112-1 SoC Design Laboratory
Lab4-2

Group 10 王語卉 吳至凌 高宇勝

I. Introduction

In lab4-1, we use the firmware code to implement the FIR engine, and we build up the interface of wishbone and user bram.

In this lab, we use the FIR.v design in lab3 to do filtering rather than using the firmware code. On the other hand, we use firmware code to control the dataflow. The two main tasks we've done in lab4-2 are: (1) WB-AXI decoder and interface (both are in **WB_AXI.v**), (2) the firmware code to control the flow (in **counter_la_fir.c**).

There're other things we also put our efforts into: (1) modify **FIR.v** to communicate with the WB successfully, (2) wrap and merge WB-AXI decoder, FIR and the original file in **user_proj_example.counter.v**, (3) define the new MMIO address for user project in **caravel.h**, (4) modify the testbench (**counter_la_fir_tb.v**) to print the received data Y and the timer.

II. Module / Interface breakdown

We will describe each important section of the lab in the following part:

(1) WB-AXI decoder:

In this section, our objective is to interpret signals received from the wishbone interface and subsequently translate them to be compatible with both the AXI Lite and AXI Stream interfaces. We will explain the assignment of those important control signals.

(a) AXI_Lite_EN, AXI_Stream_EN, AXI_Read_EN:

These AXI enable signals are derived based on specific bits (wbs_adr [12, 13, 14]) of the "wbs_adr_i" input signal. They are corresponded to the address we set in the following table.

Table I. Wishbone user-defined address map

ap signals	0x38001000
Data length info	0x38001010
Coefficient	0x38001020 - 0x38001048
Stream data	0x38002000
Stream “last” signal	0x38002008
Coefficient for config read	0x38004020 - 0x38004048

(b) Address translation:

“next_awaddr” and “araddr” are derived addresses for the AXI write and read transactions, respectively. They take the lower bits of the wishbone address and extend with zeros.

(c) Control signal translation:

“next_awvalid” and “next_wvalid” control the validity of AXI write signals.

“next_ss_tvalid”, “next_ss_tdata”,” next_ss_tlast”, “next_sm_tready” control AXI Stream transaction signals.

“next_arvalid” and “rready” control the AXI read transaction signal validity and ready conditions, respectively.

Above signals are all based on the AXI lite conditions (which AXI enable signal is turned on).

(d) Data transfer:

We simply need to assign the data from the wishbone data bus to the write data.

(2) Firmware code to control the whole process (counter_la_fir.c):

We will take a closer look at how the progress happens and how things move along.

(a) Initialization:

We set “reg_mprj_datal” to 0xAB400000, this indicates the start of the whole test.

(b) Variable initialization:

We initialize the variables N = 11 and data_length = 64 for subsequent use in the code.

(c) Sending data length and coefficients:

We set “reg_mprj_data_length” to data length, and send the value to the designated register. Then initializing an array pointer with the output from the fir() function, this is for receiving the coefficient data and input data from our outside c code.

(d) Test execution loop:

We use a loop to iterate three times for the purpose of check whether the whole process can rerun successfully.

In each loop, we send start makers (“reg_mprj_datal = 0x00A50000”) and the ap start signal (“reg_mprj_ap_signal” = 1) to commence the test. Then sending the coefficient data and input data described in (c). After the proceesing of our fir module, we receive the final output.

(e) Test marking:

The is the final step, we send end markers (“reg_mprj_datal” = “reg_la1_data_in” << 24 + 0x005A0000;) to conclude the test. With some display message to check our functionality.

(3) Modifications in our FIR.v:

The two main modifications are:

(a) adding the output” sm_tdata” buffer and some control signals to handshake successfully with “”AXI_stream”. Also, since the WB will send acknowledge signal 11 cycles after handshaking, we also add a counter to avoid receive repeated data.

(b) We modify some control signals and bram initialization to support running the engine three times.

(4) Wrapping all modules into user_proj_example.counter.v:

The module consists of various components with specific functionalities:

(a) Parameters defining the data width and addressing capabilities for Finite Impulse Response (FIR) filtering.

(b) Wires regulating communication protocols and data transmission, essential for the FIR operation.

(c) RAM structures designated for both FIR filter coefficients and data storage.

(d) Additional components for managing user project with the wishbone interface and streaming FIR data.

This module integrates these components to facilitate FIR operations and effective communication with external interfaces.

(5) Modify the testbench:

The modified process is shown as following:

(a) Initialization:

The testbench initiates by waiting for a specific value (16'hAB40) on the “checkbits” signal before commencing the test. Upon initiation, it displays the start of the test sequence.

