Weekly Progress 1주차 - 최명규

알게된 것

1. Pynq z2 보드를 노트북에 연결하여 작동시키는 법
2. Pynq z2 보드의 구조
3. HLS 에서 구현한 cpp 코드를 pynq에 이식하여 동작시키는 법

Pynq z2 보드를 노트북에 연결하는 방법

고정 ip를 받아 접속하는 방법과 노트북과 같은 네트워크를 사용하는 방법이 있다. 보통 집에서 작업 할 것이므로 노트북과 같은 네트워크를 사용해서 연결하였다.

Pynq z2 보드의 구조

Pynq z2 보드는 PS 와 PL로 이루어져 있다. PS는 Processing System의 약자로 우리가 평소에 생각하는 프로세서이다. 파이썬코드는 이 부분에서 실행된다. 만들어져 있는 프로세서 이므로 hard processor라고 부르기도 한다. PL은 Programmable Logic의 약자로 custom hardware를 만들 수 있는 부분이다. 만들 수 있는 프로세서 부분이므로 soft processor라고 부르기도 한다. 그리고 PS와 PL을 잇는 인터페이스가 존재한다. 이름은 AXI이다.

HLS 에서 짠 코드를 pynq에 이식하여 동작시키는 법

HLS에서 cpp코드를 IP로 만들고 추출하여 vivado에서 여러 부품들과 연결한다. 다 연결된 IP core 를 vivado에서 bitstream으로 만들면 bitstream과 hwh파일을 pynq에 저장한다. 파이썬에서 bitstream 파일을 불러오고 데이터를 매핑해주어 사용할 수 있다. 이 때 편리한 사용을 위해 드라이버를 만들어 사용할 수 있다.

실습

1. Pynq tutorial add 함수
2. matmul함수

Add 함수

HLS부터 IP core를 입히는 과정과 파이썬에서 불러오고 드라이버를 만드는 과정까지 진행하였다. 2020S의 add.ipnyb에 정리되어 있다.

Matmul 함수

코드를 그대로 옮겨보려 하였지만 HLS로 함수를 구현하여 옮겨보니 BRAM의 용량보다 훨씬 큰 데이터를 parameter로 받아야 되어 실패하였다. S\_axilite를 이용하여 포트 매핑을 해주어도 용량이 초과되어 실패했다. DMA와 axi stream을 이용해서 구현해보려고 했지만 stream에서 데이터를 읽을때 쓰기도 필요해서 또 실패했다.

DMA 와 axi stream을 이용해서 구현했다. 메모리 용량의 문제는 int대신 4비트 정수를 사용했다.

#include "ap\_int.h"

void matmul(ap\_int<4> input[1024\*1024\*2],ap\_int<8> output[1024\*1024]){

#pragma HLS INTERFACE s\_axilite port=return bundle=CONTROL\_BUS

#pragma HLS INTERFACE axis port = input

#pragma HLS INTERFACE axis port = output

int rowA[1024];

int matB[1024][1024];

int sum=0;

//matB

for(int i=0;i<1024;i++){

#pragma HLS PIPELINE

for(int j=0;j<1024;j++){

#pragma HLS UNROLL

matB[i][j]=input[1024\*1024+i\*1024+j];

}

}

for (int i = 0; i < 1024; i++) {

for(int r=0;r<1024;r++){

rowA[r]=input[i\*1024+r];

}

for (int j = 0; j < 1024; j++) {

for (int k = 0; k < 1024; k++) {

sum += rowA[k] \* matB[k][j];

