# Report – Lab 2

R13943009 鄭至盛

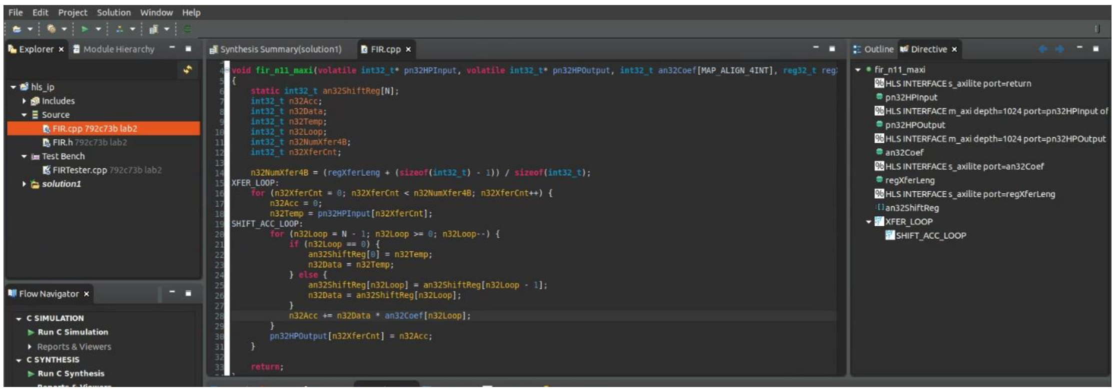
## Part 1 – AXI-Master

### Overview of the HLS Design:

#### The C code

The .h file simply defines our top function (multip_2num) & data type (int32_t).

The .cpp file defines a shift register to store the temporary filtered value & the static for-loop that describes the behavior ($A = A + B \times C$) & boundary condition of the FIR filter. The directives are set up with a .tcl file, containing AXI-Lite for the configurations & AXI-Master for the I/O samples.



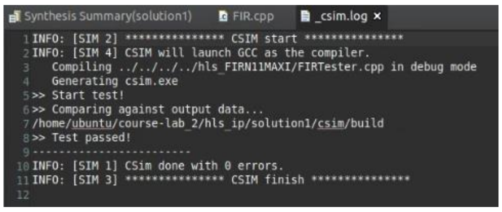
```
void fir_n11_maxi(volatile int32_t* pn32HPInput, volatile int32_t* pn32HPOutput, int32_t an32Coef[MAP_ALIGN_4INT], reg32_t reg
{
    static int32_t an32ShiftReg[N];
    int32_t n32Acc;
    int32_t n32Data;
    int32_t n32Temp;
    int32_t n32Loop;
    int32_t n32NumXfer4B;
    int32_t n32XferCnt;

    n32NumXfer4B = (regXferLeng + (sizeof(int32_t) - 1)) / sizeof(int32_t);
XFER_LOOP:
    for (n32XferCnt = 0; n32XferCnt < n32NumXfer4B; n32XferCnt++) {
        n32Acc = 0;
        n32Temp = pn32HPInput[n32XferCnt];
SHIFT_ACC_LOOP:
        for (n32Loop = N - 1; n32Loop >= 0; n32Loop--) {
            if (n32Loop == 0) {
                an32ShiftReg[0] = n32Temp;
                n32Data = n32Temp;
            } else {
                an32ShiftReg[n32Loop] = an32ShiftReg[n32Loop - 1];
                n32Data = an32ShiftReg[n32Loop];
            }
            n32Acc += n32Data * an32Coef[n32Loop];
        }
        pn32HPOutput[n32XferCnt] = n32Acc;
    }

    return;
```

```
fir_n11_maxi
    HLS INTERFACE s_axilite port=return
    pn32HPInput
    HLS INTERFACE m_axi depth=1024 port=pn32HPInput of
    pn32HPOutput
    HLS INTERFACE m_axi depth=1024 port=pn32HPOutput
    an32Coef
    HLS INTERFACE s_axilite port=an32Coef
    regXferLeng
    HLS INTERFACE s_axilite port=regXferLeng
    an32ShiftReg
    XFER_LOOP
        SHIFT_ACC_LOOP
```

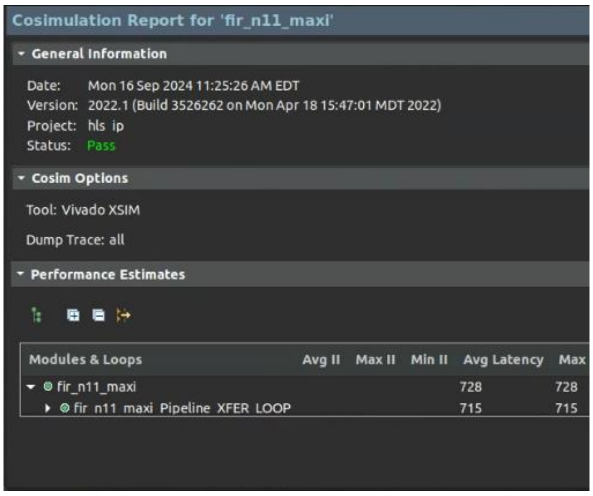
The tester file first set up the taps & I/O array for the HLS IP. Then, it would verify the NUM_SAMPLES of the filtered output & the golden data. By calling the diff command with system() API, we can determine whether the IP behaves correctly.