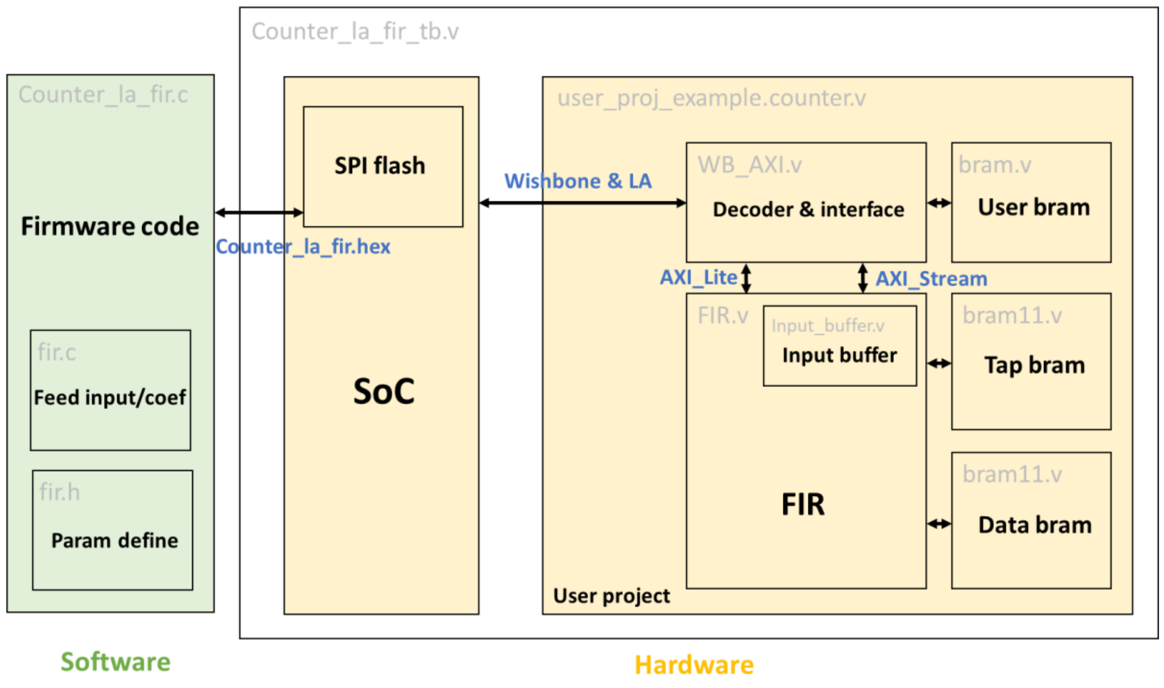
(b) Test loop:

It iterates three times, waiting for the “checkbits” signal to reach 16'h00A5 before executing the test sequence for each iteration.

(c) Execution:

The test instance initiates by monitoring the “checkbits” signal, waiting for the condition 8'h5A. It displays any intermediate changes in the signal and provides the final “checkbits” value upon meeting the condition, alongside the total cycle count for that specific instance.

1. Design block diagram – datapath, control-path



Datapath: WB -> WB-AXI (decoder) -> axi_stream -> FIR -> axi_stream -> LA

Control: WB -> WB-AXI (decoder) -> axi_lite -> FIR -> axi_lite -> WB/LA

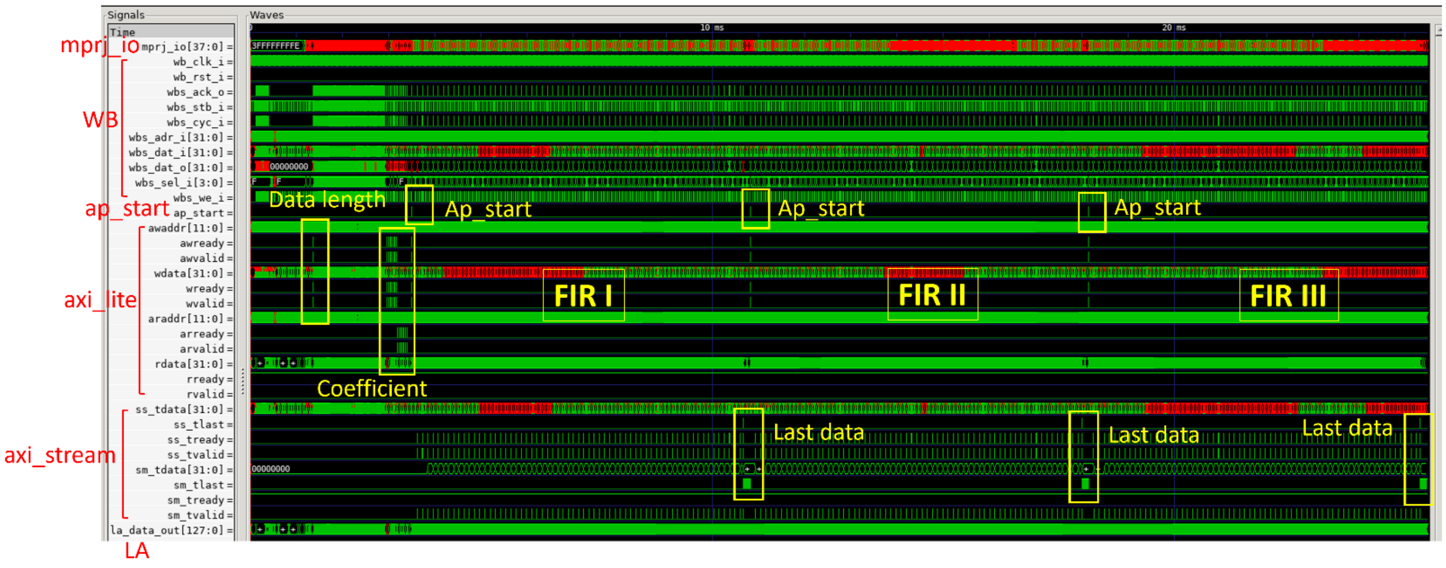
2. The interface protocol between firmware, user project and testbench

