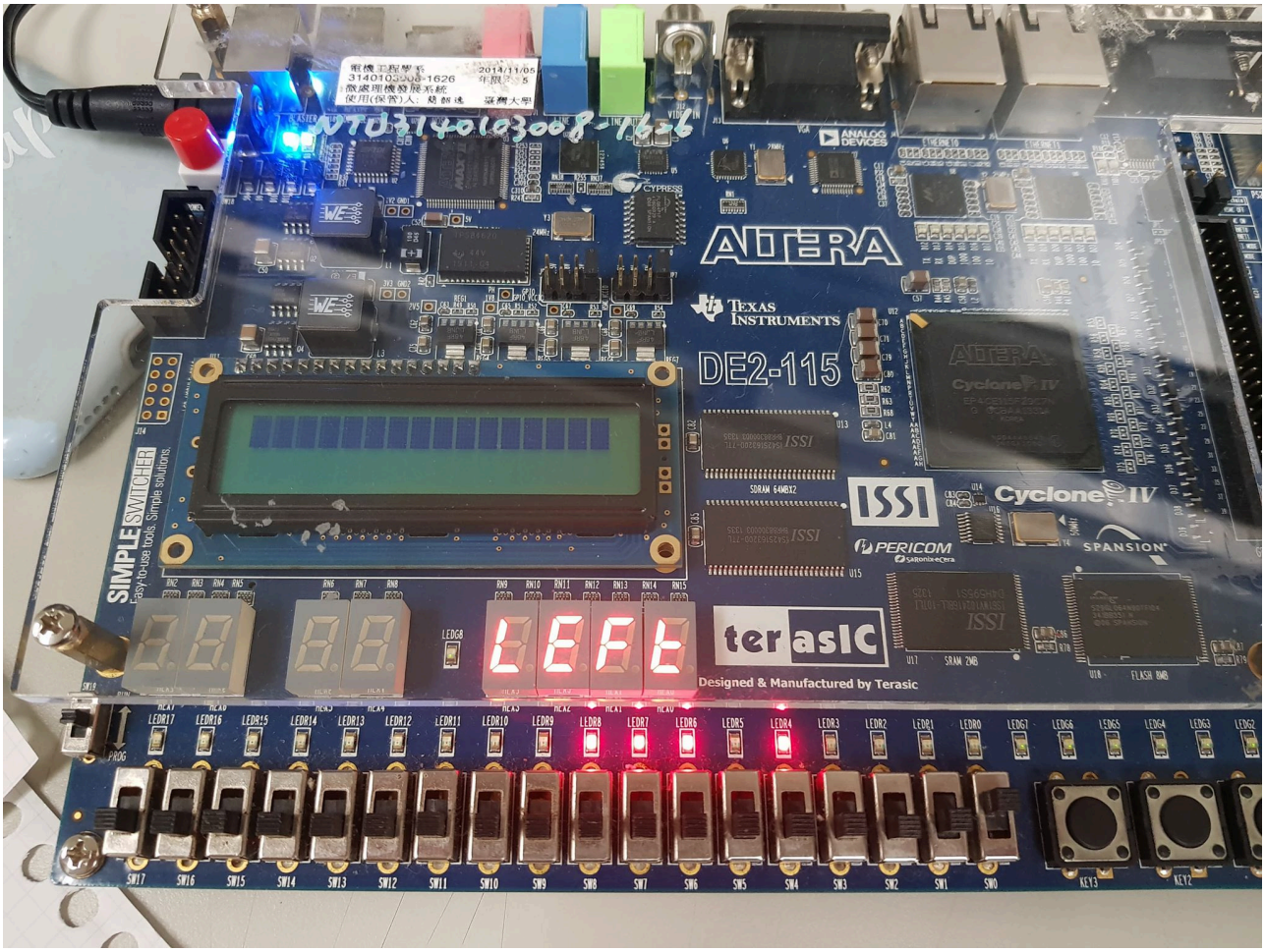
Final project 教學手冊

Team06

B04901074吳倉永 B04901137張振嘉 B04901152劉芸欣

Speech Recognition



目錄

一、簡介專題

近幾年來machine learning被廣泛應用，然而他也需要著相當大的記憶體和運算資源，需要將資料另外放到硬體上做加速。在電腦上，我們最常使用GPU做 training 和 testing 的平行運算加速，因為如果單純以CPU進行是十分耗時的。

FPGA所發揮出的優勢是，它能比GPU更靈活的控制記憶體，運用記憶體和邏輯扎針對 model 做客製化，並且所消耗的功率也比GPU少許多。由於GPU還是和CPU一樣是用instruction的方式來運作，如果我們使用FPGA，就可以減少很多關於 instruction clock 的時間，在做 predicting 也會更快速，達到硬體加速的效果。

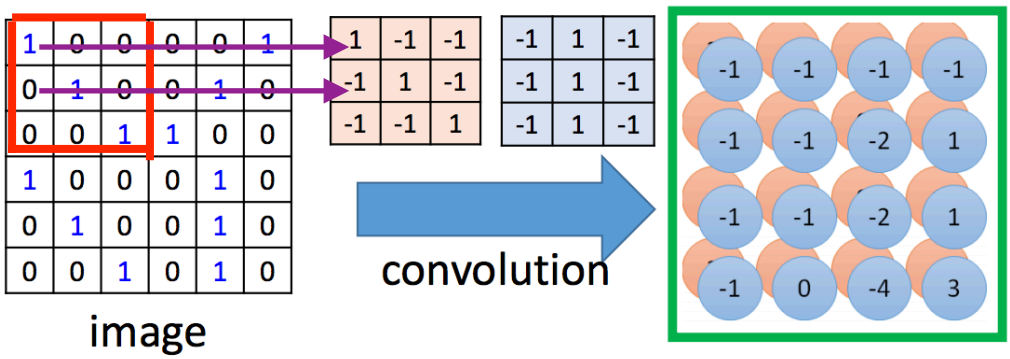
在這次期末專題中，我們這組想在使用 FPGA 實現 CNN 做語音辨識的預測，將語音分成六個 label，分別是 silence、unknown、stop、dog、right、left，辨識我們說的話是六種中的哪一種，並和使用電腦去做預測的結果比較，包刮 runtime、accuracy 等等。

二、關於CNN

Convolution neural network 簡稱 CNN，CNN在分類問題中被廣泛使用，例如影像辨識、手寫辨識，其中分成了三種 layer，分別是Convolution Layer 、 Pooling Layer、Fully Connected Layer。

1. Convolution Layer

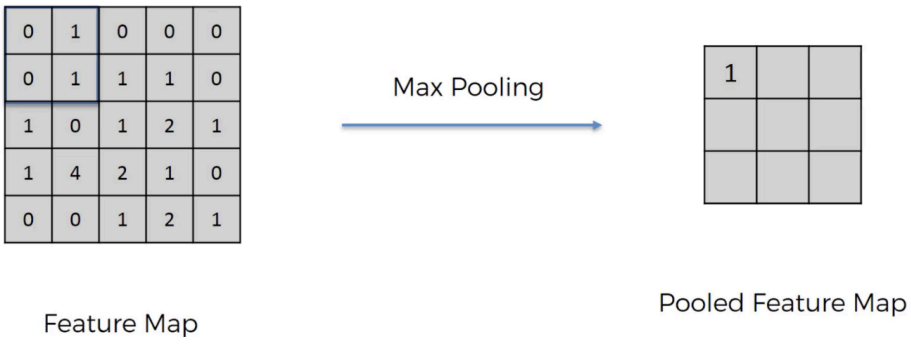
Convolution 運算就是將原始的input map 和 特定的 filter 做 convolution 運算。如下圖：



