System of Chip (SoC) Lab3 FIR

111061545 陳揚哲

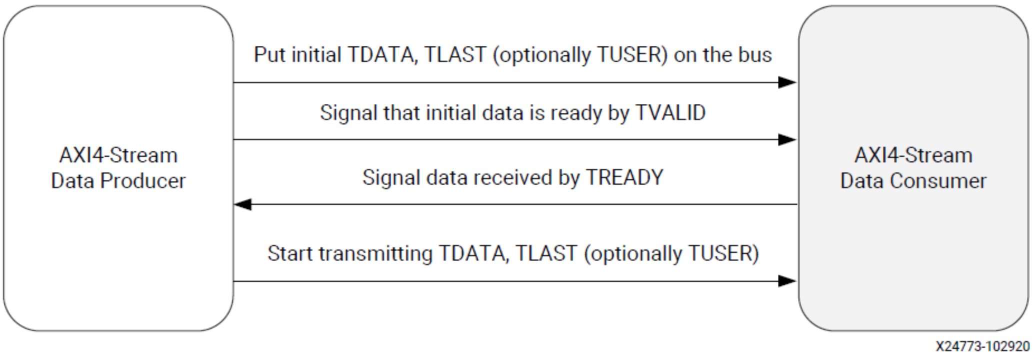
1. System specification

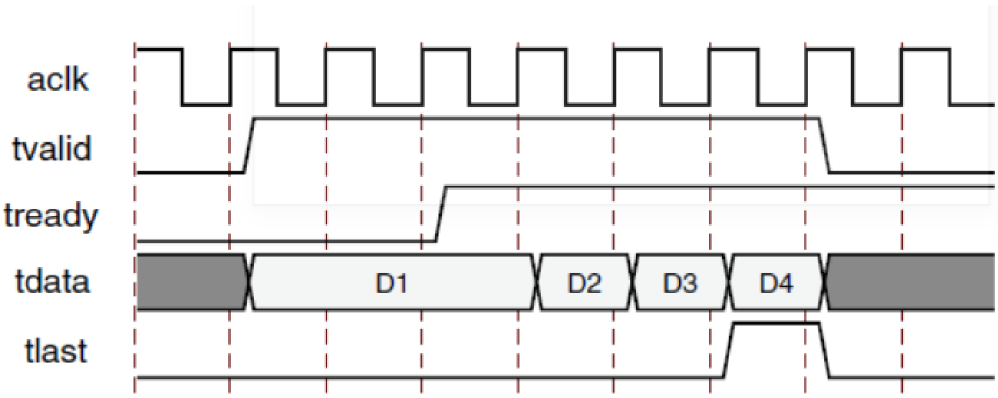
In this lab, we will need to design a FIR with tape number = 11, the system can be written as function below:

$$y[n] = \sum x[11-n] * h[n],\ \ \forall\, n < 0\,, x[n] = 0.$$

We will design a testbench as host side to program the input data (signal) and the coefficient to the FPGA side. The input data will program through the AXI-stream interface (1), and the coefficient will program through the AXI-lite interface (2). Then, we will perform the multiplication and addiction operation. The output data will be sent to host side (testbench) through AXI-stream interface. Besides, we need to apply BRAM (3) as our memory element rather than shift register. In this lab, we also limit the number of adders and multipliers in the FIR system to only one each.

(1) AXI-Stream:





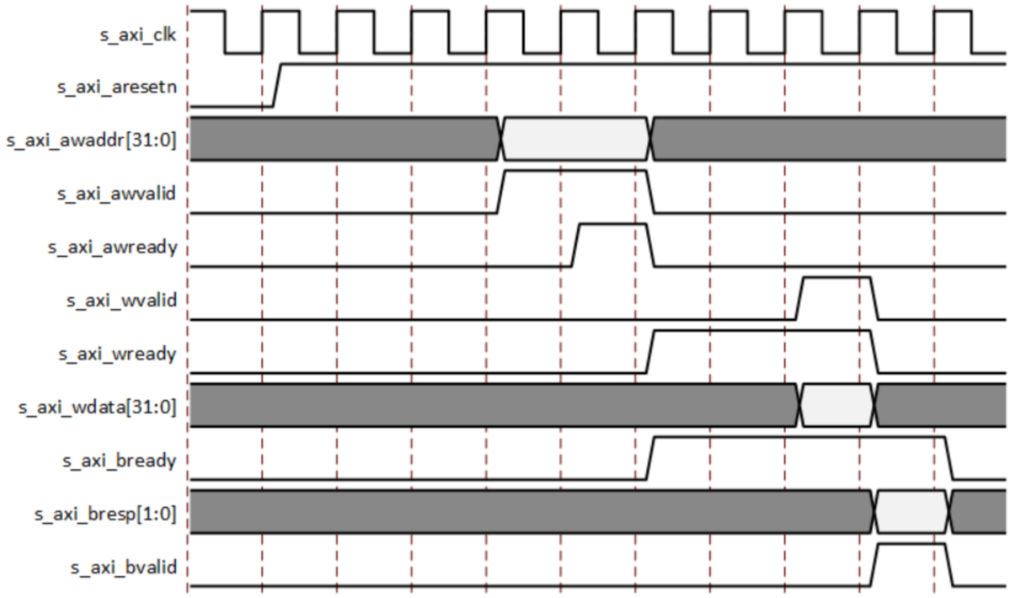
- **valid**: the master (data producer) is ready to output the data.
- **ready**: the slave (data consumer) is ready to receive the data.

