LAB-D Report

1. SDRAM controller design：

右圖是加上 prefetch buffer 後的 SDRAM controller Read 部分的 FSM 展開樣子，從原本的 Read request 一來就啟動 SDRAM，改成了當 cache 發生”miss”後才會啟動 SDRAM，而 cache hit 就只要將 data 從 cache register 中拿出即可，除此之外，我們還將 Read 更改成可以連續讀出資料模式(新增 WAIT_READ state 與修改 READ_RES state)，並根據不同的 prefetch cache 來選擇不同的連續讀出數量，以及不同的 address offset。

講解一下新增的 WAIT_READ 與 READ_RES 的作用：

WAIT_READ：

以 cache miss 為起點的 address，送 n 筆 read data command 來取得我們要的 prefetch 長度，並根據我們使用的 data 擺放方式不同，往後增加不同的 address offset，舉例來說：若是 row base access，offset 就是 +4，column base access 則有不同的 offset number，那這部分是用 counter 來控制 address 發送以及 data 的接收到不同的 cache register 位置中。

READ_RES：

用來接收剩下 read 出來的 data(因為 READ command 發送到回傳回 controller 相差了 2T)，那原本的 outvaild 控制我就把他搬出 SDRAM controller 的 FSM 中，因為現在與 wb signal control interface 是與 prefetch cache schemes 有關，而不是 SDRAM controller 了。

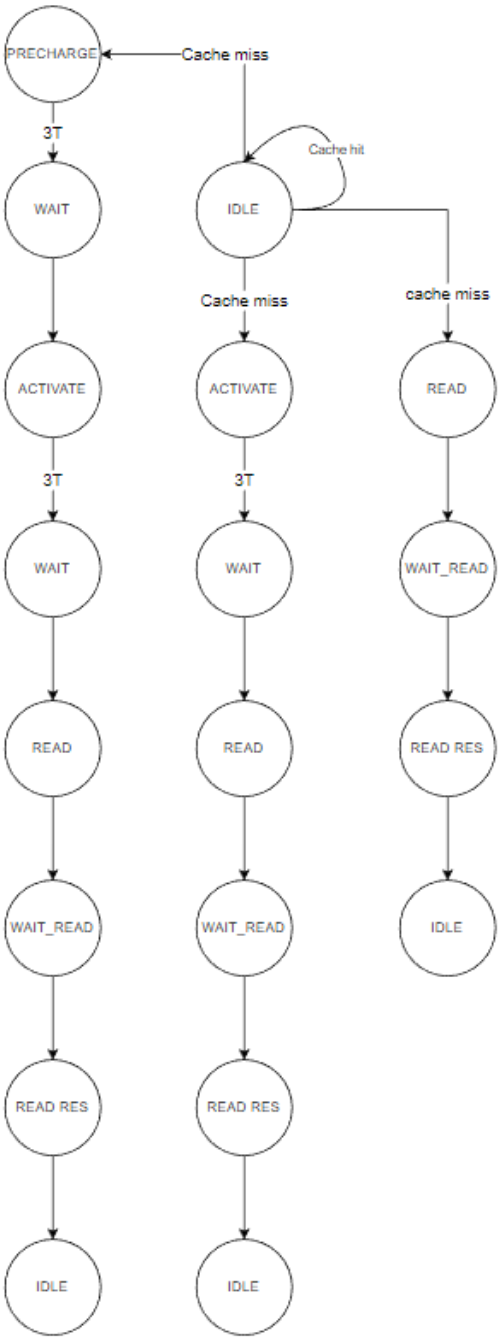
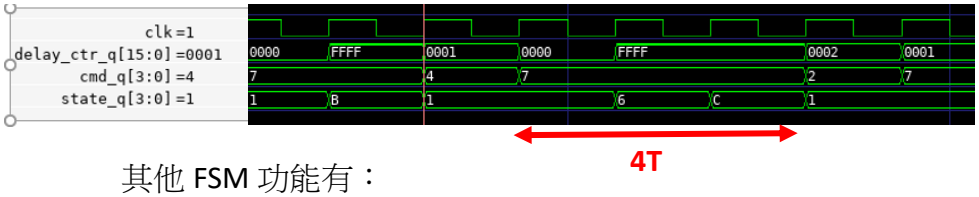


