

Block-Level I/O Protocols

2020.1

Abstract

This lab explains block-level I/O protocols and how to control them.

Objectives

After completing this lab, you will be able to:

- Modify the default block-level I/O Protocol

Introduction

The Vivado HLS tool uses the interface types `ap_ctrl_none`, `ap_ctrl_hs`, and `ap_ctrl_chain`.

The `ap_ctrl_hs` interface is used 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

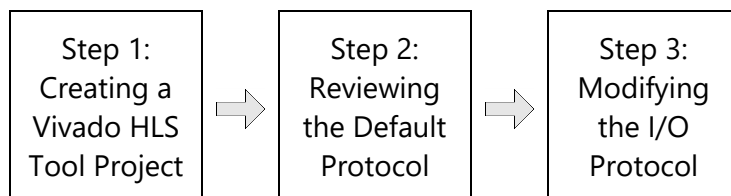
You can specify these block-level I/O protocols on the function or the function return. If the C code uses a function return, the Vivado HLS tool creates an output port `ap_return` for the return value.

	Argument Type	Scalar		Array			Pointer or Reference			HLS:: Stream
	Interface Mode	Input	Return	I	I/O	O	I	I/O	O	I and O
Block-Level Protocol	ap_ctrl_none									
	ap_ctrl_hs		D							
	ap_ctrl_chain									
AXI Interface Protocol	axis									
	s_axilite									
	m_axi									
No IO 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
Bus Protocol	ap_bus									

Supported D = Default Interface
 Not Supported

Figure 1: Block-Level Protocol

General Flow



Before starting the lab

Step 0

Browse to the `C:\Xilinx_trn\HLS_2020\lab3_block_level_protocol\source` directory using Windows Open, with your preferred text editor, and explore the C code to understand algorithm and data dependencies.

0-1. Open lab3.h file and review it.

```
1  #ifndef LAB3_H_
2  #define LAB3_H_
3
4  typedef short data_sc;
5
6  data_sc adders(data_sc in1, data_sc in2, data_sc in3);
7
8  #endif
```

Figure 2: Lab3 definition file

New data type `data_s` is defined in the file. The data type will be used in source code and testbench.

0-2. Open lab3.c file and review it.

```
1  #include "lab3.h"
2
3  data_sc lab3(data_sc in1, data_sc in2, data_sc in3)
4  {
5      // Prevent IO protocols on all input ports
6      #pragma HLS INTERFACE ap_none port=in3
7      #pragma HLS INTERFACE ap_none port=in2
8      #pragma HLS INTERFACE ap_none port=in1
9
10     data_sc sum;
11     sum = in1 + in2 + in3;
12     return sum;
13 }
```

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

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

The important points to take from this code are:

- Directives in the form of pragma have been added to the source code to prevent any *I/O port 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.
- 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.

0-3. Open lab3_test.c file and review it.

```

1  #include <stdio.h>
2  #include "lab3.h"
3  int main() {
4      data_sc inA, inB, inC;
5      data_sc sum;
6      data_sc refOut[5] = {60, 90, 120, 150, 180};
7      int pass;
8      int i;
9
10     inA = 10;
11     inB = 20;
12     inC = 30;
13
14     // Call the adder for 5 transactions
15     for (i=0; i<5; i++)
16     {
17         sum = lab3(inA, inB, inC);
18         fprintf(stdout, " %d+%d+%d=%d \n", inA, inB, inC, sum);
19         // Test the output against expected results
20         if (sum == refOut[i])
21             pass = 1;
22         else
23             pass = 0;
24
25         inA=inA+10;
26         inB=inB+10;
27         inC=inC+10;
28     }
29     if (pass)
30     {
31         fprintf(stdout, "-----Pass!-----\n");
32         return 0;
33     }
34     else
35     {
36         fprintf(stderr, "-----Fail!-----\n");
37         return 1;
38     }
39 }

```

Figure 4: C Code with the Directives in the Form of Pragmas

There are five invocations of the function lab3 and five sets of expected results are provided.

Creating a Vivado HLS Tool Project

Step 1

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

1-1. Write the commands to create a new project, associate source files, and configure solution settings Launch the Vivado HLS tool command prompt and change the directory to the **C:\Xilinx_trn\HLS_2020\lab3_block_level_protocol** working directory.