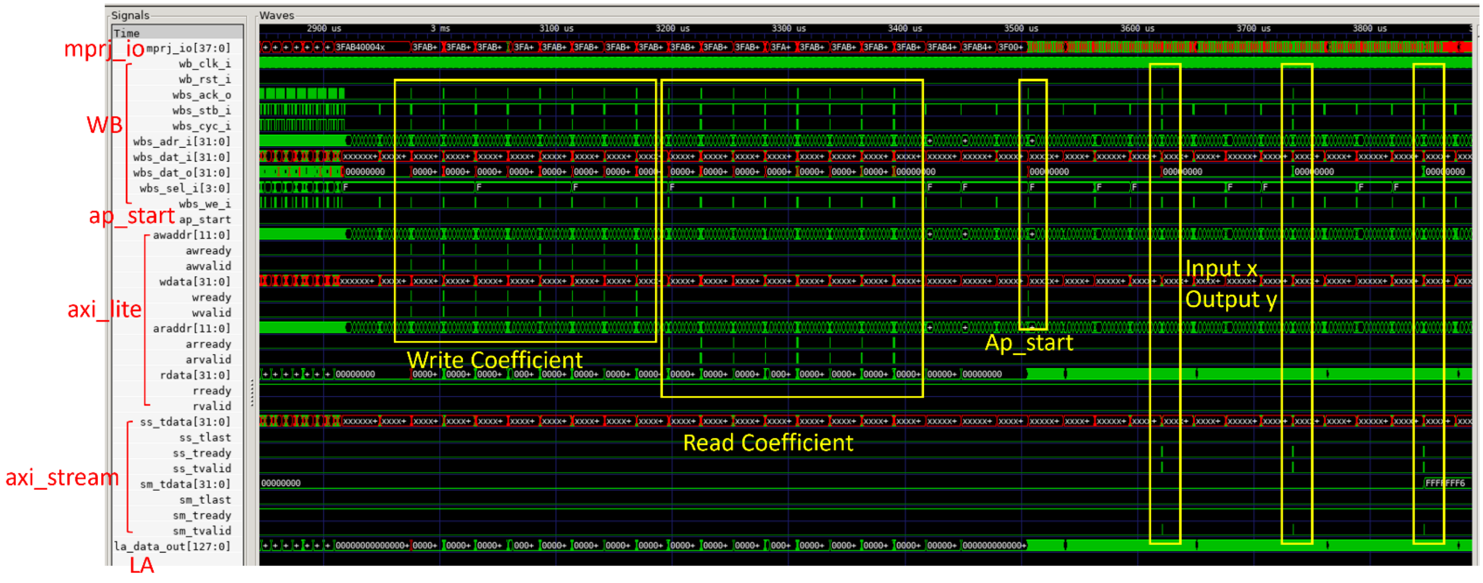
The interface protocols are listed in the block diagram above.

Firmware code are translated into hex file and stored into SPI flash. We use user defined address to map the parameter in firmware to the hardware signals. The SoC and the testbench are all use the user defined hardware mapping. SoC use WB or LA to communicate with the user project, then use AXI_lite/stream to send to the FIR engine.