}

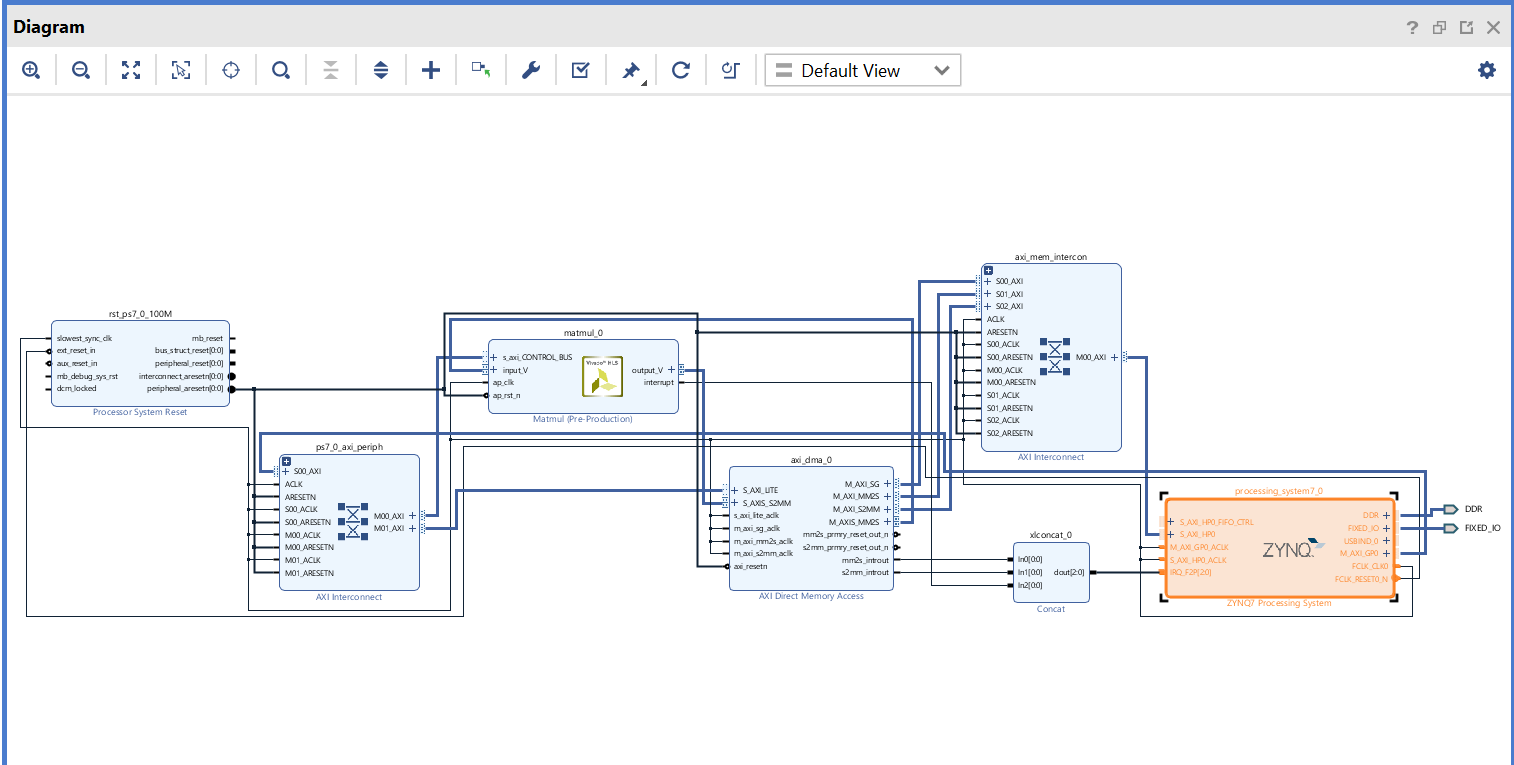
output[i\*1024+j]=sum;

sum=0;

}

}

}



Matmul IP가 불러와지지 않는다..

메모리 문제를 해결할 방법을 생각했다. matB를 저장하지 않고 그때그때 불러쓰면 된다. (stream은 같은 루프 안에서라면 순서대로 접근해야하고 unroll하면 안된다.)

#include "ap\_int.h"

struct myInt{

int data;

ap\_int<1> last;

};

void matmul(myInt input[1024\*1024\*2],myInt output[1024\*1024]){

#pragma HLS INTERFACE s\_axilite port=return bundle=CONTROL\_BUS

#pragma HLS INTERFACE axis port = input

#pragma HLS INTERFACE axis port = output

int sum=0;

int c;

int row[1024];

for (int i = 0; i < 1024; i++) {

for(int r=0;r<1024;r++){

#pragma HLS UNROLL

row[r]=input[i\*1024+r].data;

}

for (int j = 0; j < 1024; j++) {

for (int k = 0; k < 1024; k++) {

c=input[1024\*1024+k\*1024+j].data;

sum+=row[k]\*c;

}

output[i\*1024+j].data=sum;

