Lab 3 Report

Topic: FIR design

112061524 Yeh, You Sung

Description:

In this Lab3, I learned how to design a Finite impulse response (FIR) from scratch with two interface AXI Lite and AXI stream. I design the FIR engine with only 1 multiplier and 1 adder. And the design a testbench based on TA's guidance. Testbench can take the role as the host and communicate with the FIR engine. And I also have to implement the communication protocol between BRAM and FIR engine, which is ASIC flow. The following section will show the implement detail and results and conclusion.

Implementation:

Block Diagram Framework:

In the Lab3 FIR design, I control the FIR engine and AXI-lite protocol with 2 FSM, respectively. The first FSM is called state, which controls the AXI-lite protocol to write ap_ctrl, data_length register and coefficient BRAM (tap BRAM), has 5 states such as IDLE, WADDR, WDATA, RADDR, RDATA. The second FSM is called FIR_state, which controls the FIR engine calculation, has 4 state such as IDLE, PROG, COMP, DONE. The function of 2 FSM will describe in following section.

This FIR design can handle AXI-Lite to write coefficients in tap BRAM, AXI-Stream to write inputs x[i] in BRAM and access memory units with ASIC flow. And finally output y[i] is the FIR engine calculation results. The framework of my design shows in below figure.

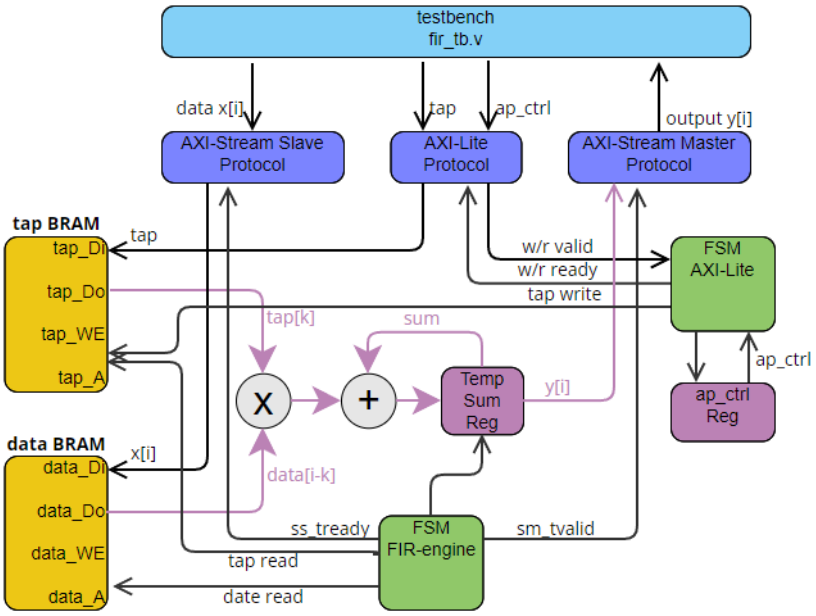


Fig. FIR block diagram.

Control signals:

The control signal for AXI-Lite, AXI-Stream and ASIC protocol is designed with respect to the Lab3 spec, and I only have 4 customized control signals, which are cnt, write_ptr, read_ptr and last. These 4 control signals will be described in the following section.

Implement detail:

FSM (for AXI-Lite):

Since AXI-Lite has 4 channels (ignore B channel), which are AW (address write), W (write), AR (address read), R (read). Thus, I design FSM to control 4 channels and WADDR, WDATA, RADDR, RDATA states is for AW, W, AR, R channels, respectively. That is, when AXI-Lite protocol is writing address, it can't write data or even other operations. This sounds very inefficient but it's easy to control the error cases and won't fall into wrong situations for the HDL design beginner. The figure shows FSM:

`WADDR: write address.

`WDATA: write data in write address.

`RADDR: read address.

`RDATA: read data in read address.

`IDLE: default state that AXI-Lite protocol is idle.

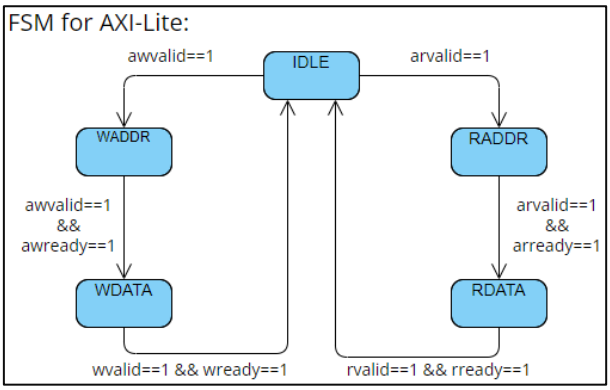


Fig. FSM for AXI-Lite.

FSM (for FIR engine):

Since FIR has to calculate the output with only 1 multiplier and 1 adder and also need to access the data and tap in BRAM. Thus, I design FSM to control the calculation and it can do once 1 multiplication and 1 addition per clock, which meets the design spec from HW. The FIR will wait for ap_start to start the engine. I set a counter for FIR engine to count the number of multiplication & addition per input x[i], since the number of multiplication & addition is 11 times per input x[i], depends on the number of tap is 11. After count to 11, it outputs the result y[i] with AXI-Stream protocol. And I add a flag last to indicate the last input x[i] coming. The figure shows FSM:

`IDLE: idle state that FIR engine is not utilized.

`PROG: program state that starts the computation and resets the counter.

`COMP: computation state to do the computation.

`DONE: done state that needs to output the results y[i].