我們把左邊的image根據中間3X3 的 filter 的 size 去照，首先，我們從左上開始，框出一個3X3 的範圍，根據每個和filter對應的位置去做相乘，在把結果加起來，就可以得到最右邊那張圖左上角的 -1 ，再把filter 往右移一格，繼續做相乘相加，以此類推，就可以得出最右邊的 feature map。

2. Pooling Layer

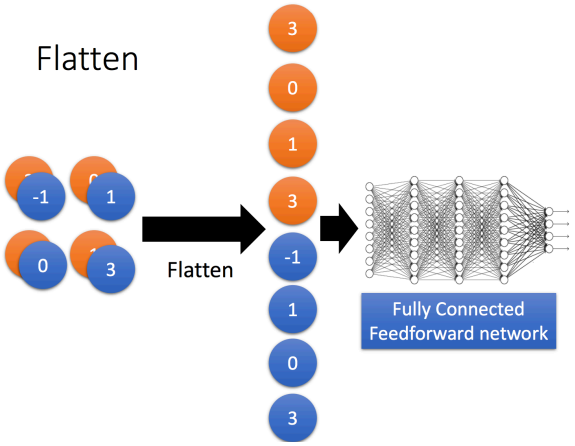
在 Pooling Layer 中我們主要採用 Max Pooling 和 Mean Pooling，Max pooling 的概念很簡單只要挑出矩陣當中的最大值就好，Max Pooling主要的好處是當 feature map 整個平移幾個 Pixel 的話對判斷上完全不會造成影響，以及有很好的抗雜訊功能。如下圖：



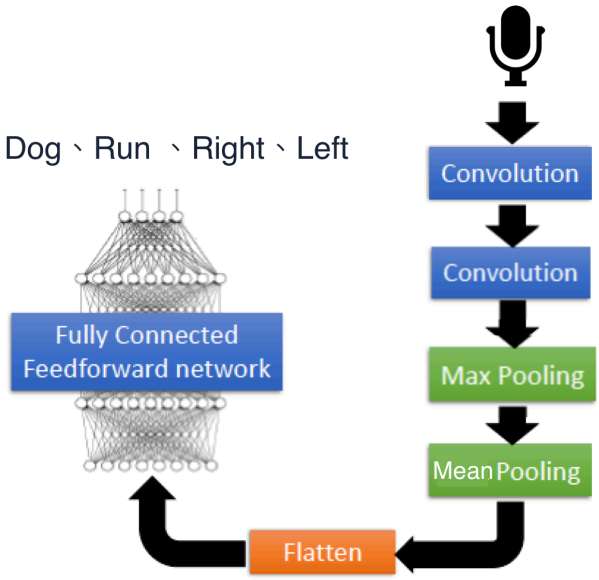
而 Mean pooling 的 方法和Max Pooling 相似，只是把取最大值的方法改成取平均。

3. Fully Connected Layer

基本上fully connected layer的部份就是將之前所的到的feature map flatten 之後，接到最基本的神經網路。如下圖:



介紹完了convolution neural network 後，我們接下來要來講解我們這次使用的model。
如下圖：



我們的input map 是一個 49 X 40 的spectrum，經過第一層convolution layer，filter size是10 X 10 X 1 X 16 ，得到的 feature map 是 40 X 31 X16。

再經過第二層的convolution layer ， filter size 是 5 X 5 X 16 X 16 ，得到 36 X 27 X 16 的 feature map 。

其中，我們每個convolution layer做完都會做一次 relu 的 activation function。把 feature map 中小於 0 的值都省略成 0，可以防止 gradient vanishing。

接著的是max pooling 和 mean pooling ，把 feature map 變成 9 X 6 X 16。

最後加上一層 fully connected layer，輸出出一個 1 X 6 的矩陣，其中這六個數值分別代表六種語音的機率，哪個數值最大就代表我們說出的語音是哪一個字。

三、專題設計

1. python預測

再著手進行硬體實測時，我們先利用 python tensorflow 套件進行實驗分析。我們時用 tensorflow speech command 此 model 實作，建立一個 6 個 label (Slience, Unknown, Stop, Left, Right, Dog) 語音指令的 CNN 模型，利用 tensorflow training，得到一個 「81.5%」的 model，並在用 numpy 當做我們的運算資源。

	precision	time
正常版	81.5%	35.2s
Weight truncate	82%	34.5s
Clip	80.5%	34s
Clip + Weight truncate	81.5%	32s

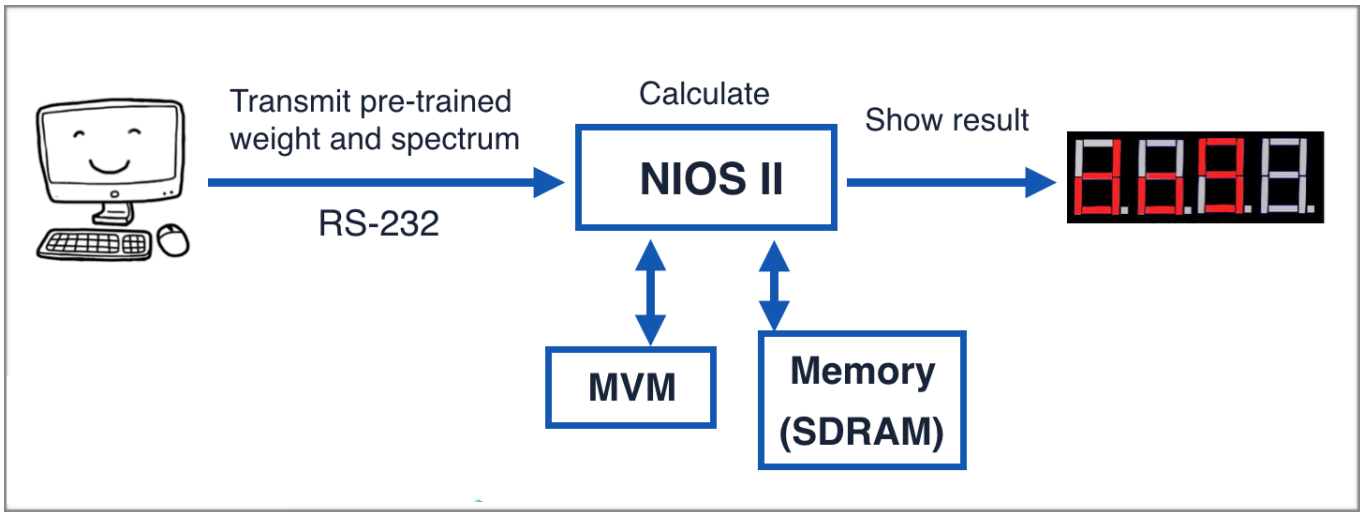
- 正常版：不做任何 data 的精準度的處理，直接跑 numpy 產生的結果
- Weight Truncate：對 model 的 weight 做 truncation 的結果，我們把所有的 weight 的 bit precision 限制再 8 位數，模擬硬體上的限制，得到還不差的結果，且再時間上也有進步。時間上的縮短是可以預期的，因為所需要計算的 bit 變少了，但精準度可能參考就好，不要有太大變動就好。
- Clip：由於 bit precision，我們 relu 出來的 range 不能式 0 ~ 無限大，必須根據我們的 bit precision 做相對硬的調整，經由實驗過後，我們得到 -7.9375 ~ 0.79375 是最合適的範圍(0xb000.0000 ~