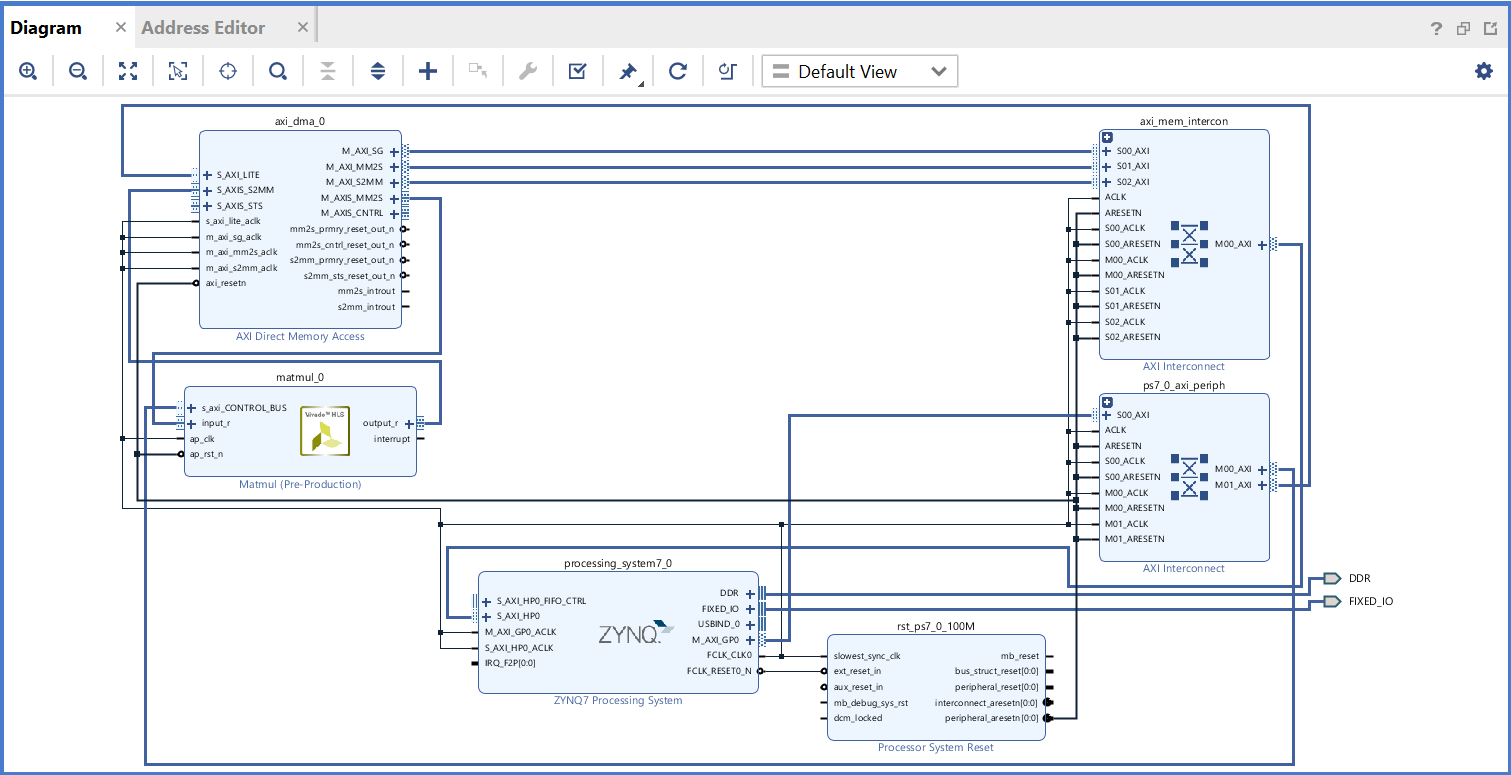
output[i\*1024+j].last=input[i\*1024+j].last;

sum=0;

}

}

}



DMA port is not idle 문제가 있다.

느낀점

매일매일 조금씩 해야겠다.

처음엔 무얼 해야하는지도 모르겠고 막막했는데 하다보니 조금씩 됐다.

질문이 생기면 혼자 해보겠다고 끙끙대는 시간을 조금 줄여야겠다.

참고문헌

HLS - <https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/sdsoc_doc/topics/calling-coding-guidelines/concept_pipelining_loop_unrolling.html>

HLS -

<https://github.com/drichmond/PYNQ-HLS>

HLS Pragma -

<https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/jit1504034365862.html>

DMA, AXI Stream -

<https://www.youtube.com/watch?v=3So1DPe2_4s>

Pynq DMA -

<http://www.fpgadeveloper.com/2018/03/how-to-accelerate-a-python-function-with-pynq.html>

Weekly Progress - 2주차 최명규

수요일 7월15일 - Axi stream과 dma를 쓰는 방식이 1024\*1024든 128\*128이든 계속 성공하지 못해서 s\_axilite를 이용, 작게작게 쪼개는 방법으로 선회하였다.

방법 1.

Row 한 줄과 column 한 줄을 곱하는 IP와 그 드라이버를 만들고 이것을 1024\*1024회 반복 실행한다.

Struct.pack 함수를 사용하면 시간을 더 단축시킬 수 있다.

void matmul1x1(int rowA[1024],int colB[1024],int& result){

#pragma HLS INTERFACE s\_axilite port=rowA

#pragma HLS INTERFACE s\_axilite port=colB

#pragma HLS INTERFACE s\_axilite port=result

#pragma HLS INTERFACE ap\_ctrl\_none port=return

int sum=0;

for(int i=0;i<1024;i++){

#pragma HLS PIPELINE

#pragma HLS UNROLL

sum+=rowA[i]\*colB[i];

}

result=sum;

}



결과비교

Numpy 행렬곱 - 약 100초

Custom HW - 약 60000초

Custom HW with sturct.pack - 약 10000초

결과 분석

약 100배 가량 차이가 난다. 한 줄 한 줄이 아니라 배치사이즈를 늘리는 것 같이 여러줄을 IP Core로 작성하면 어떨까?

포트도 여러개로 나누면 빨라지지 않을까?

1row, 64column, 1port(size=64) 는 약 65000사이클이 필요하고

1row, 64column, 2port(size=32) 는 약 32800사이클이 필요하다.

1row, 64column, 4port(size=16) 는 약 16400사이클이 필요하다.

1row, 128column ,8port(size=16)은 메모리 초과로 unroll 이 잘 안된다.