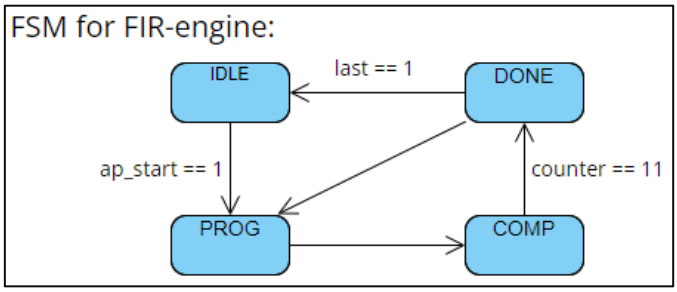


Fig. FSM for FIR engine.

Receive data-in x[i] and place in BRAM:

Since the testbench transfer the input data x[i] through AXI-Stream Slave protocol, the FIR engine will store the ss_tdata value into data BRAM in PROG state. And the write data address is controlled by the value write_ptr. The design for write_ptr will describe in shift-RAM section.

```
/*************************************************/
/*           data BRAM (Xn)                      */
/*************************************************/

assign data_EN = 1;
assign data_WE = (fir_state == fir_IDLE || fir_state == fir_PROG)? 4'b1111 : 0;
assign data_Di = (fir_state == fir_IDLE)? 0 : ss_tdata;
assign data_A  = (fir_state == fir_COMP)? (read_ptr << Addr_offset) : (write_ptr << Addr_offset);

/*************************************************/
/*           data BRAM (Xn) END                  */
/*************************************************/
```

Fig. code for data-in BRAM.

Receive tap parameters and place in BRAM:

Since the testbench transfer the coefficients through AXI-Lite protocol, the FSM for AXI-Lite will receive the write address in RADDR state and then store the tap into tap BRAM in RDATA state. And the control signals awready and wready is for this writing coefficient operation.

```
/*************************************************/
/*           tap BRAM (coefficient)              */
/*************************************************/

assign tap_EN = 1;
assign tap_WE = (state == WDATA && awaddr >= 32'h0020)? 4'b1111:0; // only AXI_lite write coefficient
assign tap_Di = wdata;
assign tap_A  = (state == WDATA && fir_state == fir_IDLE)? (awaddr-32'h0020) : // write coefficient
                (state == RADDR && fir_state == fir_IDLE)? (araddr-32'h0020) :  // read coefficient
                (cnt << Addr_offset);

/*************************************************/
/*           tap BRAM (coefficient) END          */
/*************************************************/
```

Fig. code for tap BRAM.

Access the shift-RAM for data RAM:

Since the shift operation for BRAM has to read out all values in BRAM then re-store them again, it is unrealistic to do shift in BRAM. Instead, I design a write pointer write_ptr to maintain the write address that I need to store the new input x[i], thus, I don't need to shift the data in BRAM and only have to know where the start pointer for each calculation is. The write_ptr is [0, 10] means the index in BRAM.

```
fir_DONE:begin
    write_ptr <= (write_ptr == 10)? 0 : write_ptr+1;
```

Fig. code for update write_ptr.

```
assign data_Di = (fir_state == fir_IDLE)? 0 : ss_tdata;
assign data_A  = (fir_state == fir_COMP)? (read_ptr << Addr_offset) : (write_ptr << Addr_offset);
```

Fig. code for write data BRAM in write pointer address.

Access the tap RAM:

Since the reading tap BRAM operation is directly read out the values in tap BRAM from index 0 to 10, thus, I only need to access the tap in BRAM depended on the counter I design for counting the number of multiplication & addition per input x[i]. The counter value increase during COMP state.

```
fir_COMP:begin
    cnt <= (cnt == Tape_Num)? 0 : cnt+1;
```

Fig. code for increase counter in COMP state.

```
assign tap_A  = (state == WDATA && fir_state == fir_IDLE)? (awaddr-32'h0020) : // write coefficient
                (state == RADDR && fir_state == fir_IDLE)? (araddr-32'h0020) :  // read coefficient
                (cnt << Addr_offset);
```

Fig. code for read out the tap depends on counter.

Do computation:

Since the reading data BRAM operation also need to shift for each calculation per clock, I design read_ptr read pointer to maintain the read address where the data address in the BRAM is. And do the computation per clock in COMP state. And the register sum is the storage for temporary results.

```
assign read_ptr = (write_ptr >= cnt)? (write_ptr-cnt) : Tape_Num-(cnt-write_ptr);
```

Fig. code for update read_ptr.

```
fir_COMP:begin
    cnt <= (cnt == Tape_Num)? 0 : cnt+1;
    sum <= sum + $signed(tap_Do) * $signed(data_Do);
```

Fig. code for 1 multiplication and 1 addition per clock.

Output results to testbench:

I have to use AXI-Stream Master protocol to transfer the output y[i] to testbench. When the counter is 11 means that I have done the computation for each input x[i], I transfer computation results in sum register with AXI-Stream Master.

```
assign sm_tvalid = (cnt == Tape_Num)? 1 : 0;
assign sm_tdata = sum;
assign sm_tlast  = (fir_state == fir_DONE)? ss_tlast : 0;
```

Fig. code for AXI-Stream Master protocol to transfer the output y[i] to testbench.