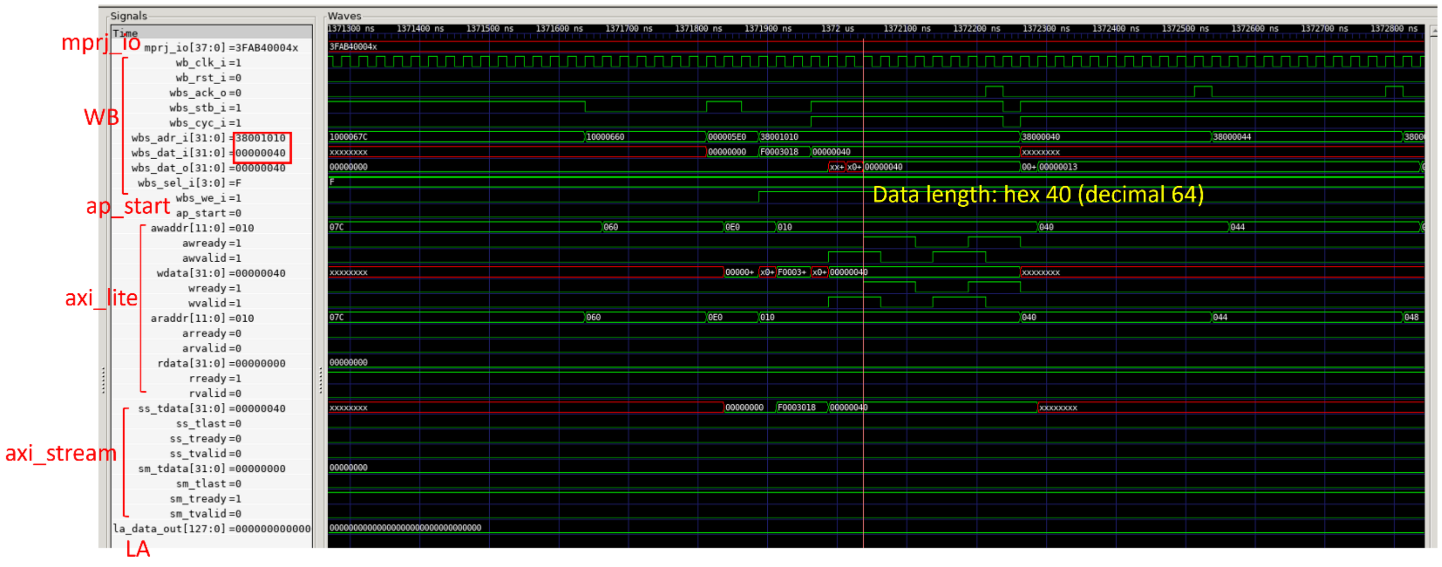
3. Waveform and analysis of the hardware/software behavior

