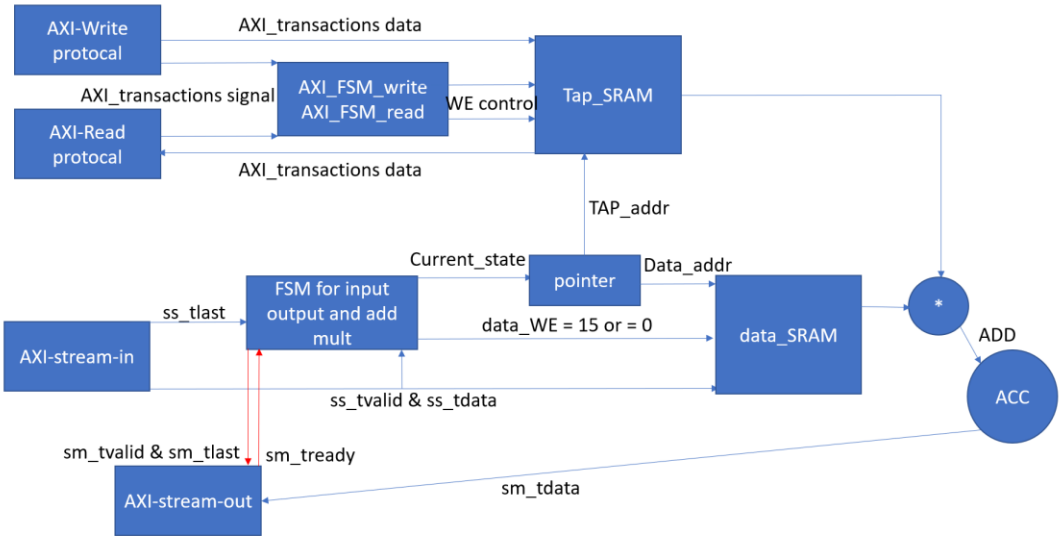
SoC Lab-FIR

312510220 張承宗

Block Diagram:



FSM 有三個：管 AXI_read 的

```
always @(*) begin
    case (c_state_read)
        READ_IDLE : n_state_read = ( arvalid ) ? READ_ADDR : READ_IDLE;
        READ_ADDR : n_state_read = (araddr == 0) ? READ_DATA_AP : READ_DATA;
        READ_DATA : n_state_read = ( rvalid ) ? READ_IDLE : READ_DATA;
        READ_DATA_AP : n_state_read = ( rvalid ) ? READ_IDLE : READ_DATA_AP;
        default: n_state_read = c_state_read;
    endcase
end

always @( posedge axis_clk or negedge axis_rst_n ) begin
    if(!axis_rst_n) c_state_read <= READ_IDLE;
    else            c_state_read <= n_state_read;
end

always @(*) begin
    arready = ( c_state_read == READ_ADDR ) ? 1 : 0;
    rvalid  = ( (c_state_read == READ_DATA || c_state_read == READ_DATA_AP) && rready == 1 ) ? 1 : 0;
    if(rvalid == 1)begin
        if( c_state_read == READ_DATA )     rdata = tap_Do;
        else if ( c_state_read == READ_DATA_AP )begin
            rdata = ap;
        end
    end
    else rdata = 0;
end
```

藉由 arvalid 和 rvalid 進行 state 的變動，中間會判斷 araddr 是不是來看 ap_idle 和 ap_done 的。

管 AXI_write 的：

```
always @(*) begin
    case (c_state_write)
        WRITE_IDLE    : n_state_write = ( awvalid ) ? WRITE_ADDR : WRITE_IDLE;
        WRITE_ADDR    : n_state_write = (awaddr == 0) ? WRITE_DATA_AP :WRITE_DATA;
        WRITE_DATA    : n_state_write = ( wvalid ) ? WRITE_IDLE : WRITE_DATA;
        WRITE_DATA_AP : n_state_write = ( wvalid ) ? WRITE_IDLE : WRITE_DATA_AP;
        default: n_state_write = c_state_write;
    endcase
end

always @( posedge axis_clk or negedge axis_rst_n ) begin
    if(!axis_rst_n) c_state_write <= WRITE_IDLE;
    else            c_state_write <= n_state_write;
end

always @(*) begin
    awready = ( c_state_write == WRITE_ADDR ) ? 1 : 0;
    wready  = ( c_state_write == WRITE_DATA || c_state_write == WRITE_DATA_AP) ? 1 : 0;
end
```

