Homework 5 Report

Student ID: 112061533 Name: 潘金生

1. Questions
   1. AXI-based (Advanced eXtensible Interface) network-on-chip (NoC).

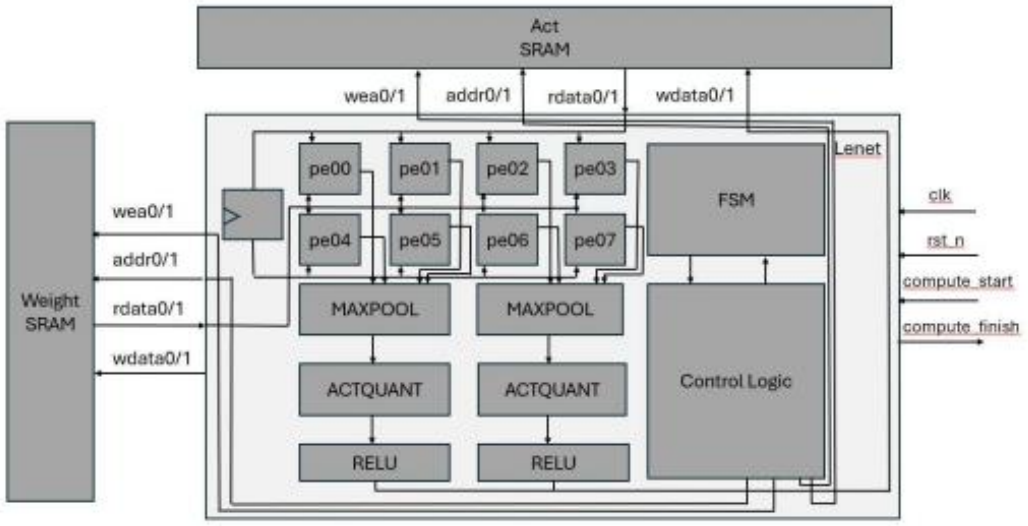
   2. 這段程式碼是用來啟動 accelerator 加速器，會在控制台輸出 Start，顯示啟動過程開始，接著會透過 iowrite32 涵式，將 CMD_MASK_START 這個啟動命令寫入指定暫存器 CMD_REG 中來開始啟動。
      在系統中，C program 會發出 conf_done(Interrrupt)的 IRQ 來啟動 accelerator 運算。

```
// Start accelerators
printf("  Start...\n");
iowrite32(dev, CMD_REG, CMD_MASK_START);
```

   3. 這段程式碼是用來等待 accelerator 加速器完成工作，首先設定 done = 0，然後進行一個迴圈，接著讀取 STATUS_REG 賦值給 done，在透過與 STATUS_MASK_DONE 進行 mask，讀到 acc_done 的值，等到結束迴圈後，將 0x0 寫入 CMD_REG，重置命令，最後控制台輸出 Done。
      在系統中，accelerator 會輸出 acc_done 的訊號啟動 IRQ 來跟 cpu 說已結束運算。

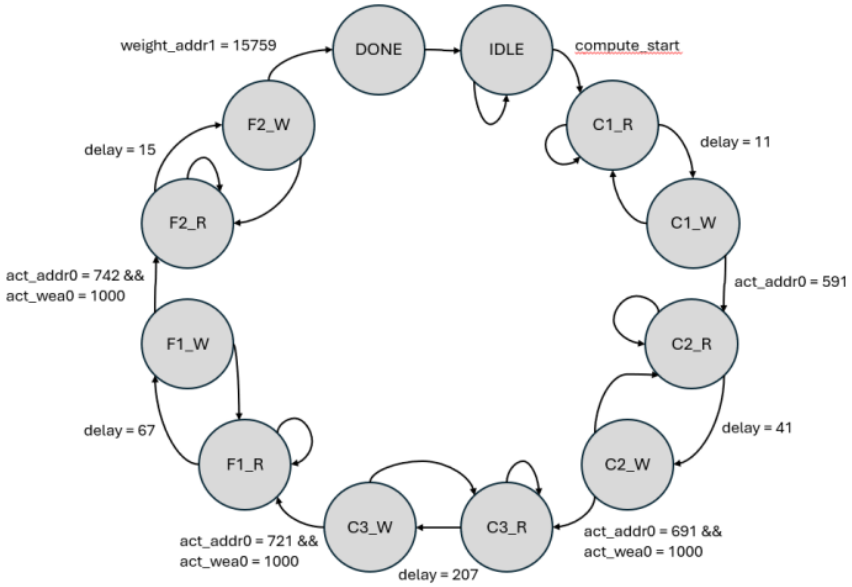
```
// Wait for completion
done = 0;
while (!done) {
  done = ioread32(dev, STATUS_REG);
  done &= STATUS_MASK_DONE;
}
iowrite32(dev, CMD_REG, 0x0);

printf("  Done\n");
```

