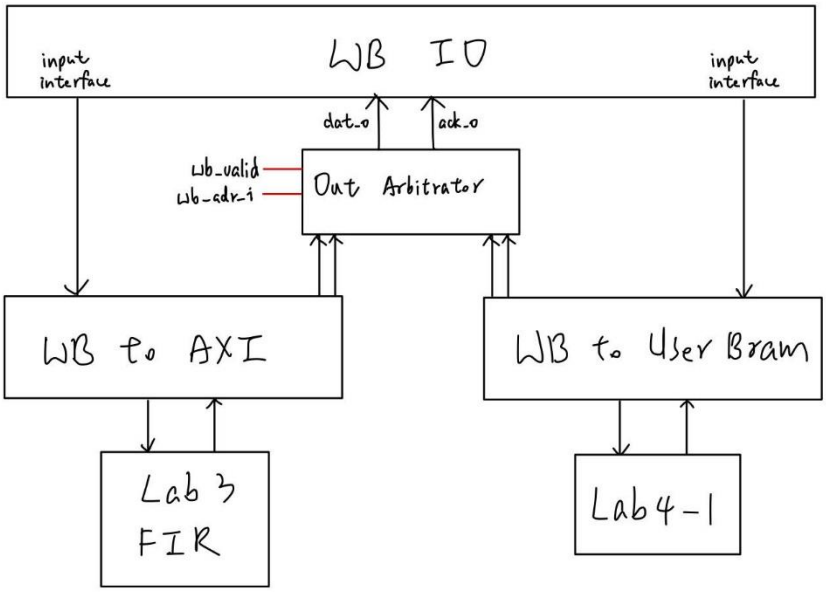
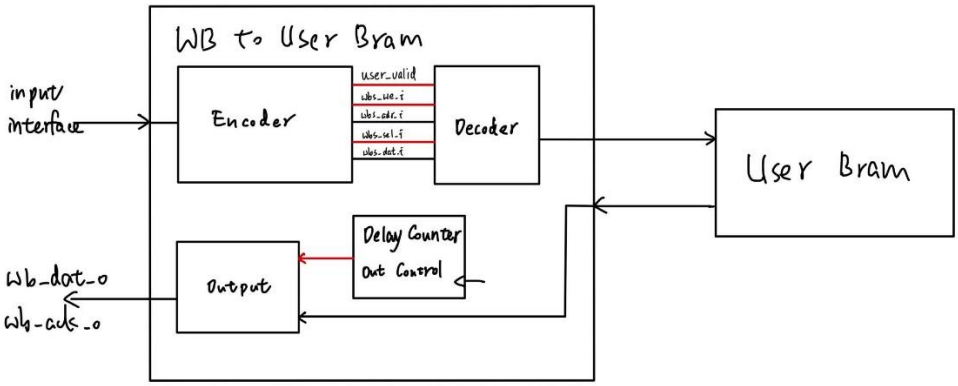
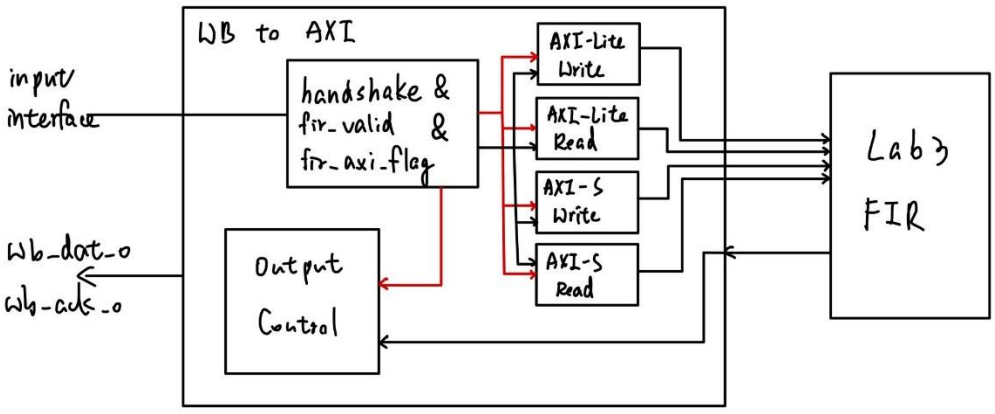
# Lab4-2 Report