Figure 1 SDRAM READ FSM

Write STATE：由於我特別分配 linker address 的擺放方式，因此 cache 內的資料並不會出現 RAW 的情況，所以 WRITE 部分的 state 並不需要有 prefetch 那部分的更改，保留原狀即可，但還發現另一個問題是當使用連續 read 的方式，會造成 Write 的行為模式有小小的改變，像是發完 Write command 後要再經過 4T 才能給 REFRESH command(如下圖)，否則會存到不正確的數值，即使 data、address、command 等送給 SDRAM 都是正確的數值，因此我將 State machine 改成 IDLE -> WRITE -> WAIT -> IDLE，delay_ctr=1 才能正常存入。



其他 FSM 功能有：

INIT：初始時的 state、正常需要幾個步驟去啟動整個 SDRAM，而這裡為了化簡流程，就直接 delay 1T 就到 IDLE。

IDLE：當沒有任何 command 要執行時的 state，cmd 會傳給 SDRAM NOP 指令。

ACTIVATE：當需要啟動新的 row bank 位置的 state，此時的 sdram_a 是送 row address，cmd 會傳 activate。

WAIT：根據不同的指令，會需要不同的等待時間，才會完成該任務，裡面有個 decrease counter 來倒數所需的時間。

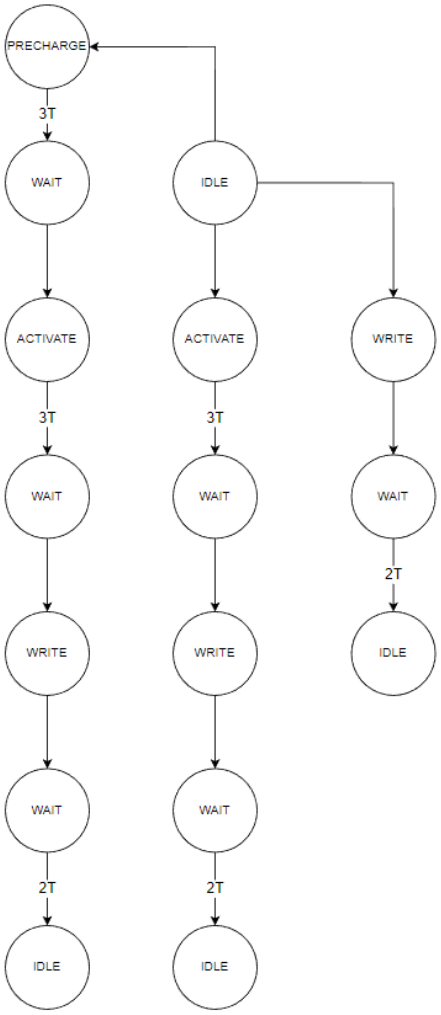


Figure 2 SDRAM WRITE FSM

PRECHARGE：當我們要更換不同的 row bank 時，需要將目前已更新好的 data 回存到 SDRAM 的對應 row 位置，才能將新的 row bank load 出來，此時 cmd 會傳 precharge。

REFRESH：SDRAM 是利用電容有儲存電荷的特性來儲存訊號，但是非理想狀態下電容會因為漏電流等因素，慢慢地減少電荷量，此時電壓就會下降，跌超過 1 的準位，因此需要定時對電容重新充電，此時 cmd 會傳 pefresh。