   4. 附圖為 Lenet 的 Block diagram 和 FSM
      Block Diagram:
      電路主要由五個 module 去撰寫，其中 top module lenet 來整合 submodule pe, actquant, maxpool and relu，且做訊號傳接，延遲訊號等處理。在 Lenet 中，這裡採用 8 個 pe(40 個 MAC)去實現 Lenet，且在過程中採用了 data reusability 來減少資料傳輸的耗能，除此之外，在 Lenet 中，由於 scale 為正數，所以我先進行 maxpool，在做 actquant, relu，在 maxpool 過後，僅會有 2 筆 output，因此只要開 2 個 actquant 和 relu 就好，便可以節省面積。在實作中，也有在 pe, actquant 切 pipeline，避免較長的 path delay，而啟動這 lenet 的訊號為 compute_start，結束時為 compute_finish。



FSM:

FSM 設計的部分，基本上就是判斷地址，這裡的 delay 是代表在讀取時所需要的 cycle，公式如下: 讀取的 data 數 + 等待 2 個 clk(sram 讀取 data 會有延遲)+pipeline cycle 數，其中，在 Fully-connected layer 時，由於我是一次寫四筆，因此他需要等待的 delay 較長，並且在 delay 中，還需留一個 cycle 讓 pe 歸 0 重新運算，因此 fully-connected layer 的公式為: 讀取的 data 數+ 3*等待 clk + 等待 2 個 clk + pipeline cycle 數，其餘層_R(讀取層)以此類推，接著再判斷何時要跳入_W(寫出層)，並且判斷 addr 到何時需要跳入下一個 layer。