0xb111.1111, 第一個是 sign bit。而此版本不做 weight truncate，得到了比較差一點的結果，

- Clip + Weight truncate：套用兩個限制，再利用 numpy 計算，我們得到了一個蠻驚人的結果，再計算時間下降下，準確度卻沒有很大的改變，也讓我們相信硬體的加速，能幫助我們再 numpy 計算的時間更短。

2. Workflow

我們希望使用nios II來控制整個weight運算的flow，藉此重複使用FPGA上面的運算單元，因此設計了這樣的流程：

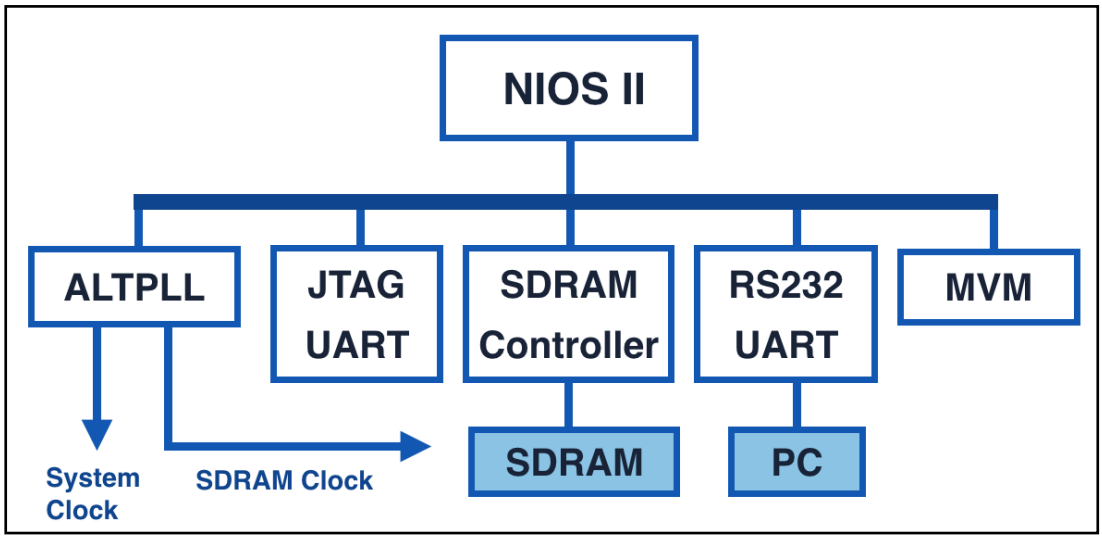


我們先使用RS-232做為傳送工具，將事先在電腦裡train好的model燒入FPGA，同時將麥克風接在電腦，將音訊做好事先處理在傳入FPGA（不過後來時間問題只測試了事先錄好的音訊，後面會再提到），接著使用Nios II來處理整個運算的流程，並且將weight和音訊都存於SDRAM，在計算時從中取出放入我們自己寫的MVM（硬體計算）作處理，本來希望結果能透過一台小車呈現，後來礙於時間關係，簡化而透過七段顯示器呈現。

四、實作方法

1. 系統架構

以下為我們設計出整個專題的架構：



- NIOS II：為FPGA上面附的CPU，我們這次使用它作為Avalon Master，而其他元件作為 Avalon Slave，藉由QSYS的protocol讓NIOS可以有各個元件的使用權，依據不同元件操作read write。
- ATPLL：用來產生不同的Clock，這次我們整個系統都使用75MHz的Clock，特別SDRAM需要快其他元件 3ns 的clock。
- JTAG UART：用來連接電腦並燒 C code 及 debug 所需元件 。
- SDRAM Controller：因為之前使用過的SRAM記憶體太小，而NIOS原本的On-chip memory也不夠，這次主要使用SDRAM作為記憶體，總共是128MB，透過NIOS controller可以省下手刻wrapper的時間，直接IORD、IOWR就可以讀寫SDRAM的資料。

- RS232 URAT：用來傳輸 train 好 model 的 weight 及資料spectrum，再存入 SDRAM 中 ，透過NIOS讀取rs232資料，直接IORD、IOWR就可以讀寫資料。
- MVM：這是自己用verilog寫出的幾個module所構成，用來運用FPGA上的硬體資源做運算，從NIOS接收運算的矩陣，算完再傳回NIOS，用此方法更新不同weight的運算，下面硬體部份會有更詳細的介紹。

