Lab3: FIR Design
SOC Design

EE25 110022124 張哲熙
National Tsing Hua University

March 22, 2025

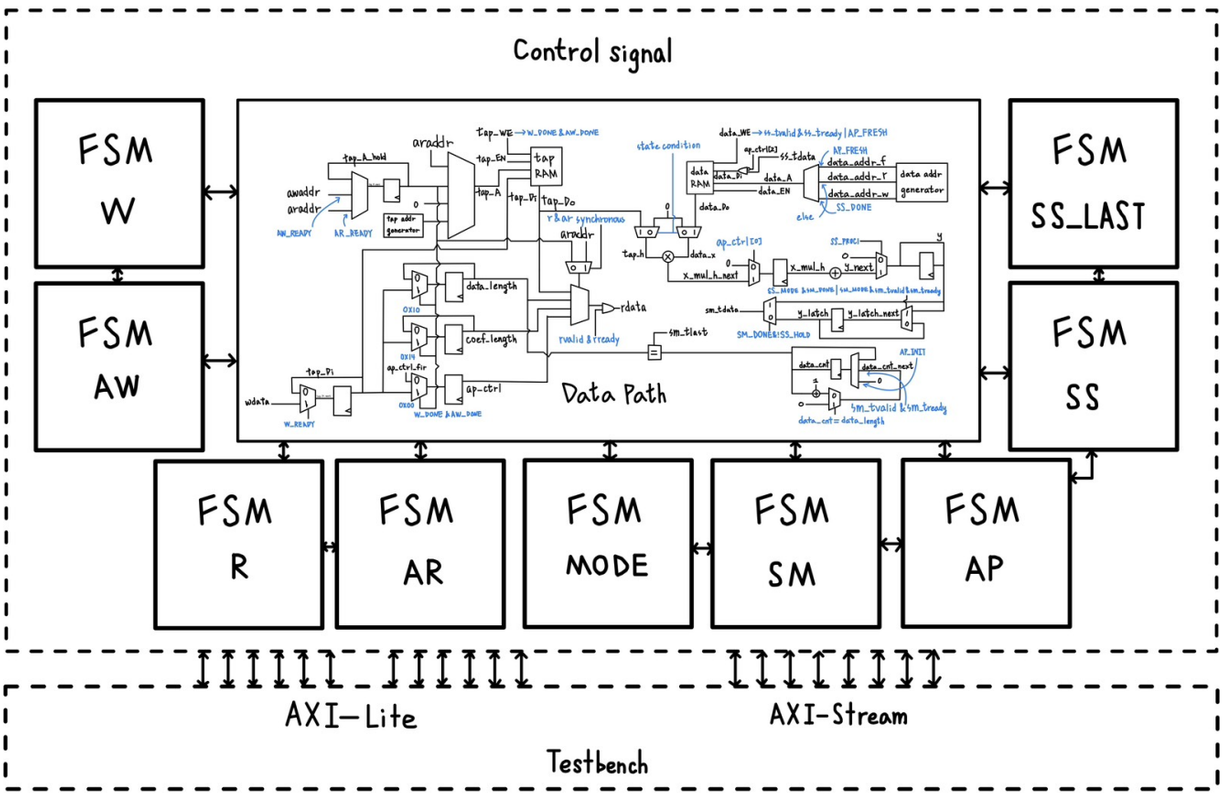
Contents

Introduction

In this Lab, we need to implement finite impulse response.

$$y[n] = \sum_{i=1}^{N} h[i]x[n-i] \tag{1}$$

We have three-part, `tap_RAM`, `data_RAM` and `fir`. The testbench(or CPU) communicates with the `fir` by **Advanced eXtensible Interface(AXI Lite and AXI Stream)**. AXI Lite will be used to access `tap_RAM` and `configuration register` in `fir`. AXI Stream will be used to access `data_RAM` and output the calculation result of `fir` to the testbench.
The data path of my design is shown in the following figure, and FSM determines the control signal.



AXI Lite Interface

In AXI Lite, we need to implement the interface, so we can access `tap_RAM` and `configuration register(data_length, coef_length, ap_ctrl)` and get the control signal to control the data-path.

We need to consider three cases, first, the aw-signal(write address) is later than the w-signal(wri -te), so we need to hold the address(`awaddr`) until the write data(`wdata`) comes. Second, the w-signal is later than the aw-signal, we need to hold the write data until the address comes. Last, the r-signal is later than the ar-signal, so we need to hold the address until the read data comes.

AXI Lite portecle

```
//-----AXI Lite write-----//
    output  wire                     awready, //control signal
    output  wire                     wready,  //control signal
    input   wire                     awvalid,
    input   wire [(pADDR_WIDTH-1):0] awaddr,
    input   wire                     wvalid,
    input   wire [(pDATA_WIDTH-1):0] wdata,
//-----AXI Lite read-----//
    output  wire                     arready, //control signal
    input   wire                     rready,
    input   wire                     arvalid,
    input   wire [(pADDR_WIDTH-1):0] araddr,
    output  wire                     rvalid,  //control signal
    output  reg  [(pDATA_WIDTH-1):0] data,
//-----control signal-----//
    assign wready = (w_state == W_READY)? 1 : 0;
    assign awready = (aw_state == AW_READY)? 1 : 0;
    assign arready = (ar_state == AR_READY)? 1 : 0;
    assign rvalid = (r_state == R_VALID)? 1 : 0;
```