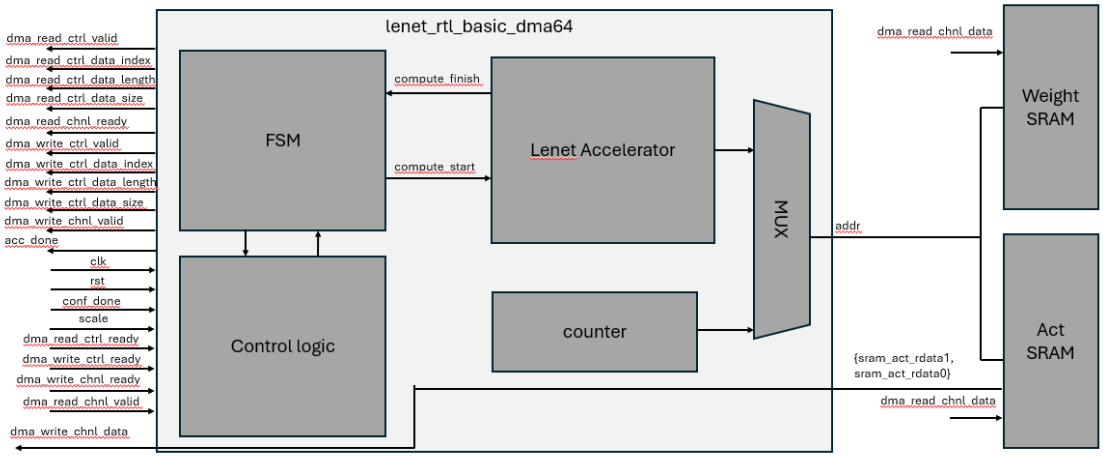


5. 附圖為 DMA controller interface 的 Block diagram 和 FSM

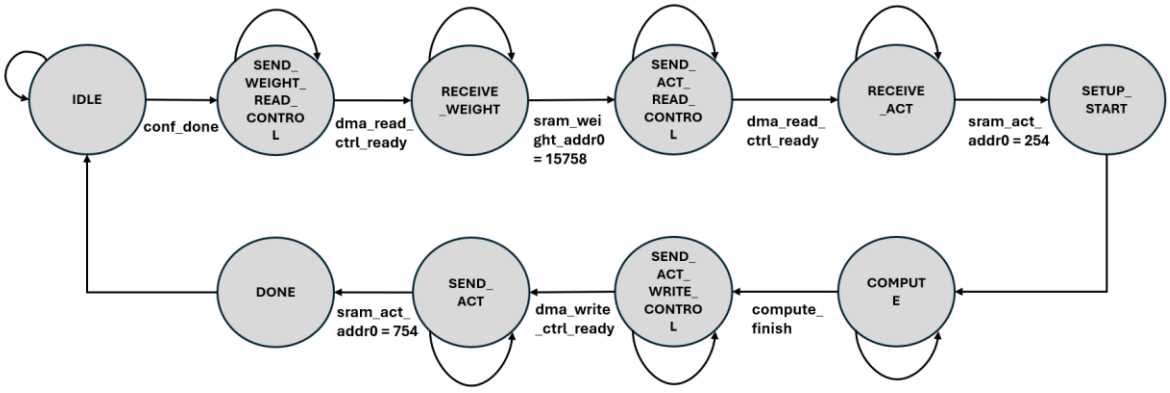
Block diagram:

此次電路主要分成兩個 module 完成，其中一個為負責 DMA interface 的 DMA64，另一個為加速器 Lenet，在本次實作中 DMA64 主要負責處理 DMA

的 Interface，來完成所需要設定的 ctrl 訊號和 transaction 的 protocol，除此之外，由於本次需要從外部的 ram load 訊號進 sram，因此 sram 的地址會由 Lenet accelerator 和 DMA 兩者之間產生，因此就需要透過一個 Mux 去選擇現在是誰要輸出，最後將 Lenet 運算完的值從 sram 寫回 DMA 中。

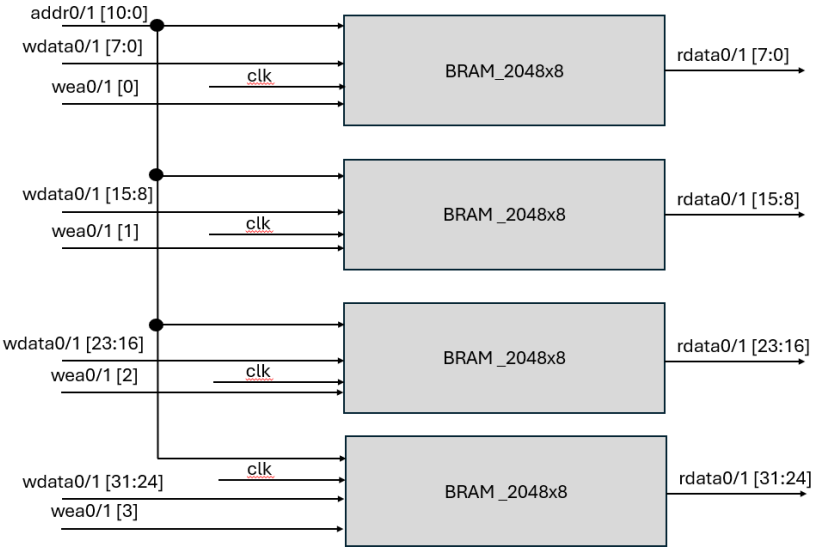


FSM:
DMA FSM 所設計的部分，這次主要分為 10 個 state 去實作。
IDLE: 等待 conf_done
SEND_WEIGHT_READ_CONTROL: 負責傳送 Read ctrl signal(weight)
RECEIVE_WEIGHT: 負責接受 data(weight)
SEND_ACT_READ_CONTROL: 負責傳送 Read ctrl signal(activation)
RECEIVE_ACT: 負責接受 data(activation)
SETUP_START: 產生 pulse signal start 訊號給 lenet accelerator
COMPUTE: lenet accelerator 開始運算
SEND_ACT_WRITE_CONTROL: 傳送 write ctrl signal
SEND_ACT: 傳送 sram data(activation)到 DMA
DONE: 結束運算。