**Block diagram:**

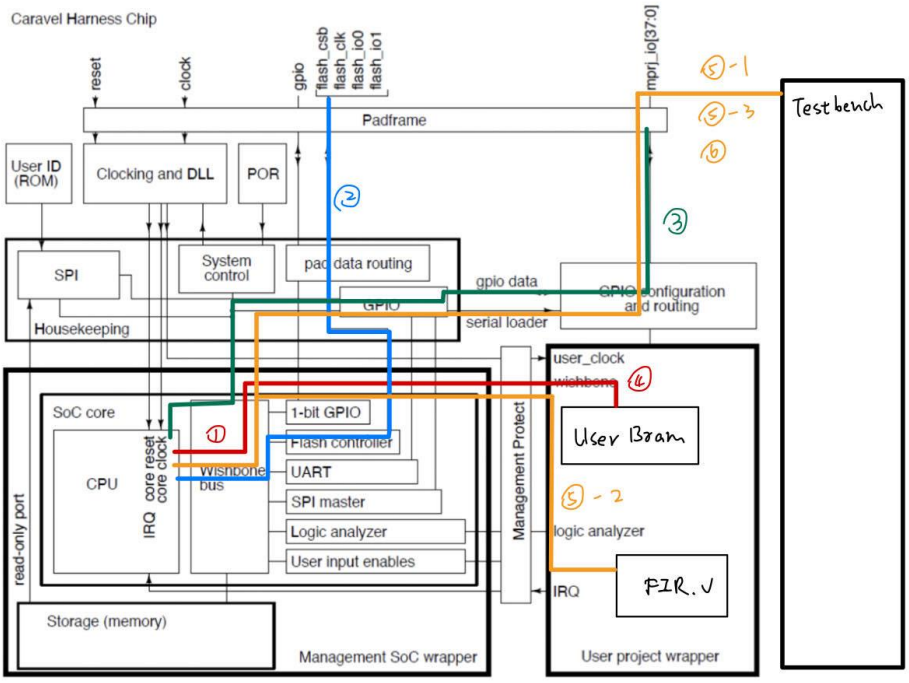



**Read line -> control path** **black line -> data path**

**The interface protocol between firmware, user project and testbench:**




1. CPU stores parts of the compiled firmware codes (initfir and fir) into the user-bram inside the User-Project through Wishbond interface.
2. CPU executes the main function defined in counter_la_fir.c where the compiled assembly codes are obtained from the flash memory through Wishbond interface.
3. CPU configures the mprj_io pins [31:16] such that it can be seen by the testbench.
4. CPU reads the fir assembly codes from user_bram to execute through Wishbond interface.
5. 

5-1. CPU sends the start flag to mprj_io through Wishbond and let the testbench start the latency counter.

5-2. CPU sends the configuration to FIR ip, sends the input data and receives the output data from the FIR ip.

5-3. CPU places the output from FIR ip on the mprj_io, whose [31:16] pins are connected by the checkbits inside the testbench to check the answer.

6. CPU sends the end flag (0x5A) to the testbench. Testbench ends the latency counter and waits for the next configuration till the main process done.

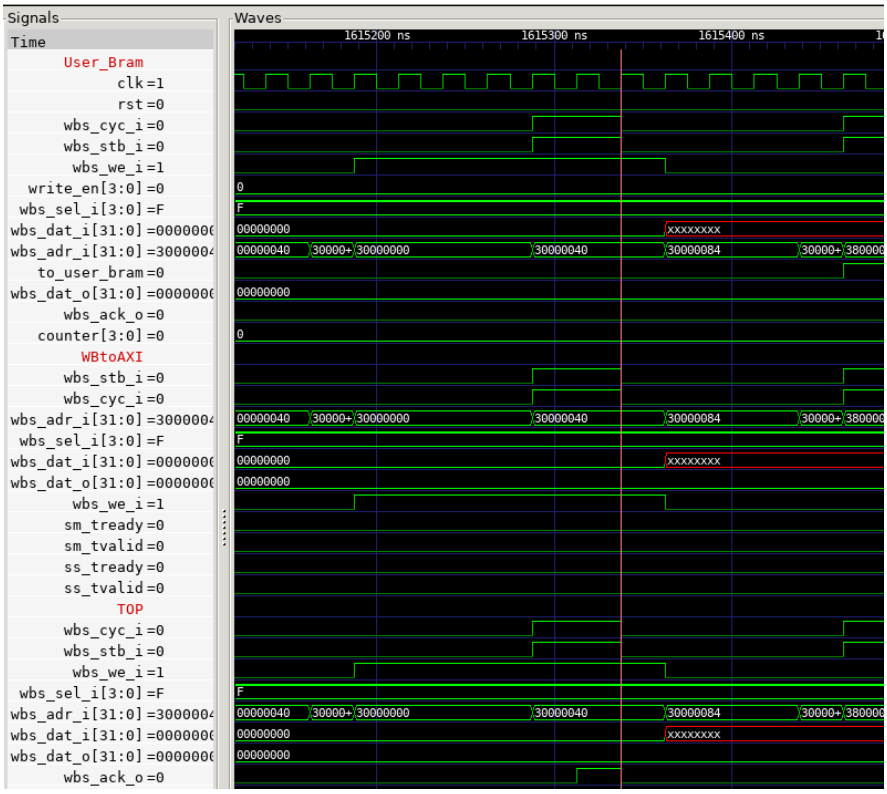
**Firmware < ------ > Wishbond < ------ > User Project**