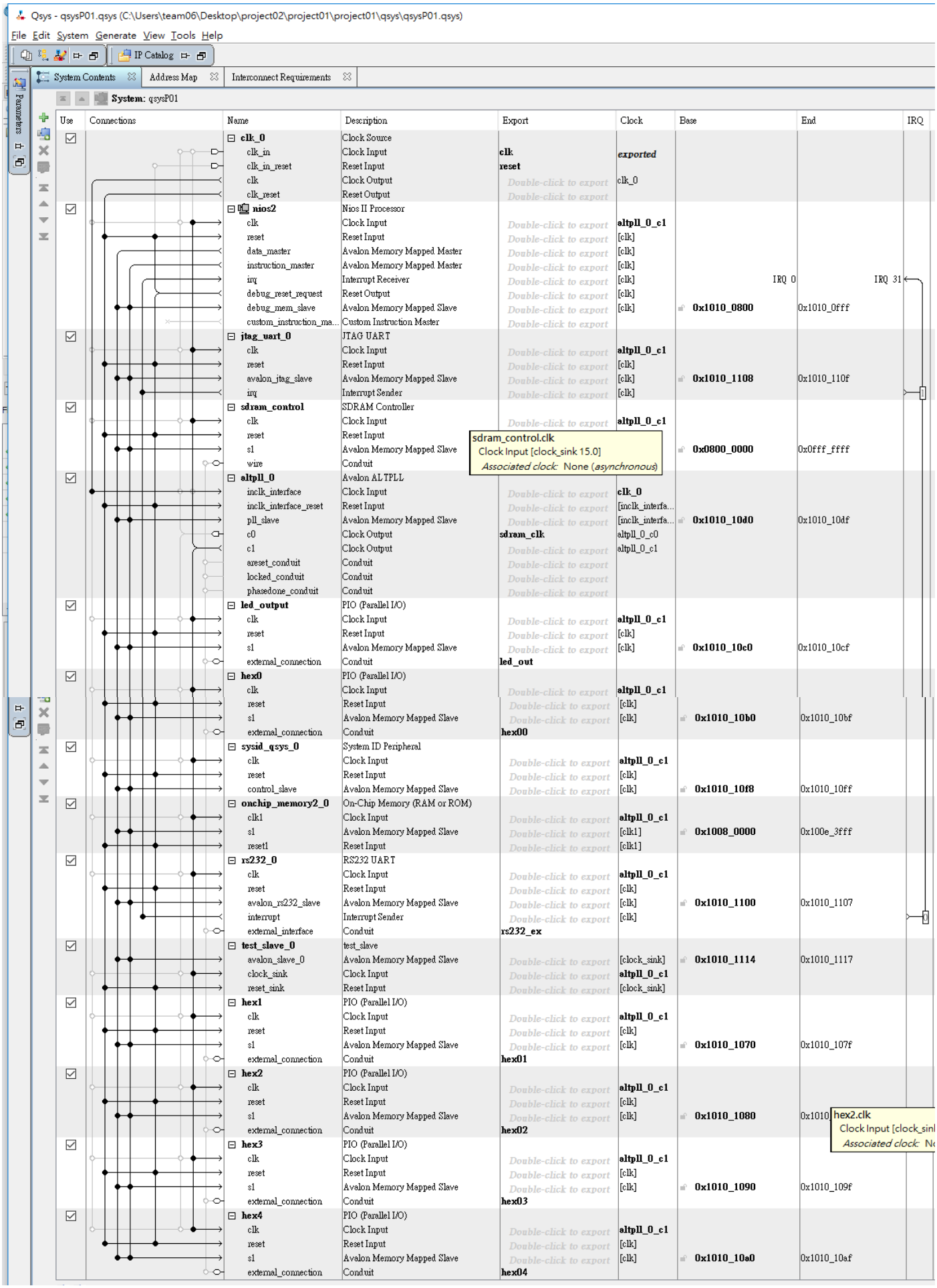
2. 軟體

- QSYS

為了使用NIOS作為主要master，我們必須透過創建NIOS來做各個元件的連接，研究這一部份就花了我們不少時間，研究在QSYS中NIOS要如何連接各個元件，以及clock該怎麼給。

