系統晶片設計
SOC Design

LABD

組員:郭銘宸、郭紘碩

學號:311511082、3115120656

系所:電機所碩二、電控所碩二

一、 The SDRAM controller design, SDRAM bus protocol

- SDRAM controller 基本概念

SDRAM controller 負責管理和控制 CPU 與 SDRAM 之間的數據傳輸，是處理器與記憶體間的重要橋梁，而其功能包括存取命令、定時和時序管理以及數據流管理等主要功能，其中，命令的處理便是將由 CPU 端收到的資料及地址等訊息進行解碼，就能判斷出指令是要讀或寫進 SDRAM 中的某個位置；而定時和時序管理為控制 SDRAM 操作時的要求包括記憶體時序如 CAS Latency 和 Refresh Latency 等延遲，以確保數據能夠在正確的時間被寫入或讀取；在數據流管理方面，每當一個 page 被 activate，就會將其資料放入 sense amp buffer 當中等待被讀取，而在加速讀取效率的改善中，便是利用這一特性讓 sense amp buffer 中的資料按照地址順序存入功能類似於 cache 的 fifo 中，隨後進來的資料地址若符合連續地址的情況，便會不斷的將 fifo 中資料送出來減少實際到記憶體中存取的次數，以加速 SDRAM 整體的讀取時間。

- SDRAM controller 設計特性

SDRAM controller 內部通常需要包含多種狀態，而每種狀態都代表著 controller 對記憶體操作過程的不同階段和任務，其中狀態包含基本的 INIT、WAIT、IDLE、REFRESH、ACTIVATE、READ、READ_RES、WRITE、PRECHARGE 等，每個狀態都對映著 controller 執行特定操作過程步驟和下一個狀態轉換的判斷。

在各個狀態中所需的執行週期都有所不同，CAS Latency 是從 row 被 activate 到能夠被讀寫的時間間隔，代表著記憶體在接收到 row address 後需要多長時間才能夠提供資料，而 Precharge Latency 和 Activate Latency 也都代表著 SDRAM 硬體上的限制，他們所需的 cycle 數會直接的影響了 SDRAM 的性能與速度。在 DRAM 刷新的特性方面也在 SDRAM 中展現，他的運行是倚賴內部的計數器，每當到達了固定的週期數量，便會執行一次刷新，這樣的操作保證了 DRAM 中資料的一致性和穩定性，確保資料不會隨電容本身效應而流失。

