

Block-Level Protocols

2022.1

Abstract

This lab explains block-level protocols and how they control the RTL block.

This lab should take approximately 30 minutes.

CloudShare Users Only

You are provided three attempts to access a lab, and the time allotted to complete each lab is 2X the time expected to complete the lab. Once the timer starts, you cannot pause the timer. Also, each lab attempt will reset the previous attempt—that is, your work from a previous attempt is not saved.

Objectives

After completing this lab, you will be able to:

- Create a new project in the Vitis™ HLS tool GUI
- Modify the default block-level protocols

Introduction

The Vitis HLS tool uses the interface types `ap_ctrl_none`, `ap_ctrl_hs`, and `ap_ctrl_chain` to specify whether the RTL is implemented with block-level handshake signals.

Block-level handshake signals specify the following:

- When the design can start to perform the operation
- When the operation ends
- When the design is idle and ready for new inputs

You can specify these block-level protocols on the function or the function return. If the C code does not return a value, you can still specify the block-level protocol on the function return. If the C code uses a function return, the Vitis HLS tool creates an output port `ap_return` for the return value.

Note that the design used in this lab is a simple design and can be implemented as a combinational implementation that uses a single clock cycle. Since the focus of the lab is to show the different block-level interfaces generated by the tool, the design is given more strict constraints to force inferring a multicycle implementation. Ignore the timing violation caused because of this and observe the block-level interfaces that are generated.

	Argument Type	Scalar		Array			Pointer or Reference			HLS:: Stream
		Input	Return	I	I/O	O	I	I/O	O	
Block-Level Protocol	ap_ctrl_none									
	ap_ctrl_hs		D							
	ap_ctrl_chain									
AXI Interface Protocol	axis									
	s_axilite									
	m_axi									
No I/O Protocol	ap_none	D					D			
	ap_stable									
Wire handshake Protocol	ap_ack									
	ap_vld								D	
	ap_ovld							D		
	ap_hs									
Memory Interface Protocol: RAM : FIFO	ap_memory			D	D	D				
	bram									
	ap_fifo									D



Supported D = Default Interface



Not Supported

Figure 3-1: Block-Level Protocols

Understanding the Lab Environment

The labs and demos provided in this course are designed to run on a Linux platform.

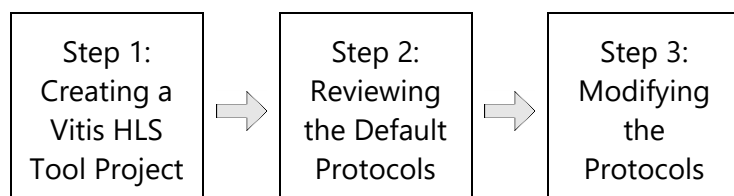
One environment variable is required: `TRAINING_PATH`, which points to where the lab files are located. This variable comes configured in the CloudShare/CustEd_VM environments.

Some tools can use this environment variable directly (that is, `$TRAINING_PATH` will be expanded), and some tools require manual expansion (`/home/xilinx/training` for the CloudShare/CustEd_VM environments). The lab instructions describe what to do for each tool.

Both the Vivado Design Suite and the Vitis platform offer a Tcl environment that is used by many labs. When the tool is launched, it starts with a clean Tcl environment with none of the procs or variables remaining from a previous launch of the tools.

This means that if you sourced a Tcl script or manually set any Tcl variables and you closed the tool, then when you reopen the tool, you will need to source the Tcl script again and set any variables that the lab requires. This is also true of terminal windows—any variable settings will be cleared when a new terminal opens.

General Flow



Creating a Vitis HLS Tool Project

Step 1

In this step, you will create a new Vitis HLS tool project, add source files, and provide solution settings for the default solution using the Vitis HLS tool command prompt.

1-1. Launch the Vitis HLS tool command prompt and change the directory to the working directory.

1-1-1. [Windows 10 users]: Select **Start > Xilinx Design Tools > Vitis HLS 2022.1 Command Prompt** to launch the Vitis HLS tool command prompt.

[Linux users]: Open a new terminal (press <Ctrl + Alt + T>) and source the `source /opt/Xilinx/Vitis/2022.1/settings64.sh` file to set up the Vitis HLS tool environment.

Note: The customer training environment (CustEd_VM) sets the Vitis tool install path to `/opt/Xilinx/Vitis`. If the tool is installed in a different location in your environment, use that install path.

[Linux users]: Enter `vitis_hls -i` in the terminal to launch the Vitis HLS command line interface.

1-1-2. Enter the following command to change the path to the lab directory:

1-1-3. `cd $::env(TRAINING_PATH)/block_level_protocol/lab`

1-2. Review and run the `run_adder.tcl` file to create the project and open the created project in the Vitis HLS tool GUI.

1-2-1. Open the `run_adder.tcl` file using any of the text editors available in the following location:

```
$TRAINING_PATH/block_level_protocol/lab
```

1-2-2. Review the following sections:

- Project name (`adders_prj`)
- Source files (`adders.c`) and test bench (`adders_test.c`) added to the project
- Specifying the top-level function for synthesis (`adders`)
- Creating a solution (`solution1`)
- Selecting the Xilinx device (`xczu7ev-ffvc1156-2-e`) and clock period (`0.5`)
- Running the C simulation

1-2-3. Close the `run_adder.tcl` file.

1-2-4. Enter the following command in the Vitis HLS tool command prompt to run the `run_adder.tcl` file:

```
vitis_hls -f run_adder.tcl
```

```
File Edit View Search Terminal Help
INFO: [HLS 200-10] On os Ubuntu 18.04.4 LTS
INFO: [HLS 200-10] In directory '/'
INFO: [HLS 200-10] Sourcing Tcl script 'run_adder.tcl'
INFO: [HLS 200-1510] Running: open_project -reset adders_prj
INFO: [HLS 200-10] Creating and opening project '/'
INFO: [HLS 200-1510] Running: add_files adders.c
INFO: [HLS 200-10] Adding design file 'adders.c' to the project
INFO: [HLS 200-1510] Running: add_files -tb adders_test.c
INFO: [HLS 200-10] Adding test bench file 'adders_test.c' to the project
INFO: [HLS 200-1510] Running: set_top adders
INFO: [HLS 200-1510] Running: open_solution -reset solution1
INFO: [HLS 200-10] Creating and opening solution '/'
INFO: [HLS 200-10] Cleaning up the solution database.
WARNING: [HLS 200-40] No /
INFO: [HLS 200-1505] Using default flow_target 'vivado'
Resolution: For help on HLS 200-1505 see www.xilinx.com/cgi-bin/docs/rdoc?v=2020.2;t=hls+guidance;d=200-1505.html
INFO: [HLS 200-1510] Running: set_part xczu7ev-ffvc1156-2-e
INFO: [HLS 200-10] Setting target device to 'xczu7ev-ffvc1156-2-e'
INFO: [HLS 200-1510] Running: create_clock -period 0.5
INFO: [SYN 201-201] Setting up clock 'default' with a period of 0.5ns.
INFO: [HLS 200-1510] Running: csim_design
INFO: [SIM 211-2] ***** CSIM start *****
INFO: [SIM 211-4] CSIM will launch GCC as the compiler.
Compiling(apcc) ../../../../adders_test.c in debug mode
INFO: [HLS 200-10] Running: /
INFO: [HLS 200-10] For user 'xilinx' on host 'xilinx' (Linux_x86_64 version 5.3.0-28-generic) on
INFO: [HLS 200-10] On os Ubuntu 18.04.4 LTS
INFO: [HLS 200-10] In directory '/'
INFO: [HLS 200-10] Tmp directory is /tmp/apcc_db_xilinx/
INFO: [APCC 202-1] APCC is done.
Compiling(apcc) ../../../../adders.c in debug mode
INFO: [HLS 200-10] Running: /
INFO: [HLS 200-10] For user 'xilinx' on host 'xilinx' (Linux_x86_64 version 5.3.0-28-generic) on
INFO: [HLS 200-10] On os Ubuntu 18.04.4 LTS
INFO: [HLS 200-10] In directory '/'
INFO: [HLS 200-10] Tmp directory is /tmp/apcc_db_xilinx/
INFO: [APCC 202-1] APCC is done.
Generating csim.exe
10+20+30=60
20+30+40=90
30+40+50=120
40+50+60=150
50+60+70=180
*****Post*****
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] ***** CSIM finish *****
INFO: [HLS 200-111] Finished Command csim_design CPU user time: 13.63 seconds. CPU system time: 0.45 seconds. Elapsed time: 13.62 seconds; current allocated memory: 179.883 MB.
vitis_hls
```

Figure 3-2: Viewing the C Simulation Results

Note: Observe that the C simulation passed.

1-2-5. Enter the following command to open the Vitis HLS tool project:

```
vitis_hls -p adders_prj
```

The Vitis HLS tool GUI opens.

Reviewing the Default Block-Level Protocols

Step 2

2-1. Open the source file and review it.

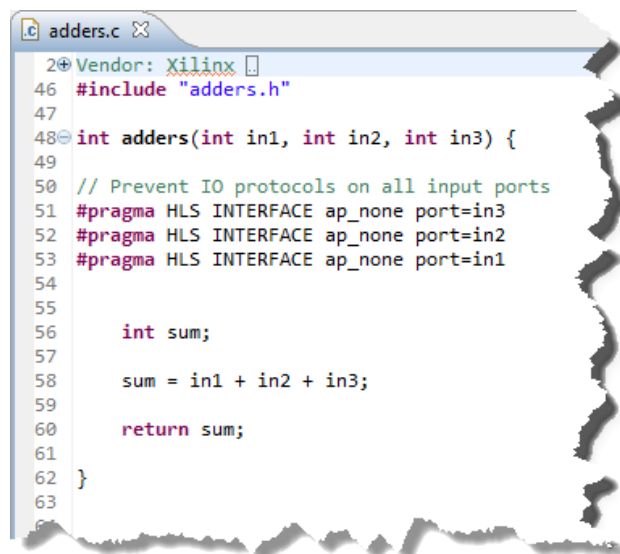
2-1-1. Expand **adders_prj** > **Source** in the Explorer pane.

2-1-2. Double-click the **adders.c** file to open it and review the design.

This example uses a simple design to focus on the I/O implementation (and not the logic in the design).

The important points to take from this code are:

- Directives in the form of pragmas have been added to the source code to prevent any I/O protocol being synthesized for any of the data ports (in1, in2, and in3). I/O port protocols are reviewed in the "Port-Level Protocols" lab exercise.
- This function returns a value, and this is the only output from the function. The port created for the function return is discussed in this lab exercise.




```
adders.c
2 Vendor: Xilinx
46 #include "adders.h"
47
48 int adders(int in1, int in2, int in3) {
49
50 // Prevent IO protocols on all input ports
51 #pragma HLS INTERFACE ap_none port=in3
52 #pragma HLS INTERFACE ap_none port=in2
53 #pragma HLS INTERFACE ap_none port=in1
54
55
56     int sum;
57
58     sum = in1 + in2 + in3;
59
60     return sum;
61 }
62
63
```

Figure 3-3: C Code with the Directives in the Form of Pragmas

2-2. Synthesize the design.

2-2-1. Use one of the following methods to launch the C synthesis tool:

- Click the **Run Flow** icon () in the toolbar (1).
- Click the pull-down next to the Run Flow icon in the toolbar and select **C Synthesis** (2).
- From the Explorer tab, select the desired solution to synthesize and right-click and select **C Synthesis** > **Active Solution** (3).

Note: More complex projects allow multiple solutions to be batched which, is what the other options in this menu support.

- From Flow Navigator, select **C Synthesis** > **Run C Synthesis** (4).

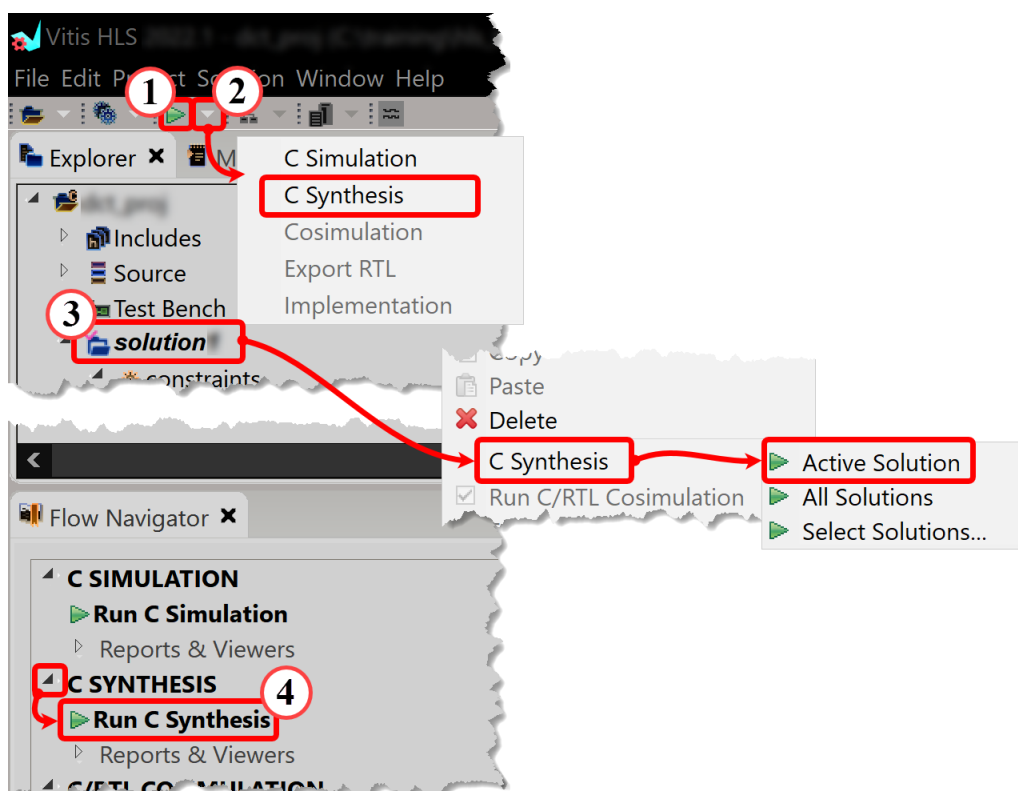


Figure 3-4: Multiple Methods for Launching C Synthesis

The C Synthesis - Active Solution dialog box opens.

The tool automatically fills in any information it already knows, such as the project's clock period, part selection, and flow type. You can modify these here if needed.

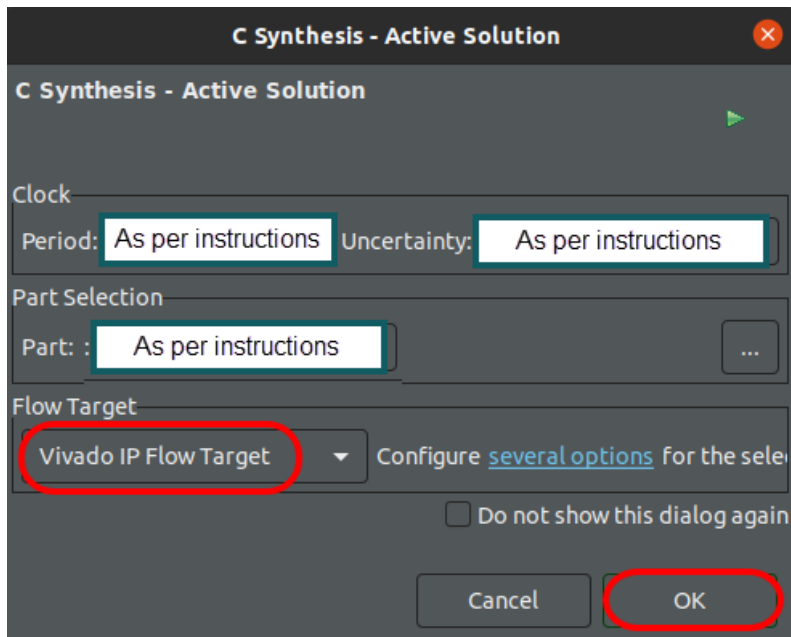


Figure 3-5: C Synthesis Dialog Box

Click **OK** to begin the synthesis process.

When synthesis completes, the Synthesis report will be displayed in the Information pane. You can ignore any timing violations as the focus of the lab is to view the block-level protocols and how they are generated.

- 2-2-2. Expand the **adders_prj** > **solution1** > **syn** > **report** folder in the Explorer pane.
- 2-2-3. Double-click the **adders_csynth.rpt** file to view the Synthesis report.

2-2-4. Review the **Performance Estimates** and **Utilization Estimates** in the Synthesis report.

2-2-5. Scroll down to the **Interface > Summary** section in the Synthesis report.

The report also shows the top-level interface signals generated by the tools.

Block-Level Protocols

Interface						
Summary						
RTL Ports	Dir	Bits	Protocol	Source Object	C Type	
ap_local_block	out	1	ap_ctrl_hs	adders	return value	
ap_local_deadlock	out	1	ap_ctrl_hs	adders	return value	
ap_clk	in	1	ap_ctrl_hs	adders	return value	
ap_rst	in	1	ap_ctrl_hs	adders	return value	
ap_start	in	1	ap_ctrl_hs	adders	return value	
ap_done	out	1	ap_ctrl_hs	adders	return value	
ap_idle	out	1	ap_ctrl_hs	adders	return value	
ap_ready	out	1	ap_ctrl_hs	adders	return value	
ap_return	out	32	ap_ctrl_hs	adders	return value	
in1	in	32	ap_none	in1	scalar	
in2	in	32	ap_none	in2	scalar	
in3	in	32	ap_none	in3	scalar	

Figure 3-6: Generated Interface Signals

- The design takes more than one clock cycle to compute, so a clock and reset have been added to the design: `ap_clk` and `ap_rst`. Both are single-bit inputs.
- A block-level protocol has been added to control the RTL design. The ports are `ap_start`, `ap_done`, `ap_idle`, and `ap_ready`.
- The design has four data ports.
 - Input ports: `in1`, `in2`, and `in3` are 32-bit inputs and have the I/O protocol `ap_none` as specified by the directives (shown in the above picture in Green color box).
 - The design has a 32-bit output port for the function return `ap_return`.
 - Apart from these, there are two other signals (`ap_local_block` and `ap_local_deadlock`) which should not be generated by default and are used for system deadlock detection. Note that this is a tool issue and will be fixed in the next release. You can ignore these signals.

The block-level protocol allows the RTL design to be controlled by additional ports independently of the data I/O ports. This protocol is associated with the function itself, not with any of the data ports. The default block-level protocol is called `ap_ctrl_hs`.

The table below summarizes the behavior of the signals for block-level protocol `ap_ctrl_hs`.

Signals	Description
<code>ap_start</code>	<p>This signal controls the block execution and must be asserted to logic 1 for the design to begin operation. It should be held at logic 1 until the associated output handshake <code>ap_ready</code> is asserted. When <code>ap_ready</code> goes high, the decision can be made on whether to keep <code>ap_start</code> asserted and perform another transaction or set <code>ap_start</code> to logic 0 and allow the design to halt at the end of the current transaction. If <code>ap_start</code> is asserted low before <code>ap_ready</code> is high, the design might not have read all input ports and might stall operation on the next input read.</p>
<code>ap_ready</code>	<p>This output signal indicates when the design is ready for new inputs. The <code>ap_ready</code> signal is set to logic 1 when the design is ready to accept new inputs, indicating that all input reads for this transaction have been completed.</p> <p>If the design has no pipelined operations, new reads are not performed until the next transaction starts. This signal is used to make a decision on when to apply new values to the inputs ports and whether to start a new transaction should using the <code>ap_start</code> input signal.</p> <p>If the <code>ap_start</code> signal is not asserted high, this signal goes low when the design completes all operations in the current transaction.</p>
<code>ap_done</code>	<p>This signal indicates when the design has completed all operations in the current transaction (transaction - equivalent to one execution of the C function).</p> <p>A logic 1 on this output indicates the design has completed all operations in this transaction. Because this is the end of the transaction, a logic 1 on this signal also indicates the data on the <code>ap_return</code> port is valid.</p> <p>Not all functions have a function return argument and hence not all RTL designs have an <code>ap_return</code> port.</p>
<code>ap_idle</code>	<p>This signal indicates if the design is operating or idle (no operation). The idle state is indicated by logic 1 on this output port. This signal is asserted low once the design starts operating.</p> <p>This signal is asserted high when the design completes operation and no further operations are performed.</p>

2-3. Perform C/RTL cosimulation.

2-3-1. Select **Solution** > **Run C/RTL Cosimulation**.

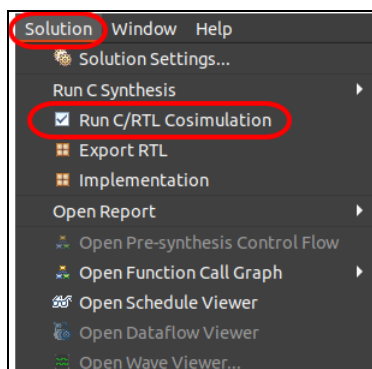


Figure 3-7: Launching from the Menu

You can also select **Run Cosimulation** under C/RTL COSIMULATION section from the Flow Navigator at the bottom.

The Run C/RTL Co-simulation dialog box opens.

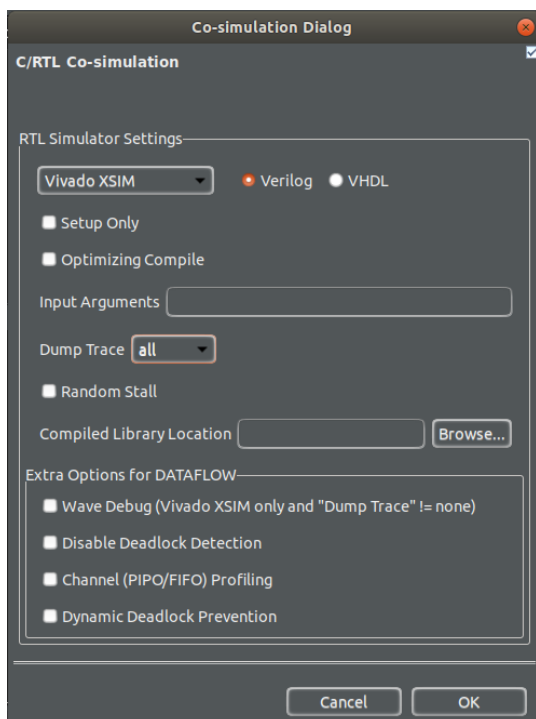


Figure 3-8: Co-simulation Dialog Box

The dialog box includes the following settings:

- Simulator: Choose from one of the supported HDL simulators in the Vivado Design Suite. The Vivado simulator is the default simulator.
- Language: Specify the use of Verilog or VHDL as the output language for simulation.

- Setup Only: Create the required simulation files, but do not run the simulation. The simulation executable can be run from a command shell at a later time.
- Optimizing Compile: Enable optimization to improve the runtime performance, if possible, at the expense of compilation time.
- Input Arguments: Specify any command-line arguments to the C test bench.
- Dump Trace: Specifies the level of trace file output written to the *sim/Verilog* or *sim/VHDL* directory of the current solution when the simulation executes. Options include:
 - all: Output all port and signal waveform data being saved to the trace file.
 - port: Output waveform trace data for the top-level ports only.
 - none: Do not output trace data.
- Random Stall: Applies a randomized stall for each data transmission.

2-3-2. Select **all** from the *Dump Trace* drop-down menu.

2-3-3. Click **OK**.

When RTL verification completes, the co-simulation report opens automatically. The report indicates if the simulation passed or failed. In addition, the report indicates the measured latency and interval.

Because the Dump Trace option was used with the Vivado simulator option, two trace files are now present in the Verilog simulation directory (Verilog was selected as an RTL option).

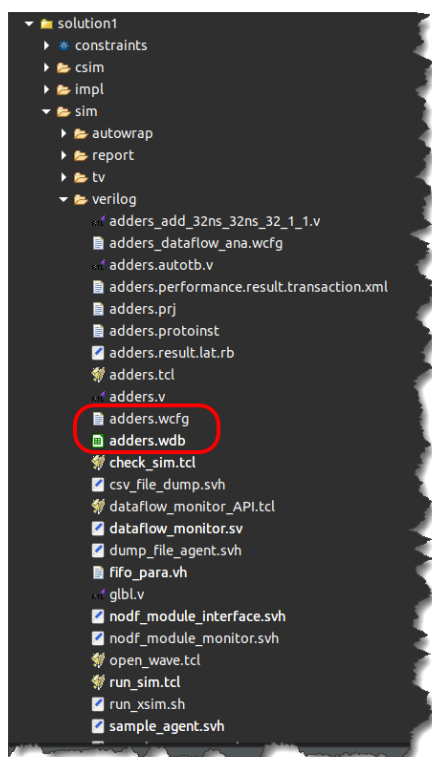


Figure 3-9: Verilog Vivado Simulator Co-simulation Results

2-4. View the trace files that have been generated for the Vivado simulator.

2-4-1. Click the **Open Wave Viewer** () icon from the toolbar.

Alternatively, select **Solution > Open Wave Viewer**.

This will open the Vivado IDE with the RTL waveforms traces.

2-4-2. Expand **Design Top Signals** and **Block-level IO Handshake** as shown below.

You can change the radix of the highlighted signals (shown below) to unsigned decimal by right-clicking the signal.

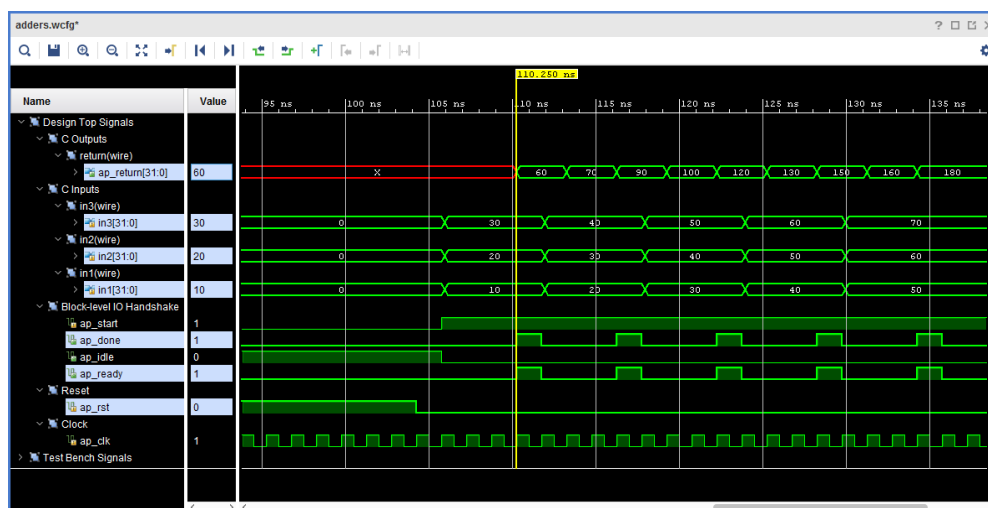


Figure 3-10: Analyzing the RTL Trace File

Note: Expand the signals in order to see the waveforms. Select the `in1`, `in2`, `in3`, and `ap_return` signals, right-click, and change the radix to unsigned decimal.

The waveform above shows the behavior of the block-level signals:

- The design does not start operation until `ap_start` is set to logic 1.
- The design indicates it is no longer idle by setting port `ap_idle` low.
- Five transactions are shown. The first three input values are 10, 20, and 30 and are applied to the input ports `in1`, `in2`, and `in3`, respectively.
- The output signal `ap_ready` goes high to indicate the design is ready for new inputs on the next clock cycle.
- The output signal `ap_done` indicates when the design is finished and that the value on the output port `ap_return` is valid (the first output value, 60, is the sum of all three inputs).
- Because `ap_start` is held high, the next transaction starts on the next clock cycle.

In the second transaction, notice on port `ap_return` that the first output has the value 70. The result on this port is not valid until the `ap_done` signal is asserted high.

2-4-3. Select **File > Exit** to close the Vivado IDE GUI.

Modifying the Block-Level Protocols

Step 3

The default block-level protocol is the `ap_ctrl_hs` protocol (the Control Handshake protocol). In this step, you will create a new solution and modify this protocol.

3-1. Create a new solution by copying the previous solution (solution1) settings.

3-1-1. Select **Project > New Solution**.

3-1-2. Leave the options at their default settings.

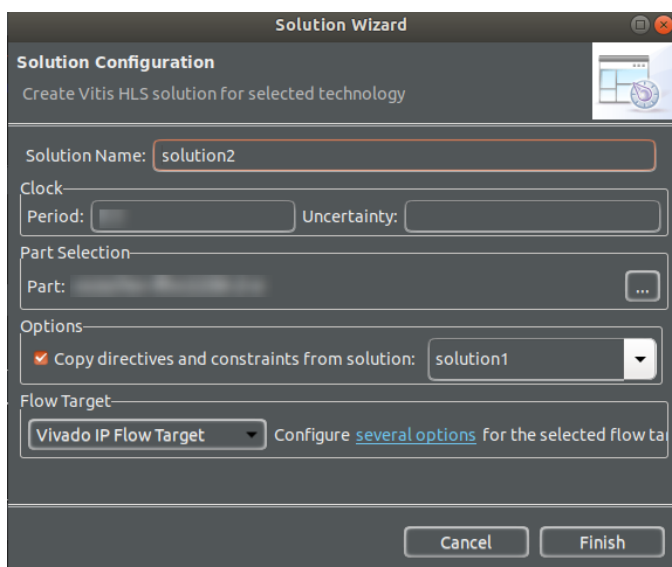


Figure 3-11: Creating a New Solution (solution2)

3-1-3. Click **Finish**.

3-2. Apply the directive `ap_ctrl_none` to make no block-level control protocol from the default `ap_ctrl_hs`.

- 3-2-1. Select **Project > Close Inactive Solution Tabs** to close all inactive solution windows.
- 3-2-2. Ensure that the **adders.c** file is open and active in the Information pane.
- 3-2-3. Right-click the **adders** function in the Directive tab and select **Insert Directive**.
- 3-2-4. Select **INTERFACE** from the Directive drop-down list.
- 3-2-5. Select **Source File** in the Destination section.
- 3-2-6. Select the *mode (optional)* as **ap_ctrl_none**.

By default, directives are placed in the `directives.tcl` file. In this example, the directive is placed in the source file with the existing I/O directives.

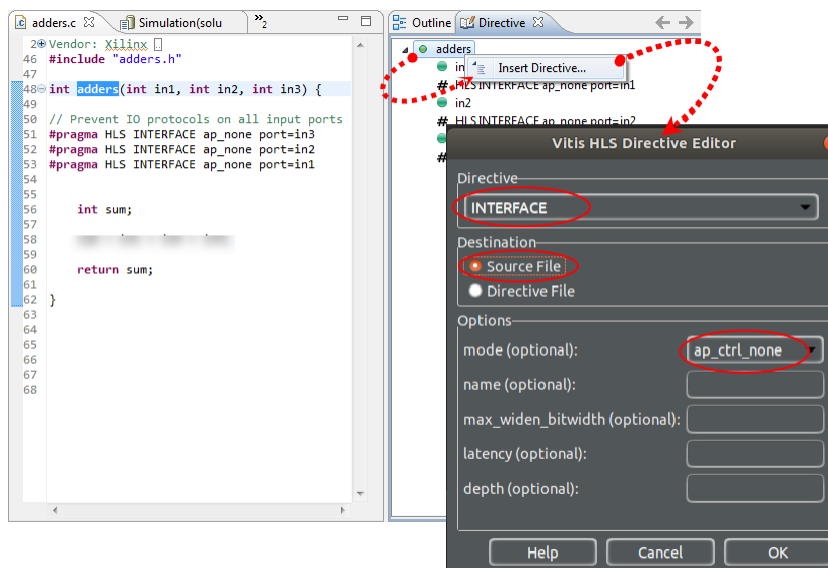


Figure 3-12: Applying the Interface Directive - `ap_ctrl_none`

The drop-down menu shows that there are four options for the block-level interface protocol:

- `ap_ctrl_none`: No block-level control protocol.
- `ap_ctrl_hs`: The block-level control handshake protocol we have reviewed.
- `ap_ctrl_chain`: The block-level protocol for control chaining. This protocol is primarily used for chaining pipelined blocks together.
- `s_axilite`: May be applied in addition to `ap_ctrl_hs` or `ap_ctrl_chain` to implement the block-level protocol as an AXI Slave Lite interface in place of separate discrete I/O ports.

- 3-2-7. Click **OK**.

Notice the source file now has a new directive highlighted in both the source code and directives tab.

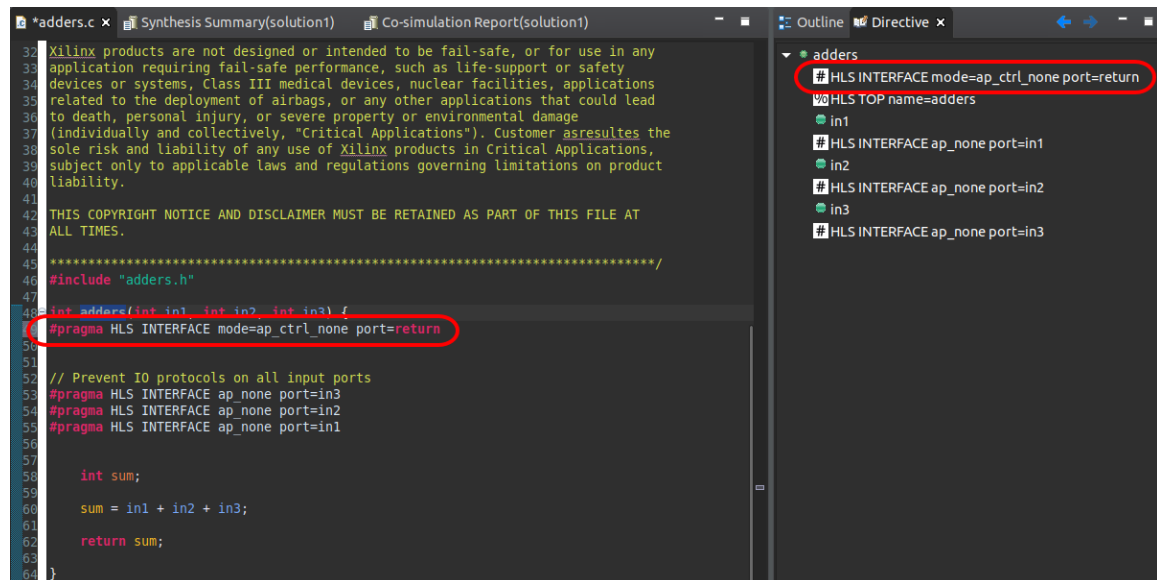



Figure 3-13: Block-Level Interface Directive `ap_ctrl_none`

3-2-8. Save the source file `adders.c`.

3-3. Synthesize the design.

3-3-1. Use one of the following methods to launch the C synthesis tool:

- Click the **Run Flow** icon () in the toolbar (1).
- Click the pull-down next to the Run Flow icon in the toolbar and select **C Synthesis** (2).
- From the Explorer tab, select the desired solution to synthesize and right-click and select **C Synthesis** > **Active Solution** (3).

Note: More complex projects allow multiple solutions to be batched which, is what the other options in this menu support.

- From Flow Navigator, select **C Synthesis** > **Run C Synthesis** (4).

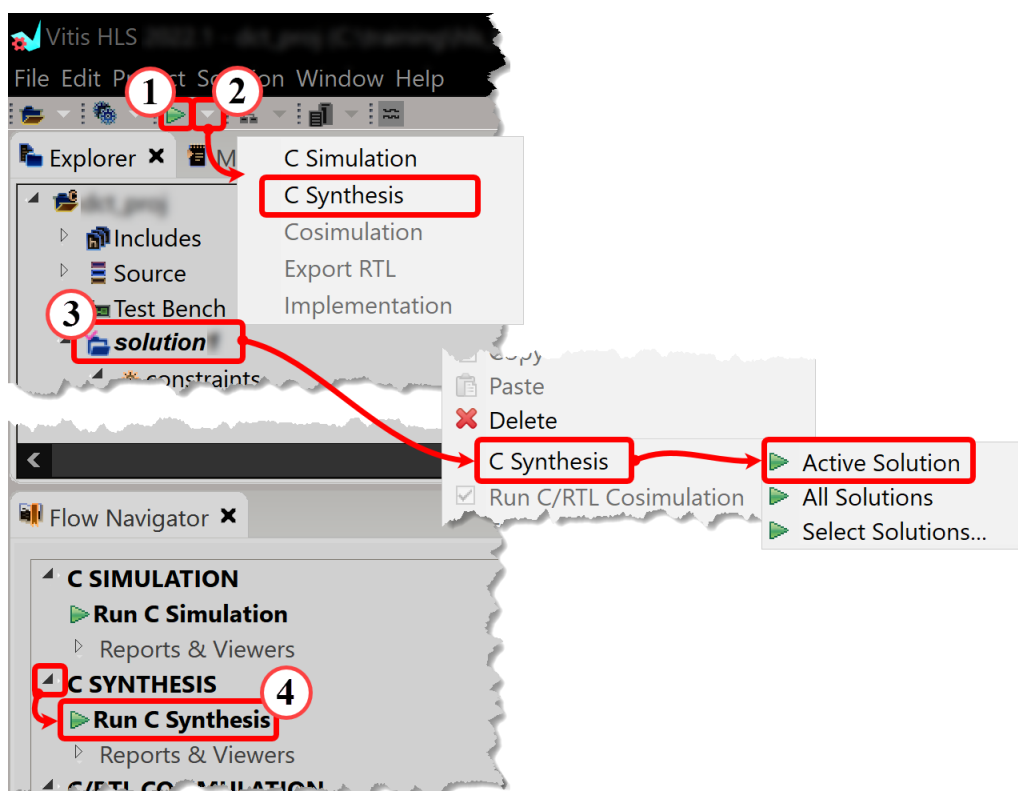


Figure 3-14: Multiple Methods for Launching C Synthesis

The C Synthesis - Active Solution dialog box opens.

The tool automatically fills in any information it already knows, such as the project's clock period, part selection, and flow type. You can modify these here if needed.

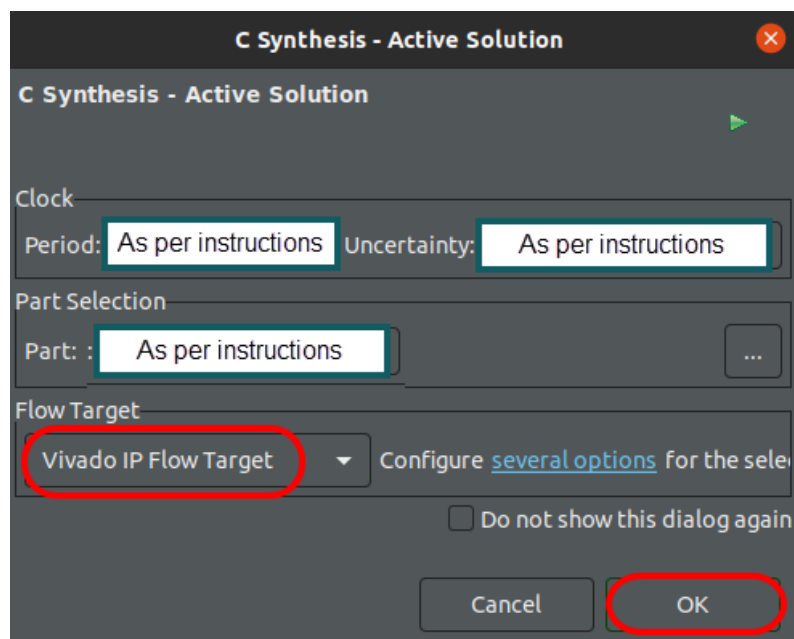


Figure 3-15: C Synthesis Dialog Box

- 3-3-2. Click **OK** to begin the synthesis process.
- 3-3-3. Expand the **adders_prj > solution2 > syn > report** folder in the Explorer pane.
- 3-3-4. Double-click the **adders_csynth.rpt** file to view the Synthesis report.
- 3-3-5. Scroll down to the **Interface > Summary** section in the Synthesis report.

The report also shows the top-level interface signals generated by the tools.

Interface					
Summary					
RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_none	adders	return value
ap_rst	in	1	ap_ctrl_none	adders	return value
ap_return	out	32	ap_ctrl_none	adders	return value
in1	in	32	ap_none	in1	scalar
in2	in	32	ap_none	in2	scalar
in3	in	32	ap_none	in3	scalar

Figure 3-16: Viewing the Interface Summary Report

When the interface protocol `ap_ctrl_none` is used, no block-level protocols are added to the design.

In addition, the RTL cosimulation feature requires a block-level protocol to sequence the test bench and RTL design for co-simulation automatically. Any attempt to use RTL co-simulation results in the following error message and RTL co-simulation with halt:

```
ERROR [COSIM 212-345] Cosim only supports the following
'ap_ctrl_none' designs: (1) combinational designs; (2) pipelined
design with task interval of 1; (3) designs with array streaming or
hls_stream or AXI4 stream ports.

ERROR [COSIM 212-4] *** C/RTL co-simulation finished: FAIL ***
```

3-3-6. Select **File > Exit** to close the Vitis HLS tool.

3-3-7. Close the Vitis HLS tool command prompt.

Some systems (particularly VMs) may be memory constrained. Removing the workspace frees a portion of the disk space, allowing other labs to be performed.

You can delete the directory containing the lab you just ran by using the graphical interface or the command-line interface. You can choose either mechanism. Both processes will recursively delete all the files in the `$TRAINING_PATH/block_level_protocol` directory.

3-4. [Optional] [Only for local VMs—not for CloudShare] Clean up the file system.

Using the GUI:

3-4-1. Using the graphical browser (Windows: press the **<Windows>** key + **<E>**; Linux: press **<Ctrl + N>**), navigate to `$TRAINING_PATH/block_level_protocol`.

3-4-2. Select **block_level_protocol**.

3-4-3. Press **<Delete>**.

-- OR --

Using the command line:

3-4-4. Open a terminal window (Windows: press the **<Windows>** key + **<R>**, then enter **cmd**; Linux: press **<Ctrl + Alt + T>**).

3-4-5. Enter the following command to delete the contents of the workspace:

[Windows users]: `rd /s /q $TRAINING_PATH/block_level_protocol`

[Linux users]: `rm -rf $TRAINING_PATH/block_level_protocol`

Summary

You have learned what block-level protocol is and have used the `INTERFACE` directive to change the block-level protocol.