

2024/XX/XX

## 實驗十二

姓名：黃文祺      學號：01057013

班級：資工 3A

E-mail：OOOOOOOOO

# 注意

---

1. 繳交時一律轉 PDF 檔
2. 繳交期限為  
隔週三上午九點
3. 一人繳交一份
4. 檔名：學號\_HW?.pdf  
檔名請按照作業檔名格式進行填寫  
未依照格式不予批改

## SPI Slave

### ■ 實驗說明：

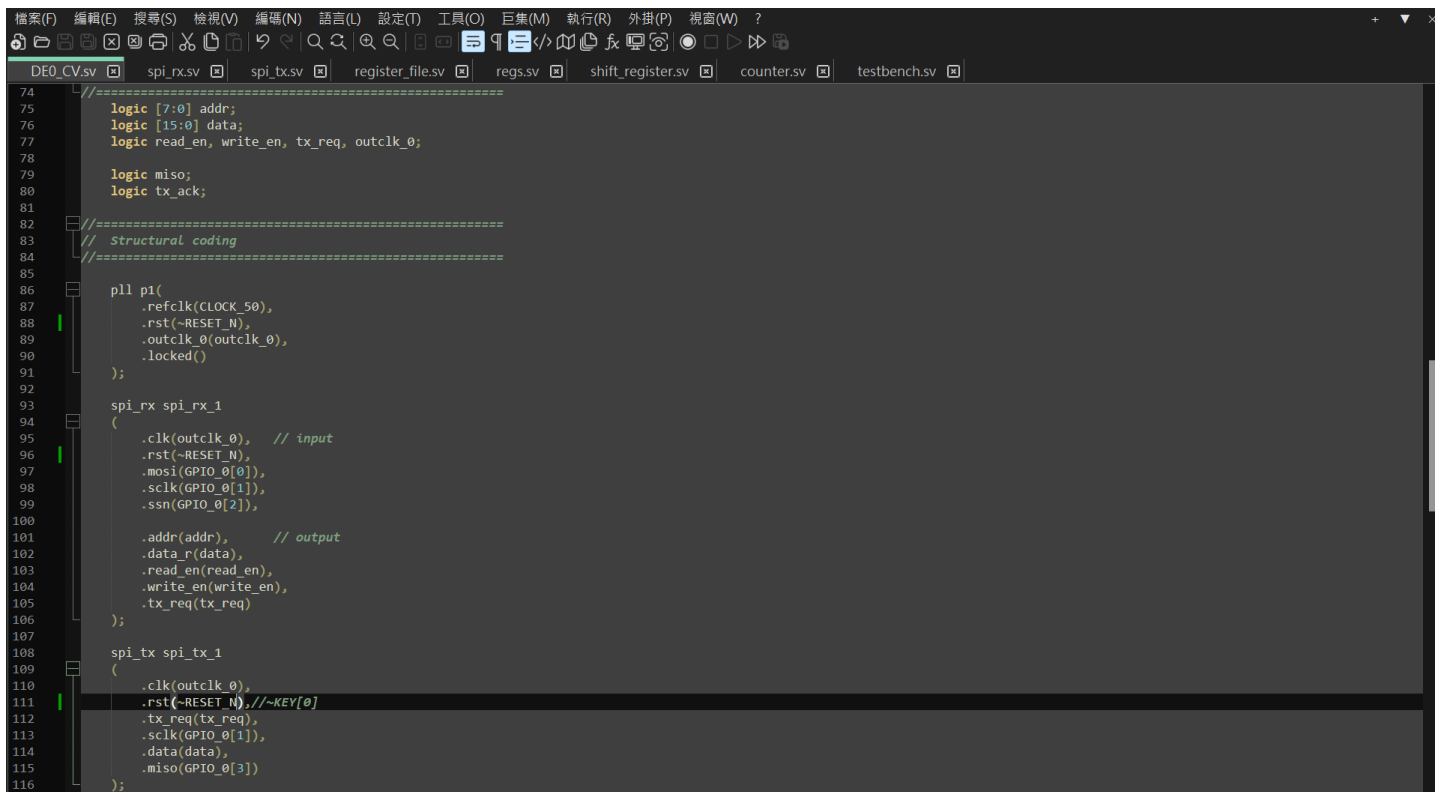
1. 將 input 和 filter 透過 SPI 分別寫進 FIFO 和 Register File 後，輸入一個指令執行兩者的 convolution 計算，再輸入另一個指令將計算完的值透過 SPI 輸出。
2. 指令格式:
  - `$$00_xxxx` : 將 filter xxxx 輸入至 register file 內的 \$\$ 位址
  - `8000_xxxx` : 將 input xxxx 輸入至 FIFO 內儲存
  - `8100_0000` : 將 3\*3 的 input 和 3\*3 的 filter 做 convolution
  - `8280_0000` : 讀出計算完的值
3. testbench.sv 會呼叫 DE0\_CV.sv，請在 DE0\_CV.sv 中完成程式碼撰寫。
4. DE0\_CV.sv 使用接腳：
  - `CLOCK_50` : 50MHz 的 clk 訊號。
  - `RESET_N` : 系統 reset，為 0 時重置系統。
  - `GPIO_0[0]` : 傳訊號進 SPI Slave 的 mosi。
  - `GPIO_0[1]` : 傳訊號進 SPI Slave 的 sclk。
  - `GPIO_0[2]` : 傳訊號進 SPI Slave 的 ssn。
  - `GPIO_0[3]` : 接收 SPI Slave 的 miso 訊號。
5. SPI.sv 輸入:
  - clk (請將 DE0\_CV 的 `CLOCK_50`，用 PLL 升至 100MHz)
  - mosi
  - sclk(testbench 已設定為 10MHz)
  - ssn
  - reset
6. SPI.sv 輸出:
  - miso

