Introduction to High Level Synthesis

Introduction

High level synthesis (HLS) allows the creation of a Register Transfer Level (RTL) design from a high-level language such as C. The RTL design can then be synthesized and implemented onto FPGA hardware. This lab introduces you to the HLS workflow using the Xilinx Vivado HLS software.

Board Support Package

An HLS design needs to target a particular FPGA or SoC. The SoC we will be using in this unit is the Zynq Ultrascale+ MPSoC. Usually these chips are part of a development board such as the Ultra96 which you will be using in later labs. While Vivado comes pre-installed with a number of board support packages, the Ultra96 is not one of them. Hence, you will need to do this manually.

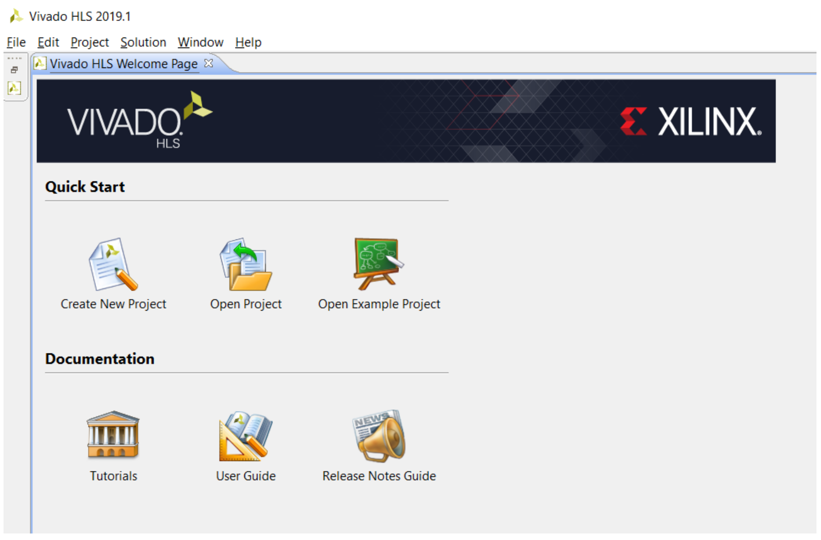
1. Goto https://github.com/Avnet/bdf. Click on the green **Code** button on the webpage and choose '**Download ZIP**'.
2. Unzip the file. You should see a folder called **bdf-master**.
3. Inside bdf-master, find the folder called **ultra96v2**. Copy this folder to **C:\Xilinx\Vivado\2019.1\data\boards\board_files** if you are using Windows or **/tools/Xilinx/Vivado/2019.1/data/boards/board_files** if you are using Linux.

HLS Workflow

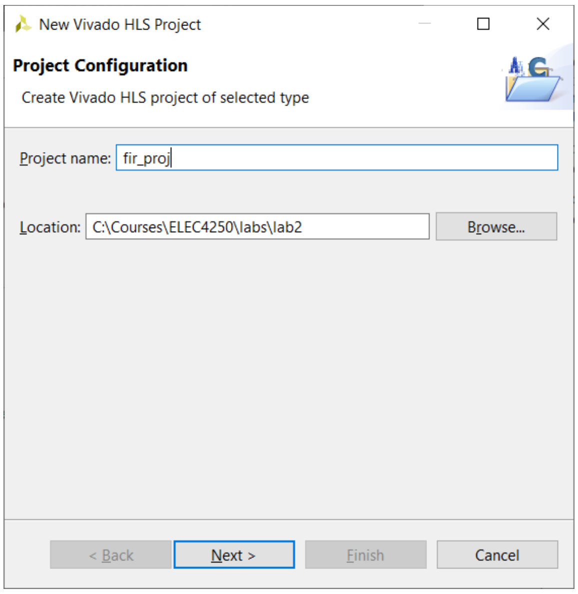
The following steps will guide you through the basic HLS workflow.

Step 1: Creating a High-Level Synthesis Project

1. Download the Lab2 files from iLearn and copy them to a location with no spaces in the path. For example, C:\Courses\ELEC4250\labs\lab2 (Windows) or /home/training/labs/lab2 (Linux).
2. Start **Vivado HLS 2019.1**. Note there are two icons that start with Vivado. Make sure you choose the one that contains HLS in the name. You should see a welcome screen open up when it finishes loading.



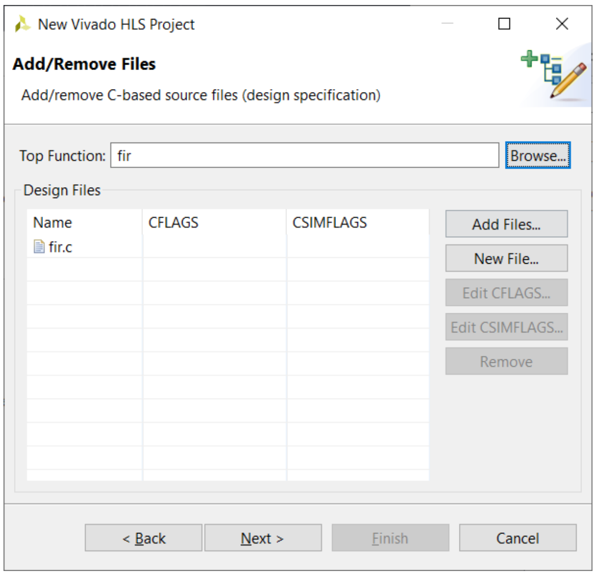
3. Select **Create New Project**.
4. In the Project Configuration window that opens up, enter **fir_proj** for the Project Name. Click on Browse and navigate to the location where you have put the Lab2 files in step 1. This information defines the name and location of the Vivado HLS project directory. Click **Next**.



Project name: fir_proj

Location: C:\Courses\ELEC4250\labs\lab2

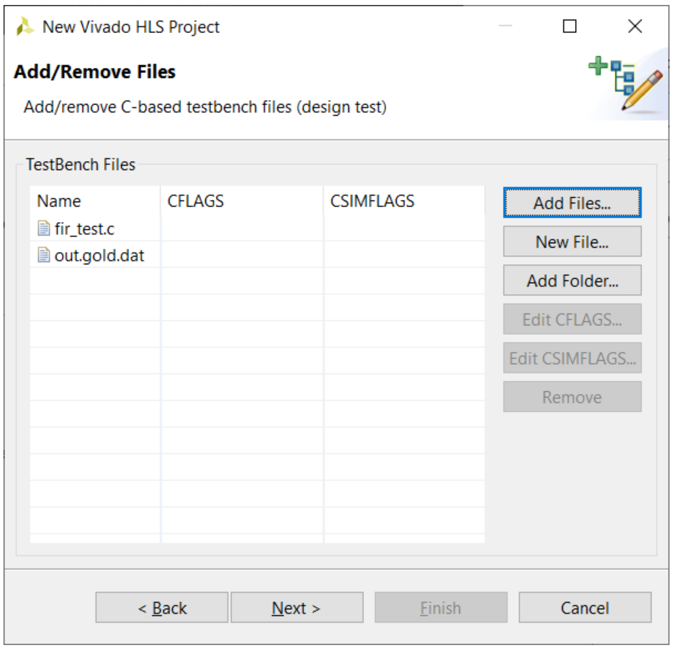
5. The next window is the Add/remove C-based source files window. In this window, we define all the C code source files that we want to use with HLS. We only need to specify the .c files. Any .h files in the same folder will get added automatically. Click on **Add Files** and select **fir.c**. Click **OK**.
6. Click on **Browse** and select **fir** as the top-level function. Click **Next**.



Top Function: fir

Name	CFLAGS	CSIMFLAGS
fir.c		

7. The next window allows us to add/remove C-based testbench files. Testbench files are important for verifying that the output of the HLS-implemented design matches that of the original C code. Click **Add Files** and add both **fir_test.c** and **out.gold.dat**. Click **Next**.



New Vivado HLS Project

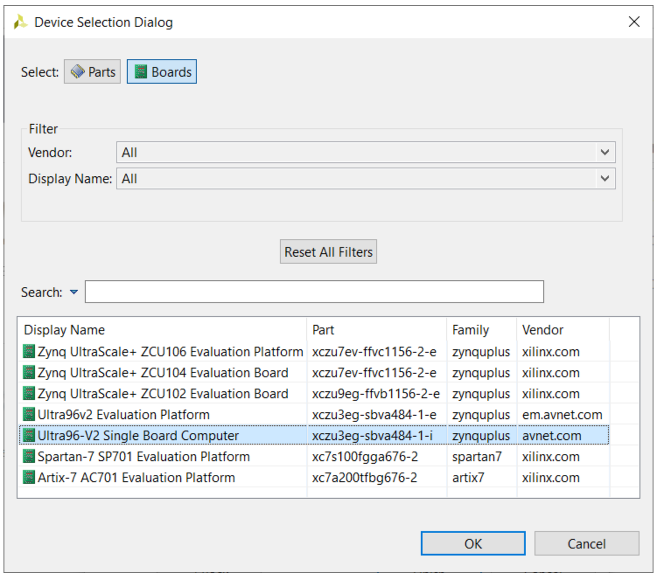
Add/Remove Files

Add/remove C-based testbench files (design test)

TestBench Files

Name	CFLAGS	CSIMFLAGS
fir_test.c		
out.gold.dat		

8. The next window shown is the Solution Configuration Window. A project can have multiple solutions that target different technologies or meet different constraints. For now, we will create a solution that targets the Ultra96 board. Click on **...** next to Parts. Click on **Boards** and choose **Ultra96-v2 Single Board Computer**. Click **OK**.



Device Selection Dialog

Select: Parts Boards

Filter

Vendor: All

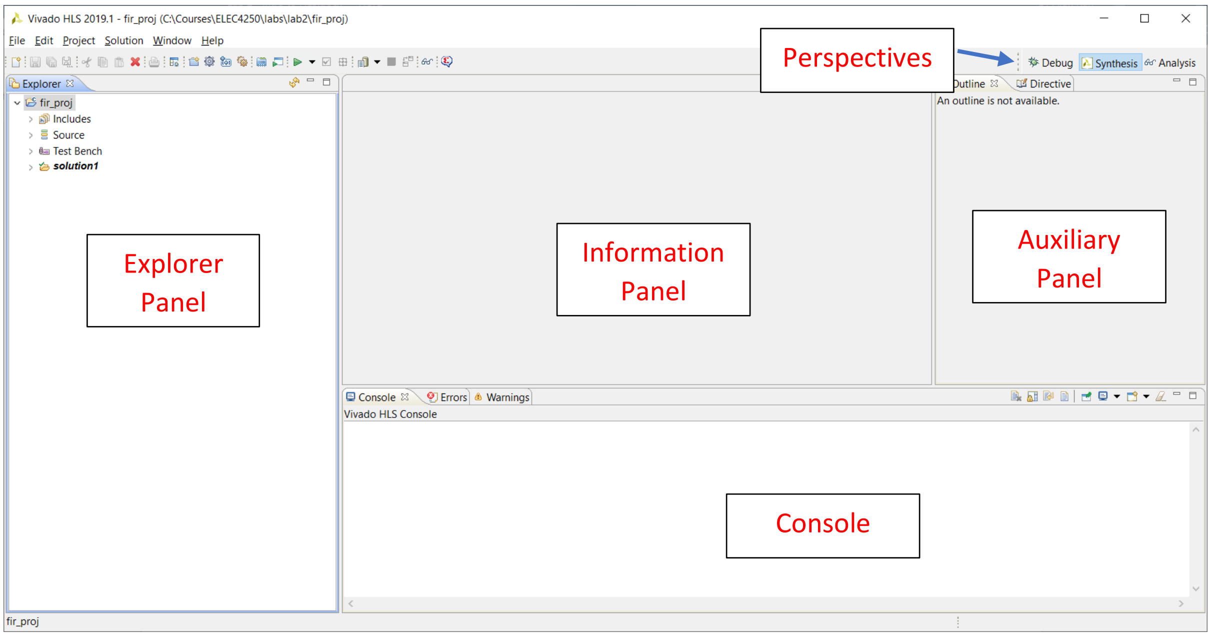
Display Name: All

Search:

Display Name	Part	Family	Vendor
Zynq UltraScale+ ZCU106 Evaluation Platform	xczu7ev-ffvc1156-2-e	zynquplus	xilinx.com
Zynq UltraScale+ ZCU104 Evaluation Board	xczu7ev-ffvc1156-2-e	zynquplus	xilinx.com
Zynq UltraScale+ ZCU102 Evaluation Board	xczu9eg-ffvb1156-2-e	zynquplus	xilinx.com
Ultra96v2 Evaluation Platform	xczu3eg-sbva484-1-e	zynquplus	em.avnet.com
Ultra96-V2 Single Board Computer	xczu3eg-sbva484-1-i	zynquplus	avnet.com
Spartan-7 SP701 Evaluation Platform	xc7s100fgga676-2	spartan7	xilinx.com
Artix-7 AC701 Evaluation Platform	xc7a200tfbg676-2	artix7	xilinx.com

9. Click **Finish**.

10. You should now see the Vivado HLS Project window.



The project window is divided into different regions.

Explorer Panel
Shows the project hierarchy, along with its associated files. For each new 'solution' you create, they will appear in this panel. As you proceed through the different steps of working through the design of the solution (validation, synthesis, verification and IP packaging), subfolders with the results of each step will be created under the solution folder.

Information Panel
Shows the contents of any files opened from the Explorer Panel. Reports are also shown in this pane at the end of different design steps.

Auxiliary Panel
Shows additional information related to that shown in the information panel.

Console
Shows the output/error and warning messages produced when Vivado HLS runs.

Perspectives
Adjusts the panes within Vivado HLS for different design steps.

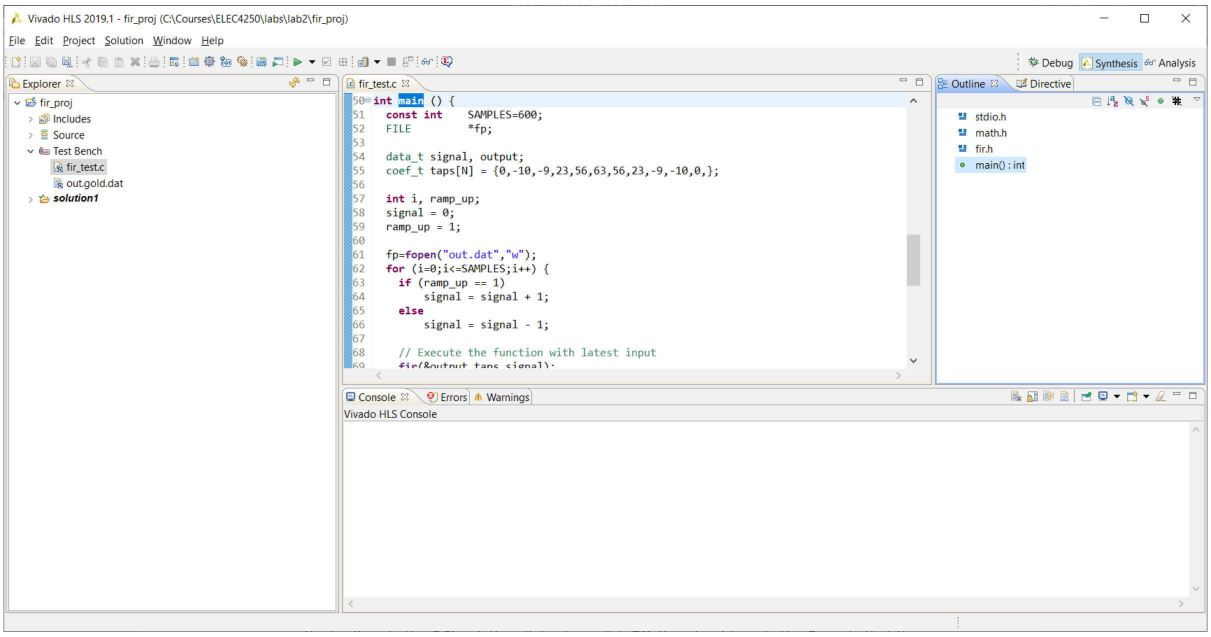
- Debug Perspective - Includes panels for debugging C code. You can open the Debug Perspective after the C code compiles.
- Synthesis Perspective - Allows you to synthesize designs, run simulations and package IP.
- Analysis Perspective - Includes panels that support analysis of synthesis results. You can only use this perspective after synthesis is complete.

Step 2: C Validation

When starting an HLS project, it is important to validate that the C code is correct. To do this, you will need a test bench that compares the output of the function with known values that are correct. In the files that you added to the project, **fir_test.c** is a test bench for **fir.c**.

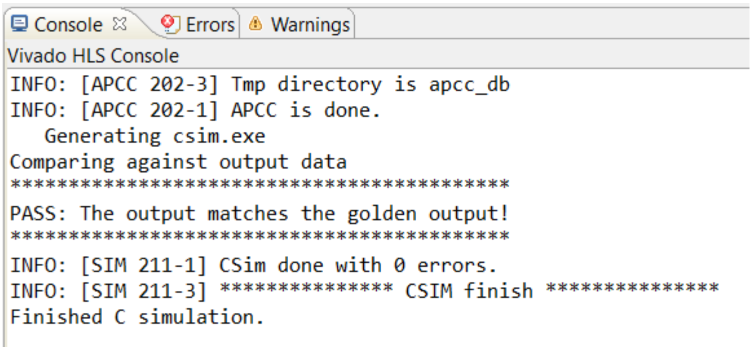
The test bench has some useful characteristics. It saves the output of the fir function into a file called **out.dat**. The output file is compared with the file **out.gold.dat** containing known good values. If the output matches, a message is printed to confirm that the results are correct and the main function returns a value of 0; otherwise, a message will be shown that indicates that the output does not match and the main function returns a value of 1. A test bench with these characteristics allows the Vivado HLS tool to reuse the same test bench to perform RTL verification.

1. Expand the '**Test Bench**' folder in the Explorer panel and double-click on the file **fir_test.c**. In the Auxiliary panel, click on **main()** to view the main function.



```
int main () {
  const int    SAMPLES=600;
  FILE         *fp;

  data_t signal, output;
  coef_t taps[N] = {0,-10,-9,23,56,63,56,23,-9,-10,0,};

  int i, ramp_up;
  signal = 0;
  ramp_up = 1;

  fp=fopen("out.dat","w");
  for (i=0;i<=SAMPLES;i++) {
    if (ramp_up == 1)
        signal = signal + 1;
    else
        signal = signal - 1;

    // Execute the function with latest input
    fir(&output,taps,signal);
```

2. In the **Project** menu, choose **Run C Simulation**. Alternatively, click on the Run C simulation icon on the toolbar. In the C Simulation dialog box, click **OK**. You should see in the Console that the simulation executes successfully and the output message that the output of the fir function matches the golden output.

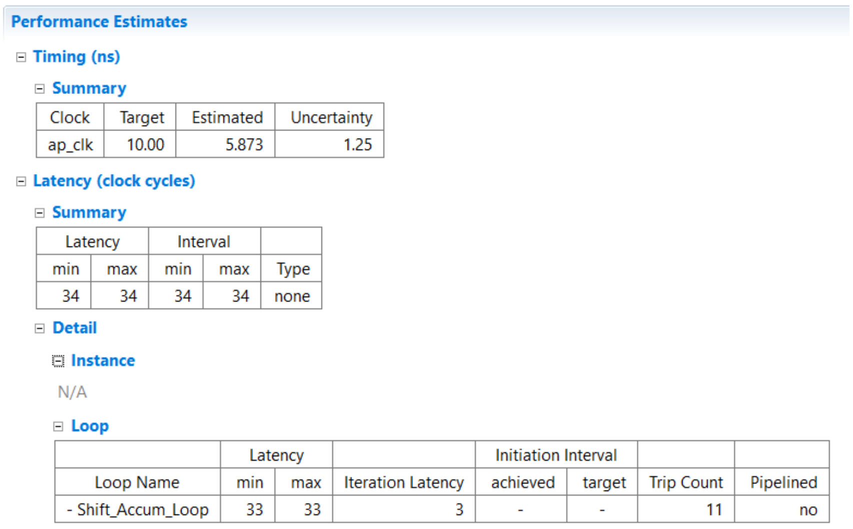


```
INFO: [APCC 202-3] Tmp directory is apcc_db
INFO: [APCC 202-1] APCC is done.
   Generating csim.exe
Comparing against output data
*******************************************
PASS: The output matches the golden output!
*******************************************
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
Finished C simulation.
```

Step 3: High-Level Synthesis

HLS converts a C design into a Register Transfer Level (RTL) design. An RTL design can then be implemented on FPGA hardware.

1. From the **Solution** menu, choose **Run C Synthesis**, then **Active Solution**. Alternatively, click on the 'Run C Synthesis' button in the toolbar.
2. Once synthesis is complete, you should see the solution 1 synthesis report open in the Information Panel. Scroll down to **Performance Estimates**.



Performance Estimates

⊟ **Timing (ns)**

⊟ **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	5.873	1.25

⊟ **Latency (clock cycles)**

⊟ **Summary**

Latency		Interval		
min	max	min	max	Type
34	34	34	34	none

⊟ **Detail**

⊟ **Instance**

N/A

⊟ **Loop**

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Shift_Accum_Loop	33	33	3	-	-	11	no

In its synthesis, Vivado HLS targets a clock period of Clock Target minus Clock Uncertainty (10-1.25 = 8.75 ns in this example). The clock uncertainty ensures there is a timing margin for any unknown delays that may occur after implementing the design into an FPGA.

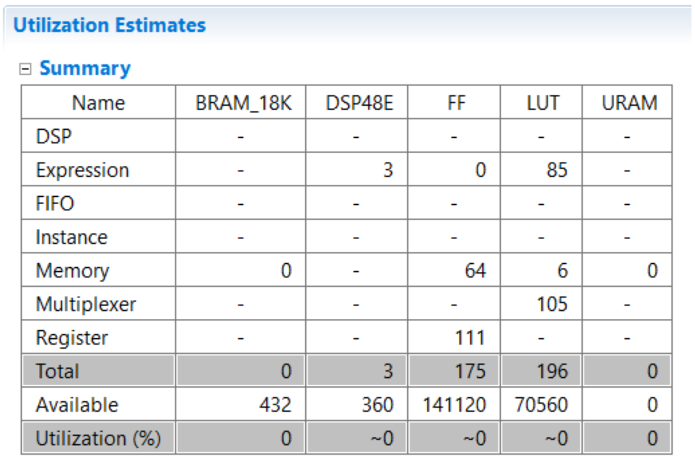
The estimated clock period is the worst-case delay of the synthesized design. Here, it is 5.873 ns which meets the 8.75 ns timing requirement.

In the Summary section, we see that the design has a latency of 34 clock cycles. That is, it takes 34 clock cycles to output the results. We also see that the interval for reading the next set of inputs is 34 clock cycles. This means that the next execution of the function only occurs after the completion of the current execution.

In the Detail section, we see that there are no submodules when we expand the Instance section. When we expand the Loop section, we see that all the latency delay is due to the RTL logic synthesized from the loop named Shift_Accum_Loop. The loop executes 11 times (Trip Count) and each iteration takes 3 clock cycles (Iteration Latency) to yield a total of 33 clock cycles.

Observe that the total latency is once clock cycle greater than the loop latency. This is because it takes one clock cycle to enter and exit a loop. In this case, the design finishes execution when the loop finishes. Hence, there is no exit clock cycle.

3. Scroll down to **Utilization Estimates**.



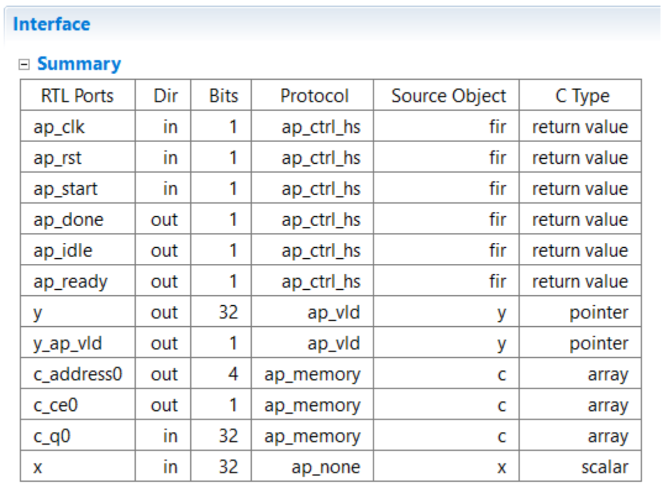
Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	3	0	85	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	0	-	64	6	0
Multiplexer	-	-	-	105	-
Register	-	-	111	-	-
Total	0	3	175	196	0
Available	432	360	141120	70560	0
Utilization (%)	0	~0	~0	~0	0

Here, we can see the estimated FPGA resources that will be used to implement the design. The design will use 3 DSP48E, 175 flip-flops and 196 LUTs (lookup tables).

4. Scroll down to **Interface**.



Interface

Summary

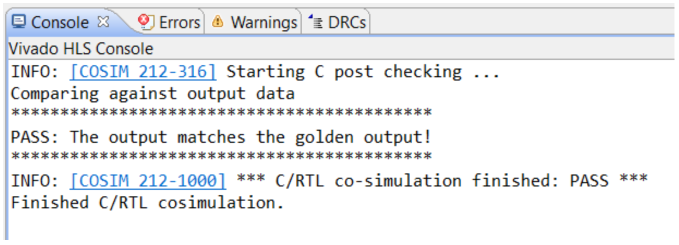
RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	fir	return value
ap_rst	in	1	ap_ctrl_hs	fir	return value
ap_start	in	1	ap_ctrl_hs	fir	return value
ap_done	out	1	ap_ctrl_hs	fir	return value
ap_idle	out	1	ap_ctrl_hs	fir	return value
ap_ready	out	1	ap_ctrl_hs	fir	return value
y	out	32	ap_vld	y	pointer
y_ap_vld	out	1	ap_vld	y	pointer
c_address0	out	4	ap_memory	c	array
c_ce0	out	1	ap_memory	c	array
c_q0	in	32	ap_memory	c	array
x	in	32	ap_none	x	scalar

This section shows the ports and I/O protocols created by the interface synthesis. We see that the fir object has a clock and reset port (ap_clk and ap_reset). There are also a number of control ports added by the synthesis (ap_start, ap_done, ap_idle and ap_ready). The output y is a 32-bit data port, and an associated output valid signal indicator (y_ap_vld) has been added. The input array of coefficients c has been implemented as a block RAM interface with a 4-bit output address port, an output CE port and a 32-bit input data port. The input argument x has been implemented as an input data port with no I/O protocol (ap_none).

Step 4: RTL Verification

You should always verify the RTL synthesis via simulation. This can be done using the C test bench.

1. From the **Solution** menu, choose **Run C/RTL CoSimulation**. Alternatively, click on the 'Run C/RTL CoSimulation' button in the toolbar. Click **OK** in the C/RTL Co-simulation dialog box to begin the RTL simulation.
2. When RTL co-simulation is complete, a report opens automatically in the Information panel. In the **Console**, you should see the output message of the C test bench stating that 'the output matches the golden output'. You will also see a message that says 'C/RTL co-simulation finished: Pass'. This message is generated from the return value of the test bench. Vivado HLS will indicate that the simulation passes if the test bench returns a value of 0. Hence, it is important in the design of the C test bench that it returns a value of 0 when the results are correct.

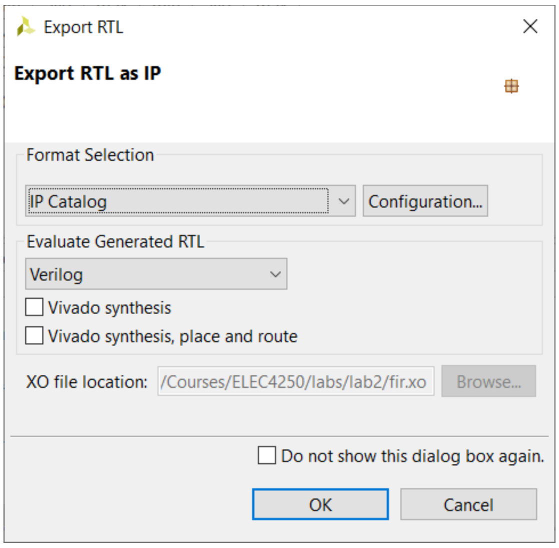


```
INFO: [COSIM 212-316] Starting C post checking ...
Comparing against output data
*******************************************
PASS: The output matches the golden output!
*******************************************
INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS ***
Finished C/RTL cosimulation.
```

Step 5: IP Creation

Packaging the design as an IP block allows the design to be used in other tools in the Vivado Design Suite. This is the last step in the HLS workflow.

1. From the **Solution** menu, choose **Export RTL**. Alternatively, click on the 'Export RTL button' in the toolbar. Make sure the **Format Selection** drop-down menu in the 'Export RTL as IP' dialog shows **IP Catalog**. Click OK.



Export RTL as IP

Format Selection: IP Catalog

Evaluate Generated RTL: Verilog

☐ Vivado synthesis

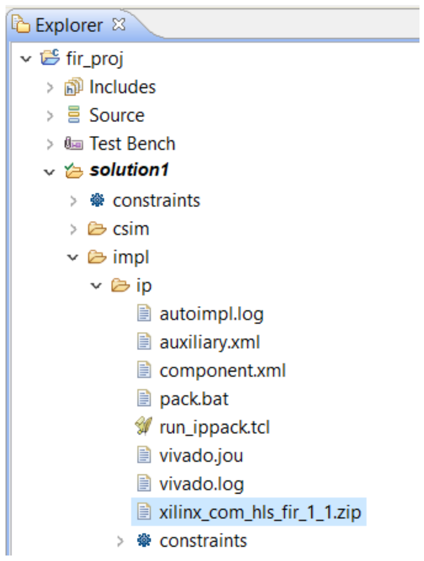
☐ Vivado synthesis, place and route

XO file location: /Courses/ELEC4250/labs/lab2/fir.xo

☐ Do not show this dialog box again.

If you run into issues creating IP, it is likely that your installation of Vivado HLS needs to be patched. Follow the instructions in the Vivado installation guide on iLearn to apply the patch.

2. When execution is complete, expand **solution1** in the Explorer Panel. Expand the **impl** folder, then **ip** folder. Within the ip folder, you should find the IP packaged as a. zip file. On the computer, this file is located in **/fir_prog/solution1/impl/ip/**. This file can be used in other Vivado Design Suite tools.

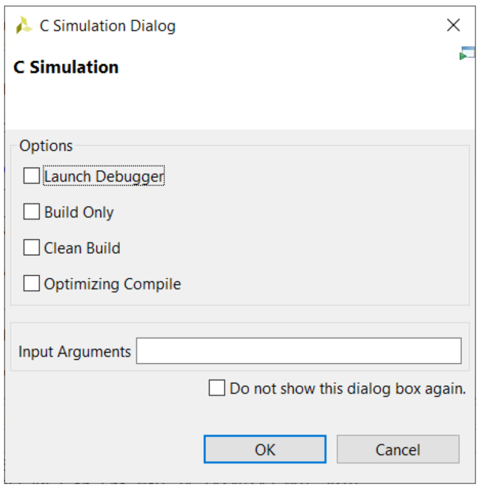


Explorer
fir_proj
Includes
Source
Test Bench
solution1
constraints
csim
impl
ip
autoimpl.log
auxiliary.xml
component.xml
pack.bat
run_ippack.tcl
vivado.jou
vivado.log
xilinx_com_hls_fir_1_1.zip
constraints

Code Debugging

A C debugger is included in Vivado HLS to help with debugging your code.

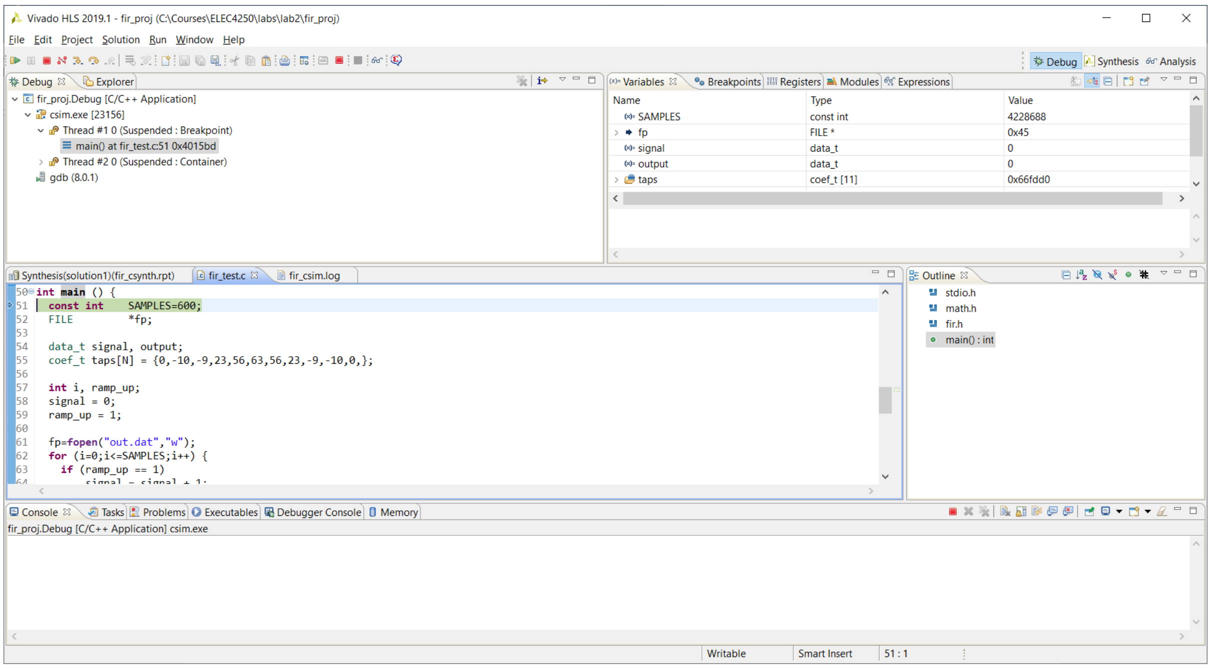
1. To run the debugger environment, click on **Run C Simulation** under the Project Menu.
2. Select the **Launch Debugger** option. Click **OK**.



C Simulation Dialog
C Simulation
Options
☐ Launch Debugger
☐ Build Only
☐ Clean Build
☐ Optimizing Compile
Input Arguments
☐ Do not show this dialog box again.
OK Cancel

3. After compilation of the code, the **Debug** perspective is opened. You can return to the Synthesis perspective by clicking on the corresponding button in the perspective section of the interface at any time.

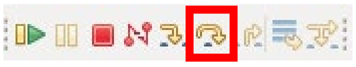
   You will see that one of the lines of code has been highlighted in green. This is the line of code that the debugger is up to.


Vivado HLS 2019.1 - fir_proj (C:\Courses\ELEC4250\labs\lab2\fir_proj)
File Edit Project Solution Run Window Help
Debug Synthesis Analysis
Debug Explorer
fir_proj.Debug [C/C++ Application]
csim.exe [23156]
Thread #1 0 (Suspended : Breakpoint)
main() at fir_test.c:51 0x4015bd
Thread #2 0 (Suspended : Container)
gdb (8.0.1)
Variables Breakpoints Registers Modules Expressions
Synthesis(solution1)(fir_csynth.rpt) fir_test.c fir_csim.log
Outline
stdio.h
math.h
fir.h
main() : int
Console Tasks Problems Executables Debugger Console Memory
fir_proj.Debug [C/C++ Application] csim.exe
Writable Smart Insert 51 : 1

Name	Type	Value
SAMPLES	const int	4228688
fp	FILE *	0x45
signal	data_t	0
output	data_t	0
taps	coef_t [11]	0x66fdd0

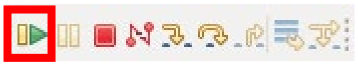
```
50 int main () {
51   const int     SAMPLES=600;
52   FILE          *fp;
53
54   data_t signal, output;
55   coef_t taps[N] = {0,-10,-9,23,56,63,56,23,-9,-10,0,};
56
57   int i, ramp_up;
58   signal = 0;
59   ramp_up = 1;
60
61   fp=fopen("out.dat","w");
62   for (i=0;i<=SAMPLES;i++) {
63     if (ramp_up == 1)
```

4. If you click on the **Step Over** button, or press F6, a few times. You should see the green line advance down the lines of code.

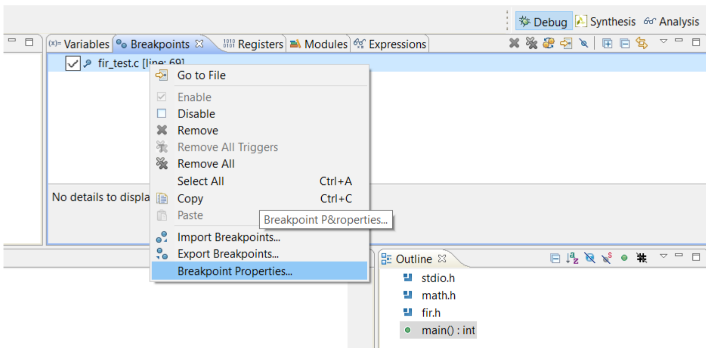


There is also a **Step Into** button in the toolbar (or F5). Using this button allows you to *step into* the code of any function that is being called.

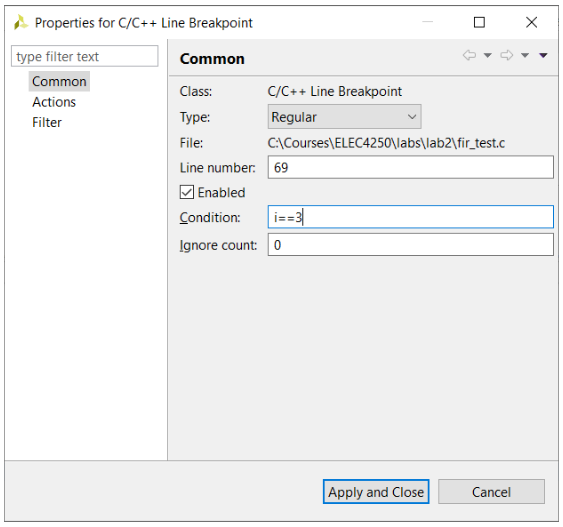
5. Scroll to **line 69** in **fir_test.c**. Right-click in the pink margin left of 69 and select **Toggle Breakpoint**. You should see a blue mark appear in the margin on line 69 to indicate a breakpoint.
6. Click on the **Resume** button in the toolbar, or press F8.



7. You should see the debugger has now run up to the line of code on line 69 and stopped.
8. You can also set a conditional breakpoint. A conditional breakpoint allows the debugger to keep running the code until some condition is met before it stops on the line with the conditional breakpoint. Open the Breakpoint tab in the top-right panel. You will also see the breakpoint you have just inserted here. Right-click on it and select Breakpoint Properties.



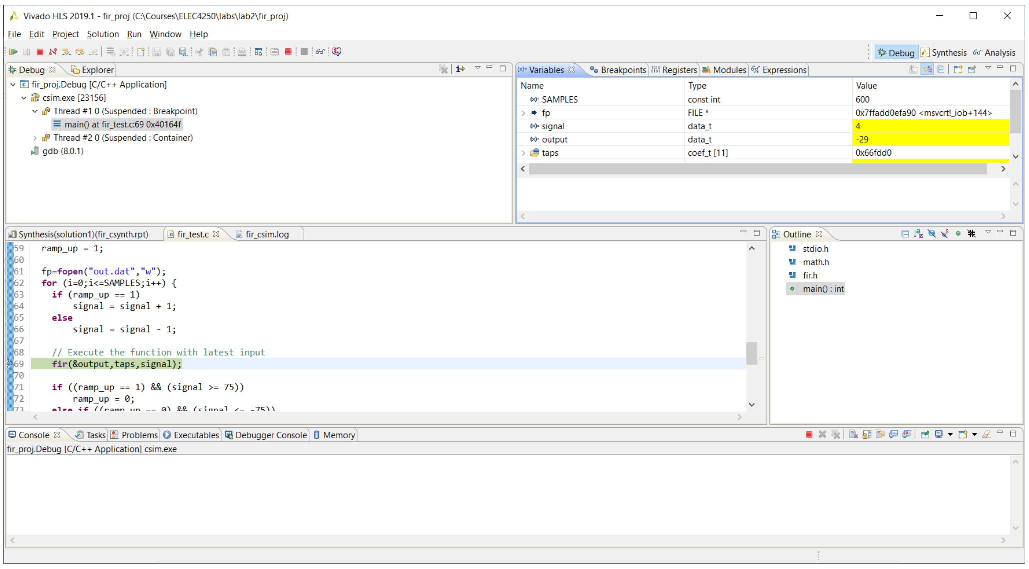
9. Click on **Common** in the left menu. Add **‘i==3’** as the **Condition**. This condition means that the debugger will only stop at this breakpoint when the variable i is equal to 3. Click **Apply and Close**.



Common

Class:	C/C++ Line Breakpoint
Type:	Regular
File:	C:\Courses\ELEC4250\labs\lab2\fir_test.c
Line number:	69
☑ Enabled	
Condition:	i==3
Ignore count:	0

10. Go back to the **Variables** tab. This is where you can observe the state of each of the variables. Note the value of the variable i. It should be 0.
11. Click on the **Resume** button in the toolbar, or press F8. You should see that the variable i has increased to 3, which has caused the debugger to stop the execution of the code.



```
fir_proj.Debug [C/C++ Application]
  csim.exe [23156]
    Thread #1 0 (Suspended : Breakpoint)
      main() at fir_test.c:69 0x40164f
    Thread #2 0 (Suspended : Container)
  gdb (8.0.1)
```

Name	Type	Value
SAMPLES	const int	600
fp	FILE *	0x7ffadd0efa90 <msvcrt!_iob+144>
signal	data_t	4
output	data_t	-29
taps	coef_t [11]	0x66fdd0

```
  ramp_up = 1;

  fp=fopen("out.dat","w");
  for (i=0;i<=SAMPLES;i++) {
    if (ramp_up == 1)
        signal = signal + 1;
    else
        signal = signal - 1;

    // Execute the function with latest input
    fir(&output,taps,signal);

    if ((ramp_up == 1) && (signal >= 75))
        ramp_up = 0;
```

```
fir_proj.Debug [C/C++ Application] csim.exe
```

12. Click the red **Terminate** (square) button in the toolbar to end the debug session. If you need to restart the debug session, click on the Run C simulation button.
13. You can return to the Synthesis perspective by clicking Synthesis in the top right hand corner.

Exercise

Using the matrix addition code you wrote in Lab 1, create an HLS project and synthesize an IP that can perform a 4x4 matrix addition. Be sure to create an appropriate test bench that can be used to test the C code and perform RTL verification.

Acknowledgement

This lab was adapted from UG871 Vivado Design Suite Tutorial: High-Level Synthesis for this unit of study.