以下為我們建立QSYS的連接圖：

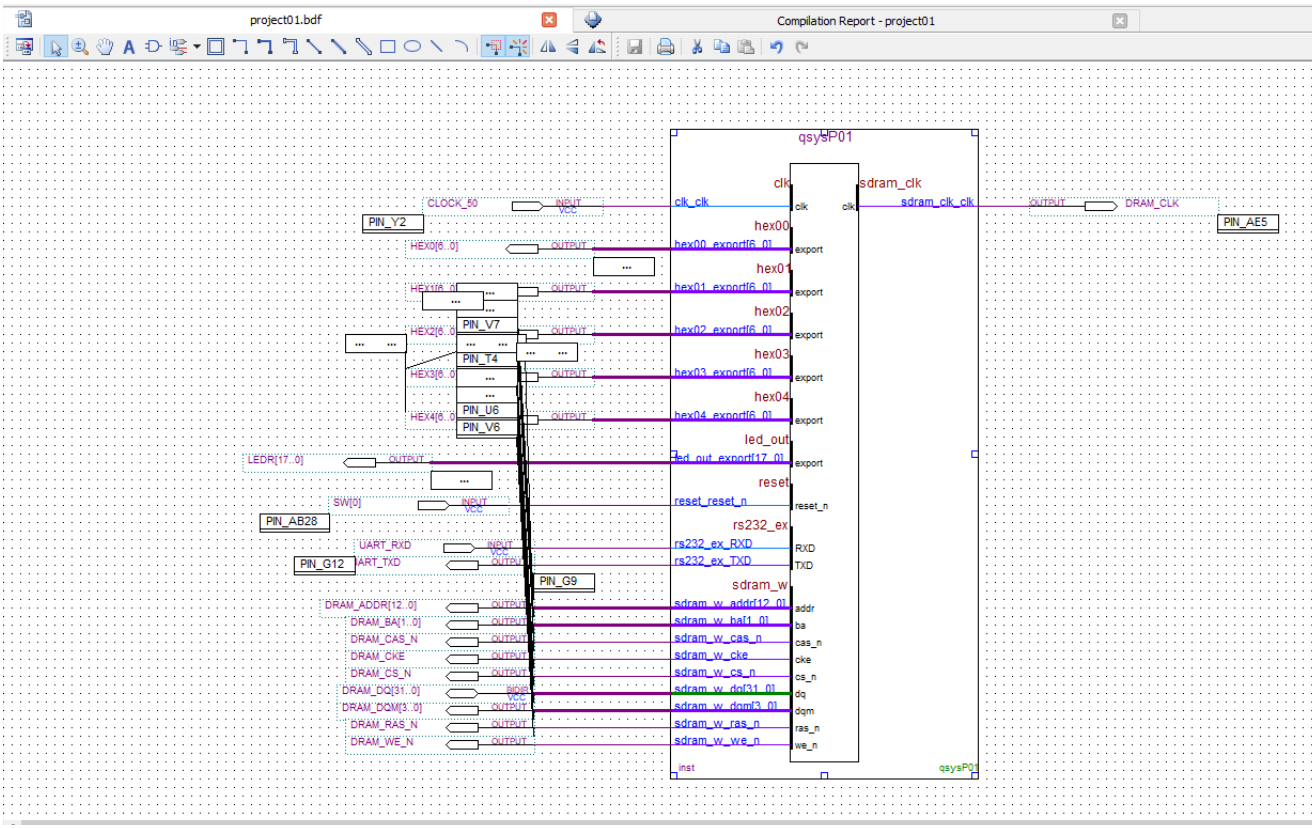
（當中的test_slave就是我們自己打的MVM）


Qsys - qsysP01.qsys (C:\Users\team06\Desktop\project02\project01\project01\qsys\qsysP01.qsys)
File Edit System Generate View Tools Help
IP Catalog
System Contents
Address Map
Interconnect Requirements
System: qsysP01
Use
Connections
Name
Description
Export
Clock
Base
End
IRQ
clk_0 Clock Source
clk_in Clock Input clk exported
clk_in_reset Reset Input reset
clk Clock Output Double-click to export clk_0
clk_reset Reset Output Double-click to export
nios2 Nios II Processor
clk Clock Input Double-click to export altpll_0_c1
reset Reset Input Double-click to export [clk]
data_master Avalon Memory Mapped Master Double-click to export [clk]
instruction_master Avalon Memory Mapped Master Double-click to export [clk]
irq Interrupt Receiver Double-click to export [clk] IRQ 0 IRQ 31
debug_reset_request Reset Output Double-click to export [clk]
debug_mem_slave Avalon Memory Mapped Slave Double-click to export [clk] 0x1010_0800 0x1010_0fff
custom_instruction_ma... Custom Instruction Master Double-click to export
jtag_uart_0 JTAG UART
clk Clock Input Double-click to export altpll_0_c1
reset Reset Input Double-click to export [clk]
avalon_jtag_slave Avalon Memory Mapped Slave Double-click to export [clk] 0x1010_1108 0x1010_110f
irq Interrupt Sender Double-click to export [clk]
sdram_control SDRAM Controller
clk Clock Input Double-click to export altpll_0_c1
reset Reset Input
s1 Avalon Memory Mapped Slave 0x0800_0000 0x0fff_ffff
wire Conduit
sdram_control.clk
Clock Input [clock_sink 15.0]
Associated clock: None (asynchronous)
altpll_0 Avalon ALTPLL
inclk_interface Clock Input Double-click to export clk_0
inclk_interface_reset Reset Input Double-click to export [inclk_interfa...
pll_slave Avalon Memory Mapped Slave Double-click to export [inclk_interfa... 0x1010_10d0 0x1010_10df
c0 Clock Output sdram_clk altpll_0_c0
c1 Clock Output Double-click to export altpll_0_c1
areset_conduit Conduit Double-click to export
locked_conduit Conduit Double-click to export
phasedone_conduit Conduit Double-click to export
led_output PIO (Parallel I/O)
clk Clock Input Double-click to export altpll_0_c1
reset Reset Input Double-click to export [clk]
s1 Avalon Memory Mapped Slave Double-click to export [clk] 0x1010_10c0 0x1010_10cf
external_connection Conduit led_out
hex0 PIO (Parallel I/O)
clk Clock Input Double-click to export altpll_0_c1
reset Reset Input Double-click to export [clk]
s1 Avalon Memory Mapped Slave Double-click to export [clk] 0x1010_10b0 0x1010_10bf
external_connection Conduit hex00
sysid_qsys_0 System ID Peripheral
clk Clock Input Double-click to export altpll_0_c1
reset Reset Input Double-click to export [clk]
control_slave Avalon Memory Mapped Slave Double-click to export [clk] 0x1010_10f8 0x1010_10ff
onchip_memory2_0 On-Chip Memory (RAM or ROM)
clk1 Clock Input Double-click to export altpll_0_c1
s1 Avalon Memory Mapped Slave Double-click to export [clk1] 0x1008_0000 0x100e_3fff
reset1 Reset Input Double-click to export [clk1]
rs232_0 RS232 UART
clk Clock Input Double-click to export altpll_0_c1
reset Reset Input Double-click to export [clk]
avalon_rs232_slave Avalon Memory Mapped Slave Double-click to export [clk] 0x1010_1100 0x1010_1107
interrupt Interrupt Sender Double-click to export [clk]
external_interface Conduit rs232_ex
test_slave_0 test_slave
avalon_slave_0 Avalon Memory Mapped Slave [clock_sink] 0x1010_1114 0x1010_1117
clock_sink Clock Input Double-click to export altpll_0_c1
reset_sink Reset Input Double-click to export [clock_sink]
hex1 PIO (Parallel I/O)
clk Clock Input Double-click to export altpll_0_c1
reset Reset Input Double-click to export [clk]
s1 Avalon Memory Mapped Slave Double-click to export [clk] 0x1010_1070 0x1010_107f
external_connection Conduit hex01
hex2 PIO (Parallel I/O)
clk Clock Input Double-click to export altpll_0_c1
reset Reset Input Double-click to export [clk]
s1 Avalon Memory Mapped Slave Double-click to export [clk] 0x1010_1080 0x1010
external_connection Conduit hex02
hex2.clk
Clock Input [clock_sink
Associated clock: No
hex3 PIO (Parallel I/O)
clk Clock Input Double-click to export altpll_0_c1
reset Reset Input Double-click to export [clk]
s1 Avalon Memory Mapped Slave Double-click to export [clk] 0x1010_1090 0x1010_109f
external_connection Conduit hex03
hex4 PIO (Parallel I/O)
clk Clock Input Double-click to export altpll_0_c1
reset Reset Input Double-click to export [clk]
s1 Avalon Memory Mapped Slave Double-click to export [clk] 0x1010_10a0 0x1010_10af
external_connection Conduit hex04

我們了解到Avalon有master跟slave這樣規則，透過data和controller兩個連接傳送溝通，像是當我們使用NIOS作為master時，他就可以主動使用slave的各項資訊，像是寫入讀出slave，並且內部就包好該有的protocol，我們就不用在處一些address或是wait request的溝通，然而，也是因為都包起來的特性會不能看到當中很多溝通的訊息，反而造成debug的困難。而我們在自己打MVM時，也是上網查了有關slave的一些規定照著協定來打，這樣NIOS才可以正確跟MVM連接上。

接著需要給予各個元件PLL輸出的clock，我們給予所有元件75MHZ的clock（也就是圖中的c1），原本以為需直接給予SDRAM controller快 3ns的clock，但在幾次嘗試錯誤後發現，在QSYS中須給予SDRAM controller原本沒有phase改變的clock，而快3ns的clock則須Export出去（也就是圖中的c0），在外面直接連給SDRAM，而不是他的Controller。

而創建QSYS之後，我們使用bdf來連接外面線路：



- NIOS

在使用NIOS時，最主要的就是使用各個元件的傳輸（像是RS232、SDRAM、MVM），我們將SDRAM設為nios所使用的記憶體，所以在c code中創立變數所使用的記憶體就會是SDRAM。而NIOS當中運作的方式就是從QSYS創建中產生的SOPC檔得知各個元件的address，藉由address進行IORD和IOWR，然後藉由改變IORD和IOWR當中的變數REGNUM，每加一就會往下個數字移動，以此讀寫各個元件，而avalon有包好一些function就可以直接使用，像是alt_up_rs232_read_data()。

hello_world.c convolution.h

```
static int FC_UNITS = 864;
static int FIRST_CONV = 10 * 10 * 1 * 16;
static int SECOND_CONV = 5 * 5 * 16 * 16;
static int DATA_SHAPE = 49 * 40;
alt_u8 transform(alt_u8 data) {
    if(data == 128)
        return (alt_u8)0;
    if((data >> 7) == 1)
        data = (((data << 1) >> 1) ^ 0xFF)+1+128;
    return data;
}
void read_event(alt_up_rs232_dev* rs232, alt_u8* w_data, int read_num){
  alt_u8 read_FIFO_used = 0;
  alt_u8 fail = 1;
  alt_u8 data;
  int count = 0;
  while(1){
    //read_FIFO_used = alt_up_rs232_get_used_space_in_read_FIFO(rs232_ptr);
    fail = alt_up_rs232_read_data(rs232 , &data, &par);
    if(!fail){
      data = transform(data);
      *(w_data + count) = data;
      alt_up_rs232_disable_read_interrupt(rs232);
      alt_up_rs232_write_data(rs232, 1);
      alt_up_rs232_enable_read_interrupt(rs232);
      count++;
      //printf("Read num: %d\n",count);
    }
    if(count == read_num){
      printf("Reading data successfully!!!\n");
      break;
    }
  }
}

void printBinary(int val2, int mode, int num_bit) {
   /*
    * Used to print 32 bits, e.g. 10000000_10000000_10000000_10000000
    */
   val2 = (val2 << (32 - num_bit));
   unsigned char *p = (unsigned char*)&(val2) + 3;
   int k = 0;
   int i = 0;
   for(k = 0; k < num_bit >> 3; k++) {
      int val2 = *(p-k);
      for (i = 7; i >= 0; i--) {
```

不過我們在使用時發現了一些問題，之前使用過的rs232，在nios因為不能得到wait request之類的資料，會在當中遺失一些訊息，因為rs232傳進來時會有個暫存空間，nios再一一讀進記憶體中，但當電腦端傳輸過快時會導致暫存器已滿，而nios接收較慢，則會被 drop掉，我們嘗試又失敗了好幾次，

最後則是靠自己在python端寫一層protocal來實作，當傳完訊息要再傳回去回報電腦端，藉此來回讀寫的方式，使訊息不會遺失，雖然用這樣的方法就是完全拋棄原本rs232的protocol，不過為了實作的可行性以及正確性，以及我們傳weight都是在開始語音辨識之前，所以時間延誤還可以接受，我們決定使用此方法。