Storage and generate ap_control signal:

I store the ap_control signal in address 0x00 for the design spec. I design a register ap_ctrl_reg to maintain the ap_control signal since the FIR engine need to access the ap_control value during operation. When the design reset, the ap_idle is high and ap_start is low and ap_done is low. When the write address is 0x00, that is, testbench needs to write ap_control signal, I write wdata into register. When the read address is 0x00, that is, ap_done is read by testbench, ap_done will reset to 0.

```
// ap_ctrl & data_length register configuration
integer i;
always @(posedge axis_clk) begin
    if (~axis_rst_n) begin
        ap_ctrl_reg[0] <= 0; // ap_start
        ap_ctrl_reg[1] <= 0; // ap_done
        ap_ctrl_reg[2] <= 1; // ap_idle
        for (i = 3; i < pDATA_WIDTH; i=i+1) begin
            ap_ctrl_reg[i] <= 0;
        end
        data_length_reg <= 0;
    end
    else begin
        if (state == WDATA) begin
            if (awaddr == 32'h0000) begin // 0x00
                ap_ctrl_reg <= wdata;
            end
            else if (awaddr == 32'h0010) begin // 0x10
                data_length_reg <= wdata;
            end
        end
        else if (state == RDATA) begin // When ap_done is read, i.e. address 0x00 is read
            ap_ctrl_reg[1] <= (awaddr == 32'h0000)? 0 : ap_ctrl_reg[1];
        end
        else begin
            // ap_start
            ap_ctrl_reg[0] <= (ap_ctrl_reg[0] == 0)? 0 :
                              (fir_state == fir_IDLE)? 1 : 0;
            // ap_done is asserted when engine completes last data processing and data is transferred
            ap_ctrl_reg[1] <= (ap_ctrl_reg[1] == 1)? 1 :
                              (sm_tlast == 1 && fir_state == fir_DONE)? 1 : 0;
            // ap_idle
            ap_ctrl_reg[2] <= (ap_ctrl_reg[2] == 1)? ((ap_ctrl_reg[0] == 1)? 0 : 1) :
                              (fir_state == fir_IDLE)? 1 : 0;
        end
    end
end
```

Fig. code for ap_control signal.

Result & Discussion:

Resource usage screenshot:

There is no Latch in my design, thus, it won't have the unstable signal in circuit. And since the BRAM is external module is this Lab, there is no BRAM utilization.

1. Slice Logic

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	192	0	0	53200	0.36
LUT as Logic	192	0	0	53200	0.36
LUT as Memory	0	0	0	17400	0.00
Slice Registers	112	0	0	106400	0.11
Register as Flip Flop	112	0	0	106400	0.11
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

Fig. FF, LUT utilization screenshot in synthesis report file.

2. Memory

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

Fig. BRAM utilization screenshot in synthesis report file.

3. DSP

Site Type	Used	Fixed	Prohibited	Available	Util%
DSPs	3	0	0	220	1.36
DSP48E1 only	3				

Fig. DSP utilization screenshot in synthesis report file.

Timing Report screenshot:

I try to synthesize the design with maximum frequency. I choose the 4ns clock time period the timing report shows below. **The slack is positive value**. And the next figure shows the Max Delay Path in my design. The source and the destination point in my design is the sum_reg, which stores the temporary calculation results for FIR.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.111 ns	Worst Hold Slack (WHS):	0.146 ns	Worst Pulse Width Slack (WPWS):	1.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	152	Total Number of Endpoints:	152	Total Number of Endpoints:	113

All user specified timing constraints are met.

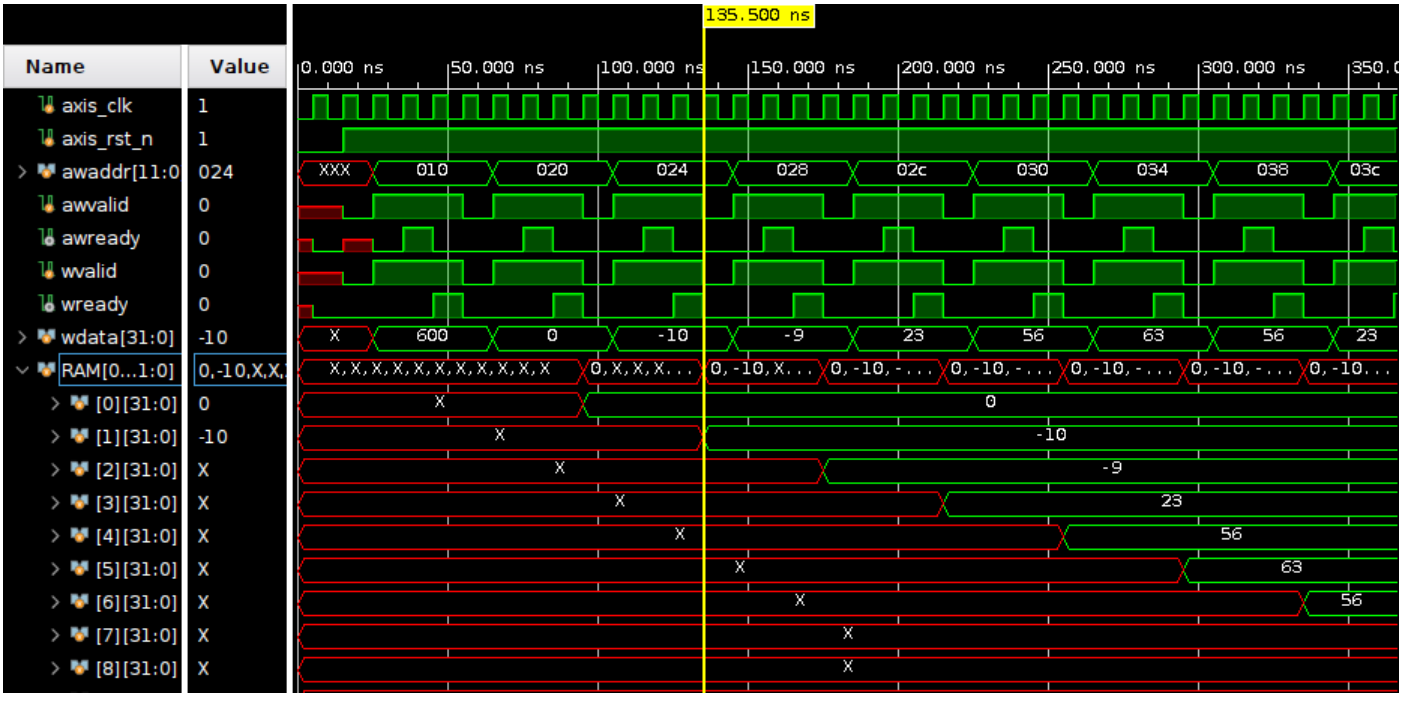
Fig. Timing report with clock cycle time 4 ns.

```
Max Delay Paths
--------------------------------------------------------------------------------------
Slack (MET) :             0.111ns  (required time - arrival time)
  Source:                 genblk1.sum_reg[1]/C
                            (rising edge-triggered cell FDRE clocked by axis_clk  {rise@0.000ns fall@2.000ns period=4.000ns})
  Destination:            genblk1.sum_reg[31]/D
                            (rising edge-triggered cell FDRE clocked by axis_clk  {rise@0.000ns fall@2.000ns period=4.000ns})
  Path Group:             axis_clk
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            4.000ns  (axis_clk rise@4.000ns - axis_clk rise@0.000ns)
  Data Path Delay:        3.753ns  (logic 2.638ns (70.290%)  route 1.115ns (29.710%))
  Logic Levels:           10  (CARRY4=8 LUT2=2)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    2.128ns = ( 6.128 - 4.000 )
    Source Clock Delay      (SCD):    2.456ns
    Clock Pessimism Removal (CPR):    0.184ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
```