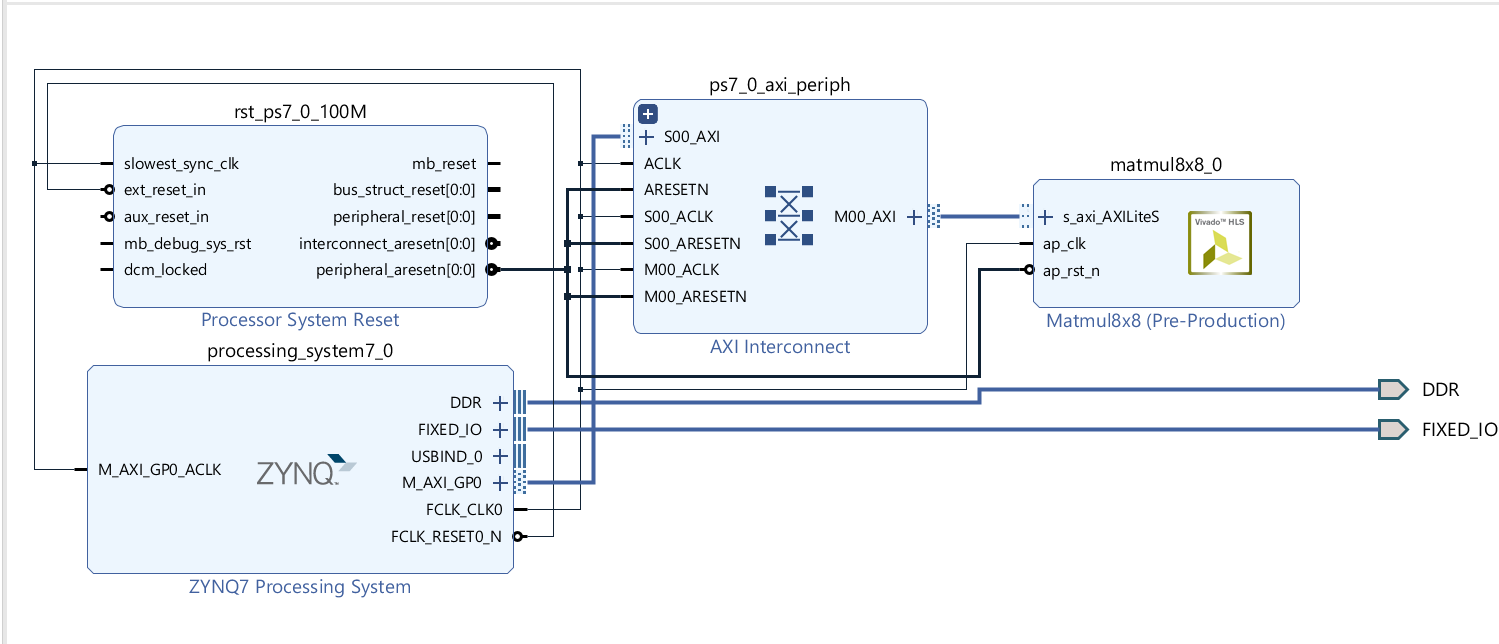
1\*(16\*4)보다 8\*8이 더 빠르지 않을까? 8\*8은 약 75000 사이클이 필요하다

그렇다면 시간을 줄이는 데에 포트 개수가 중요한게 아닐까? (1\*8)\*(1\*8)이 약 1100 사이클을 소모한다 포트를 늘려서 한꺼번에 접근하는 것이 중요한 것 같다.

1row, 64column, 4port(size=16) 은 약 5086 초가 걸렸고

8row 8column 8port each(size=1) 는 약 1290초가 걸렸다.

8x8 블록다이어그램



소스코드

void matmul8x8(int rowA[1024],

int rowB[1024],

int rowC[1024],

int rowD[1024],

int rowE[1024],

int rowF[1024],

int rowG[1024],

int rowH[1024],

int colA[1024],

int colB[1024],

int colC[1024],

int colD[1024],

int colE[1024],

int colF[1024],

int colG[1024],

int colH[1024],

int (&result)[64]){

#pragma HLS INTERFACE s\_axilite port=rowA

#pragma HLS INTERFACE s\_axilite port=rowB

#pragma HLS INTERFACE s\_axilite port=rowC

#pragma HLS INTERFACE s\_axilite port=rowD

#pragma HLS INTERFACE s\_axilite port=rowE

#pragma HLS INTERFACE s\_axilite port=rowF

#pragma HLS INTERFACE s\_axilite port=rowG

#pragma HLS INTERFACE s\_axilite port=rowH

#pragma HLS INTERFACE s\_axilite port=colA

#pragma HLS INTERFACE s\_axilite port=colB

#pragma HLS INTERFACE s\_axilite port=colC

#pragma HLS INTERFACE s\_axilite port=colD

#pragma HLS INTERFACE s\_axilite port=colE

#pragma HLS INTERFACE s\_axilite port=colF

#pragma HLS INTERFACE s\_axilite port=colG

#pragma HLS INTERFACE s\_axilite port=colH

#pragma HLS INTERFACE s\_axilite port=result

#pragma HLS INTERFACE ap\_ctrl\_none port=return

int sum[64];

for(int i=0;i<64;i++){

#pragma HLS PIPELINE

#pragma HLS UNROLL

sum[i]=0;

