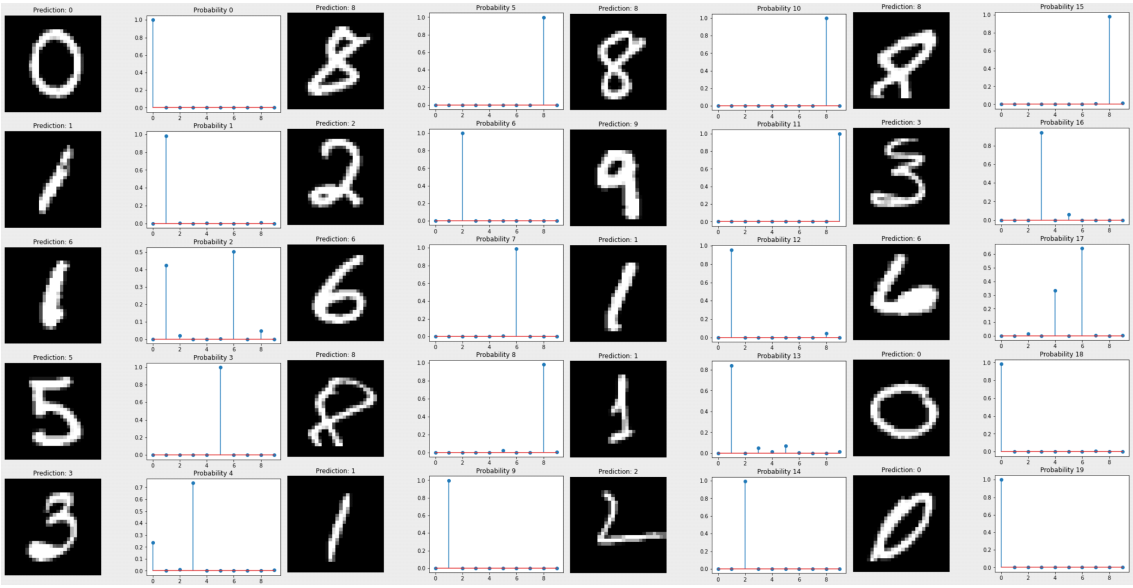


< Waveform 3 >

개별 모듈인 conv1.v의 waveform을 확인하면 ptr, 즉 circular queue의 front 위치가 59에 도달했을 때 state transition이 발생하며 그 다음부터 output data generation이 발생하는 것을 확인할 수 있다. 따라서 데이터 입력이 시작된 후 5900ns가 지난 후부터 output이 발생한다.

이렇게 개별 모듈이 reset을 제외하면 788clk동안 동작하는 것을 waverform simulation을 통해 확인하였다.

VI Image recognition result



< Figure 5 >

구글 코랩으로 output을 분석한 결과 Fig. 5를 확인하면 전체 20개의 인식 데이터 중 1개의 데이터에서 오답이 발생하여 95% 정도의 정확도를 보여주는 것을 확인하였다.

```
set DESIGN_NAME "conv1"

set REPORTS_DIR "${DESIGN_REF_DATA_PATH}/reports"
set RESULTS_DIR "${DESIGN_REF_DATA_PATH}/outputs"

analyze -format verilog ${DESIGN_REF_DATA_PATH}/rtl/conv1.v
elaborate ${DESIGN_NAME}
create_clock {"clk"} -name "clk" -period 10 -waveform {0 5}
set_dont_touch_network {clk}
check_design -summary
compile
write -f ddc -h -o ${RESULTS_DIR}/${DESIGN_NAME}_mapped.ddc
write -f verilog -h -o ${RESULTS_DIR}/${DESIGN_NAME}_mapped.v
report_timing > ${REPORTS_DIR}/${DESIGN_NAME}.mapped.timing.rpt
report_area > ${REPORTS_DIR}/${DESIGN_NAME}.mapped.area.rpt
exit
```

< tcl file >

위와 같은 tcl file을 작성하여 timing과 area report를 분석하였다.

Ⅶ Area report

Register Name	Type	Width	Bus	MB	AR	AS	SR	SS	ST
state_reg	Flip-flop	1	N	N	N	N	N	N	N
ptr_reg	Flip-flop	6	Y	N	N	N	N	N	N
data_out_reg	Flip-flop	32	Y	N	N	N	N	N	N
buffer_reg	Flip-flop	1024	Y	N	N	N	N	N	N
offset_reg	Flip-flop	54	Y	N	N	N	N	N	N

