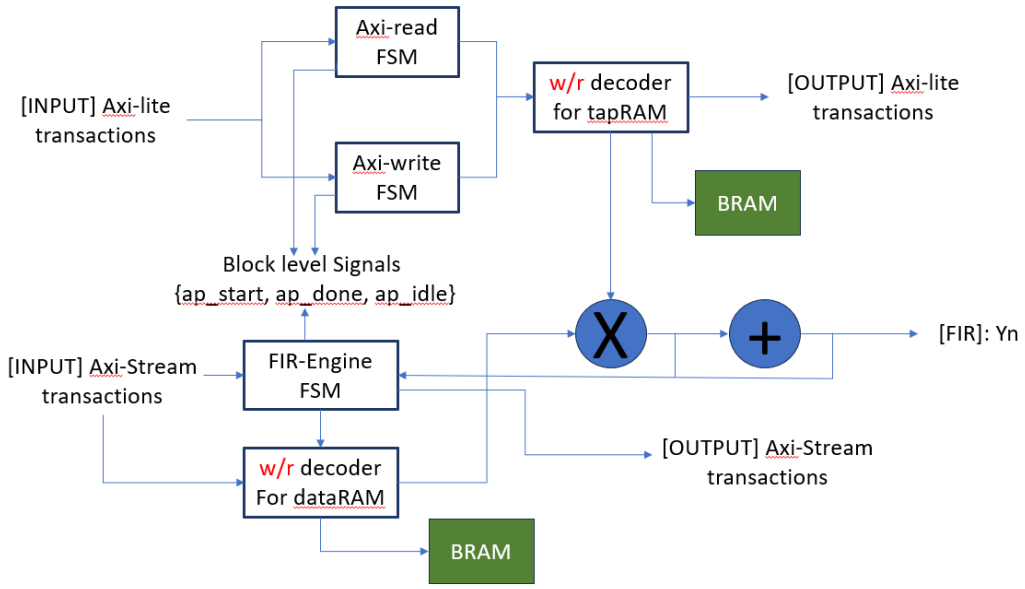
# SoC-Lab3 FIR verilog simulation report

312510140 何律明

## Block Diagram and datapath:




## How to receive data-in and tap parameters and place into SRAM?

我是使用 axi-lite-write 自己的 FSM 去做 tapRAM 的控制信號。

```
always @(*) begin
    case (axi_write_state)
        AXI_WRITE_IDLE: n_axi_write_state = (awvalid == 1) ? AXI_WRITE_ADDR : AXI_WRITE_IDLE;
        AXI_WRITE_ADDR: n_axi_write_state = AXI_WRITE_DATA;
        AXI_WRITE_DATA: n_axi_write_state = (wvalid == 1) ? AXI_WRITE_IDLE : AXI_WRITE_DATA;
        default: n_axi_write_state = AXI_WRITE_IDLE;
    endcase
end

always @(posedge axis_clk or negedge axis_rst_n) begin
    if (!axis_rst_n) begin
        axi_write_state  <= AXI_WRITE_IDLE;
        awready <= 0;
        wready  <= 0;
    end else begin
        axi_write_state  <= n_axi_write_state;
        awready <= (n_axi_write_state == AXI_WRITE_ADDR) ? 1 : 0;
        wready  <= (n_axi_write_state == AXI_WRITE_DATA) ? 1 : 0;
    end
end
```

可以看到我有 3 個 state。

當 input 收到 awvalid 代表 host 發起 transaction，並進入 handshake addr 的 state，下一個 cycle 直接進入接 data 的 state，直到 data handshake 完成後回到 idle。

```
// tap_WE, tap_Di
always @(*) begin
    if (wready == 1 && wvalid == 1 && tapWriteAddr != 'h00 && tapWriteAddr != 'h10) begin
        tap_WE = 4'b1111;
        tap_Di = wdata;
    end
    else begin
        tap_WE = 0;
        tap_Di = 0;
    end
end
```

Access BRAM 的部分，我是在 data handshake 時候把 EN & WE pull high。如此可以在此 cycle access BRAM，順便從 wdata 直接吐 data 到 BRAM 的 data-in port。