AXI Lite waveform

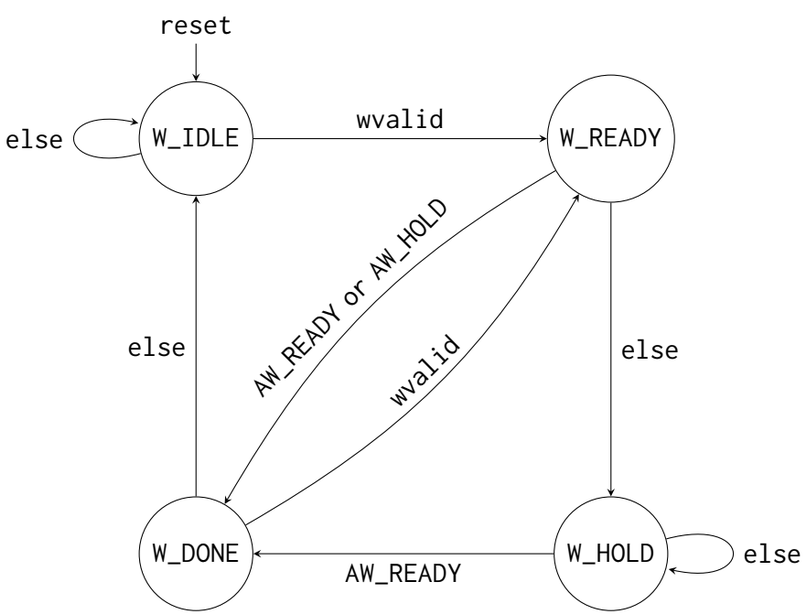
```
1 //case1: aw-signal delay, store the wdata in the register until the awaddr comes.
2 wvalid   ____/‾‾‾‾‾‾‾‾\_____________________________________________________
3 wready   ________/‾‾‾‾‾‾‾‾\_____________________________________________________
4 wdata    ____________|  data  |_____________________________________________________
5 awvalid  ____________________________/‾‾‾‾‾‾‾‾\_____________________________
6 awready  ____________________________________/‾‾‾‾‾‾‾‾\_____________________
7 awaddr   ____________________________________|  addr  |_____________________
8 wstate   ____ IDLE ____|__ READY __|_____ HOLD _____|__ DONE __|
9 awstate  ____ IDLE ____|__ READY __|__ DONE __|
```

AXI Lite waveform

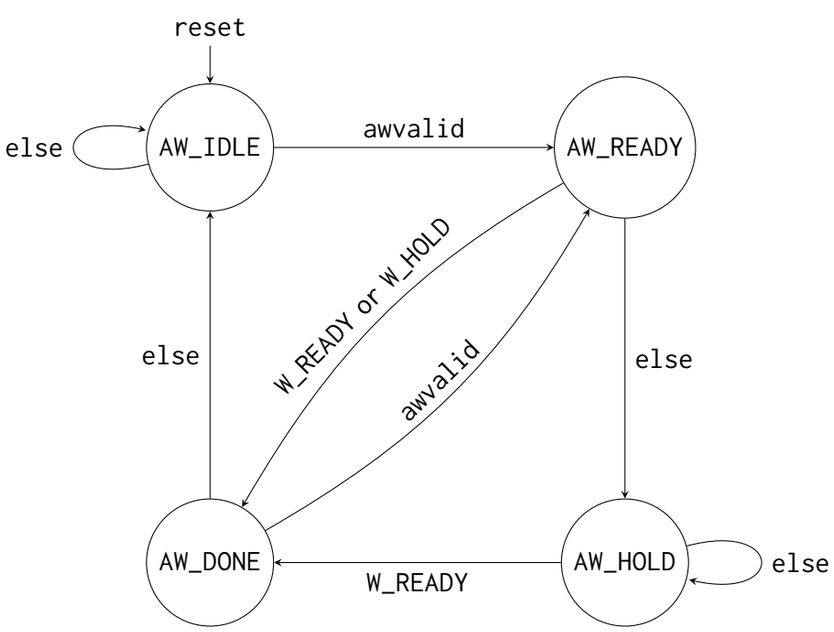
```
1 //case2: w-signal delay, store the awaddr in the register until the wdata comes
2 wvalid   ____________________________/‾‾‾‾‾‾‾‾\_____________________________
3 wready   ____________________________________/‾‾‾‾‾‾‾‾\_____________________
4 awvalid  ____/‾‾‾‾‾‾‾‾\_____________________________________________________
5 awready  ________/‾‾‾‾‾‾‾‾\_____________________________________________________
6 wdata    ____________________________________|  data  |_____________________
7 awaddr   ____________|  addr  |_____________________________________________________
8 wstate   ____ IDLE ____|__ READY __|__ DONE __|
9 awstate  ____ IDLE ____|__ READY __|_____ HOLD _____|__ DONE __|
```

Now, we focus on case1 and case2, since their logic is symmetry, the FSM of w-signal and aw-signal will be similar. When the earlier signal is `READY` then it will switch to the `HOLD`, then store the information into the register. Until the another signal is `READY`, both `w_state` and `aw_state` will switch to `DONE`, open the `tap_WE`.

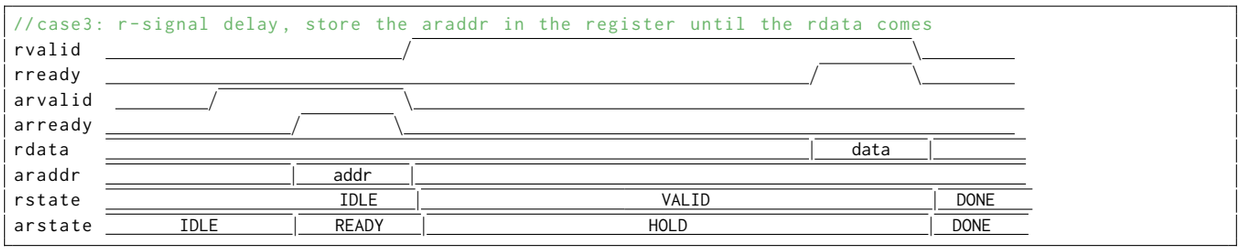
FSM of write:



