SOC LAB

Final project Report

郭桓愷

2024/01/04

- Topic:

1. 實現 fir,matmal 硬體加速器
2. 以 srdram 取代 bram
3. Address remapping
4. 在 uart 添加 reciver FIFO 與 transmitter FIFO

UART

- Improvement:

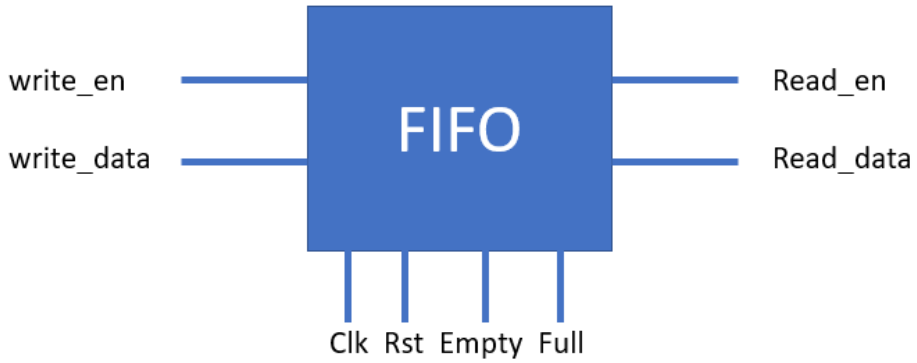
添加了 reciver FIFO 與 transmitter FIFO。

- 目的:

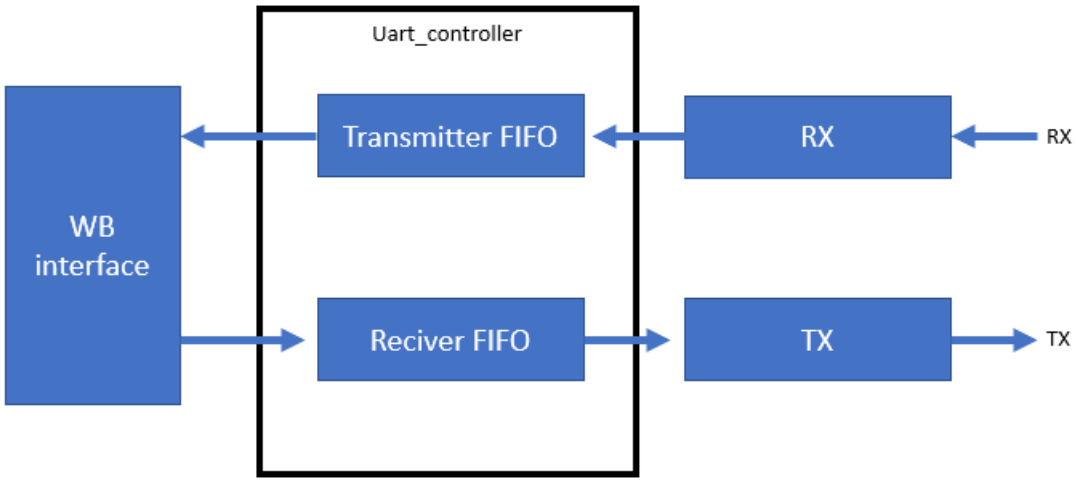
在尚未添加 fifo 時，當 rx 完成接收 data 時，會發出中斷訊號且 rx 會進入 wait 的狀態，並直到 data 通過 wishbone 離開時，rx 才會繼續接收下一筆 data。

加入 fifo 後，當 rx 完成接收 data 時，一樣會發出中斷訊號，但 data 直接存入 fifo，此時 rx 即可進行下一筆 data 的接收。

另一方面，在尚未添加 fifo 時，processor 須等待 tx 完成傳輸後才會接續工作。但加入 fifo 後即可將 data 存入 fifo 中，而 processor 則可直接進行其餘工作。