- program

```
alt_u32 transform_8to32(alt_u8* data_8) {
void printMatrix(alt_u8* matrix, int ROW, int COLUMN, int DEPTH, double precision) {
void flatten_matrix(alt_u8* flattened_matrix, alt_u8* input_fmap,

alt_u8 truncation(alt_16 data) {

void matrix_to_imageCube(alt_16* matrix_result, alt_u8* o_fmap, int height, int width, int depth) {
void mat_mul(alt_u8* matrix, alt_16* result) {
void conv( alt_u8 * data_mat, int image_h, int image_w,

void max_pooling(alt_u8* input_fmap, alt_u8* output_fmap, int height, int width, int depth){
void mean_pooling(alt_u8* input_fmap, alt_u8* output_fmap, int height, int width, int depth){
void conv_interface_1(alt_u8* first_conv, alt_u8* spectrum, alt_u8* con2_fmap){

void conv_interface_2(alt_u8* second_conv, alt_u8* con2_fmap, alt_u8* conv_finish_fmap){

void total_conv(alt_u8* first_conv, alt_u8* second_conv, alt_u8* spectrum, alt_u8* conv_finish_fmap){

void fully_connencted( alt_u8 * data_mat, int image_h, int image_w,
```

接下來來介紹一下我們nios裡的方程式，按照兩層convolution，pooling layer，fully cpnnected layer去完成語音預測。:

- Transform_8to32

  因為我們每一個weight跟data都是8 bits，所以當我們要傳進乘法器中的時候，會需要先把四個8 bits的數組成一個32bits的數字當成乘法器的輸入，如此一來可以省去傳進乘法器的次數

- Flatten_matrix

  每當我們在做convolution的時候，會先把data feature map根據filter size做flatten，在丟進去乘法器去做矩陣運算。

- Matrix_to_imageCube

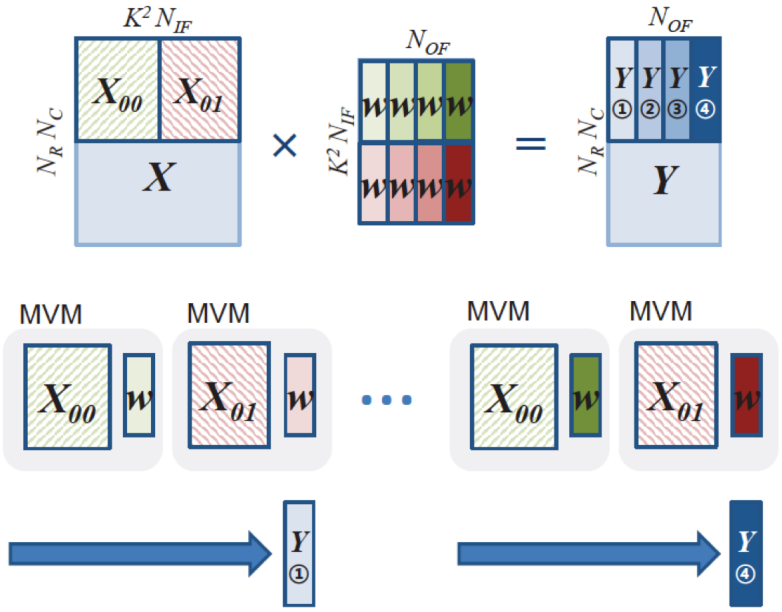
  把CNN flatten轉成matrix去運算完後，出來的矩陣也是一個flatten過的矩陣，所以我們需要透過此function，將算出來的矩陣轉換回來，方便max pooling和fully connected layer使用。

- Mat_mal

  在預測過程中，抓出要運算的256X128的矩陣，還有1X128的weight後，透過此方程式我們會把他們組裝成一個257X128的矩陣，再丟進去乘法器中運算。

- Conv

  這是運作convolution layer的主要方程式，也是最耗工的一個。第一步，先把矩陣分為很多個256X128的矩陣，再來，一個一個找到對應的weight丟進乘法器運算，最後在累加起來成我們output feature map如下圖:



- Max_pooling

顧名思義，這就是運作max pooling layer的方程式，我們把input feature map，用iteration的方式，每次取2X2的size來找出其中最大的feature。

- Mean_pooling

  跟上述方法一樣，來達成mean pooling layer。

- Conv_interface_1

  這是我們第一層convolution layer的主要方程式，裡面先做flatten_matrix，接著使用conv做運算，最後透過image_to_cube把feature map還原，準備傳入下一層的convolution layer。

- Conv_interface_2

  接著的是第二層的convolution layer，基本上和conv_interface_1一樣，不同的點是後面我們加了max pooling和mean pooling，完成要進入fully_connected的feature map。

- Total_conv

  此方程式的功能即是把上述兩個方程式包起來，方便我們使用。

  Fully_connected

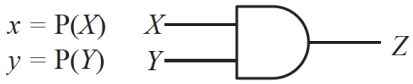
  在做完兩層convolution layer 和pooling layer後，執行fully connected layer的運算，也要把矩陣分成很多個256X128，結果得出一個6X1的解答，只要比較這六個data的大小，就可以知道我們的語音所表達的是什麼。

3. 硬體計算

我們實作出一個一個矩陣加速器，採用 stochastic computing 的方法，而會使用這個方法並無考慮太多，而是期望能成功 predict 出結果。以下會說明演算法及我們所使用的 modules。

- SC Algorithm

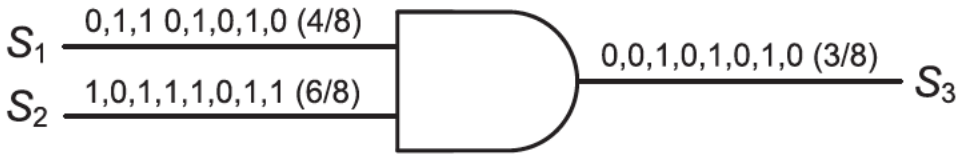
此演算法把 x*y 的運算子簡化成一個 and gate 及 bitstream 就能達成，以期減少時間及運算面積。



$$z = \mathrm{P}(Z) = \mathrm{P}(X) \cdot \mathrm{P}(Y) = x \cdot y$$

Fig. 1: An AND gate computes multiplication of two uncorrelated stochastic bitstreams under unipolar encoding.