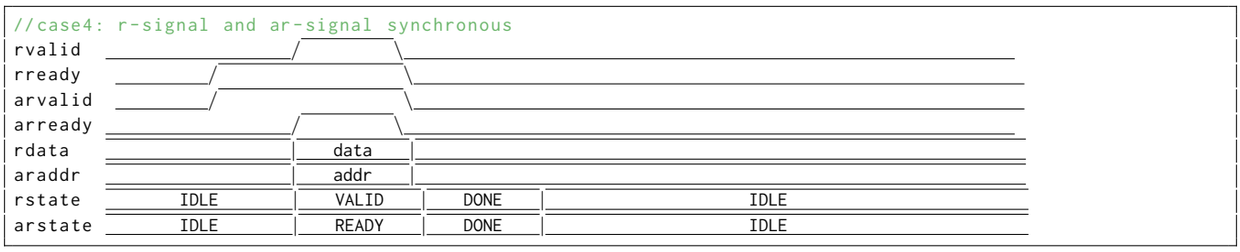
FSM of write address:



AXI Lite waveform



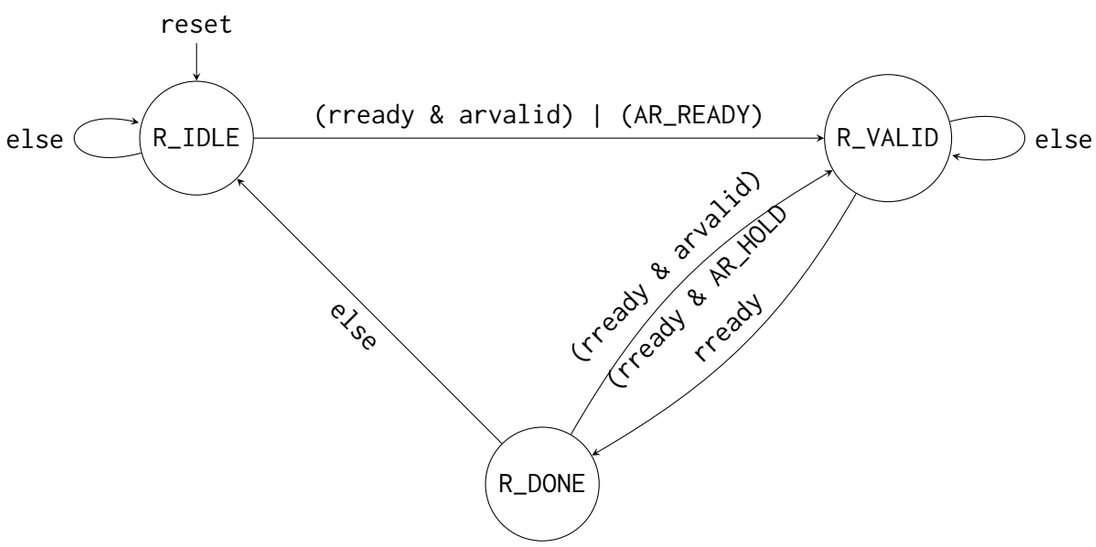
AXI Lite waveform



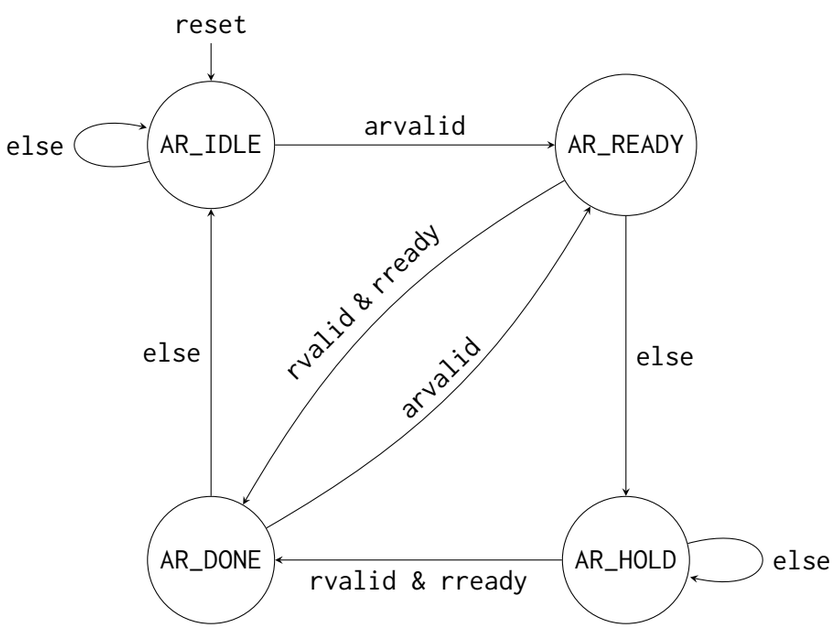
For case3, the FSM of ar-signal is similar too, when the ar-signal is `READY`, it will check whether r-signal is coming, if not, switch to `HOLD` and wait for r-signal. For r-signal, if `ar_state` is `READY`, it will which to `R_VALID`, which means that it is ready for outgoing the read data. When `rready`(from testbench) pulls up, the `rdata` passes to the testbench.

Besides, I have do some special handling on synchronous r-signal and ar-signal, since `araddr to tap_A` and `tap_RAM` read out have 1 clock delay in my design(total of 2 clock), but the AXI interface stipulates that if `rready & rvalid`, the `rdata` needs to be read out immediately. I solve this problem by detecting whether the r-signal and ar-signal are synchronous, if yes, the `araddr` will directly go to `tap_A`, don't need to wait for store register `tap_A_hold`.