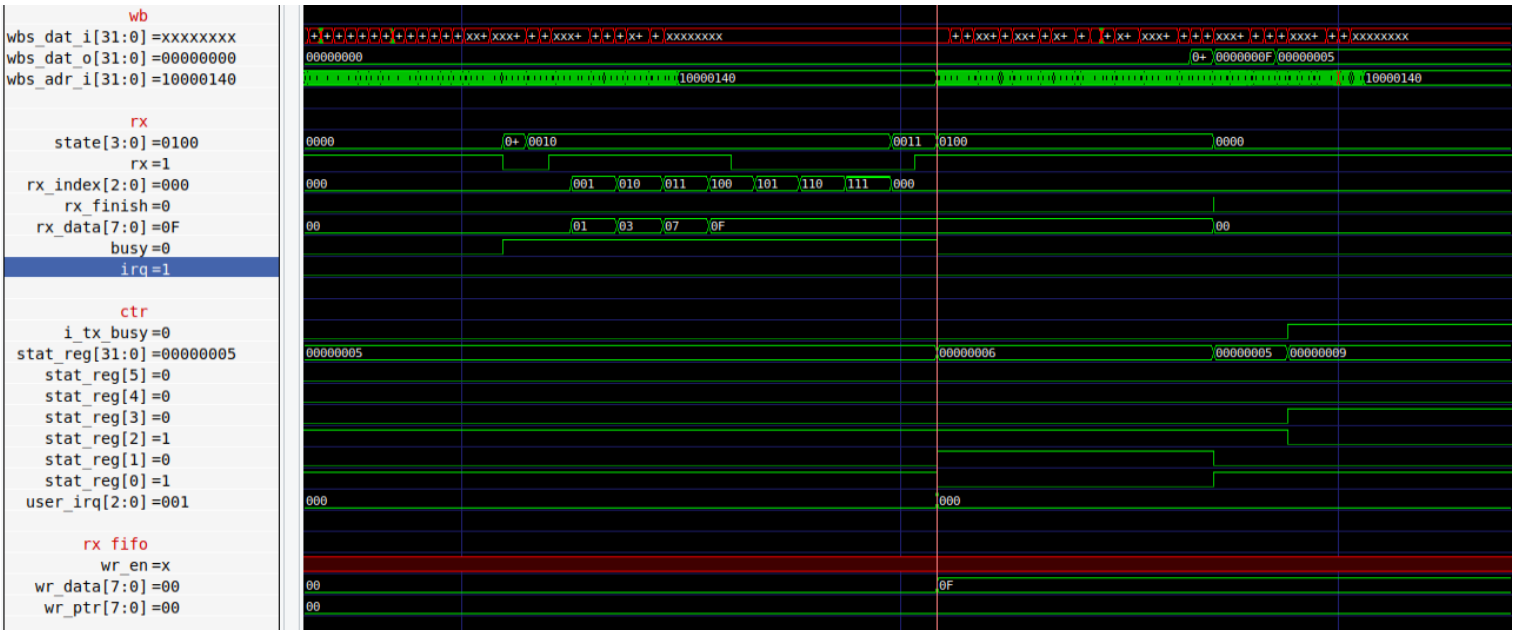
- 架構:



- 模擬結果:

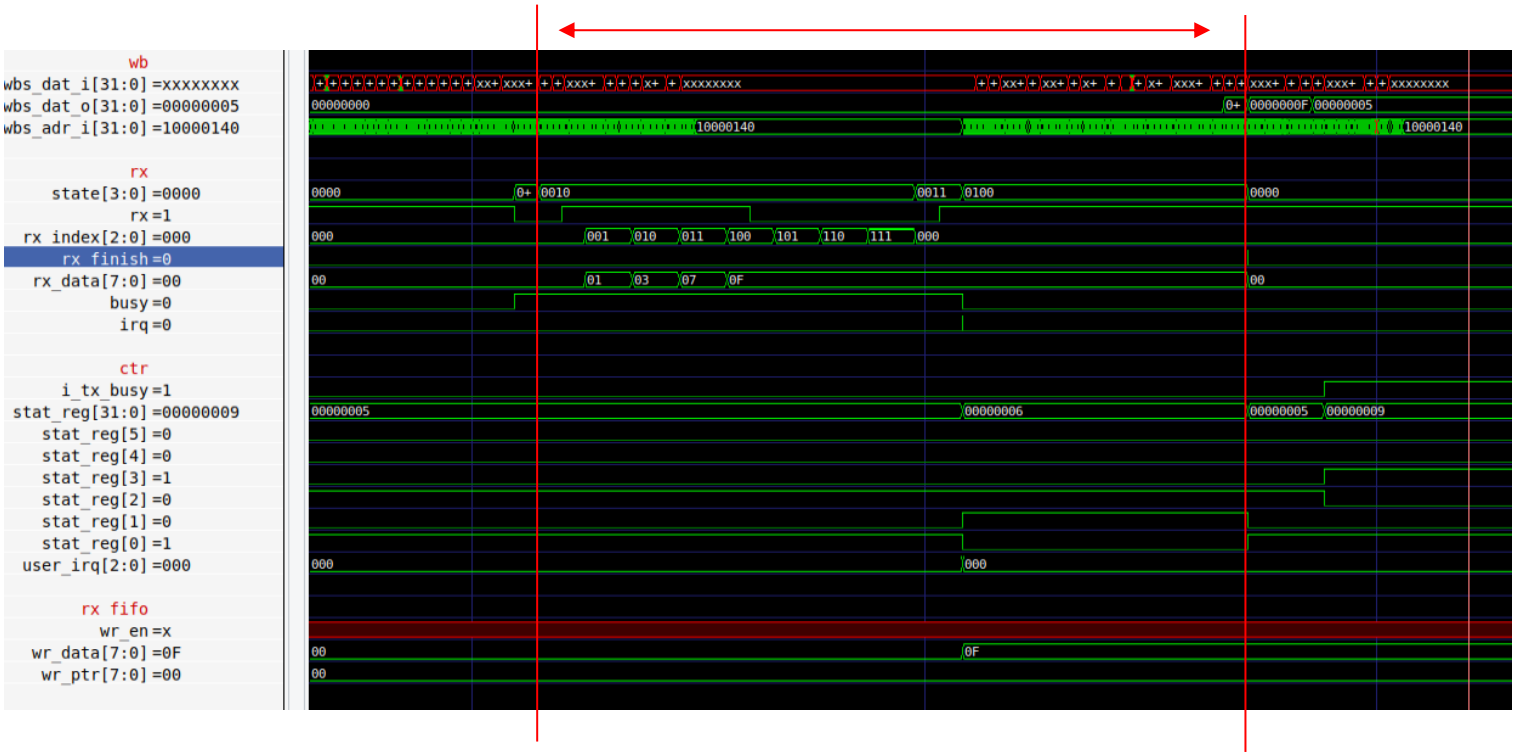
無添加 FIFO:

接收完 Rx 的 data 後會發送 irq 訊號

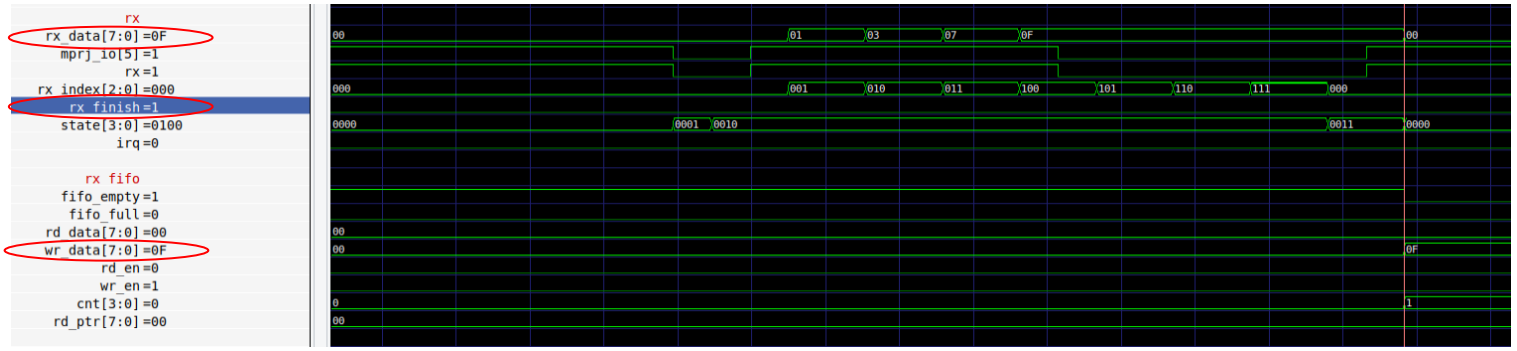


在 wb_addr=3000 0000 時 rx_data 會透過 wb 傳輸，完成後 finish 拉高。而 finish 拉高後 rx 可進行下一筆的 data 接收。