}

for(int k=0;k<1024;k++){

#pragma HLS PIPELINE

sum[0]+=rowA[k]\*colA[k];

sum[1]+=rowA[k]\*colB[k];

sum[2]+=rowA[k]\*colC[k];

sum[3]+=rowA[k]\*colD[k];

sum[4]+=rowA[k]\*colE[k];

sum[5]+=rowA[k]\*colF[k];

sum[6]+=rowA[k]\*colG[k];

sum[7]+=rowA[k]\*colH[k];

sum[8]+=rowB[k]\*colA[k];

sum[9]+=rowB[k]\*colB[k];

sum[10]+=rowB[k]\*colC[k];

sum[11]+=rowB[k]\*colD[k];

sum[12]+=rowB[k]\*colE[k];

sum[13]+=rowB[k]\*colF[k];

sum[14]+=rowB[k]\*colG[k];

sum[15]+=rowB[k]\*colH[k];

sum[16]+=rowC[k]\*colA[k];

sum[17]+=rowC[k]\*colB[k];

sum[18]+=rowC[k]\*colC[k];

sum[19]+=rowC[k]\*colD[k];

sum[20]+=rowC[k]\*colE[k];

sum[21]+=rowC[k]\*colF[k];

sum[22]+=rowC[k]\*colG[k];

sum[23]+=rowC[k]\*colH[k];

sum[24]+=rowD[k]\*colA[k];

sum[25]+=rowD[k]\*colB[k];

sum[26]+=rowD[k]\*colC[k];

sum[27]+=rowD[k]\*colD[k];

sum[28]+=rowD[k]\*colE[k];

sum[29]+=rowD[k]\*colF[k];

sum[30]+=rowD[k]\*colG[k];

sum[31]+=rowD[k]\*colH[k];

sum[32]+=rowE[k]\*colA[k];

sum[33]+=rowE[k]\*colB[k];

sum[34]+=rowE[k]\*colC[k];

sum[35]+=rowE[k]\*colD[k];

sum[36]+=rowE[k]\*colE[k];

sum[37]+=rowE[k]\*colF[k];

sum[38]+=rowE[k]\*colG[k];

sum[39]+=rowE[k]\*colH[k];

sum[40]+=rowF[k]\*colA[k];

sum[42]+=rowF[k]\*colB[k];

sum[42]+=rowF[k]\*colC[k];

sum[43]+=rowF[k]\*colD[k];

sum[44]+=rowF[k]\*colE[k];

sum[45]+=rowF[k]\*colF[k];

sum[46]+=rowF[k]\*colG[k];

sum[47]+=rowF[k]\*colH[k];

sum[48]+=rowG[k]\*colA[k];

sum[49]+=rowG[k]\*colB[k];

sum[50]+=rowG[k]\*colC[k];

sum[51]+=rowG[k]\*colD[k];

sum[52]+=rowG[k]\*colE[k];

sum[53]+=rowG[k]\*colF[k];

sum[54]+=rowG[k]\*colG[k];

sum[55]+=rowG[k]\*colH[k];

sum[56]+=rowH[k]\*colA[k];

sum[57]+=rowH[k]\*colB[k];

sum[58]+=rowH[k]\*colC[k];

sum[59]+=rowH[k]\*colD[k];

sum[60]+=rowH[k]\*colE[k];

sum[61]+=rowH[k]\*colF[k];

sum[62]+=rowH[k]\*colG[k];

sum[63]+=rowH[k]\*colH[k];

}

for(int i=0;i<64;i++){

#pragma HLS PIPELINE

#pragma HLS UNROLL

result[i]=sum[i];

}

}

Weekly Progress 3주차

1. Systolic-Array Implementation of Matrix-By-Matrix Multiplication

이 방법은 행렬곱이 AB=C 라고 했을 때 A와 B의 row 와 column들이 slide되면서 계산하는 방식이다. 이는 얼핏 봐서는 평범한 행렬곱 연산같이 O(n^3)의 시간복잡도를 갖는 듯 보이지만 HW에서 hardwire 방식으로 구현한다면 C가 될 행렬의 각 원소들에 각자 mapping 해줄 수 있어서 훨씬 빠르다.