**Firmware ------ > Wishbond ------ > Testbench (mprj_io)**

**Waveform and analysis of the hardware/software behavior.**

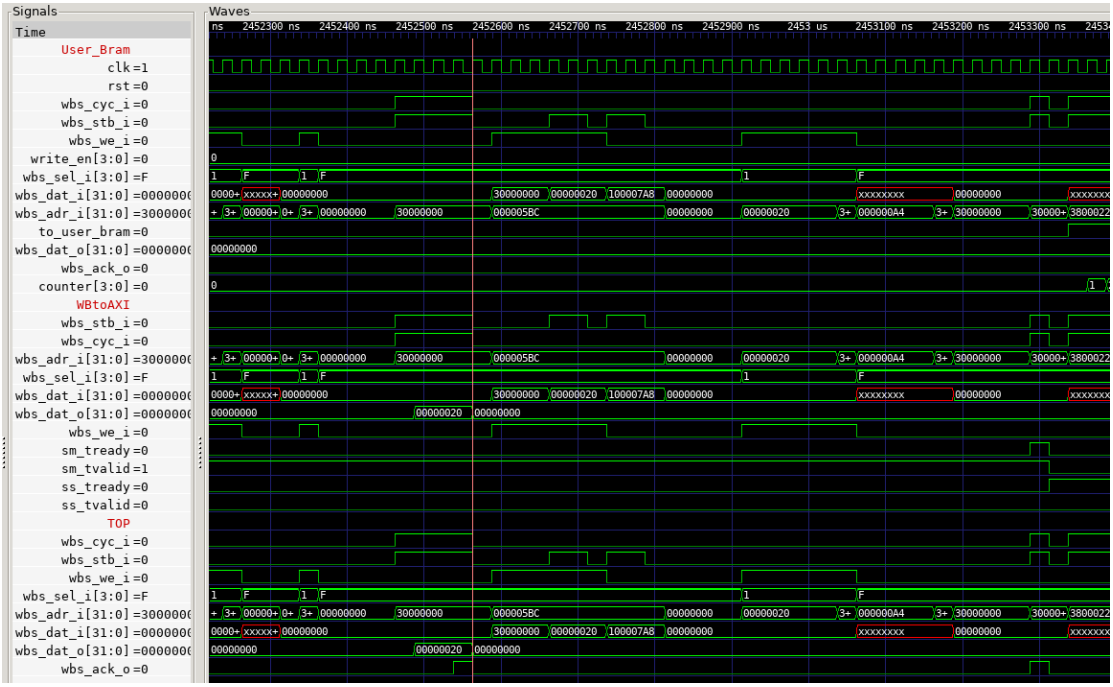
```c
#include "fir.h"
#include <defs.h>

void __attribute__ ( ( section ( ".mprjram" ) ) ) initfir() {
    //initial your fir
    // {0,-10,-9,23,56,63,56,23,-9,-10,0};
    tap_1 = 0;
    tap_2 = -10;
    tap_3 = -9;
    tap_4 = 23;
    tap_5 = 56;
    tap_6 = 63;
    tap_7 = 56;
    tap_8 = 23;
    tap_9 = -9;
    tap_10= -10;
    tap_11= 0;

    datalength = 64;

    reg_mprj_datal = 0x00A50000;
    status = 0x00000001;

}
```

```c
// fir input X[n]
#define inputsignal         (*(volatile uint32_t*)0x30000080)
// fir output Y[n]
#define outputsignal        (*(volatile uint32_t*)0x30000084)
// taps[n]
#define tap_1       (*(volatile uint32_t*)0x30000040)
#define tap_2       (*(volatile uint32_t*)0x30000044)
#define tap_3       (*(volatile uint32_t*)0x30000048)
#define tap_4       (*(volatile uint32_t*)0x3000004c)
#define tap_5       (*(volatile uint32_t*)0x30000050)
#define tap_6       (*(volatile uint32_t*)0x30000054)
#define tap_7       (*(volatile uint32_t*)0x30000058)
#define tap_8       (*(volatile uint32_t*)0x3000005c)
#define tap_9       (*(volatile uint32_t*)0x30000060)
#define tap_10      (*(volatile uint32_t*)0x30000064)
#define tap_11      (*(volatile uint32_t*)0x30000068)
// data length
#define datalength (*(volatile uint32_t*)0x30000010)
// status
#define status      (*(volatile uint32_t*)0x30000000)
```