- **data**: the data we want to transfer from master to slave.
- **last**: indicate the last data is the last data we want to transfer.

To operate the AXI-stream interface, we require 5 I/O signals: clock, valid, ready, data, and last. The graph above provides a visual representation of the process.

Initially, only the clock signal is active. In the second step, the valid signal is set to 1, accompanied by the data signal (D1). Subsequently, the ready signal is set to 1, indicating that the slave is ready to receive the data signal (D1). Data transfer occurs only when both valid and ready are set to 1, signifying that a handshake has been established between the master and the slave. It's important to note that data transfer occurs in a single cycle. The 'last' signal indicates that it is the final signal in the sequence.

(2.1) AXI-lite (Write):



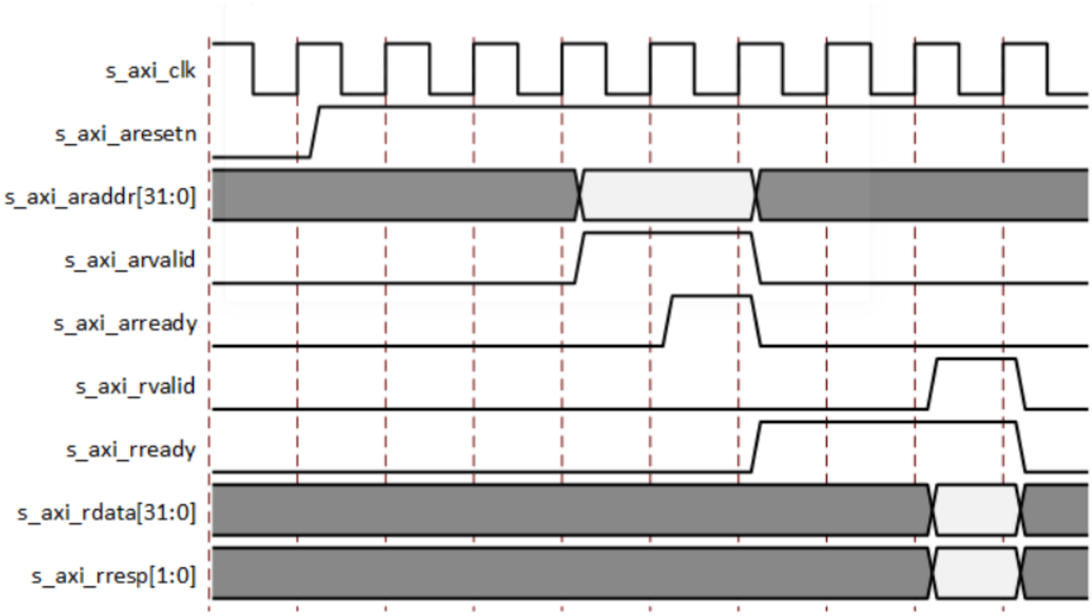
- **awvalid:** Write address valid. Master generates this signal when Write Address and control signals are valid to read.
- **awready:** Write address ready. Slave generates this signal when it can accept Write Address and control signals
- **awaddr**: Write address, usually 32-bits wide. In our case, it's 12 bits
- **wvalid**: the master (data producer) is ready to output the data.
- **wready**: the slave (data consumer) is ready to receive the data.
- **wdata**: the data we want to transfer from master to slave. (32-bit only).

In this lab, we use a simplified AXI-lite write interface, requiring 7 I/O signals: clock, awready, awvalid, awaddr, wready, wvalid, and wdata, respectively. The graph above provides an overview of the process.

Initially, only the clock signal is present. Subsequently, the awvalid signal is set to 1 along with the awaddr signal. At this point, the awready signal is also set to 1, indicating that the slave is ready to receive the awaddr signal. Once the handshake is formed, it signifies that the write address has been sent from the master to the slave. Address transfer only occurs when a handshake is established between the master and the slave, i.e., when awvalid and awready are both active.