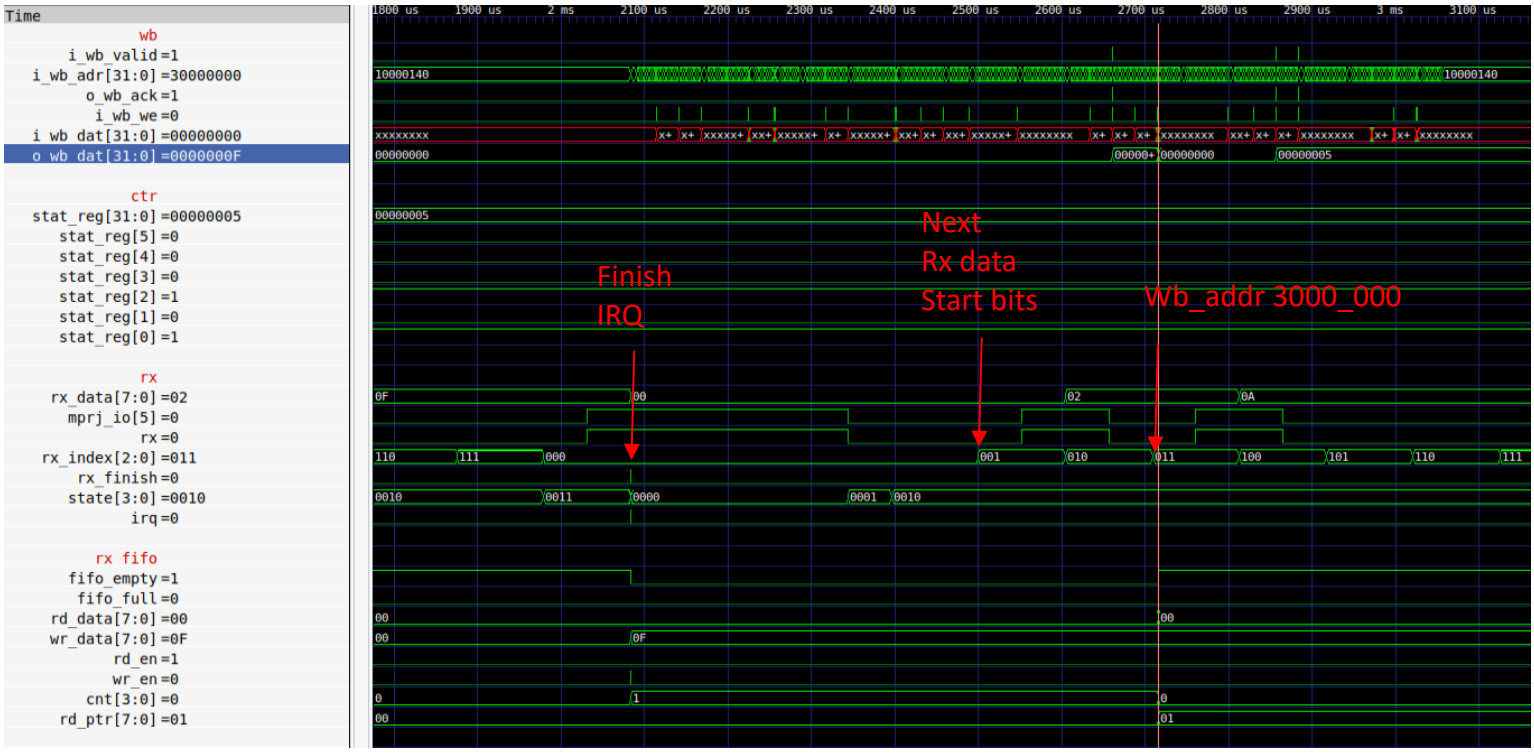
從 rx 完成 data 的接收後發出 Irq，到 controller 回傳 finish 時所經過的時間，約 630us。
而 rx 完整一次傳輸，由 rx start_bits 直到 data 送入 wishbone，約需要 2664us，(rx_state 由 start_bits 到 wait_read 結束)