2. SDRAM bus protocol：

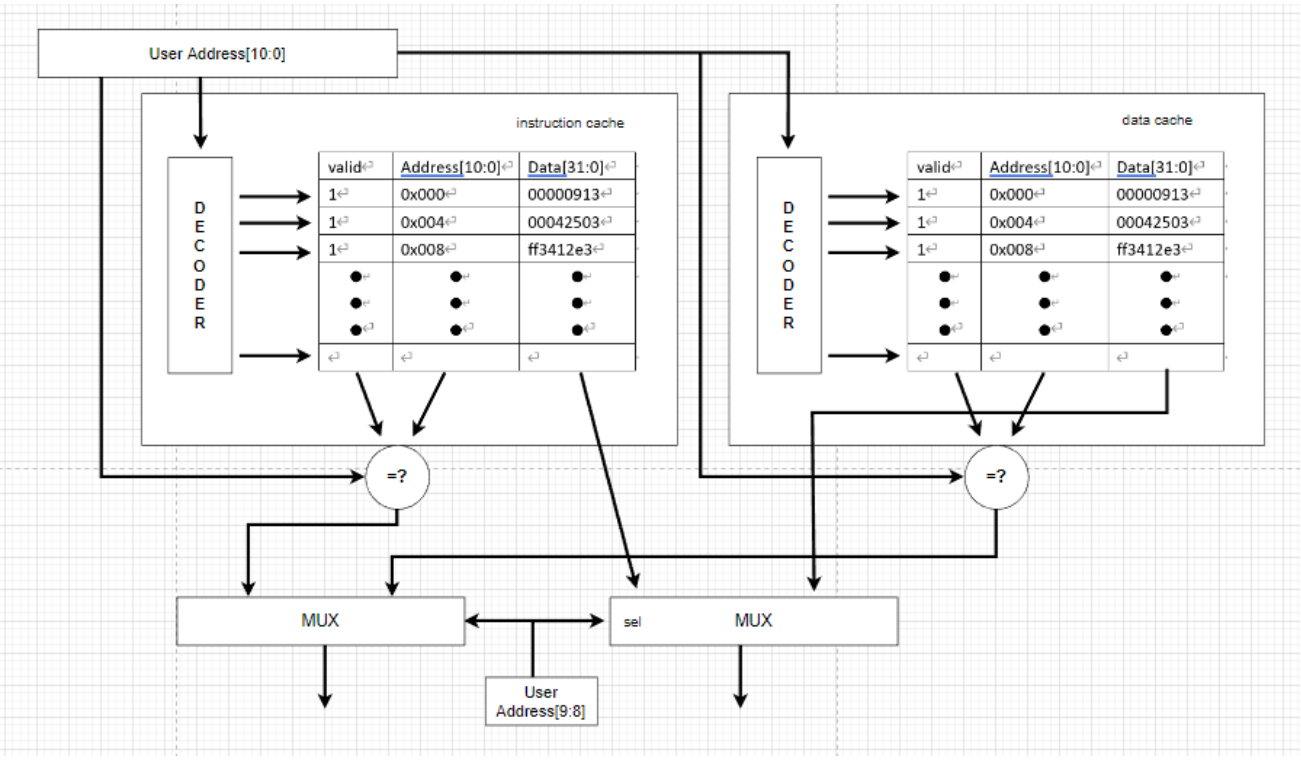
SDRAM Device IO：
sdram_cle: =1 表示 clock 有 enable, =0 則是不會在接收任何 command，但會保持 self-fresh、power-down 等操作。

sdram_cs、sdram_ras、sdram_cas、sdram_we：用四個 port 的組合來產生 READ、WRITE、ACTIVE、PRECHARGE 等…SDRAM command，詳細來說 sdram_cs 用來 enable 整個 SDRAM chip 運作，sdram_ras 與 sdram_cas 分別是指現在的指令要對 row 與 column 操作(LOW activate)，sdram_we 選擇要 Read (HIGH)/ Write(LOW)。
sdram_dqm：原本的 SDARM 的 data port 是只有一個 DQ 來輸入/輸出 data information，因此在它要切換時，需要一個遮罩來阻擋訊號去灌回到該 port 中(因為對於 port 來說 input/output 這兩種會是截然不同的結構與特性)，等轉換完成後才會讓 port 接收或發出 data，但助教提供的 SDRAM 有把輸入輸出的 port 分開來，所以我們就不需要 control 這 signal，那應用上(sdram_dqm=1 是把遮罩打開、sdram_dqm=0 則是關閉、讓訊號自由進出)。

sdram_ba[1:0]： bank address。
sdram_a[12:0]： 依據現在的 command 來表示成 column address / row address。
sdram_dqi[31:0]：從 SDRAM 讀出來的 data。
sdram_dqo[31:0]：要寫入 SDRAM 的 data。