Data transfer follows a similar procedure to address transfer. When wdata is ready to be output, wvalid is set to 1. Once wvalid and wready establish a handshake, data is transferred. It's important to note that data transfer doesn't have to wait for the address handshake to be established; it can occur at the same time or even earlier.

(2.2) AXI-lite (Read):



- **arvalid:** Read address valid. Master generates this signal when Read Address and the control signals are valid.
- **arready:** Read address ready. Slave generates this signal when it can accept the read address and control signals.
- **araddr**: Read address, usually 32-bit wide. In our case, it’s 12 bits.
- **rvalid**: Read valid. Slave generates this signal when Read Data is

valid.

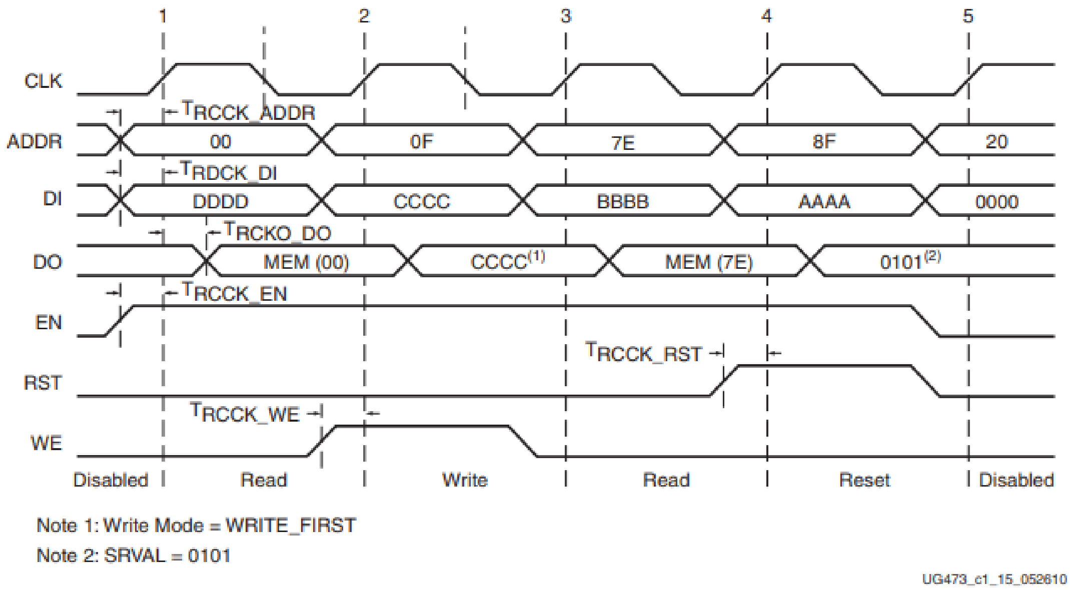
- **rready**: Read ready. Master generates this signal when it can accept the Read Data and response.
- **rdata**: Read Data (32-bit only).

In this lab, we utilize a simplified AXI-lite read interface, which involves 7 I/O signals: clock, arready, arvalid, araddr, rready, rvalid, and rdata. The graph above provides a visual representation of the process.

Initially, only the clock signal is active. In the second step, the arvalid signal is set to 1, along with the araddr signal. Subsequently, the arready signal is set to 1, indicating that the slave is ready to output the araddr signal. Once a handshake is formed, it signifies that the read address has been sent from the master to the slave. Address transfer only occurs when a handshake is established between the master and the slave, i.e., when arvalid and arready are both active.

Data transfer follows a procedure similar to the write protocol. When rdata is ready to be output, the slave sets rvalid to 1. Simultaneously, when the master is ready to receive the data, it sets rready to 1. Once rvalid and rready form a handshake, data is transferred.

(3) SRAM (BRAM):



- **ADDR:** The SRAM address wants to be read/ written.
- **Di:** The input(write) data.
- **Do:** The output(read) data.
- **EN:** The enable signal for SRAM to be written/read.