添加 FIFO 後:



Rx_data 接收完成後發送 irq 訊號且會存進 fifo，完成後 contoller 隨即發送 finish。
(在此 tb 預設在收到 finish 後 255us 發送下一筆 rx)



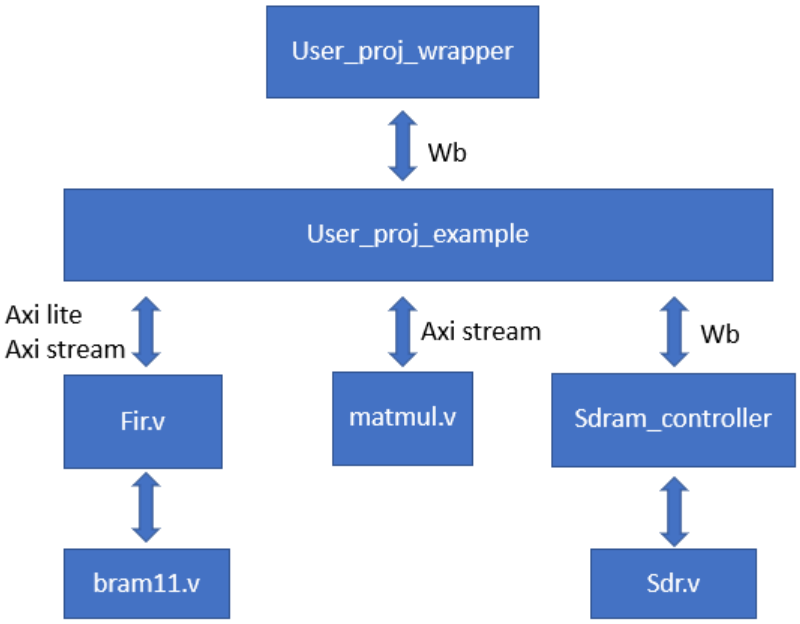
經過約 630us 後當 wb_addr 為 3000_0000 時，會觸發 FIFO 讀取 data 並透過 wb 送出 data。與無添加 FIFO 的波型圖相比， 添加 FIFO 後的 rx 在 wb 尚未送出 data 前就已在進行第二筆 rx_data 的傳輸，大幅提升工作效率。

- Observe:

可以依照需求來決定要每接收到一筆 data 就發送一次 interupt，或是當 FIFO 內存滿 data 時才發送 data。後者由於發送 interupt 的次數較少因此具有較高的效率。但由於時間的關係，本次實作並未將其實做出來。

Fir、matmul

- User_proj_area



- Hardware accelerator implementation
  - fir
    在 fir 中我們透過 axi lite 接口傳送 data length、ap_ctr 訊號及 coef。而 data 的 input 與 output 則是透過 axi stream 傳輸。
  - matmul
    使用 axi stream 將 A，B 矩陣送入，待運算完成後逐一送出結果。

```
## Headers
-v ../../rtl/header/defines.v
-v ../../rtl/header/user_defines.v

## User project
-v ../../rtl/user/user_project_wrapper.v
-v ../../rtl/user/user_proj_example.counter.v
-v ../../rtl/user/bram11.v
-v ../../rtl/user/fir.v
-v ../../rtl/user/Matmul.v
-v ../../rtl/user/sdr.v
-v ../../rtl/user/sdram_controller.v
#-v ../../rtl/user/bram.v

## VIP
-v ../../vip/tbuart.v
-v ../../vip/spiflash.v
```

本次期末專題我們將原先用來存放 execute code 的 bram 換成 sdram，並將.h 檔內所宣告的 data 一併存入 sdram 中。

- 修改 firmware 中的 section.lds

```
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 = 0x00100000
        mprjram : ORIGIN = 0x38000000, LENGTH = 0x00000800
        all_data : ORIGIN = 0x38000800, LENGTH = 0x00000800
        hk : ORIGIN = 0x26000000, LENGTH = 0x00100000
        csr : ORIGIN = 0xf0000000, LENGTH = 0x00010000
}
```

自行定義出 all_data: 0x38000800~0x38001600 這段區間，用來存放 data，而 mprjram 該區間則是用來存放 execute code。