- 務必擷取到 get\_data 的波型

## ■ 系統架構程式碼與程式碼說明

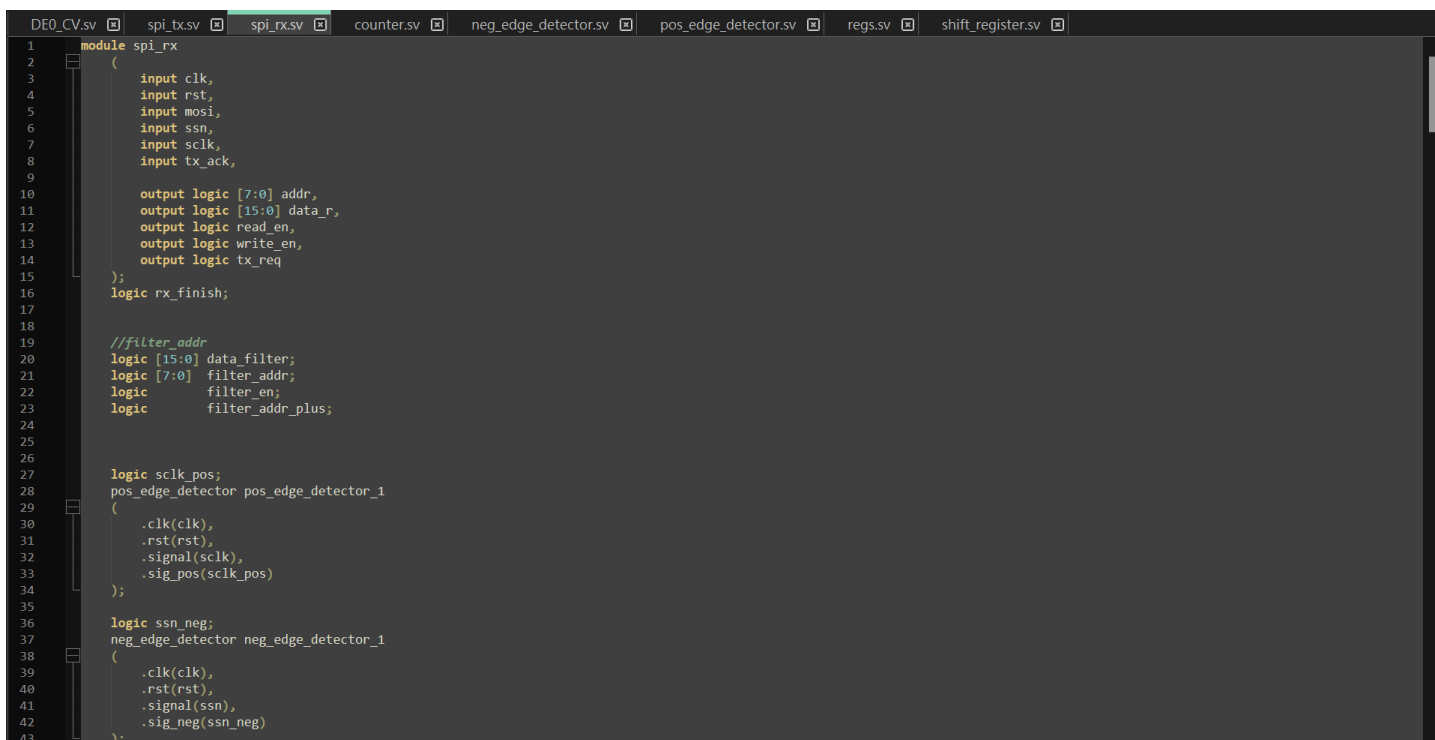
截圖請善用 win+shift+S

### DE0\_CV:



```
74 //=====
75 logic [7:0] addr;
76 logic [15:0] data;
77 logic read_en, write_en, tx_req, outclk_0;
78
79 logic miso;
80 logic tx_ack;
81
82 //=====
83 // Structural coding
84 //=====
85
86 pll p1(
87     .refclk(CLOCK_50),
88     .rst(~RESET_N),
89     .outclk_0(outclk_0),
90     .locked()
91 );
92
93 spi_rx spi_rx_1
94 (
95     .clk(outclk_0), // input
96     .rst(~RESET_N),
97     .mosi(GPIO_0[0]),
98     .sclk(GPIO_0[1]),
99     .ssn(GPIO_0[2]),
100
101     .addr(addr), // output
102     .data_r(data),
103     .read_en(read_en),
104     .write_en(write_en),
105     .tx_req(tx_req)
106 );
107
108 spi_tx spi_tx_1
109 (
110     .clk(outclk_0),
111     .rst(~RESET_N), //~KEY[0]
112     .tx_req(tx_req),
113     .sclk(GPIO_0[1]),
114     .data(data),
115     .miso(GPIO_0[3])
116 );
```

