tags: **MSOC**

MSOC Lab2 Report

Lab2-1

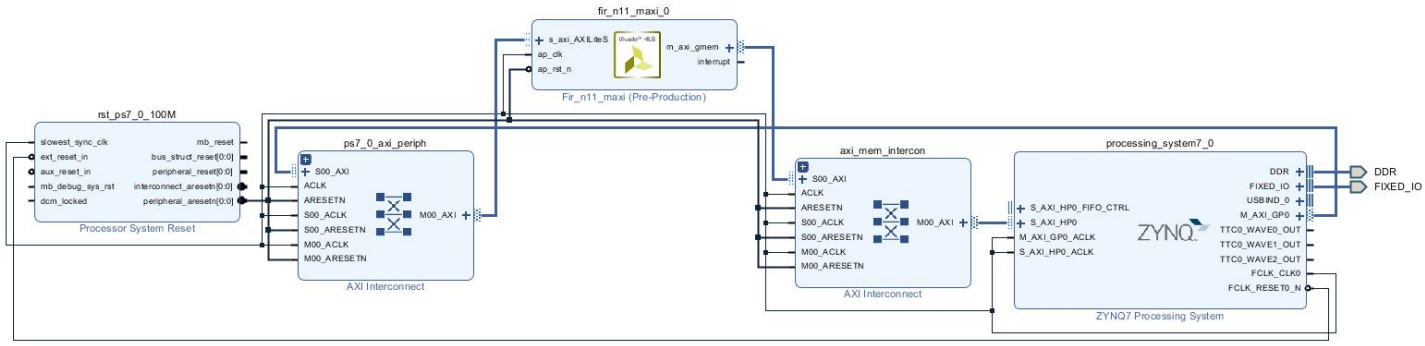
Objective

After completing this lab, we will be able to learn:

- Use Vivado-HLS API to generate a 11-tap FIR filter RTL IP with AXI-Master interface from System-C code.
- Build a software project and verify the FIR design functionality in hardware.
- [Additional] Study and implement the pipeline directive on Vivado-HLS.
- [Additional] Run Vivado in command mode and implement different optimization commands to improve the orignal design.

System diagram

This lab intends to develop a **11-tap FIR filter** through the AXIMaster interface.



Interface

This design utilize the **AXI-Master interface** to transfer the data and **AXILiteS interface** to transfer parameters. The following table shows the usage for some address.

Addr. (+Base addr.)	bit	Usage
0x00	0	ap_start
0x00	1	ap_done
0x00	2	ap_idle
0x00	3	ap_ready
0x00	7	auto_restart
0x04	0	Global interrupt
0x10		Input data
0x18		Output data
0x80		transfer length
0x40~0x7f		Coefficient

C code

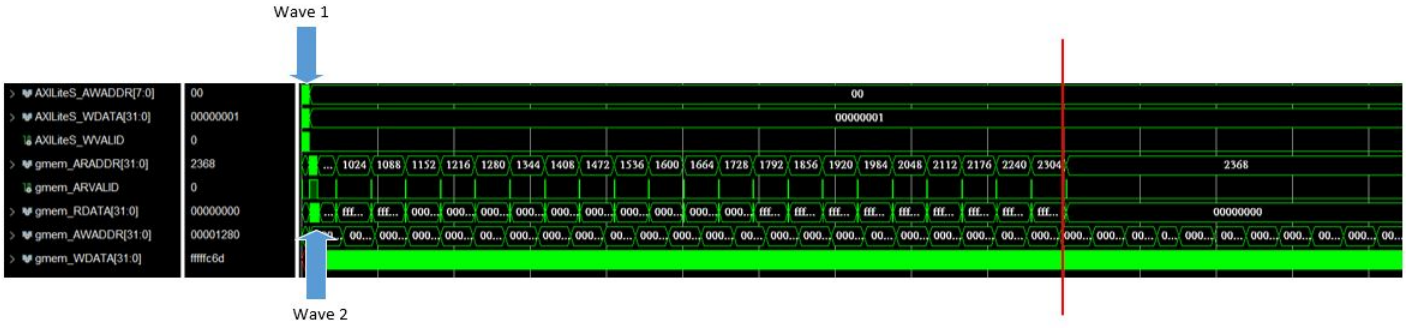
FIR.cpp:

```
1 #include "fir.h"
2
3 void fir_n11_maxi(volatile int32_t* pn32HPInput
4     , volatile  int32_t* pn32HPOutput, int32_t an32Coef[MAP_ALIGN_4INT]
5     , reg32_t regXferLeng){
6       static int32_t an32ShiftReg[N];
7       int32_t n32Acc;
8       int32_t n32Data;
9       int32_t n32Temp;
10      int32_t n32Loop;
11      int32_t n32NumXfer4B;
12      int32_t n32XferCnt;
13
14      n32NumXfer4B = (regXferLeng + (sizeof(int32_t) - 1)) / sizeof(int32_t);
15      XFER_LOOP:
16      for (n32XferCnt = 0; n32XferCnt < n32NumXfer4B; n32XferCnt++) {
17              n32Acc = 0;
18              n32Temp = pn32HPInput[n32XferCnt];
19              SHIFT_ACC_LOOP:
20              for (n32Loop = N - 1; n32Loop >= 0; n32Loop--) {
21                      if (n32Loop == 0) {
22                              an32ShiftReg[0] = n32Temp;
23                              n32Data = n32Temp;
24                      } else {
25                              an32ShiftReg[n32Loop] = an32ShiftReg[n32Loop-1];
26                              n32Data = an32ShiftReg[n32Loop];
27                      }
28                      n32Acc += n32Data * an32Coef[n32Loop];
29              }
30              pn32HPOutput[n32XferCnt] = n32Acc;
31      }
32      return;
33 }
```