藉由 awvalid 和 wvalid 進行 state 的變動，中間會判斷 awaddr 是不是來寫 ap_start 的。

管累加和 stream I/O 的：

```
always @(*) begin
    case (c_state)
        RESET    : n_state = (counter == 10) ? IDLE : RESET;
        IDLE     : n_state = ( ss_tvalid == 1 ) ? INPUT_1 : IDLE;
        INPUT_1  : n_state = WAIT;
        WAIT     : n_state = ADD_1 ;
        ADD_1    : n_state = ADD_2 ;
        ADD_2    : n_state = ADD_3 ;
        ADD_3    : n_state = ADD_4 ;
        ADD_4    : n_state = ADD_5 ;
        ADD_5    : n_state = ADD_6 ;
        ADD_6    : n_state = ADD_7 ;
        ADD_7    : n_state = ADD_8 ;
        ADD_8    : n_state = ADD_9 ;
        ADD_9    : n_state = ADD_10;
        ADD_10   : n_state = ADD_11;
        ADD_11   : n_state = OUTPUT;
        OUTPUT   : n_state = (done == 1 && sm_tready == 1) ? RESET : (done == 0 && sm_tready == 1) ? INPUT_1 : OUTPUT;
        default: n_state = c_state;
    endcase
end

always @(posedge axis_clk or negedge axis_rst_n) begin
    if(!axis_rst_n) done <= 0;
    else            done <= (c_state == IDLE && n_state == INPUT_1) ? 0 : (ss_tlast == 1 && ss_tready == 1) ? 1 : done;
end

always @( posedge axis_clk or negedge axis_rst_n ) begin
    if(!axis_rst_n) c_state <= RESET;
    else            c_state <= n_state;
end
```

簡單的 11 連續乘加的 state 和 OUTPUT 時判斷是不是最後一個 data 的 MUX 來進行迴圈。

Ap signal 則是按照 STATE 和 wdata 來決定:

```
//ap signal
always @( posedge axis_clk or negedge axis_rst_n ) begin
    if(!axis_rst_n) begin
       ap[0] <= 0;
       ap[1] <= 0;
       ap[2] <= 1;
    end
    else begin
        ap[0] <= (c_state_write == WRITE_DATA_AP && wvalid == 1) ? wdata : ( c_state == INPUT_1 ) ? 0 : ap[0];
        ap[1] <= (done == 1 && n_state == RESET) ? 1 : (c_state_write == WRITE_DATA_AP && wvalid == 1) ? 0 : ap [1];
        ap[2] <= (done == 1 && n_state == RESET) ? 1 : (c_state_write == WRITE_DATA_AP && wvalid == 1) ? 0 : ap [2];
    end
end
```

Done 這個額外控制訊號是在 stream in 拉 last 的時候就拉起來了，那等到我 stream out 也丟 last 然後要 reset data_sram 的時候，我就會把 ap_done 和 ap_idle 給拉起來

Tap_BRAM access：

```
always @( posedge axis_clk or negedge axis_rst_n ) begin
    if(!axis_rst_n)begin
       tap_WE <= 0;
       tap_EN <= 0;
       tap_Di <= 0;
       tap_A  <= 0;
       addr_save <= 0;
    end
    else begin
       tap_EN    <= 1;
       addr_save <= ( awvalid )  ? awaddr : addr_save;
       tap_A     <= ( wvalid && addr_save >= 32 ) ? addr_save - 'h20 : ( arvalid && c_state == RESET || c_state == IDLE) ? araddr - 'h20 : (n_state >= 12 && n_state <= 2
       tap_Di    <= ( wvalid ) ? wdata  : 0;
       tap_WE    <= ( wvalid && addr_save >= 32 ) ? 15 : 0;
    end
end
```