Rx:主要更改 rx，左邊有綠色的線是這次有修改的



```
1 module spi_rx
2 (
3     input clk,
4     input rst,
5     input mosi,
6     input ssn,
7     input sclk,
8     input tx_ack,
9
10     output logic [7:0] addr,
11     output logic [15:0] data_r,
12     output logic read_en,
13     output logic write_en,
14     output logic tx_req
15 );
16 logic rx_finish;
17
18 //filter_addr
19 logic [15:0] data_filter;
20 logic [7:0] filter_addr;
21 logic filter_en;
22 logic filter_addr_plus;
23
24
25
26
27 logic sclk_pos;
28 pos_edge_detector pos_edge_detector_1
29 (
30     .clk(clk),
31     .rst(rst),
32     .signal(sclk),
33     .sig_pos(sclk_pos)
34 );
35
36 logic ssn_neg;
37 neg_edge_detector neg_edge_detector_1
38 (
39     .clk(clk),
40     .rst(rst),
41     .signal(ssn),
42     .sig_neg(ssn_neg)
43 );
```

```

44
45 logic rst shift regs;
46 logic [15:0] shift_data;
47 shift_register shift_register_1
48 (
49     .rst(rst),
50     .clk(clk),
51     .sclk_pos(sclk_pos), // 每次 sclk 正緣觸發時
52     .mosi(mosi),         // 就把一個 bit 的資料寫入移位暫存器
53     .shift_data(shift_data)
54 );
55
56 logic rst_data_cnt;
57 logic load_data_cnt;
58 logic [15:0] rcv_data_cnt;
59 counter counter_1
60 (
61     .clk(clk),
62     .rst(rst_data_cnt),
63     .load(sclk_pos), // 每次 sclk 正緣觸發 bit 寫入移位暫存器時, 計數現在傳了幾個 bit
64     .cnt(rcv_data_cnt)
65 );
66
67
68 logic load_addr;
69 logic load_cmd;
70 logic load_data;
71 //Logic [7:0] address;
72 logic command;
73 //Logic [15:0] data;
74 logic [15:0] data;
75 regs regs_1
76 (
77     .clk(clk),
78     .rst(rst),
79     .load_addr(load_addr),
80     .load_cmd(load_cmd),
81     .load_data(load_data),
82     .shift_data(shift_data),
83
84     .address(addr),
85     .command(command),
86     .data(data)
87 );
88
89
90 logic [15:0] data_reg;
91 logic [15:0] reg_file [255:0];
92
93 always_ff @(posedge clk)begin
94     if(write_en) reg_file[addr] <= data;
95     if(read_en) data_reg <= reg_file[addr];
96
97 end
98
99 logic wrreq;
100 logic rdreq;
101 logic wrreq_neg;
102 logic rdreq_neg;
103 logic almost_empty;
104 logic [15:0] read_data_fifo;
105 logic [15:0] usedw;
106 logic [15:0] write_data_neg;
107
108 always_ff @(negedge clk)
109 begin
110     if(rst) begin
111         write_data_neg <= 0;
112         wrreq_neg <= 0;
113         rdreq_neg <= 0;
114     end
115     else begin
116         write_data_neg <= data;
117         wrreq_neg <= wrreq;
118         rdreq_neg <= rdreq;
119     end
120 end
121
122 fifo fifo_1
123 (
124     .clock(clk),
125     .data(write_data_neg),
126     .rdreq(rdreq_neg),
127     .sclr(rst),
128     .wrreq(wrreq_neg),
129     .almost_empty(almost_empty),
130     .empty(empty),
131     .full(full),
132     .q(read_data_fifo),
133     .usedw(usedw)
134 );
135

```

```

136 //conv
137 logic [15:0] conv;
138 logic [15:0] acc;
139 logic acc_load;
140 always_ff @(negedge clk)
141 begin
142     if(rst) begin
143         conv <= 0;
144         acc <= 0;
145         data_filter <= 0;
146     end
147     if(filter_en) data_filter <= reg_file[filter_addr];
148     if(rdreq) begin
149         conv <= data_filter * read_data_fifo;
150         //acc <= acc+conv;
151     end
152     if(acc_load) begin
153         //conv <= data_filter * read_data_fifo;
154         acc <= acc+conv;
155     end
156 end
157
158 assign data_r = addr[7] ? acc : data_reg;
159
160 //wrreq_cnt
161 //用來判斷fifo中有幾個數，類似可控制初始值的empty
162 logic [15:0] wrreq_cnt;
163 always_ff @(negedge clk)
164 begin
165     if(rst) begin
166         wrreq_cnt <= 1;
167     end
168     else if(rdreq) begin
169         wrreq_cnt <= wrreq_cnt - 1;
170     end
171     else if(wrreq) begin
172         wrreq_cnt <= wrreq_cnt + 1;
173     end
174 end
175