3. Prefetch scheme：

我觀察了 firmware code execution waveform 會是 8 個連續的 instruction，重複二到三次(兩兩八筆之間的 address 並不連續)，在最後會讀一筆 A 與 B 的 data，以及 Write 一筆 Result 進去，透過上面特性，我設計的 instruction 與 data prefetch cache，那 instruction 一次 prefetch 8 筆連續資料，Data cache 則是一次取 4 筆資料，而由於矩陣乘法中，B 矩陣是以 column 的方向取值，因此 B 矩陣的 offset(+16)與其他 row major order(+4)不同，instruction cache size 為 16，也就是說可以暫存兩筆不連續 address 的 prefetch data，至於怎麼選擇要刷新哪一邊的 cache 是用一個簡單的 flag 去操作，也就是說，當我更新左邊的 cache location，我就將 flag 反向，下次就更新右邊的；而 data cache size 為 12，分成 A B Result 3 區，每區只存一次 prefect 大小(4 筆 Data)，由於我們矩陣大小為 4*4，所以每次 miss 就剛好是重抓一個點的矩陣相乘所需的資料量，這樣就能最有效利用 cache size 了。
下圖是 prefetch cache 的 block diagram，由於 firmware code 並不是很大，所以 address 的部分我只取前 11bits 存起來，並配合 decoder 將 cache 裡的 address and valid flag 拿出來比較，並根據 bank address[9:8]來決定輸出的資訊是要從 instruction cache / data cache 出來。



4. Introduce the bank interleave for code and data：

Code interleave：

我看 firmware execution code，並不會超過 0x1FF 的長度，我就想說比起只用 1 個 bank 去存 instruction，我可以用兩個 bank 來存放我的 firmware code，這樣的好處是，我就不需要更換 row address 時的 pre-charge state，因為我的 code 有可能會在這兩個 row 中交替讀取，那這樣就會造成而外的 execution cycle，因此用兩個 bank 去存是最好的解法了。

Data interleave：

我分成 A、B matrix 一個 bank，Result 一個 bank，主要是為了方便 prefetch cache 去分辨目前要索取的 data 是輸入還是輸出的變數，那我就依序將 A、B 矩陣從 0X200 開始存進去，Result 則是從 0x300 開始存入。

5. how to modify the linker：

.mprjram 會指向 firmware execution code 所存的位置中，.data 指向的會是 software code 中已經賦予數值的那些變數(以 matmul.h 為例就是 A 跟 B 矩陣)，而.bss 則是指向沒有給予初始值的變數—也就是 result 矩陣，那我們要將這些資料從原本的 dff address map 轉乘 user project ram 中，也就是 0x38000000 中，那我修改右圖紅框部分，左邊的 mprjram 代表的是 VMA (Virtual Memory Address)，當程式運作時，section 會得到這個記憶體位址，而右邊的 flash 則是 LMA (Load Memory Address)，當該區段被載入時，會被放到這個記憶體位址，那根據 memory mapping 得知 mprjram 就是對應到 0x38000000，另一方面是要將 instruction，mapping 到 0x000~0x1FF，A 與 B mapping 到 0x200~0x2FF，result mapping 到 0x300~0x3FF 中，因為 linker 是用一個 local counter，從 0 開始去一個個的將 address allocate 至該 address line 中，為此. mprjram 要先宣告，他就會從 0x000 開始放 instruction，並且我有用 complier optimizer -O1，將 execution code 限制在 0x1FF 中，接要將 A、B 放入 0x200 開頭，那我使用的是 ALIGN(offset) function(藍框部分)，他會將 local counter 從 original + offset 的地方開始計數(以.data 為例 起始點為 0x00、offset 為 0x200→從 0x200 開始計數；.bss 的起始點因前面.data 改成 0x200、offset 為 0x100 →從 0x300 開始計數)。

```
74      .mprjram :
75      {
76          . =  ALIGN(8); /*0x1B8 ;*/
77          _fsram = .;
78
79      } > mprjram AT > flash
80
81      .data ALIGN(0x200):
82      {
83          . = ALIGN(4);
84          _fdata = .;
85          *(.data)
86          *(.data.*)  /* .gnu.linkonce.d.*)*/
87          *(.sdata)
88          *(.sdata.*) /*.gnu.linkonce.s.*)*/
89          . = ALIGN(4);
90           edata = .;
91      } > mprjram AT > flash
92      .bss ALIGN(0x100):
93      {
94          . =  ALIGN(8); /*0x80  ;*/
95          _fbss = .;
96          *(.dynsbss)
97          *(.sbss)
98          *(.sbss.*) /*.gnu.linkonce.sb.*)*/
99          *(.scommon)
100         *(.dynbss)
101         *(.bss )
102         *(.bss.*) /*.gnu.linkonce.b.*)*/
103         *(COMMON)
104         . = ALIGN(8);
105         _ebss = .;
106          end = .;
107     } > mprjram AT > flash
```

```
11  MEMORY {
12      vexriscv_debug : ORIGIN = 0xf00f0000, LENGTH = 0x00000100
13      dff : ORIGIN = 0x00000000, LENGTH = 0x00000400
14      dff2 : ORIGIN = 0x00000400, LENGTH = 0x00000200
15      flash : ORIGIN = 0x10000000, LENGTH = 0x01000000
16      mprj : ORIGIN = 0x30000000, LENGTH = 0x00100000
17      mprjram : ORIGIN = 0x38000000, LENGTH = 0x00400000
18      hk : ORIGIN = 0x26000000, LENGTH = 0x00100000
19      csr : ORIGIN = 0xf0000000, LENGTH = 0x00010000
20  }
```