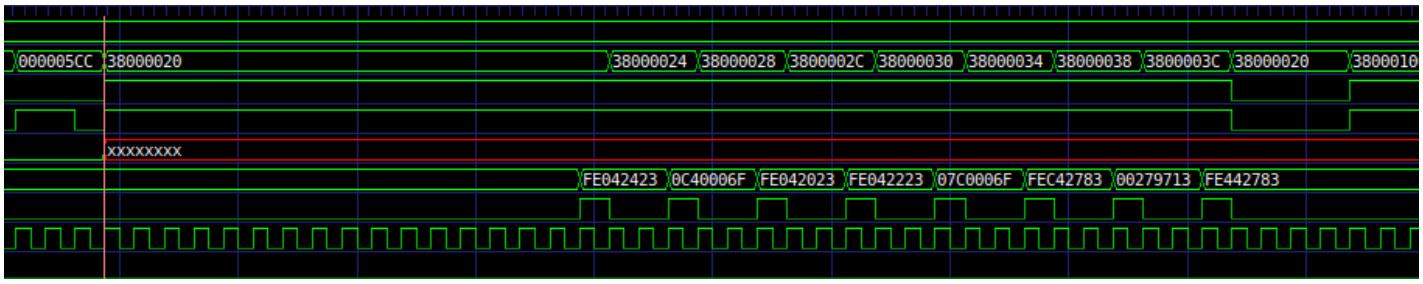
二、 Introduce the prefetch scheme

- Prefetch 設計概念

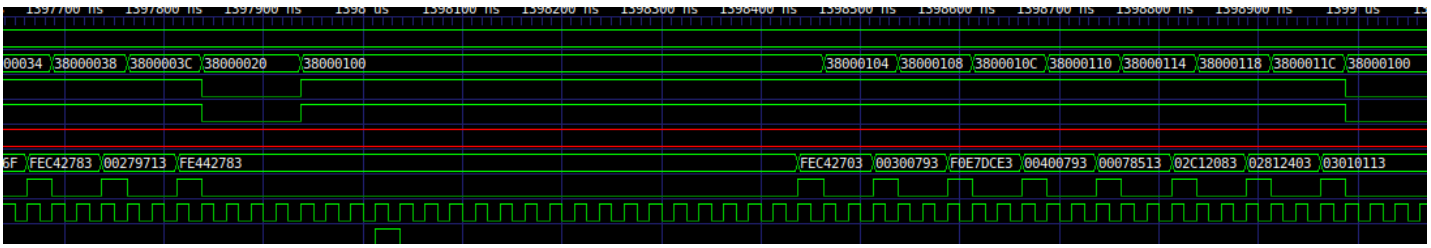
(1) 透過觀察波型，可以發現 CPU 在讀 instruction 時會連續讀 8 個 address 的 instruction，所以我們可以知道將 prefetch buffer 設定在 8 個其效益最大。如下圖(1)所示。

(2) 因為 instruction 的讀取非常規律，也發現知道 8 個連續地址的第一筆，後面 7 筆都會依序 address 加 4，所以可以利用連續 8 筆這個特性，用 fifo 來當作 prefetch buffer，依序與 fifo 比對，將資料送到 WB，prefetch buffer 設計如下圖(4)所示。

(3) 可以透過下面 SDRAM controller 設計圖(下圖(3))可知。若與 fifo address 比對錯誤(這會是新的連續 8 個 address 撈取會發生的事)，則會依照連續 8 個 address 的第一筆 address 往後讀取，包含第一個 address 共會撈 8 筆 instruction。因為連續 8 個 address 的第一筆 address，與下一個連續 8 個 address 的第一筆 address 並不會有關聯，如下圖(2)所示。所以我將 prefetch 方法設計為將連續 8 個 address 的第一筆 address 當作 offset。將 8 個 instruction 儲存完畢，後面的 address 若有比對成功，則直接將 fifo 內的 data 讀到 WB。