```

```

176 //filter_addr
177
178 always_ff @(negedge clk)
179 begin
180     if(rst) begin
181         filter_addr <= 1;
182     end
183     else if(filter_addr_plus) begin
184         filter_addr <= filter_addr + 1;
185     end
186     if(filter_addr == 10) begin
187         filter_addr <= 1;
188     end
189 end
190

```

```

191 typedef enum {
192     INIT,
193     START_SPI_RX,
194     RECEIVE_ADDRESS,
195     DUMMY,
196     DUMMY2,
197     CHECK_COMMAND,
198     TX_REQ,
199     FINISH,
200     RECEIVE_DATA,
201     WRITE
202 } state_t;
203
204 state_t ps, ns;
205
206 always_ff @(posedge clk)
207 if(rst) ps <= INIT;
208 else ps <= ns;
209

```

```

210 always_comb begin
211     rst_shift_regs = 0;
212     rst_data_cnt = 0;
213     load_data_cnt = 0;
214     rst_shift_regs = 0;
215     load_data_cnt = 0;
216
217     load_addr = 0;
218     load_cmd = 0;
219     load_data = 0;
220
221     write_en = 0;
222     rx_finish = 0;
223
224     read_en = 0;
225     tx_req = 0;
226
227     wrreq = 0;
228     rdreq = 0;
229
230     filter_en=0;
231     filter_addr_plus=0;
232     acc_load=0;
233
234     ns = ps;
235     case(ps)
236     INIT:
237         begin
238             ns = START_SPI_RX;
239         end
240
241     START_SPI_RX:
242         begin
243             if(ssn_neg)
244                 begin
245                     rst_data_cnt = 1;
246                     rst_shift_regs = 1;
247                     ns = RECEIVE_ADDRESS;
248                 end
249         end
250

```

```

251 RECEIVE_ADDRESS:
252     begin
253         if(rcv_data_cnt >= 8) begin
254             load_addr = 1;
255             ns = DUMMY;
256         end
257         load_data_cnt = 1;
258     end
259
260 DUMMY:
261     begin
262         if(rcv_data_cnt >= 16) begin
263             load_cmd = 1;
264             rst_data_cnt = 1;
265             ns = CHECK_COMMAND;
266         end
267         load_data_cnt = 1;
268     end
269
270 CHECK_COMMAND:
271     begin
272         if(command) begin //read
273
274             //if(addr[7]) rdreq = 1;
275             //else read_en = 1;
276             read_en = 1;
277             ns = TX_REQ;
278         end
279         else //write
280             if(addr == 8'h81) begin //在write中遇到addr為81時 139行
281                 filter_en = 1;
282                 rdreq = 1; //要求從fifo中讀取值 ==> if(rdreq) begin
283
284                 ns = DUMMY2; //做一個dummy等待加一次 conv <= conv + read_data_fifo;
285                 // end
286             end
287             else begin
288                 ns = RECEIVE_DATA;
289             end
290         end
291
292 DUMMY2:
293     begin
294         if(wrreq_cnt>0) begin //wrreq_cnt 146行
295             filter_addr_plus = 1;
296             acc_load=1;
297             ns = CHECK_COMMAND;
298         end
299         else begin
300             acc_load=1;
301             ns = FINISH;
302         end
303     end
304
305 TX_REQ:
306     begin
307         tx_req = 1;
308         ns = FINISH;
309     end
310
311 RECEIVE_DATA:
312     begin
313         if(rcv_data_cnt >= 16) begin
314             load_data = 1;
315             rst_data_cnt = 1;
316             ns = WRITE;
317         end
318         load_data_cnt = 1;
319     end
320
321 WRITE:
322     begin
323
324         if(addr == 8'h80) wrreq = 1; // if(addr[7]) wrreq = 1;
325         else write_en = 1;
326
327         ns = FINISH;
328     end
329
330 FINISH:
331     begin
332         rx_finish = 1;
333         ns = INIT;
334     end
335
336 endcase
337 end
338
339 endmodule
340
341
342