#### C-simulation (left) & Co-simulation (right)

The co-sim error would occur if the DEPTH of the testbench & that of the directive is mismatched.



```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../../hls_FIRN11MAXI/FIRTester.cpp in debug mode
   Generating csim.exe
>> Start test!
>> Comparing against output data...
/home/ubuntu/course-lab_2/hls_ip/solution1/csim/build
>> Test passed!
-----------------------
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```



Cosimulation Report for 'fir_n11_maxi'

General Information

| | |
|---|---|
| Date: | Mon 16 Sep 2024 11:25:26 AM EDT |
| Version: | 2022.1 (Build 3526262 on Mon Apr 18 15:47:01 MDT 2022) |
| Project: | hls_ip |
| Status: | Pass |

Cosim Options

Tool: Vivado XSIM

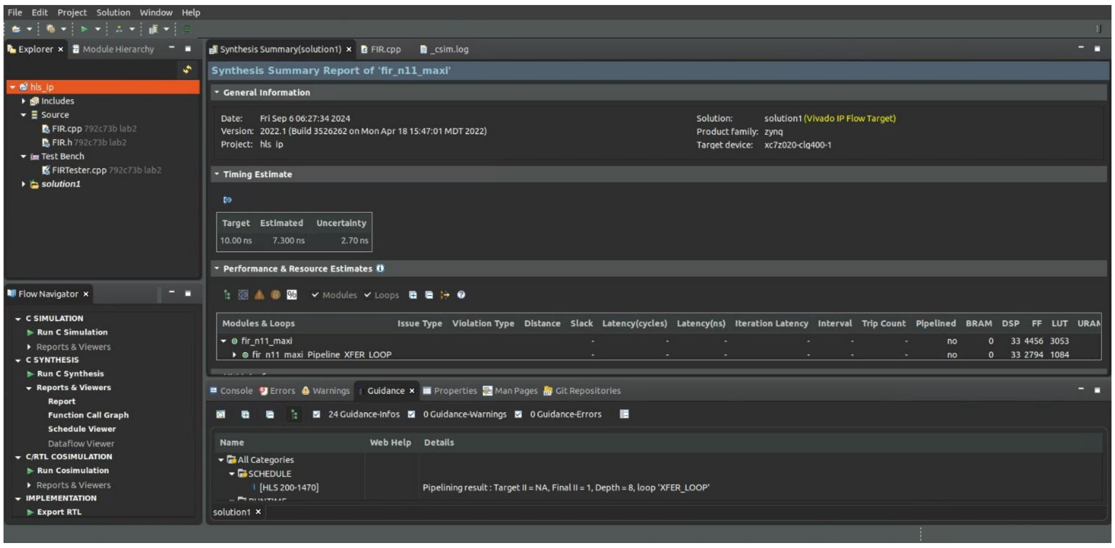
Dump Trace: all

Performance Estimates

| Modules & Loops | Avg II | Max II | Min II | Avg Latency | Max |
|---|---|---|---|---|---|
| fir_n11_maxi | | | | 728 | 728 |
| fir_n11_maxi_Pipeline_XFER_LOOP | | | | 715 | 715 |

## C-synthesis

The design is synthesized with clock period=10ns



Synthesis Summary Report of 'fir_n11_maxi'

General Information

| | | | |
|---|---|---|---|
| Date: | Fri Sep 6 06:27:34 2024 | Solution: | solution1 (Vivado IP Flow Target) |
| Version: | 2022.1 (Build 3526262 on Mon Apr 18 15:47:01 MDT 2022) | Product family: | zynq |
| Project: | hls_ip | Target device: | xc7z020-clg400-1 |

Timing Estimate

| Target | Estimated | Uncertainty |
|---|---|---|
| 10.00 ns | 7.300 ns | 2.70 ns |

Performance & Resource Estimates

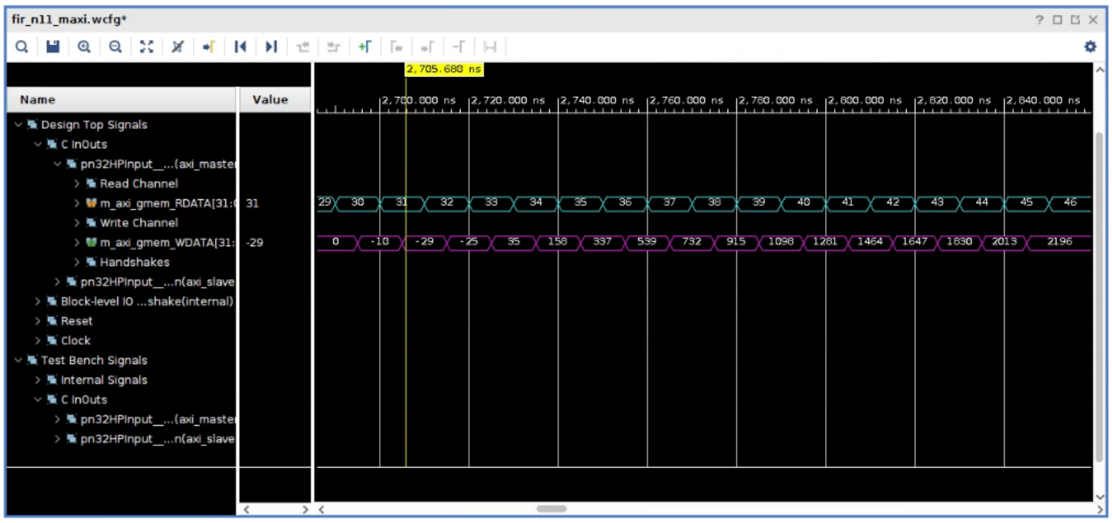
| Modules & Loops | Issue Type | Violation Type | Distance | Slack | Latency(cycles) | Latency(ns) | Iteration Latency | Interval | Trip Count | Pipelined | BRAM | DSP | FF | LUT | URAM |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| fir_n11_maxi | | | | - | - | - | - | - | - | no | 0 | 33 | 4456 | 3053 | |
| fir_n11_maxi_Pipeline_XFER_LOOP | | | | - | - | - | - | - | - | no | 0 | 33 | 2794 | 1084 | |

24 Guidance-Infos   0 Guidance-Warnings   0 Guidance-Errors

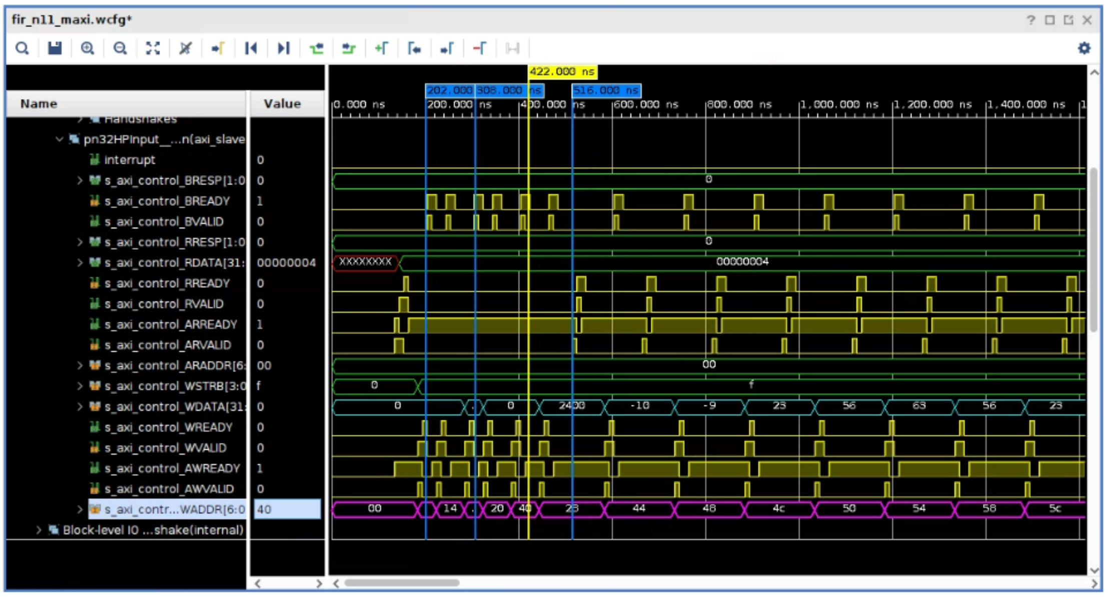
| Name | Web Help | Details |
|---|---|---|
| All Categories | | |
| SCHEDULE | | |
| [HLS 200-1470] | | Pipelining result : Target II = NA, Final II = 1, Depth = 8, loop 'XFER_LOOP' |

## Generated Waveform

The aqua/purple part is the input/output signal of the RTL part.



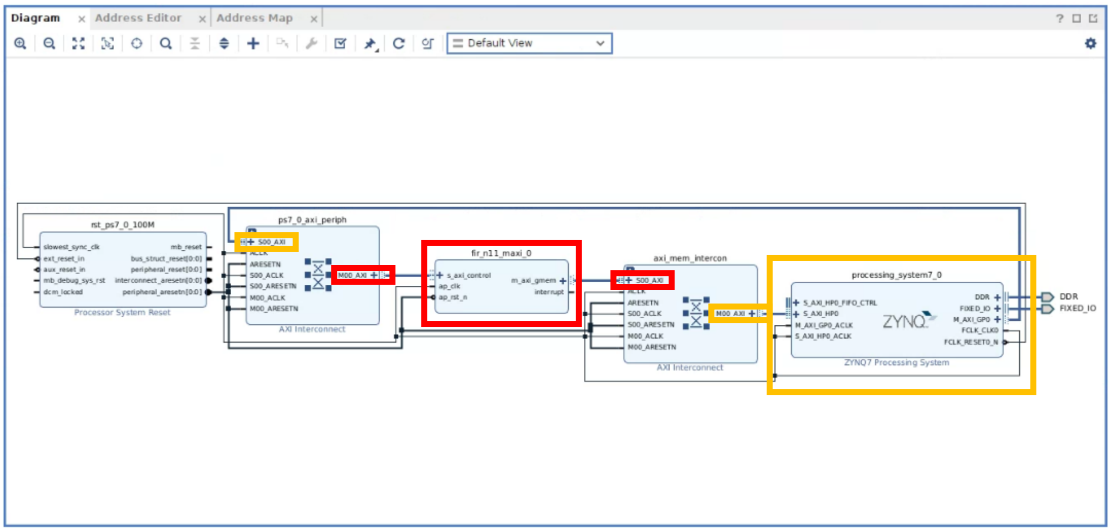
The aqua/purple part & the marker reveal the data/address signal of the RTL part, including addresses of I/O buffers, the coefficient of each tap, and the number of samples.



## Overview of the Hardware Wrapper:

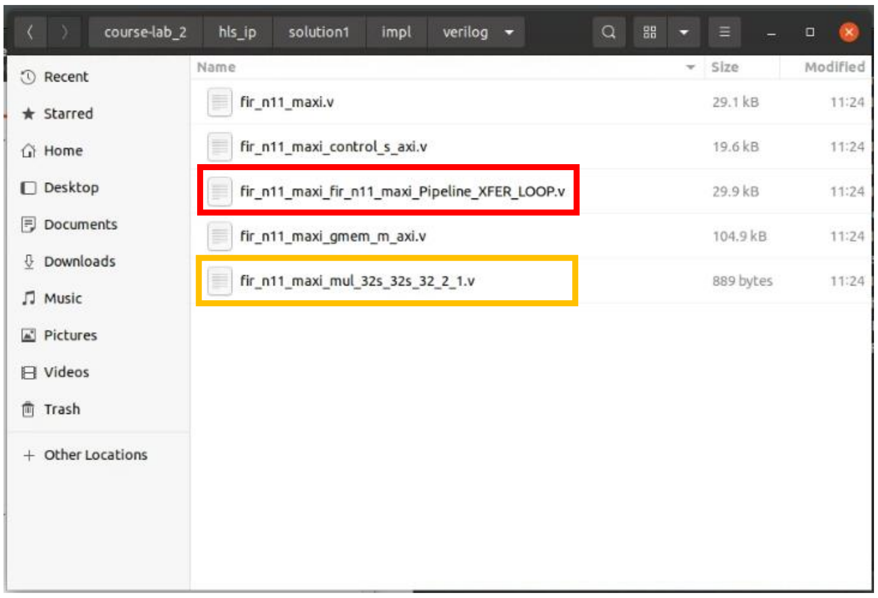
### Block Diagram

In the block diagram, we can see 2 AXI peripherals are connected between the generated HLS IP (fir_n11_maxi_0) & ZYNQ7 Processing System. With the protocol of AXI-Lite, the configuration such as the tap coefficient & buffer address can be sent to HLS IP. On the other hand, the AXI-Master protocol can handle the transfer of data before/after filtering.



### Verilog Code

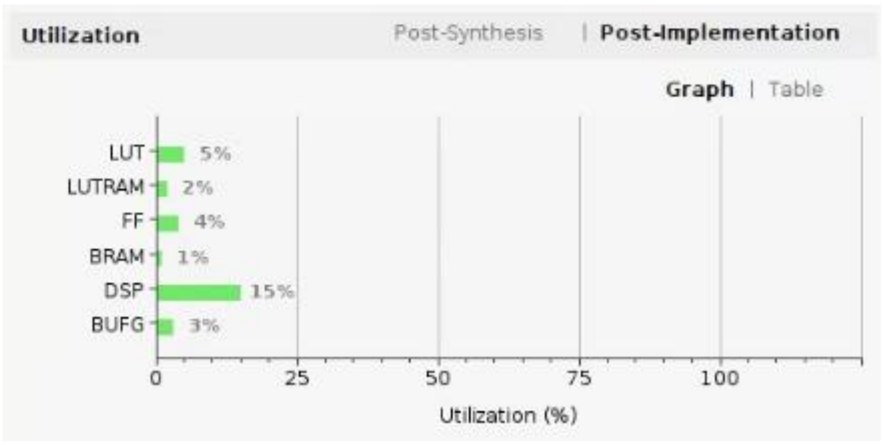
The generated Verilog code can be divided into 2 parts. The computation unit can be further divided into a pipelined loop (containing pipelined MAC structure) & multiplier unit. On the other hand, the controlling unit deals with the AXI-Lite & AXI-Master protocol respectively.


course-lab_2 / hls_ip / solution1 / impl / verilog

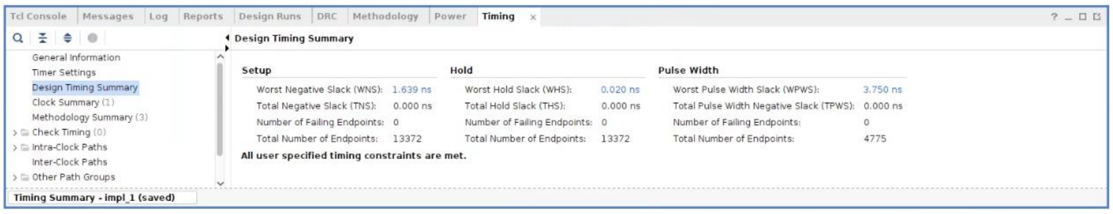
| Name | Size | Modified |
|---|---|---|
| fir_n11_maxi.v | 29.1 kB | 11:24 |
| fir_n11_maxi_control_s_axi.v | 19.6 kB | 11:24 |
| fir_n11_maxi_fir_n11_maxi_Pipeline_XFER_LOOP.v | 29.9 kB | 11:24 |
| fir_n11_maxi_gmem_m_axi.v | 104.9 kB | 11:24 |
| fir_n11_maxi_mul_32s_32s_32_2_1.v | 889 bytes | 11:24 |

## Hardware Utilization

From the aspect of DSP, the 11-tap multipliers require 33 DSP48E1. On the other hand, the HLS IP/AXI-Lite/AXI-Master requires ~3000/~400/~600 FFs respectively.



Under 100MHz clock signals, both setup/hold slack are met.

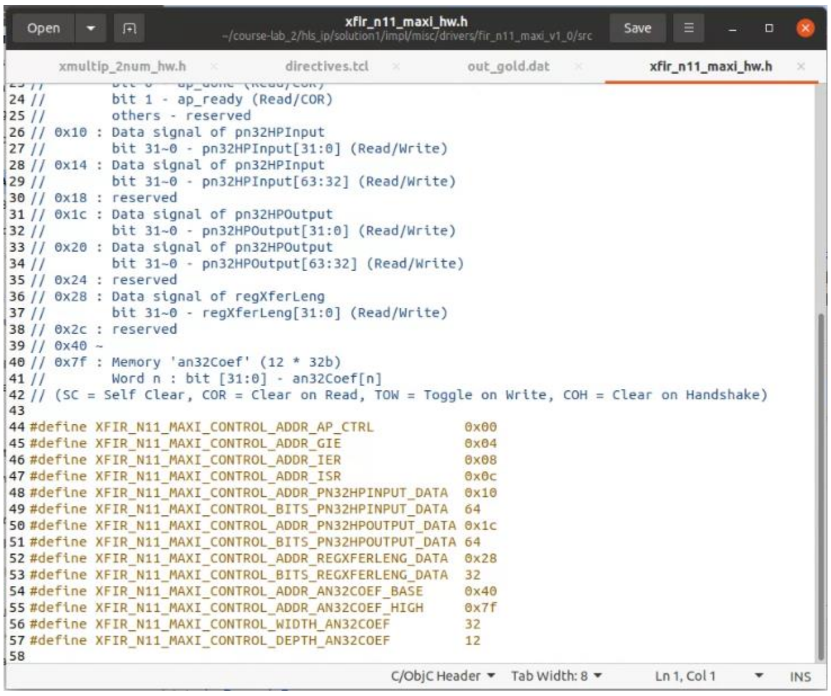


Design Timing Summary

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 1.639 ns | Worst Hold Slack (WHS): | 0.020 ns | Worst Pulse Width Slack (WPWS): | 3.750 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: | 13372 | Total Number of Endpoints: | 13372 | Total Number of Endpoints: | 4775 |

All user specified timing constraints are met.

In the driver files of MISC, we can find the address of different control signals which PYNQ APIs can further exploit.

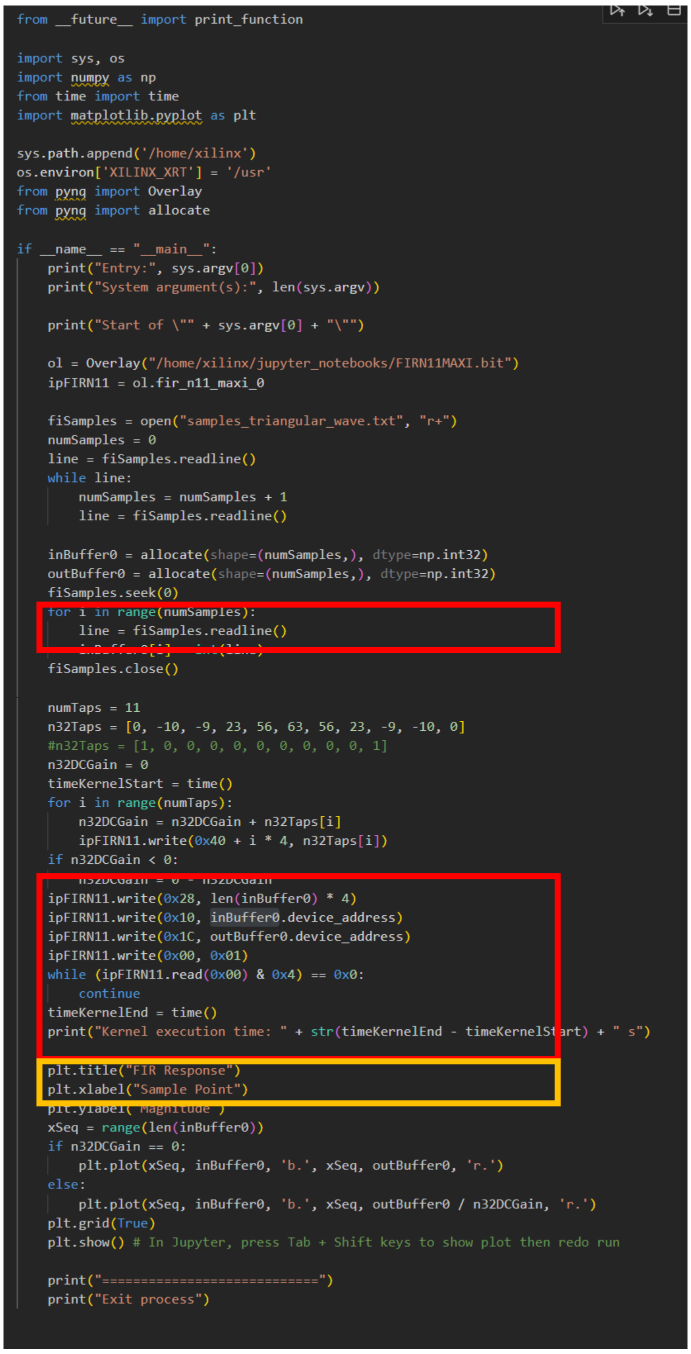