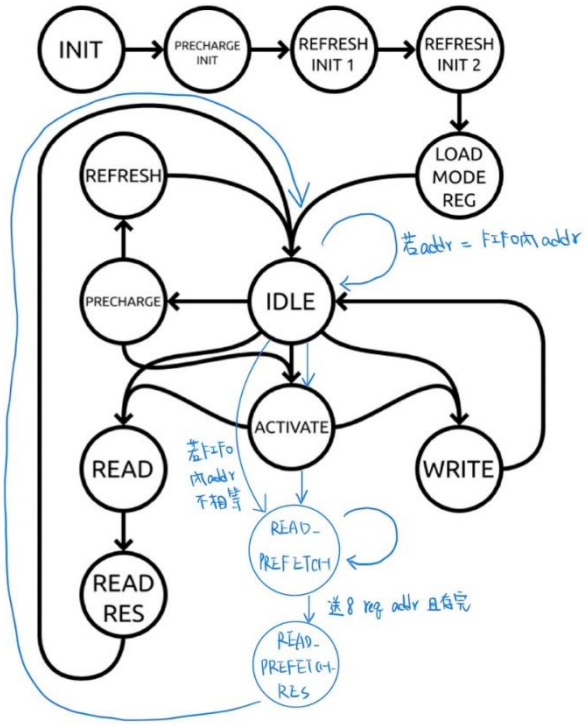


圖(1) instruction 讀取波型圖



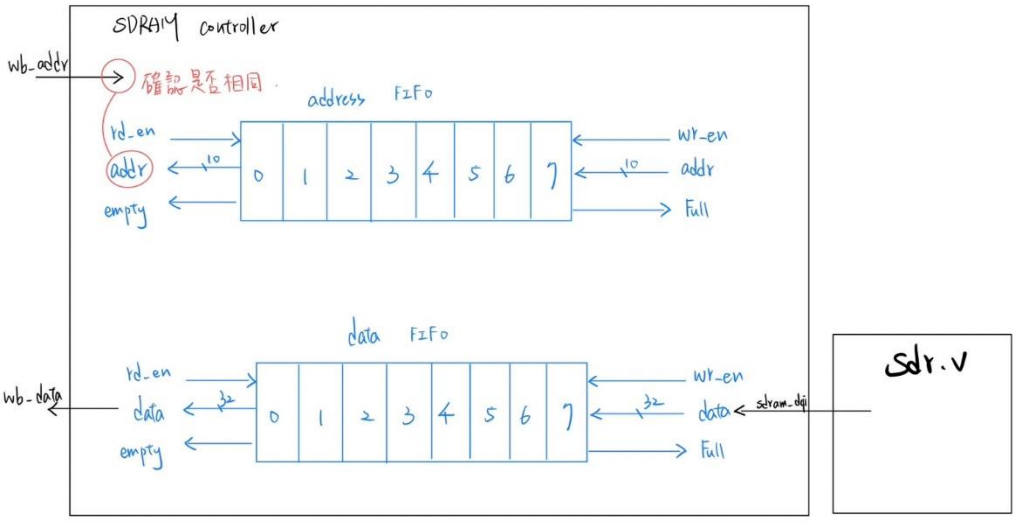
圖(2) instruction 讀取波型圖之兩個連續地址的 offset 不同

- SDRAM controller



圖(3) SDRAM controller 設計

- Prefetch buffer 設計

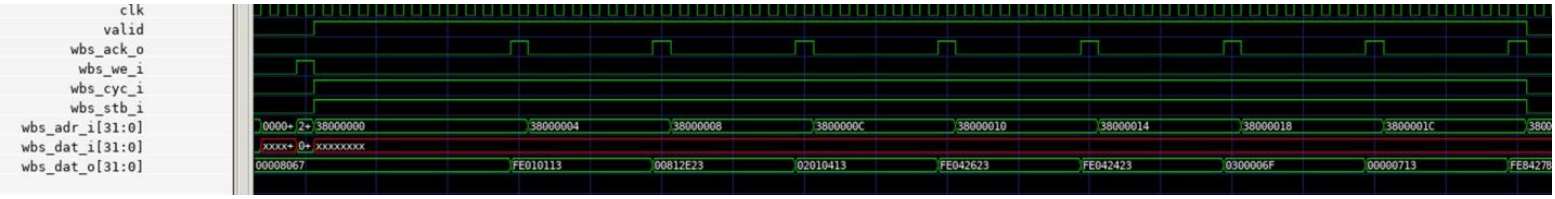


圖(4) prefetch buffer 設計圖

➢ Prefetch 優化後成果

(1) 沒有 prefetch: 如下圖(5)波型所示。

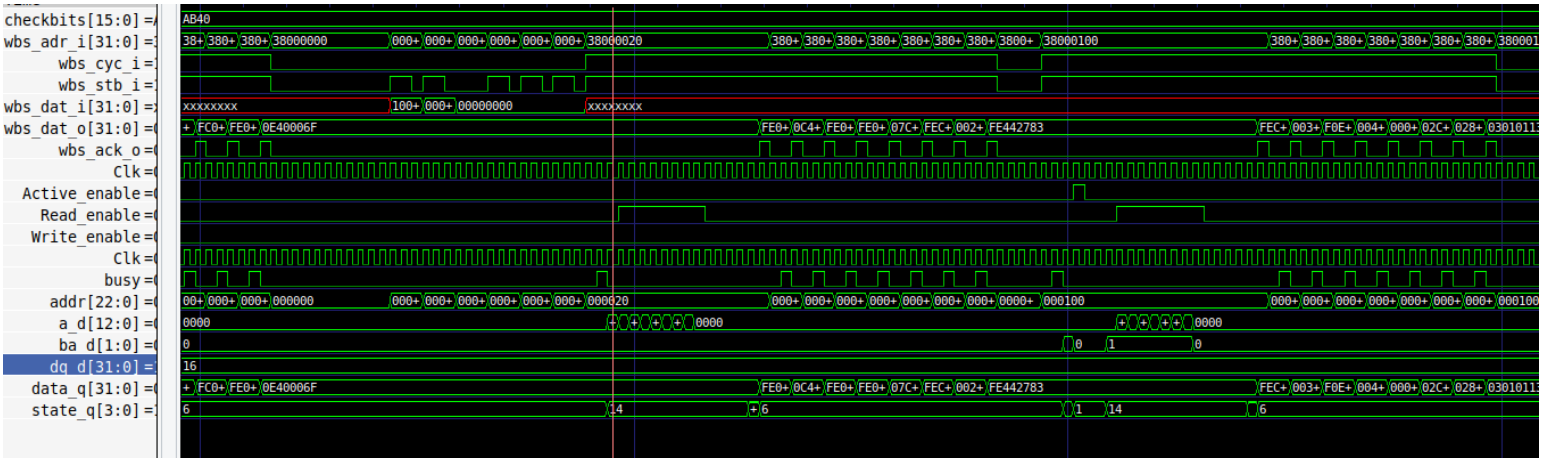
每 8 個 instruction 所需要 cycle 數	
要 Activate	76 cycle
不用 Active	72 cycle