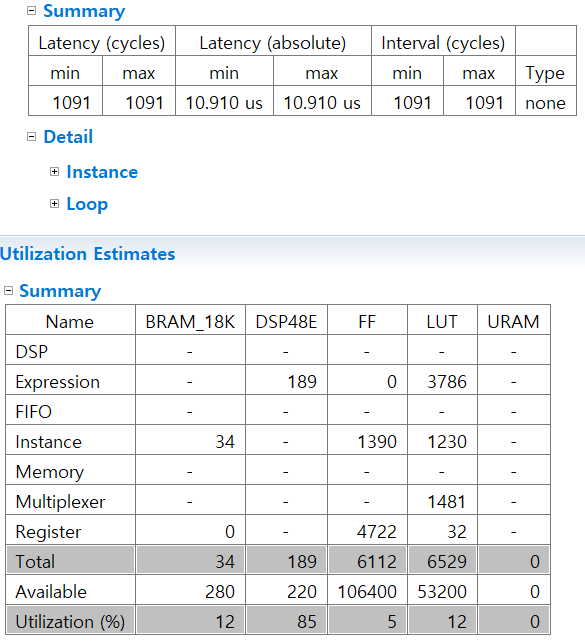
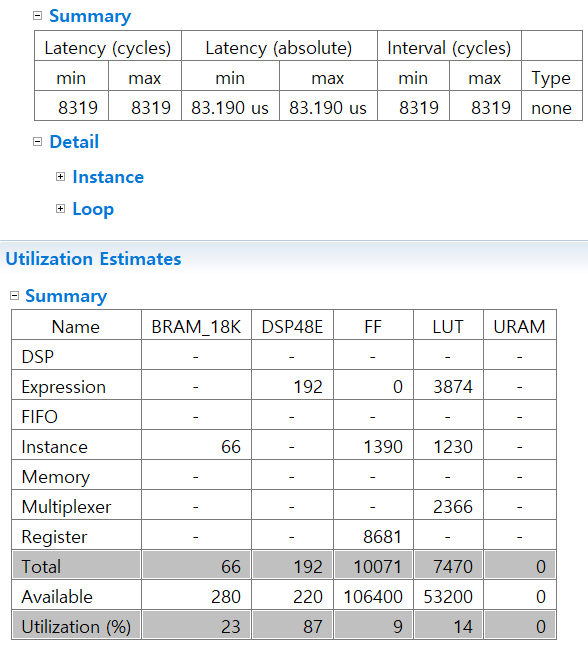
\* 구현시 주의점 - 이중배열로 계산하면 memory dependency 가 pipelining을 방해한다.

1. 결과

같은 양을 계산하는데 2주차 코드보다 리소스도 많이쓰면서 훨씬 느리다.

대신 이 방법은 여러개의 포트가 필요 없을 것이다.

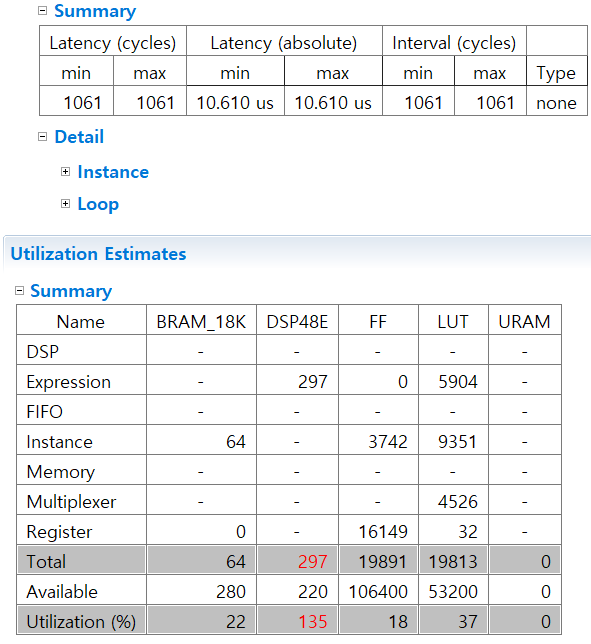
Systolic array 8x8



\*문제점 1

sum[i\*8+j]+=a[i][j]\*b[i][j] 부분에서 warning이 발생한다.

오류가 발생하는 이유는 HLS 가 배열을 synthesize할 때 IO Port를 한개만 만들기 때문이다. 그래서 작업이 끝나기 전에 다시 sum배열에 접근하려니 에러가 발생하는 것이다. #pragma HLS array\_partition complete로 해결하였다.



\* 문제점 2

DSP 가 너무 많이 필요하다.

#pragma HLS resource를 써보았다.

Core 참고문헌

<https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_3/ug902-vivado-high-level-synthesis.pdf>