The overall waveform

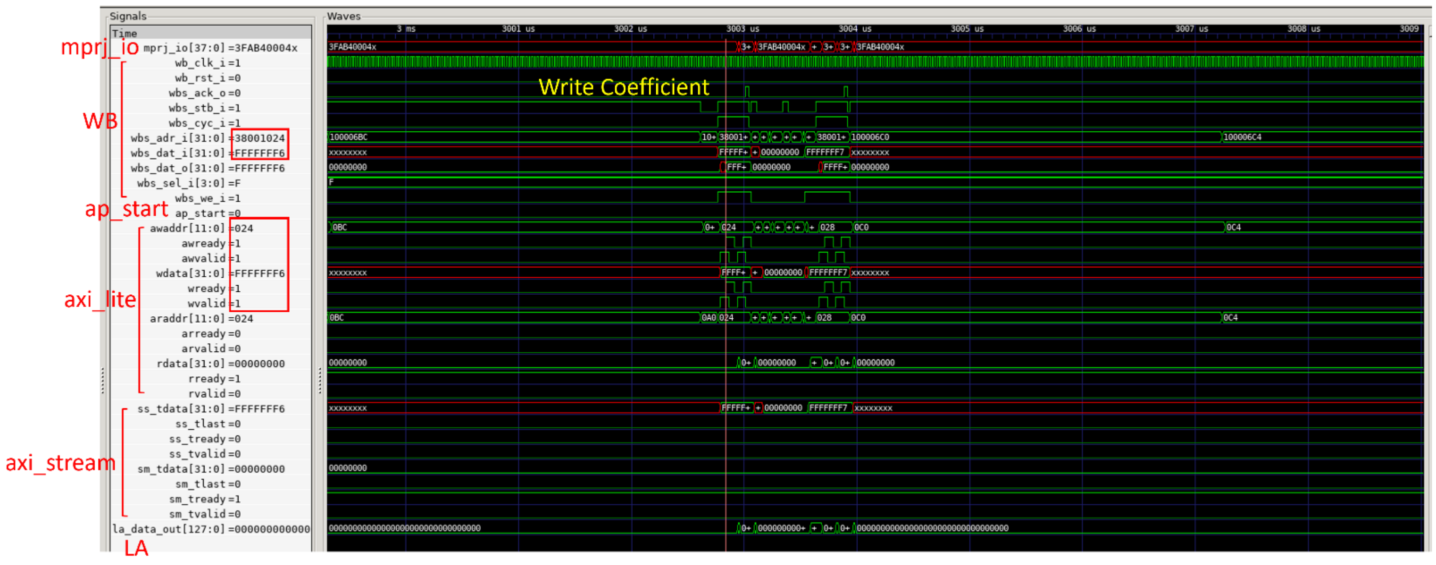




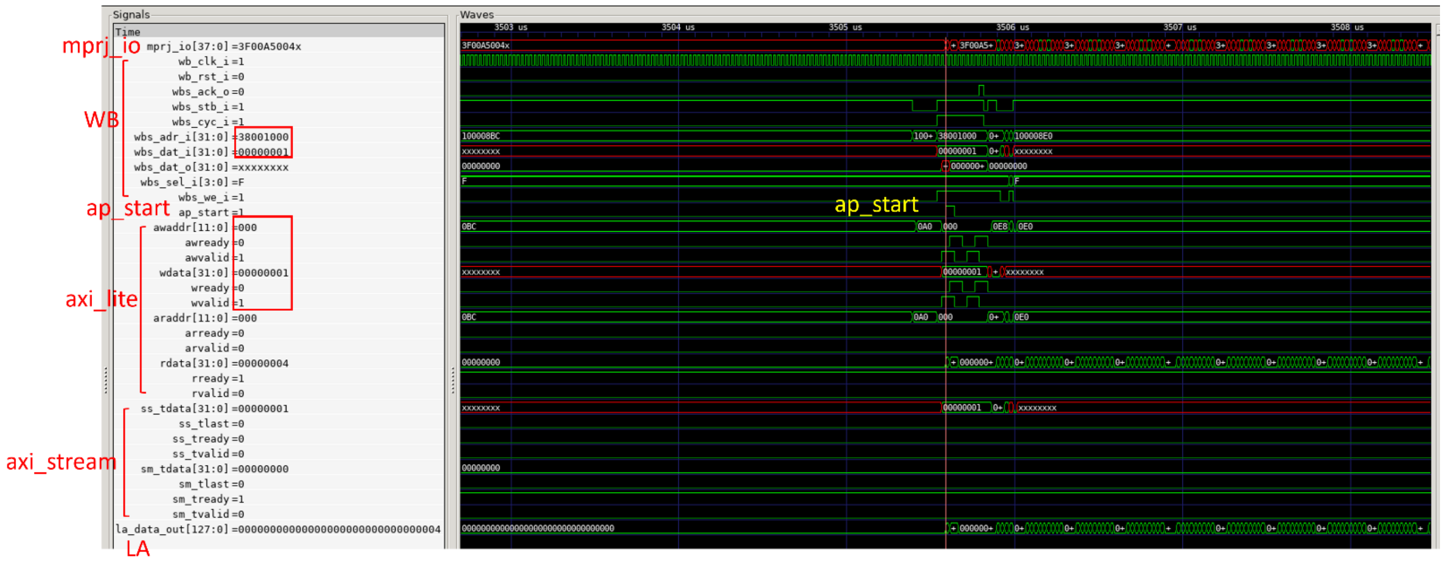
From the WB interface, i.e., WB address and WB data input, we can observe the mapping of the MMIO address and the data we assign in firmware code.

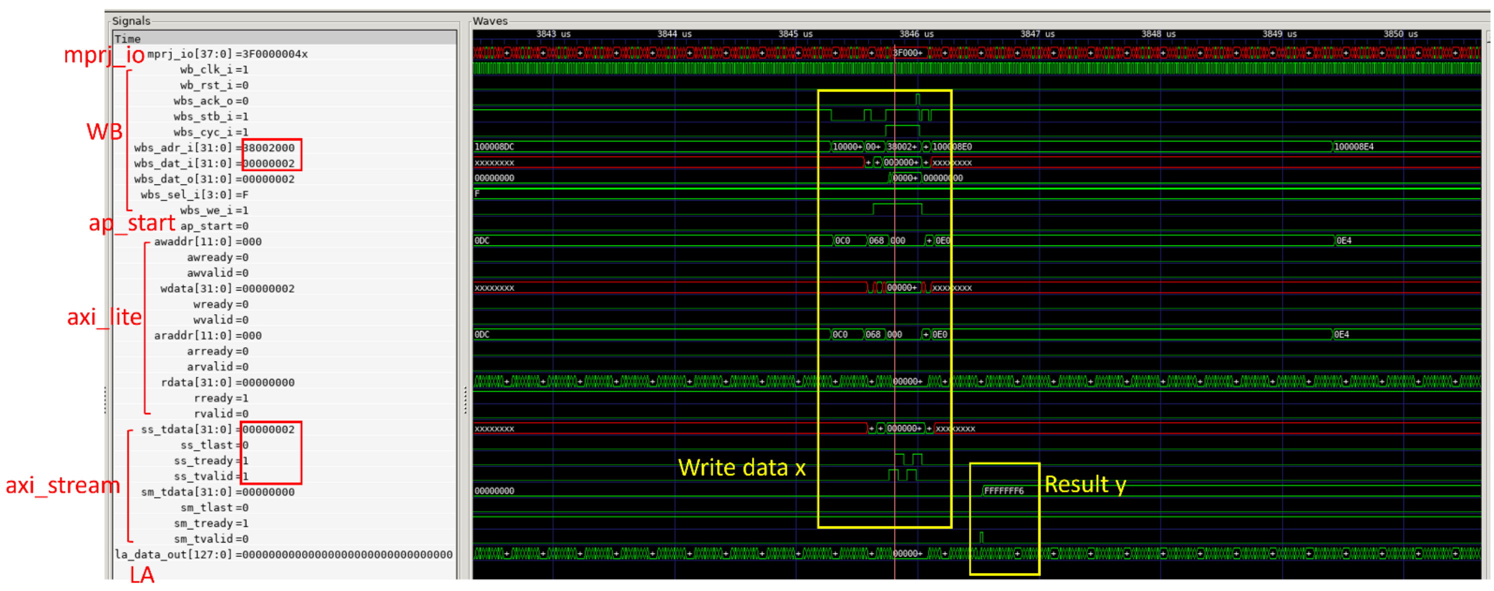


The wishbone address 0x38001010 correspond to **data length information** as we defined in user-defined address.