- **WE:** The write enable signal. Write behavior only take place when WE set as 1

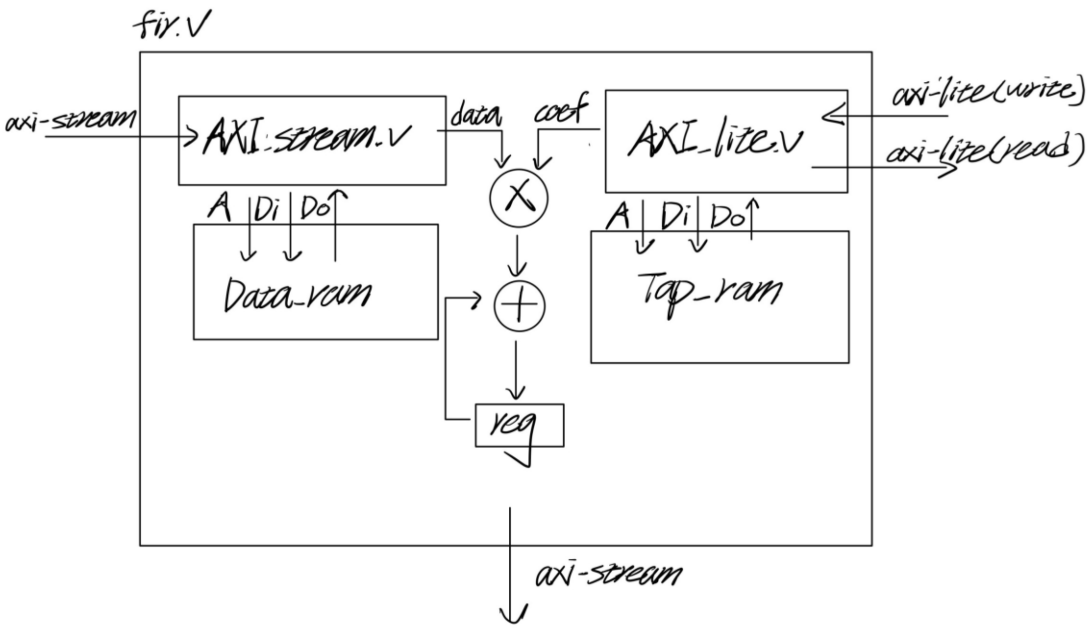
When we wish to write data into SRAM, we send the ADDR and Di (data) to the SRAM. If EN (Enable) and WE (Write Enable) are both set to 1, it indicates that the SRAM is enabled to write the data. As a result, the SRAM's address is updated with the data Di.

Conversely, when we want to read data, we send the ADDR to the SRAM. If EN is set to 1, the data will be output by the SRAM in the next cycle.

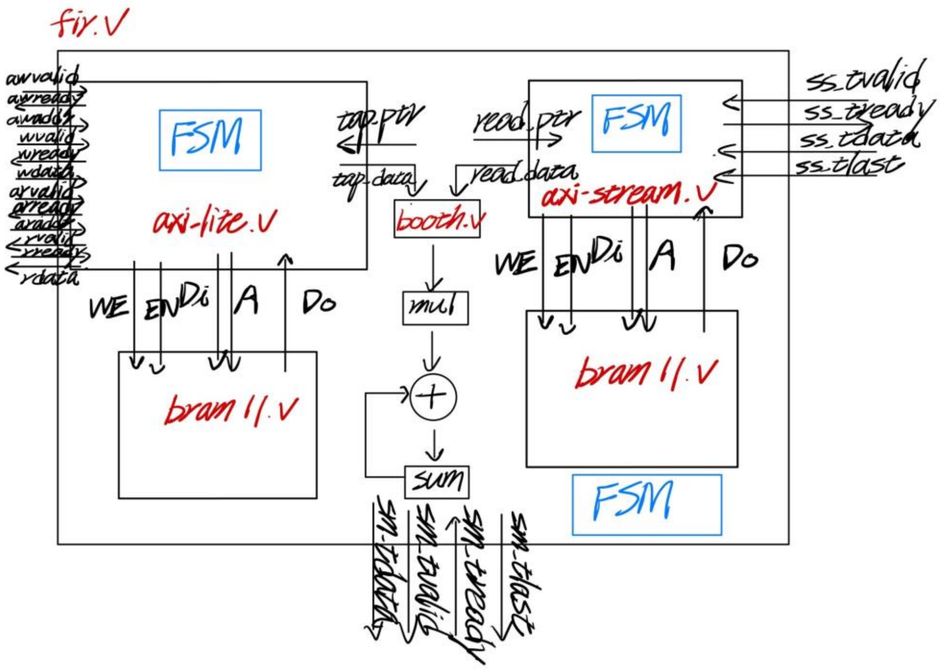
It's important to note that the EN signal controls whether the SRAM is in a read or write mode, while the WE signal specifically controls the SRAM's write capability. If only the EN signal is active, the data cannot be written into the SRAM; it will only generate data at the assigned address in the subsequent cycle.