When software assigns a value to the predefined address, CPU will start a Wishbond write cycle to send the value to that address space. Take the above pictures as example, when the software wants to initialize the tap_1 which is defined at 0x30000040, the CPU will generate a Wishbond transaction to write the value to 0x30000040. The same things happen when we want to assign value to other predefined address.

```
int* __attribute__ ( ( section ( ".mprjram" ) ) ) fir(int method){
    initfir();
    //write down your fir

    int x[64];
    int s;
    for (int i = 0; i < 64; i++) {
        if(method == 1)
            x[i] = i;
        else if(method == 2)
            x[i] = 64 - i;
        else if(method == 3)
            x[i] = 1;
    }

    s = status;
    for (int i = 0; i < 64; i++) {
        while (!((s >> 4) & 1) && i != 0)
            s = status;
        inputsignal = x[i];

        while (!((s >> 5) & 1))
            s = status;
        ans[i] = outputsignal;
    }

    s = status;
    reg_mprj_datal = ((0x000000FF & ans[63]) << 24) | 0x005A0000;

    return ans;
}
```



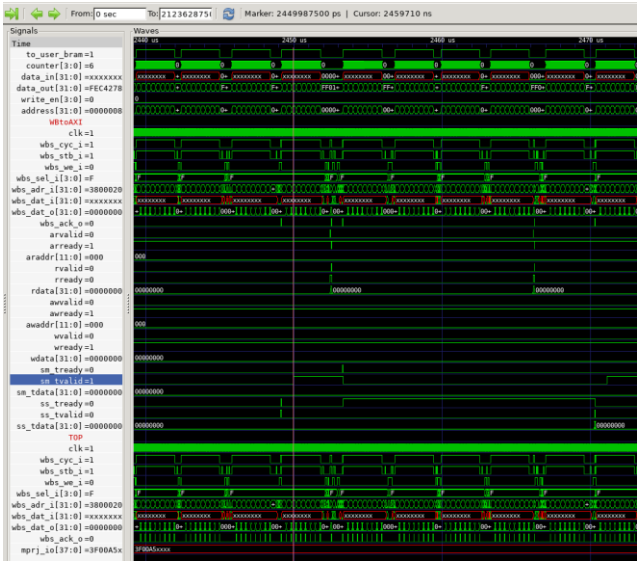

When the software extracts the value from a variable with predefined address, CPU will start a read cycle to that address, those slaves whoever have the control over that specific address space will return the corresponding data to CPU through Wishbond. For example, when software wants to get the value at the status's address, CPU will start the read cycle of Wishbond and then get the value at that specific address.

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

When computing the fir, once we get the input data from AXI-Stream, we can directly start the multiplication and accumulation process. At the same time, the input data is held and stored into the data-bram until the answer is fully computed. At this moment, we can perform handshake and send the output data. Hence, we only need 11 + 1 cycles to compute a result point theoretically.