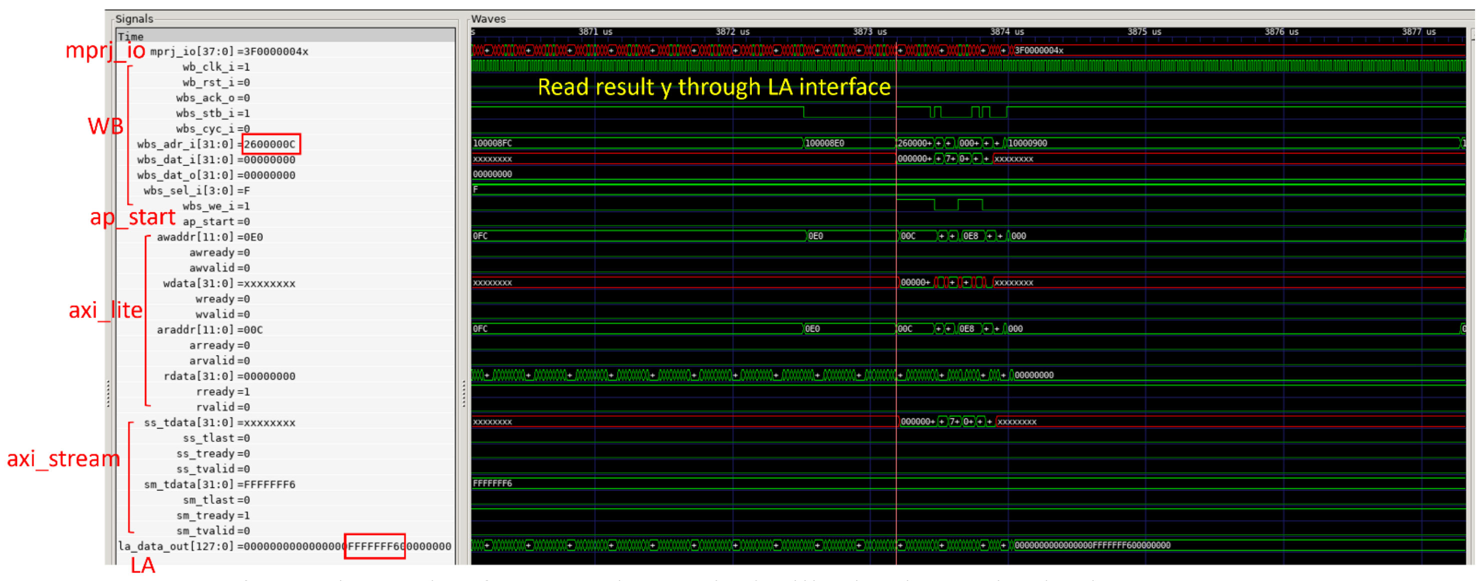


The wishbone address 0x38001024 correspond to **coefficient address.**

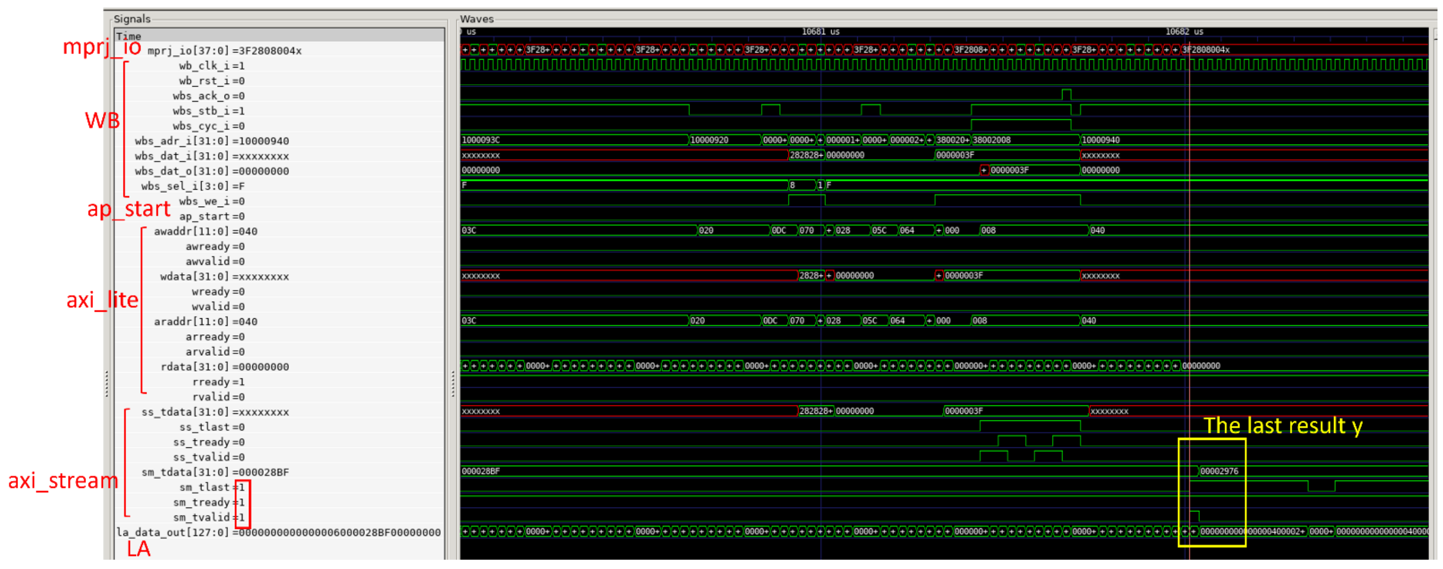




The wishbone address 0x38002000 correspond to **input stream data x.**



After read a result y from LA, the terminal will print the received value.