2. Block diagram

The simplified design structure:



Block diagram:



3. Module design

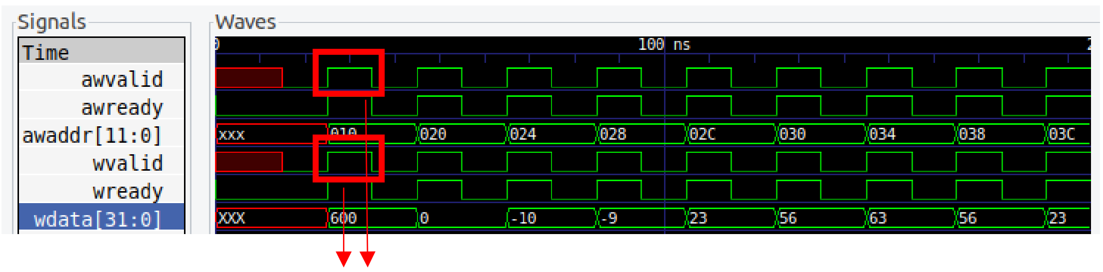
- ### axi-lite.v:

This module is designed to communicate with the host (testbench) via an AXI-lite interface. I have designed a three-state finite state machine (FSM), as outlined in the table below.

State	Action	Next state
IDLE	When the system has not been reset, it will remain in the IDLE state, awaiting the **axis_rst_n** signal.	When **axis_rst_n** ==1, go to **WAIT**
WAIT	In the AXI-lite write case, when the host sends the **awvalid** signal, we do not respond to the signal immediately. In our design, we wait until both **awvalid** and **wvalid** are valid, and only then do we send the **awready** and **wready** signals to the host. It's important to note that, according to the address map, addresses in the range of 0x20 to 0xFF are designated for coefficient storage, and only addresses within this range will be written into the **Tap_RAM** (BRAM).	When **ap_start** == 1, go to **CAL**

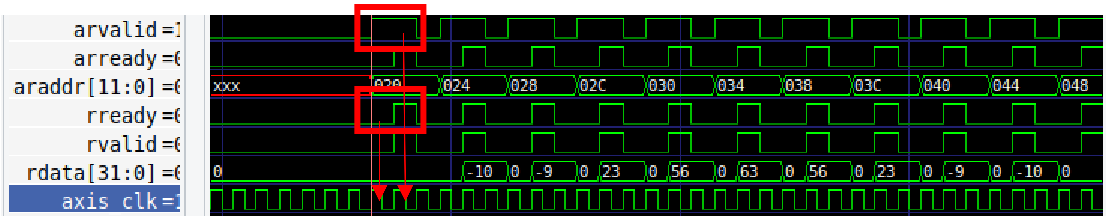
	For the AXI-lite read case, when the host sends the **arvalid** signal, the following actions are taken based on the address: • If the address is 0x00, we prepare the **ap_control** signal for AXI-lite output. • If the address is 0x10, we prepare the data length. • If the address is in the range of 0x20 to 0xFF, these addresses are sent to **Tap_RAM** (BRAM) to read the data at the specified addresses. We wait for the **rready** signal to arrive, and then we send the **arready** and **rvalid** signals for the AXI-lite read case. Once **ap_start** is assigned to the **ap_control** register, which signifies the start of the FIR calculation, the system transitions to the CAL state in the next cycle.	
CAL	During the calculation state, as FIR calculations require the use of **Tap_RAM** (BRAM), the data read from **Tap_RAM** is not accessible for the host side. The priority is given to the **tap_ptr** signal for the FIR calculation process. However, the **ap_control** and **data length** are still available for the host to read.	When **ap_done** == 1, go to **WAIT**

AXI-lite write:



When both **awvalid** and **wvalid** == 1, we send the **awready** and **wready** to accomplish AXI-lite write case.

AXI-lite read:



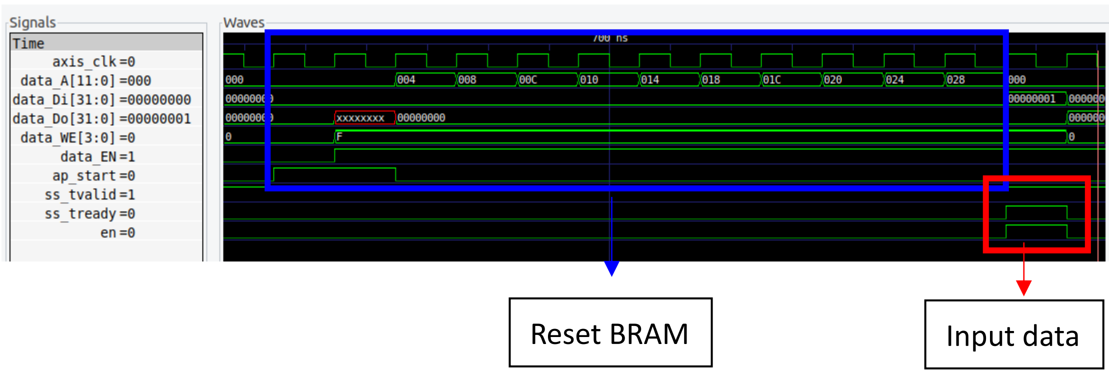
When **arvalid** == 1, prepare the correspond data for the address. Once the **rready** == 1, we send **arready** and **rvalid** to accomplish AXI-lite read case.