```

Tx:

```
檔案(F) 編輯(E) 搜尋(S) 檢視(V) 編碼(N) 語言(L) 設定(T) 工具(O) 巨集(M) 執行(R) 外掛(P) 視窗(W) ?
DE0_CV.sv spi_rx.sv spi_tx.sv register_file.sv regs.sv shift_register.sv counter.sv testbench.sv

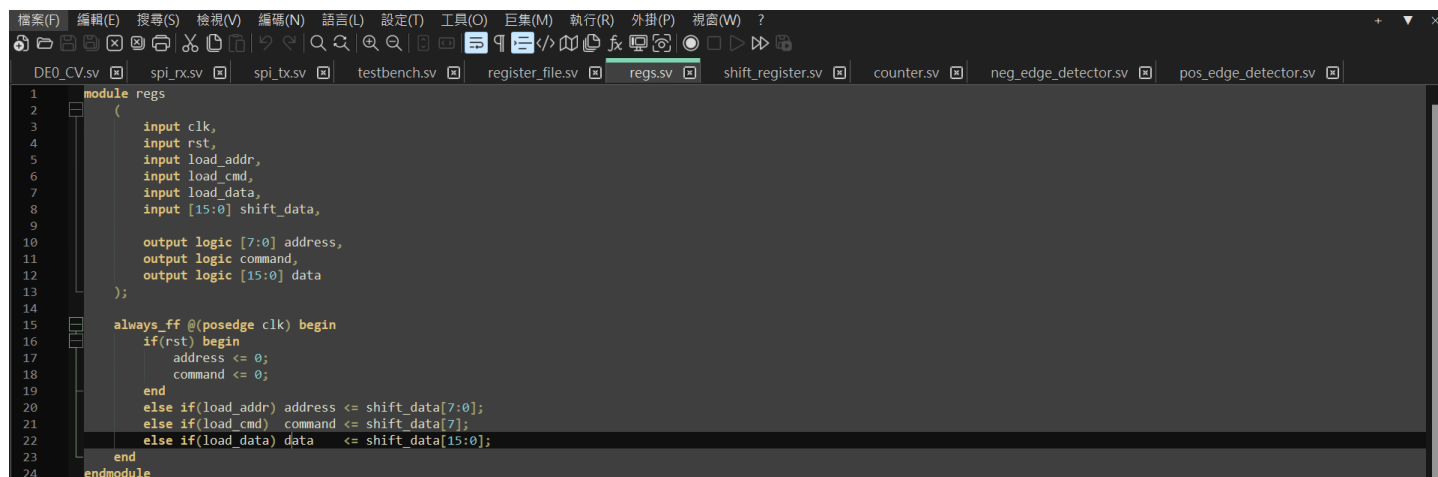
1 module spi_tx
2 (
3     input clk,
4     input rst,
5     input tx_req,
6     input sclk,
7     input logic [15:0] data,
8     output logic miso
9 );
10
11 logic sclk_neg;
12 neg_edge_detector neg_edge_detector_1
13 (
14     .clk(clk),
15     .rst(rst),
16     .signal(sclk),
17     .sig_neg(sclk_neg)
18 );
19
20 logic rst_data_cnt;
21 logic load_data_cnt;
22 logic [15:0] rcv_data_cnt;
23 counter counter_1
24 (
25     .clk(clk),
26     .rst(rst_data_cnt),
27     .load(sclk_neg), // 每次 sclk 正緣觸發 bit 寫入移位寄存器時, 計數現在傳了幾個 bit
28     .cnt(rcv_data_cnt)
29 );
30
31 // shift register
32 logic load_shift_data;
33 logic [15:0] shift_data;
34 always_ff @(posedge clk) begin
35     if(rst) shift_data <= 0;
36     else if(load_shift_data) shift_data <= data;
37     else if(sclk_neg) {miso, shift_data} <= {shift_data, 1'b0};
38 end
39
40
41
42 typedef enum {
43     INIT,
44     START_SPI_TX,
45     SEND_DATA,
46     FINISH
47 } state_t;
48
49 state_t ps, ns;
50
51 always_ff @(posedge clk)
52 if(rst) ps <= INIT;
53 else ps <= ns;
54
55 always_comb begin
56     rst_data_cnt = 0;
57     load_shift_data = 0;
58     ns = ps;
59     case(ps)
60     INIT:
61         begin
62             ns = START_SPI_TX;
63         end
64     START_SPI_TX:
65         begin
66             rst_data_cnt = 1;
67             if(tx_req) begin
68                 load_shift_data = 1;
69                 ns = SEND_DATA;
70             end
71         end
72     SEND_DATA:
73         begin
74             if(rcv_data_cnt >= 16) begin
75                 rst_data_cnt = 1;
76                 ns = FINISH;
77             end
78         end
79     FINISH: begin
80         ns = INIT;
81     end
82     endcase
83 end
84
85 endmodule
```