FSM of read:



FSM of read address:



AXI Stream Interface

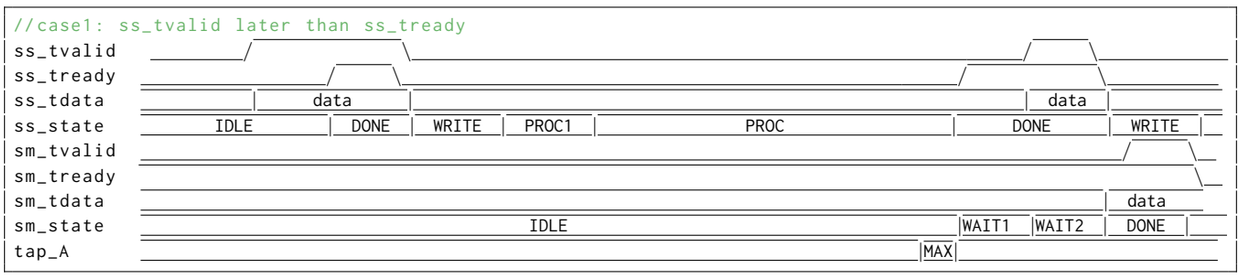
In AXI Stream, we need to implement the interface, so we can access `data_RAM`, and output the result from `fir` to `testbench`.

We need to consider two special cases, first, the `ss_tvalid`(from `testbench`) is later than `ss_tready`(control by Stream Slave FSM). Another one is that the sm-signal is later than the ss-signal. The operation mode is determined by FSM of `SS_MODE` and `SM_MODE`, when the master(sm-signal) is lower than slave(ss-signal), the `fir` operates in `SM_MODE`, else, operate in `SS_MODE`.

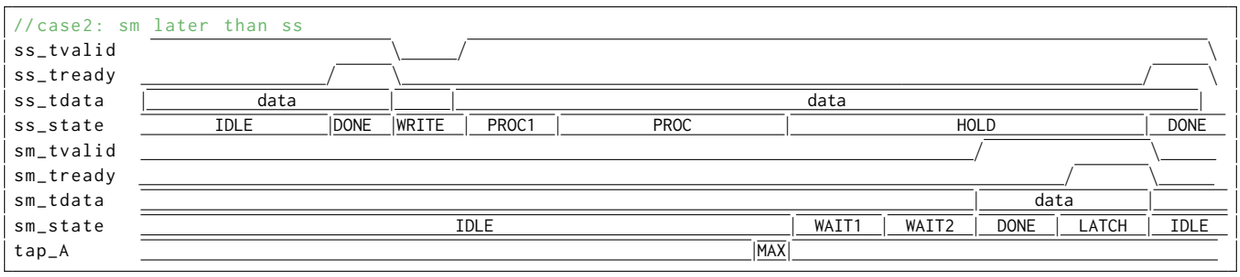
AXI Stream portecle

```
//-----AXI Stream, input data Xn-----//
    input   wire                        ss_tvalid,
    input   wire [(pDATA_WIDTH-1):0] ss_tdata,
    input   wire                        ss_tlast,
    output  reg                         ss_tready, //control signal
//-----AXI Stream, output data Yn-----//
    input   wire                        sm_tready,
    output  wire                        sm_tvalid, //control signal
    output  wire [(pDATA_WIDTH-1):0] sm_tdata,
    output  wire                        sm_tlast,
//-----control signal-----//
    assign ss_tready = (ss_state == SS_DONE)? 1 : 0;
    assign sm_tvalid = (sm_state == SM_DONE | sm_state == SM_LATCH)? 1 : 0;
```

AXI Stream waveform



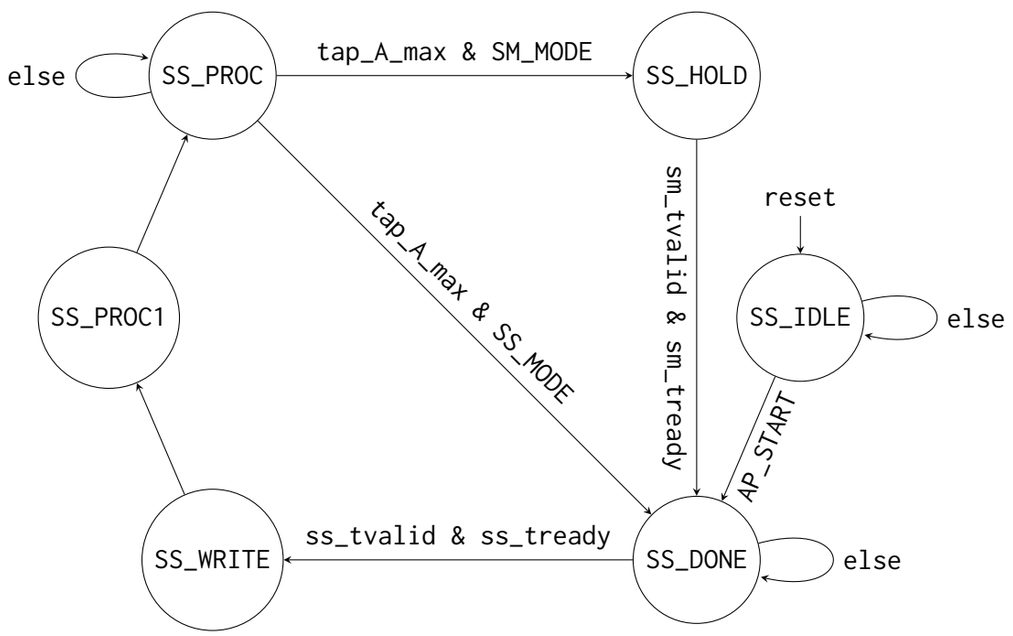
AXI Stream waveform



First, consider case1, when the convolution is complete(`tap_A==tap_A_max`), `ss_ready` should pull up until `ss_valid` comes, this control signal is generated by FSM of SS.

