Lab: Introduction to Task Level Optimization

Lab Prerequisites:

HLS Version	**2017.4 or 2018.1**

Introduction

The LAB focuses on analyzing task execution based on the co-simulation waveforms. The sample design used throughout the various steps of the LAB needs adaptation in order to obtain the expected task parallelism that the DATAFLOW pragma (a.k.a. directive) enables.

Objectives

1. Implement C code modifications to enable task parallelism with DATAFLOW
2. Learn to analyze waveforms to confirm the impact of the optimization

Procedure

General Flow of the Lab

- Step 1: Review of a*p_ctrl_chain* IO protocol
- Step 2: DATAFLOW optimization requiring adaptation for **outputs**
- Step 3: DATAFLOW optimization requiring adaptation for **inputs**
- Step 4: DATAFLOW optimization requiring adaptation for **middle function**
- Step 5: Analysis of Performance Improvements

Step 1: The ap_ctrl_chain IO Protocol

In this step we learn about the a*p_ctrl_chain* block-level I/O protocol which is also the handshaking protocol used between tasks in a DATAFLOW region. This protocol uses a valid, ready handshaking scheme between the producer and consumer tasks and implements ping-pong buffers as channels between these tasks.

The protocol signals of interest are:

Input Handshake

- *ap_start [input, forward path]* - The core has its arguments setup and is asked to start the current instance on this cycle, if possible.

- *ap_ready [output,reverse path]* - The core has processed its arguments and informs that it can start the next instance on the next cycle if asked.

Output Handshake

- *ap_done [output, forward path]* - The core has setup its return value and informs that it finished all the computation of the current instance.

- *ap_continue [input, reverse path]* - The core is allowed to give the result of the next instance on the next cycle if any.

The number of iterations of the task is equal to the number of times both *ap_start* and *ap_ready* are “HIGH, this is equal to the number of times at which both *ap_done* and *ap_continue* are “HIGH”.

The following diagram depicts 3 tasks, the middile one is both a consumer and producer. The task on the left is a producer only and the one on the right is a consumer only.

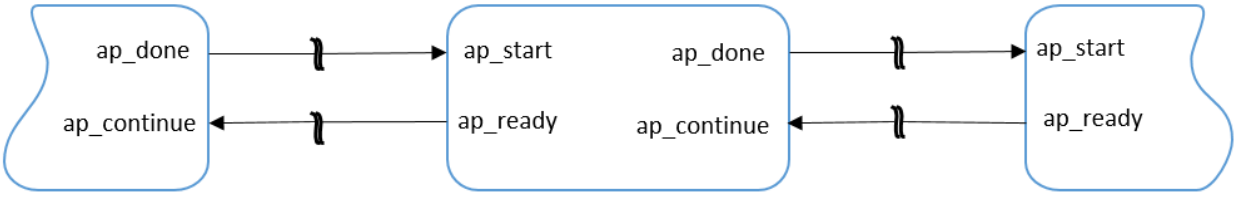

Figure: ***ap_ctrl_chain* Interface**

Step 2: DATAFLOW Optimization – Output Bypass

In this section we'll first run a script to create a project that we'll open in the Vivado HLS GUI to review co-sim waveforms and inspect protocol signals.

1. Create and open the first Vivado HLS project:

a. Open a terminal

Navigate to the "<LAB_INSTALL_PATH>/TSC/**step2_output_bypass/**" folder. Launch script.tcl to perform C simulation, C synthesis, and Co-simulation.

```
% vivado_hls -f script.tcl
```

b. Open project created above in HLS GUI

```
% vivado_hls -p proj
```

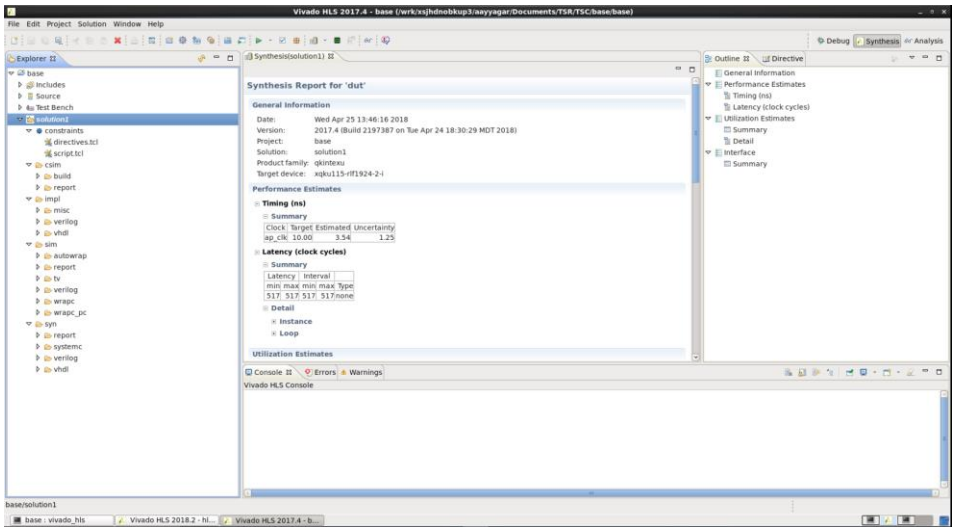

Figure: The Vivado HLS GUI with the created project

The design is a simple function (**dut**) with a as an input, and b and c as outputs (both copies of a). Review the code and move to the next step once ready.

