SOC Final Project Report

第 16 組

Members :
111061605 周聖平
111061621 蔡以心
111061631 張煒倫

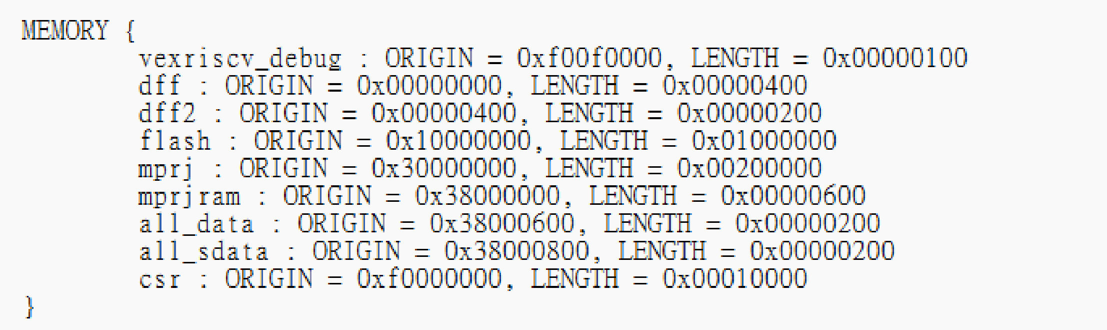
1. Data Replacement and Bank Interleaving

我們的 instruction code 是放在 mprjam，可以看到初始位址是 0x3800_0000，
length 為 600，因此放置的位址為 0x3800_0000 ~ 0x3800_05FF
Data 是放在 all_data，初始位址是 0x3800_0600，
length 為 200，因此放置的位址為 0x3800_0600 ~ 0x3800_07FF
然後由於我們 bank 判斷的 address 是在第 8 跟第 9 個 bit，所以 Data 的 bank 只會有 10、11，但 Instruction 的 bank 會是 00 ~ 11 都有

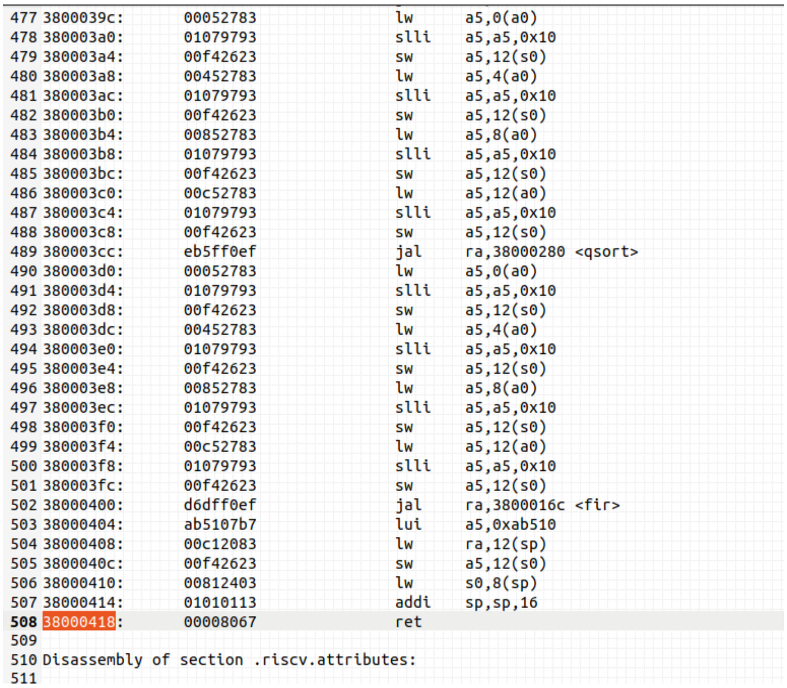
bank 的分配如下 ：

Data : 2、3

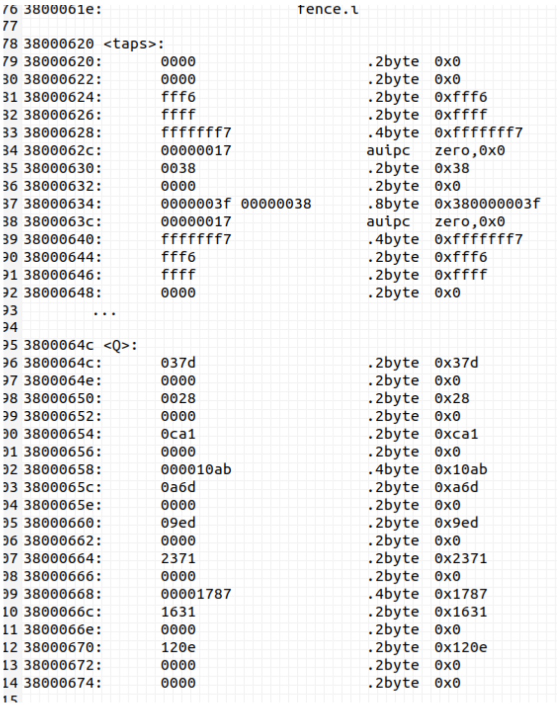
Instruction : 0 ~ 3



```
MEMORY {
        vexriscv_debug : ORIGIN = 0xf00f0000, LENGTH = 0x00000100
        dff : ORIGIN = 0x00000000, LENGTH = 0x00000400
        dff2 : ORIGIN = 0x00000400, LENGTH = 0x00000200
        flash : ORIGIN = 0x10000000, LENGTH = 0x01000000
        mprj : ORIGIN = 0x30000000, LENGTH = 0x00200000
        mprjram : ORIGIN = 0x38000000, LENGTH = 0x00000600
        all_data : ORIGIN = 0x38000600, LENGTH = 0x00000200
        all_sdata : ORIGIN = 0x38000800, LENGTH = 0x00000200
        csr : ORIGIN = 0xf0000000, LENGTH = 0x00010000
}
```



```
3800039c:	00052783          	lw	a5,0(a0)
380003a0:	01079793          	slli	a5,a5,0x10
380003a4:	00f42623          	sw	a5,12(s0)
380003a8:	00452783          	lw	a5,4(a0)
380003ac:	01079793          	slli	a5,a5,0x10
380003b0:	00f42623          	sw	a5,12(s0)
380003b4:	00852783          	lw	a5,8(a0)
380003b8:	01079793          	slli	a5,a5,0x10
380003bc:	00f42623          	sw	a5,12(s0)
380003c0:	00c52783          	lw	a5,12(a0)
380003c4:	01079793          	slli	a5,a5,0x10
380003c8:	00f42623          	sw	a5,12(s0)
380003cc:	eb5ff0ef          	jal	ra,38000280 <qsort>
380003d0:	00052783          	lw	a5,0(a0)
380003d4:	01079793          	slli	a5,a5,0x10
380003d8:	00f42623          	sw	a5,12(s0)
380003dc:	00452783          	lw	a5,4(a0)
380003e0:	01079793          	slli	a5,a5,0x10
380003e4:	00f42623          	sw	a5,12(s0)
380003e8:	00852783          	lw	a5,8(a0)
380003ec:	01079793          	slli	a5,a5,0x10
380003f0:	00f42623          	sw	a5,12(s0)
380003f4:	00c52783          	lw	a5,12(a0)
380003f8:	01079793          	slli	a5,a5,0x10
380003fc:	00f42623          	sw	a5,12(s0)
38000400:	d6dff0ef          	jal	ra,3800016c <fir>
38000404:	ab5107b7          	lui	a5,0xab510
38000408:	00c12083          	lw	ra,12(sp)
3800040c:	00f42623          	sw	a5,12(s0)
38000410:	00812403          	lw	s0,8(sp)
38000414:	01010113          	addi	sp,sp,16
38000418:	00008067          	ret

Disassembly of section .riscv.attributes:
```