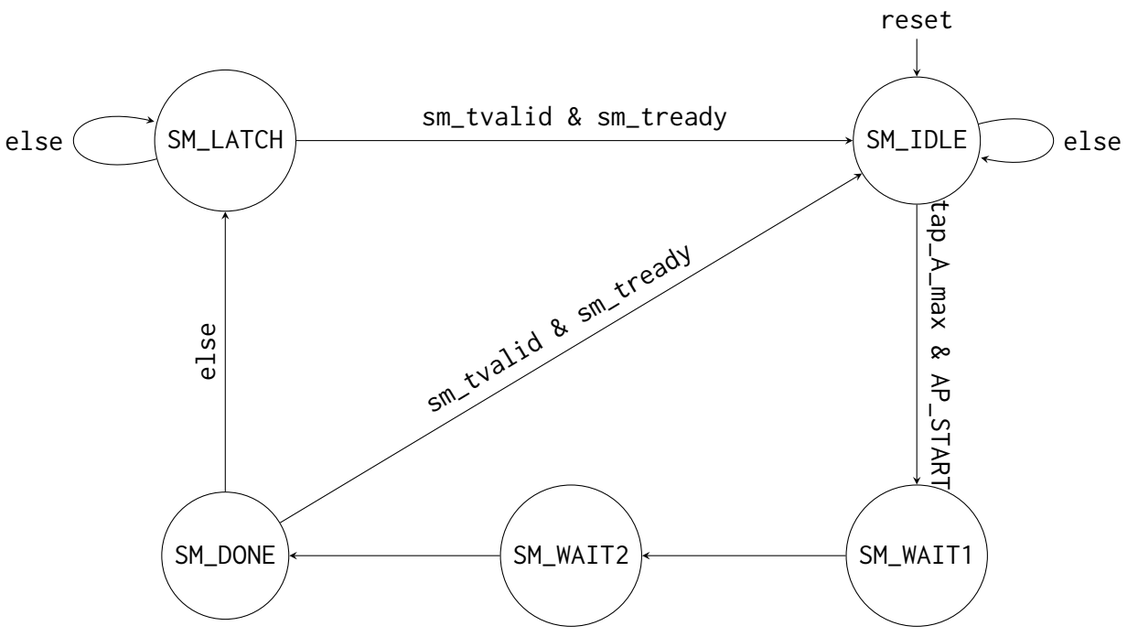
Besides, consider case2, when `SM_DONE` but `sm_tready != 1`, we need to latch the output by a register(`y_latch`) until `sm_tready` comes from `testbench`.

FSM of Stream Slave:



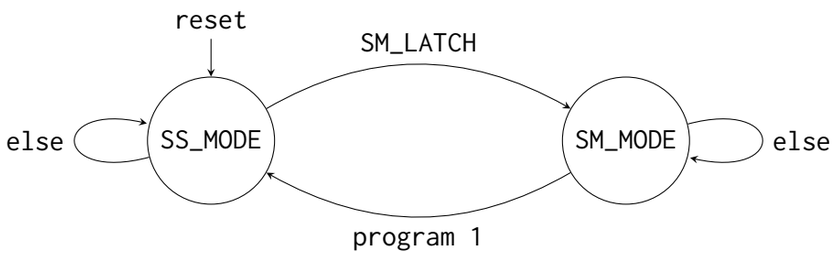
FSM of Stream Master:

Since addition and multiplication totally use two clock cycles, we need to wait for two clock cycles before we receive the result.



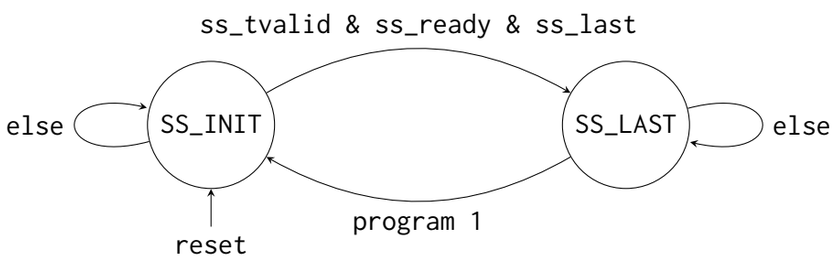
FSM of operation mode:

If sm-signal is later than ss-signal, the `fir` will operate in `SM_MODE`. That is, when the calculation of convolution is complete, `SS_PROC` switch to `SS_HOLD(ss_tvalid=0)` until the master receives the result`(sm_tvalid & sm_tready)`.



FSM of Stream Slave Last:

This FSM is used to fix the problem that when the first dataset didn't finish, the AXI Stream starts to receive the data from the second dataset, this will cause us to operate less on some data. Therefore, after receiving the first dataset data, we need to wait until `program 1` to start receiving the second data.