< Figure 6 >

```
****************************************
Report : area
Design : conv1
Version: Q-2019.12-SP5-5
Date   : Mon Dec 20 23:27:21 2021
****************************************

Library(s) Used:

    saed32rvt_ff1p16v125c (File: /data3/Class/SID/2021_2/2021_2_sid

Number of ports:                         2138
Number of nets:                         15101
Number of cells:                        12467
Number of combinational cells:          11262
Number of sequential cells:              1117
Number of macros/black boxes:               0
Number of buf/inv:                       1252
Number of references:                      35

Combinational area:              32033.580281
Buf/Inv area:                     2294.666273
Noncombinational area:            7384.408306
Macro/Black Box area:                0.000000
Net Interconnect area:           15714.137732

Total cell area:                 39417.988587
Total area:                      55132.126319
```

< Figure 7 >

Fig.6를 통해 각 모듈에서 선언한 state register, ptr register, data_out register, buffer register, offset register를 확인할 수 있다. 그중 가장 큰 Area를 차지하는 부분은 16bit 정보를 64개를 담는 buffer이며 따라서 width가 1024로 설정된 것을 확인할 수 있다. 여기서 state를 복잡하게 가져가면 buffer의 크기를 58로 6만큼 작게 가져가는 것이 가능하다. 추가적으로 Fig. 7을 확인하면 Total Area는 combinational, Noncombinational, Net Interconnect area의 합으로 총 55132.126319인 것을 확인할 수 있다.

만일 모든 데이터를 저장한 후 출력하는 모듈을 설계한다면 buffer의 width 크기가 $28time28 \times 16bit = 12544$로 매우 크게 설정되므로 area penalty와 길어진 data preparation에 따른 throughput penalty가 존재한다.