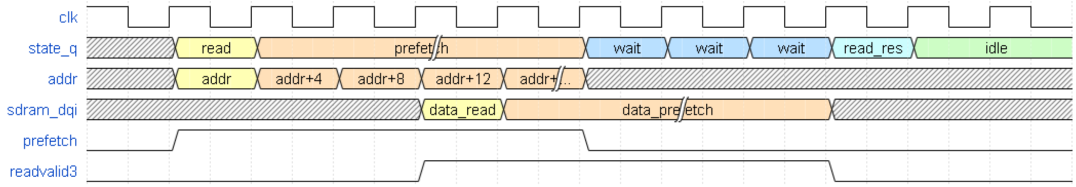


```
3800061e:	                  	fence.i

38000620 <taps>:
38000620:	0000                	.2byte	0x0
38000622:	0000                	.2byte	0x0
38000624:	fff6                	.2byte	0xfff6
38000626:	ffff                	.2byte	0xffff
38000628:	fffffff7            	.4byte	0xfffffff7
3800062c:	00000017          	auipc	zero,0x0
38000630:	0038                	.2byte	0x38
38000632:	0000                	.2byte	0x0
38000634:	0000003f 00000038 	.8byte	0x380000003f
3800063c:	00000017          	auipc	zero,0x0
38000640:	fffffff7            	.4byte	0xfffffff7
38000644:	fff6                	.2byte	0xfff6
38000646:	ffff                	.2byte	0xffff
38000648:	0000                	.2byte	0x0
	...

3800064c <Q>:
3800064c:	037d                	.2byte	0x37d
3800064e:	0000                	.2byte	0x0
38000650:	0028                	.2byte	0x28
38000652:	0000                	.2byte	0x0
38000654:	0ca1                	.2byte	0xca1
38000656:	0000                	.2byte	0x0
38000658:	000010ab            	.4byte	0x10ab
3800065c:	0a6d                	.2byte	0xa6d
3800065e:	0000                	.2byte	0x0
38000660:	09ed                	.2byte	0x9ed
38000662:	0000                	.2byte	0x0
38000664:	2371                	.2byte	0x2371
38000666:	0000                	.2byte	0x0
38000668:	00001787            	.4byte	0x1787
3800066c:	1631                	.2byte	0x1631
3800066e:	0000                	.2byte	0x0
38000670:	120e                	.2byte	0x120e
38000672:	0000                	.2byte	0x0
38000674:	0000                	.2byte	0x0
```

Instruction Address

Data Address

2. Burst



傳送位址:

在 read 和 wait 增加一個新的狀態 prefetch，狀態 prefetch 會持續送增加 4 的位址進入 SDRAM，狀態結束後一樣跳到狀態 wait，而紀錄位址的動作會在狀態 read 執行，將 prefetch 的位址存入暫存器 prefetch_addr。

接收資料:

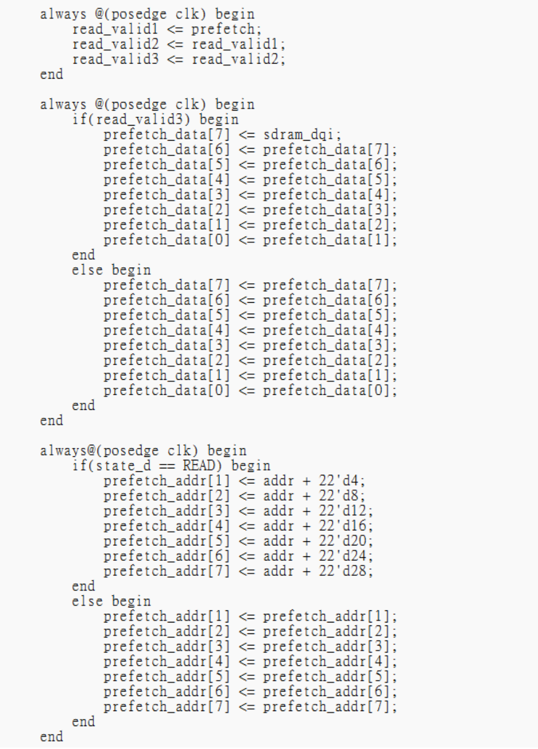
在狀態 read、prefetch 時，會將 reg prefetch 拉高為 1，進入狀態 wait 時，將 reg prefetch 降低為 0，並將 reg prefetch、readvalid1、 readvalid2、readvalid3 組成移位暫存器。使用 1+ prefetch 次數個移位暫存器接收 SDRAM 回傳的資料，將 readvalid3 做為控制訊號，當 readvalid3=1，會進行位移，
當 readvalid3=0，會保持不變。