1-1-1. Browse to the C:\Xilinx_trn\HLS_2020\lab3_block_level_protocol directory using Windows Explorer and open **lab3.tcl** with your preferred text editor .

1-1-2. Complete the following sections in the file opened:

- Project name (*lab3_prj*)
- Top-level function for synthesis (*lab3*)
- Source files (*./source/lab3.c*)
- Test bench (*./source/lab3_test.c*)
- A solution name (*base*)
- Xilinx device (*xa7a12tcs325-1Q*)
- Clock period (*3*)
- Clock uncertainty (*0.1*)
- Run the C simulation

An example of the file completed is on the figure.

```
1  # Insert the command to create new project
2  open_project -reset lab3_prj
3
4  # Insert the command to add design file
5  add_files ./source/lab3.c
6
7  # Insert the command to specify the top-level function
8  set_top lab3
9
10 # Insert the command to add testbench file
11 add_files -tb ./source/lab3_test.c
12
13 # Insert the command to create the solution named solution1
14 open_solution -reset "base"
15
16 # Insert the command to associate the device to the solution1
17 set_part {xa7a12tcs325-1Q}
18
19 # Insert the command to associate clock period to the solution1
20 create_clock -period 3 -name clk
21 set_clock_uncertainty 0.1
22
23 # Insert the command to run C simulation
24 csim_design -clean
25
26 exit
```

Figure 5: Vivado HLS with base solution

1-1-3. Save and close the `lab3.tcl` file.

1-2. Launch the Vivado HLS tool command prompt and change the directory to the `C:\Xilinx_trn\HLS_2020\lab3_block_level_protocol` working directory.

1-2-1. For Windows 10: **Start > Xilinx Design Tools > Vivado HLS 2020.1 Command Prompt.**

1-2-2. Enter the following command to change the directory to the current working directory:

```
cd C:\Xilinx_trn\HLS_2020\lab3_block_level_protocol
```

1-2-3. Enter the following command in the Vivado HLS tool command prompt to run the `lab3.tcl` file:

```
vivado_hls -f lab3.tcl
```

Note: Observe that the C simulation passed.

1-2-4. Enter the following command to open the Vivado HLS tool project:

```
vivado_hls -p lab3_prj
```

The Vivado HLS tool GUI opens.

Reviewing the Default Block-Level I/O Protocol

Step 2

2-1. Open the Vivado HLS tool GUI.

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

```
vivado_hls -p lab3_prj
```

The Vivado HLS tool GUI opens.

2-1-2. In the GUI opened you should see base solution.

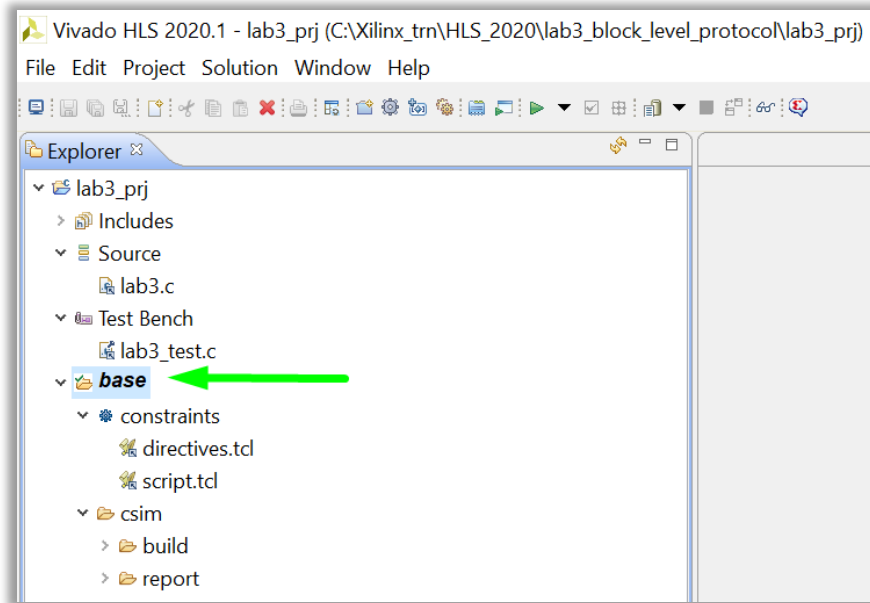


Figure 6: Vivado HLS with base solution

2-2. Synthesize the design.