Firmware execution code address mapping(0x000 ~ 0x100)

```
38000000 <matmul>:
38000000:   fc010113          addi    sp,sp,-64
38000004:   02112e23          sw  ra,60(sp)
38000008:   02812c23          sw  s0,56(sp)
3800000c:   02912a23          sw  s1,52(sp)
38000010:   03212823          sw  s2,48(sp)
38000014:   03312623          sw  s3,44(sp)
38000018:   03412423          sw  s4,40(sp)
3800001c:   03512223          sw  s5,36(sp)
38000020:   03612023          sw  s6,32(sp)
38000024:   01712e23          sw  s7,28(sp)
38000028:   01812c23          sw  s8,24(sp)
3800002c:   01912a23          sw  s9,20(sp)
38000030:   01a12823          sw  s10,16(sp)
38000034:   01b12623          sw  s11,12(sp)
38000038:   38000bb7          lui s7,0x38000
3800003c:   200b8b93          addi    s7,s7,512 # 38000200 <A>
38000040:   010b8993          addi    s3,s7,16
38000044:   38000cb7          lui s9,0x38000
38000048:   304c8c93          addi    s9,s9,772 # 38000304 <result>
3800004c:   00000d13          li  s10,0
38000050:   38000db7          lui s11,0x38000
38000054:   200d8d93          addi    s11,s11,512 # 38000200 <A>
38000058:   00400c13          li  s8,4
3800005c:   040d8b13          addi    s6,s11,64
38000060:   000c8a93          mv  s5,s9
38000064:   00000a13          li  s4,0
38000068:   000b0493          mv  s1,s6
3800006c:   000b8413          mv  s0,s7
38000070:   00000913          li  s2,0
38000074:   0004a583          lw  a1,0(s1)
38000078:   00042503          lw  a0,0(s0)
3800007c:   d8000097          auipc   ra,0xd8000
38000080:   360080e7          jalr    864(ra) # 100003dc <__mulsi3>
38000084:   00a90933          add s2,s2,a0
38000088:   00440413          addi    s0,s0,4
3800008c:   01048493          addi    s1,s1,16
38000090:   ff3412e3          bne s0,s3,38000074 <matmul+0x74>
38000094:   012aa023          sw  s2,0(s5)
38000098:   001a0a13          addi    s4,s4,1
3800009c:   004a8a93          addi    s5,s5,4
380000a0:   004b0b13          addi    s6,s6,4
380000a4:   fd8a12e3          bne s4,s8,38000068 <matmul+0x68>
380000a8:   004d0d13          addi    s10,s10,4
380000ac:   010b8b93          addi    s7,s7,16
380000b0:   01098993          addi    s3,s3,16
380000b4:   010c8c93          addi    s9,s9,16
380000b8:   01000793          li  a5,16
380000bc:   fafd10e3          bne s10,a5,3800005c <matmul+0x5c>
380000c0:   38000537          lui a0,0x38000
380000c4:   30450513          addi    a0,a0,772 # 38000304 <result>
380000c8:   03c12083          lw  ra,60(sp)
380000cc:   03812403          lw  s0,56(sp)
380000d0:   03412483          lw  s1,52(sp)
380000d4:   03012903          lw  s2,48(sp)
380000d8:   02c12983          lw  s3,44(sp)
380000dc:   02812a03          lw  s4,40(sp)
380000e0:   02412a83          lw  s5,36(sp)
380000e4:   02012b03          lw  s6,32(sp)
380000e8:   01c12b83          lw  s7,28(sp)
380000ec:   01812c03          lw  s8,24(sp)
380000f0:   01412c83          lw  s9,20(sp)
380000f4:   01012d03          lw  s10,16(sp)
380000f8:   00c12d83          lw  s11,12(sp)
380000fc:   04010113          addi    sp,sp,64
38000100:   00008067          ret
```