資料回傳與預取:

當狀態為 read_res 時，會回傳移位暫存器的最後一個暫存器的值，也就是一開始位址為 addr 的值 data_read，接著進入狀態 idle，如果狀態不是 write 則會進入判斷 addr 是否存在於暫存器 prefetch_addr 內，並未在存在於暫存器 prefetch_addr 內則會重新進入狀態 read，再重新進行 read 跟 prefetch。

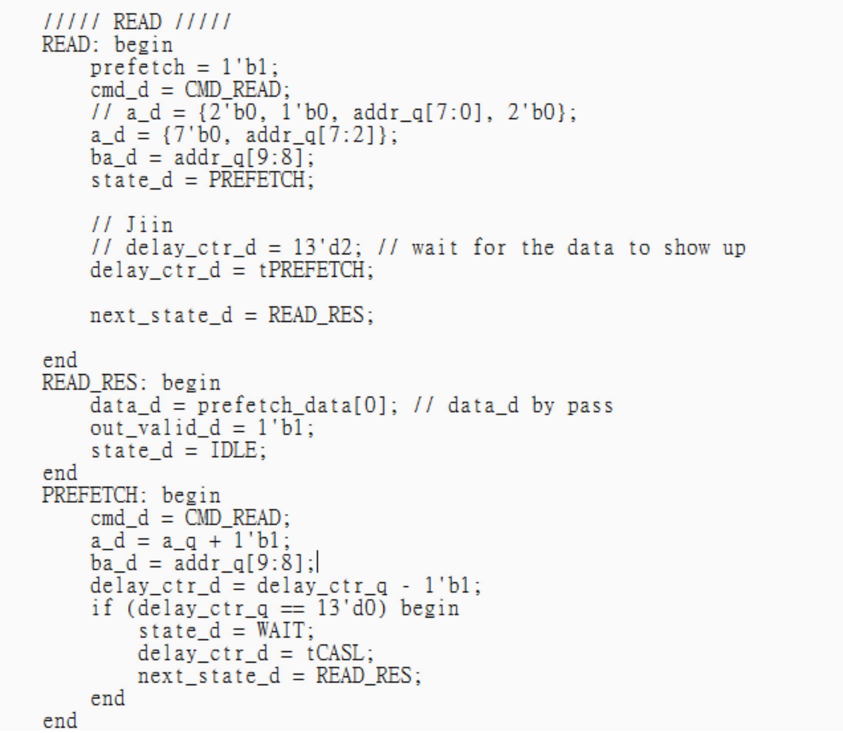
但原先我們以為我們優化的部分是 Prefetch 的功能，但後來經老師指正後瞭解我們的功能比較偏向於 Burst。

prefetch、read_valid1、read_valid2、read_valid3 組成位移暫存器



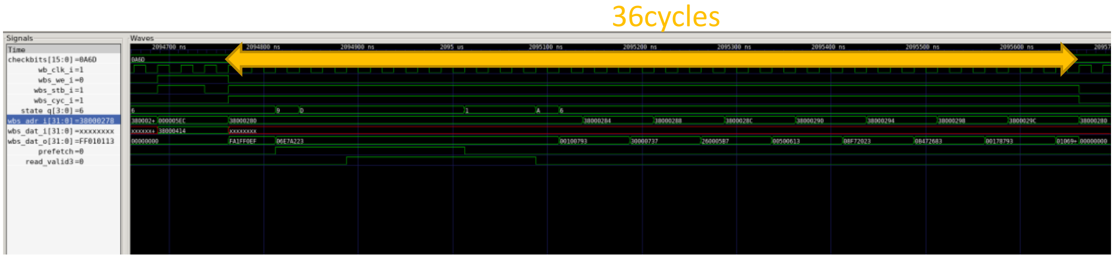
```
always @(posedge clk) begin
    read_valid1 <= prefetch;
    read_valid2 <= read_valid1;
    read_valid3 <= read_valid2;
end

always @(posedge clk) begin
    if(read_valid3) begin
        prefetch_data[7] <= sdram_dqi;
        prefetch_data[6] <= prefetch_data[7];
        prefetch_data[5] <= prefetch_data[6];
        prefetch_data[4] <= prefetch_data[5];
        prefetch_data[3] <= prefetch_data[4];
        prefetch_data[2] <= prefetch_data[3];
        prefetch_data[1] <= prefetch_data[2];
        prefetch_data[0] <= prefetch_data[1];
    end
    else begin
        prefetch_data[7] <= prefetch_data[7];
        prefetch_data[6] <= prefetch_data[6];
        prefetch_data[5] <= prefetch_data[5];
        prefetch_data[4] <= prefetch_data[4];
        prefetch_data[3] <= prefetch_data[3];
        prefetch_data[2] <= prefetch_data[2];
        prefetch_data[1] <= prefetch_data[1];
        prefetch_data[0] <= prefetch_data[0];
    end
end

always@(posedge clk) begin
    if(state_d == READ) begin
        prefetch_addr[1] <= addr + 22'd4;
        prefetch_addr[2] <= addr + 22'd8;
        prefetch_addr[3] <= addr + 22'd12;
        prefetch_addr[4] <= addr + 22'd16;
        prefetch_addr[5] <= addr + 22'd20;
        prefetch_addr[6] <= addr + 22'd24;
        prefetch_addr[7] <= addr + 22'd28;
    end
    else begin
        prefetch_addr[1] <= prefetch_addr[1];
        prefetch_addr[2] <= prefetch_addr[2];
        prefetch_addr[3] <= prefetch_addr[3];
        prefetch_addr[4] <= prefetch_addr[4];
        prefetch_addr[5] <= prefetch_addr[5];
        prefetch_addr[6] <= prefetch_addr[6];
        prefetch_addr[7] <= prefetch_addr[7];
    end
end
```