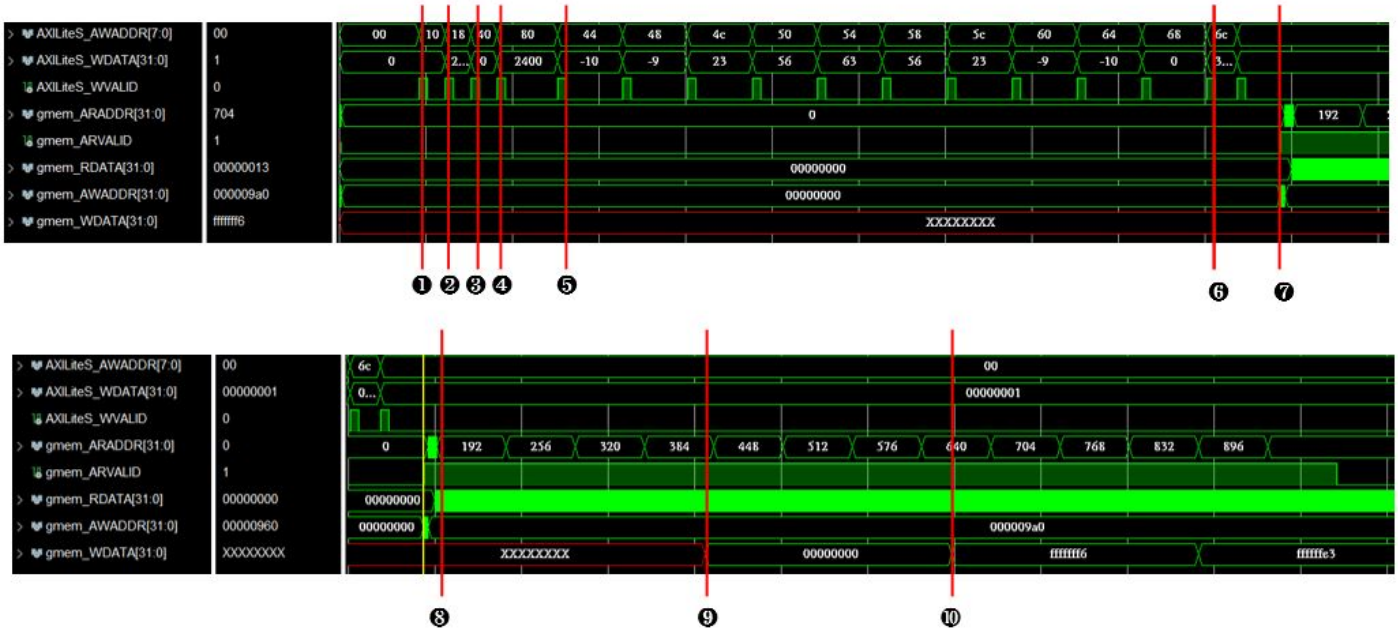
- Line 16~31: Traverse through all the input datas and calculate the FIR output datas.
- Line 20~29: Right shift the data in array “an32ShiftReg” and put “n32Temp” to the first place in the array. Also calculate the output data by accumulating the partial
- Line 30: Assign the output data to the output reg.