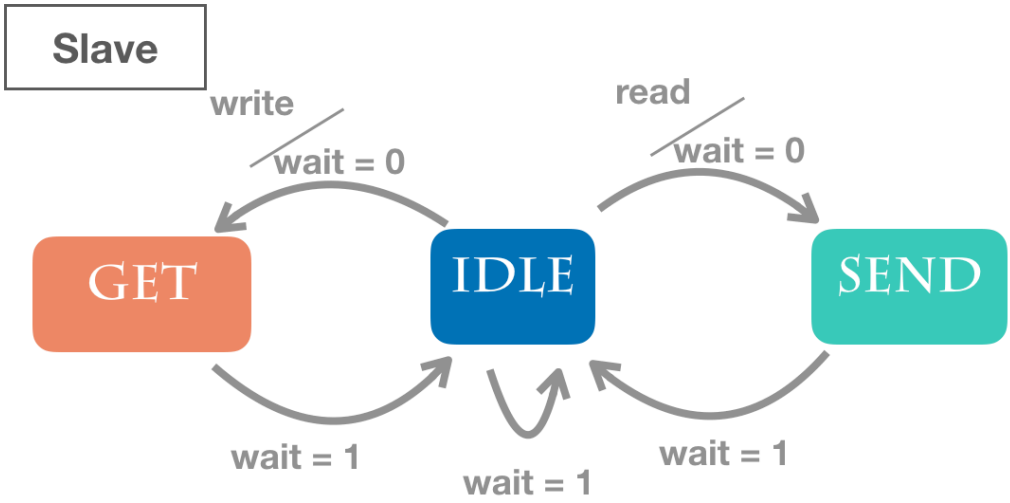
利用機率的方法，計算產生的 Z 中出現 1 的機率，而前面 bitstream 則根據 x, y 的大小，根據精準度 n (bit 的 precision 到第幾位, e.g. 4 位) ，產生 2^n 次方個 bit 並包含 x(or y) 個 1。舉個例子當 x = 4, w = 6 , precision 到 3 時，此時就是計算 (4/8) * (6/8) 的機率，先產生下圖左邊的 bit stream 再 and 每個 bit，就能得知 1 出現的機率回推出結果。



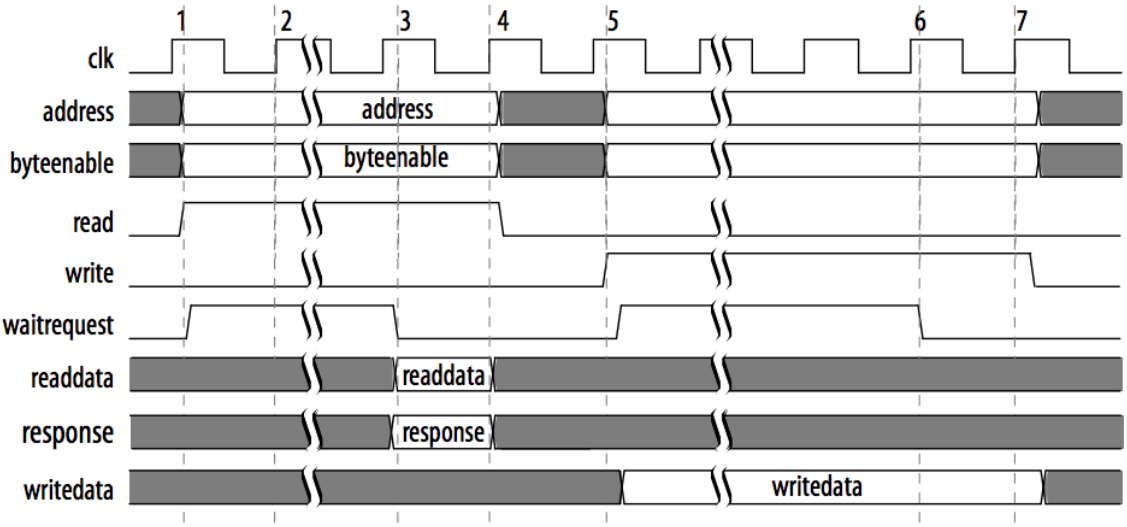
如此，我們只要 2 個 SNG 以及 and gate 取代一個乘法器，此演算法也可支援 2’s complement。不過缺點再於， 計算所需 cycle 以及 error 跟 bit precision 有關，error 可以達到 (n / 2^(n+1)) 個 bit 的數量

- Slave

此 module 為用來接收 NIOS 的資料並傳送給矩陣乘法器進行運算，我們利用 Qsys 當做媒介，宣告自己的 protocol 且必須配合 qsys 所需的介面。使用 read、write、wait_request、write_data、read_data 當做傳送的線路。

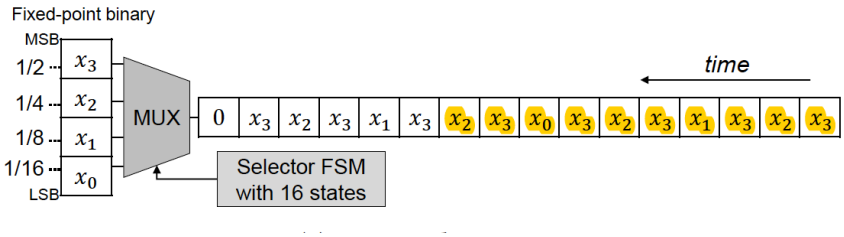


以上是我們的 state diagram，而比較特別的是 wait_request，它是用來告訴自己準備好了沒，達到 flow control 的目的。而我們是根據以下 Documentation 中的 waveform 設計出這個 Finite state machine 。



- Fsm_mux

  這邊是我們的核心之一，也就是 SC 提到的 bit stream 該如何產生。我們可以發現再產生 x, w 兩組 stream 時，如果兩者相關性太高，error 會很大，而 paper 提出一個產生 pseudo random number 的方法。

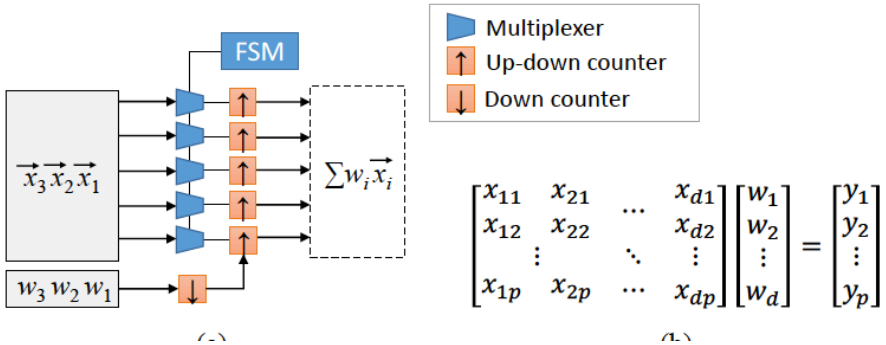


  上圖闡述的就是，利用一個 FSM，產生出 16 個 state 的訊號，控制 mux 輸出 x3, x2 ... (就是 x 的 binary 表示法)，可以發現它產生的 bit stream 是有週期性的，以 x3 的週期最短(因為最大)，paper 聲稱如此能產生較不相關的 bit stream。

  我們再 module 實作上就是設計一個 2^n 個 state 的 FSM，利用一個 counter 算週期，並接上 MUX 控制輸出的 bit stream。

- Mvm

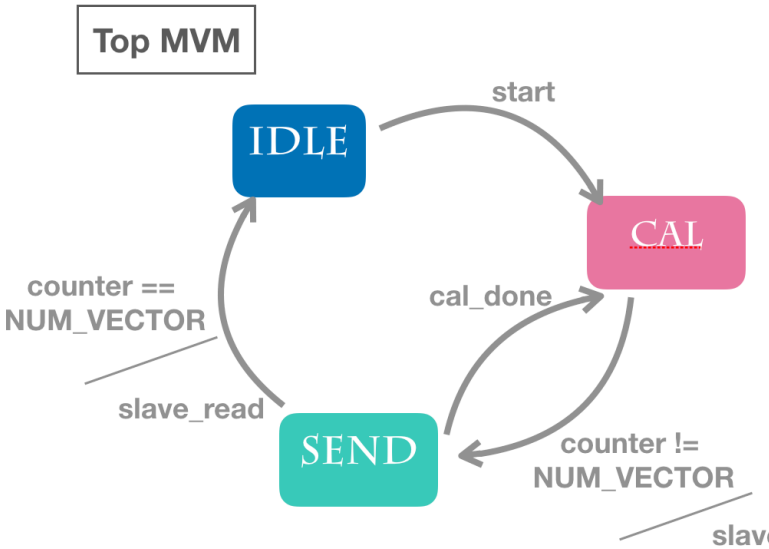
  有了 SNG(Stochastic Number Generator)，再利用 MVM 來當做 Matrix Multiplication 的運算資源。