4. **What is the FIR engine theoretical throughput, i.e. data rate? Actually measured throughput?**

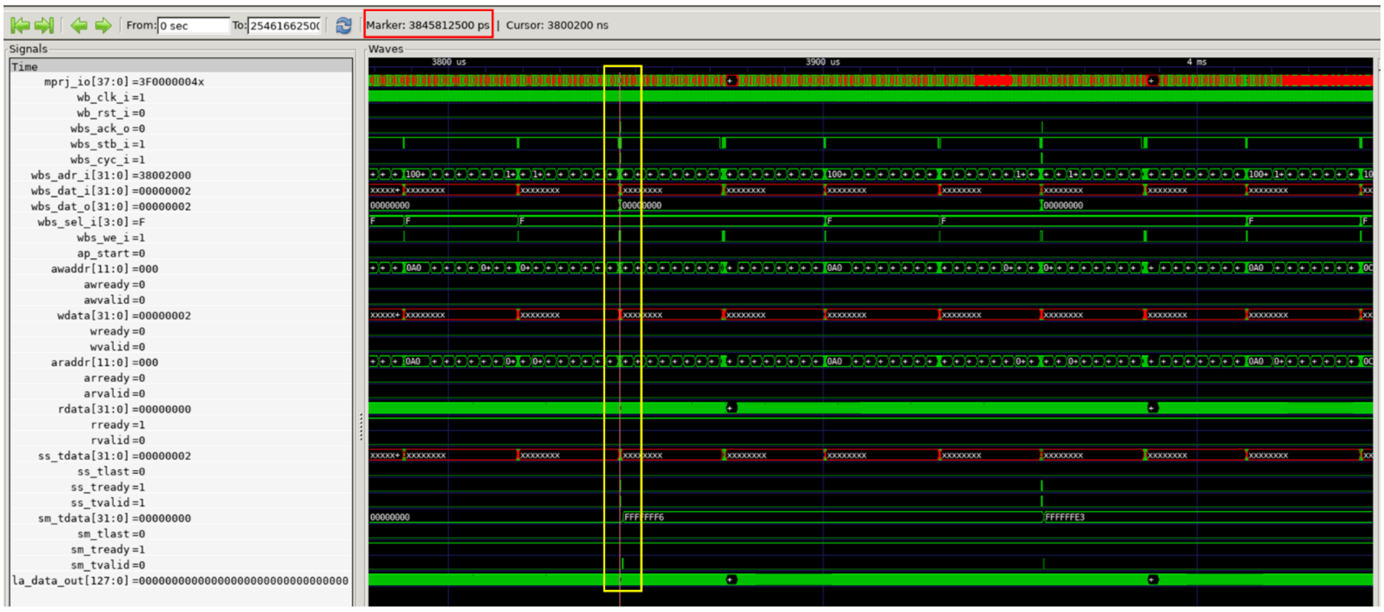
   The throughput in Lab3 is 6607 cycle/ 600 data, **11cycles/data**.

   The throughput measured in this lab is 290335 cycle/ 64 data, **4537cycles/data**.