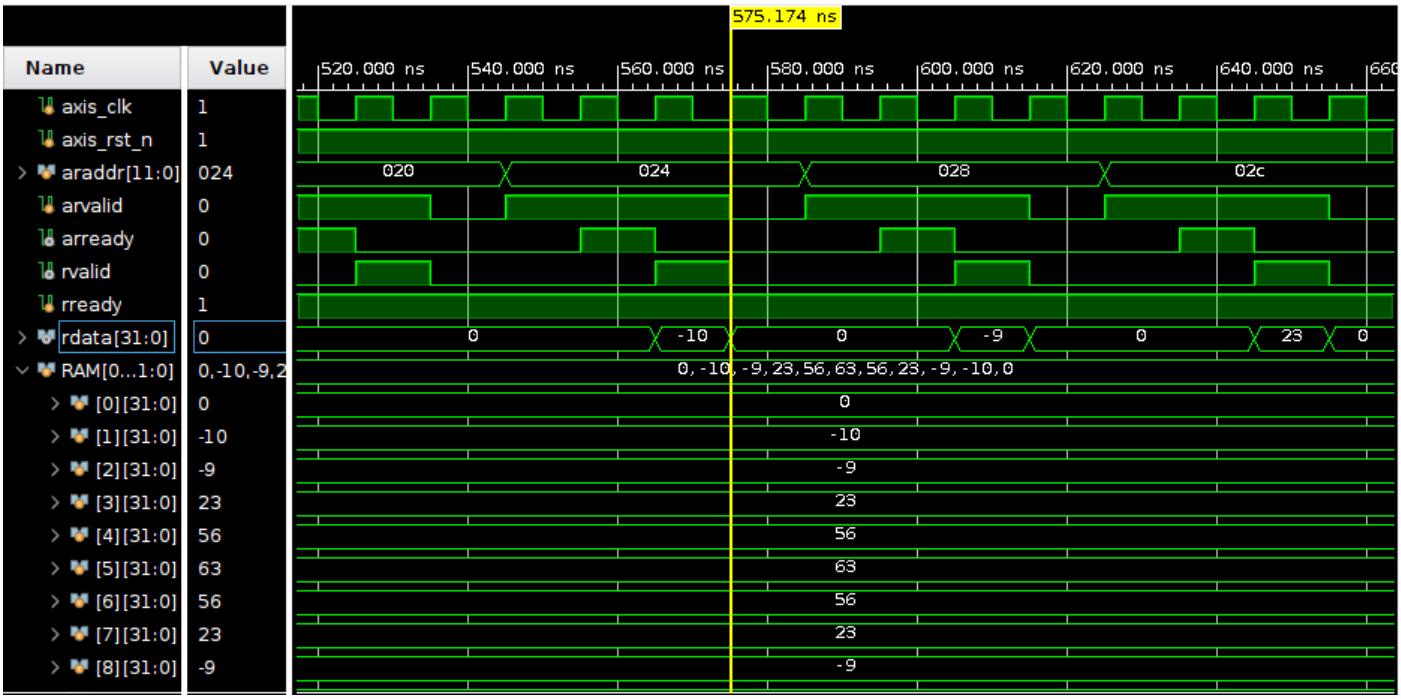
Fig. Max Delay Path in timing report file.