而之所以這樣定義區間是因為從 testbench 中的.out 檔中可以確認 data 與 execute code 所占用的空間，我們定義的區間需足夠裝下那些資料，再加上我們須將 data 與 excecute code 存放在 sdram 中的不同 bank，而我最後是決定利用 addr[11:10]的值來分配 bank。

- 將存放 data 的位置由原先的 dff 改為我們自己定義的區間 all_data

```
   .data :
   {
           . = ALIGN(8);
           _fdata = .;
           *(.data .data.* .gnu.linkonce.d.*)
           *(.data1)
           _gp = ALIGN(16);
           *(.sdata .sdata.* .gnu.linkonce.s.*)
           . = ALIGN(8);
           _edata = .;
   } > all_data AT > flash

   .bss :
   {
           . = ALIGN(8);
           _fbss = .;
           *(.dynsbss)
           *(.sbss .sbss.* .gnu.linkonce.sb.*)
           *(.scommon)
           *(.dynbss)
           *(.bss .bss.* .gnu.linkonce.b.*)
           *(COMMON)
           . = ALIGN(8);
           _ebss = .;
           _end = .;
   } > all_data AT > flash
```

- Address remapping

```
// MA Map
// BA (Bank Address)  - 9:8
// RA (Row Address)   - 22:10
// CA (Col Address)   - 2'b0, 1'b0, <7:0>, 2'b0
`define BA      9:8
`define RA      22:10
`define CA      7:0
```

Sdram_controller 中預設:
addr[22:10]為 row address
addr[9:8]為 bank address
addr[7:0]為 colum address

此外，在 sdr.v 及 sdram_controller.v 中定義了 a_d[5:0]為 addr_q[7:2]，而用於 blockram 的 addr 為{row[4:0],col[5:0]}。

```
///// WRITE /////
WRITE: begin
    cmd_d = CMD_WRITE;

    dq_d = data_q;
    // data_d = data_q;
    dq_en_d = 1'b1; // enable out bus
    // a_d = {2'b0, 1'b0, addr_q[7:0], 2'b00};
    a_d = {7'b0, addr_q[7:2]};
    ba_d = addr_q[9:8];

    state_d = IDLE;
end
```

```
blkRam#(.SIZE(mem_sizes), .BIT_WIDTH(DQ_BITS))
Bank0(
    .clk(Sys_clk),
    .we(bwen[0]),
    .re(bren[0]),
    .waddr({Row[4:0], Col_brst[5:0]}),
    .raddr({Row[4:0], Col_brst[5:0]}),
    .d(bdi[0]),
    .q(bdq[0])
);
```

然而我們在 section.lds 定義的 address 是[11:10]的值來分配 bank。而其餘[9:0]的部

分又超出 col addr 的範圍，因此需要對 address 做 remapping。

bank0

bit	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Wb adr	...	...	...	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

bit	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Wb adr	...	...	...	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1

bank1

bit	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Wb adr	...	...	...	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0

bit	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Wb adr	...	...	...	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1

bank2

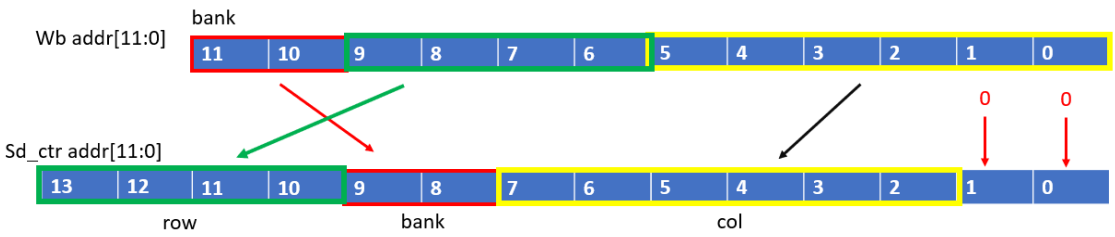
bit	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Wb adr	...	...	...	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

bit	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Wb adr	...	...	...	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1

bank3

bit	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Wb adr	...	...	...	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0