c. Click on the below Icon which opens the RTL waveform (it launches Vivado…)

d. Add waveforms…

Group the signals to each task as shown in the below figure:

1. Block level (**dut**) – The default block level I/O protocol is ap_ctrl_hs. It implements control ports for the design operation and indicates when the design is idle, done and ready for new input data
2. **split**, **pass** – Vivado HLS creates task level control ports to indicate when the design is idle, done and ready for new input data

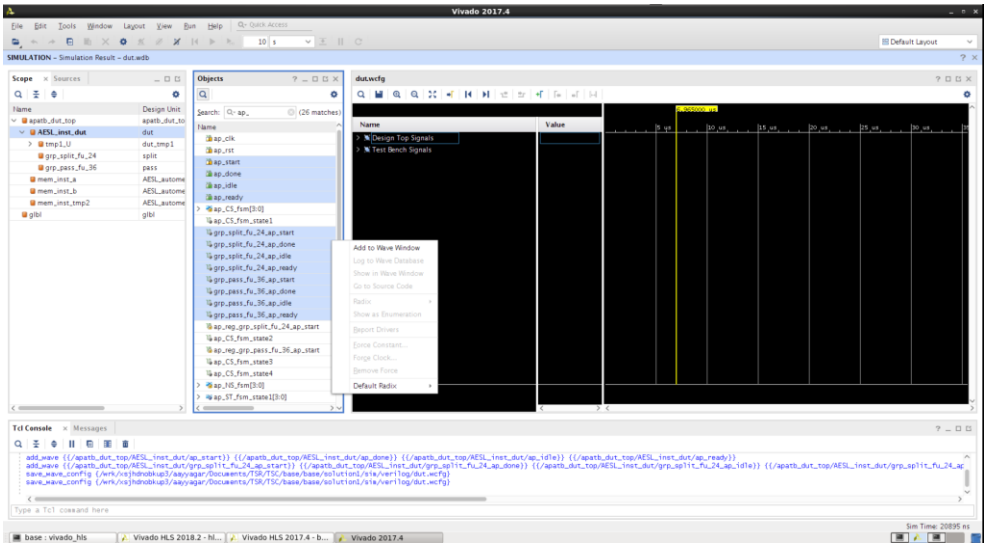

Figure: ***The Vivado waveform viewer***

Output Waveform

Browse and add waveform similarly to below:

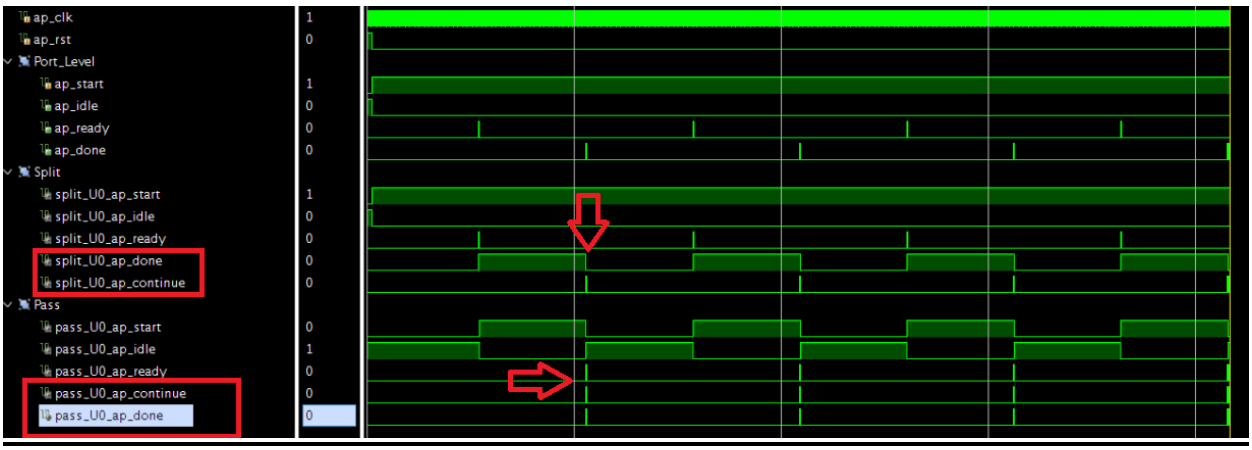

Figure: ***Co-simulation Waveform***

The DATAFLOW design provides an additional handshaking signal, ap_continue that's not created otherwise. The arrows are pointing to the ap_done, ap_continue of the **split**, **pass** task. The ap_done of the **split** task is asserted when it has finished computing. This signal will stay high until it receives ap_continue from the downstream block (**pass** task).

Here is the sneak peek of the code. As you can see we already applied the DATAFLOW pragma to the code.

```
void pass(int tmp1[128], int b[128])
{
#pragma HLS INLINE off

static int j;
  for(int i=0;i<128;i++)
    {
        j++;
      b[i] = tmp1[i];
    }
}

void split(int a[128], int tmp1[128], int tmp2[128])
{
#pragma HLS INLINE off

static int j;
  for(int i=0;i<128;i++)
    {
     j++;
      tmp1[i] = a[i];
      tmp2[i] = a[i];
    }
}

void dut(int a[128], int b[128], int c[128])
{
#pragma HLS DATAFLOW

  int tmp1[128];

  split(a,tmp1,c);
  pass(tmp1,b);

}
```

Figure: ***C Code***

Dataflow Viewer

This view is only available when a DATAFLOW directive is applied to a task. This viewer can be accessed through the Analysis Perspective Window (Click on the analysis view shown below).

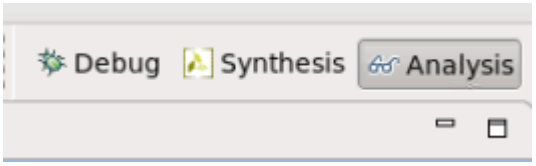

Figure: ***Analysis Perspective Button***

This viewer provides a graphical view of the design structures of the dataflow region, the connections between tasks and the channel implementation (BRAM or FIFO).

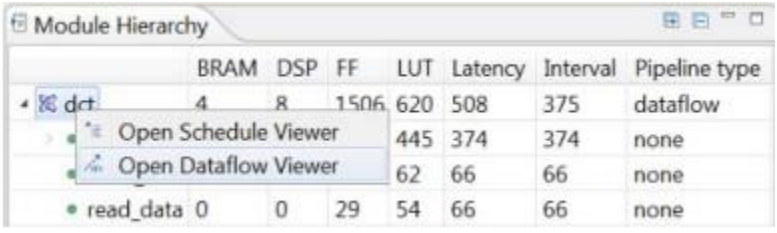

Figure: ***Dataflow Viewer (right-mouse-click)***

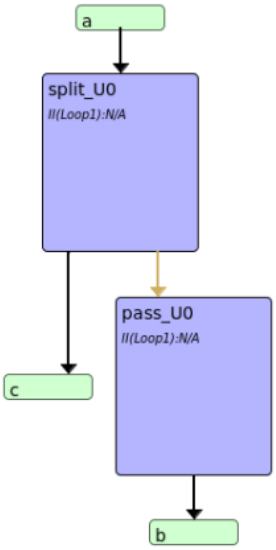

Figure: ***Dataflow Viewer***

Questions

1. After the **split** task asserting ap_done, why split task stalled to start the second iteration? (Hint: ap_continue).
2. Why are the tasks (Split, Pass) executing in sequential order? (Hint: ap_done, ap_continue)
3. What is the II of the Top function? (II – Initiation Interval, it will be available in Co-simulation report)

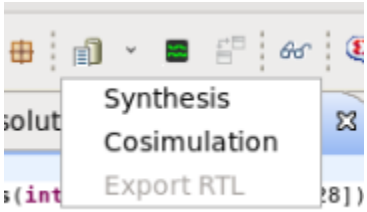

Figure: ***Co-simulation Report***

Solution

- As we studied earlier in the *ap_ctrl_chain* protocol, the *ap_continue* is connected to the *ap_ready* of the next block. This *ap_ready* will only be asserted when “all of its inputs are registered”.
- In this case an internal signal (*ap_continue_sync*) which drives the ap_continue for both the blocks will only be asserted when it receives both inputs, i.e., from the **split** task and the pass task (which is still processing the data).
- This phenomenon will cause the **split** task to stall for few cycles.

Dataflow Viewer

Modify the code as shown in the below figure, we pass the output of the **split** task through a buffer task, which should perform as a copy function. The solution is provided in the "step3_output_bypass/solved_pingpong/" folder.

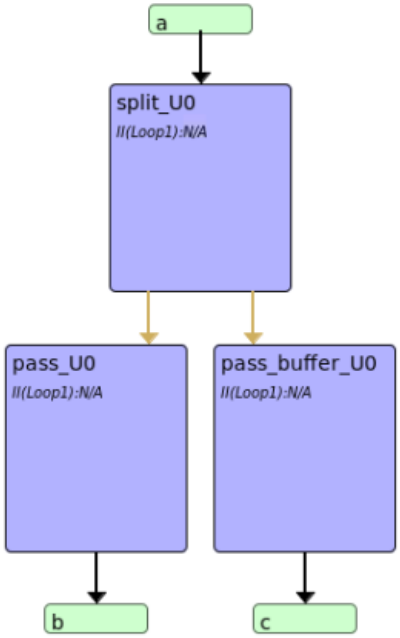

Figure: ***Dataflow viewer***

Output Waveform

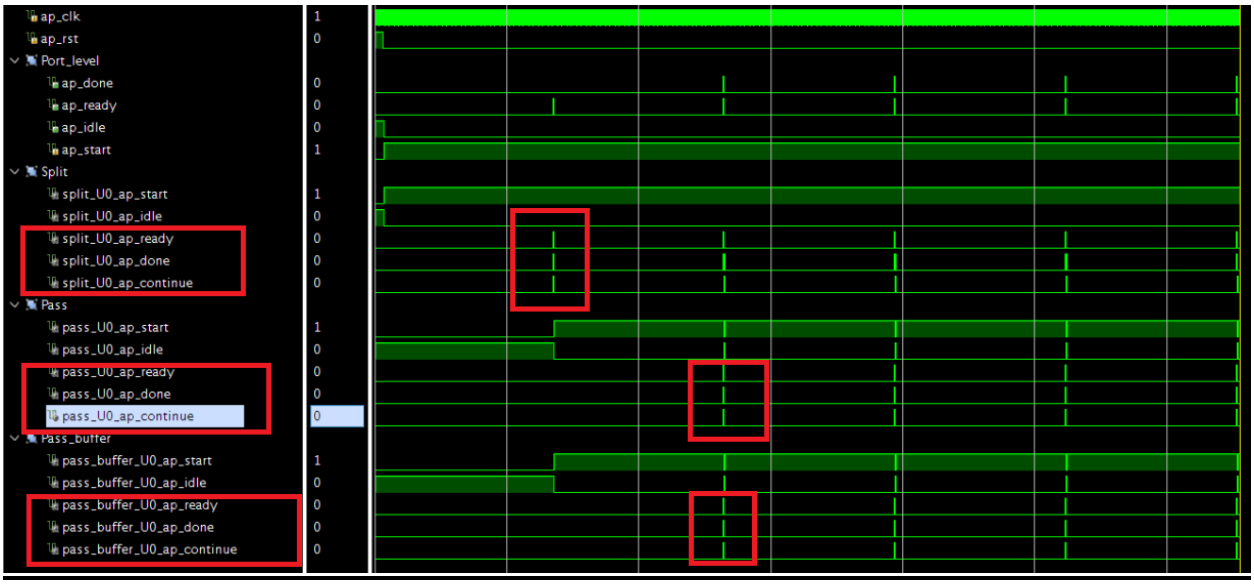

Figure: ***Co-simulation Waveform***

After changing the design structure as described above, the **split** task will start immediately without stalling.

Step 3: DATAFLOW Optimization – Input Bypass

In this section we run the design where in the dataflow region, inputs are read in the middle. We are going to run the design, observe the co-sim and understand how the tasks interact with each other and the limitations when dataflow pragma is used.

Open the step4_input_bypass project folder.

a. Run the Script file
b. Open Vivado HLS GUI
c. Open Dataflow Viewer
d. Open Co-sim Waveform
e. Group the signals of interest

Here is a sneak peek of the code. As you can see in the below figure the b input is bypassing the first task and is read by the double pass function.

```
void add_kernel(int tmp1[128], int tmp2[128], int tmp3[128])
{

  for(int i=0;i<128;i++)
    {
      tmp3[i] = tmp1[i] + tmp2[i];
    }

}

void double_pass(int b[128], int tmp2[128], int tmp1[128], int tmp4[128])
{

  for(int i=0;i<128;i++)
    {
      tmp2[i] = b[i];
      tmp4[i] = tmp1[i];
    }

}

void pass(int a[128], int tmp1[128])
{
  for(int i=0;i<128;i++)
    {
      tmp1[i] = a[i];
    }
}

void dut(int a[128], int b[128], int c[128])
{
#pragma HLS DATAFLOW

  int tmp1[128], tmp2[128], tmp3[128];

  pass(a,tmp1);
  double_pass(b, tmp2, tmp1, tmp3);
  add_kernel(tmp3, tmp2, c);

}
```

Figure: ***C Code***

Dataflow Viewer

The dataflow viewer clearly shows the structure of the design.

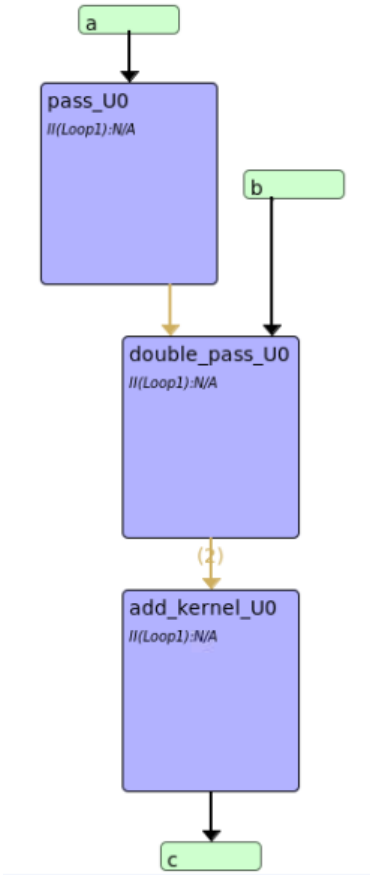

Figure**: *Dataflow Viewer***

Output Waveform

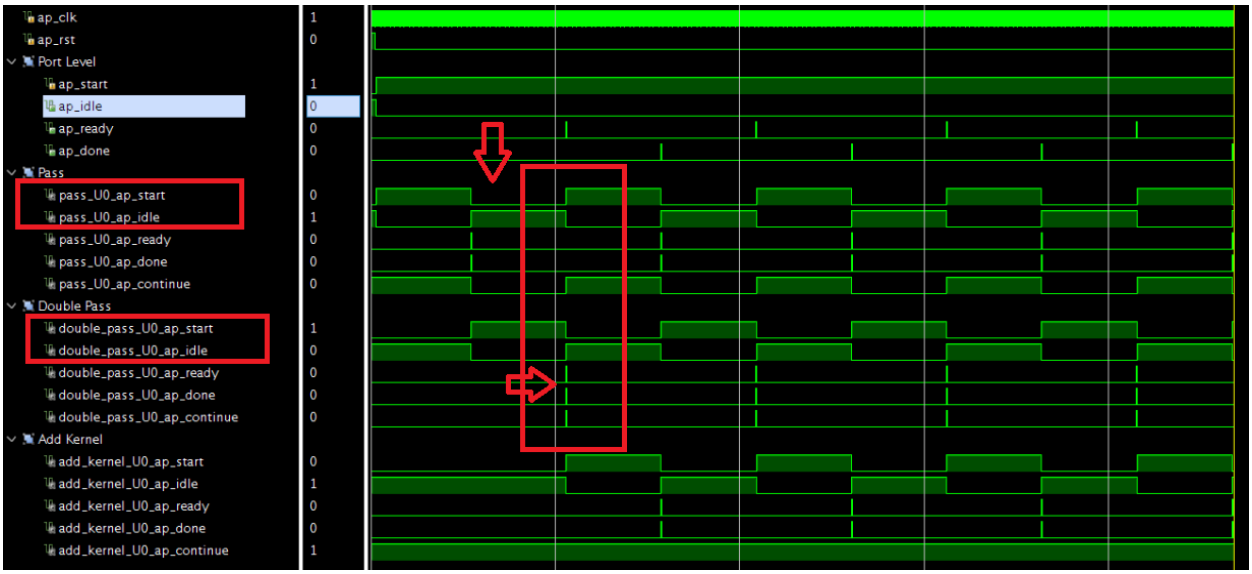

Figure: ***Co-simulation Waveform***

The arrows in the figure point to the ap_start/ap_ready of the **pass, double pass** task. The **pass**, **double pass** could not start the next instance, thought it asserted ap_done and they continue to execute sequentially.

Questions

1. Why the pass, double pass task's executing sequentially. (Hint: ap_ready)
2. What is the II of Top function??

Solution

- The same principle is also applied in this case, the double pass task won't start until it receives all the inputs.
- Once it receives it inputs it asserts *ap_ready* and this will be acknowledged by the internal signal to start the next iteration and then pass will receive a new input.
- This scenario causes the degradation of throughput in the design.

Dataflow Viewer

Modify the code as shown in the below figure. We can either pass both the inputs through the pass task or create a new task and pass it through it. The solution is provided in the "step4_input_bypass/solved_pingpong" folder

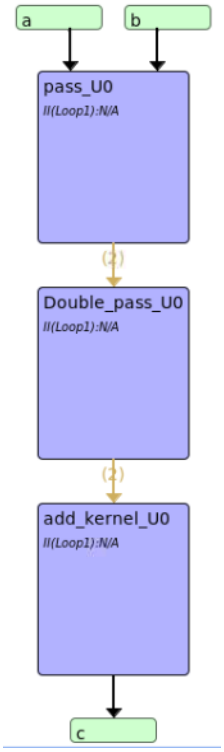

Figure: ***Dataflow Viewer***

Output Waveform

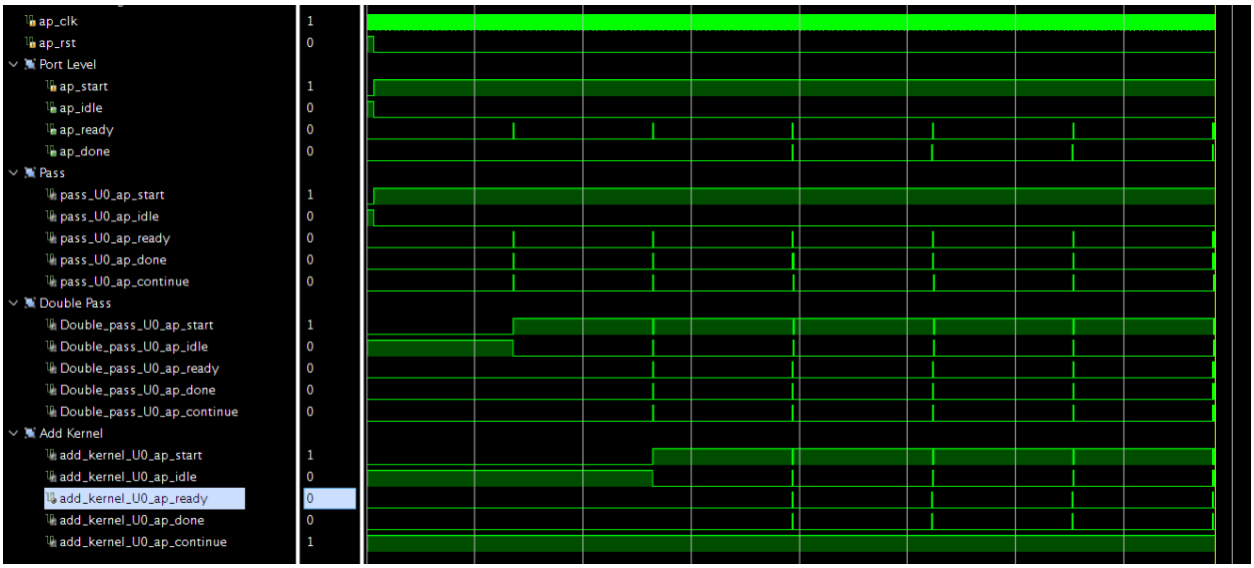

Figure: ***Co-simulation Waveform***

After changing the coding structure as described above, the pass, double pass tasks will start immediately without stalling.

Step 4: DATAFLOW Optimization – Middle Bypass

In this section we run the design where in the dataflow region, the producer and consumer skip a pipeline stage. We are going to run the design, observe the co-sim and understand how the tasks interact with each other and the limitations when dataflow pragma is used.

Open the step5_middle_bypass project folder.

a. Run the Script file
b. Open Vivado HLS GUI
c. Open Dataflow Viewer
d. Open Co-sim Waveform
e. Group the signals of interest

Here is a sneak peek of the code. As you can see in the below figure the intermediate buffer tmp1 is read by the bypass function.

```
void bypass(int tmp1[128], int tmp2[128], int tmp3[128])
{
static int j;
  for(int i=0;i<128;i++)
    {
      j++;
      tmp3[i] = tmp1[i] +tmp2[i];
    }
}


void pass(int tmp2[128], int tmp4[128])
{
static int j;
  for(int i=0;i<128;i++)
    {
      j++;
      tmp4[i] = tmp2[i];
    }
}

void double_pass(int a[128], int b[128], int tmp1[128], int tmp2[128])
{
static int j;
  for(int i=0;i<128;i++)
    {
      j++;
      tmp1[i]= a[i];
      tmp2[i] = b[i];
    }
}

void dut(int a[128], int b[128], int c[128])
{
#pragma HLS DATAFLOW

  int tmp1[128], tmp2[128], tmp3[128];

  double_pass(a,b,tmp1,tmp2);
  pass(tmp2, tmp3);
  bypass(tmp1,tmp3,c);
}
```

Figure: ***C Code***

Dataflow Viewer

The dataflow viewer clearly shows the structure of the design.

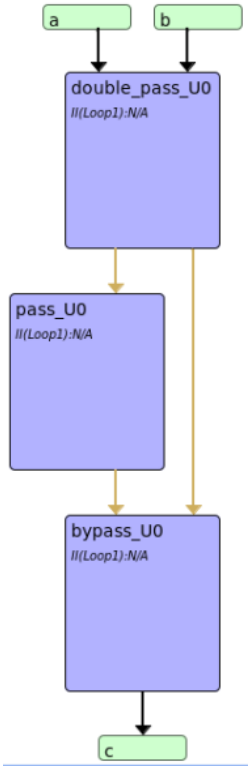

Figure: ***Dataflow Viewer***

Output Waveform

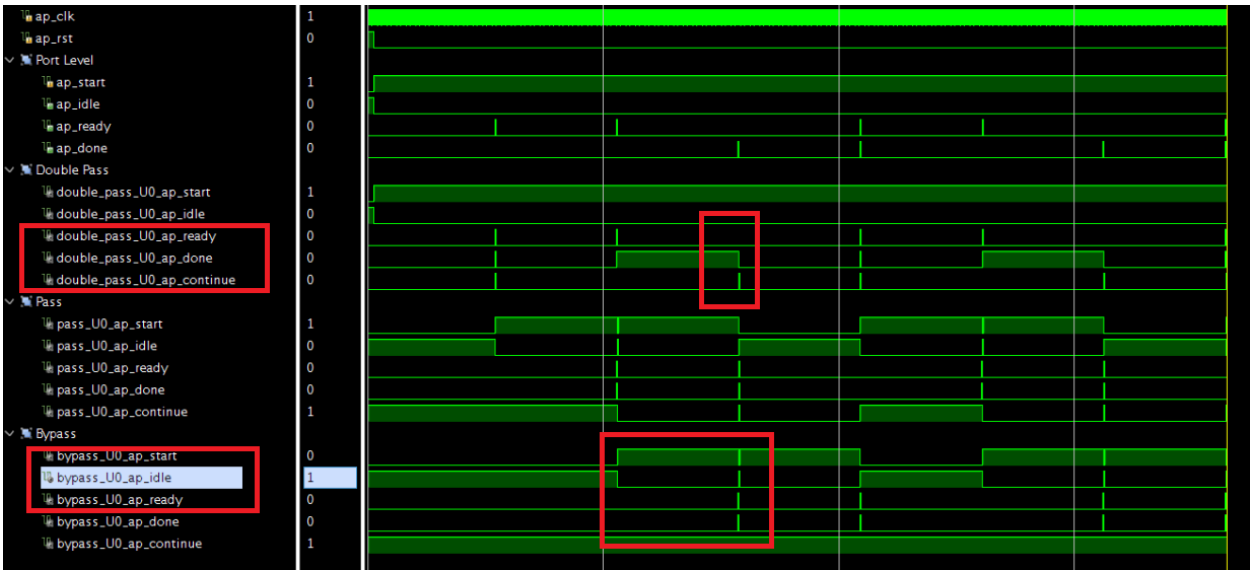

Fig: ***Co-simlation Waveform***

It is difficult to visualize from the above waveform but the below figure should give you a good idea. In the below figure

1. T1 – Double Pass function
2. T2 – Pass function
3. T3 – bypass function
4. B - Bubble.

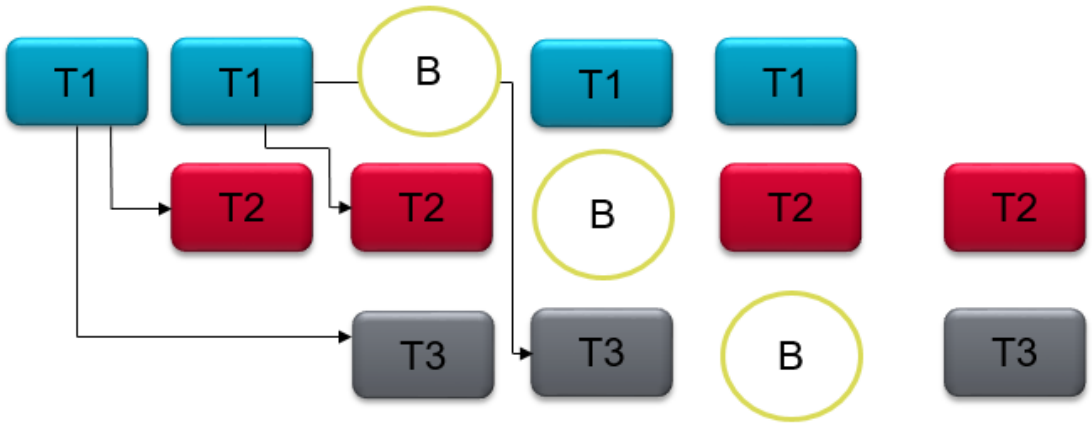

Figure: ***Visualisation for Middle bypass***

Questions

- Why do we have bubbles in between the tasks T1 and T2??

Solution

- The channels between the tasks are implemented as ping-pong buffers.
- T1 will write the ping and the pong of the RAM block before T3 task can start reading.
- When the T3 start's reading the ping channel, T1 cannot write into this ping at the same time and have to stall until T3 has completed.
- This scenario will cause performance degradation.
- The solution to this scenario is to pass the arguments through the pass function.

Dataflow Viewer

- Modify the code as shown in the below figure. The solution is provided in the "step5_middle_bypass/*soved_pingpong*" folder

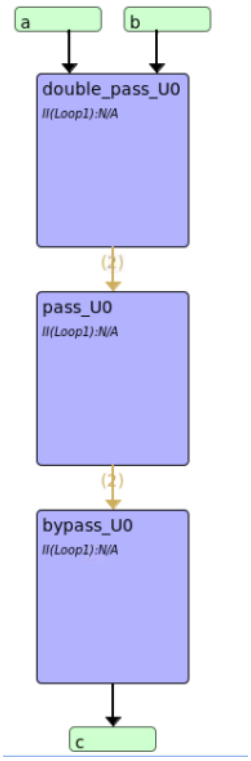

Figure: ***Dataflow Viewer***

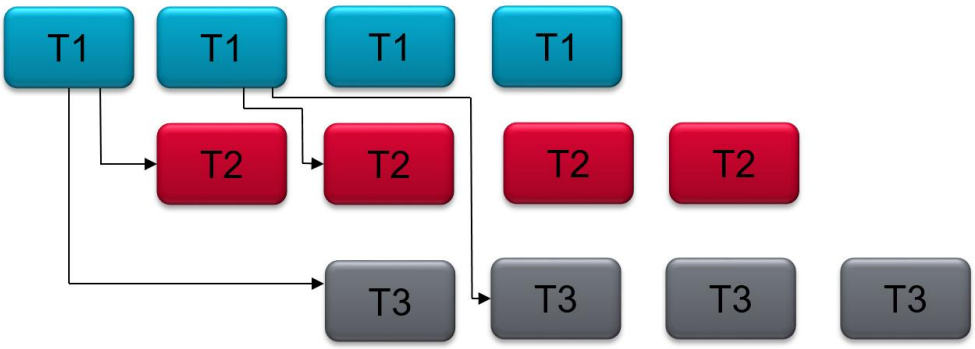

Figure: ***Visualisation for Middle bypass***

Output Waveform

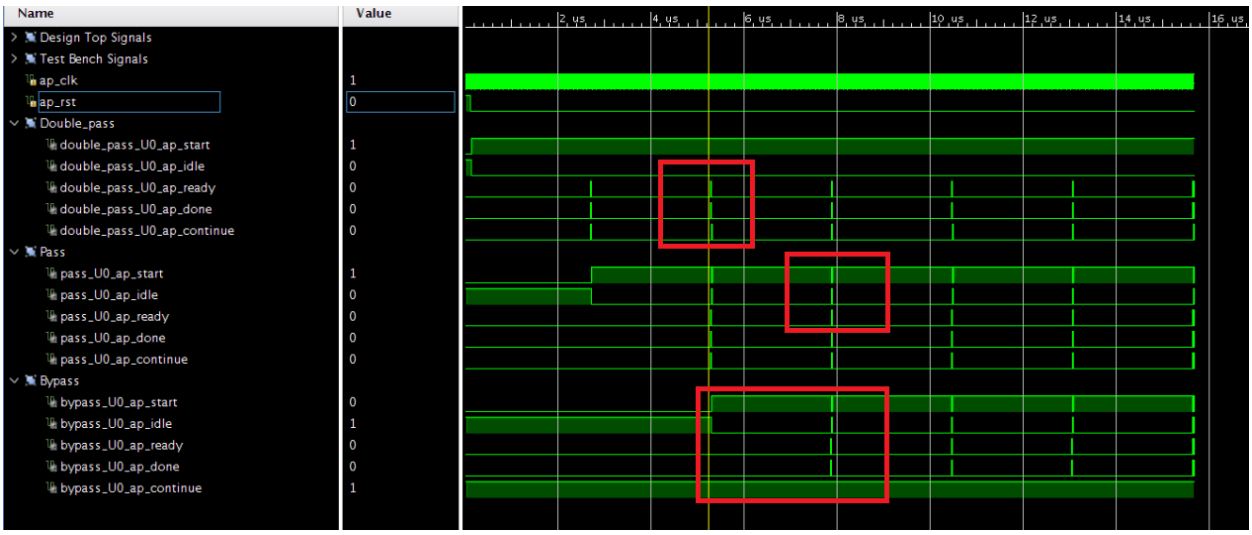

Fig: ***Co-simulation Waveform***

After changing the coding structure as described above, the Double Pass, Pass function will start immediately without stalling.

Step 5: Analysis of Performance Improvements

Test Case	Latency			II		
	Min	Avg	Max	Min	Avg	Max
Output Bypass	515	708	773	258	430	516
Output Bypass - Solved_pingpong	515	516	517	258	258	259
Input Bypass	773	773	773	516	516	516
Input Bypass - Solved_pingpong	773	774	775	258	258	259
Middle Bypass	773	838	1032	258	344	517
Middle Bypass - Solved_pingpong	773	774	775	258	258	259

Note: The above reports are collected from co-simulation using Vivado HLS 17.4