Matrix A and B address mapping(0x200~0x27c)

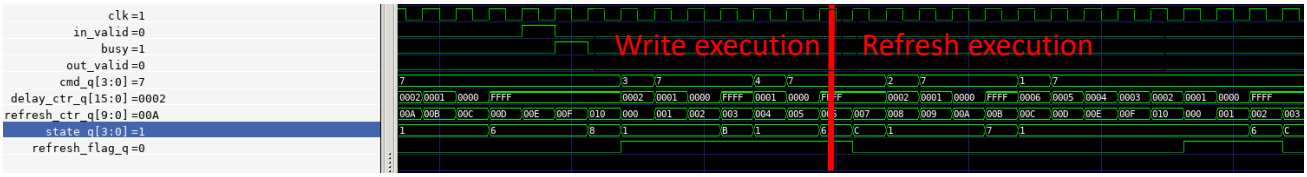
```
38000200 <A>:
38000200:   0000                .2byte  0x0
38000202:   0000                .2byte  0x0
38000204:   0001                .2byte  0x1
38000206:   0000                .2byte  0x0
38000208:   0002                .2byte  0x2
3800020a:   0000                .2byte  0x0
3800020c:   00000003            lb  zero,0(zero) # 0 <__DYNAMIC>
38000210:   0000                .2byte  0x0
38000212:   0000                .2byte  0x0
38000214:   0001                .2byte  0x1
38000216:   0000                .2byte  0x0
38000218:   0002                .2byte  0x2
3800021a:   0000                .2byte  0x0
3800021c:   00000003            lb  zero,0(zero) # 0 <__DYNAMIC>
38000220:   0000                .2byte  0x0
38000222:   0000                .2byte  0x0
38000224:   0001                .2byte  0x1
38000226:   0000                .2byte  0x0
38000228:   0002                .2byte  0x2
3800022a:   0000                .2byte  0x0
3800022c:   00000003            lb  zero,0(zero) # 0 <__DYNAMIC>
38000230:   0000                .2byte  0x0
38000232:   0000                .2byte  0x0
38000234:   0001                .2byte  0x1
38000236:   0000                .2byte  0x0
38000238:   0002                .2byte  0x2
3800023a:   0000                .2byte  0x0
3800023c:   00000003            lb  zero,0(zero) # 0 <__DYNAMIC>
38000240 <B>:
38000240:   0001                .2byte  0x1
38000242:   0000                .2byte  0x0
38000244:   0002                .2byte  0x2
38000246:   0000                .2byte  0x0
38000248:   00000003            lb  zero,0(zero) # 0 <__DYNAMIC>
3800024c:   0004                .2byte  0x4
3800024e:   0000                .2byte  0x0
38000250:   0005                .2byte  0x5
38000252:   0000                .2byte  0x0
38000254:   0006                .2byte  0x6
38000256:   0000                .2byte  0x0
38000258:   00000007            .4byte  0x7
3800025c:   0008                .2byte  0x8
3800025e:   0000                .2byte  0x0
38000260:   0009                .2byte  0x9
38000262:   0000                .2byte  0x0
38000264:   000a                .2byte  0xa
38000266:   0000                .2byte  0x0
38000268:   0000000b            .4byte  0xb
3800026c:   000c                .2byte  0xc
3800026e:   0000                .2byte  0x0
38000270:   000d                .2byte  0xd
38000272:   0000                .2byte  0x0
38000274:   000e                .2byte  0xe
38000276:   0000                .2byte  0x0
38000278:   0000000f            fence   unknown,unknown
3800027c:   0010                .2byte  0x10
```