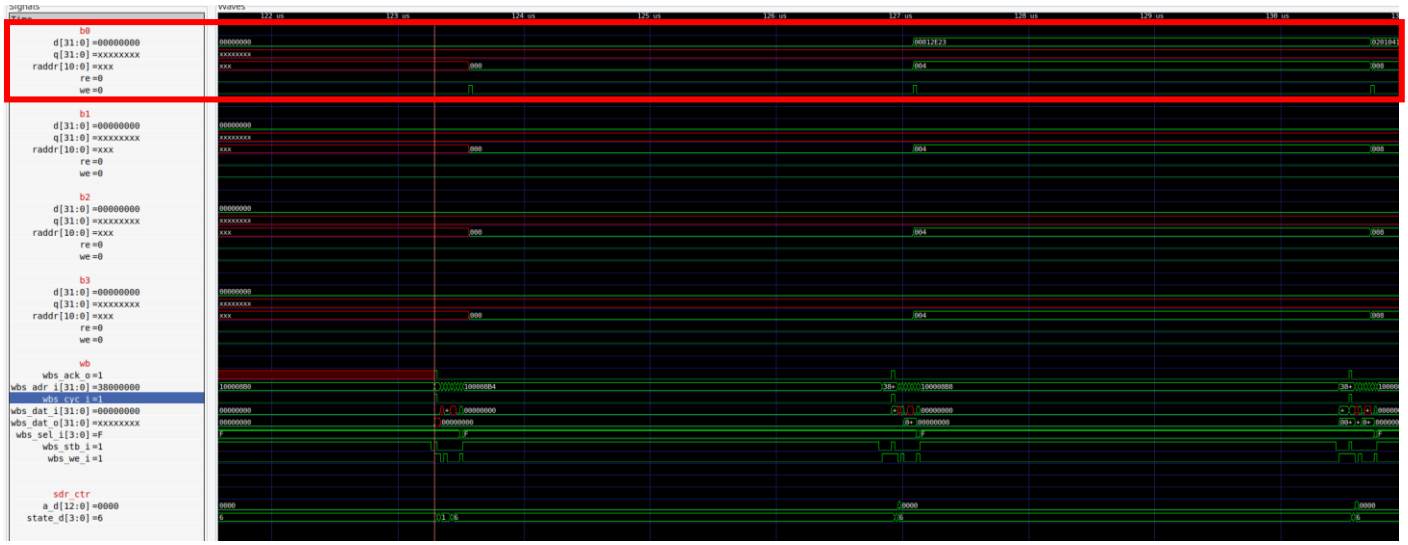
bit	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Wb adr	...	...	...	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1



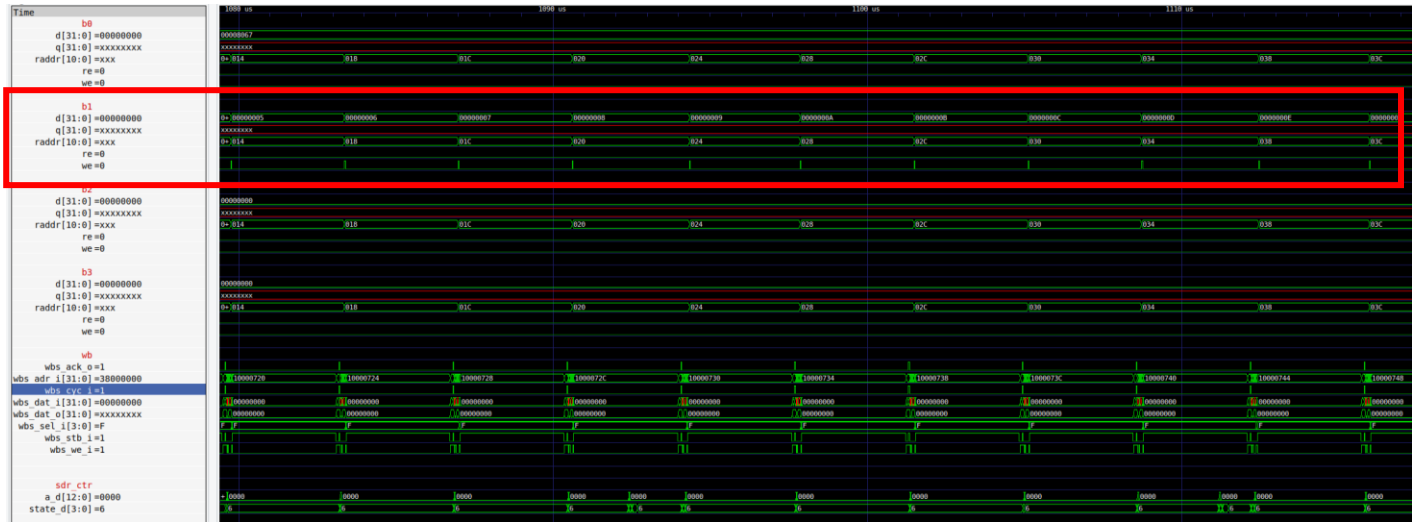
```
wire [22:0] addr;
wire [9:6] Mapped_RA;
wire [1:0]  Mapped_BA;
wire [5:0]  Mapped_CA;
assign Mapped_RA = user_addr[10:6];
assign Mapped_BA = user_addr[12:11];
assign Mapped_CA = user_addr[5:0];
assign addr = {9'b0,Mapped_RA, Mapped_BA, Mapped_CA , 2'b0};
```