用簡單的 counter 和 axi_lite 訊號決定現在要寫入那裡或讀出哪裡，讀出通常就是直接用 state 直接減去常數就會是我要的 addr。

Data_bram access:

```
235   always @( posedge axis_clk or negedge axis_rst_n ) begin
236       if(!axis_rst_n)begin
237           data_A  <= 0;
238           data_EN <= 0;
239           data_Di <= 0;
240           counter <= 0;
241           data_WE <= 0;
242       end
243       else begin
244           counter <= ( c_state == RESET ) ? counter + 1 : counter;
245           data_A  <= ( c_state == RESET ) ? counter<<2 : n_pointer<<2;
246           data_EN <= 1;
247           data_Di <= ( c_state == RESET ) ? 0 : (c_state == INPUT_1) ? ss_tdata : 0;
248           data_WE <= ( c_state == INPUT_1 || c_state == RESET ) ? 15 : 0;
249       end
250   end
251
252   //axi stream input
253   always @(*) begin
254       ss_tready = (c_state == INPUT_1) ? 1 : 0;
255   end
256
257   //pointer
258   always @(*) begin
259       n_pointer = pointer;
260       if(c_state == OUTPUT && n_state == INPUT_1)begin
261           if (pointer == 10) n_pointer = 0;
262           else n_pointer = pointer + 1;
263       end
264       else if (c_state >= 12 && c_state <= 22)begin
265           if (pointer == 0) n_pointer = 10;
266           else              n_pointer = pointer - 1;
267       end
268   end
```

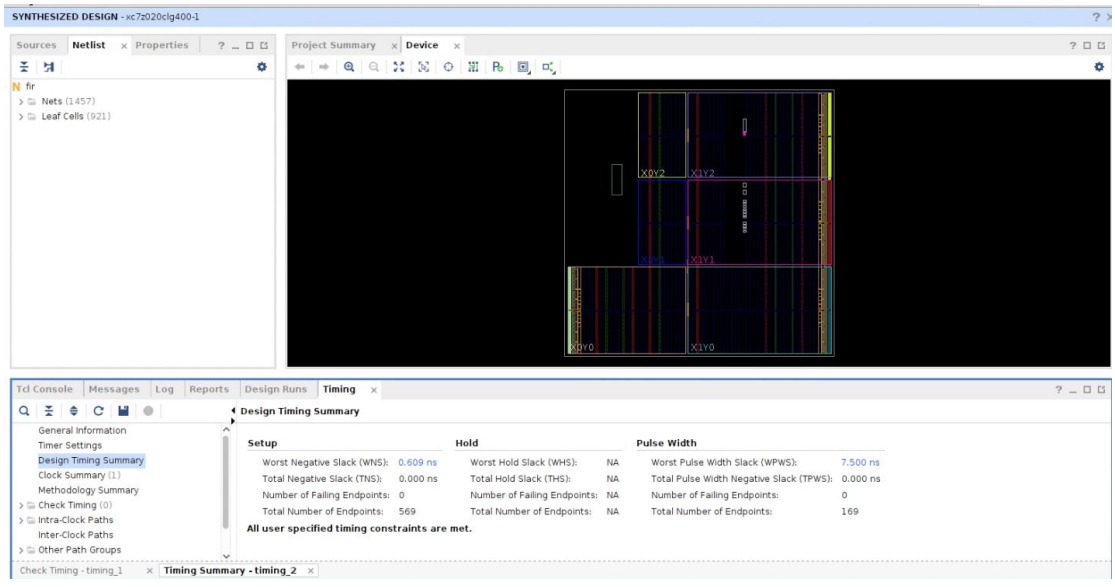
每次在 input state 時根據 pointer 決定要寫入的位置，pointer 會不斷在連加時減一來指定相對應的 bram data，在進 output state 時會走完一圈。
但是接下來 input_state 的位子要是上個起點下一格，所以 pointer 會在 output state 時再 + 1。

Resource Usage: Including FF, LUT, BRAM:

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	336	0	0	53200	0.63
LUT as Logic	336	0	0	53200	0.63
LUT as Memory	0	0	0	17400	0.00
Slice Registers	168	0	0	106400	0.16
Register as Flip Flop	168	0	0	106400	0.16
Register as Latch	0	0	0	106400	0.00
F7 Muxes	0	0	0	26600	0.00
F8 Muxes	0	0	0	13300	0.00

FF:
FF 用了大概 168 個，主要是我在累加時把 data length 開到 32 bit，有點浪費。FSM 的部分我也是有點宣告太大，但這次的 spec 沒有特別要求這部分，如果很努力去省 FF 的話，有機會省到約 100 個左右。
LUT:
336 個
BRAM:
因為沒有在 fir.v 中宣告，所以這裡並沒有合成。