- 2-2-1. Select **Solution > Run C Synthesis > Active Solution** or click the **Run Synthesis** icon in the menu bar.



Figure 3-7: Launching Synthesis

This option synthesizes the currently selected solution.

All solutions (or selected solutions) can be synthesized by using the drop-down menu next to the synthesis icon. You can synthesize all solutions or synthesize selected solutions in addition to the default.

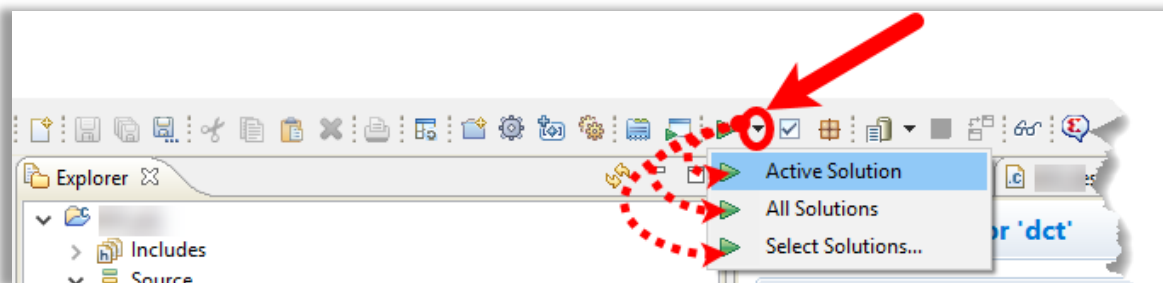


Figure 8: Options for What to Synthesize

When the synthesis completes, the Synthesis report will be displayed in the Information pane.

2-2-2. Select **Interface** > **Summary** in the Outline pane.

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

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	lab3	return value
ap_rst	in	1	ap_ctrl_hs	lab3	return value
ap_start	in	1	ap_ctrl_hs	lab3	return value
ap_done	out	1	ap_ctrl_hs	lab3	return value
ap_idle	out	1	ap_ctrl_hs	lab3	return value
ap_ready	out	1	ap_ctrl_hs	lab3	return value
ap_return	out	16	ap_ctrl_hs	lab3	return value
in1	in	16	ap_none	in1	scalar
in2	in	16	ap_none	in2	scalar
in3	in	16	ap_none	in3	scalar

Figure 9: 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 I/O 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 16-bit inputs and have the I/O protocol `ap_none` as specified by the directives (shown in Step 0).
 - The design has a 16-bit output port for the function return `ap_return`.

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

The table below summarizes the behavior of the signals for block-level I/O 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.</p> <p>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>

Signals	Description
ap_idle	<p>This signal indicates if the design is operating or idle (no operation).</p> <p>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** or click the **Run C/RTL Cosimulation** icon ().