```
//        bit 0  - ap_done (Read/COR)
//        bit 1  - ap_ready (Read/COR)
//        others - reserved
// 0x10 : Data signal of pn32HPInput
//        bit 31~0 - pn32HPInput[31:0] (Read/Write)
// 0x14 : Data signal of pn32HPInput
//        bit 31~0 - pn32HPInput[63:32] (Read/Write)
// 0x18 : reserved
// 0x1c : Data signal of pn32HPOutput
//        bit 31~0 - pn32HPOutput[31:0] (Read/Write)
// 0x20 : Data signal of pn32HPOutput
//        bit 31~0 - pn32HPOutput[63:32] (Read/Write)
// 0x24 : reserved
// 0x28 : Data signal of regXferLeng
//        bit 31~0 - regXferLeng[31:0] (Read/Write)
// 0x2c : reserved
// 0x40 ~
// 0x7f : Memory 'an32Coef' (12 * 32b)
//        Word n : bit [31:0] - an32Coef[n]
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)

#define XFIR_N11_MAXI_CONTROL_ADDR_AP_CTRL            0x00
#define XFIR_N11_MAXI_CONTROL_ADDR_GIE                0x04
#define XFIR_N11_MAXI_CONTROL_ADDR_IER                0x08
#define XFIR_N11_MAXI_CONTROL_ADDR_ISR                0x0c
#define XFIR_N11_MAXI_CONTROL_ADDR_PN32HPINPUT_DATA   0x10
#define XFIR_N11_MAXI_CONTROL_BITS_PN32HPINPUT_DATA   64
#define XFIR_N11_MAXI_CONTROL_ADDR_PN32HPOUTPUT_DATA  0x1c
#define XFIR_N11_MAXI_CONTROL_BITS_PN32HPOUTPUT_DATA  64
#define XFIR_N11_MAXI_CONTROL_ADDR_REGXFERLENG_DATA   0x28
#define XFIR_N11_MAXI_CONTROL_BITS_REGXFERLENG_DATA   32
#define XFIR_N11_MAXI_CONTROL_ADDR_AN32COEF_BASE      0x40
#define XFIR_N11_MAXI_CONTROL_ADDR_AN32COEF_HIGH      0x7f
#define XFIR_N11_MAXI_CONTROL_WIDTH_AN32COEF          32
#define XFIR_N11_MAXI_CONTROL_DEPTH_AN32COEF          12

```

## Overview of the PS-side (PYNQ)

### Python code

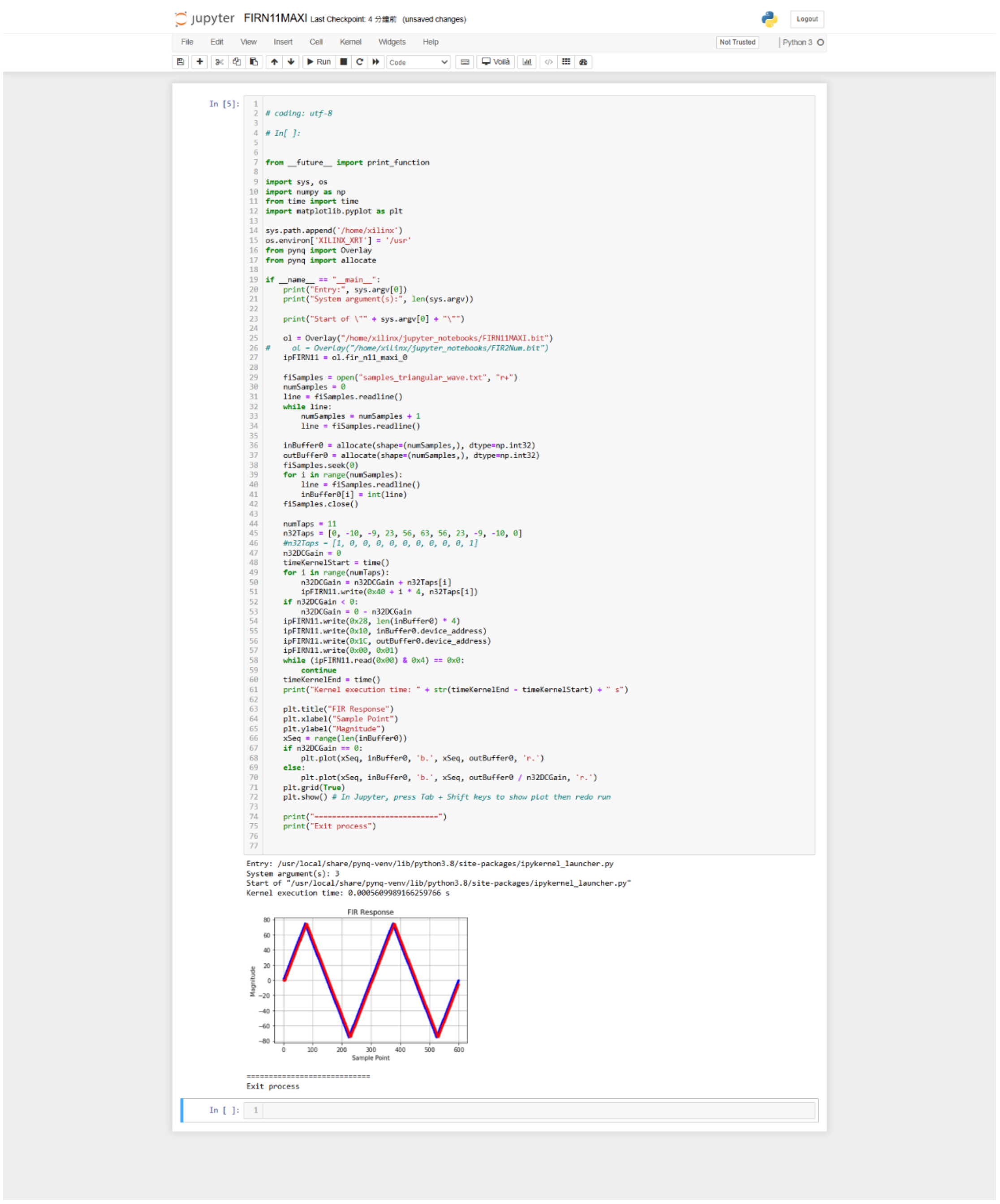
We can see the Python code simply loads the bitstream file, selects IP, set up configurations, and writes & reads until there's no output according to the address defined in the driver files.