## How to access shiftram and tapRAM to do computation?

這部分我是跟隨著 FIR engine 的 state 去對應的 BRAM 拿取資料。

```
case (fir_state)
    FIR_IDLE:   n_fir_state = (wready == 1 && wvalid == 1 && wdata == 1) ?  DATA_RST : FIR_IDLE;
    DATA_RST:   n_fir_state = (dataRam_rst_cnt == 10) ? FIR_WAIT : DATA_RST;
    FIR_WAIT:   n_fir_state = (ss_tvalid == 1) ? FIR_SSIN : FIR_WAIT;
    FIR_SSIN:   n_fir_state = (ss_tready == 1) ? FIR_STOR : FIR_SSIN;
    FIR_STOR:   n_fir_state = FIR_S0;
    FIR_S0:     n_fir_state = FIR_S1;
    FIR_S1:     n_fir_state = FIR_S2;
    FIR_S2:     n_fir_state = FIR_S3;
    FIR_S3:     n_fir_state = FIR_S4;
    FIR_S4:     n_fir_state = FIR_S5;
    FIR_S5:     n_fir_state = FIR_S6;
    FIR_S6:     n_fir_state = FIR_S7;
    FIR_S7:     n_fir_state = FIR_S8;
    FIR_S8:     n_fir_state = FIR_S9;
    FIR_S9:     n_fir_state = FIR_SA;
    FIR_SA:     n_fir_state = FIR_OUT;
    FIR_OUT: begin
        if      (last_flg == 1)     n_fir_state = FIR_IDLE; // reset
        else if (sm_tready == 1)    n_fir_state = FIR_WAIT; // wait for next Stream-In data
        else                        n_fir_state = FIR_OUT;  // wait for handshake
    end
```

其中的 FIR_S0 ~ FIR_SA 則是 11 個 cycle 各自對應的 state，每一個 State 去兩個 bram 各自拿出自己的資料並做累加到 Yn。

```
// Accumulator and Multiplier (1 adder & 1 multiplier for FIR)
always @(posedge axis_clk or negedge axis_rst_n) begin
    if (!axis_rst_n)    Yn <= 0;
    else begin
        if      (fir_state == FIR_SSIN)                         Yn <= 0;            // reset accumulator
        else if (FIR_S0 <= n_fir_state && n_fir_state <= FIR_SA)    Yn <= Yn + (Xn * Hn);   // accumulate Xn * Hn
        else                                                    Yn <= Yn;
    end
end
```

累加的部分當然也可以把* + pipline 到不同 Cycle 做，這樣的話 frequency 可以跑得更快。

## How ap_done is generated?

```
// ap_start
always @(*) begin
    if      (tapWriteAddr == 'h00 && wready == 1 && wvalid == 1 && wdata == 1)  n_ap_start = wdata;     // Host program ap_start = 1
    else if (fir_state == FIR_SSIN)                                             n_ap_start = 0;         // Reset to 0, when 1st AXI-Stream handshake
    else                                                                        n_ap_start = ap_start;
end
```

在 axi-write && addr = 0 && wdata = 1 的時候到表 host/TB 對 ap_start 做 Assert。這個時候我們的 n_ap_start 就是 1 在下一個 posedge 的時候送進 ap_start 這個 FF。此外在 1st AXI-Stream data handshake 的時候 reset ap_start，其餘的時候 ap_start 則保持原值。

## Resource Usage: Including FF, LUT, BRAM:

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice LUTs* | 318 | 0 | 0 | 53200 | 0.60 |
| LUT as Logic | 318 | 0 | 0 | 53200 | 0.60 |
| LUT as Memory | 0 | 0 | 0 | 17400 | 0.00 |
| Slice Registers | 131 | 0 | 0 | 106400 | 0.12 |
| Register as Flip Flop | 131 | 0 | 0 | 106400 | 0.12 |
| 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:**

可以看到 syn report 寫到我的 FF 用了大概 **131** 個，這是因為我有多存一個 32bit 的 data-length，但其實這個 FF 是可以不必要的，因為 axi-stream 有 tlast 的機制。此外，在進行 Yn = SUM(Hn, Xn-i)的時候我也把 Hn & Xn 開到 32bit，這也是不必要的，因為累加結果並不會超過 32bit，其實對 FF 是相當浪費的。所以說其實我估計可能 7~80 個 FF 就可以完成這個 Design。

**LUT:**

總共 318 個

**BRAM:**

顯示在 design(fir.v)當中並無使用，確實是這樣我們並沒有在 fir.v 當中宣告 bram11。

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Block RAM Tile | 0 | 0 | 0 | 140 | 0.00 |
| RAMB36/FIFO* | 0 | 0 | 0 | 140 | 0.00 |
| RAMB18 | 0 | 0 | 0 | 280 | 0.00 |

## Timing Report:

```
From Clock:  axis_clk
  To Clock:  axis_clk

Setup :            0  Failing Endpoints,  Worst Slack        0.994ns,  Total Violation        0.000ns
Hold  :           NA  Failing Endpoints,  Worst Slack           NA  ,  Total Violation           NA
PW    :            0  Failing Endpoints,  Worst Slack        9.500ns,  Total Violation        0.000ns
```