Cosim waveform

The whole waveform is shown in the figure below.



We expand the “Wave 1” and “Wave 2” part in order to make clear analysis on iit.



❶ AWADDR = 10, WDATA = 0: Input data array address start from 0 (AXILiteS)
❷ AWADDR = 18, WDATA = 2400_{dec}: Output data array address start from 2400_{dec} (AXILiteS)
❸ AWADDR = 40, WDATA = 0_{dec}: The first coef. of the filter is 0_{dec} (AXILiteS)
❹ AWADDR = 80, WDATA = 2400_{dec}: The size of the buffer is 2400_{dec}, which is 4 (int32 bytes) times 600 (number of samples). (AXILiteS)
❺ AWADDR = 40+4, WDATA = 0-10_{dec}: The second coef. of the filter is 0-10_{dec} (AXILiteS)
❺~❻ Input the remaining coef. of the 10 taps (AXILiteS)
❻ Get ‘end’ signal of the coef. array. (AXILiteS)
❼ Start Kernel functions.
❽~❾ The data is input (through RDATA) into the kernel function and the corresponding address (ARADDR) increase (AXI-Master).
❾ The first data output from the kernel (through WDATA) with corresponding physical address (AWADDR) (AXI-Master).
❿ The second data output from the kernel (through WDATA) with corresponding physical address (AWADDR) (AXI-Master).

We can see that ARADDR start from 0, 64, 128, … (in decimal) and AWADDR start from 2400, 2464, 2528, … (in decimal). That is because we initialize **the physical address of input and output data to 0 and 2400 through the AXILiteS**.

Performance and Analysis

We can see that it **consumes 55 cycles** for calculation and the **estimated clock rate is 8.75ns** from the report below:

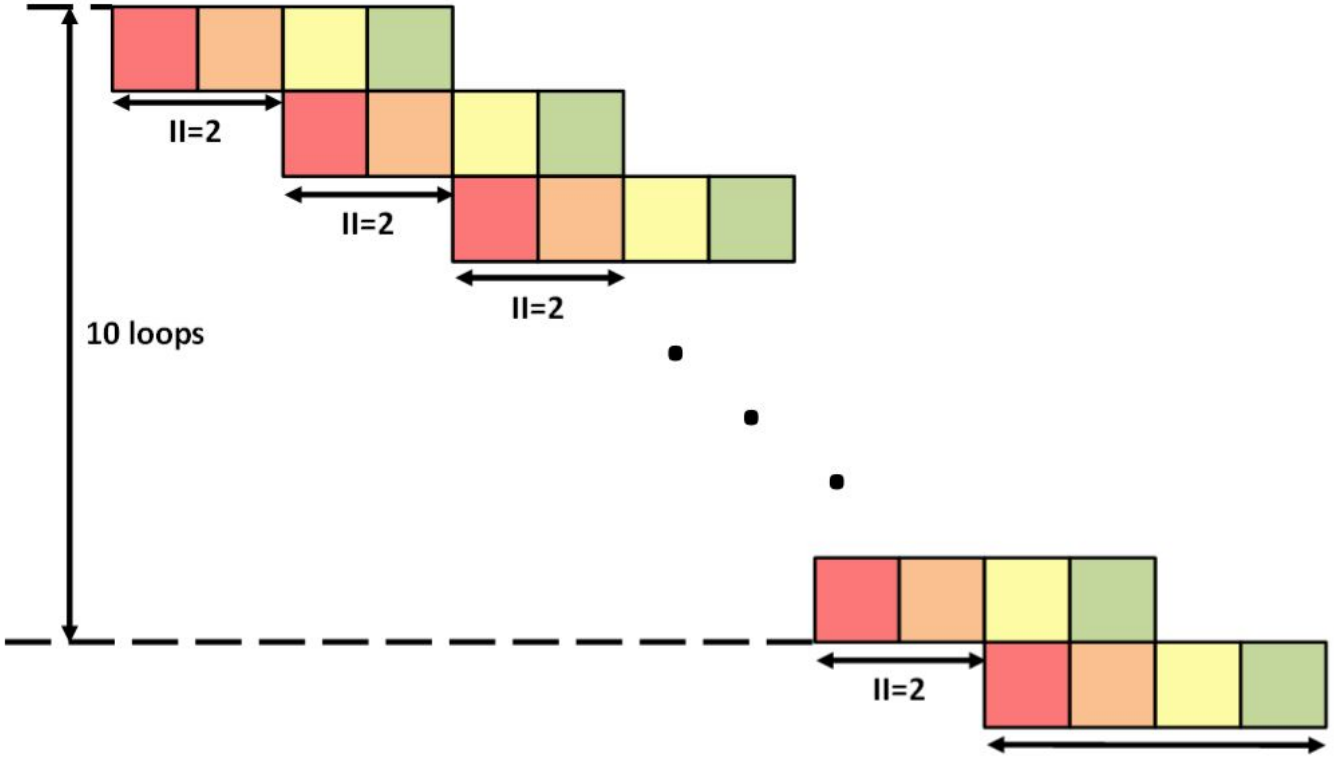
Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	8.750 ns	1.25 ns

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- XFER_LOOP	?	?	57	-	-	?	no
+ SHIFT_ACC_LOOP	55	55	5	-	-	11	no

We can **add pipeline directives** on the Loop.
We set initiation intervals (II) to 2 in this case, then the cycle can reduce from 55 to 23 (The latency for each loop is 4 after pipeline).

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- XFER_LOOP	?	?	26	-	-	?	no
+ SHIFT_ACC_LOOP	23	23	4	2	2	11	yes

The initiation intervals indicates the number of cycle delays for the next loop (refer to the following figure).



From the above figure we can conclude a simple relation of the latency and the initiation interval:
$\text{Latency} = \text{II} \times (N-1) + \text{Internal Latency} - 1$

The "-1" is due to read/write of the function can be done at the same time (same as Lab1).

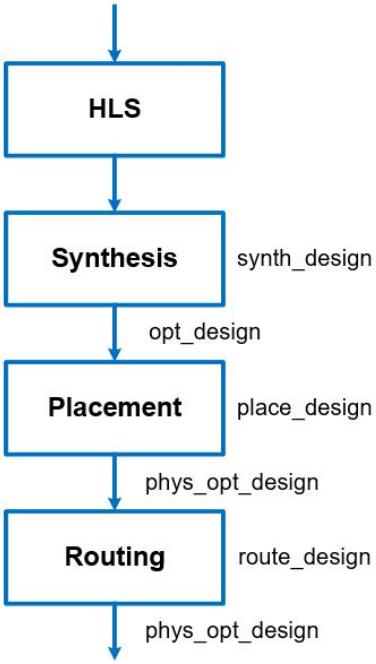
Utilization

The utilization estimation is shown below:

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	3	0	187	-
FIFO	-	-	-	-	-
Instance	4	-	742	882	-
Memory	0	-	64	6	0
Multiplexer	-	-	-	225	-
Register	-	-	374	-	-
Total	4	3	1180	1300	0
Available	280	220	106400	53200	0
Utilization (%)	1	1	1	2	0

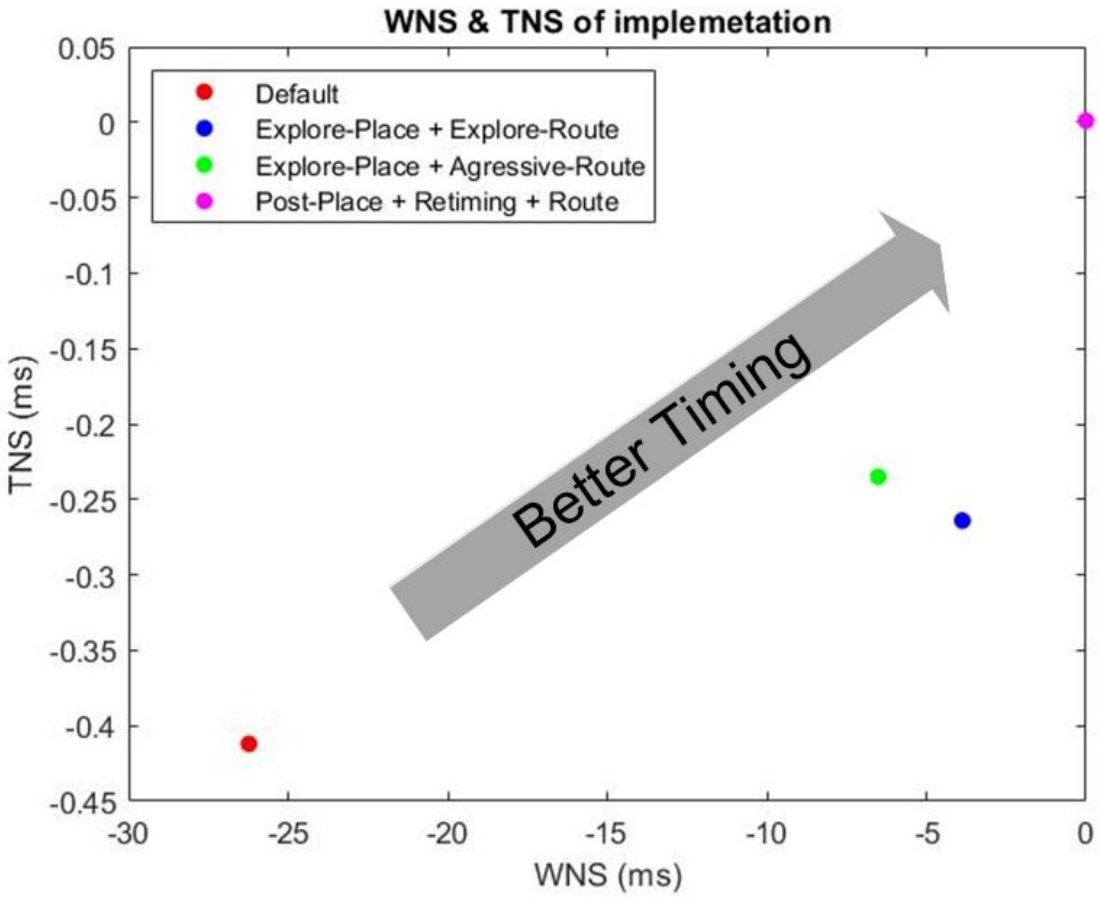
Vivado optimization commands

The following diagram shows the cooresponding commands at each stage.



- Syth_design: perform synthesize
- Opt_design: perform logic & memory optimization
- Place_design: perform placing
- Route_design: perform routing
- Phys_opt_design: physical design optimization (post placed or post route)

The following figure shows the results of different optimization commands under 200M clock rate. The default optimziation will cause negative slack, while our optimization can avoid negative slack.


WNS & TNS of implemetation
Default
Explore-Place + Explore-Route
Explore-Place + Agressive-Route
Post-Place + Retiming + Route
Better Timing
TNS (ms)
WNS (ms)
0.05
0
-0.05
-0.1
-0.15
-0.2
-0.25
-0.3
-0.35
-0.4
-0.45
-30
-25
-20
-15
-10
-5
0

Python code

```
1  import numpy as np
2  from pynq import Overlay, allocate
3
4  ol = Overlay("/home/xilinx/IPBitFile/FIRN11MAXI.bit")
5  ipFIRN11 = ol.fir_n11_maxi_0
6
7  fiSamples = open("samples_triangular_wave.txt", "r+")
8  numSamples = 0
9  line = fiSamples.readline()
10 while line:
11    numSamples = numSamples + 1
12    line = fiSamples.readline()
13
14 inBuffer0 = allocate(shape=(numSamples,), dtype=np.int32)
15 outBuffer0 = allocate(shape=(numSamples,), dtype=np.int32)
16 fiSamples.seek(0)
17 for i in range(numSamples):
18     line = fiSamples.readline()
19     inBuffer0[i] = int(line)
20 fiSamples.close()
21
22 numTaps = 11
23 n32Taps = [0, -10, -9, 23, 56, 63, 56, 23, -9, -10, 0]
24 n32DCGain = 0
25 for i in range(numTaps):
26     n32DCGain = n32DCGain + n32Taps[i]
27     ipFIRN11.write(0x40 + i * 4, n32Taps[i])
28 if n32DCGain < 0:
29     n32DCGain = 0 - n32DCGain
30 ipFIRN11.write(0x80, len(inBuffer0) * 4)
31 ipFIRN11.write(0x10, inBuffer0.device_address)
32 ipFIRN11.write(0x18, outBuffer0.device_address)
33 ipFIRN11.write(0x00, 0x01)
34 while (ipFIRN11.read(0x00) & 0x4) == 0x0:
35     continue
```

- Line 4: Build Overlay class and burn the bitstream file into the Board.
- Line 7~12: Read the data of the weight vector of FIR filter.
- Line 14~15: Allocate a continue physical memory for input and output by Xlnk.
- Line 16~20: Load the data into the DDR memory in PS side.
- Line 25~27: Write the data of the weight vector of FIR filter. (Coefficient for AXI-Master)
- Line 30: Write the inBuffer size to addr. BS+0x80 (Transfer length for AXI-Master)
- Line 31: Write the inBuffer addr. to addr. BS+0x10 (Input data for AXI-Master)
- Line 32: Write the outBuffer addr. to addr. BS+0x18 (Output data for AXI-Master)
- Line 33: Trigger data in addr. 0x0 to be 0x1 (ap_start for AXILiteS)
- Line 34: Wait data in addr. 0x0 be 0x4 (ap_idle for AXILiteS)

Encountered bugs

- 5ns clock would cause negative slack in Vivado implementation.

Learnt

- The function of the AXILiteS and AXI-Master protocal.
- The usage of pipeline directive in the Vivado-HLS.
- Optimization commands in Vivado

Lab2-2

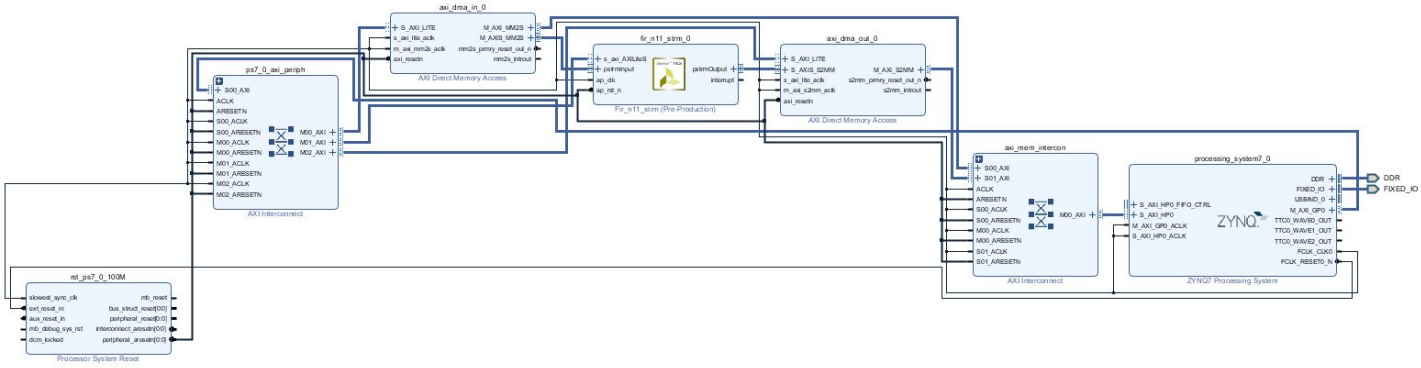
Objective

After completing this lab, we will be able to learn:

- Use Vivado-HLS API to generate a 11-tap FIR filter RTL IP with AXI-Stream interface from System-C code.
- Build a software project and verify the FIR design functionality in hardware.
- [Additional] Study and implement the pipeline directive on Vivado-HLS.

Block diagram

This lab intends to develop a **11-tap FIR filter** through the AXI-Stream interface. Notice that since AXI-Stream does not have address transfered, it requries Directly Memory Access (DMA) for input and output data control.



Interface

This design utilize the **AXI-Stream interface** to transfer the data and **AXILiteS interface** to transfer parameters. The following table shows the usage for some address.

Addr. (+Base addr.)	bit	Usage
0x00	0	ap_start
0x00	1	ap_done
0x00	2	ap_idle
0x00	3	ap_ready
0x00	7	auto_restart
0x04	0	Global interrupt
0x80		transfer length
0x40~0x7f		Coefficient

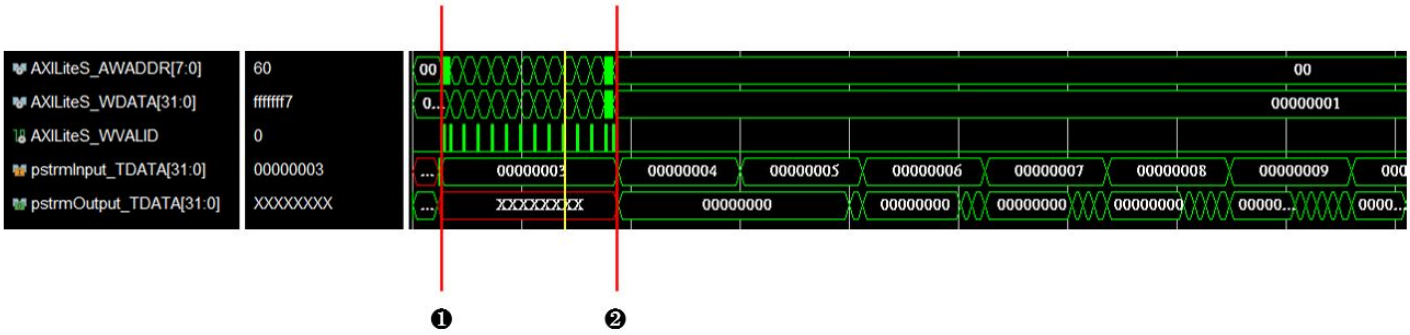
C code

```
1  #include "fir.h"
2
3  void fir_n11_strm(stream_t* pstrmInput, stream_t* pstrmOutput
       , int32_t an32Coef[MAP_ALIGN_4INT], reg32_t regXferLeng)
4  {
5       static int32_t an32ShiftReg[N];
6       int32_t n32Acc;
7       int32_t n32Data;
8       int32_t n32Temp;
9       int32_t n32Loop;
10      int32_t n32NumXfer4B;
11      int32_t n32XferCnt;
12      value_t valTemp;
13
14      n32NumXfer4B = (regXferLeng + (sizeof(int32_t) - 1)) / sizeof(int32_t);
15      XFER_LOOP:
16      for (n32XferCnt = 0; n32XferCnt < n32NumXfer4B; n32XferCnt++) {
17              n32Acc = 0;
18              value_t valTemp = pstrmInput->read();
19              n32Temp = valTemp.data;
20              SHIFT_ACC_LOOP:
21              for (n32Loop = N - 1; n32Loop >= 0; n32Loop--) {
22                      if (n32Loop == 0) {
23                              an32ShiftReg[0] = n32Temp;
24                              n32Data = n32Temp;
25                      } else {
26                              an32ShiftReg[n32Loop] = an32ShiftReg[n32Loop - 1];
27                              n32Data = an32ShiftReg[n32Loop];
28                      }
29                      n32Acc += n32Data * an32Coef[n32Loop];
30              }
31              valTemp.data = n32Acc;
32              pstrmOutput->write(valTemp);
33              if (valTemp.last) break;
34      }
35      return;
36 }
```

Most of the code is the same as Lab2-1, except:

- Line 18~19: The data is read by a stream class function.
- Line 31~32: The data is write by a stream class function.

Cosim waveform



❶~❷ The initialization steps are all the same as ❶~❼ in Lab2-1.
❷~ The data input and output is controlled by the DMA.

From the waveform, we can see that AXI-Stream's input and output will flow in and out at the same rate, while AXI-Master's output will lag a lot after its input.

Performance and Analysis

We can see that it **consumes 99~110 cycles** for calculation and the **estimated clock rate is 4.102ns** from the report below:

Clock	Target	Estimated	Uncertainty
ap_clk	5.00 ns	4.102 ns	0.63 ns

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- XFER_LOOP	?	?	102 ~ 113	-	-	?	no
+ SHIFT_ACC_LOOP	99	110	9 ~ 10	-	-	11	no

Notice that pipeline is still allow in this design, same as Lab2-1. For instance, we set II to 2, then we can done the calculation in 27 cycles.

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- XFER_LOOP	?	?	31	-	-	?	no
+ SHIFT_ACC_LOOP	27	27	8	2	2	11	yes

Utilization

The utilization estimation is shown below:

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	167	-
FIFO	-	-	-	-	-
Instance	2	3	369	175	-
Memory	0	-	64	6	0
Multiplexer	-	-	-	164	-
Register	-	-	354	-	-
Total	2	3	787	512	0
Available	280	220	106400	53200	0
Utilization (%)	~0	1	~0	~0	0

Python code

```
1  import sys
2  import numpy as np
3  from pynq import Overlay, allocate
4
5  ol = Overlay("/home/xilinx/IPBitFile/FIRN11Stream.bit")
6  ipFIRN11 = ol.fir_n11_strm_0
7  ipDMAIn = ol.axi_dma_in_0
8  ipDMAOut = ol.axi_dma_out_0
9
10 fiSamples = open("samples_triangular_wave.txt", "r+")
11 numSamples = 0
12 line = fiSamples.readline()
13 while line:
14    numSamples = numSamples + 1
15    line = fiSamples.readline()
16
17 inBuffer0 = allocate(shape=(numSamples,), dtype=np.int32)
18 outBuffer0 = allocate(shape=(numSamples,), dtype=np.int32)
19 fiSamples.seek(0)
20 for i in range(numSamples):
21    line = fiSamples.readline()
22    inBuffer0[i] = int(line)
23 fiSamples.close()
24
25 numTaps = 11
26 n32Taps = [0, -10, -9, 23, 56, 63, 56, 23, -9, -10, 0]
27 n32DCGain = 0
28 for i in range(numTaps):
29    n32DCGain = n32DCGain + n32Taps[i]
30    ipFIRN11.write(0x40 + i * 4, n32Taps[i])
31 if n32DCGain < 0:
32    n32DCGain = 0 - n32DCGain
33 ipFIRN11.write(0x80, len(inBuffer0) * 4)
34 ipFIRN11.write(0x00, 0x01)
35 ipDMAIn.sendchannel.transfer(inBuffer0)
36 ipDMAOut.recvchannel.transfer(outBuffer0)
37 ipDMAIn.sendchannel.wait()
38 ipDMAOut.recvchannel.wait()
```

Most of the code is the same as Lab2-1, except:

- Line 33: Write the inBuffer size to addr. BS+0x80 (Transfer length for AXI-Master)
- Line 34: Trigger data in addr. 0x0 to be 0x1 (ap_start for AXILiteS)
- Line 35~36: Transfer the input_buffer to the send DMA, and read back from the recv DMA to the output buffer.
- Line 37~38: The wait() method ensures the DMA transactions have completed.

Leanrt

- The function of the AXILiteS and AXI-Stream protocal.

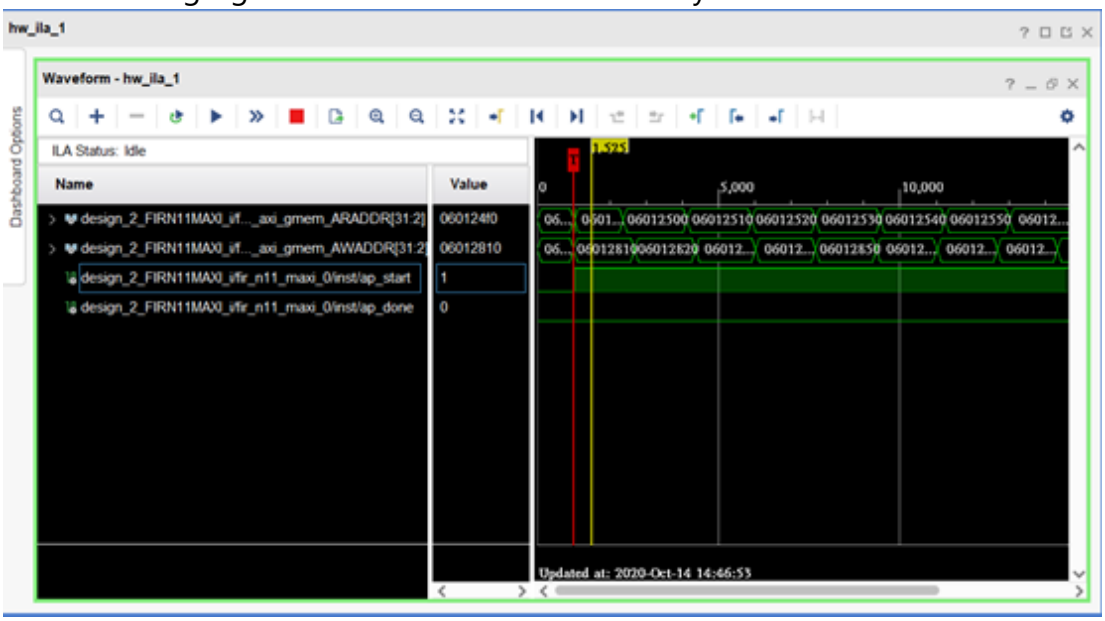
Lab2-3 ILA

Objective

- Add the debug core for specific nets and burn into the board (Use Lab2-1 as an example here).
- See waveform on the hardware manager after triggering the board by the python application code.

Implemetation Result

The following figure shows the waveform see by the ILA.



Encounter bugs

- The sample of data depth cannot be 131072 (not support by this version).
- The hardware can only be burn once (through program botton or python code).
- The ‘Overlay’ part in the python code can only be run once; otherwise, re-burned the hardware will cause false trigger.