- **axi-stream.v:**

This module is designed to read input data from the host (testbench) via an AXI-stream interface. I have designed a three-state finite state machine (FSM), as described in the table below.

State	Action	Next state
IDLE	When the system has not been activated, it will remain in the **IDLE** state, awaiting the **ap_start** signal.	When **ap_start** ==1, go to **INIT**
INIT	In this state, we will address two issues. The first issue is resetting the **Data_RAM** (BRAM), and the second issue is related to addressing caused by the reset. **1. Data_RAM Reset:** In this state, we reset the data in the **Data_RAM**. Since the BRAM does not have a built-in reset port, we must manually reset the data in the **Data_RAM**. The Data input during this state is always set to 0. **2. Addressing with write_ptr:** Due to the need for manual reset of the BRAM, the address for the BRAM should be managed by us. In this case, I have designed the **write_ptr** signal to handle the address input for the BRAM.	When **init_done** == 1, go to **WAIT**
WAIT	There are two issues that need to be addressed: the **write_ptr** and **ss_tready** signal. **1. Handling Write_ptr:** Given that the AXI-stream interface doesn't inherently include an address signal, we need to	When **ap_done** == 1, go to **IDLE**

	manage the **write_ptr** for input data. The **write_ptr** will increase by 1 each time data transfer occurs, and when it reaches the maximum value (in this case, 10), it will wrap around to 0, effectively cycling through the available BRAM addresses. **2. ss_tready Signal:** The operation of the FIR has resource limitations, meaning we can only accept the next input data when the output data has been processed by our system. To control this, we have designed the EN signal to regulate **ss_tready**. When both EN and **ss_tvalid** are set to 1, the **ss_tready** signal will be activated. This acts as a flow control mechanism for the input data.	

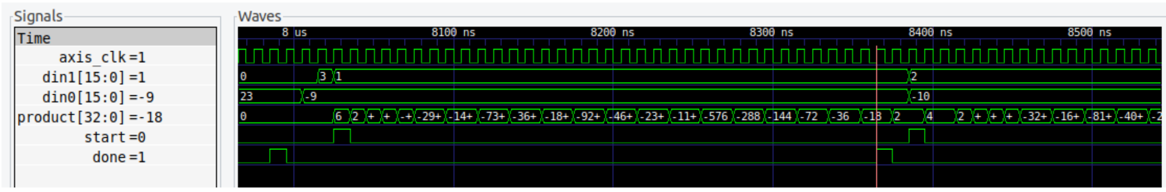


- ## booth.v

We design the multiplier with booth algorithm. Using booth multiplier, we can reduce the design area and resource use. Without using DSP, we can use adder to accomplish multiplication. Here we use the 16 bit booth multiplier.

State	Action	Next state
IDLE	Wait the **mul_start** signal to activate the multiplication operation. In this step, we use a 33-bit register **product** to temporary store the multiplication data, which is {16’b0, Din1, 1’b0}. The count value is for recording the operation times. In this state, count is initialized to 0 (count <= 0).	When **mul_start** ==1, go to **CAL**

CAL	***count <= count + 1*** ***If last 2 bits == {01}*** ***product <= {(product[32:17] + Din0_reg), product[16:0]};*** ***else if last 2 bits == {10}*** ***product <= {(product[32:17] + sDin0_reg),product[16:0]};*** ***(sDin0_reg is Din0's complement)*** ***else*** ***product <= product;***	Next state == **SHIFT**
SHIFT	***product <= {product[32], product[32:1]};***	When **count** == 16, go to **IDLE**
DONE	Generate the multiply output and the mul_done signal	Next state == **IDLE**



Take 34 cycle

- **fir.v**

This Module control the whole system control signal and fir operation. Once we receive the ap_start signal from axi-lite.v, the fir.v will be activated.

State	Action	Next state
IDLE	Wait the **ap_start** signal.	If **ap_start** == 1, go to **INIT**
INIT	Wait the data_RAM to be initialized.	If **wait_ram** == 1, go to **RUN_INIT**
RUN_INIT	Wait the input data is write into the Data_RAM.	If **shift** == 1, go to **RUN_TAP**
RUN_TAP	Generate the **mul_start** signal	Next state == **WAIT_MUL**
WAIT_MUL	Wait until the **mul_done** signal rise. If **mul_done** update the **read_ptr** and **tap_ptr.**	If **mul_done** == 1, go to SUM
SUM	The **read_ptr** and **tap_ptr** have been updated. Sum up the **mul** (multiplier output) of all 11 elements. ***sum <= sum + mul;***	If **times** == 11 go to OUTPUT else go to RUN_TAP
OUTPUT	Wait the data be read by the host.	If **sm_tready** == 1, go to RUN_INIT.