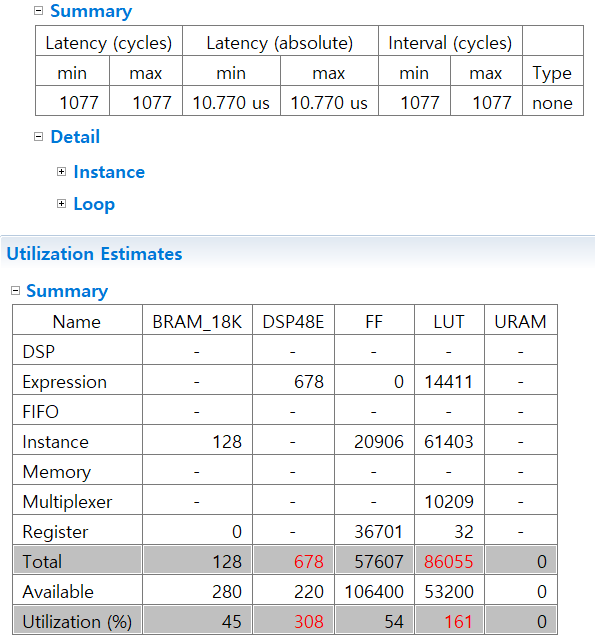
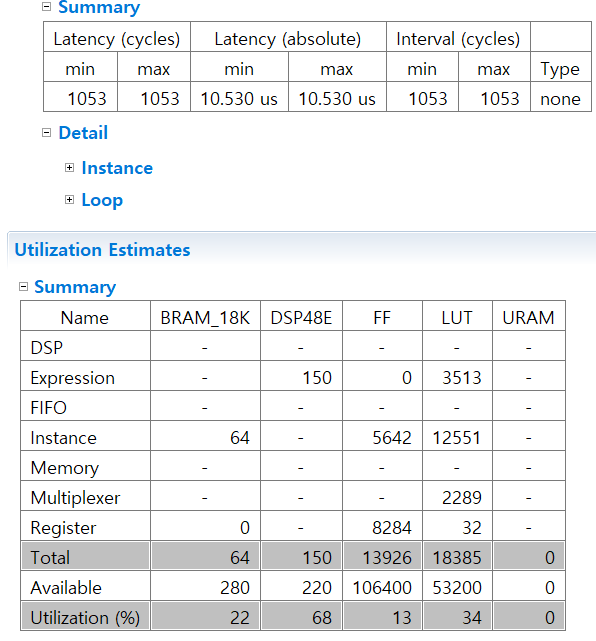
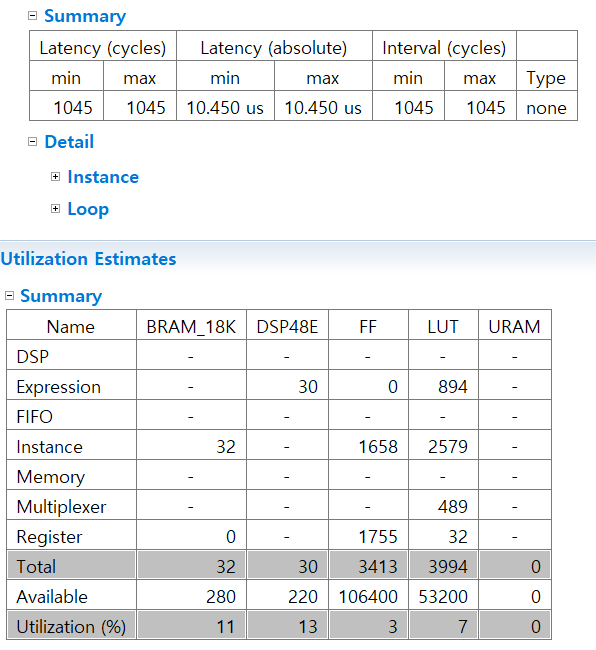
Core Fmul\_nodsp 를 쓰면 array partitioning 이 되지않는다…

Weekly Progress 4주차

Integer with nodsp core 가 존재하지 않기때문에 4\*4로 사이즈를 줄여보았다

systolic array 의 장점은 결과로 나오는 k\*k의 각 원소들이 parallel하게 돌아간다는 것이기 때문에 4x4로 줄이면 resource만 적게 쓰고 걸리는 시간은 동일하다.