Code



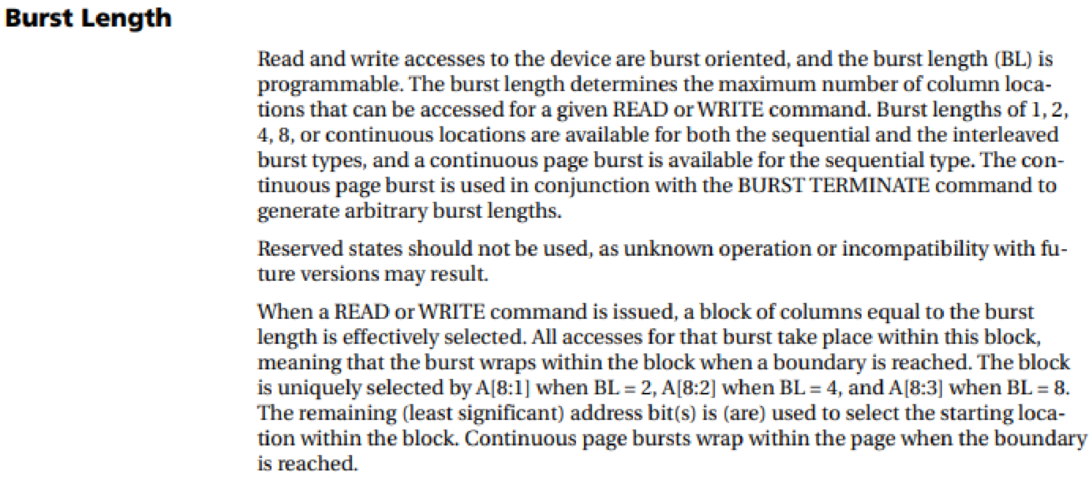
```
///// READ /////
READ: begin
    prefetch = 1'b1;
    cmd_d = CMD_READ;
    // a_d = {2'b0, 1'b0, addr_q[7:0], 2'b0};
    a_d = {7'b0, addr_q[7:2]};
    ba_d = addr_q[9:8];
    state_d = PREFETCH;

    // Jiin
    // delay_ctr_d = 13'd2; // wait for the data to show up
    delay_ctr_d = tPREFETCH;

    next_state_d = READ_RES;

end
READ_RES: begin
    data_d = prefetch_data[0]; // data_d by pass
    out_valid_d = 1'b1;
    state_d = IDLE;
end
PREFETCH: begin
    cmd_d = CMD_READ;
    a_d = a_q + 1'b1;
    ba_d = addr_q[9:8];
    delay_ctr_d = delay_ctr_q - 1'b1;
    if (delay_ctr_q == 13'd0) begin
        state_d = WAIT;
        delay_ctr_d = tCASL;
        next_state_d = READ_RES;
    end
end
```

Waveform



可以觀察到經過 burst 後，原本 read 8 個 instruction 需要 80 個 cycles，現在只需要 36 個 cycles。

參考 github lab-sdram 中的 256Mb_sdr.pdf，對於 Burst Length 的說明，



Burst Length

Read and write accesses to the device are burst oriented, and the burst length (BL) is programmable. The burst length determines the maximum number of column locations that can be accessed for a given READ or WRITE command. Burst lengths of 1, 2, 4, 8, or continuous locations are available for both the sequential and the interleaved burst types, and a continuous page burst is available for the sequential type. The continuous page burst is used in conjunction with the BURST TERMINATE command to generate arbitrary burst lengths.

Reserved states should not be used, as unknown operation or incompatibility with future versions may result.

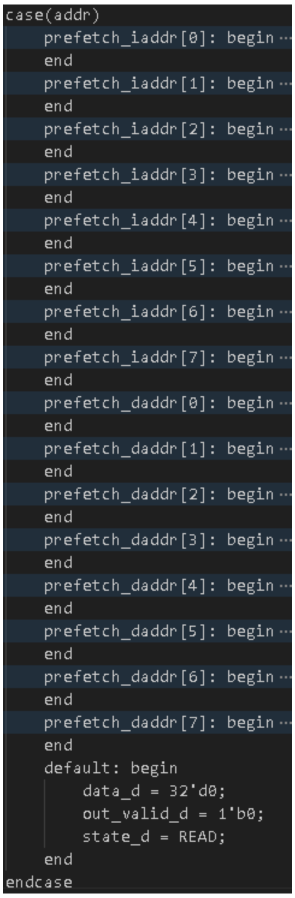
When a READ or WRITE command is issued, a block of columns equal to the burst length is effectively selected. All accesses for that burst take place within this block, meaning that the burst wraps within the block when a boundary is reached. The block is uniquely selected by A[8:1] when BL = 2, A[8:2] when BL = 4, and A[8:3] when BL = 8. The remaining (least significant) address bit(s) is (are) used to select the starting location within the block. Continuous page bursts wrap within the page when the boundary is reached.