DONE	If the operation count (**count**) is equal to the **data length**, the operation should be halted, and the **ap_done** signal will be set to 1. If a second operation command is assigned, the system will transition back to the **INIT** state.	If **ap_start** == 1, go to **INIT**

4. Resource usage

1. LUT and FF

```
1. Slice Logic
--------------

+-------------------------+------+-------+------------+-----------+-------+
|        Site Type        | Used | Fixed | Prohibited | Available | Util% |
+-------------------------+------+-------+------------+-----------+-------+
| Slice LUTs*             |  314 |     0 |          0 |     53200 |  0.59 |
|   LUT as Logic          |  314 |     0 |          0 |     53200 |  0.59 |
|   LUT as Memory         |    0 |     0 |          0 |     17400 |  0.00 |
| Slice Registers         |  210 |     0 |          0 |    106400 |  0.20 |
|   Register as Flip Flop |  210 |     0 |          0 |    106400 |  0.20 |
|   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 |
+-------------------------+------+-------+------------+-----------+-------+
```

2. BRAM

```
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 |
+----------------+------+-------+------------+-----------+-------+
```

3. DSP

```
3. DSP
------

+-----------+------+-------+------------+-----------+-------+
| Site Type | Used | Fixed | Prohibited | Available | Util% |
+-----------+------+-------+------------+-----------+-------+
| DSPs      |    0 |     0 |          0 |       220 |  0.00 |
+-----------+------+-------+------------+-----------+-------+
```

5. Timing report

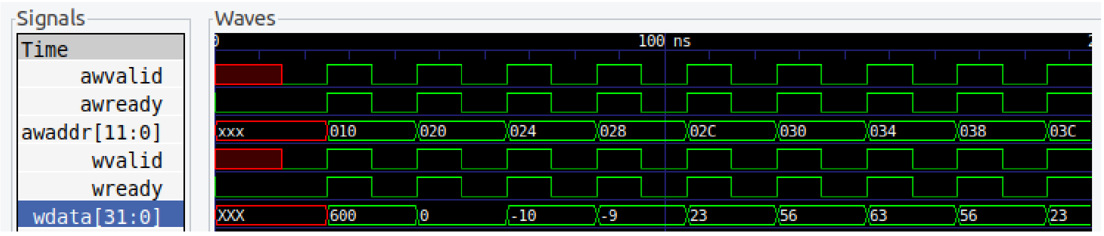
Clock is set as 5.860 ns, the critical path is shown below:

```
Max Delay Paths
--------------------------------------------------------------------------------------
Slack (MET) :              0.002ns  (required time - arrival time)
  Source:                  axilite_U/data_length_reg_reg[5]/C
                             (rising edge-triggered cell FDCE clocked by axis_clk  {rise@0.000ns fall@2.930ns period=5.860ns})
  Destination:             axi_stream_U/write_ptr_reg[0]/CE
                             (rising edge-triggered cell FDCE clocked by axis_clk  {rise@0.000ns fall@2.930ns period=5.860ns})
  Path Group:              axis_clk
  Path Type:               Setup (Max at Slow Process Corner)
  Requirement:             5.860ns  (axis_clk rise@5.860ns - axis_clk rise@0.000ns)
  Data Path Delay:         5.476ns  (logic 1.649ns (30.113%)  route 3.827ns (69.887%))
  Logic Levels:            6  (CARRY4=1 LUT3=1 LUT5=1 LUT6=3)
  Clock Path Skew:         -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    2.128ns = ( 7.988 - 5.860 )
    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
```

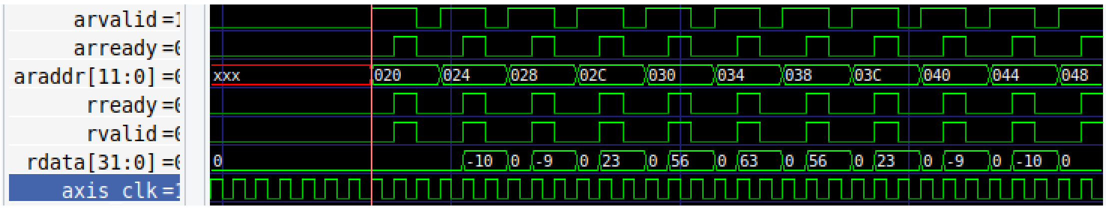
6. Simulation Waveform

1. AXI-lite:

Write:

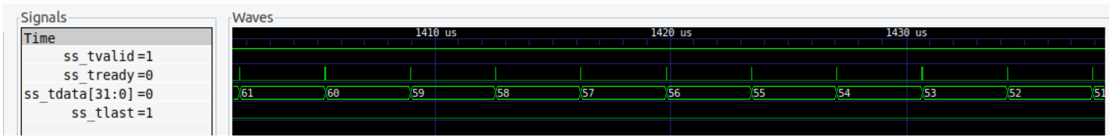


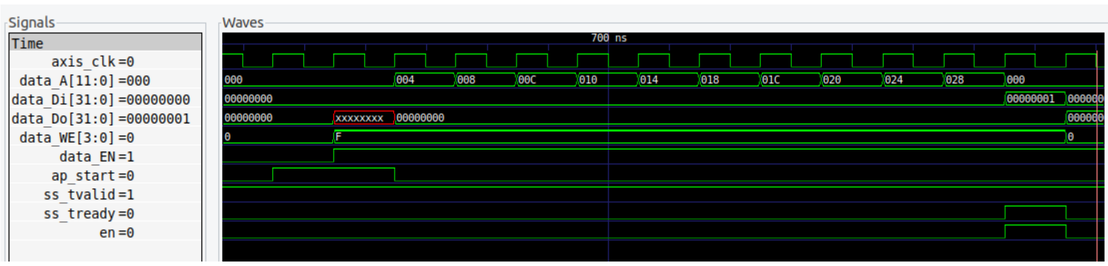
Read:



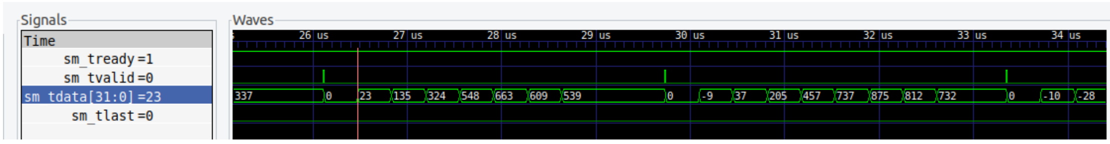
2. AXI-stream:

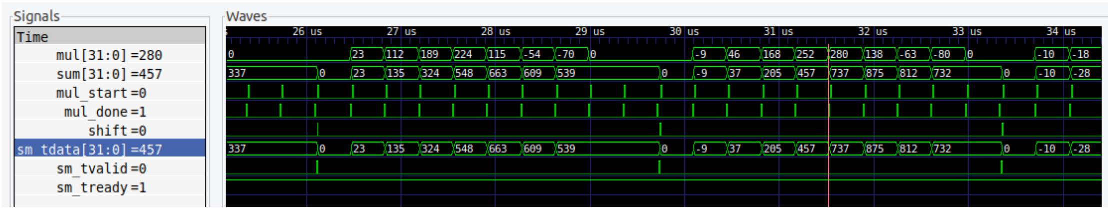
Input:



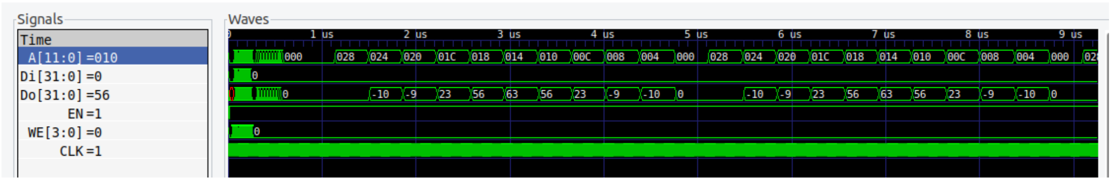


Output:

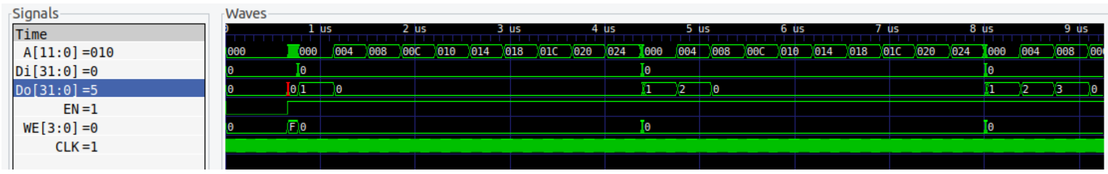




3. Tap_RAM:



4. Data_RAM:



7. Github link

https://github.com/ZheChen-Bill/SoC_Design_Lab3