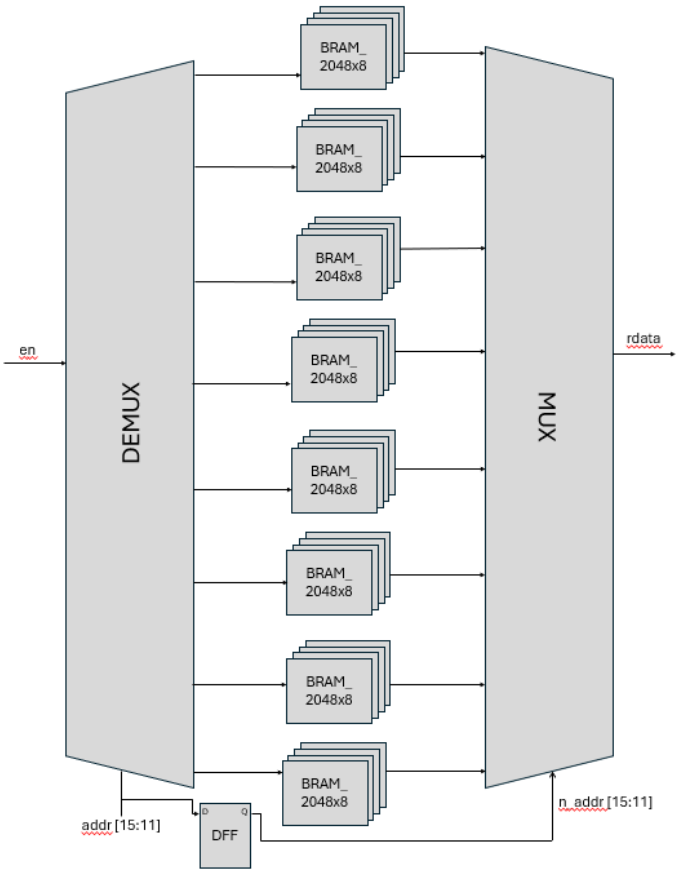


6. 本次作業實作 2 個 sram 的方式，主要透過 ESP 內部提供的 BRAM 2048*8 來實作，其中 2 個 sram 大小分別為 activation_sram 1024*32 和 weight_sram 16384 * 32。

Activation sram: 由於其大小為 1024 * 32，因此需要 4 個 BRAM 來實作，來滿足 32 bits 的資料大小。



Weight sram: 由於其大小為 16384 * 32，因此需要 32 個 BRAM 來實作，其中可以分成兩部分探討，由於資料大小為 32 bits，因此需要 4 個 BRAM 來實作，除此之外，由於 BRAM 的地址最多只能到 2047，因此需要 8 個 array 來完成地址，因此 32 個 BRAM 可以描述為 8 * 4 BRAM，而由於，8 個 array 地址是不同的，因此需要控制 BRAM 的 CE port，來判斷現在要操作第幾個 array，而其控制方式為透過剩餘的 addr[15:11]去判斷，除此之外，由於 BRAM 會延遲 1 個 clk，因此需要在把先前的 addr 延遲一個 clk，我把他稱作 n_addr，用這個 n_addr 去選擇正確的 data，來完成本次的實作。



7. 這是由於我們所設計的 sram 資料大小為 32 bits，但 dram 的資料大小為 64 bits，也就是說 sram 的兩筆資料需要兩個地址，而 dram 的兩筆資料即需要 1 筆資料即可。

8. 設定五組 configuration register，其中的值為五組 scale，並寫入加速器中。

9. 先將 done 設為 0，接著進入 done 等於 0 的就會一直執行的迴圈，接著讀取 STATUS_REG 賦值給 done，透過與 STATUS_MASK_DONE 進行 mask，讀到 acc_done 的值，等到結束迴圈後，將 0x0 寫入 CMD_REG。

2. Result

Item	Description	Unit
RTL simulation	PASS	---

```
#   ------------------
#   Generate input...
#   -> Non-coherent DMA
#   Start...
#   Done
#   validating...
# ===> Image pass!
# ===> Conv1 pass!
# ===> Conv2 pass!
# ===> Conv3 pass!
# ===> FC1 pass!
# ===> FC2 pass!
# [PASS] Congratulation! All results are correct
# ** Failure: Program Completed!
#    Time: 13169494400 ps  Iteration: 0  Process: /testbench/cpu/line__214 File: /users/course/2024S/VLSI20240002/g112061533/esp/socs/xilinx-vcu128-xcvu37p/top.vhd
```

3. Others (optional)
   - 謝謝助教在討論區積極的回覆。