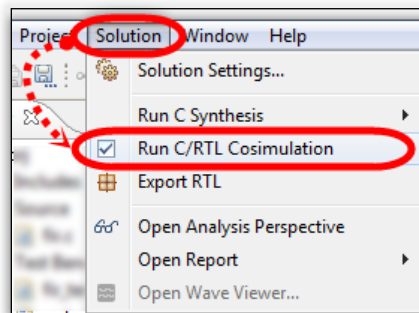


Figure 10: Launching from the Menu

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

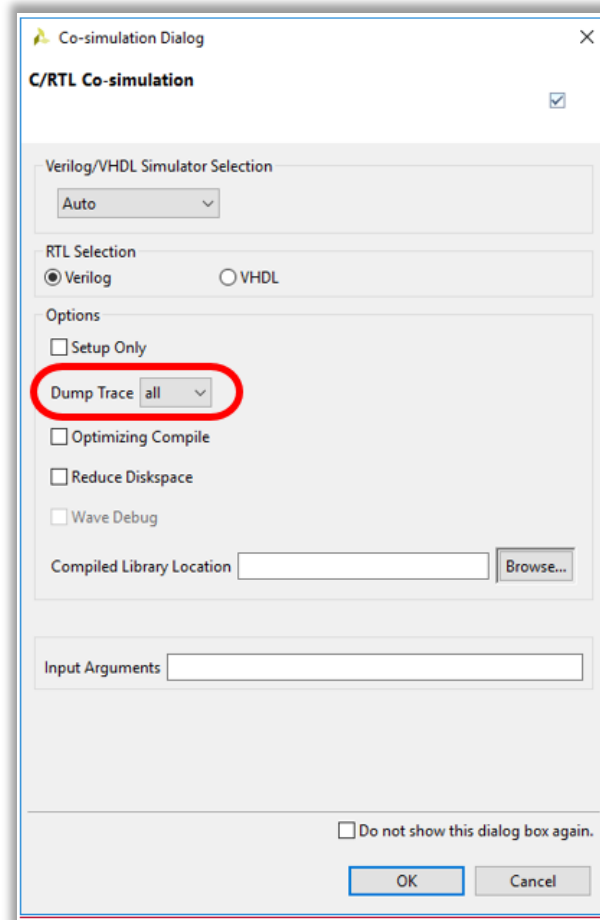


Figure 11: Co-simulation Dialog Box

Each of the options controls how C/RTL co-simulation is run:

- RTL Selection: Select the RTL that is simulated (Verilog/VHDL).
- Setup Only: This creates all the files (wrappers, adapters, and scripts) required to run the simulation but does not execute the simulator.
- Dump Trace: During RTL verification, the trace files can be saved and viewed using an appropriate viewer. By selecting this option, the trace file will be saved to the `<solution>/sim/<RTL>` folder.
- Optimizing Compile: This ensures that a high level of optimization is used to compile the C test bench. Using this option increases the compile time but the simulation executes faster.
- Reduce Diskspace: Saves the results for all transactions before executing RTL simulation. In some cases, this can result in large data files. This option can be used to execute one transaction at a time and reduce the amount of disk space required for the file. If the function is executed N times in the C test bench, the

- `reduce_diskspace` option ensures N separate RTL simulations are performed. This causes the simulation to run slower.
- Compiled Library Location: This specifies the location of the compiled library for a third-party RTL simulator.
 - Input Arguments: This allows the specification of any arguments required by the test bench.

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).

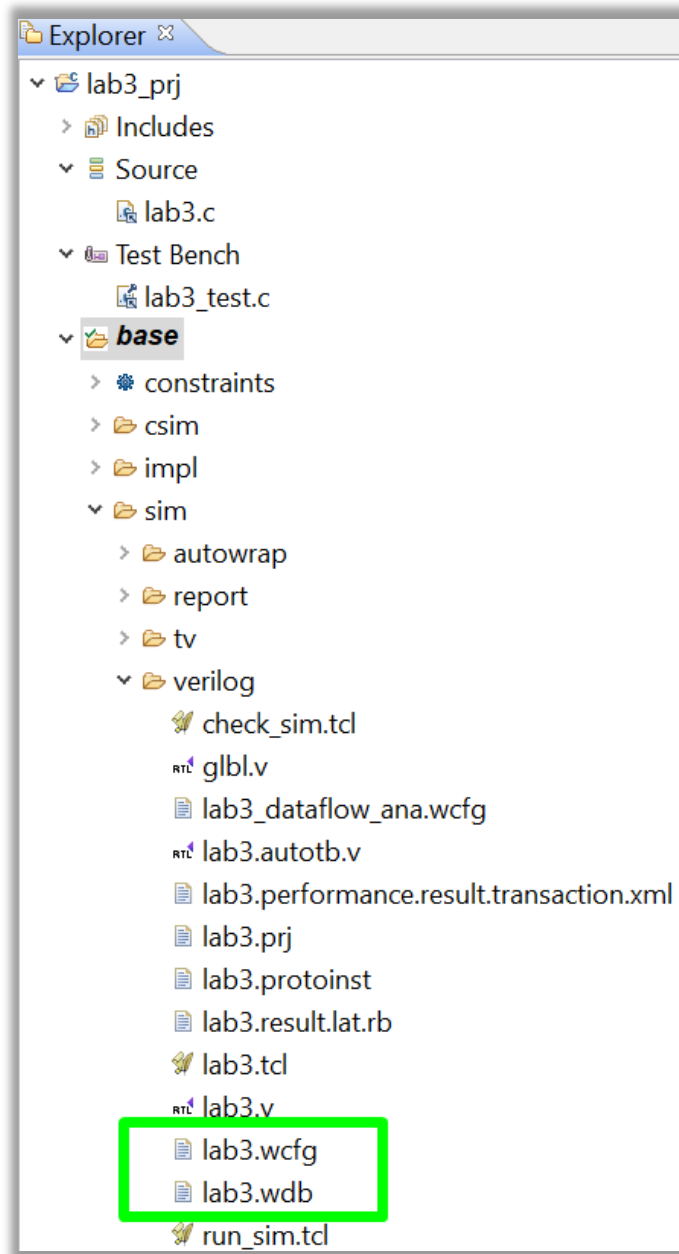


Figure 12: Verilog Vivado Simulator Co-simulation Results

2-4. Now, view the trace files that has been generated for the Vivado simulator.

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

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

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

2-4-2. Expand the Design Top Signals as shown below.

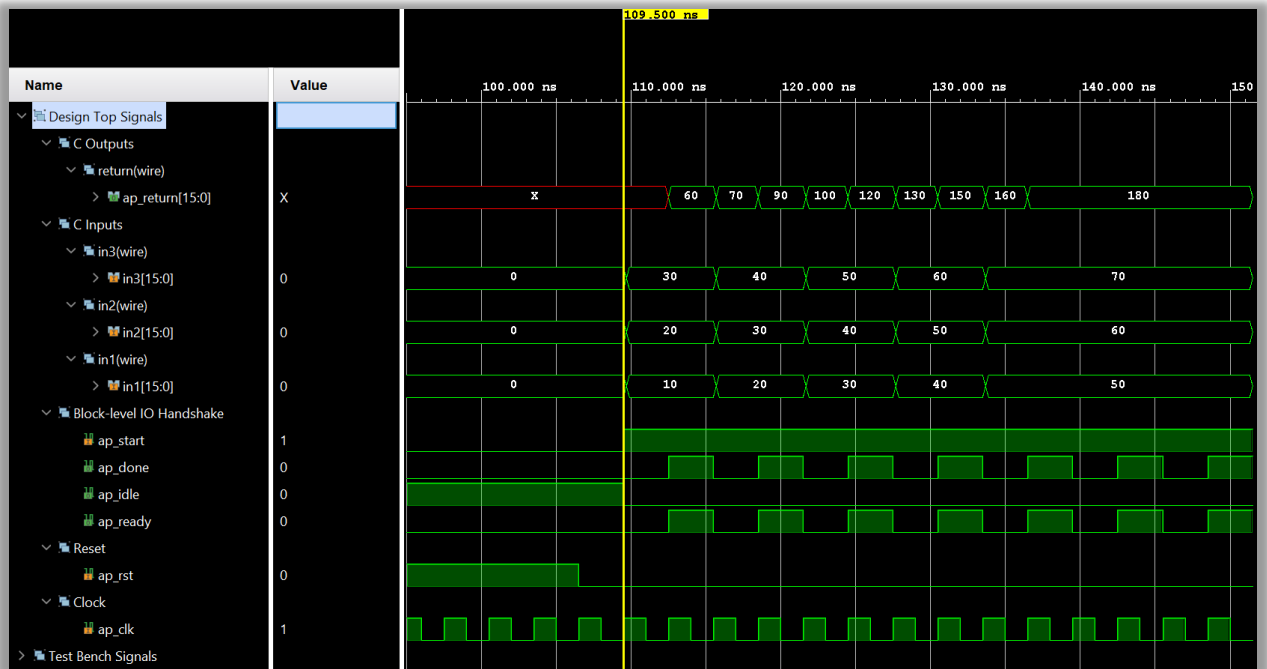


Figure 13: Analyzing the RTL Trace File

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

The waveform above shows the behavior of the block-level I/O 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 input ports `in1`, `in2`, and `in3` respectively.
- Output signal `ap_ready` goes high to indicate the design is ready for new inputs on next clock cycle.
- Output signal `ap_done` indicates when the design is finished and that the value on 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 tool GUI.

Modifying the Block-Level I/O Protocol

Step 3

The default block-level I/O 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 (base) settings.

- 3-1-1. Select **Project** > **New Solution** or click the **New Solution** icon () from the toolbar.
- 3-1-2. Leave the options at their default settings.

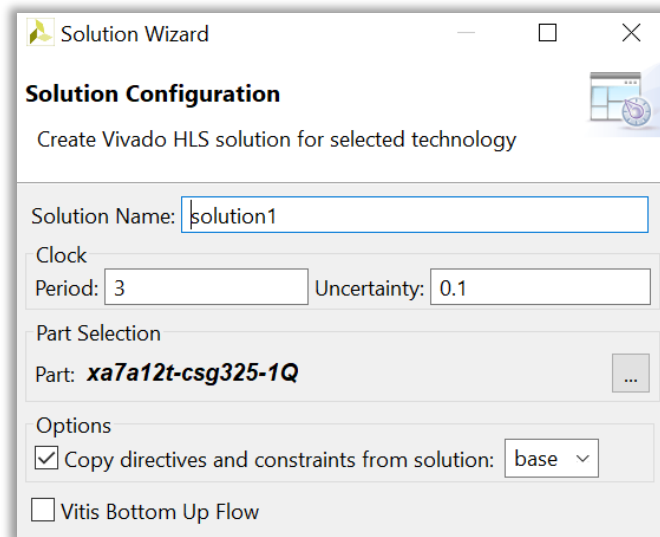


Figure 14: Creating a New Solution (solution2)

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

3-2. Apply the directive `ap_ctrl_none` to make no block-level I/O 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 **lab3.c** file is open and active in the Information pane.
- 3-2-3. In the Directive tab right-click the **lab3** function in the Directive tab and select **Insert Directive**.
- 3-2-4. Select **INTERFACE** from the Directive drop-down list.
- 3-2-5. Select the *mode (optional)* as **`ap_ctrl_none`**.
- 3-2-6. Select **Source File** in the Destination section.

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.

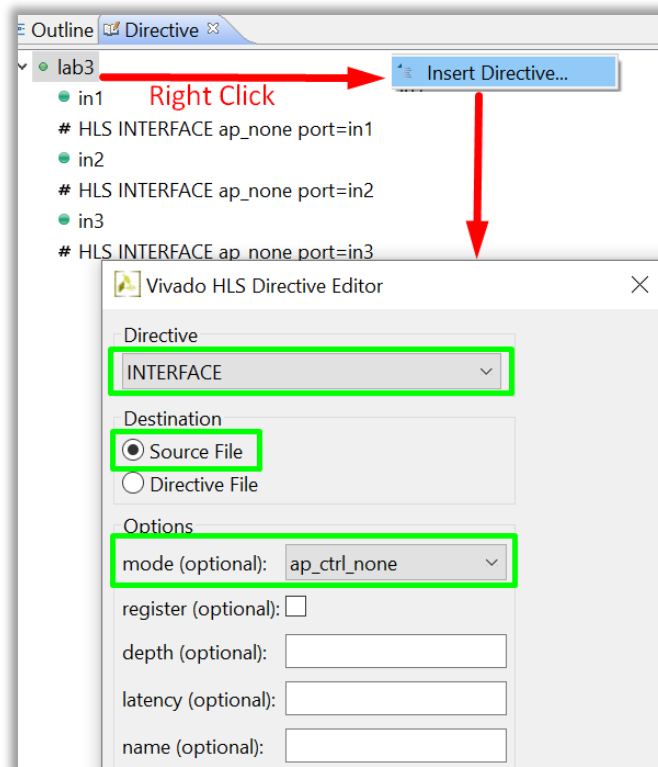


Figure 15: Applying the Interface Directive - `ap_ctrl_none`

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

- `ap_ctrl_none`: No block-level I/O control protocol.
- `ap_ctrl_hs`: The block-level I/O control handshake protocol we have reviewed.
- `ap_ctrl_chain`: The block-level I/O protocol for control chaining. This I/O protocol is primarily used for chaining pipelined blocks together.

- o `s_axilite`: May be applied in addition to `ap_ctrl_hs` or `ap_ctrl_chain` to implement the block-level I/O 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.

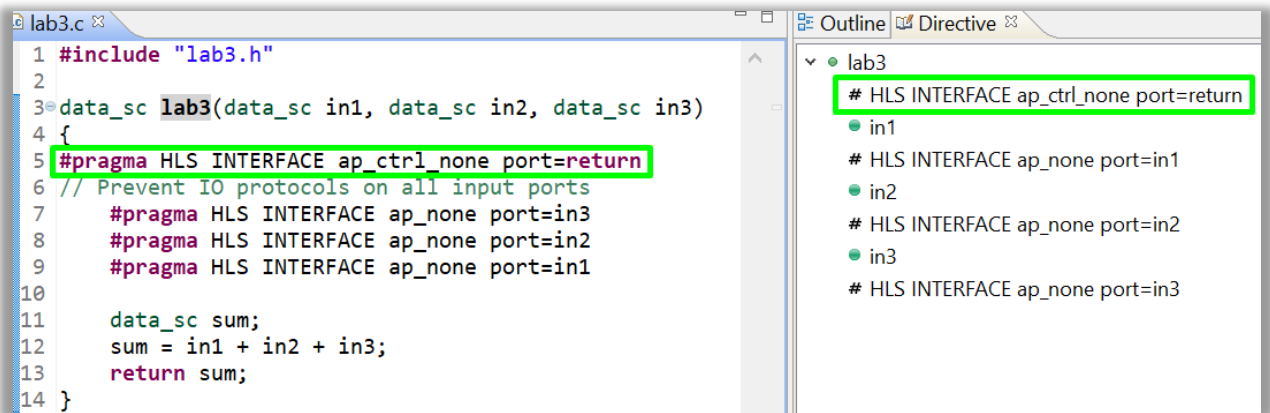


Figure 16: Block-Level Interface Directive `ap_ctrl_none`

3-3. Synthesize the design.

- 3-3-1. Select **Solution** > **Run C Synthesis** > **Active Solution** or click the **Run Synthesis** icon in the menu bar.
- 3-3-2. Click **Yes** to accept the changes to the source file.
- 3-3-3. Select **Interface** > **Summary** in the Outline pane when the synthesis report opens.

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

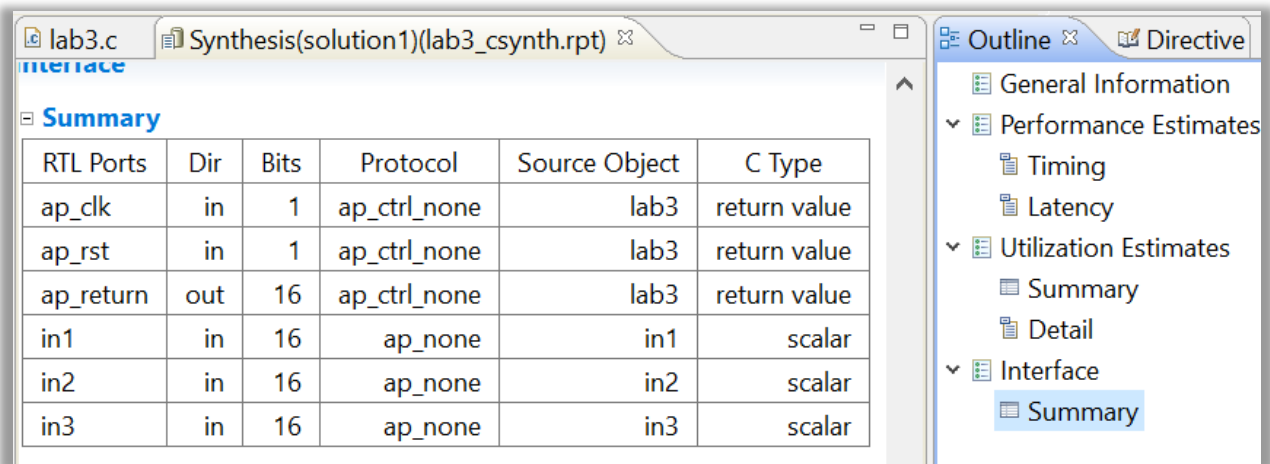


Figure 17: Viewing the Interface Summary Report

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

In addition, the RTL cosimulation feature **requires** a block-level I/O 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 ***
```

3-3-4. Close the Vivado HLS tool GUI and the command prompt.

Summary

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