Simulation Waveform screenshot:

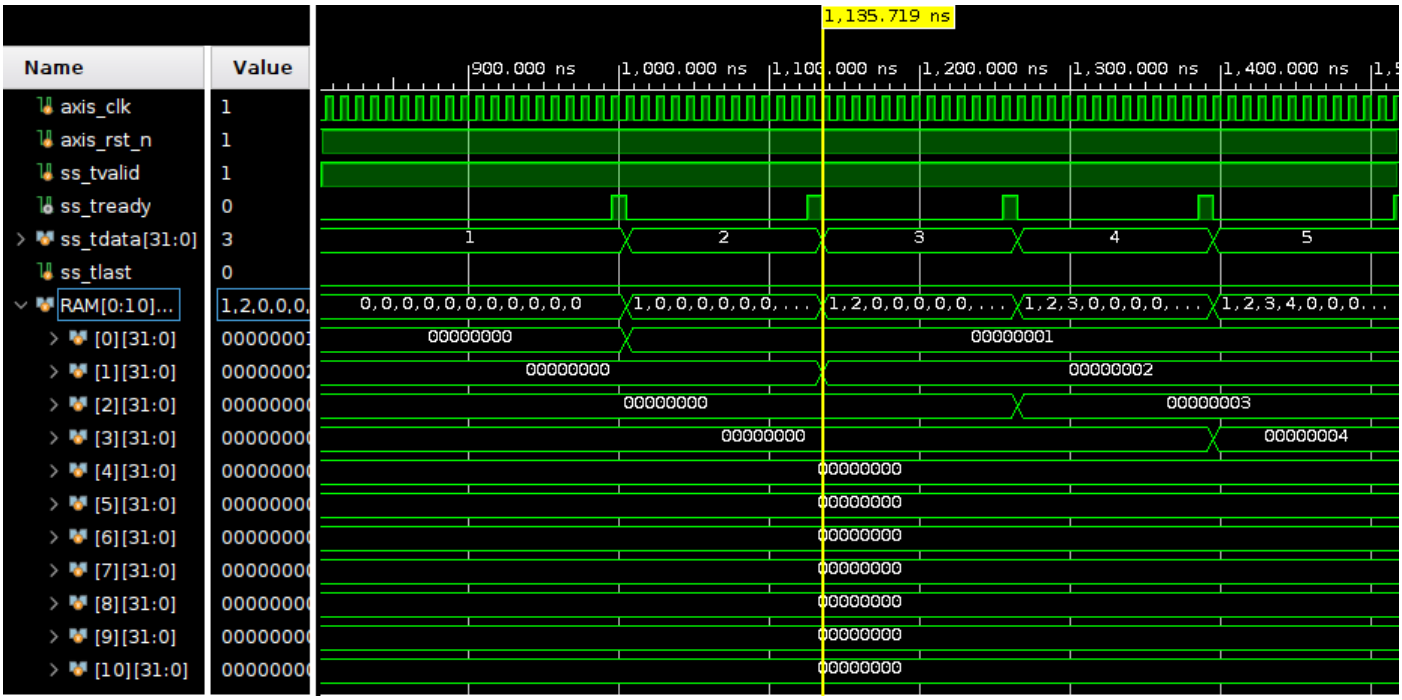
Coefficient program and read back:



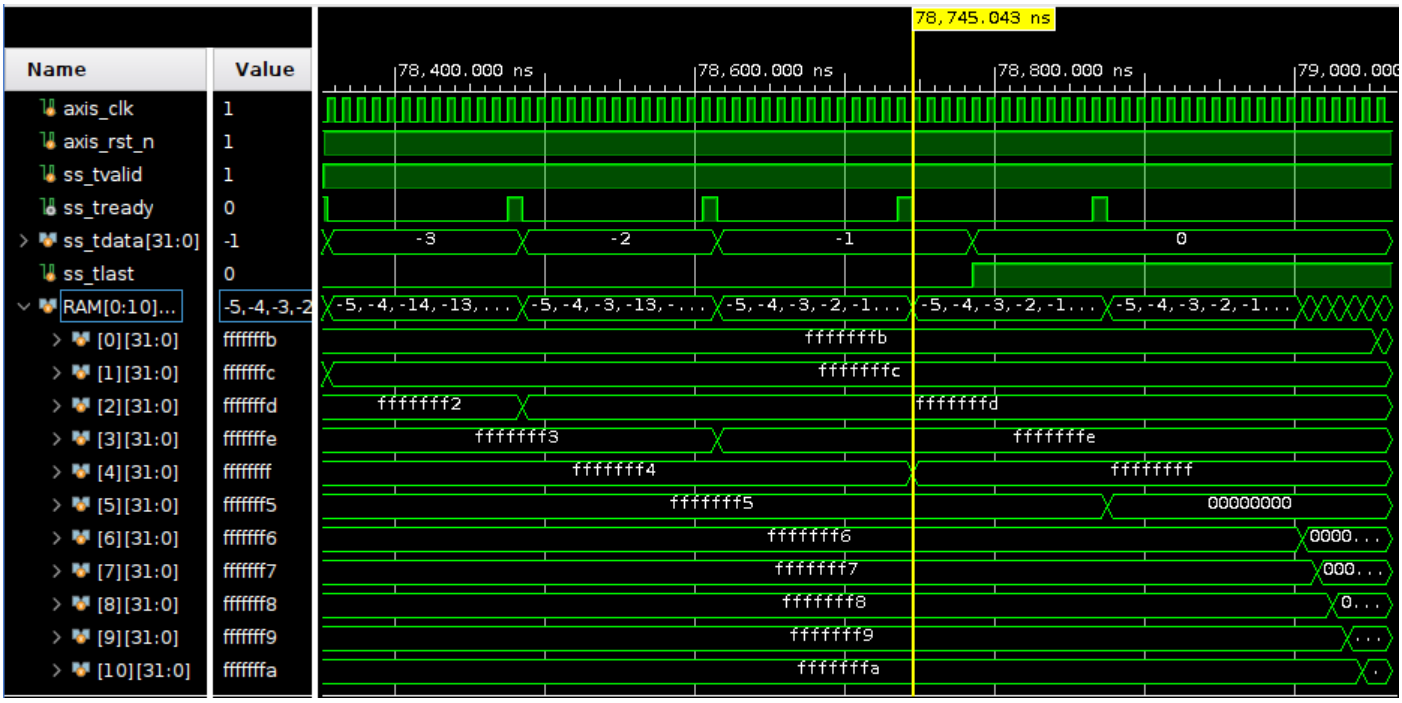
Coefficient read back:



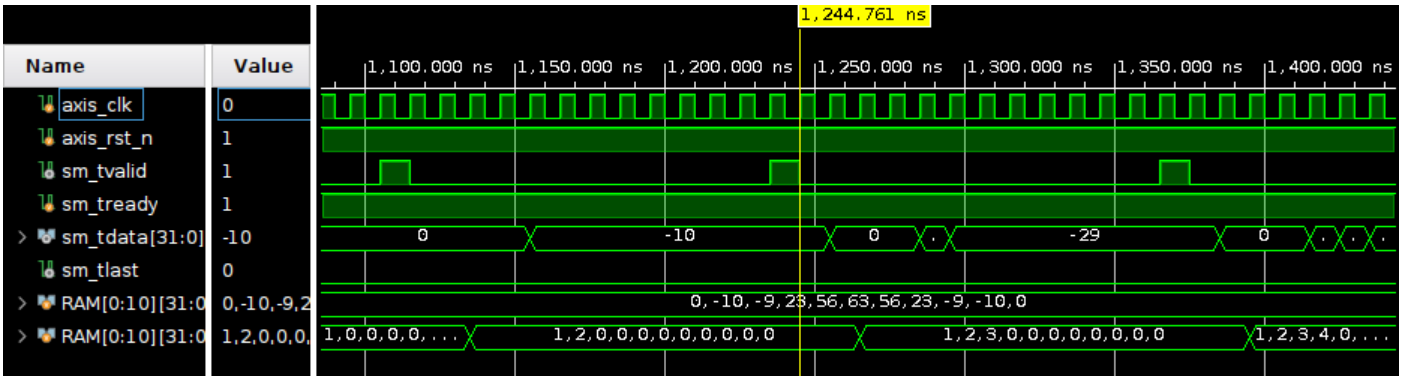
Data-in AXI-Stream-in:



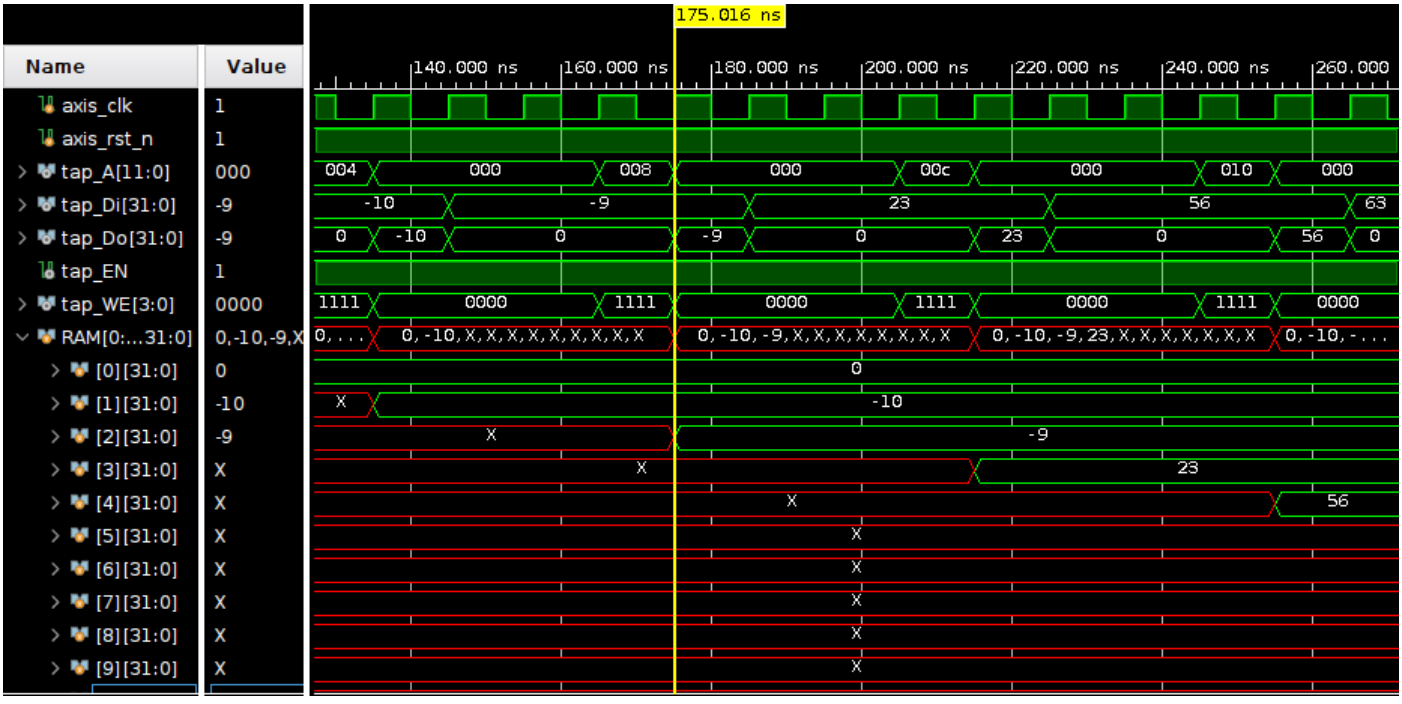
Data-in AXI-Stream-in (data tlast):



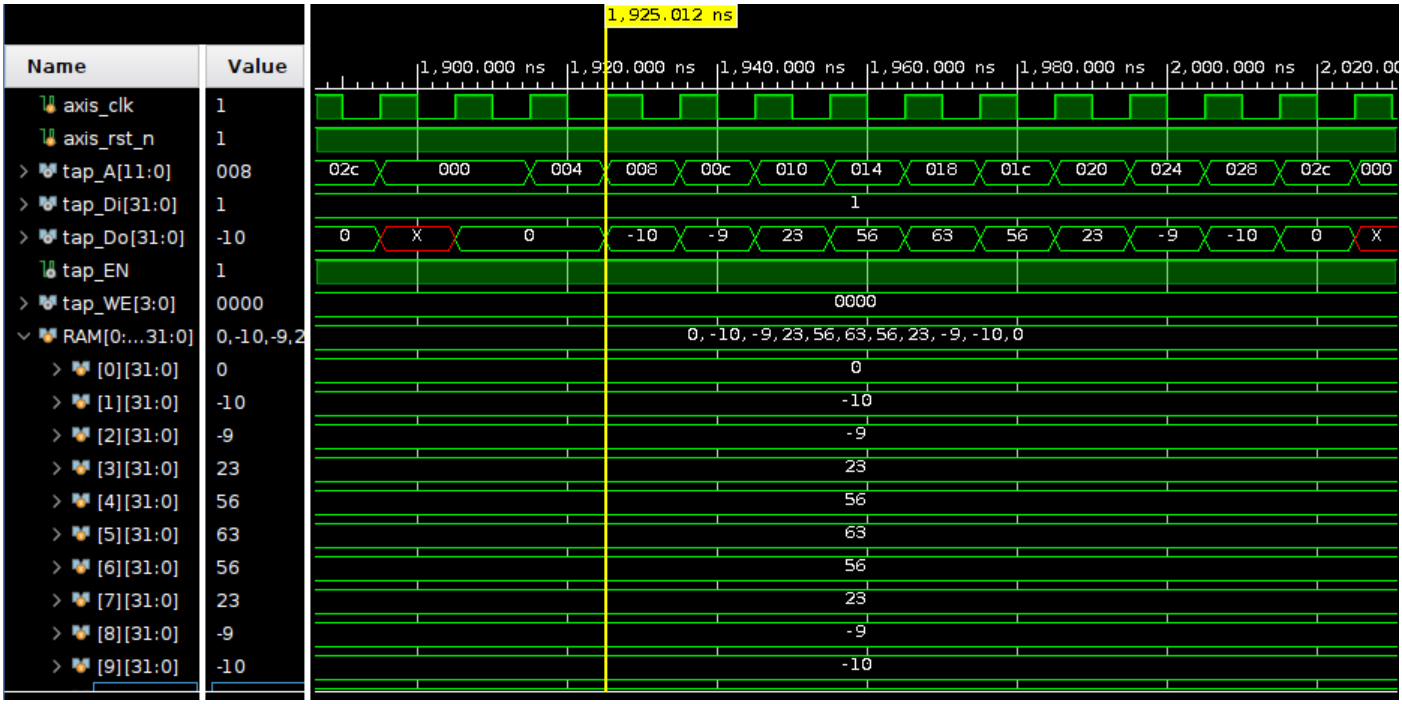
Data-out AXI-Stream-out:



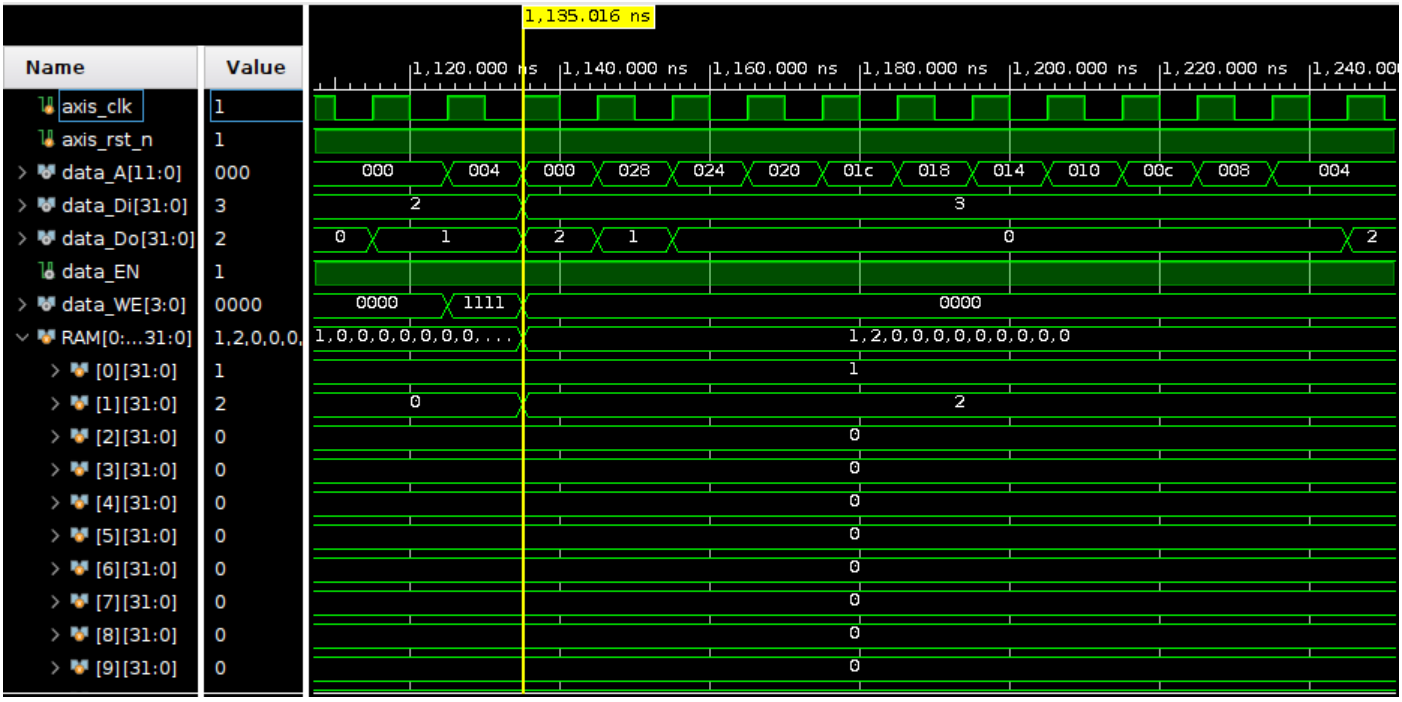
BRAM access control (write tap BRAM):



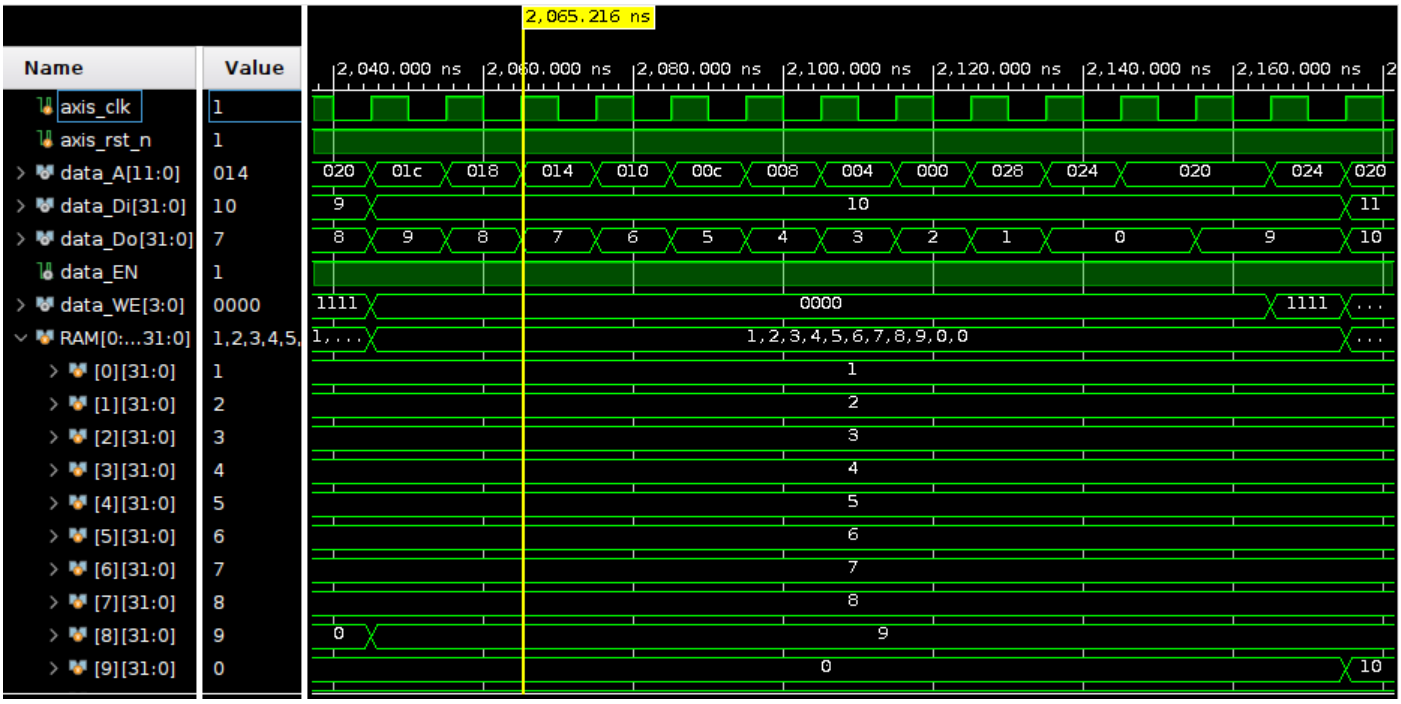
BRAM access control (read tap BRAM):



BRAM access control (write data BRAM):



BRAM access control (read data BRAM):



Conclusion:

I learned how to design a FIR design for scratch and also learn how to set up the protocol such as AXI-Lite, AXI-Stream and ASIC flow memory access protocol. The project and results can be accessed on my GitHub personal repository, which is called course_Lab3. Finally, I am familiar with the design flow about Verilog HDL.

GitHub Link:

Repository: course-lab_3
Link: https://github.com/yousungyeh/course-lab_3

Reference:

- TA lab tutorial, lab workbook and course lectures.