## 其他:



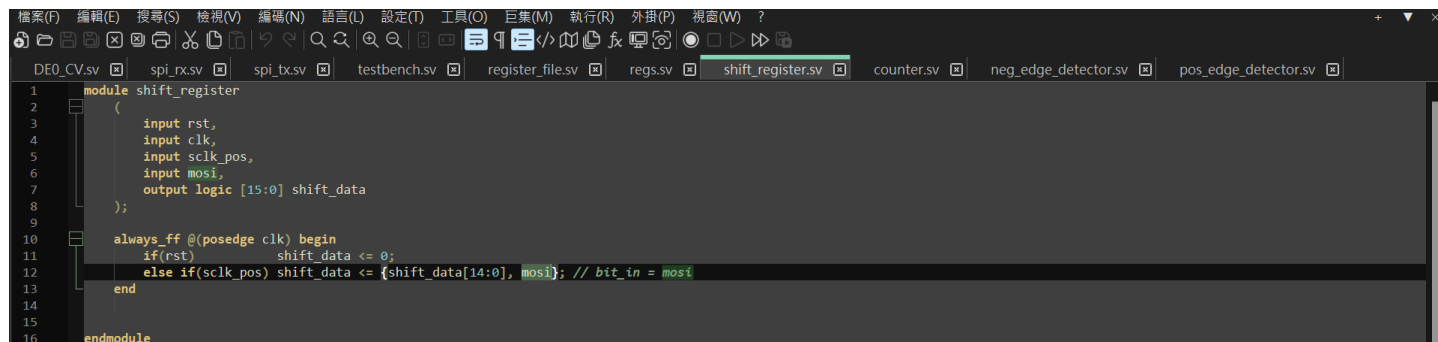
This screenshot shows the Verilog code for the `register_file` module. The module has inputs for `clk`, `rst`, `write_en`, `read_en`, a 7-bit `addr`, a 16-bit `data`, and a 16-bit `data_r` output. It contains a 256-bit `reg_file` array and an `always_ff` block that updates the register file on the clock edge based on the `write_en` and `read_en` signals.

```
1 module register_file
2 (
3     input clk,
4     input rst,
5     input write_en,
6     input read_en,
7     input [7:0] addr,
8     input [15:0] data,
9     output logic [15:0] data_r
10 );
11
12 logic [15:0] reg_file [255:0];
13
14 always_ff @(posedge clk) begin
15     if(write_en) reg_file[addr] <= data;
16     if(read_en) data_r <= reg_file[addr];
17 end
18
19 endmodule
```



This screenshot shows the Verilog code for the `regs` module. It has inputs for `clk`, `rst`, `load_addr`, `load_cmd`, `load_data`, and a 16-bit `shift_data`. It has outputs for a 7-bit `address`, a `command`, and a 16-bit `data`. The `always_ff` block handles the reset and loading of registers based on the `load_cmd` and `load_data` signals.

```
1 module regs
2 (
3     input clk,
4     input rst,
5     input load_addr,
6     input load_cmd,
7     input load_data,
8     input [15:0] shift_data,
9
10     output logic [7:0] address,
11     output logic command,
12     output logic [15:0] data
13 );
14
15 always_ff @(posedge clk) begin
16     if(rst) begin
17         address <= 0;
18         command <= 0;
19     end
20     else if(load_addr) address <= shift_data[7:0];
21     else if(load_cmd) command <= shift_data[7];
22     else if(load_data) data <= shift_data[15:0];
23 end
24 endmodule
```



This screenshot shows the Verilog code for the `shift_register` module. It has inputs for `rst`, `clk`, `sclk_pos`, and `mosi`, and a 16-bit `shift_data` output. The `always_ff` block shifts the data on the clock edge, with a comment indicating that the `mosi` signal is the most significant bit.

```
1 module shift_register
2 (
3     input rst,
4     input clk,
5     input sclk_pos,
6     input mosi,
7     output logic [15:0] shift_data
8 );
9
10 always_ff @(posedge clk) begin
11     if(rst) shift_data <= 0;
12     else if(sclk_pos) shift_data <= {shift_data[14:0], mosi}; // bit_in = mosi
13 end
14
15
16 endmodule
```



This screenshot shows the Verilog code for the `counter` module. It has inputs for `clk`, `rst`, and `load`, and a 16-bit `cnt` output. The `always_ff` block increments the counter on the clock edge, resetting it to 0 when `rst` is asserted or `load` is asserted.

```
1 module counter
2 (
3     input clk,
4     input rst,
5     input load,
6     output logic [15:0] cnt
7 );
8
9 always_ff @(posedge clk)
10     if (rst) cnt <= 0;
11     else if (load) cnt <= cnt + 1;
12
13 endmodule
```