所以我們在選擇 prefetch buffer 的大小有 1、3、7，共 3 種選擇。
以下將針對 prefetch buffer 大小與執行時間、讀取次數進行測試:

prefetch buffer size	endtime	read times
0(no prefetch)	2381588	3666
1	2209563	1927
3	2130763	1058
7	2108988	631

可以看到使用 prefetch buffer 可以減少執行時間，從原本的 2381588 減少到最佳的 2108988，降低了 11%，而讀取次數從 3666 減少最佳的 631，只剩原本的 17%。

由於我們觀看波形時發現在讀取 data 時，cpu 讀取的位址比較不連續，所以我們打算新增一個 prefetch buffer 給 data 使用。因為原本的設計，每次讀取都會把整個 buffer 替換掉，然而，將 data buffer 跟 instruction buffer 分開，每次讀取只會把其中一個 buffer 替換掉，在狀態 idle 判斷讀取位址是否位於 buffer 內，應該有更高的機率會命中，因為有更多選擇而且不會額外花費更多時間，還有更高的彈性。

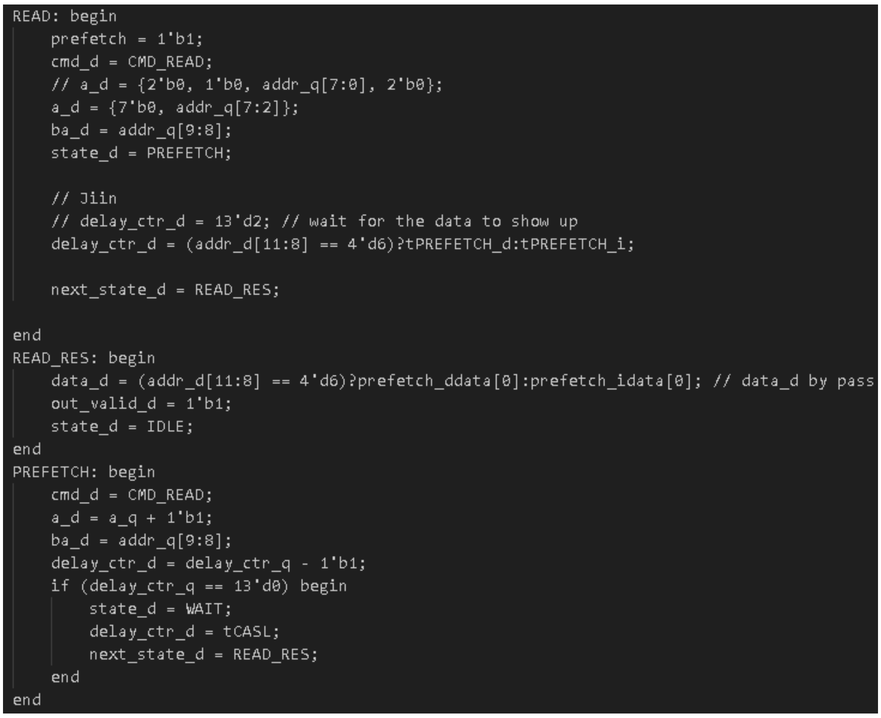


```
case(addr)
    prefetch_iaddr[0]: begin ···
    end
    prefetch_iaddr[1]: begin ···
    end
    prefetch_iaddr[2]: begin ···
    end
    prefetch_iaddr[3]: begin ···
    end
    prefetch_iaddr[4]: begin ···
    end
    prefetch_iaddr[5]: begin ···
    end
    prefetch_iaddr[6]: begin ···
    end
    prefetch_iaddr[7]: begin ···
    end
    prefetch_daddr[0]: begin ···
    end
    prefetch_daddr[1]: begin ···
    end
    prefetch_daddr[2]: begin ···
    end
    prefetch_daddr[3]: begin ···
    end
    prefetch_daddr[4]: begin ···
    end
    prefetch_daddr[5]: begin ···
    end
    prefetch_daddr[6]: begin ···
    end
    prefetch_daddr[7]: begin ···
    end
    default: begin
        data_d = 32'd0;
        out_valid_d = 1'b0;
        state_d = READ;
    end
endcase
```

接下來將介紹如何修改 sdram_controller:

狀態機:

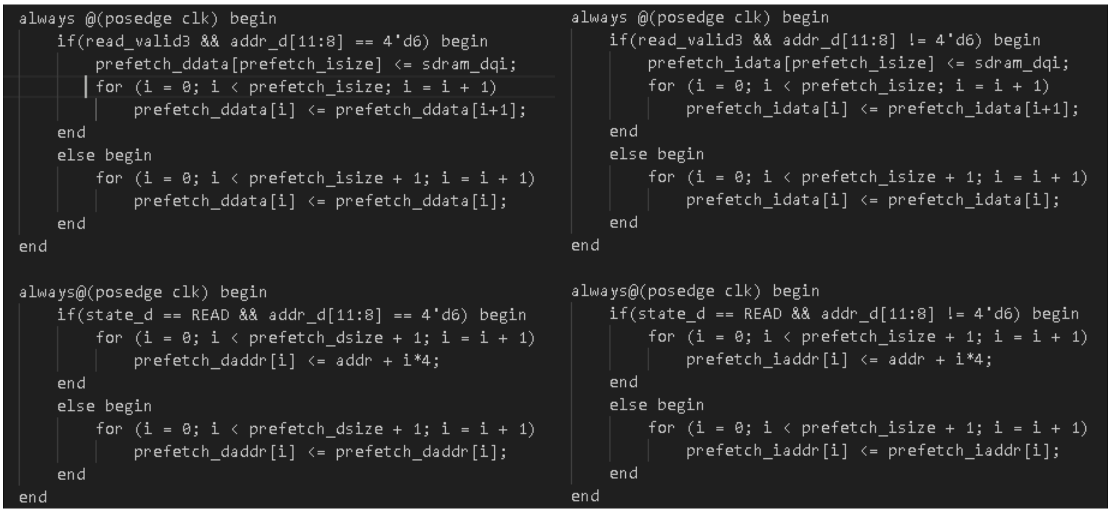
在狀態 READ，我們將 delay_ctr_d 加入位址判斷，根據讀取的位址判斷，下一個狀態 PREFETCH，將保持多少個 cycle。由於有 data buffer 跟 instruction buffer，在狀態 READ_RES 需要修改 data_d，根據讀取的位址選擇讀取 data buffer 還是 instruction buffer，至於狀態 PREFETCH 不需要修改。



```
READ: begin
    prefetch = 1'b1;
    cmd_d = CMD_READ;
    // a_d = {2'b0, 1'b0, addr_q[7:0], 2'b0};
    a_d = {7'b0, addr_q[7:2]};
    ba_d = addr_q[9:8];
    state_d = PREFETCH;

    // Jiin
    // delay_ctr_d = 13'd2; // wait for the data to show up
    delay_ctr_d = (addr_d[11:8] == 4'd6)?tPREFETCH_d:tPREFETCH_i;

    next_state_d = READ_RES;

end
READ_RES: begin
    data_d = (addr_d[11:8] == 4'd6)?prefetch_ddata[0]:prefetch_idata[0]; // data_d by pass
    out_valid_d = 1'b1;
    state_d = IDLE;
end
PREFETCH: begin
    cmd_d = CMD_READ;
    a_d = a_q + 1'b1;
    ba_d = addr_q[9:8];
    delay_ctr_d = delay_ctr_q - 1'b1;
    if (delay_ctr_q == 13'd0) begin
        state_d = WAIT;
        delay_ctr_d = tCASL;
        next_state_d = READ_RES;
    end
end
```

