tags: MSOC

MSOC FP_ACCUM Report

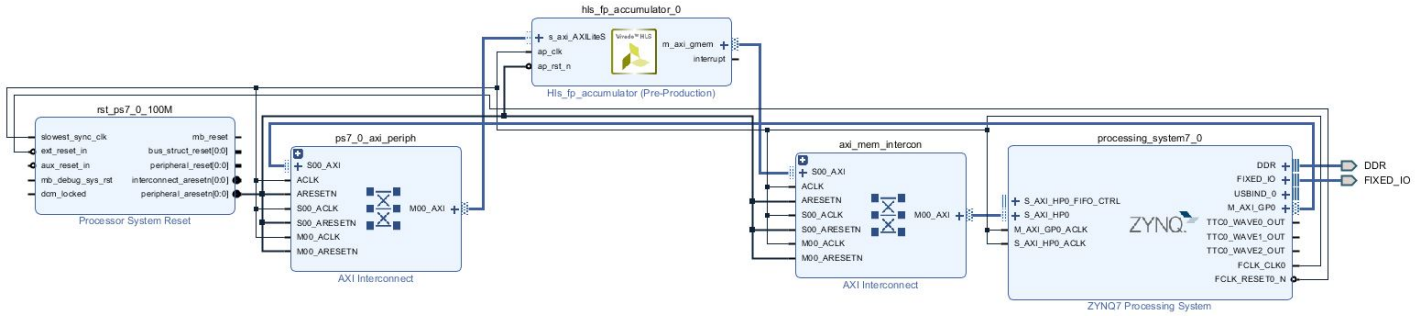
Objective

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

- Analyze the latency and hardware cost by implementing the operation in different structure
- [Additional] Further optimize the baseline hls solution.
- [Additional] Use Vivado-HLS API to generate a Floating-point accumulator RTL IP with AXI_M interface from C code.
- [Additional] Build a PYNQ Host program and verify the design functionality on FPGA.

System diagram

This lab intends to develop a **Floating-point Accumulator** through the AXI_M interface.



Interface

This design utilize the **AXIM interface** to transfer the input and output data. 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
0x10		input data addr.
0x18		output data addr.

HLS implementation & optimization

What we have optimized will be marked **Bold**.

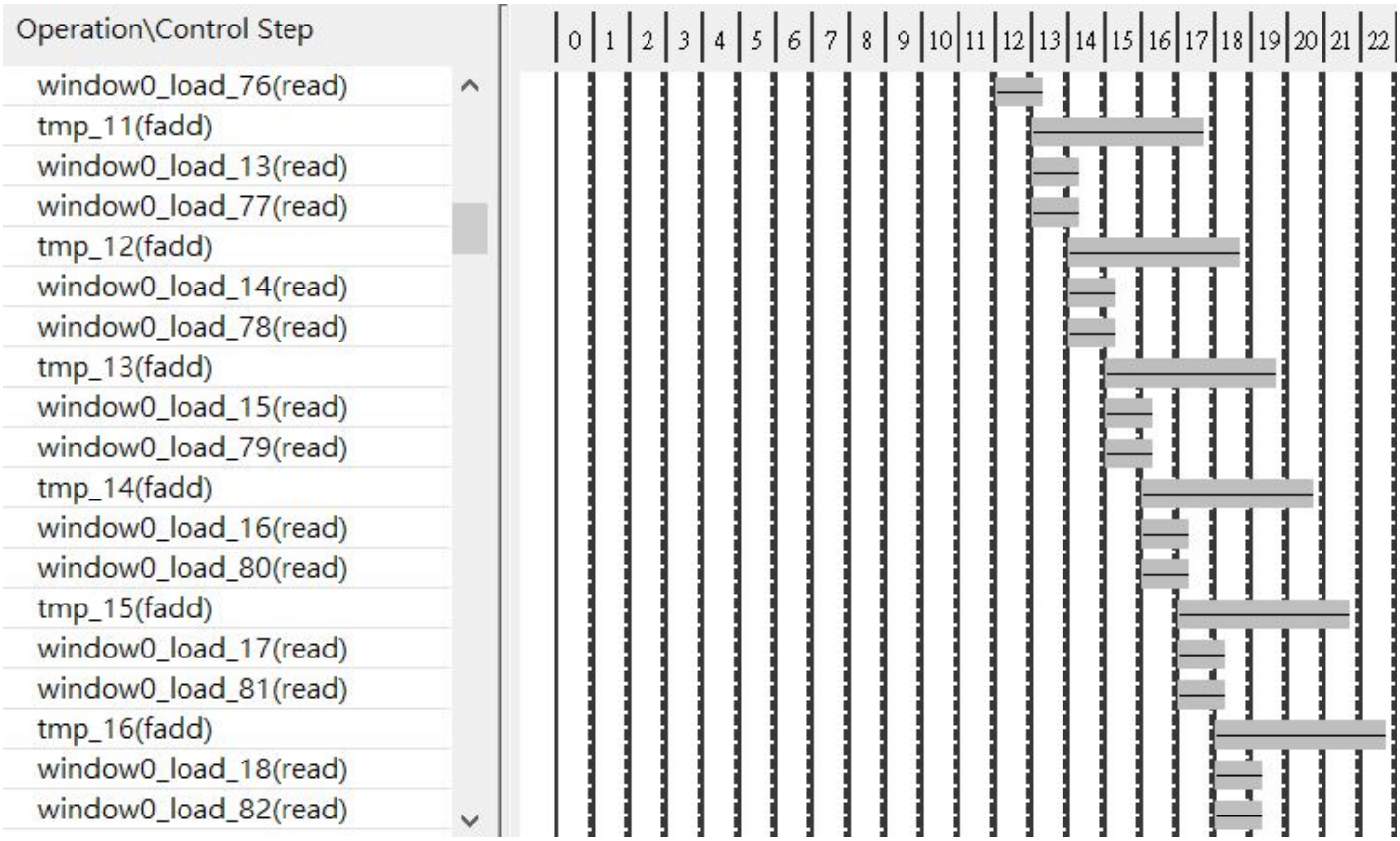
Solution 1 (Single adder accumulator)

Follow the concept of calculating latency in Report “CORDIC_SQRT”, we can get **the total latency 5x127+7+1=643 (II=5, array_size=128, iteration_latency=7)**.

Notice that the **II is 5 in this solution** (due to 5 cycle latency of the fadd operation), and **we cannot pipeline the “fadd” into multiple stage** to further reduce II (just as pipelined the multiplier in “CORDIC_SQRT”) since vivado hls does not provide pipelined “fadd” resources.

Solution 2 (Adder-tree accumulator)

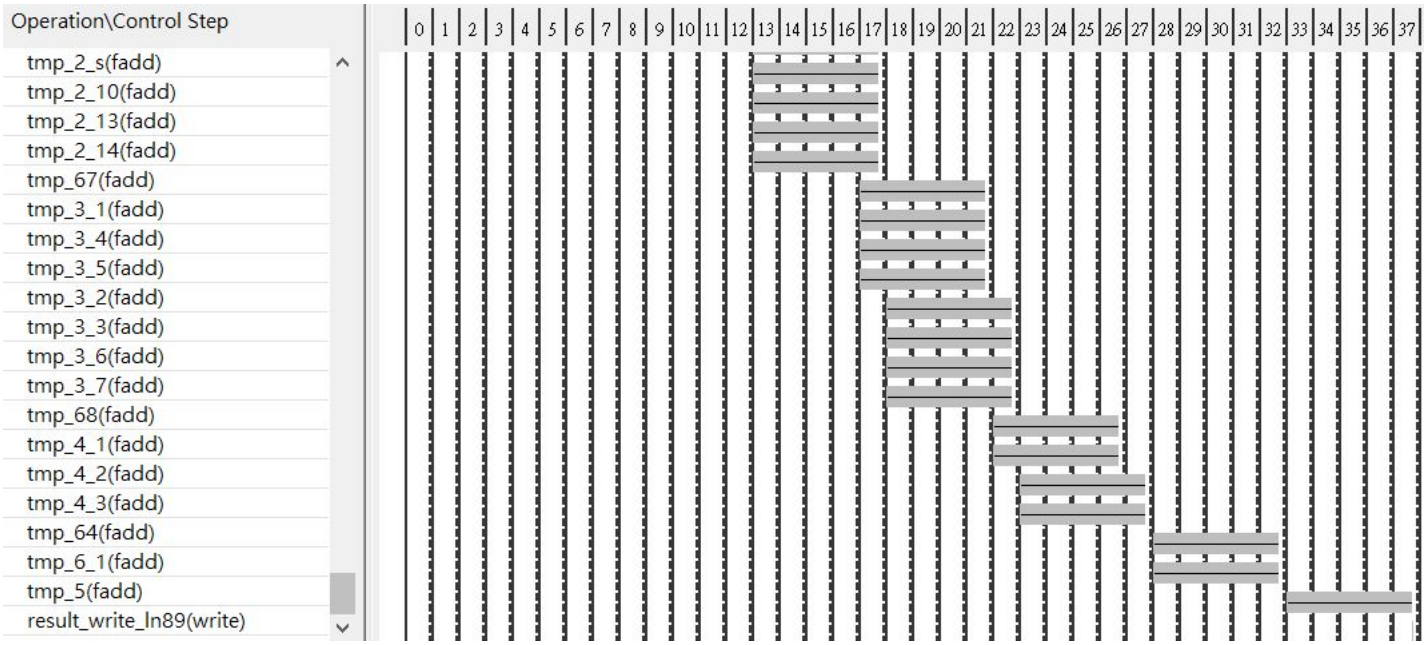
The following figure shows a portion of the top-level timeline.



We aim to parallize "fadd" module, but the memory is default specify as a dual-port BRAM. Therfore, the operation is memory bounded. Thus, we need to further partition the array to increase the data access bandwidth.

Solution 3 (Solution2 + array partition)

We set **array partition** on the array to further increase the data access bandwidth. The following figure shows a portion of the adder-tree structure. Notice that we only partition 32 for the 128 size array (since hardware resource of the**full partition case will exceed the resource of the board**), therefore, it requires 2 cycle for the dual-port BRAM.



Solution 4 (Solution1 + Interface)

We add AXI_M interface for the size 128 floating point input array and AXILite for the output value.

Comparison Table

The final latency table:

		solution1	solution2	solution3	solution4
Latency (cycles)	min	643	233	37	650
	max	643	233	37	650
Latency (absolute)	min	6.430 us	2.330 us	0.370 us	6.500 us
	max	6.430 us	2.330 us	0.370 us	6.500 us
Interval (cycles)	min	643	64	2	650
	max	643	64	2	650

The final utilization table:

	solution1	solution2	solution3	solution4
BRAM_18K	0	0	0	2
DSP48E	2	4	128	2
FF	264	3486	21269	1027
LUT	497	2193	27838	1361
URAM	0	0	0	0

Csim and Cosim results

Csim:

```
3670058.0000    3670058.0000    00000.0000
TEST OK!
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] ***************** CSIM finish ****************
```

Cosim:

```
INFO: [Common 17-206] Exiting xsim at Wed Dec 23 17:37:04 2020...
INFO: [COSIM 212-316] Starting C post checking ...

3670058.0000    3670058.0000    00000.0000
TEST OK!
INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS ***
```

PYNQ implementation

Our host program random generates 128 32-bit floating point data and check where the value of the kernel output is exactly the same with software. The interconnection and interface is shown in the beginning of the report.

```
Entry: /usr/lib/python3/dist-packages/ipykern
System argument(s): 3
Start of "/usr/lib/python3/dist-packages/ipyk
============================
Kernel execution time: 0.0001461505889892578
software sum: 64.0405883789...
hardware sum: 64.0405883789...
total error:  0.0
TEST OK!

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