都有 MET。

**Frequency**

Timing constraints 的部分我是下 period = 20，input & output delay 都是 5，是在 pynq-z2 去做合成。但我發現在其他 FPGA 上我發現可以跑出更快的結果。

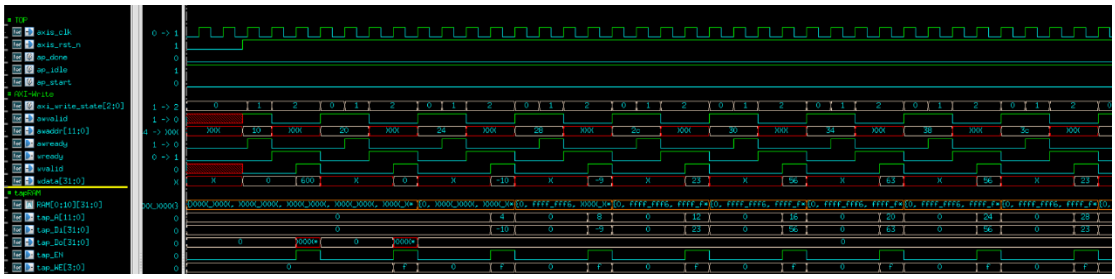
**Longest path, slack**

Longest path: 是判斷 n_fir_state 的地方。因為我的 FIR ENGINE state 很多，所以說會造成 mux delay 很久，此外，我也使用了很多 n_fir_state 當作 BRAM addr 的控制信號，這樣導致 delay 很長。但這些是可以用其他的 FF 當作 logic control 的。這樣我相信可以減少不少的 delay。

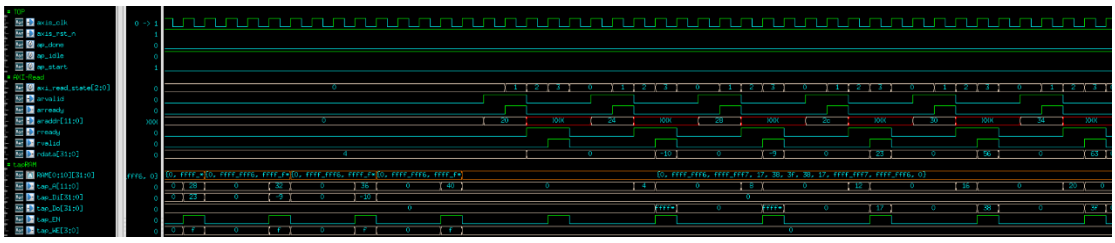
Slack: **+0.994**

## Waveform:

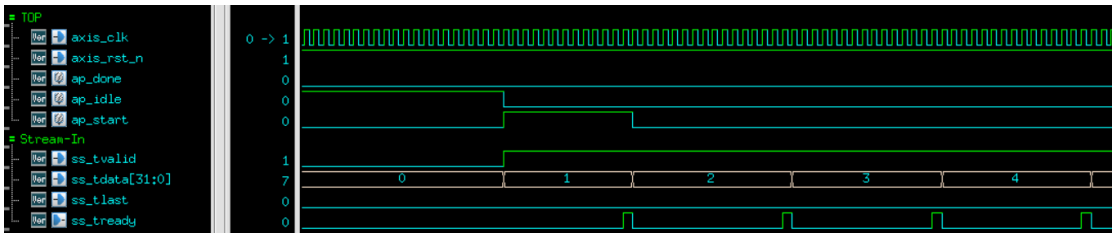
**Coefficient program & RAM access control**



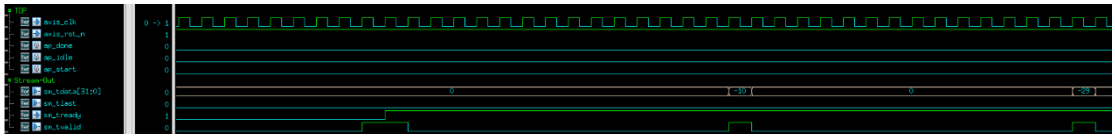
**Read back**



**Data-in Stream-in**




**Data-out Stream-out**


TOP
axis_clk
axis_rst_n
ap_done
ap_idle
ap_start
Stream-Out
sm_tdata[31:0]
sm_tlast
sm_tready
sm_tvalid
0 -> 1
1
0
0
0
0
0
1
0
0
-10
0
-29