圖(5) 沒有 prefetch READ 波型圖

(2) 有 prefetch: 如下圖(6)波型所示。8 個指令所需要的 cycle 數從 76/72 降到 50/46，共減少 26 cycle，平均一次 instruction 讀取減少 3.25 cycle。

每 8 個 instruction 所需要 cycle 數	
要 Activate	50 cycle
不用 Active	46 cycle



圖(6) 有 prefetch READ 波型圖

三、 Introduce the bank interleave for code and data

在平常的 SoC 運作過程中，都是先將 firmware 搬至 3800 位置後，再進到 flash 中去執行，由 cpu 端接收指令及數據來完成運算，並且將結果存放在 dff 中。在這次實驗中我們在 section.lds 中切分出了一個新的名為 alldata 的空間，其資料位置是接續著 firmware 放置位置 3800-0000 ~ 3800-0200 後的 3800-0200 ~ 3800-0600，這個 alldadta 做為資料群放的另一個 bank，在 SDRAM controller 中將被視為 bank[1]，並且取代之前 dff 存取被運算數據的功能，改為到 alldata 這個 bank 中拿資料來做矩陣運算。

四、 Introduce how to modify the linker to load address/data in two different bank

在 section.lds 中除了新增一個自訂的 bank 來存放資料，還需要修改其中的 linker 來讓資料進到正確的 bank，在 linker 中有幾項分類分別進到不同的記憶體空間，如 text、rodata 等就會進到 flash，而 cpu 運算過程中的資料則是被分類放到 dff 中，在 firmware 方面就如同之前會被放到 mprjram 的記憶體空間，而我們要進行運算的資料不同於以往直接在 linker 中放入 dff，我們修改存放位置讓他能夠在自訂的 bank 中被儲存並運算，如下圖(7)的修改方式，便能夠達到讓資料放到與 firmware 不同的 bank 中達到 bank interleave 的目的。

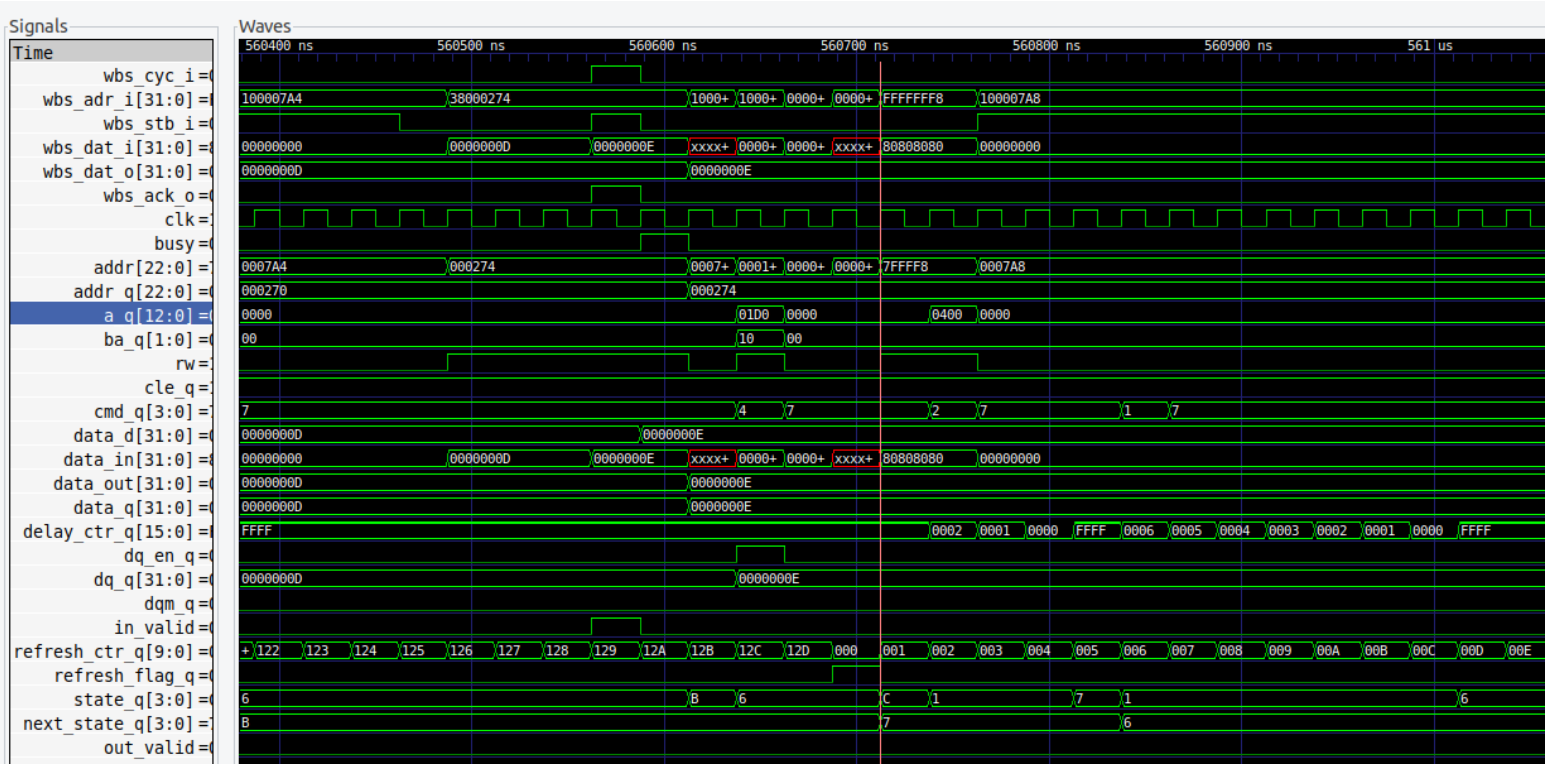
```
48      .data :
49      {
50          . = ALIGN(8);
51          _fdata = .;
52          *(.data .data.* .gnu.linkonce.d.*)
53          *(.data1)
54          _gp = ALIGN(16);
55          *(.sdata .sdata.* .gnu.linkonce.s.*)
56          . = ALIGN(8);
57          _edata = .;
58      } > alldata AT > flash
59
60      .bss :
61      {
62          . = ALIGN(8);
63          _fbss = .;
64          *(.dynsbss)
65          *(.sbss .sbss.* .gnu.linkonce.sb.*)
66          *(.scommon)
67          *(.dynbss)
68          *(.bss .bss.* .gnu.linkonce.b.*)
69          *(COMMON)
70          . = ALIGN(8);
71          _ebss = .;
72          _end = .;
73      } > dff AT > flash
74
75      .mprjram :
76      {
77          . = ALIGN(8);
78          _fsram = .;
79
80      } > mprjram AT > flash
81  }
```

圖(7) modify the linker

五、 Observe SDRAM access conflict with SDRAM refresh(reduce the refresh period)

將 refresh 週期從 750T 改短為 300T 後，能夠更明顯地觀察到當資料讀或寫到一半遇到 refresh 時，SDRAM controller 內部狀態所發生的變化與反應，在下圖(8)中是矩陣乘法的運算在寫入 data bank 時遇到 refresh 所產生的反應。

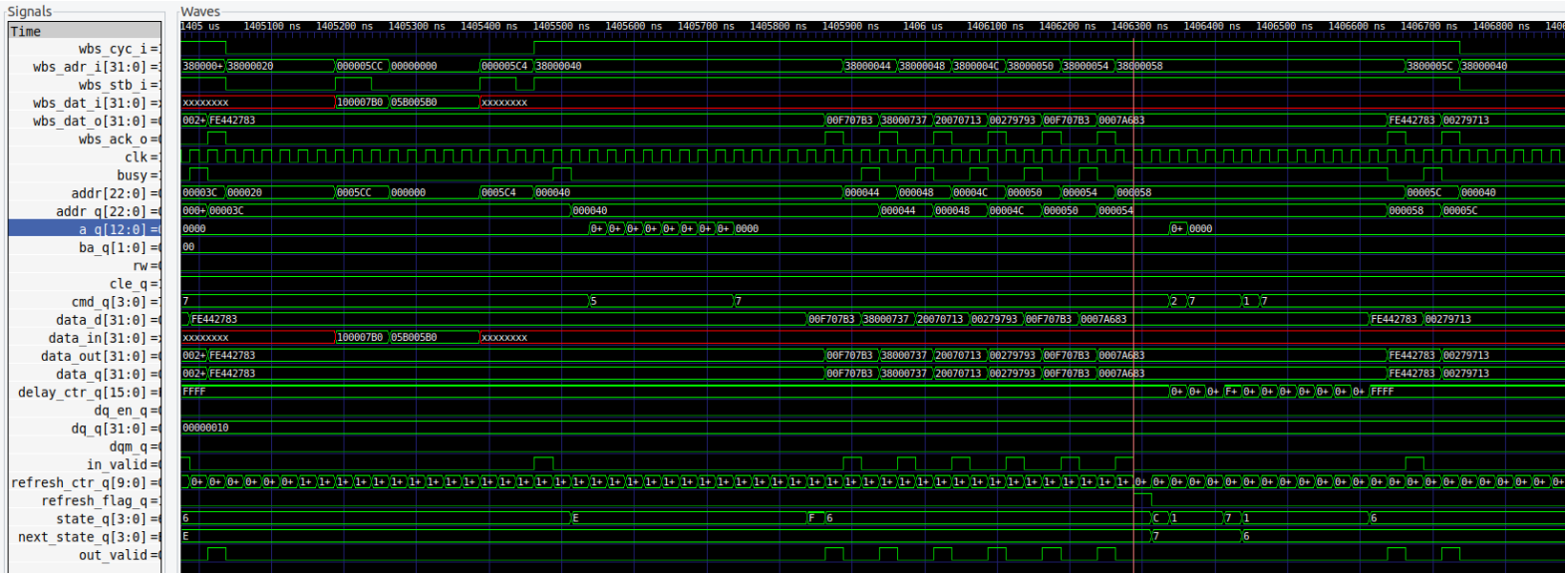
透過觀察 refresh_flag、state 以及 SDRAM controller 的 busy 狀態，我們能夠發現當 refresh_flag 升起時，next_state 將無條件決定為 state 7，在此同時必須將放置在 sense amp buffer 中的資料 precharge 回 SDRAM，而這個過程需要 wait 經過 3 個 cycle，隨後便會將 state 轉為 7，進入 refresh 狀態，再經過 6 個 cycle 後完成 refresh，state 便會回到 idle 等待下一次的寫入。



圖(8) 運算資料寫進 data bank 遇到 refresh

在圖(9)中為從 SDRAM 中讀取資料時被 refresh 打斷所發生的狀態，由於我們在 SDRAM 上做了地址的 prefetch，因此相比於前面寫入被打斷的情況更能夠觀察 refresh 回來後會如何繼續完成原本的工作。

在下圖中能看到我們在 idle 狀態時 prefetch 了八個地址，而 refresh 發生在第六個地址來後，與前面 write 時相同，當 refresh_flag 升起時，需要將目前 sense amp buffer 中的資料 precharge 回 SDRAM，接著才會進入 refresh state，值得注意的是，當 refresh 發生時 controller 的 busy 會一直是為 1 的狀態，在做完 refresh 回到 idle state 進行下一個地址的 prefetch 後，busy 才會由 1 變回 0，這意味著 refresh 完成後會繼續正常運行原本要做的事，但會延遲 2 個 cycle 才回覆 wbs_ack，以及耗費多個 cycle 在 precharge 資料，因此若能夠降低 refresh 與 SDRAM access 的衝突次數，也能夠增加 SDRAM 本身的性能。



圖(9) 從 data bank 讀取資料遇到 refresh