**Theoretical throughput: 1/12 (1/cycle)**

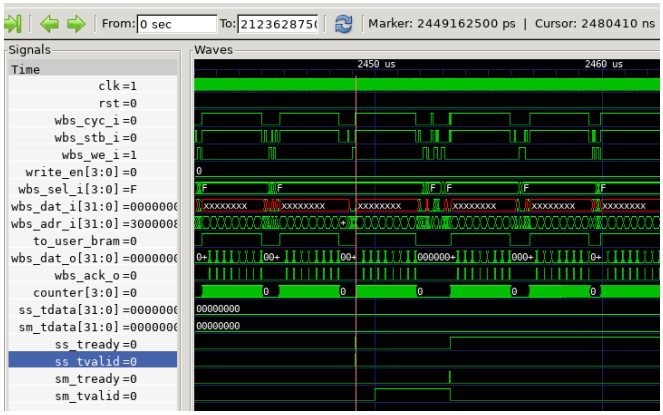
**Data rate: 12 (cycle)**

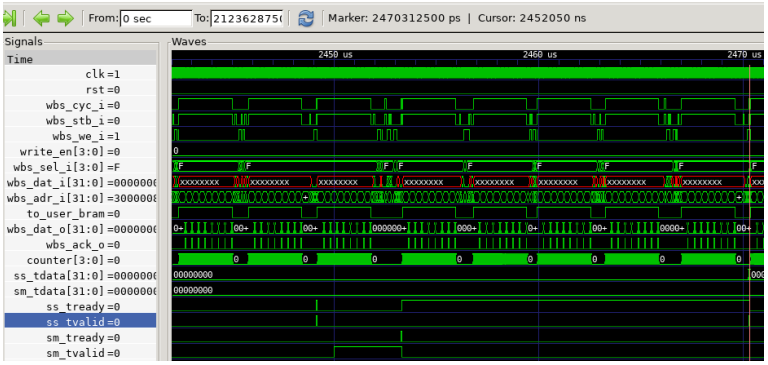


**Our true throughput is about $(2449987500 - 2449137500)_{ps} / (25000)_{ps}$ = 31 cycles**

Reason: nonefficient usage of data bram.

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





**Latency: $(2470312500 - 2449162500)_{ps} / (25000)_{ps}$ = 847 cycles**

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

1. Take use of the separated read and write port bram, we can hide the previous write process in the current read process.
2. Let Wishbond to AXI interface pass signals without latching data to avoid extra latency.

**Does bram12 give better performance, in what way?**

If the limitation on the MAC hardware still exists, using bram12 will not give better performance over bram11, since you still need to compute the fir in 11 cycles.

If we can use two MACs, we can pipeline the tap coefficients sent into the MACs and we need to store two data into the bram in continuous addresses. By doing this, we can compute two results without access the same data multiple times, thus giving better performance.

**Can you suggest other method to improve the performance?**

1. Provide more multipliers and adders to improve the parallelism of fir process.
2. Use shift pointer instead of actually shifting the data in data bram to mimic the behavior of shift registers.
3. Use shift registers instead of bram to access input data and tap coefficient with no extra latency.
4. Use an output buffer along with multiple and continuous data input to hide the calculation latency between receiving and sending data.

**More discussion:**

In this project, we have three bram units, user bram, data bram and tap bram, but only one of them is synthesized to real Block RAM on FPGA. This is because the user_bram is prompt with the synthesis command: (* ram_style = "block" *).

2. Memory
---

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Block RAM Tile | 1 | 0 | 0 | 140 | 0.71 |
| RAMB36/FIFO* | 1 | 0 | 0 | 140 | 0.71 |
| RAMB36E1 only | 1 | | | | |
| RAMB18 | 0 | 0 | 0 | 280 | 0.00 |

Other two brams are synthesized into distributed RAMs which are implemented by

LUTs.

Distributed RAM: Preliminary Mapping Report (see note below)

| Module Name | RTL Object | Inference | Size (Depth x Width) | Primitives |
|---|---|---|---|---|
| user_proj_example | data_ram/RAM_reg | Implied | 16 x 32 | RAM16X1S x 32 |
| user_proj_example | tap_ram/RAM_reg | Implied | 16 x 32 | RAM16X1S x 32 |

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice LUTs* | 460 | 0 | 0 | 53200 | 0.86 |
| LUT as Logic | 396 | 0 | 0 | 53200 | 0.74 |
| LUT as Memory | 64 | 0 | 0 | 17400 | 0.37 |
| LUT as Distributed RAM | 64 | 0 | | | |
| LUT as Shift Register | 0 | 0 | | | |
| Slice Registers | 379 | 0 | 0 | 106400 | 0.36 |
| Register as Flip Flop | 379 | 0 | 0 | 106400 | 0.36 |
| 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 |