```python
from __future__ import print_function

import sys, os
import numpy as np
from time import time
import matplotlib.pyplot as plt

sys.path.append('/home/xilinx')
os.environ['XILINX_XRT'] = '/usr'
from pynq import Overlay
from pynq import allocate

if __name__ == "__main__":
    print("Entry:", sys.argv[0])
    print("System argument(s):", len(sys.argv))

    print("Start of \"" + sys.argv[0] + "\"")

    ol = Overlay("/home/xilinx/jupyter_notebooks/FIRN11MAXI.bit")
    ipFIRN11 = ol.fir_n11_maxi_0

    fiSamples = open("samples_triangular_wave.txt", "r+")
    numSamples = 0
    line = fiSamples.readline()
    while line:
        numSamples = numSamples + 1
        line = fiSamples.readline()

    inBuffer0 = allocate(shape=(numSamples,), dtype=np.int32)
    outBuffer0 = allocate(shape=(numSamples,), dtype=np.int32)
    fiSamples.seek(0)
    for i in range(numSamples):
        line = fiSamples.readline()
        inBuffer0[i] = int(line)
    fiSamples.close()

    numTaps = 11
    n32Taps = [0, -10, -9, 23, 56, 63, 56, 23, -9, -10, 0]
    #n32Taps = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
    n32DCGain = 0
    timeKernelStart = time()
    for i in range(numTaps):
        n32DCGain = n32DCGain + n32Taps[i]
        ipFIRN11.write(0x40 + i * 4, n32Taps[i])
    if n32DCGain < 0:
        n32DCGain = 0 - n32DCGain
    ipFIRN11.write(0x28, len(inBuffer0) * 4)
    ipFIRN11.write(0x10, inBuffer0.device_address)
    ipFIRN11.write(0x1C, outBuffer0.device_address)
    ipFIRN11.write(0x00, 0x01)
    while (ipFIRN11.read(0x00) & 0x4) == 0x0:
        continue
    timeKernelEnd = time()
    print("Kernel execution time: " + str(timeKernelEnd - timeKernelStart) + " s")

    plt.title("FIR Response")
    plt.xlabel("Sample Point")
    plt.ylabel("Magnitude")
    xSeq = range(len(inBuffer0))
    if n32DCGain == 0:
        plt.plot(xSeq, inBuffer0, 'b.', xSeq, outBuffer0, 'r.')
    else:
        plt.plot(xSeq, inBuffer0, 'b.', xSeq, outBuffer0 / n32DCGain, 'r.')
    plt.grid(True)
    plt.show() # In Jupyter, press Tab + Shift keys to show plot then redo run

    print("============================")
    print("Exit process")
```

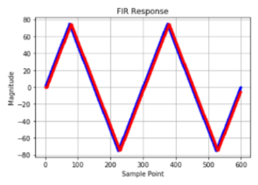
## Final results

As shown in the notebook, the computed results from HLS IP are correct & it requires 5.6e-4 seconds for 600 samples.



```

# coding: utf-8

# In[ ]:


from __future__ import print_function

import sys, os
import numpy as np
from time import time
import matplotlib.pyplot as plt

sys.path.append('/home/xilinx')
os.environ['XILINX_XRT'] = '/usr'
from pynq import Overlay
from pynq import allocate

if __name__ == "__main__":
    print("Entry:", sys.argv[0])
    print("System argument(s):", len(sys.argv))

    print("Start of \"" + sys.argv[0] + "\"")

    ol = Overlay("/home/xilinx/jupyter_notebooks/FIRN11MAXI.bit")
#     ol = Overlay("/home/xilinx/jupyter_notebooks/FIR2Num.bit")
    ipFIRN11 = ol.fir_n11_maxi_0

    fiSamples = open("samples_triangular_wave.txt", "r+")
    numSamples = 0
    line = fiSamples.readline()
    while line:
        numSamples = numSamples + 1
        line = fiSamples.readline()

    inBuffer0 = allocate(shape=(numSamples,), dtype=np.int32)
    outBuffer0 = allocate(shape=(numSamples,), dtype=np.int32)
    fiSamples.seek(0)
    for i in range(numSamples):
        line = fiSamples.readline()
        inBuffer0[i] = int(line)
    fiSamples.close()

    numTaps = 11
    n32Taps = [0, -10, -9, 23, 56, 63, 56, 23, -9, -10, 0]
    #n32Taps = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
    n32DCGain = 0
    timeKernelStart = time()
    for i in range(numTaps):
        n32DCGain = n32DCGain + n32Taps[i]
        ipFIRN11.write(0x40 + i * 4, n32Taps[i])
    if n32DCGain < 0:
        n32DCGain = 0 - n32DCGain
    ipFIRN11.write(0x28, len(inBuffer0) * 4)
    ipFIRN11.write(0x10, inBuffer0.device_address)
    ipFIRN11.write(0x1C, outBuffer0.device_address)
    ipFIRN11.write(0x00, 0x01)
    while (ipFIRN11.read(0x00) & 0x4) == 0x0:
        continue
    timeKernelEnd = time()
    print("Kernel execution time: " + str(timeKernelEnd - timeKernelStart) + " s")

    plt.title("FIR Response")
    plt.xlabel("Sample Point")
    plt.ylabel("Magnitude")
    xSeq = range(len(inBuffer0))
    if n32DCGain == 0:
        plt.plot(xSeq, inBuffer0, 'b.', xSeq, outBuffer0, 'r.')
    else:
        plt.plot(xSeq, inBuffer0, 'b.', xSeq, outBuffer0 / n32DCGain, 'r.')
    plt.grid(True)
    plt.show() # In Jupyter, press Tab + Shift keys to show plot then redo run

    print("----------------------------")
    print("Exit process")


```

```
Entry: /usr/local/share/pynq-venv/lib/python3.8/site-packages/ipykernel_launcher.py
System argument(s): 3
Start of "/usr/local/share/pynq-venv/lib/python3.8/site-packages/ipykernel_launcher.py"
Kernel execution time: 0.0005609989166259766 s
```



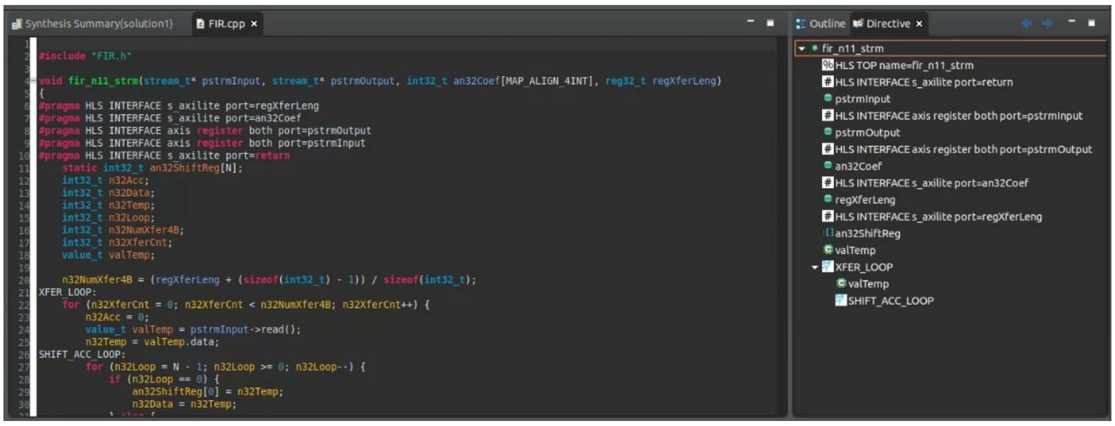
```
============================
Exit process
```

## Part 2 – AXI-Stream

## Overview of the HLS Design:

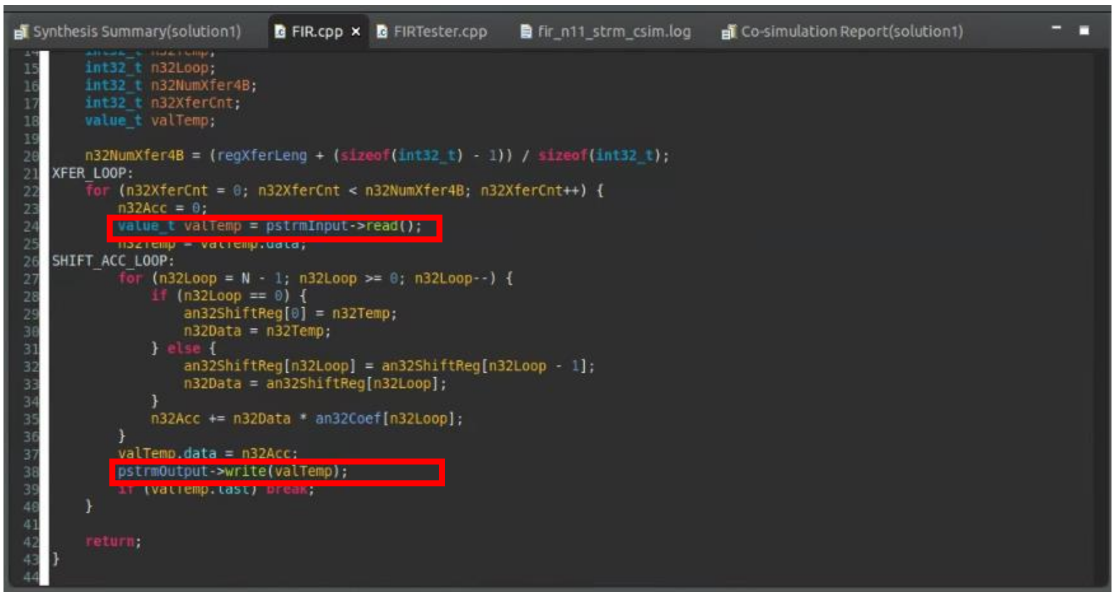
### The C code

Compared to the last examples, the code simply replaces the AXI-Master interface with AXI-Stream one. We also need to add read/write functions into .cpp file.



```
#include "FIR.h"

void fir_n11_strm(stream_t* pstrmInput, stream_t* pstrmOutput, int32_t an32Coef[MAP_ALIGN_4INT], reg32_t regXferLeng)
{
#pragma HLS INTERFACE s_axilite port=regXferLeng
#pragma HLS INTERFACE s_axilite port=an32Coef
#pragma HLS INTERFACE axis register both port=pstrmOutput
#pragma HLS INTERFACE axis register both port=pstrmInput
#pragma HLS INTERFACE s_axilite port=return
    static int32_t an32ShiftReg[N];
    int32_t n32Acc;
    int32_t n32Data;
    int32_t n32Temp;
    int32_t n32Loop;
    int32_t n32NumXfer4B;
    int32_t n32XferCnt;
    value_t valTemp;

    n32NumXfer4B = (regXferLeng + (sizeof(int32_t) - 1)) / sizeof(int32_t);
XFER_LOOP:
    for (n32XferCnt = 0; n32XferCnt < n32NumXfer4B; n32XferCnt++) {
        n32Acc = 0;
        value_t valTemp = pstrmInput->read();
        n32Temp = valTemp.data;
SHIFT_ACC_LOOP:
        for (n32Loop = N - 1; n32Loop >= 0; n32Loop--) {
            if (n32Loop == 0) {
                an32ShiftReg[0] = n32Temp;
                n32Data = n32Temp;
```

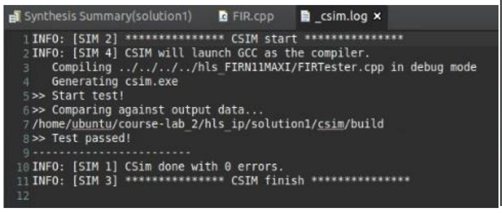
```
fir_n11_strm
    HLS TOP name=fir_n11_strm
    HLS INTERFACE s_axilite port=return
    pstrmInput
    HLS INTERFACE axis register both port=pstrmInput
    pstrmOutput
    HLS INTERFACE axis register both port=pstrmOutput
    an32Coef
    HLS INTERFACE s_axilite port=an32Coef
    regXferLeng
    HLS INTERFACE s_axilite port=regXferLeng
    an32ShiftReg
    valTemp
  XFER_LOOP
      valTemp
      SHIFT_ACC_LOOP
```



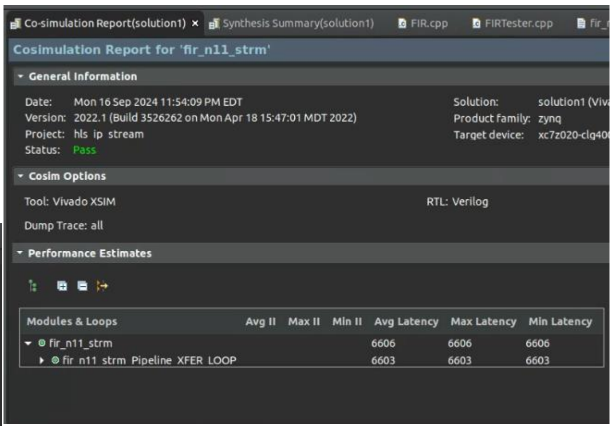
```
    int32_t n32Loop;
    int32_t n32NumXfer4B;
    int32_t n32XferCnt;
    value_t valTemp;

    n32NumXfer4B = (regXferLeng + (sizeof(int32_t) - 1)) / sizeof(int32_t);
XFER_LOOP:
    for (n32XferCnt = 0; n32XferCnt < n32NumXfer4B; n32XferCnt++) {
        n32Acc = 0;
        value_t valTemp = pstrmInput->read();
        n32Temp = valTemp.data;
SHIFT_ACC_LOOP:
        for (n32Loop = N - 1; n32Loop >= 0; n32Loop--) {
            if (n32Loop == 0) {
                an32ShiftReg[0] = n32Temp;
                n32Data = n32Temp;
            } else {
                an32ShiftReg[n32Loop] = an32ShiftReg[n32Loop - 1];
                n32Data = an32ShiftReg[n32Loop];
            }
            n32Acc += n32Data * an32Coef[n32Loop];
        }
        valTemp.data = n32Acc;
        pstrmOutput->write(valTemp);
        if (valTemp.last) break;
    }

    return;
}
```

For the tester file, we need to add control of last signals & manually pull up

### C-simulation (left) & Co-simulation (right)



```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../../hls_FIRN11MAXI/FIRTester.cpp in debug mode
   Generating csim.exe
>> Start test!
>> Comparing against output data...
/home/ubuntu/course-lab_2/hls_ip/solution1/csim/build
>> Test passed!
------------------------
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```



Cosimulation Report for 'fir_n11_strm'

General Information

Date: Mon 16 Sep 2024 11:54:09 PM EDT
Version: 2022.1 (Build 3526262 on Mon Apr 18 15:47:01 MDT 2022)
Project: hls_ip_stream
Status: Pass
Solution: solution1 (Viva
Product family: zynq
Target device: xc7z020-clg40

Cosim Options

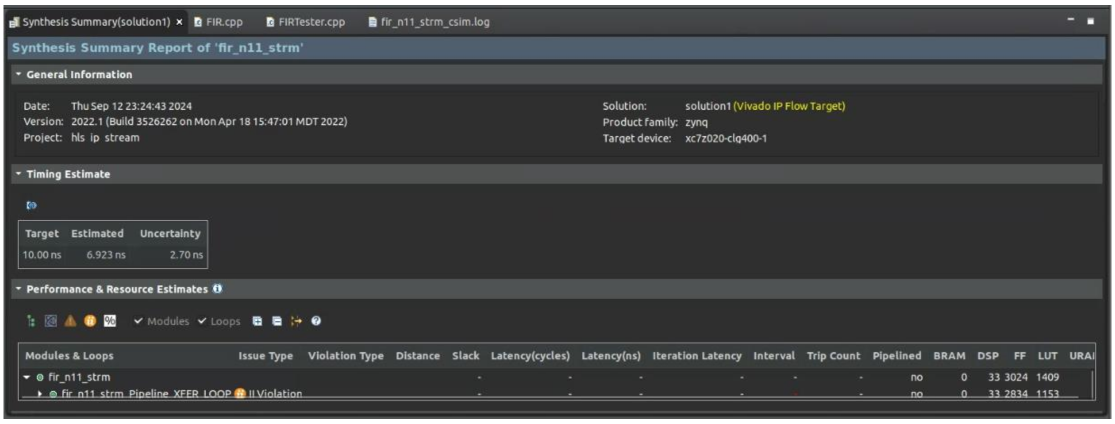
Tool: Vivado XSIM
RTL: Verilog
Dump Trace: all

Performance Estimates

| Modules & Loops | Avg II | Max II | Min II | Avg Latency | Max Latency | Min Latency |
|---|---|---|---|---|---|---|
| fir_n11_strm | | | | 6606 | 6606 | 6606 |
| fir_n11_strm_Pipeline_XFER_LOOP | | | | 6603 | 6603 | 6603 |

## C-synthesis

The design is synthesized with clock period=10ns



Synthesis Summary Report of 'fir_n11_strm'

General Information

Date: Thu Sep 12 23:24:43 2024
Version: 2022.1 (Build 3526262 on Mon Apr 18 15:47:01 MDT 2022)
Project: hls_ip_stream
Solution: solution1 (Vivado IP Flow Target)
Product family: zynq
Target device: xc7z020-clg400-1

Timing Estimate

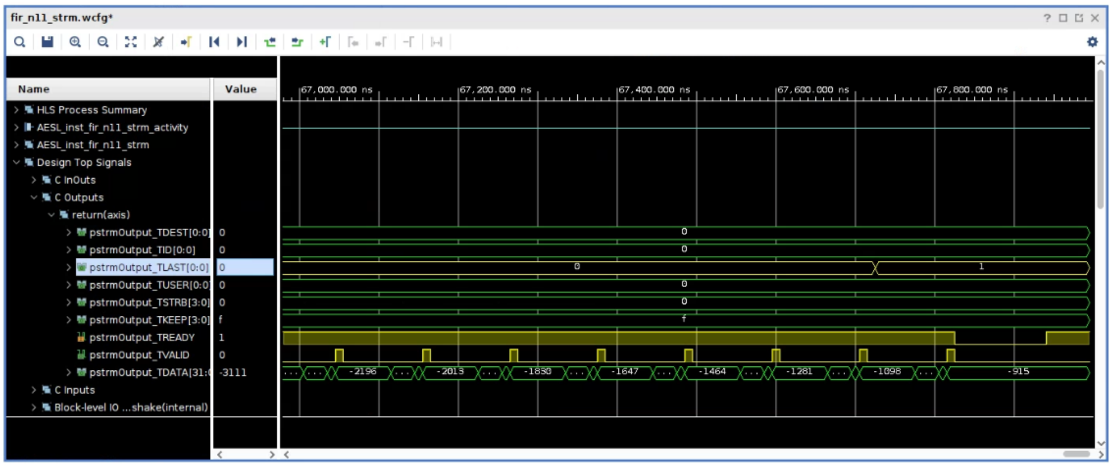
| Target | Estimated | Uncertainty |
|---|---|---|
| 10.00 ns | 6.923 ns | 2.70 ns |

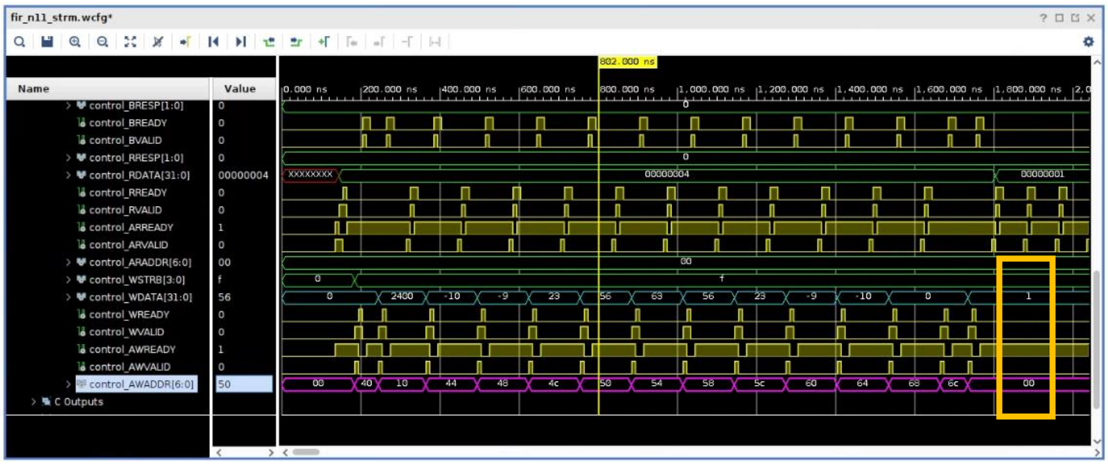
Performance & Resource Estimates

| Modules & Loops | Issue Type | Violation Type | Distance | Slack | Latency(cycles) | Latency(ns) | Iteration Latency | Interval | Trip Count | Pipelined | BRAM | DSP | FF | LUT | URAI |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| fir_n11_strm | | | | - | - | - | - | - | - | no | 0 | 33 | 3024 | 1409 | |
| fir_n11_strm_Pipeline_XFER_LOOP | II Violation | | | - | - | - | - | - | - | no | 0 | 33 | 2834 | 1153 | |

## Generated Waveform

We can see that the output is terminated when the TLAST signals is pulled up.



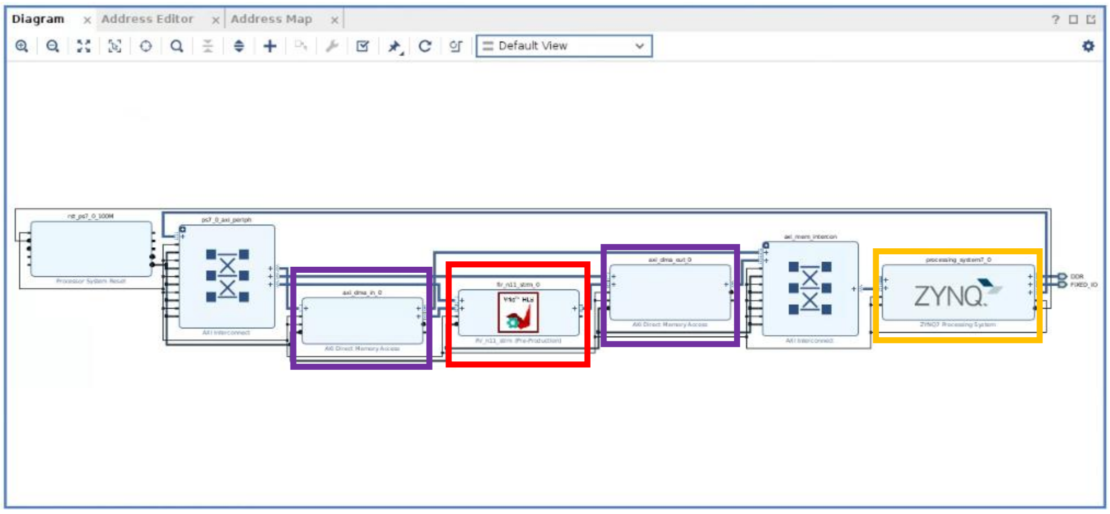


| Name | Value |
|---|---|
| control_BRESP[1:0] | 0 |
| control_BREADY | 0 |
| control_BVALID | 0 |
| control_RRESP[1:0] | 0 |
| control_RDATA[31:0] | 00000004 |
| control_RREADY | 0 |
| control_RVALID | 0 |
| control_ARREADY | 1 |
| control_ARVALID | 0 |
| control_ARADDR[6:0] | 00 |
| control_WSTRB[3:0] | f |
| control_WDATA[31:0] | 56 |
| control_WREADY | 0 |
| control_WVALID | 0 |
| control_AWREADY | 1 |
| control_AWVALID | 0 |
| control_AWADDR[6:0] | 50 |
| C Outputs | |

## Overview of the Hardware Wrapper:

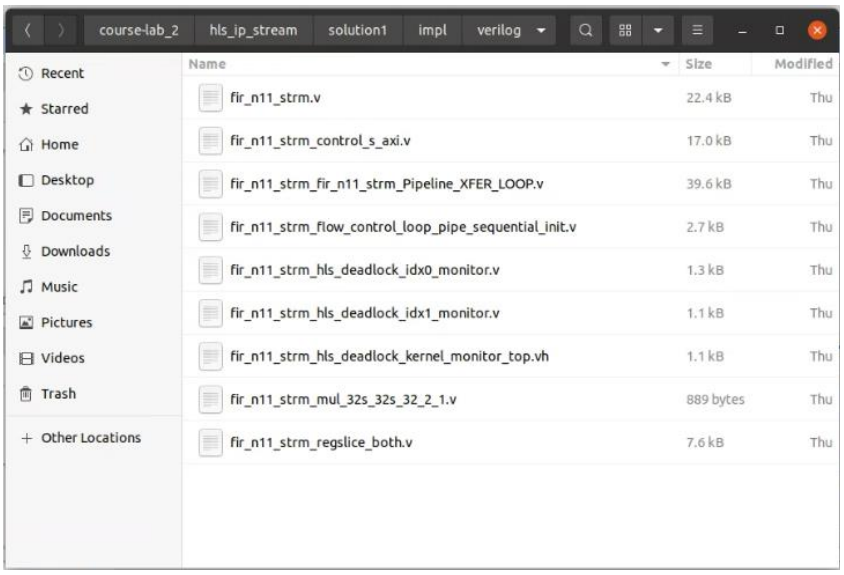
### Block Diagram

In the block diagram, unlike the last example, we need to manually add DMA IP for both input/output of HLS IP. The DMA IP facilitates high-bandwidth data transfers without involving the CPU, while AXI peripherals typically only handle memory-mapped transactions.



### Verilog Code

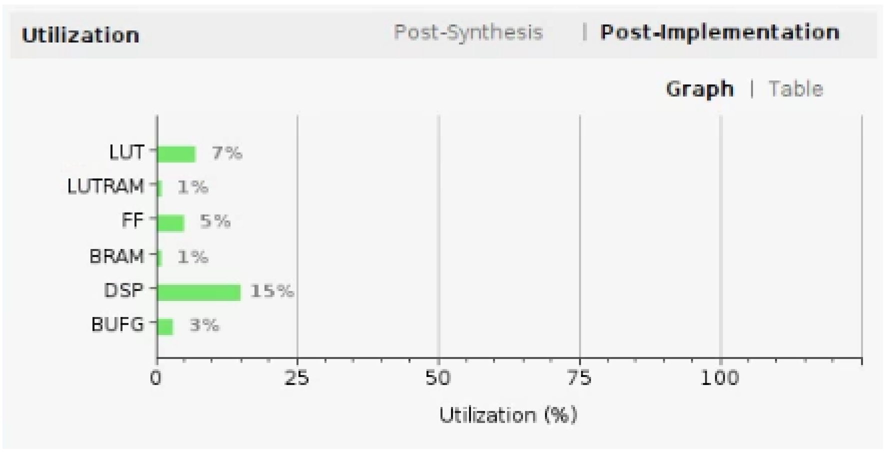
The generated Verilog code can be divided into 2 parts. The computation unit can be further divided into a pipelined loop (containing pipelined MAC structure) & multiplier unit. On the other hand, the controlling unit deals with the AXI-Lite & AXI-Stream protocols respectively.



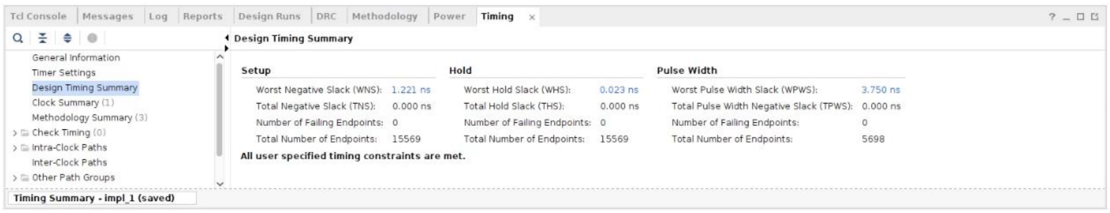
| Name | Size | Modified |
|---|---|---|
| fir_n11_strm.v | 22.4 kB | Thu |
| fir_n11_strm_control_s_axi.v | 17.0 kB | Thu |
| fir_n11_strm_fir_n11_strm_Pipeline_XFER_LOOP.v | 39.6 kB | Thu |
| fir_n11_strm_flow_control_loop_pipe_sequential_init.v | 2.7 kB | Thu |
| fir_n11_strm_hls_deadlock_idx0_monitor.v | 1.3 kB | Thu |
| fir_n11_strm_hls_deadlock_idx1_monitor.v | 1.1 kB | Thu |
| fir_n11_strm_hls_deadlock_kernel_monitor_top.vh | 1.1 kB | Thu |
| fir_n11_strm_mul_32s_32s_32_2_1.v | 889 bytes | Thu |
| fir_n11_strm_regslice_both.v | 7.6 kB | Thu |

## Hardware Utilization

From the aspect of DSP, the 11-tap multipliers also require 33 DSP48E1. On the other hand, the HLS IP/AXI-Lite/AXI-Master requires ~1400/~600/~1100 FFs respectively. The DMA in/out IP also introduces ~750/~1400 additional FFs.



Under 100MHz clock signals, both setup/hold slack are met.

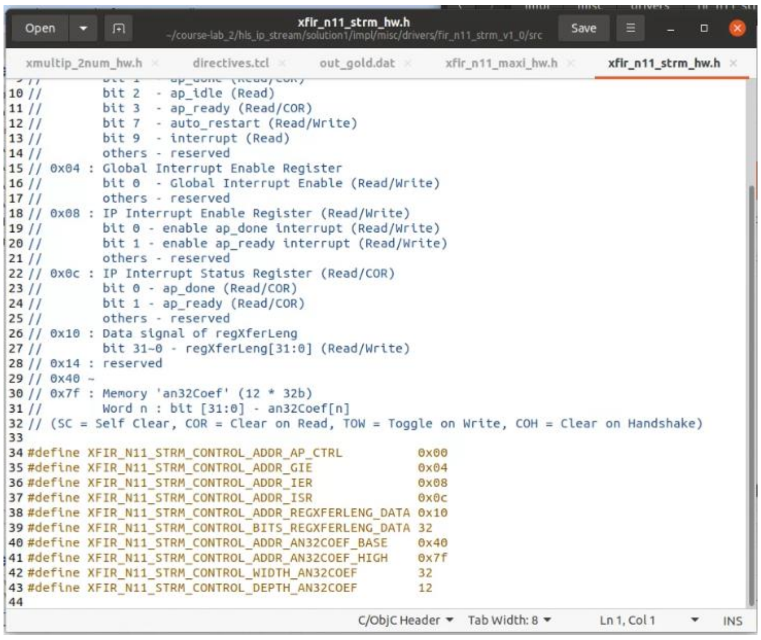


Design Timing Summary

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 1.221 ns | Worst Hold Slack (WHS): | 0.023 ns | Worst Pulse Width Slack (WPWS): | 3.750 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: | 15569 | Total Number of Endpoints: | 15569 | Total Number of Endpoints: | 5698 |

All user specified timing constraints are met.

In the driver files of MISC, we can find the address of different control signals which PYNQ APIs can further exploit.

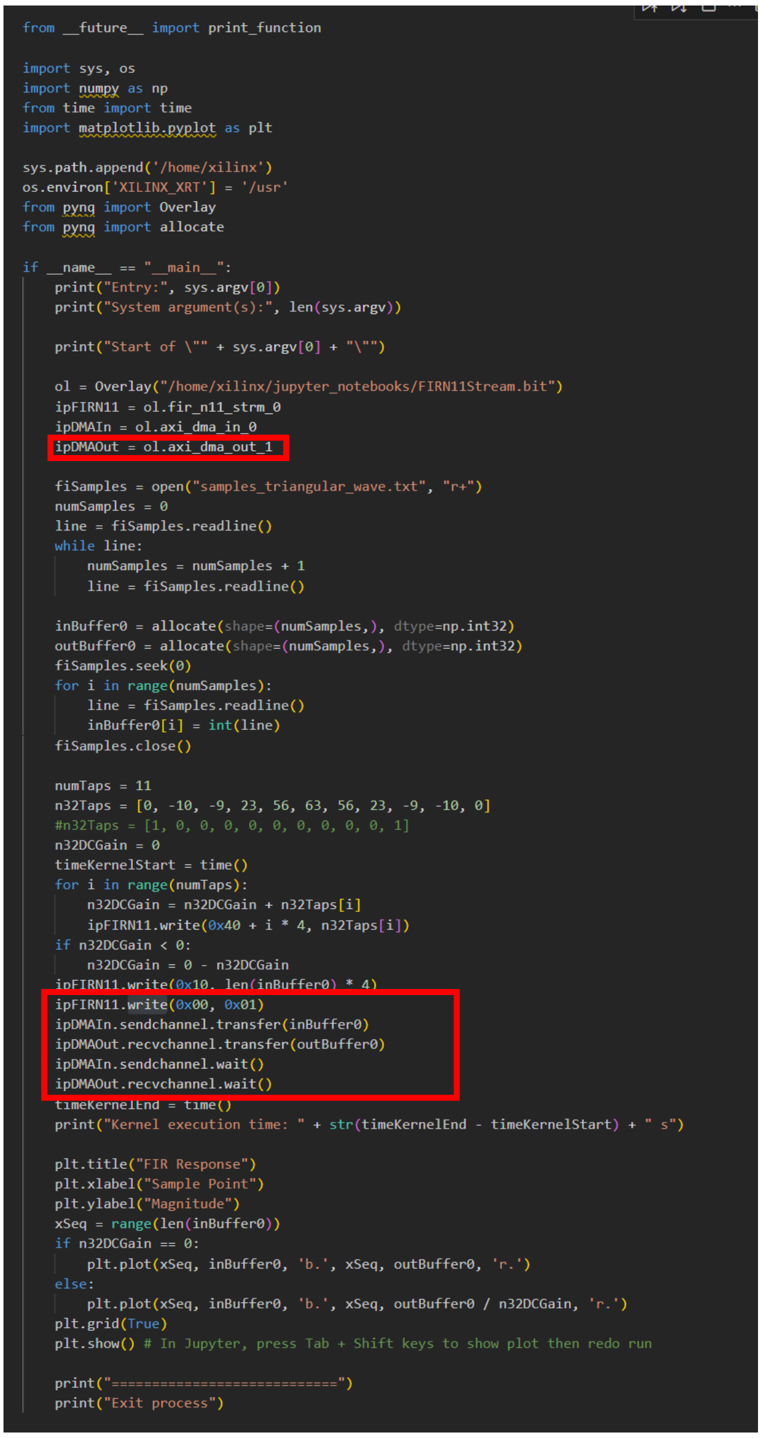


```
//         bit 2  - ap_idle (Read)
//         bit 3  - ap_ready (Read/COR)
//         bit 7  - auto_restart (Read/Write)
//         bit 9  - interrupt (Read)
//         others - reserved
// 0x04 : Global Interrupt Enable Register
//         bit 0  - Global Interrupt Enable (Read/Write)
//         others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//         bit 0 - enable ap_done interrupt (Read/Write)
//         bit 1 - enable ap_ready interrupt (Read/Write)
//         others - reserved
// 0x0c : IP Interrupt Status Register (Read/COR)
//         bit 0 - ap_done (Read/COR)
//         bit 1 - ap_ready (Read/COR)
//         others - reserved
// 0x10 : Data signal of regXferLeng
//         bit 31~0 - regXferLeng[31:0] (Read/Write)
// 0x14 : reserved
// 0x40 ~
// 0x7f : Memory 'an32Coef' (12 * 32b)
//         Word n : bit [31:0] - an32Coef[n]
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)

#define XFIR_N11_STRM_CONTROL_ADDR_AP_CTRL            0x00
#define XFIR_N11_STRM_CONTROL_ADDR_GIE                0x04
#define XFIR_N11_STRM_CONTROL_ADDR_IER                0x08
#define XFIR_N11_STRM_CONTROL_ADDR_ISR                0x0c
#define XFIR_N11_STRM_CONTROL_ADDR_REGXFERLENG_DATA 0x10
#define XFIR_N11_STRM_CONTROL_BITS_REGXFERLENG_DATA 32
#define XFIR_N11_STRM_CONTROL_ADDR_AN32COEF_BASE    0x40
#define XFIR_N11_STRM_CONTROL_ADDR_AN32COEF_HIGH    0x7f
#define XFIR_N11_STRM_CONTROL_WIDTH_AN32COEF        32
#define XFIR_N11_STRM_CONTROL_DEPTH_AN32COEF        12