- 模擬結果:

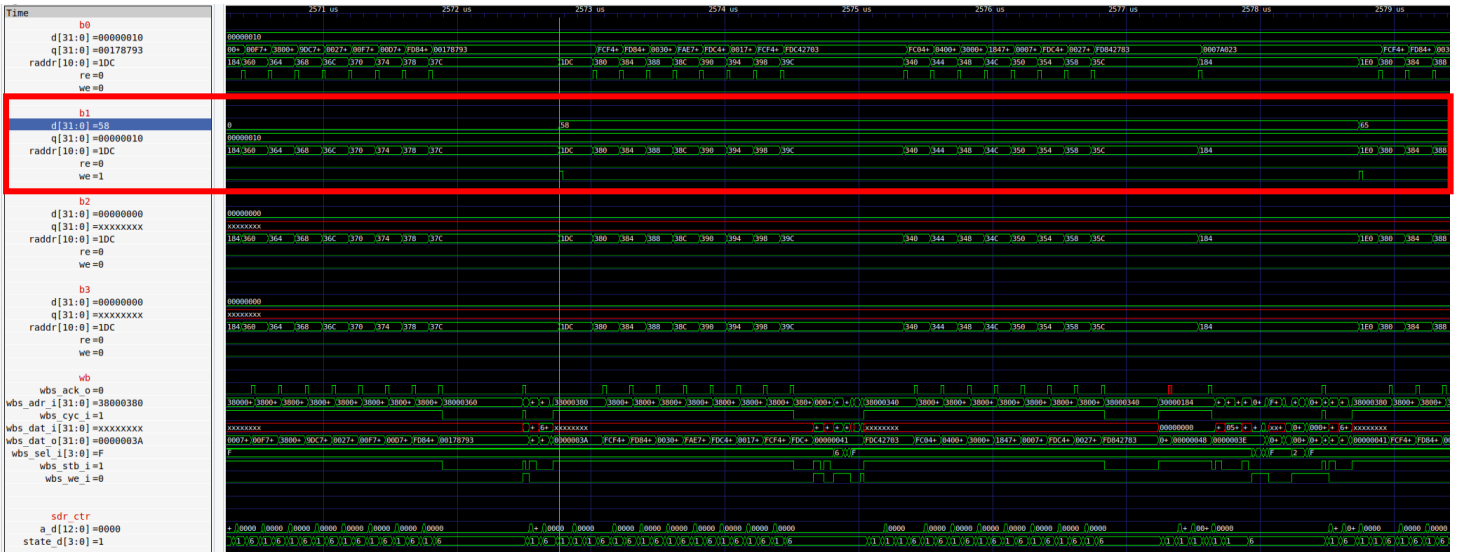
Write execute code into bank0



write data into bank1



Write the result into bank1



- Observe:

目前只有找到將 data 與 execute code 分開存放的方法，還尚未找到可以將 fir、mm、qs 的 data 分成不同 banck 存放的方式，若可以達成將以上的 data 分開存放，將有機會實現更加有效率的取值途徑。