Result addres mapping (0x304~0x344)

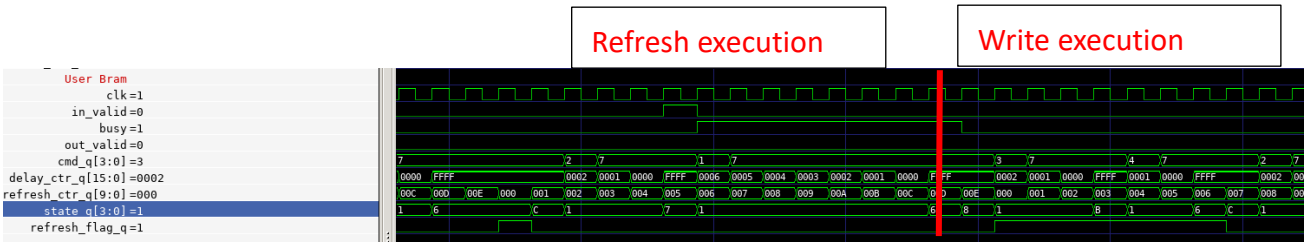
```
38000300 <flag>:
38000300:   0000                .2byte  0x0
    ...

38000304 <result>:
    ...

Disassembly of section .riscv.attributes:
```

6. SDRAM access conflicts with SDRAM refresh：

有兩種 case： 有 command 還在執行當中，SDRAM 的 refresh flag 拉起，這情況會等待 command 執行完成後，才會接著做 refresh 的 job，如下圖，Controller 先收到 Write command，因此它會先執行 Write 後才會接著做 Refresh command，而 refresh counter 並不會等 refresh 後才開始下一次 Refresh 計數，而是不停的計數。



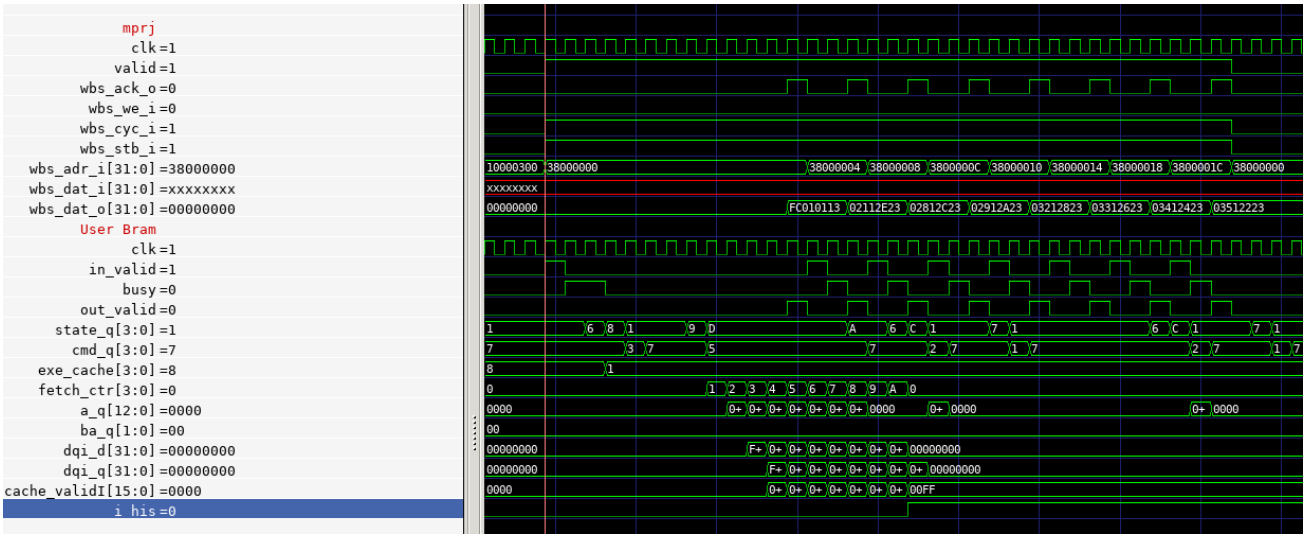
另一個則是正在執行 SDRAM refresh command 時，有新的 command 進來，那這樣就有點像是 controller 正在執行其他 command，然後把新進來的 command 暫存起來，並且 busy 拉 High，表示目前已經沒辦法再接收新的指令了。