- Up-down Counter：它是用來「累積」計算結果。它會根據算出來的 Z(SC有提到)，去數出現幾個 1 當做出現 1 的機率
- Down Counter：而這邊比較特別的是當我們計算 x * w 時，我們不用計算 w 的 SNG，而使把 w 當做 counter，告知我們需要看 x 所產生到第幾個週期(也就是要數 Z 數多久)，如此我們可以少一個 FSM 且控制計算 CLOCK cycle，而不必每次都必需數完 2^n 個 bit 才能統計出 Z 的結果。
- 綜合以上兩個 Components，我們需要兩個 counter 來計算 x * w，而當今天 x 是 vector 時，我們只要產生大量的 Up-down counter ，且只需要一個 down-counter(因為 w 都一樣)，就能計算出 X i* wi 的結果，mvm 內就是擁有多個 counter，且只要累積多個 Xi * wi，就能計算一個矩陣，且 cycle 數最大為 2^n * N(n 為準確度，N 為 vector 數量)

- Top_mvm

主要乘法器的 interface，用來蒐集 MVM 的計算結果，並告訴 slave 說何時算完(isAccumulating 訊號)，運算流程如下。當一組 X * w，也就是一組 vector 乘法完成後， top_mvm 就會和 slave(也就是 nios) 要求新的一組 vector 計算，如此我們不需要一開始就將所要算的數字寫成 array 寫在 module 內，耗用資源，而只需再 nios 存好就好，需要算再讓slave write 進去。而利用累積的方式，累積一個 Matrix 所有 vector 的結果出去，因此當 module 內的 counter 與 vector 數量一樣，會傳出slave read，讓nios 把值讀出去。



- 計算方法

  綜合以上的 modules，我們設計以下參數：

  Dimension：256

  Vector 數量：128

  Precision n：n = 8 bit

  由於我們的再做 SC 乘法時它是計算「機率」，因此統計出來的數量要在除以 「2^n」當做結果。我們的 weight 都是 -1~ 1 的數字，而 data 卻因為需要實作 relu 的關係，必須比 |1| 大，因此我們使用「clipping」，將數字 confine 再 -7.9375 ~ 79375 之間，因此原先計算 x / 128 * w / 128 = z / 128 的算式會變成：x / 8 * w / 128 = z / * 16，將結果 shift 4 位就足夠，因此只差再最後讀值的方法。

五、實驗分析

- 硬體計算結果：

  我們利用 1 個 256*128 的 MVM，利用 SC 演算法計算了 predict 200 個結果所需的時間，及準確度，發現結果並沒有想像中的好（如下圖）

	precision	time
正常版	81.5%	35.2s
Weight truncate	82%	34.5s
Clip	80.5%	34s
Clip + Weight truncate	81.5%	32s
NIOS 硬體	45.5%	120.2 s

- 分析：
  - Error rate：

    上述提到，SC 演算法會有 error，且此 error 是累積性的，我們發現在運算的時候會有 offset，因此當我們再產生 mat block 時，補 vector 0 的部分依然會貢獻出 1 的值，讓我們的 offset 最大可以到 128(跟 vector 數目一樣)，造成小數字的乘法會不准。
  - Memory Access：

    我們利用 NIOS 作為主要的 controller，負責控制記憶體的流動，但是我們利用了很多 for loop再產生 array 上，僅僅利用 75 MHz 的 CPU。因此許多時間都耗費再記憶體位置 allocate 上面，整整比利用 2GHz 的 CPU還要 4 倍的時間。
- 結論：

  由於面對的問提是真實世界的聲音，雜訊相當多又受限於精確度的問題，表現不如預期，也發現到 SC 本身的一些缺陷，latency 過高，且如果需要做更多平行運算的話，硬體資源也是需要倍數成長。而 error 則是另一個問題，如果要把 error 降低的話，precision 勢必要提高，而提高後又會有 latency 的問題，兩者之間必須利用實驗做出最好的效果

  我們認為這個實驗讓我們了解到演算法優化的重要性，未來可以針對 SC 的缺點 (time latency, error) 做改進，或許可以利用 clustering 的方法，找出 weight 最有可能的群數，而非限制再我們 define 的 bit precision 內，如此可以在有限的硬體上面，發揮到最小的 error rate。

六、未來展望

- 在NIOS中使用更好的演算法來做feature map 的flattening

我們這次算完一次convolution，都是用for迴圈來把一個矩陣拉直，如此一來會浪費大部分的時間在做flatten，如果我們能運用C code中記憶體的連續性，得出一個更省時間的演算法來做，將會節省大部分的時間，也可以實際去檢測硬體真正在運算的時間。

- 使用更多硬體資源做平行運算

因為我們最後時間不太夠，因為我們一次是算256維，但其實可以複製出更多我們寫出來的乘法器出來一起做運算，讓硬體使用最大化，達到更多的平行運算。

- 改變矩陣運算的SC演算法

使用此演算法我們實驗後發現它會造成得誤差過大，讓我們最後的成功率只有50%，應該選擇別的演算法來讓我們的準確率不會失真，在速度和準確度中間取的一個平衡。

- 實際運用辨識出的語音訊息(ex:機械狗)

我們這次是電腦先錄音，才傳入板子做predicting，未來可以讓錄音的功能實現在FPGA上，這次是因為我們板子資源不夠，沒辦法再騰出訊號spectrum轉換的空間。如果可以實現此功能，就能做出一隻機械狗，可以讓我們呼叫他向前或轉彎，達到我們專題原始預期的效果。

- 如有較高級的FPGA可以嘗試更複雜的model

如上一點所說，這次的DE2-115板子資源不夠，如果能有更大的硬體空間，我們可以實現出更多weight更多層layer的model，並且也可以不用把矩陣分割，造成一次只能算256個單元，如此一來可以節省更多clock的時間，讓predicting的速度再上一層樓。

七、參考資料

[1] Chen, Yu-Hsin, et al. "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks." IEEE Journal of Solid-State Circuits 52.1 (2017): 127-138.

[2] Sim, Hyeonuk, and Jongeun Lee. "A new stochastic computing multiplier with application to deep convolutional neural networks." Proceedings of the 54th Annual Design Automation Conference 2017. ACM, 2017.

[3]Kim, Daewoo, et al. "FPGA implementation of convolutional neural network based on stochastic computing." Field Programmable Technology (ICFPT), 2017 International Conference on. IEEE, 2017.

[4]Alaghi, Armin, and John P. Hayes. "Survey of stochastic computing." ACM Transactions on Embedded computing systems (TECS) 12.2s (2013): 92.

[5]上學期做SC-DNN第六組同學　錢柏均 吳翊玄 何榮晟