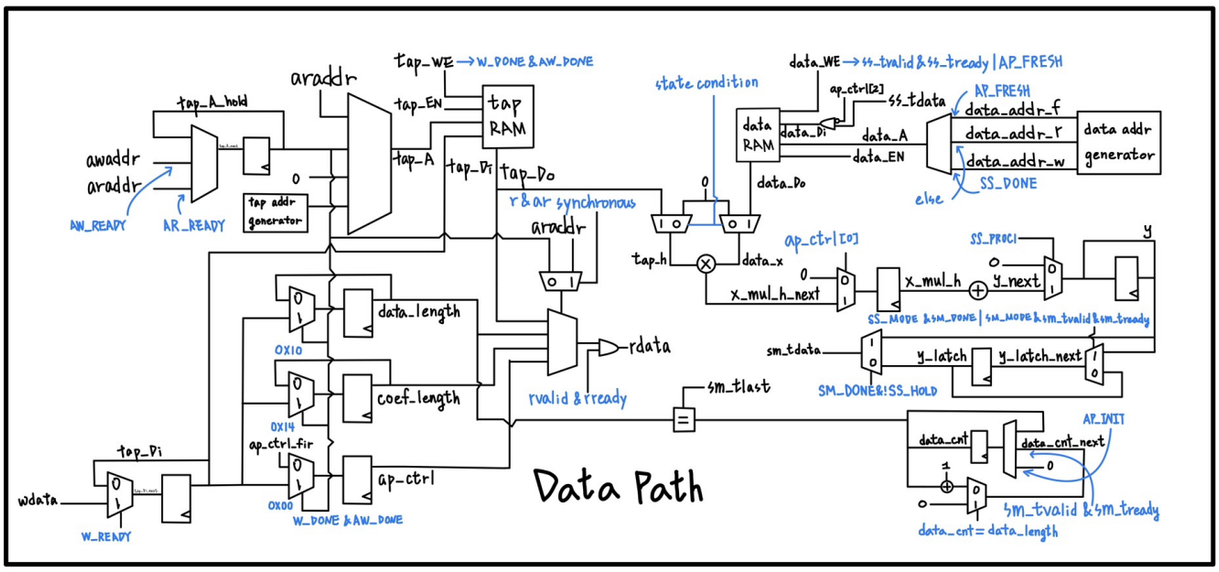


revise `ss_tready` control

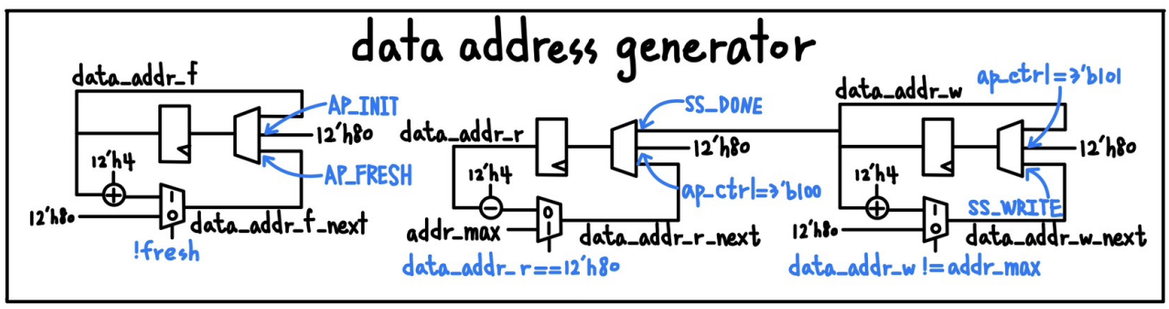
```
//-----control ss_tready by ss FSM-----//
always @(*) begin
  if (ss_last_state == SS_LAST) begin
    ss_tready = 0; //do not pull up if ss last until ap state program 1
  end else if (ss_state == SS_DONE) begin
    ss_tready = 1;
  end else begin
    ss_tready = 0;
  end
end
```

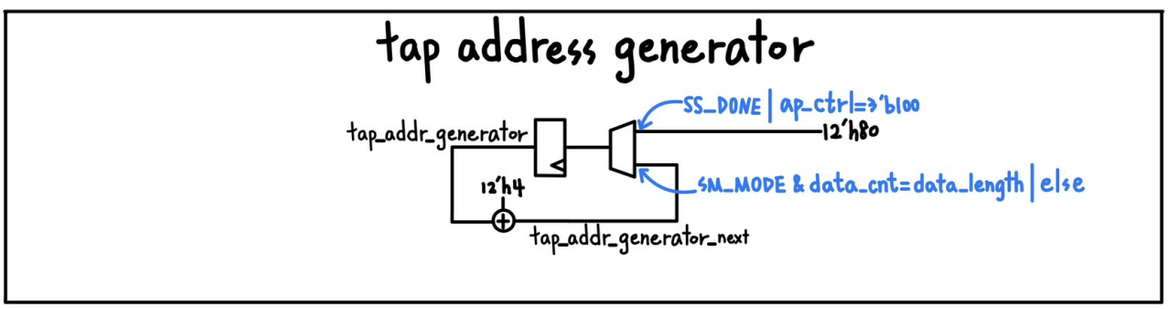
Datapath & Address Generator

As we show in the figure of datapath, the core engine gets the data from the tap_RAM and data_RAM and does the addition and multiplication to implement convolution.



The main thing we need to design is the address generator, it will generate the correct address to access data_RAM and tap_RAM.





The design of the tap_RAM address generator is straightforward. At the beginning of each processing cycle (SS_DONE), it resets to the start address (0x80) of tap_RAM. In each clock cycle, it generates the address for the next word in tap_RAM.

The design of data_RAM address generator includes three parts, write-address, read-address, and-fresh address. When using the AXI Stream to input data, the write address is used to access data_RAM. During convolution computation, the read address is used to access data_RAM. After processing a dataset, the data_RAM must be refreshed before executing the next dataset. The refresh address traverses the entire data_RAM to clear its stored data.