5. **What is latency for firmware to feed data?**

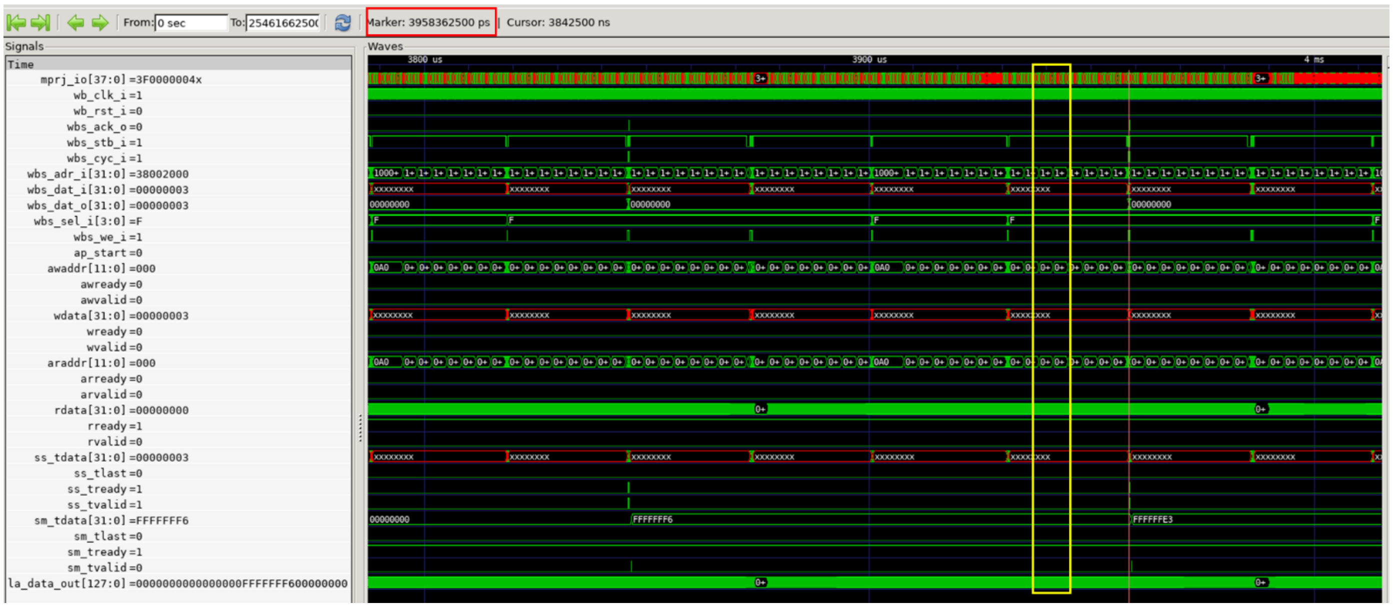
   The clock period is 25000ps/cycle

   (3958362500ps - 3845812500ps) /25000ps = 112550000ps/25000ps = 4502 cycles



From: 0 sec To: 2546166250(Marker: 3845812500 ps | Cursor: 3800200 ns

```
Signals
Time
      mprj_io[37:0] =3F0000004x
          wb_clk_i =1
          wb_rst_i =0
         wbs_ack_o =0
         wbs_stb_i =1
         wbs_cyc_i =1
  wbs_adr_i[31:0] =38002000
  wbs_dat_i[31:0] =00000002
  wbs_dat_o[31:0] =00000002
   wbs_sel_i[3:0] =F
          wbs_we_i =1
          ap_start =0
      awaddr[11:0] =000
           awready =0
           awvalid =0
     wdata[31:0] =00000002
            wready =0
            wvalid =0
      araddr[11:0] =000
           arready =0
           arvalid =0
     rdata[31:0] =00000000
            rready =1
            rvalid =0
  ss_tdata[31:0] =00000002
          ss_tlast =0
         ss_tready =1
         ss_tvalid =1
  sm_tdata[31:0] =00000000
          sm_tlast =0
         sm_tready =1
         sm_tvalid =0
la_data_out[127:0] =00000000000000000000000000000000
```



From: 0 sec To: 2546166250(Marker: 3958362500 ps | Cursor: 3842500 ns

```
Signals
Time
      mprj_io[37:0] =3F0000004x
          wb_clk_i =1
          wb_rst_i =0
         wbs_ack_o =0
         wbs_stb_i =1
         wbs_cyc_i =1
  wbs_adr_i[31:0] =38002000
  wbs_dat_i[31:0] =00000003
  wbs_dat_o[31:0] =00000003
   wbs_sel_i[3:0] =F
          wbs_we_i =1
          ap_start =0
      awaddr[11:0] =000
           awready =0
           awvalid =0
     wdata[31:0] =00000003
            wready =0
            wvalid =0
      araddr[11:0] =000
           arready =0
           arvalid =0
     rdata[31:0] =00000000
            rready =1
            rvalid =0
  ss_tdata[31:0] =00000003
          ss_tlast =0
         ss_tready =1
         ss_tvalid =1
  sm_tdata[31:0] =FFFFFFF6
          sm_tlast =0
         sm_tready =1
         sm_tvalid =0
la_data_out[127:0] =0000000000000000FFFFFFF600000000
```

The latency for firmware to feed data is **4502 cycles**.

6. **What techniques used to improve the throughput?**

   Since the latency to feed the input is very large, we know that one firmware code needs a huge number of clock cycles to execute. Therefore, we should use the firmware code as less as possible to improve the throughput.

   The other way to improve the throughput is to parallel feed the input. We can use LA interface to send four data in one time, then use the FIFO in WB-AXI to store the remaining data.