4x4 8x8 16x16



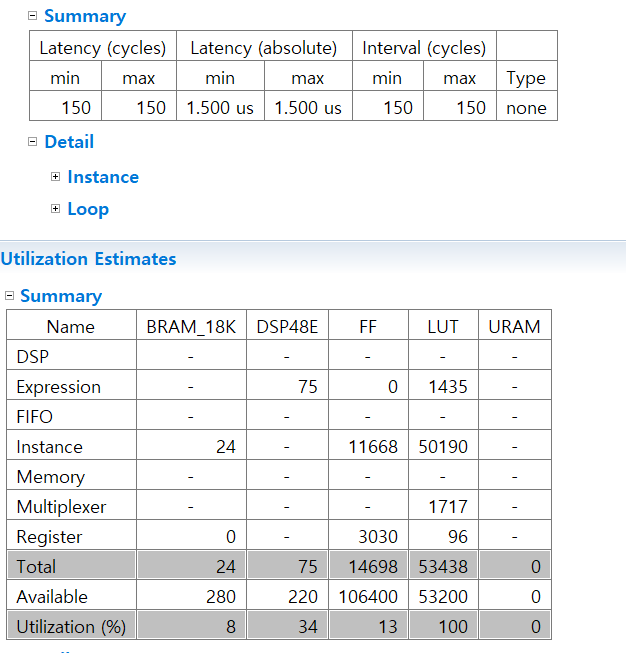
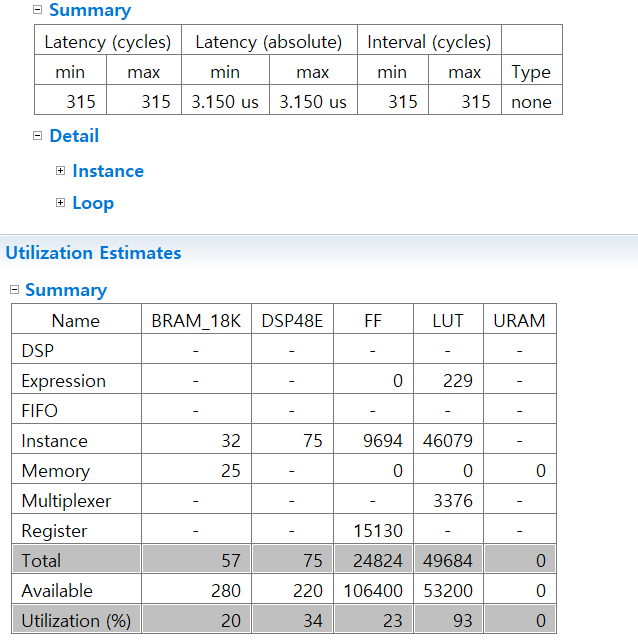
Weekly Progress 5주차

IP 간 통신 실패

Weekly Progress 6주차

한 ip에 두개의 함수를 만들어서 im2col 후 matmul과 평범하게 convolution을 비교

16\*16 ifmap 을 5\*5 filter로 convolution

Im2col basic convolution

Im2col을 하지 않는게 더 성능이 좋았다.

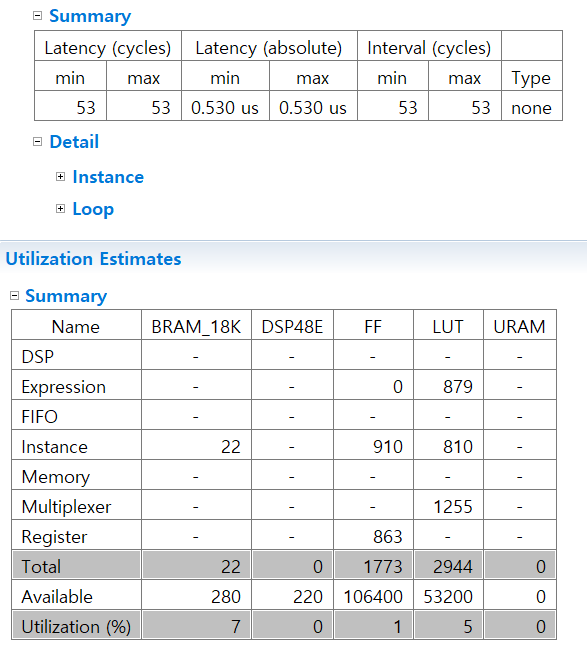
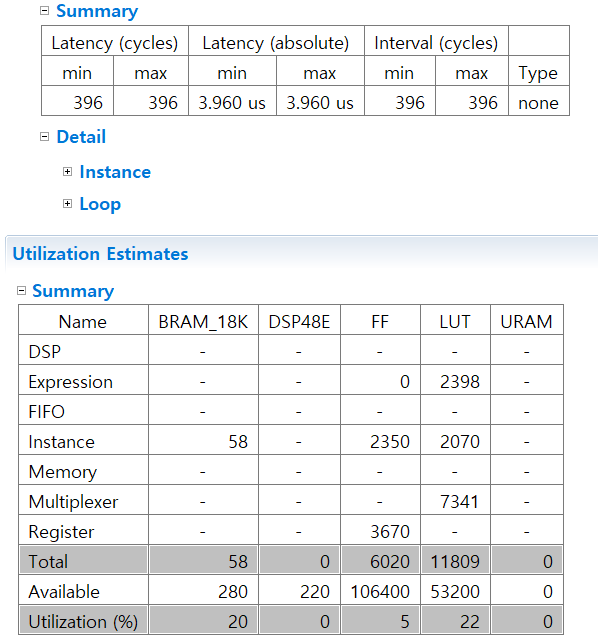
Weekly Progress 7주차

휴식

Weekly Progress 8주차

이번주는 무엇을 하라고 시키지 않으셨기 때문에 LeNet 5에서 사용하는 max Pooling Layer를 만들어 보았다.

28x28 10x10



방학인턴 느낀점

HW로 넘겨줘서 작업을 시키면 뚝딱 빨라질 줄 알았는데 HW가 생각보다 제약이 많았다 Resource도 적고 클럭도 낮아서 가속화를 경험하기 쉽지않았다. 이런 제약을 통해 적은 리소스를 효율적으로 사용하는 법과 욕심을 버리고 작은 부분을 계산할 수 있는 코드를 짜서 여러번 시행하는게 중요함을 배웠다.