資料暫存器和位址暫存器::

接收資料我們使用移位暫存器接收從 sdram 回傳的 sdram_dqi，在原本的 enable 訊號再加上讀取的位址判斷，這樣可以根據讀取的位址控制對應的移位暫存器。然而位址資料的儲存，跟移位暫存器一樣，在 enable 訊號再加上讀取的位址判斷，於狀態 READ 根據讀取的位址將對應位址放到對應的暫存器。



```
always @(posedge clk) begin
    if(read_valid3 && addr_d[11:8] == 4'd6) begin
        prefetch_ddata[prefetch_isize] <= sdram_dqi;
        for (i = 0; i < prefetch_isize; i = i + 1)
            prefetch_ddata[i] <= prefetch_ddata[i+1];
    end
    else begin
        for (i = 0; i < prefetch_isize + 1; i = i + 1)
            prefetch_ddata[i] <= prefetch_ddata[i];
    end
end

always@(posedge clk) begin
    if(state_d == READ && addr_d[11:8] == 4'd6) begin
        for (i = 0; i < prefetch_dsize + 1; i = i + 1)
            prefetch_daddr[i] <= addr + i*4;
    end
    else begin
        for (i = 0; i < prefetch_dsize + 1; i = i + 1)
            prefetch_daddr[i] <= prefetch_daddr[i];
    end
end
```

```
always @(posedge clk) begin
    if(read_valid3 && addr_d[11:8] != 4'd6) begin
        prefetch_idata[prefetch_isize] <= sdram_dqi;
        for (i = 0; i < prefetch_isize; i = i + 1)
            prefetch_idata[i] <= prefetch_idata[i+1];
    end
    else begin
        for (i = 0; i < prefetch_isize + 1; i = i + 1)
            prefetch_idata[i] <= prefetch_idata[i];
    end
end

always@(posedge clk) begin
    if(state_d == READ && addr_d[11:8] != 4'd6) begin
        for (i = 0; i < prefetch_isize + 1; i = i + 1)
            prefetch_iaddr[i] <= addr + i*4;
    end
    else begin
        for (i = 0; i < prefetch_isize + 1; i = i + 1)
            prefetch_iaddr[i] <= prefetch_iaddr[i];
    end
end
```

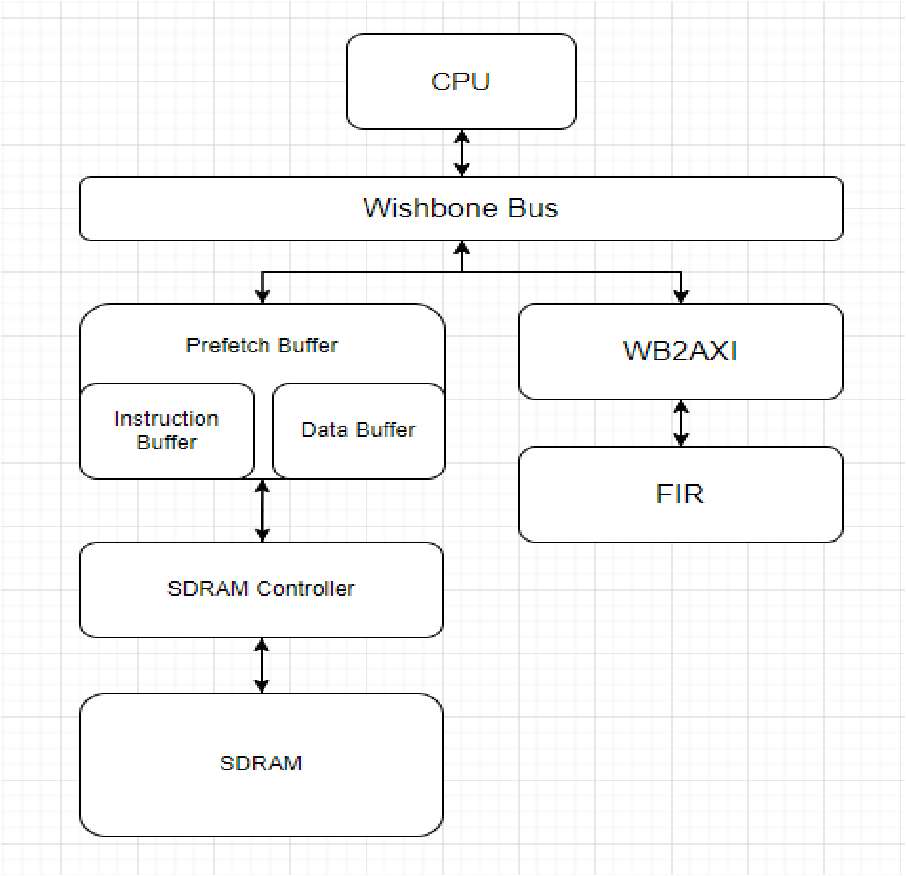
以下將針對 data prefetch buffer 大小與執行時間、讀取次數進行實驗:

prefetch buffer size (instruction ,data)	endtime	read times
(7,0)	2108988	631
(7,1)	2104438	617
(7,3)	2099213	602
(7,7)	2097863	597