Ⅷ Timing report

```
Point                                                   Incr       Path
-----------------------------------------------------------------------------
clock clk (rise edge)                                   0.00       0.00
clock network delay (ideal)                             0.00       0.00
offset_reg[4][0]/CLK (DFFX1_RVT)                        0.00 #     0.00 r
offset_reg[4][0]/Q (DFFX1_RVT)                          0.06       0.06 r
U6074/Y (AND2X1_RVT)                                    0.02       0.08 r
U6075/Y (AND2X1_RVT)                                    0.02       0.10 r
U1535/Y (NBUFFX2_RVT)                                   0.03       0.13 r
U6606/Y (AO22X1_RVT)                                    0.03       0.16 r
U6607/Y (AO221X1_RVT)                                   0.03       0.19 r
U6614/Y (NOR4X0_RVT)                                    0.05       0.24 f
U6624/Y (OA22X1_RVT)                                    0.02       0.27 f
U6625/Y (OAI221X1_RVT)                                  0.06       0.33 r
mult_38_5/B[13] (conv1_DW02_mult_4)                     0.00       0.33 r
mult_38_5/U107/Y (XOR2X1_RVT)                           0.05       0.38 f
mult_38_5/S2_2_11/S (FADDX1_RVT)                        0.06       0.44 r
mult_38_5/U35/Y (XOR2X1_RVT)                            0.05       0.48 f
mult_38_5/U114/Y (XOR2X1_RVT)                           0.05       0.53 r
mult_38_5/S2_5_8/S (FADDX1_RVT)                         0.06       0.59 f
mult_38_5/U110/Y (XOR2X1_RVT)                           0.05       0.64 r
mult_38_5/S2_7_6/S (FADDX1_RVT)                         0.06       0.69 f
mult_38_5/U112/Y (XOR2X1_RVT)                           0.05       0.74 r
mult_38_5/S2_9_4/S (FADDX1_RVT)                         0.06       0.80 f
mult_38_5/S2_10_3/S (FADDX1_RVT)                        0.06       0.86 r
mult_38_5/U34/Y (XOR2X1_RVT)                            0.05       0.90 f
mult_38_5/U2/Y (XOR2X1_RVT)                             0.05       0.95 r
mult_38_5/FS_1/A[11] (conv1_DW01_add_11)                0.00       0.95 r
mult_38_5/FS_1/U81/Y (NAND2X0_RVT)                      0.03       0.98 f
mult_38_5/FS_1/U14/Y (AND2X1_RVT)                       0.03       1.01 f
mult_38_5/FS_1/SUM[11] (conv1_DW01_add_11)              0.00       1.01 f
mult_38_5/PRODUCT[13] (conv1_DW02_mult_4)               0.00       1.01 f
add_1_root_add_38_3/B[13] (conv1_DW01_add_5)            0.00       1.01 f
add_1_root_add_38_3/U1_13/S (FADDX1_RVT)                0.05       1.06 r
add_1_root_add_38_3/SUM[13] (conv1_DW01_add_5)          0.00       1.06 r
add_0_root_add_38_3/B[13] (conv1_DW01_add_3)            0.00       1.06 r
add_0_root_add_38_3/U1_13/CO (FADDX1_RVT)               0.04       1.10 r
add_0_root_add_38_3/U1_14/CO (FADDX1_RVT)               0.04       1.14 r
add_0_root_add_38_3/U1_15/CO (FADDX1_RVT)               0.04       1.17 r
add_0_root_add_38_3/U1_16/CO (FADDX1_RVT)               0.04       1.21 r
add_0_root_add_38_3/U1_17/CO (FADDX1_RVT)               0.04       1.25 r
add_0_root_add_38_3/U1_18/CO (FADDX1_RVT)               0.04       1.29 r
add_0_root_add_38_3/U1_19/CO (FADDX1_RVT)               0.04       1.33 r
add_0_root_add_38_3/U1_20/CO (FADDX1_RVT)               0.04       1.36 r
add_0_root_add_38_3/U1_21/CO (FADDX1_RVT)               0.04       1.40 r
add_0_root_add_38_3/U1_22/CO (FADDX1_RVT)               0.04       1.44 r
add_0_root_add_38_3/U1_23/CO (FADDX1_RVT)               0.04       1.48 r
add_0_root_add_38_3/U1_24/CO (FADDX1_RVT)               0.04       1.52 r
add_0_root_add_38_3/U1_25/CO (FADDX1_RVT)               0.04       1.56 r
add_0_root_add_38_3/U1_26/CO (FADDX1_RVT)               0.04       1.59 r
add_0_root_add_38_3/U1_27/CO (FADDX1_RVT)               0.04       1.63 r
add_0_root_add_38_3/U1_28/CO (FADDX1_RVT)               0.04       1.67 r
add_0_root_add_38_3/U1_29/CO (FADDX1_RVT)               0.04       1.71 r
add_0_root_add_38_3/U1_30/S (FADDX1_RVT)                0.07       1.78 f
add_0_root_add_38_3/SUM[30] (conv1_DW01_add_3)          0.00       1.78 f
add_1_root_add_0_root_sub_38_2/A[30] (conv1_DW01_add_1)
                                                        0.00       1.78 f
add_1_root_add_0_root_sub_38_2/U1_30/S (FADDX1_RVT)     0.06       1.83 r
add_1_root_add_0_root_sub_38_2/SUM[30] (conv1_DW01_add_1)
                                                        0.00       1.83 r
add_0_root_add_0_root_sub_38_2/B[30] (conv1_DW01_add_0)
                                                        0.00       1.83 r
add_0_root_add_0_root_sub_38_2/U1_30/CO (FADDX1_RVT)
                                                        0.04       1.87 r
add_0_root_add_0_root_sub_38_2/U1_31/S (FADDX1_RVT)     0.05       1.92 f
add_0_root_add_0_root_sub_38_2/SUM[31] (conv1_DW01_add_0)
                                                        0.00       1.92 f
U9235/Y (AO22X1_RVT)                                    0.02       1.95 f
data_out_reg[31]/D (DFFX1_RVT)                          0.00       1.95 f
data arrival time                                                  1.95

clock clk (rise edge)                                  10.00      10.00
clock network delay (ideal)                             0.00      10.00
data_out_reg[31]/CLK (DFFX1_RVT)                        0.00      10.00 r
library setup time                                     -0.02       9.98
data required time                                                 9.98
-----------------------------------------------------------------------------
data required time                                                 9.98
data arrival time                                                 -1.95
-----------------------------------------------------------------------------
slack (MET)                                                        8.03
```

< Figure 8 >

Fig. 8을 통해 path에서 가장 큰 portion을 차지하는 부분은 multiplier와 adder임을 확인하였다. data required time = 9.98, data arrival time = 1.95임으로 그 차이인 slack이 총 8.03인 것을 확인할 수 있다. 이를 통해 clock 주기가 필요 주기보다 더 긴 것을 확인할 수 있고, clock 주기를 1/5만큼 줄여서 throughput를 증가시킬 수 있다.

IX Conclusion

이번 프로젝트에서 line buffer를 활용하여 788클럭 동안 작동하는 convolution 모듈을 제작하였으며 area는 55132, slack은 8.03을 만족하는 모듈을 제작하였다. 이후 모듈을 통해 제작된 output 파일을 google colab으로 검증한 결과 95%의 정확도를 보인다는 것을 근거로 정상 작동한다는 것을 증명하였다.

기여도

매번 ZOOM을 활용하여 진행상황을 공유하고 피드백을 얻어가며 진행하였기에 정확히 나누기는 어렵지만 처음 프로젝트 시작 시 배분한 프로젝트 내용은 다음과 같습니다.

윤희원 : testbench 설계, waveform 분석, 리포트 작성

박재영 : convolution 모듈 설계, area, timing 분석, 리포트 작성

따라서 두 조원의 기여도는 동등합니다.