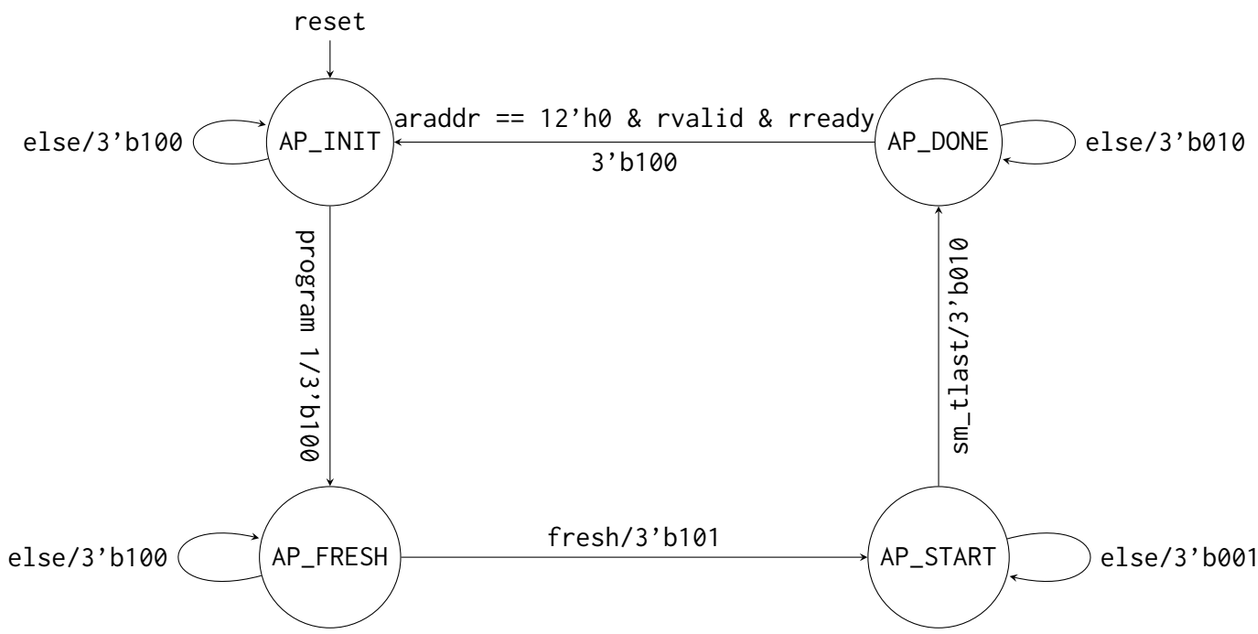
The write-address generator will reset to the start address ox80 of data_RAM when testbench program start(1) to ap_ctrl[0]. When each data comes SS_WRITE, it generates the address of data_RAM to write with FIFO.

The read-address generator will inverse traverse the entire data_RAM, the start address is the same as the write address for each data.

The fresh-address generator will traverse the entire data_RAM start from (0x80) when AP_FRESH.

FSM of Accelerator Protocol

The following FSM is a Mealy machine (`input: condition / output: ap_ctrl_fir`) used to control the state of `fir`, and is stored in the configuration register `ap_ctrl`. When we `program 1 to ap_ctrl` from the `testbench`, `fir` transitions to the `AP_FRESH` state, which activates the fresh-address generator to refresh the `data_RAM`. Once the refresh process is complete (`fresh == 1`), the system begins computing the convolution until the last data point is processed. Then, the state transitions to `AP_DONE`. After `AP_DONE` has been read, it automatically returns to the `AP_INIT` state.



Resource Usage

Resource Usage

```
****************************************
Report : area
Design : fir
Version: R-2020.09-SP5
Date   : Fri Mar 21 15:50:03 2025
****************************************

Library(s) Used:

    slow (File: /usr/cadtool/ee5216/CBDK_TSMC90GUTM_Arm_f1.0/CIC/SynopsysDC/db/slow.db)

Number of ports:                          330
Number of nets:                          3451
Number of cells:                         3113
Number of combinational cells:           2808
Number of sequential cells:               292
Number of macros/black boxes:               0
Number of buf/inv:                        577
Number of references:                      97

Combinational area:              13810.003429
Buf/Inv area:                     2434.320040
Noncombinational area:            4706.352012
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:                 18516.355441
Total area:                 undefined
```