可以看到將 data buffer 跟 instruction buffer 分開，執行時間和讀取次數都減少了，但效果都不明顯，猜測是矩陣乘法造成的，因為矩陣乘法在計算時讀取 data 的位址不是連續的，讓 prefetch 的到 data 可能都沒有使用到，減少的幅度都不大，所以我們沒有使用這個方式，而是將 buffer 放在 sdram controller 外部的方式。

3. Prefetch for Memory access

Block Diagram

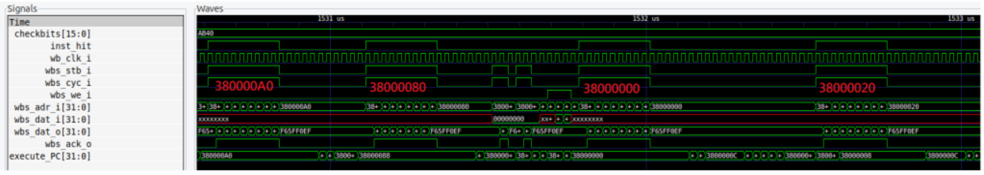


Data & Instruction Buffer

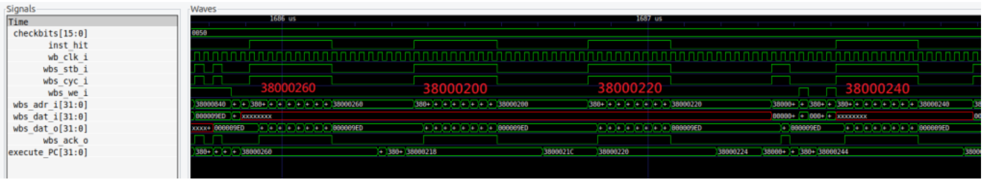
- 128 bytes
- Fully associative
- Write through
- FIFO Replacement

Hardware spec explanation:

Size(Instruction)



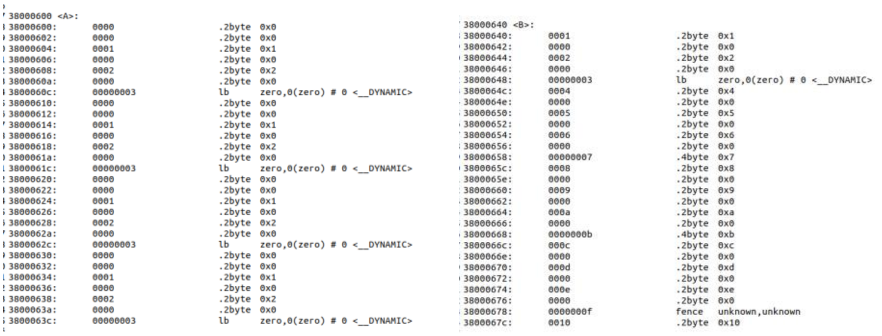
MM for loop Pattern 38000080 – 380000A0 - 38000000 – 38000020



QS for loop Pattern 38000200 – 38000220 - 38000240 – 38000260

由於 MM 和 QS 函式中包含多個 for 迴圈，而它們的執行時間被這些 for 迴圈所主宰，因此在 CPU 執行 for 迴圈時，記憶體的存取速度能夠越快速越好。因此，若 Instruction buffer 的 Size 能夠同時容納這四個位址的指令，將有助於提高 Buffer 的命中率。

Size(Data)



```
38000600 <A>:
38000600:       0000                    .2byte  0x0
38000602:       0000                    .2byte  0x0
38000604:       0001                    .2byte  0x1
38000606:       0000                    .2byte  0x0
38000608:       0002                    .2byte  0x2
3800060a:       0000                    .2byte  0x0
3800060c:       00000003                lb      zero,0(zero) # 0 <__DYNAMIC>
38000610:       0000                    .2byte  0x0
38000612:       0000                    .2byte  0x0
38000614:       0001                    .2byte  0x1
38000616:       0000                    .2byte  0x0
38000618:       0002                    .2byte  0x2
3800061a:       0000                    .2byte  0x0
3800061c:       00000003                lb      zero,0(zero) # 0 <__DYNAMIC>
38000620:       0000                    .2byte  0x0
38000622:       0000                    .2byte  0x0
38000624:       0001                    .2byte  0x1
38000626:       0000                    .2byte  0x0
38000628:       0002                    .2byte  0x2
3800062a:       0000                    .2byte  0x0
3800062c:       00000003                lb      zero,0(zero) # 0 <__DYNAMIC>
38000630:       0000                    .2byte  0x0
38000632:       0000                    .2byte  0x0
38000634:       0001                    .2byte  0x1
38000636:       0000                    .2byte  0x0
38000638:       0002                    .2byte  0x2
3800063a:       0000                    .2byte  0x0
3800063c:       00000003                lb      zero,0(zero) # 0 <__DYNAMIC>
```

```
38000640 <B>:
38000640:       0001                    .2byte  0x1
38000642:       0000                    .2byte  0x0
38000644:       0002                    .2byte  0x2
38000646:       0000                    .2byte  0x0
38000648:       00000003                lb      zero,0(zero) # 0 <__DYNAMIC>
3800064c:       0004                    .2byte  0x4
3800064e:       0000                    .2byte  0x0
38000650:       0005                    .2byte  0x5
38000652:       0000                    .2byte  0x0
38000654:       0006                    .2byte  0x6
38000656:       0000                    .2byte  0x0
38000658:       00000007                .4byte  0x7
3800065c:       0008                    .2byte  0x8
3800065e:       0000                    .2byte  0x0
38000660:       0009                    .2byte  0x9
38000662:       0000                    .2byte  0x0
38000664:       000a                    .2byte  0xa
38000666:       0000                    .2byte  0x0
38000668:       0000000b                .4byte  0xb
3800066c:       000c                    .2byte  0xc
3800066e:       0000                    .2byte  0x0
38000670:       000d                    .2byte  0xd
38000672:       0000                    .2byte  0x0
38000674:       000e                    .2byte  0xe
38000676:       0000                    .2byte  0x0
38000678:       0000000f                fence   unknown,unknown
3800067c:       0010                    .2byte  0x10
```