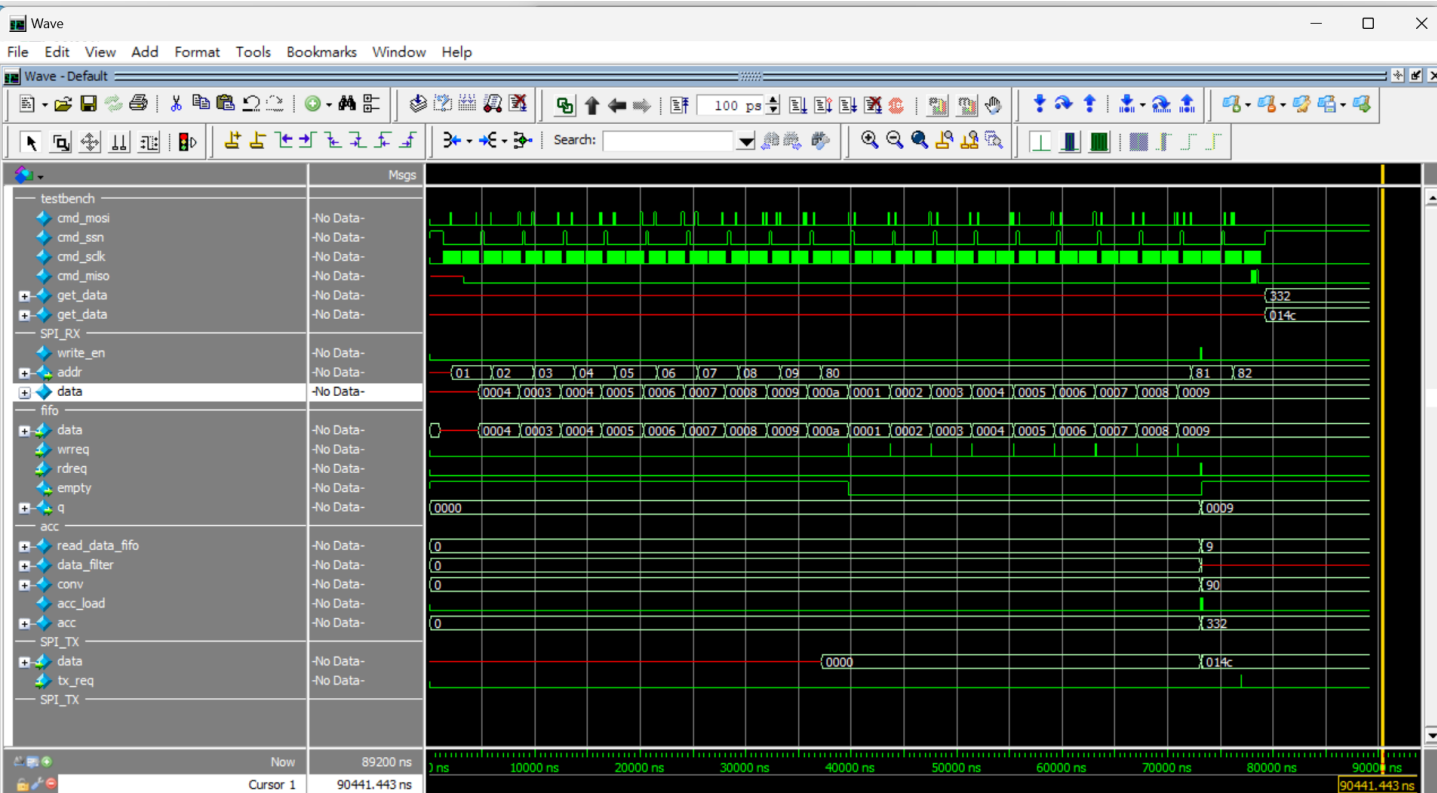
```
檔案(F) 編輯(E) 搜尋(S) 檢視(V) 編碼(N) 語言(L) 設定(T) 工具(O) 巨集(M) 執行(R) 外掛(P) 視窗(W) ?
DE0_CV.sv spi_rx.sv spi_tx.sv testbench.sv register_file.sv regs.sv shift_register.sv counter.sv neg_edge_detector.sv pos_edge_detector.sv

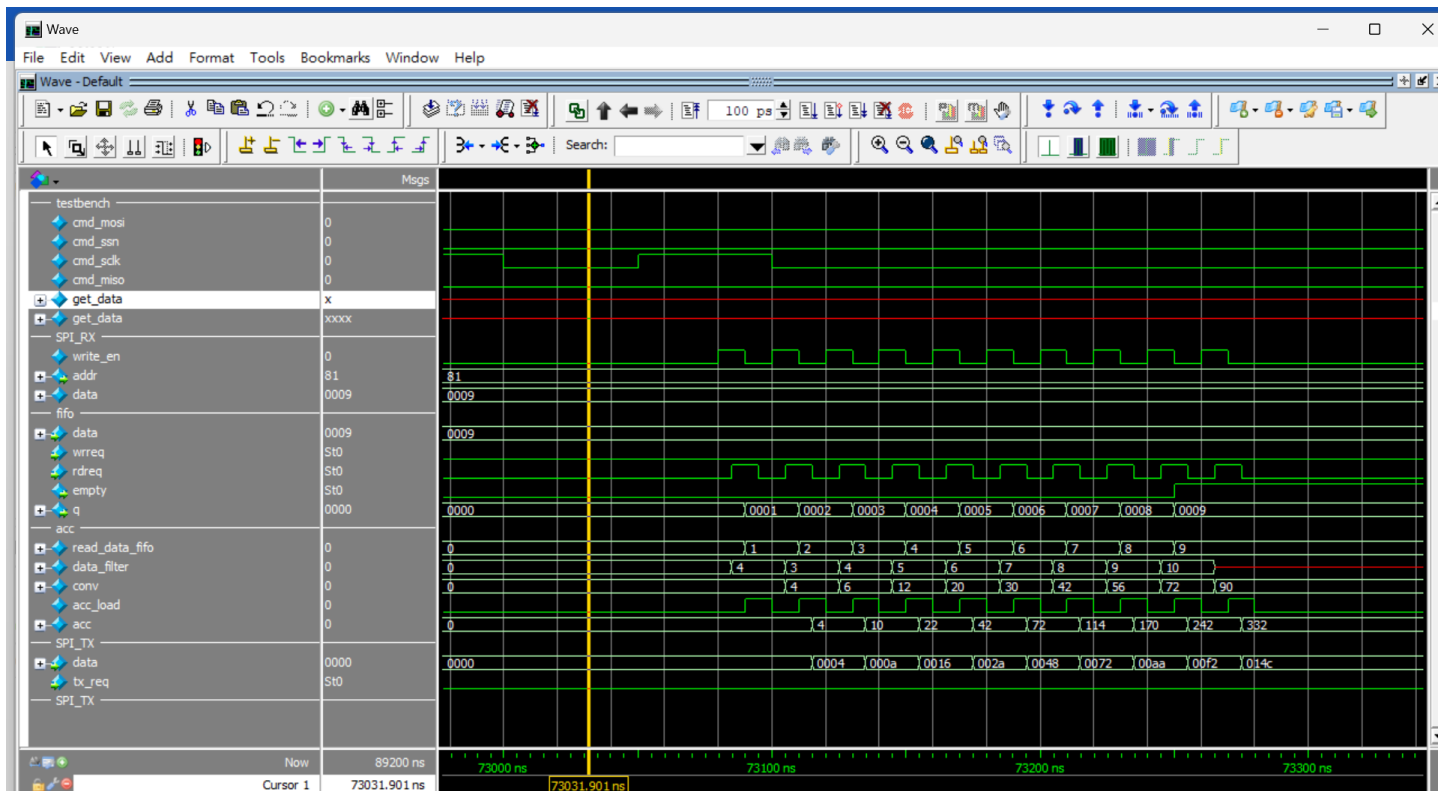
1 module neg_edge_detector
2 (
3     input clk,
4     input rst,
5     input signal,
6     output logic sig_neg
7 );
8
9     logic d_signal;
10    logic s_signal;
11
12    always_ff @(posedge clk) begin
13        if(rst) begin
14            s_signal <= 1'b1;
15            d_signal <= 1'b1;
16            sig_neg <= 1'b0;
17        end
18        else begin
19            {d_signal, s_signal} <= {s_signal, signal};
20            sig_neg <= d_signal & ~s_signal;
21        end
22    end
23
24 endmodule
```

```
檔案(F) 編輯(E) 搜尋(S) 檢視(V) 編碼(N) 語言(L) 設定(T) 工具(O) 巨集(M) 執行(R) 外掛(P) 視窗(W) ?
DE0_CV.sv spi_rx.sv spi_tx.sv testbench.sv register_file.sv regs.sv shift_register.sv counter.sv neg_edge_detector.sv pos_edge_detector.sv

1 module pos_edge_detector
2 (
3     input clk,
4     input rst,
5     input signal,
6     output logic sig_pos
7 );
8
9     logic d_signal;
10    logic s_signal;
11
12    always_ff @(posedge clk) begin
13        if(rst) begin
14            s_signal <= 1'b1;
15            d_signal <= 1'b1;
16            sig_pos <= 1'b0;
17        end
18        else begin
19            {d_signal, s_signal} <= {s_signal, signal};
20            sig_pos <= ~d_signal & s_signal;
21        end
22    end
23
24 endmodule
```