Simulation Waveform

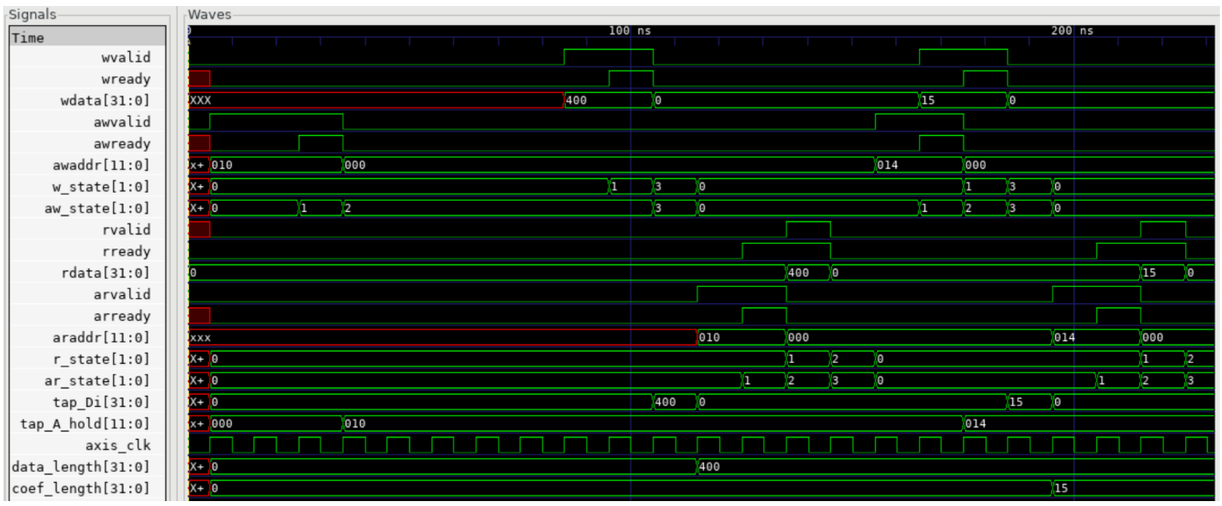


Figure 1: **AXI Lite waveform**

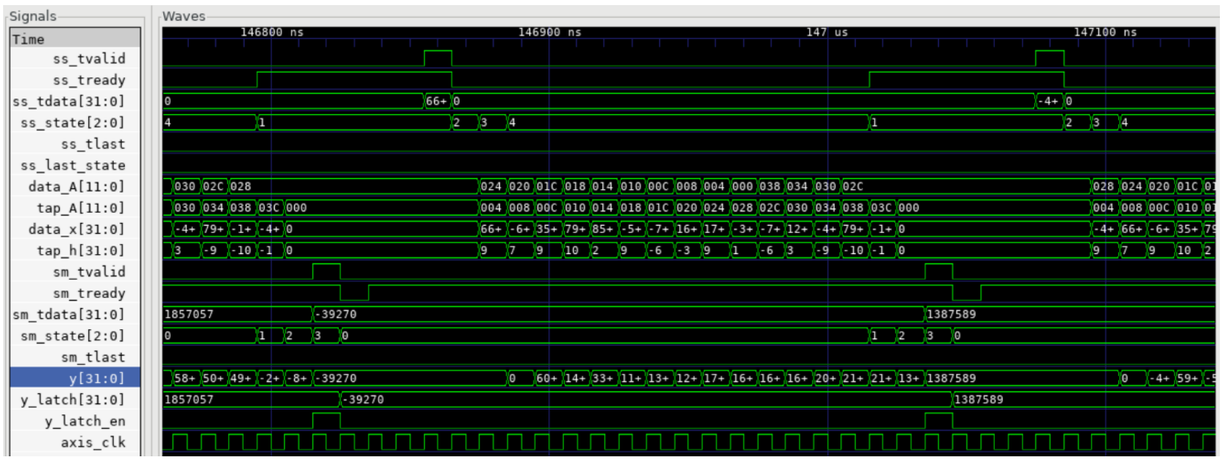


Figure 2: **AXI Stream waveform**

Timing Report

The longest path, is shown in the following report.

Timing Report

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 10
Design : fir
Version: R-2020.09-SP5
Date   : Fri Mar 21 15:50:03 2025
****************************************

Operating Conditions: slow   Library: slow
Wire Load Model Mode: top

  Startpoint: data_Do[17]
              (input port clocked by axis_clk)
  Endpoint: x_mul_h_reg[31]
            (rising edge-triggered flip-flop clocked by axis_clk)
  Path Group: axis_clk
  Path Type: max

  Point                                     Incr       Path
  -----------------------------------------------------------
  clock axis_clk (rise edge)                0.00       0.00
  clock network delay (ideal)               0.50       0.50
  input external delay                      1.00       1.50 r
  data_Do[17] (in)                          0.00       1.50 r
  U2412/Y (AND2X2)                          0.16       1.66 r
  U2438/Y (XOR2X1)                          0.14       1.80 f
  U2440/Y (NOR2X2)                          0.23       2.03 r
  U1142/Y (NAND2X1)                         0.09       2.12 f
  U1405/Y (OAI21XL)                         0.06       2.18 r
  U1404/Y (XOR2XL)                          0.10       2.28 f
  U1403/CO (ADDHXL)                         0.12       2.40 f
  U1279/CO (ADDHXL)                         0.10       2.50 f
  U1153/CO (ADDHXL)                         0.11       2.61 f
  U2715/CO (ADDFXL)                         0.17       2.78 f
  U2716/CO (ADDFXL)                         0.21       2.99 f
  mult_x_32/U423/CO (CMPR42X1)              0.43       3.42 f
  mult_x_32/U418/S (CMPR42X1)               0.42       3.84 f
  U2974/Y (OR2X2)                           0.10       3.94 f
  U1519/Y (AOI21XL)                         0.08       4.02 r
  U1517/Y (OAI21XL)                         0.10       4.12 f
  U2975/Y (AOI21X1)                         0.09       4.21 r
  U1513/Y (OAI21XL)                         0.09       4.30 f
  U1509/Y (AOI21XL)                         0.13       4.43 r
  U2976/Y (OAI21X1)                         0.09       4.52 f
  U2977/Y (AOI21X1)                         0.09       4.61 r
  U1507/Y (OAI21XL)                         0.09       4.70 f
  U1503/Y (AOI21XL)                         0.13       4.82 r
  U2978/Y (OAI21X1)                         0.08       4.90 f
  U3101/CO (ADDFXL)                         0.17       5.08 f
  U3160/Y (XOR2X1)                          0.09       5.17 r
  U3375/Y (AND2X2)                          0.09       5.26 r
  x_mul_h_reg[31]/D (DFFSRXL)               0.00       5.26 r
  data arrival time                                    5.26

  clock axis_clk (rise edge)                5.00       5.00
  clock network delay (ideal)               0.50       5.50
  clock uncertainty                        -0.10       5.40
  x_mul_h_reg[31]/CK (DFFSRXL)              0.00       5.40 r
  library setup time                       -0.14       5.26
  data required time                                   5.26
  -----------------------------------------------------------
  data required time                                   5.26
  data arrival time                                   -5.26
  -----------------------------------------------------------
  slack (MET)                                          0.00
```

Performance Report

From `testbench`, the third dataset has 300 data with no delay on ss-signal and sm-signal, the maximum latency of my design can be calculated by:

$$\text{latency} = \frac{9651(\text{clock cycle})}{300(\text{data})} = 32.17(\frac{\text{clock cycle}}{\text{data}}) \tag{2}$$

```
-------------------------------------------
----------Congratulations! Pass------------
------------------third dataset complete cycle count       9651(to know best case throuput)
```

Figure 3: **Latency**