Matmul 的矩陣變數(length = 0x80)

MM 在做矩陣乘法運算時，CPU 會頻繁去存取兩個矩陣的資料進入 CPU 中， 因此，Data buffer 的設計需要能夠容納這兩個矩陣的資料，這樣在做 MM 運算時，才不會時常發生 Miss 的情況。

而之所以會優先考慮優化 MM 的原因在於它的執行時間相對於其他兩個函式更為耗時。理論上來說，優化 MM 所帶來的效益很可能會比其他兩個函式更加顯著。

Fully associative

QS for loop pattern :

Wbs_adr_i[7:0]

00 -> 20 -> 40 -> 60

[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0
0	1	1	0	0	0	0	0

MM for loop pattern :

Wbs_adr_i[7:0]

80 -> A0 -> 00 -> 20

[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
1	0	1	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0

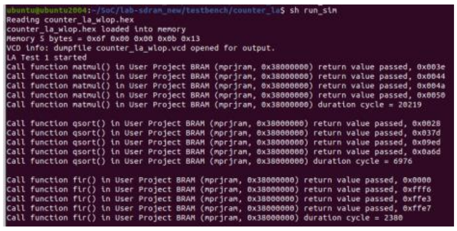
由於在執行 MM 和 QS 函式時，CPU 會需要連續執行這些 Pattern 好幾次，因此如果要降低 Buffer Miss，我們需要能夠把這些 Pattern 同時存在 Buffer 中。然而，MM 和 QS 的 Pattern 無法用特定某個 bit 去區分，如果採用直接映射的方式，將無法把這些 Pattern 同時存在 Buffer 中，最後我們選擇採用 Fully associative 去做映射，便能夠很好的解決此問題。

FIFO Replacement

當 Prefetch Buffer 發生 Miss 的情況時，將會挑選最早進入的資料進行替換，這是因為程式的 locality 特性，最早進入的資料在下一個時間被使用的可能性最低，因此替換它將能夠有效提高 Buffer 的 hit rate。

Result:

No Prefetch Buffer



```
ubuntu@ubuntu2004:~/SoC/lab-sdram_new/testbench/counter_la$ sh run_sim
Reading counter_la_wlop.hex
counter_la_wlop.hex loaded into memory
Memory 5 bytes = 0x6f 0x00 0x00 0x0b 0x13
VCD info: dumpfile counter_la_wlop.vcd opened for output.
LA Test 1 started
Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x003e
Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x0044
Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x004a
Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x0050
Call function matmul() in User Project BRAM (mprjram, 0x38000000) duration cycle = 20219

Call function qsort() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x0028
Call function qsort() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x037d
Call function qsort() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x09ed
Call function qsort() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x0a6d
Call function qsort() in User Project BRAM (mprjram, 0x38000000) duration cycle = 6976

Call function fir() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x0000
Call function fir() in User Project BRAM (mprjram, 0x38000000) return value passed, 0xfff6
Call function fir() in User Project BRAM (mprjram, 0x38000000) return value passed, 0xffe3
Call function fir() in User Project BRAM (mprjram, 0x38000000) return value passed, 0xffe7
Call function fir() in User Project BRAM (mprjram, 0x38000000) duration cycle = 2380
```

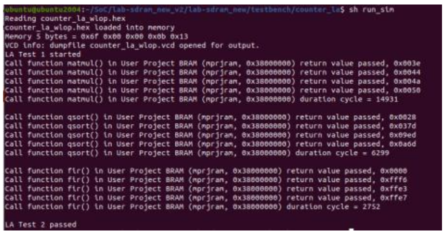
MM: 20219 cycles

QS: 6976 cycles

FIR: 2380 cycles

Total: 29575 cycles

With Prefetch Buffer



```
ubuntu@ubuntu2004:~/SoC/lab-sdram_new_v2/lab-sdram_new/testbench/counter_la$ sh run_sim
Reading counter_la_wlop.hex
counter_la_wlop.hex loaded into memory
Memory 5 bytes = 0x6f 0x00 0x00 0x0b 0x13
VCD info: dumpfile counter_la_wlop.vcd opened for output.
LA Test 1 started
Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x003e
Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x0044
Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x004a
Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x0050
Call function matmul() in User Project BRAM (mprjram, 0x38000000) duration cycle = 14931

Call function qsort() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x0028
Call function qsort() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x037d
Call function qsort() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x09ed
Call function qsort() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x0a6d
Call function qsort() in User Project BRAM (mprjram, 0x38000000) duration cycle = 6299

Call function fir() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x0000
Call function fir() in User Project BRAM (mprjram, 0x38000000) return value passed, 0xfff6
Call function fir() in User Project BRAM (mprjram, 0x38000000) return value passed, 0xffe3
Call function fir() in User Project BRAM (mprjram, 0x38000000) return value passed, 0xffe7
Call function fir() in User Project BRAM (mprjram, 0x38000000) duration cycle = 2752

LA Test 2 passed
```

MM: 14931 (x1.35) cycles

QS: 6299 (x1.10) cycles

FIR: 2752 (x0.86) cycles

Total: 23982 (x1.23) cycles

其中 MM 和 QS 相較於之前有小幅加速，而 FIR 的執行時間則有增加的現象，這是因為佔據 FIR 大部分執行時間的 for loop 小於 16 個指令，而 CPU 內部的 Cache 容量為 16 個指令。在這種情況下，執行 FIR 時 CPU 幾乎不必向記憶體存取指令，因此 Prefetch Buffer 很難有效發揮，而 Prefetch Buffer 的 Overhead 反而會去拖累了 FIR 的執行時間，最後導致 FIR 的執行時間不減反增。