■ 模擬結果與結果說明：





## ■ 結論與心得：

這次作業大致上是基於上次 fifo\_sum 來更改，其中因為 empty 不太會用，會跟結果上差 1 所以就在這段寫了一個 counter 來當作 empty\_mine，但他可以設定初始值，解決差 1 的問題：

```

160 //wrreq_cnt
161 //用來判斷fifo中有幾個數，類似可控制初始值的empty
162 logic [15:0] wrreq_cnt;
163 always_ff @(negedge clk)
164 begin
165     if(rst) begin
166         wrreq_cnt <= 1;
167     end
168     else if(rdreq) begin
169         wrreq_cnt <= wrreq_cnt - 1;
170     end
171     else if(wrreq) begin
172         wrreq_cnt <= wrreq_cnt + 1;
173     end
174 end
175

```

這段是拿來調用 filter 的 address 的 counter:

```
176 //filter_addr
177
178 always_ff @(negedge clk)
179 begin
180     if(rst) begin
181         filter_addr <= 1;
182     end
183     else if(filter_addr_plus) begin
184         filter_addr <= filter_addr + 1;
185     end
186     if(filter_addr == 10) begin
187         filter_addr <= 1;
188     end
189 end
```

DUMMY2 是多一個狀態讓他做相加的時間:

```
270 CHECK_COMMAND:
271 begin
272     if(command) begin //read
273
274         //if(addr[7]) rdreq = 1;
275         //else read_en = 1;
276         read_en = 1;
277         ns = TX_REQ;
278     end
279     else //write
280         if(addr == 8'h81) begin //在write中遇到addr為81時 139行
281             filter_en = 1;
282             rdreq = 1; //要求從fifo中讀取值 ==> if(rdreq) begin
283
284             ns = DUMMY2; //做一個dummy等待加一次 conv <= conv + read_data_fifo;
285             // end
286
287         else begin
288             ns = RECEIVE_DATA;
289         end
290     end
291
292 DUMMY2:
293 begin
294     if(wrreq_cnt>0) begin //wrreq_cnt 146行
295         filter_addr_plus = 1;
296         acc_load=1;
297         ns = CHECK_COMMAND;
298     end
299     else begin
300         acc_load=1;
301         ns = FINISH;
302     end
303 end
```