```

## Overview of the PS-side (PYNQ)

### Python code

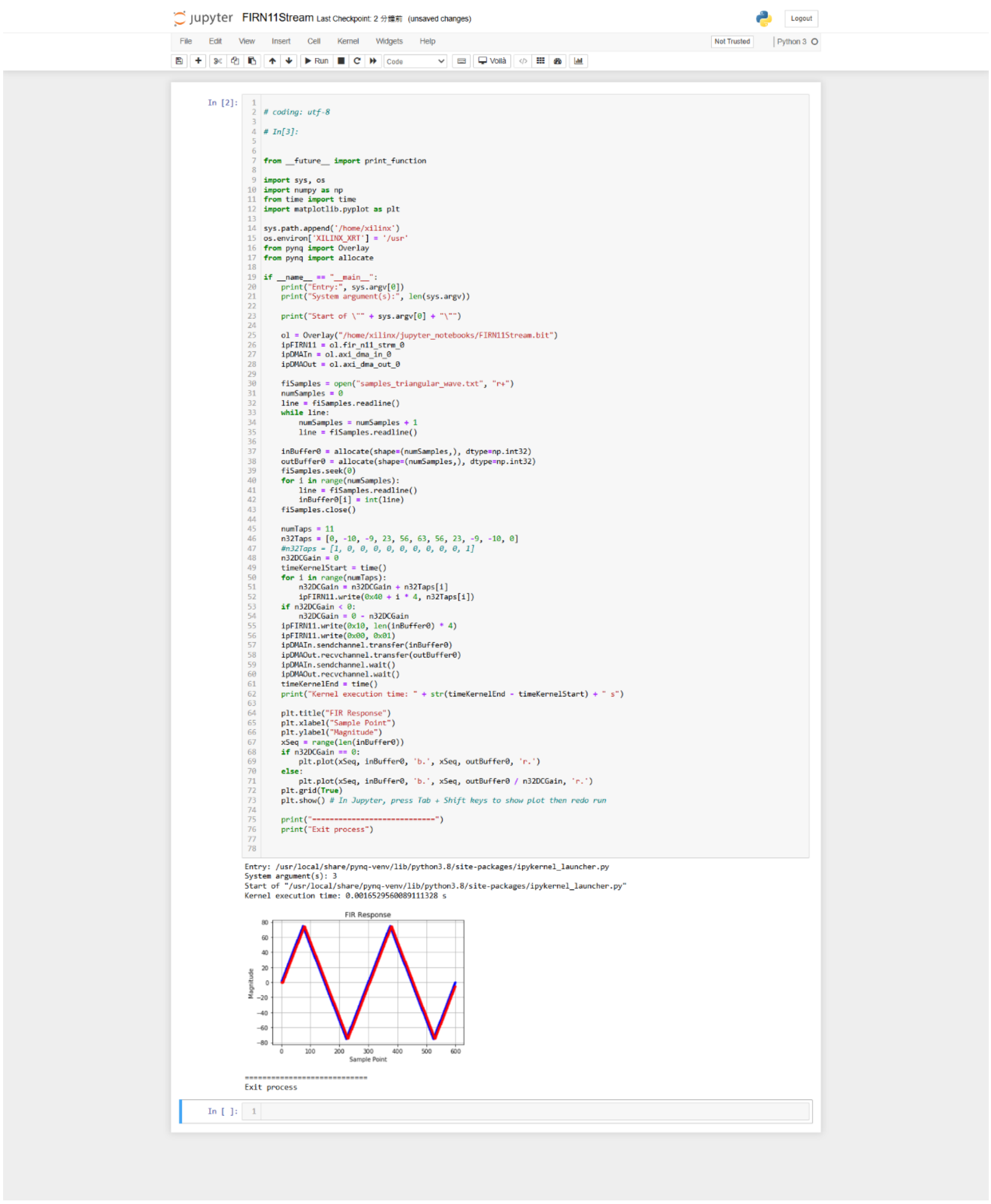
We can see the Python code simply loads the bitstream file, selects IP, and writes to & reads from the address defined in the driver files. In addition, the ipDMAOut needs to be manually modified to “ol.axi_dma_out_0”. We can also see that the order of transfer & start signals is different between AXI-Master & AXI-Stream.



```python
from __future__ import print_function

import sys, os
import numpy as np
from time import time
import matplotlib.pyplot as plt

sys.path.append('/home/xilinx')
os.environ['XILINX_XRT'] = '/usr'
from pynq import Overlay
from pynq import allocate

if __name__ == "__main__":
    print("Entry:", sys.argv[0])
    print("System argument(s):", len(sys.argv))

    print("Start of \"" + sys.argv[0] + "\"")

    ol = Overlay("/home/xilinx/jupyter_notebooks/FIRN11Stream.bit")
    ipFIRN11 = ol.fir_n11_strm_0
    ipDMAIn = ol.axi_dma_in_0
    ipDMAOut = ol.axi_dma_out_1

    fiSamples = open("samples_triangular_wave.txt", "r+")
    numSamples = 0
    line = fiSamples.readline()
    while line:
        numSamples = numSamples + 1
        line = fiSamples.readline()

    inBuffer0 = allocate(shape=(numSamples,), dtype=np.int32)
    outBuffer0 = allocate(shape=(numSamples,), dtype=np.int32)
    fiSamples.seek(0)
    for i in range(numSamples):
        line = fiSamples.readline()
        inBuffer0[i] = int(line)
    fiSamples.close()

    numTaps = 11
    n32Taps = [0, -10, -9, 23, 56, 63, 56, 23, -9, -10, 0]
    #n32Taps = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
    n32DCGain = 0
    timeKernelStart = time()
    for i in range(numTaps):
        n32DCGain = n32DCGain + n32Taps[i]
        ipFIRN11.write(0x40 + i * 4, n32Taps[i])
    if n32DCGain < 0:
        n32DCGain = 0 - n32DCGain
    ipFIRN11.write(0x10, len(inBuffer0) * 4)
    ipFIRN11.write(0x00, 0x01)
    ipDMAIn.sendchannel.transfer(inBuffer0)
    ipDMAOut.recvchannel.transfer(outBuffer0)
    ipDMAIn.sendchannel.wait()
    ipDMAOut.recvchannel.wait()
    timeKernelEnd = time()
    print("Kernel execution time: " + str(timeKernelEnd - timeKernelStart) + " s")

    plt.title("FIR Response")
    plt.xlabel("Sample Point")
    plt.ylabel("Magnitude")
    xSeq = range(len(inBuffer0))
    if n32DCGain == 0:
        plt.plot(xSeq, inBuffer0, 'b.', xSeq, outBuffer0, 'r.')
    else:
        plt.plot(xSeq, inBuffer0, 'b.', xSeq, outBuffer0 / n32DCGain, 'r.')
    plt.grid(True)
    plt.show() # In Jupyter, press Tab + Shift keys to show plot then redo run

    print("============================")
    print("Exit process")
```

## Final results

As shown in the notebook, the computed results from HLS IP are correct & it requires 1.6e-3 seconds for 600 samples.


jupyter FIRN11Stream Last Checkpoint: 2 分鐘前 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
Python 3
Run
Code
Voilà
In [2]:
FIR Response
Sample Point
Magnitude
In [ ]:

```python

# coding: utf-8

# In[3]:


from __future__ import print_function

import sys, os
import numpy as np
from time import time
import matplotlib.pyplot as plt

sys.path.append('/home/xilinx')
os.environ['XILINX_XRT'] = '/usr'
from pynq import Overlay
from pynq import allocate

if __name__ == "__main__":
    print("Entry:", sys.argv[0])
    print("System argument(s):", len(sys.argv))

    print("Start of \"" + sys.argv[0] + "\"")

    ol = Overlay("/home/xilinx/jupyter_notebooks/FIRN11Stream.bit")
    ipFIRN11 = ol.fir_n11_strm_0
    ipDMAIn = ol.axi_dma_in_0
    ipDMAOut = ol.axi_dma_out_0

    fiSamples = open("samples_triangular_wave.txt", "r+")
    numSamples = 0
    line = fiSamples.readline()
    while line:
        numSamples = numSamples + 1
        line = fiSamples.readline()

    inBuffer0 = allocate(shape=(numSamples,), dtype=np.int32)
    outBuffer0 = allocate(shape=(numSamples,), dtype=np.int32)
    fiSamples.seek(0)
    for i in range(numSamples):
        line = fiSamples.readline()
        inBuffer0[i] = int(line)
    fiSamples.close()

    numTaps = 11
    n32Taps = [0, -10, -9, 23, 56, 63, 56, 23, -9, -10, 0]
    #n32Taps = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
    n32DCGain = 0
    timeKernelStart = time()
    for i in range(numTaps):
        n32DCGain = n32DCGain + n32Taps[i]
        ipFIRN11.write(0x40 + i * 4, n32Taps[i])
    if n32DCGain < 0:
        n32DCGain = 0 - n32DCGain
    ipFIRN11.write(0x10, len(inBuffer0) * 4)
    ipFIRN11.write(0x00, 0x01)
    ipDMAIn.sendchannel.transfer(inBuffer0)
    ipDMAOut.recvchannel.transfer(outBuffer0)
    ipDMAIn.sendchannel.wait()
    ipDMAOut.recvchannel.wait()
    timeKernelEnd = time()
    print("Kernel execution time: " + str(timeKernelEnd - timeKernelStart) + " s")

    plt.title("FIR Response")
    plt.xlabel("Sample Point")
    plt.ylabel("Magnitude")
    xSeq = range(len(inBuffer0))
    if n32DCGain == 0:
        plt.plot(xSeq, inBuffer0, 'b.', xSeq, outBuffer0, 'r.')
    else:
        plt.plot(xSeq, inBuffer0, 'b.', xSeq, outBuffer0 / n32DCGain, 'r.')
    plt.grid(True)
    plt.show() # In Jupyter, press Tab + Shift keys to show plot then redo run

    print("============================")
    print("Exit process")


```

```
Entry: /usr/local/share/pynq-venv/lib/python3.8/site-packages/ipykernel_launcher.py
System argument(s): 3
Start of "/usr/local/share/pynq-venv/lib/python3.8/site-packages/ipykernel_launcher.py"
Kernel execution time: 0.0016529560089111328 s
```

```
============================
Exit process
```

## Summary

The 2 examples of lab 2 simply walk us through 2 different AXI protocols: AXI-Master & AXI-Stream. We can also discover some differences between the behavior & performance of them.

First, we can discover that the simulation time of AXI-Stream is much longer than that of AXI-Master from the co-simulation result. It might be caused by the additional control signals in the input for-loop of the .cpp tester.

Second, we can discover that the simulation time of AXI-Stream is also still longer than that of AXI-Master from the PYNQ on-board test. Although I cannot fully investigate the black-box behavior inside both PYNQ API & the board, when using DMA with AXI-Stream interfaces, there may be additional software overhead involved in managing the data flow compared to using an AXI-Master interface directly with memory-mapped operations. This overhead might further slow down performance when the amount of data is low.

In summary, while AXI-Stream is suitable for specific applications like video or audio streaming where continuous data flow is essential, it may not match the efficiency of AXI-Master interfaces in scenarios requiring high throughput and low latency due to its inherent design limitations and the complexities involved in managing backpressure and flow control.