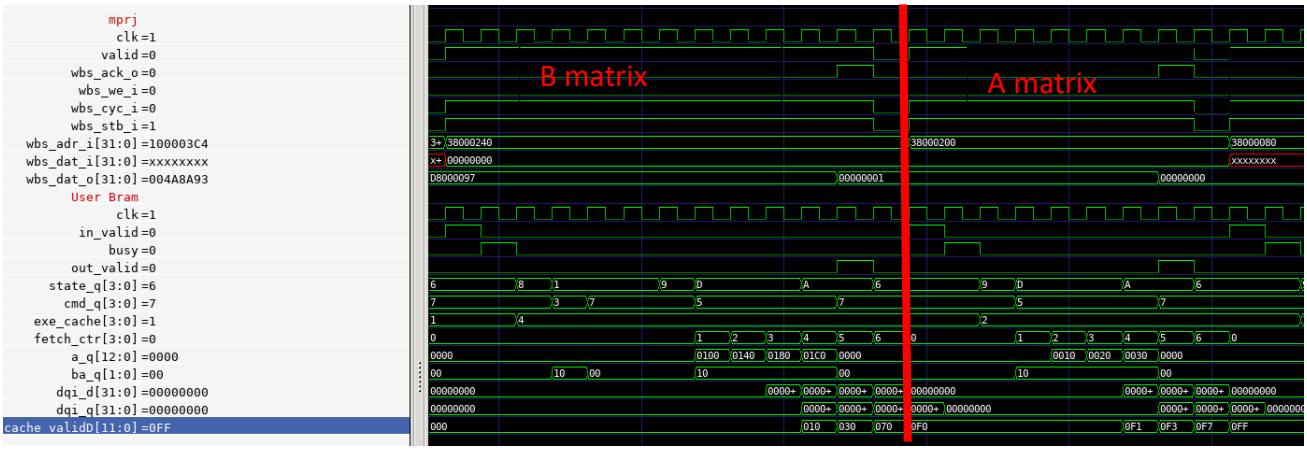
總括來說，遇到 SDRAM refresh 並不會影響到我們正在執行的指令，但是太常 refresh 會讓 controller 頻繁的跑到 pre-charge and refresh state，拖延到 WB 對 SDRAM 讀寫的時間，並且跑完整個 refresh 的流程需要花費 12T，因此 Refresh cycle 不能低於 13，否則整個 SDRAM controller 都會一直在做 refresh。

7. Waveform analysis：

下圖為 instruction 的 READ 的 waveform，可以看到每次都是向 SDRAM controller 要連續 8 筆 data，根據 prefetch schemes，instruction 會在第一個 Read 發生 cache miss，然後向 SDRAM 連續要 8 個 read command(a_q 在 fetch_ctr 計數 1~8)，而讀出來的 data 會在 2T 後出現，此時就會依序存進 cache 中，並將對應的 cache_validI 拉 H，而且 out_valid 只要發現 cache 裡有 hit 到它要的 data 就會馬上拉 H 出去(儘管 SDRAM controller 還在收其它 prefetch data)，因此將 out_valid 與 SDRAM controller 的 FSM 之間獨立分開，是能有效加大 interface 的 throughput。



下圖為 data 的 READ 的 waveform，由於我將 out_valid 與 SDRAM controller 的 FSM 之間獨立分開，所以在 cache miss 時，所需要的 cycle 數就會是與原本沒有 prefetch schemes 的一樣，由於矩陣為 4*4 大小，各取 4 個 element 來相乘，所以我 prefetch 4 個 data 來存入我的 cache 中，除此之外矩陣 A*B 的 B 矩陣會使用 column 方向的 data，也就是說 prefetch address 的 offset 要與其他不同，才算 prefetch 成功。



下圖是 WB 下次對 A B 矩陣發出 read request，因為是 cache hit，SDRAM controller 的 FSM 並沒有啟動，資料直接從 cache 中拿出來，in_valid 在經過 1T 後 out_valid 拉 H。