Timing Report:



使用的是 16ns 的 cycle time，input 和 ouput delay 都是 5ns，如果不是用 pynq 的板子的話據說能夠更快。

Longest path, slack Longest path:

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock
Path 1	0.609	3	4	35	rready	rdata[12]	5.356	3.756	1.599	16.0	axis_clk	axis_clk
Path 2	0.609	3	4	35	rready	rdata[14]	5.356	3.756	1.599	16.0	axis_clk	axis_clk
Path 3	0.609	3	4	35	rready	rdata[16]	5.356	3.756	1.599	16.0	axis_clk	axis_clk
Path 4	0.609	3	4	35	rready	rdata[18]	5.356	3.756	1.599	16.0	axis_clk	axis_clk
Path 5	0.609	3	4	35	rready	rdata[20]	5.356	3.756	1.599	16.0	axis_clk	axis_clk
Path 6	0.609	3	4	35	rready	rdata[22]	5.356	3.756	1.599	16.0	axis_clk	axis_clk
Path 7	0.609	3	4	35	rready	rdata[24]	5.356	3.756	1.599	16.0	axis_clk	axis_clk
Path 8	0.609	3	4	35	rready	rdata[26]	5.356	3.756	1.599	16.0	axis_clk	axis_clk
Path 9	0.609	3	4	35	rready	rdata[28]	5.356	3.756	1.599	16.0	axis_clk	axis_clk
Path 10	0.609	3	4	3	tap_Do[2]	rdata[2]	5.356	3.756	1.599	16.0	axis_clk	axis_clk

可以看到是 rready 輸入到 rdata 輸出的部分，因為我使用的是 rready 一輸入我就 rvalid 拉起來輸出這種 I/O 模式，上課有提到這種依賴 input 直接 comb 過去的電路其實不是很好，但我這邊寫起來比較直觀所以這樣使用，事實證明教授提到的缺點是存在的。

Waveform

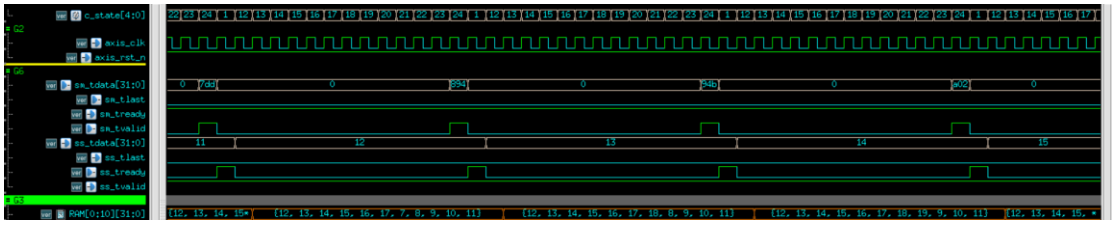
Sent in Data + tap_SRAM write in

Axi-LITE Coef Check & TAP_SRAM read out:

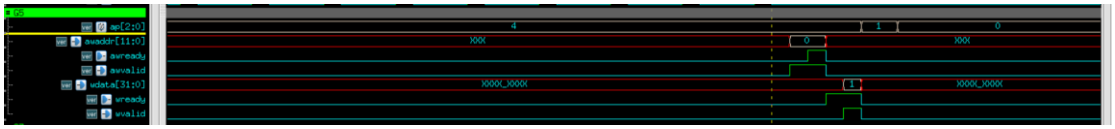
Data_stream_in & Data_stream_out with data_sram access control :

只要 data_WE = 15 就是要寫入新的 data 的時候，我們放棄用一直慢慢讀出寫入的 shift SRAM，決定用 index 來指 data_SRAM 的內部 data。

因為 vivado 好像沒辦法開自建的 FF 波型，所以這邊用 verdi 來顯示 FSM 的情況: 基本上可以和 vivado 的 waveform 進行對照。



Ap_start:當 check 完 coef 的寫入後，wdata 寫入 ap start，我的 design 將 ap start 拉起來一 cycle 並開始計算。



Ap_done & ap_idle : 當輸出最後一個 data 時拉起來。

INPUT_stream_last:

OUTPUT_stream_last: