

HLS Component Development Flow

Using the Vitis Unified IDE 2024.1

Lab-1

Feb-2025

Abstract

This lab introduces how to perform the most common processes related to the high-level synthesis (HLS) design flow using the Vitis™ Unified IDE.

Objectives

After completing this lab, you will be able to:

- Create an HLS component in the Vitis Unified IDE.
- Simulate a C design by using a self-checking test bench.
- Synthesize the design.
- Perform design analysis using the Analysis Perspective view.
- Perform co-simulation on a generated RTL design by using a provided C test bench.
- Implement the design.

Introduction

This lab introduces major features of the Vitis Unified IDE applicable for the high-level synthesis (HLS) component development flow. You will use the Vitis Unified IDE in GUI mode to create a project. You will also simulate, synthesize, and implement the provided design.


This design implements a matrix multiplication which is provided as C source "`matrixmul.cpp`". The corresponding C test-bench "`matrixmul_test.cpp`" is also provided.

Lab Steps

Step 1: Creating an HLS Component

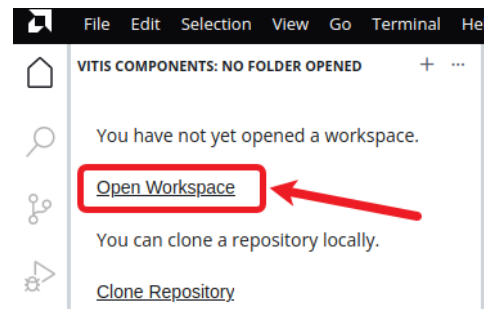
In this step, you will launch the Vitis Unified IDE tool and create a new HLS component based on a provided matrix multiplication (matrixmul) design.

1.1 Launch the Vitis Unified IDE

Click the *Vitis Unified IDE 2024.1* icon () from the desktop to launch the tool.

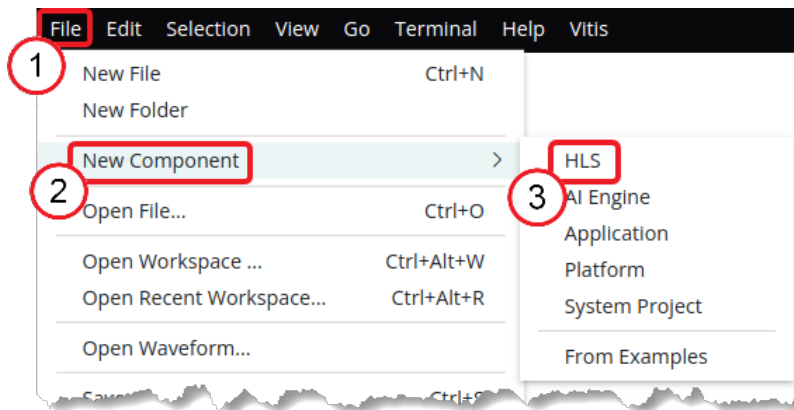
1.2 Set the workspace

- From the Vitis Components window, click the **Open Workspace** link.
- Browse to your desired location e.g. `/home/indi/MnRA/Lab1`
- Click **Ok** to open the new workspace. The tool relaunches using the new workspace.



1.3 Create the HLS component for matrixmul design

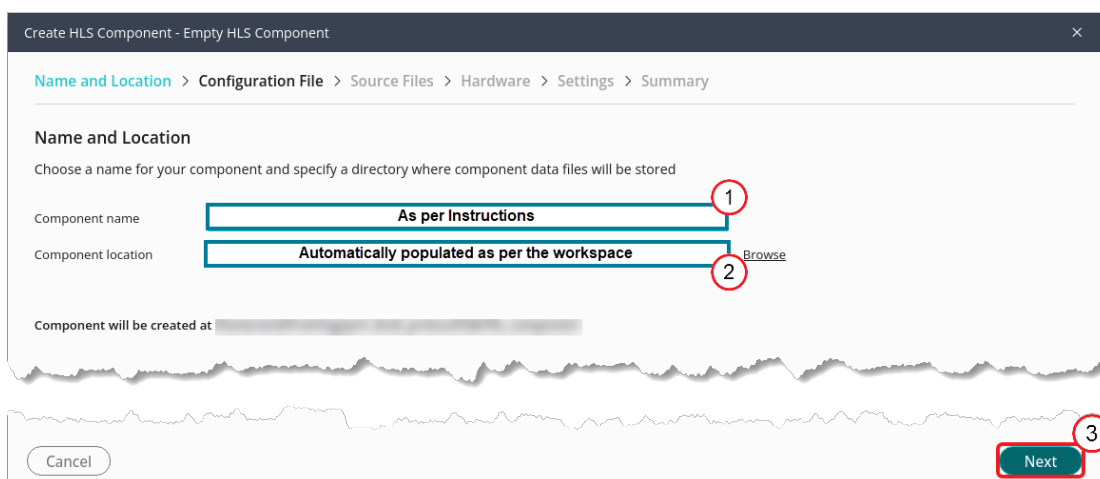
- Navigate to the toolbar and click **File** (1).
- Hover over **New Component** (2).
- Select **HLS** from the options to create the HLS component (3).



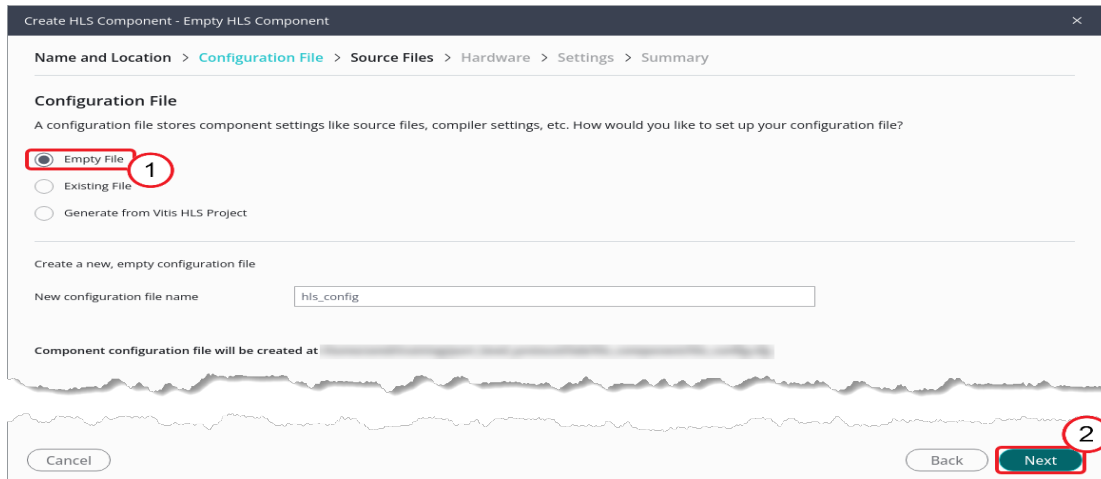
The Create HLS Component dialog box appears.

- Enter `v1_MM_no_opt` in the **Component name** field (1). This sets the name of the HLS component, and the associated directories and files are then populated within the current workspace. Note that the Component location field is automatically set to the workspace location (2)

Note: You can choose any name however it should be meaningful based on your optimization strategies to be applied. e.g. above name mentions that it is **version-1** (abbreviated as **v1**) of **matrixmul** component (abbreviated as **MM**) where no optimization has been applied (reflected as **_no_opt**).

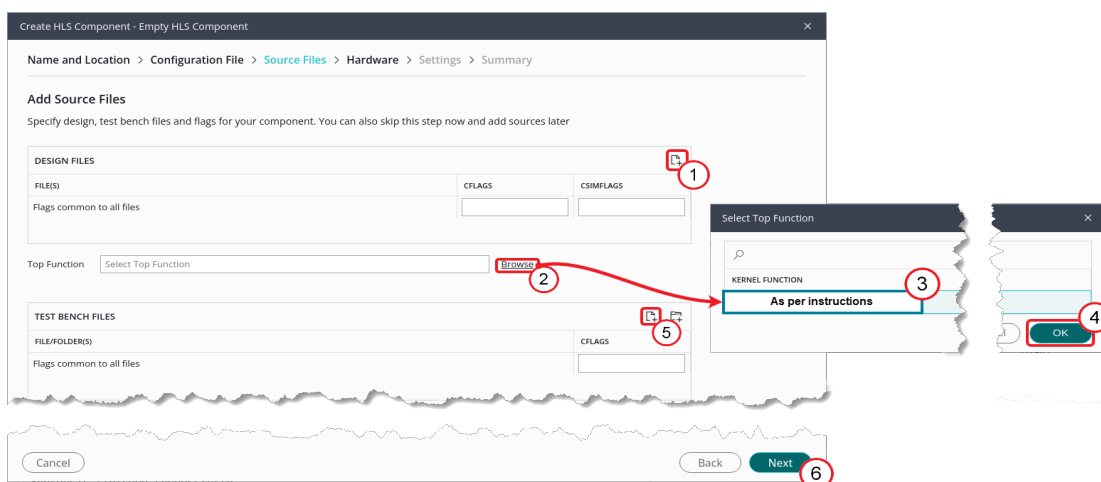


- e. Click **Next** to advance to the Configuration File page (3).
- f. Select **Empty File** (1). The remaining fields can be left at their default settings.
Note: that the new configuration name will be set to `hls_config`.
- g. Click **Next** to advance to the Source Files page (2).



1.4 Add the source and testbench files to the HLS component

- a. Click the **Add Files** icon in the Design Files section (1).
- b. Browse to the following directory: `/home/indi/MnRA/Lab1/src`
- c. Select **matrixmul.cpp** and click **Open**.
- d. Click **Browse** next to the Top Function field (2).
- e. Select the **matrixmul** kernel function (3) and click OK (4).
- f. Click the Add Files icon in the Test Bench Files section (5).
- g. Browse to the following directory: `/home/indi/MnRA/Lab1/src`
- h. Select **matrixmul_test.cpp** and click **Open**.



- i. The test bench contains a flag (`HW_COSIM`) to control the invocation of the `matrixmul()` function during hardware co-simulation.
- j. To enable this, add the flag `-DHW_COSIM` in the **CFLAG** field.
- k. If you don't have any flags defined, you can leave this field empty.

Name and Location > Configuration File > **Source Files** > Hardware > Settings > Summary

Add Source Files

Specify design files, test bench files and flags for your component. You can also skip this step now and add sources later.

DESIGN FILES		
FILE(S)	CFLAGS	CSIMFLAGS
Flags common to all files		
/home/indi/MnRA/MM_COSIM/src/matrixmul.cpp		

Top Function: [Browse](#)

TEST BENCH FILES	
FILE/FOLDER(S)	CFLAGS
Flags common to all files	
/home/indi/MnRA/MM_COSIM/src/matrixmul_test.cpp	-DHW_COSIM

[Cancel](#) [Back](#) [Next](#)

Click Next to advance to the Part selection page (6).

1.5 Specify the part number and settings for the HLS component

- Ensure that the **Part** option is selected (1).
- Select **xczu7ev-ffvc1156-2-e** from the list of parts (2).

Create HLS Component - Empty HLS Component

Name and Location > Configuration File > Source Files > **Hardware** > Settings > Summary

Part

Specify a board, device, or platform you are compiling your component for

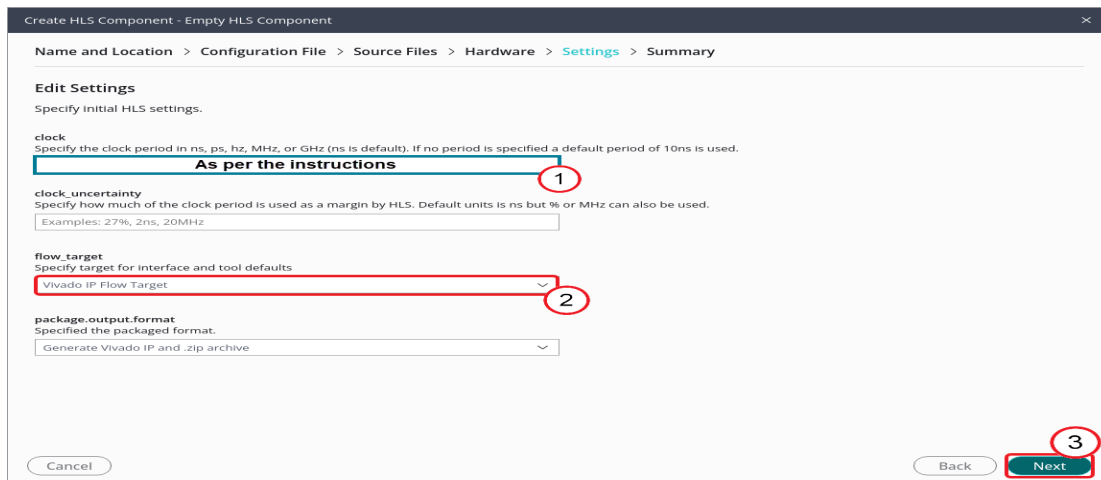
☒ **Part** ☐ Platform ☐ Hardware Design

As per instructions

PART	FAMILY	PACKAGE	SPEED	LUT	FF	DSP	BRAM
xczu7ev-ffvc1156-2-e	7	1156	2	e			

[Cancel](#) [Back](#) [Next](#)

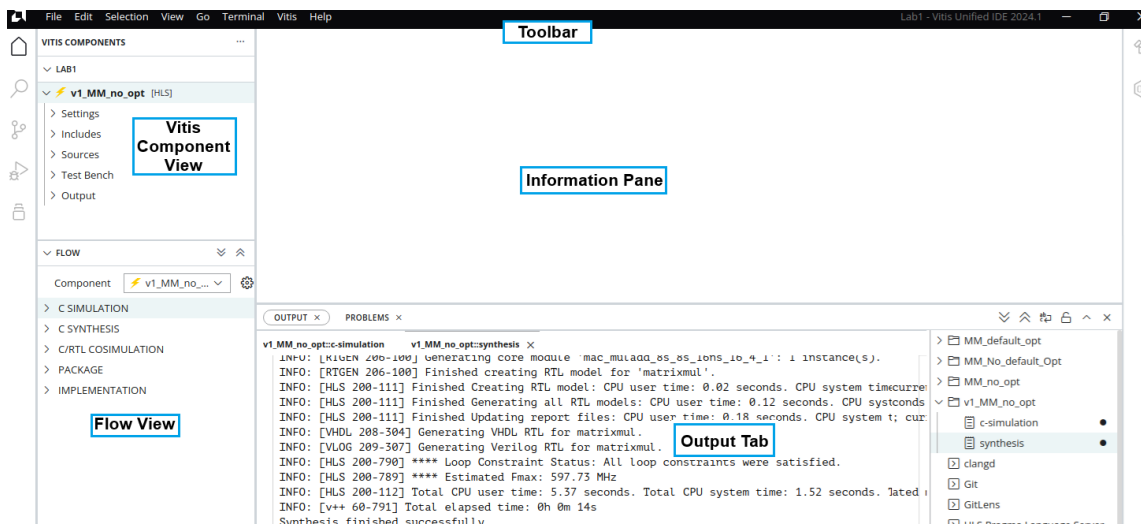
- Click **Next** to advance to the Settings page (3).
- Enter **10** for the **clock** setting (1).
- Select **Vivado IP Flow Target** from the **flow_target** drop-down list (2). Note: By default, the **package.output.format** option is selected as **Generate Vivado IP** and **.zip** archive.



f. Click **Next** to advance to the Summary page (3).

g. Review the HLS component summary and click **Finish**.

You will see the created HLS component in the Vitis Components view.



The Vitis Unified IDE GUI consists of various panes for proceeding with HLS development:

- The **Vitis Components view** enables you to navigate through the component hierarchy. Expandable sub-folders organize various components, such as source files and test benches.
- The **Information pane** displays report summaries and shows the contents of open files. Files can be opened from the Vitis Components window.
- The **Output view** displays the output when the Vitis Unified IDE is running synthesis or simulation.
- The **Flow view** provides access to commands and processes to take through simulation, synthesis, and exported output.

There are several other views that you can open according to need as part of the development process, such as the **HLS Directive** and **HLS Module Hierarchy** views.

Step 2: Running C Simulation

With the component created, you will now validate the C code for both syntax and behavior. You will use a provided self-checking C test bench code to validate the behavior.

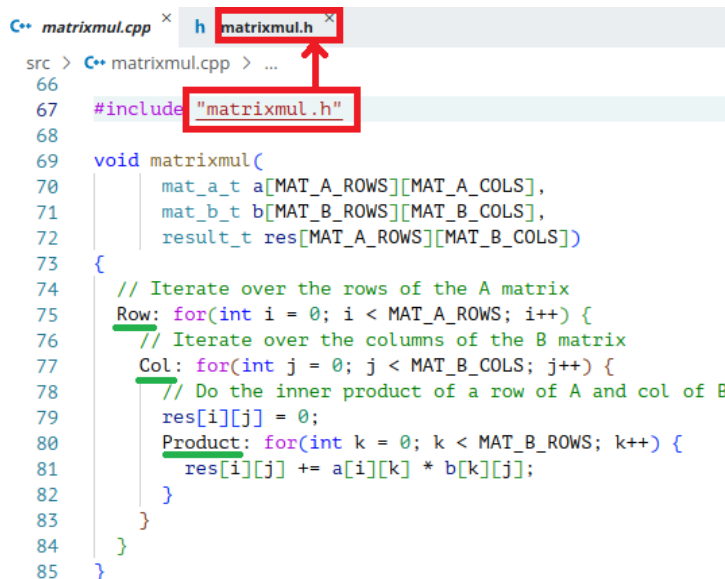
2.1 Review the provided source and test bench files.

a. Expand **Sources** in the Vitis Components window.

b. Click **matrixmul.cpp** to open the file in the Information pane.

c. Review the code structures.

It can be seen that the design consists of three nested loops. The **Product** loop is the inner most loop performing the actual Matrix elements product and sum. The **Col** loop is the outer-loop which feeds the next column element data with the passed row element data to the Product loop. Finally, **Row** is the outer-most loop. The **res[i][j]=0** (line 79) resets the result every time a new row element is passed and new column element is used.



```
src > C++ matrixmul.cpp > ...
66
67 #include "matrixmul.h"
68
69 void matrixmul(
70     mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
71     mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
72     result_t res[MAT_A_ROWS][MAT_B_COLS])
73 {
74     // Iterate over the rows of the A matrix
75     Row: for(int i = 0; i < MAT_A_ROWS; i++) {
76         // Iterate over the columns of the B matrix
77         Col: for(int j = 0; j < MAT_B_COLS; j++) {
78             // Do the inner product of a row of A and col of B
79             res[i][j] = 0;
80             Product: for(int k = 0; k < MAT_B_ROWS; k++) {
81                 res[i][j] += a[i][k] * b[k][j];
82             }
83         }
84     }
85 }
```

d. In the initial section of the code, locate `#include "matrixmul.h"` and perform `<Ctrl> + click` on `"matrixmul.h"`.

This opens the `matrixmul.h` file in editor view besides the main source file.

e. Review the contents of the header file.

f. Expand the **Test Bench** folder in the Vitis Components window.

g. Click **matrixmul_test.cpp** to open it in the Information pane.

This test bench is self-checking; i.e., the computed output is compared against a reference golden output and returns either pass or fail.

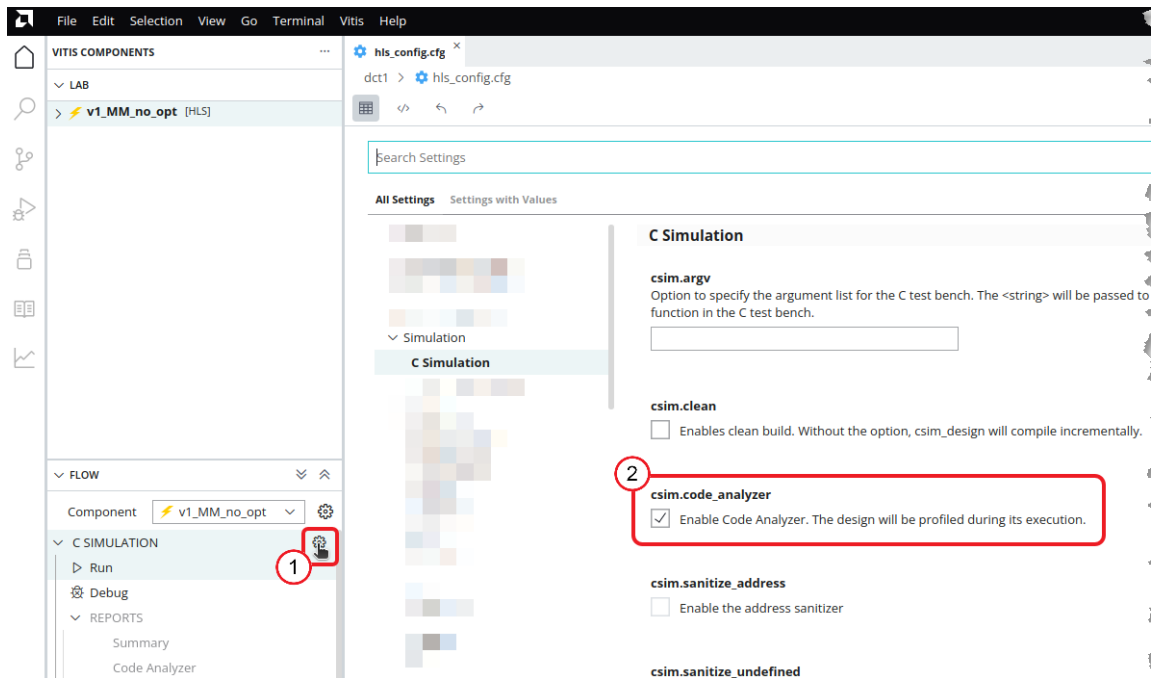
2.2 Explore the C simulation configuration settings and enable Vitis HLS Code Analyzer.

a. From the **Flow** view, hover over **C Simulation** and click the **Settings** icon that appears (1).

b. The `hls_config.cfg` configuration file opens in the Information pane.

c. Select `csim.code_analyzer` to enable Vitis HLS Code Analyzer (2).

Note: The Vitis HLS Code Analyzer feature will be available only under the new HLS license.



There are several other options and settings such as **argv**, **clean**, **idflags**, **setup** etc. that can be enabled to control how simulation runs.

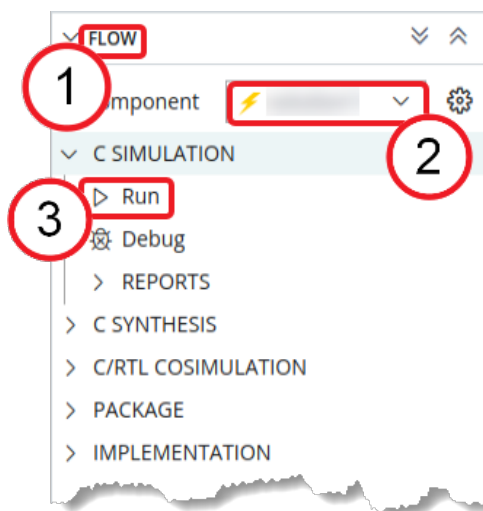
2.3 Simulate the design

- a. Go to the **Flow** view (1).

The Flow view displays C Simulation, C Synthesis, C/RTL Cosimulation, Package, and Implementation as the primary steps of the HLS component workflow.

- b. Select the **active HLS component** (v1_MM_no_opt) to simulate from the Component drop-down list (2).

Note: You can have multiple components in a system project. These components will be displayed here.



- c. Click **Run** under C Simulation (3).

You can view the simulation log in the Output tab.

- d. Select the **Output** tab in the lower portion of the Vitis Unified IDE GUI. You may need to scroll to view the output produced by the simulation.

```

Vitis Server  clang  dct::c-simulation x  HLS Pragma Language Server  GitLens  Git
| INFO: [HLS 200-1611] Setting target device to 'xczu7ev-ffvc1156-2-e'

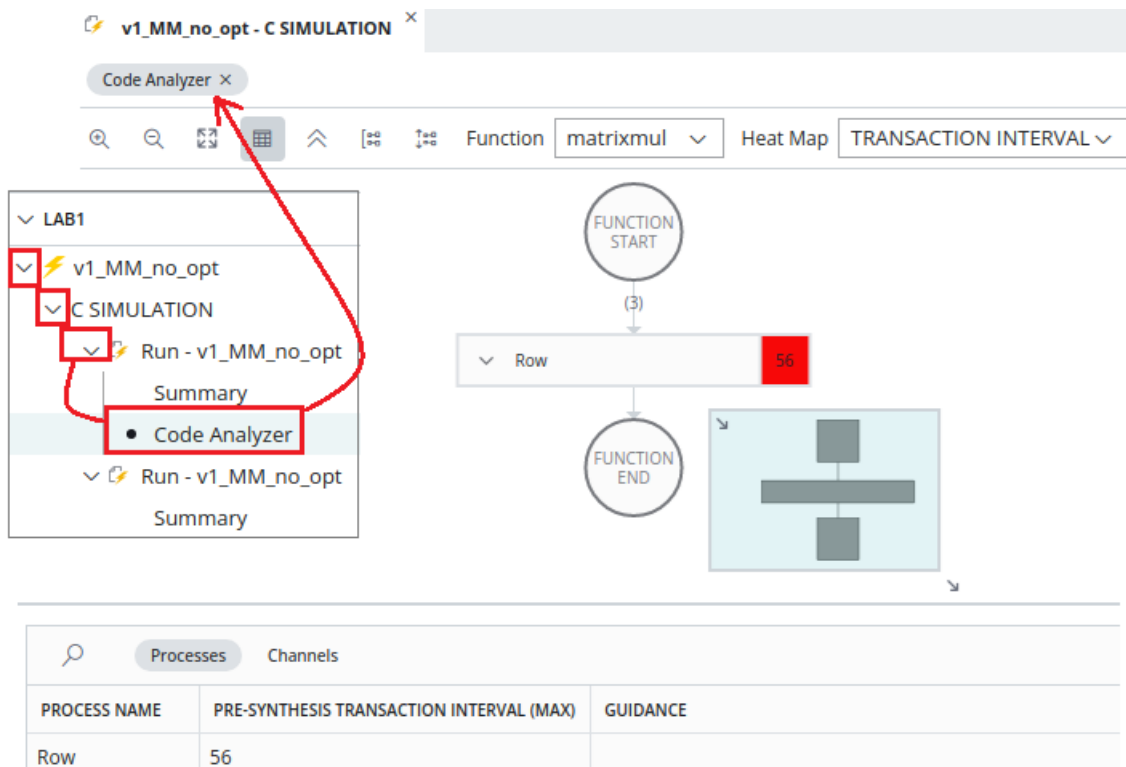
INFO: [HLS 200-112] Total CPU user time: 5.66 seconds. Total CPU system time: 0.98 seconds.
INFO: [Common 17-206] Exiting vitis_hls at
INFO: [vitis-run 60-791] Total elapsed time: 0h 0m 12s
C-simulation finished successfully

```

You will see "C-simulation finished successfully" message at the end.

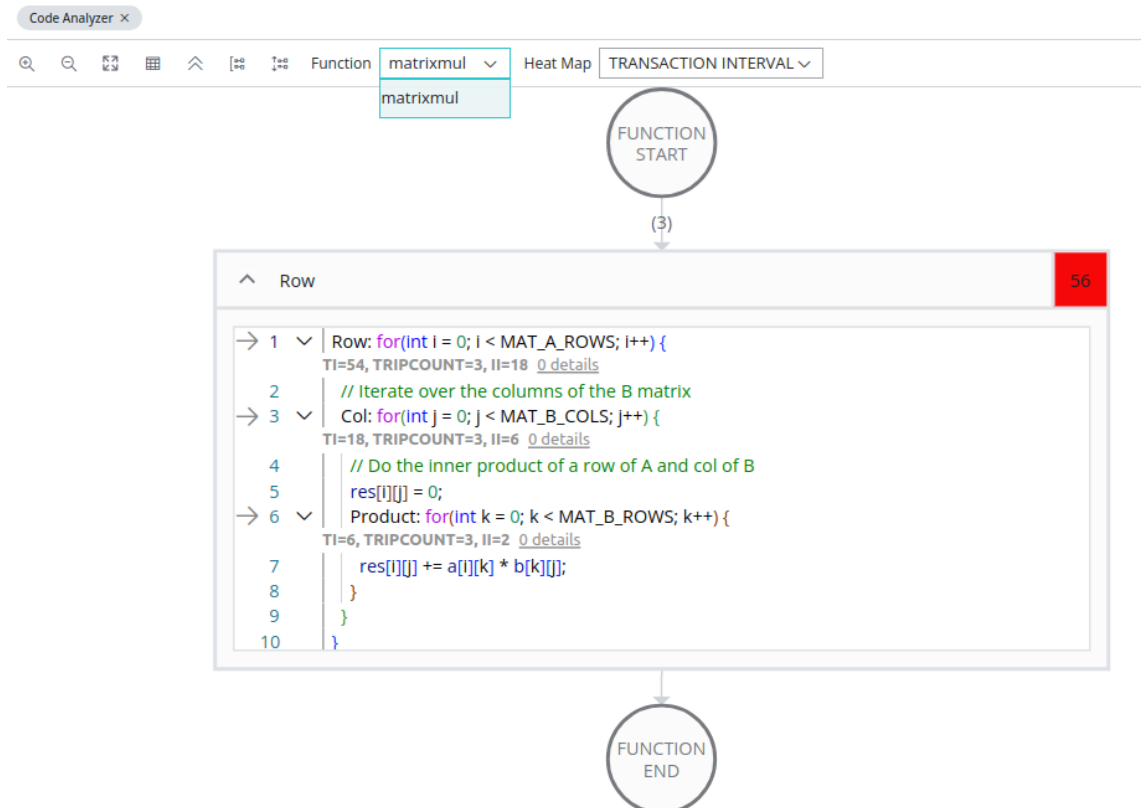
2.4 Visualize the HLS code using Code Analyzer.

- Select **View > Analysis** from the toolbar.
- From the Analysis view, expand **v1_MM_no_opt > C Simulation > Run - v1_MM_no_op (1)**.
- Click **Code Analyzer** to open the Code Analyzer report in the Information pane (2).
The Code Analyzer report initially displays a graph of the processes and channels defined by the top-level function of the component.



- With this graph view, nodes represent dataflow processes with their names being function names or loops labels (unnamed loops are named **Process #N**).
- Estimated transaction intervals (TI) are also presented next to the node name.
- Edges are the communication channels extracted from the variables of the design. You can see their names, the volume of data and the average throughput, expressed by default in bits per second, both estimated from the C test bench run.

- For the **row** node, click the **down arrow** to show the code (3).
The performance analysis of Code Analyzer is overlaid in the code snippet. Here, this is shown after line 1.
Similarly the performance for **col** and **product** loops can be seen.



Observe the following:

- Code Analyzer estimates the performance of the innermost line first.
- The transaction interval is $TI = TRIPCOUNT * II$, so for **Product Loop**, you get $TI = 3 * 2 = 6$.
- You can repeat the investigations hierarchically up into the parent loop: the sum of all $TI(s)$ of all the statements in the loop body is computed.

Note about performance analysis numbers: Performance analysis numbers are shown at the beginning of a loop, within its curly braces, or right after a function call site.

Note about reported trip count values:

- If loop bounds are compile-time constants, they are reported in Code Analyzer.
- Otherwise, if a trip count pragma is provided, this value is used by Code Analyzer.
- Otherwise, the trip count value is measured when running the C test bench.

In any case, the reported trip counts account for the loop unrolling: loop trip counts are divided by unrolling factors. Fully unrolled loops will always have a reported trip count of 1.

Step 3: Synthesizing the Design

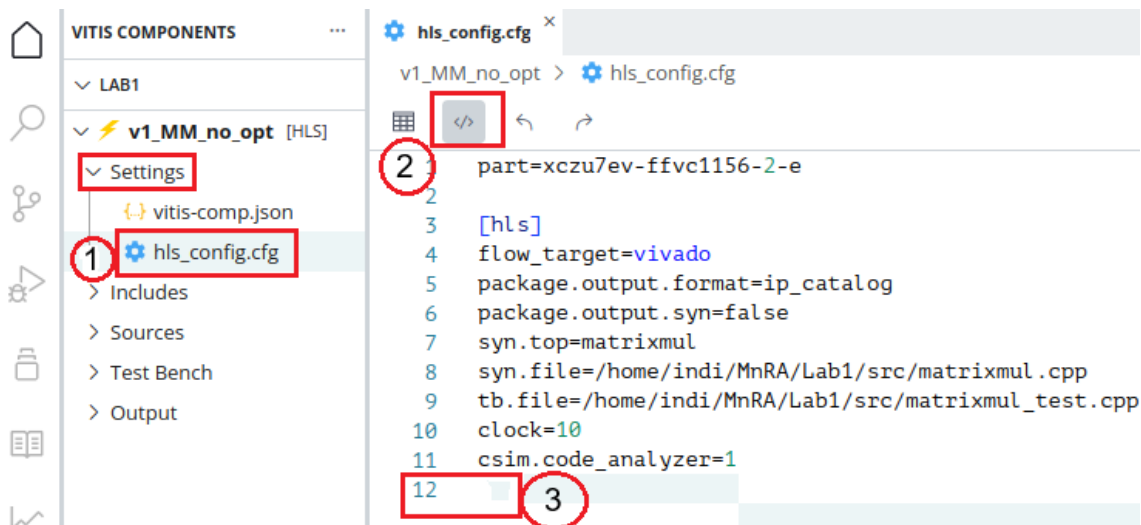
Next, you will synthesize the design by using the default settings. You will then analyze the result to find the resources needed to implement the C function.

3.1 Disable default optimizations

The tool can automatically identify basic optimizations to enhance the performance of the design. These optimizations are applied during the Synthesis phase, which alters the behavior of the code flow. To understand the default behavior of the code after synthesis, it is advisable to first disable these default optimizations. You can apply or disable optimizations using "HLS Directives" view (more on this in Lab-2).

- Select **View > Vitis Components** from the toolbar.
- From the Vitis Components, expand **v1_MM_no_opt > Settings > hls_onfig.cfg** (1).

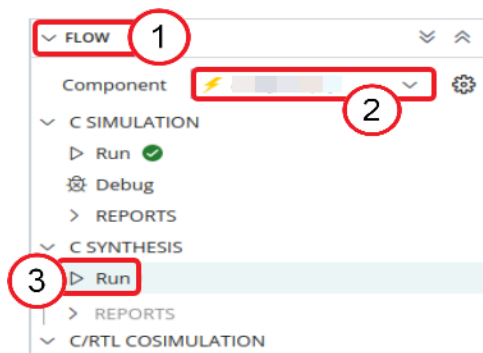
- c. Click on **Source Editor** button (2).
It will open the configuration file. Please take a look into its contents.



- d. Add following lines at end of this file (3). i.e. at line 12.
`syn.directive.pipeline=matrixmul/Col off`
`syn.directive.pipeline=matrixmul/Product off`
- e. Close the hls_onfig.cfg file. And click save All from file menu.

3.2 Synthesize the design

- a. Go to the **Flow** view (1).
- b. Select the active HLS component to synthesize from the Component drop-down list (2).
- c. Click Run under C synthesis (3).



You can view the synthesis log in the Output tab.

3.3 View the Synthesis report

- a. expand `v1_MM_no_opt > Output > syn > report > matrixmul_csynth.rpt`
- b. Click `matrixmul_csynth.rpt` to view the Synthesis report.
- c. Review the Performance Estimates and Utilization Estimates in the Synthesis report.

```

1  =====
2  == Vitis HLS Report for 'matrixmul'
3  =====
4  * Project:      matrixmul
5  * Solution:     hls (Vivado IP Flow Target)
6  * Product family: zynqplus
7  * Target device: xczu7ev-ffvc1156-2-e
8  =====
9  == Performance Estimates
10 =====
11
12 + Timing:
13   * Summary:
14   +-----+-----+-----+-----+
15   | Clock | Target | Estimated| Uncertainty|
16   +-----+-----+-----+-----+
17   |lap_clk | 10.00 ns| 1.673 ns| 2.70 ns|
18   +-----+-----+-----+-----+
19 + Latency:
20   * Summary:
21   +-----+-----+-----+-----+-----+-----+
22   | Latency (cycles) | Latency (absolute) | Interval | Pipeline|
23   | min | max | min | max | min | max | Type |
24   +-----+-----+-----+-----+-----+-----+
25   | 160| 160| 1.600 us| 1.600 us| 161| 161| no|
26   +-----+-----+-----+-----+-----+-----+
27
28 + Detail:
29   * Loop:
30   +-----+-----+-----+-----+-----+-----+
31   | Loop Name | Latency (cycles) | Iteration| Initiation Interval | Trip |
32   | min | max | Latency | achieved | target | Count| Pipelined|
33   +-----+-----+-----+-----+-----+-----+
34   | - Row | 159| 159| 53| -| -| 3| no|
35   | + Col | 51| 51| 17| -| -| 3| no|
36   | ++ Product | 15| 15| 5| -| -| 3| no|
37   +-----+-----+-----+-----+-----+-----+

```

Observe the Timing section in "Performance Estimates", which shows the estimated clock period that this solution can be run. As long as the estimated time, which includes the uncertainty factor, is less than the target value, the design will meet timing. It also shows the Latency of your top module and further details of loops.

Observe the iteration latency and trip count of nested loops.

The **Product** loop completes one iteration in 5 clock cycles which is called **Iteration latency**. Since we are multiplying 3x3 matrices, the **Product** loop iterates 3 times which is given as **Trip count**.

In total one complete execution of **Product** loop will consume 15 clock cycles (**Iteration latency * Trip count**).

On the otherhand, one iteration of **Col** loop results in one complete execution of **Product** loop whose latency is 15 cycles. As a result the iteration latency of **Col** loop is 15 cycles of **Product** loop + one cycle to enter the **Product** loop + one cycle to exit from **Product** loop which totals to 17 cycles. Since **Col** loop also iterates 3 times so its total latency is $17 \times 3 = 51$ cycles.

Similarly **Row** loop has iteration latency of 53 cycles ($51 + 1 + 1$). And total latency of $53 \times 3 = 159$ cycles.

The top level function **matrixmul** requires 1 cycle to initialize the controls before entering into **Row** loop hence total latency is summarized as 160 cycles ($159 + 1$).

The **Interval** is the number of cycles required before accepting the next set of inputs. Since the design takes 160 cycles to compute the product of one set of matrices, it will be able to accept a new set of matrices for product calculation after 161 cycles ($160 + 1$). This additional cycle is needed to enter and initialize the module again.

The report also shows the **Utilization Estimates** of resources (FF, LUT, DSP, BRAM etc.) required and the top level **Interfaces**.

```

51  =====
52  == Utilization Estimates
53  =====
54  * Summary:
55  +-----+-----+-----+-----+-----+-----+
56  |      Name      | BRAM_18K| DSP | FF  | LUT  | URAM|
57  +-----+-----+-----+-----+-----+-----+
58  | DSP             |         | 1   | -   | -    | -   |
59  | Expression      |         | -   | 0   | 104  | -   |
60  | FIFO            |         | -   | -   | -    | -   |
61  | Instance        |         | -   | -   | -    | -   |
62  | Memory          |         | -   | -   | -    | -   |
63  | Multiplexer     |         | -   | 0   | 85   | -   |
64  | Register        |         | -   | 46  | -    | -   |
65  +-----+-----+-----+-----+-----+-----+
66  | Total           |         | 1   | 46  | 189  | 0   |
67  +-----+-----+-----+-----+-----+-----+
68  | Available       |        624| 1728| 460800| 230400| 96  |
69  +-----+-----+-----+-----+-----+-----+
70  | Utilization (%) |         0| ~0  | ~0   | ~0   | 0   |
71  +-----+-----+-----+-----+-----+-----+

143  =====
144  == Interface
145  =====
146  * Summary:
147  +-----+-----+-----+-----+-----+-----+
148  | RTL Ports | Dir | Bits| Protocol | Source Object | C Type |
149  +-----+-----+-----+-----+-----+-----+
150  | ap_clk     | in  | 1   | ap_ctrl_hs| matrixmul      | return value|
151  | ap_rst     | in  | 1   | ap_ctrl_hs| matrixmul      | return value|
152  | ap_start   | in  | 1   | ap_ctrl_hs| matrixmul      | return value|
153  | ap_done    | out | 1   | ap_ctrl_hs| matrixmul      | return value|
154  | ap_idle    | out | 1   | ap_ctrl_hs| matrixmul      | return value|
155  | ap_ready   | out | 1   | ap_ctrl_hs| matrixmul      | return value|
156  | a_address0 | out | 4   | ap_memory | a              | array|
157  | a_ce0      | out | 1   | ap_memory | a              | array|
158  | a_q0       | in  | 8   | ap_memory | a              | array|
159  | b_address0 | out | 4   | ap_memory | b              | array|
160  | b_ce0      | out | 1   | ap_memory | b              | array|
161  | b_q0       | in  | 8   | ap_memory | b              | array|
162  | res_address0 | out | 4   | ap_memory | res           | array|
163  | res_ce0    | out | 1   | ap_memory | res           | array|
164  | res_we0    | out | 1   | ap_memory | res           | array|
165  | res_d0     | out | 16  | ap_memory | res           | array|
166  +-----+-----+-----+-----+-----+-----+

```

There are two type of Interfaces:

Block level Control interfaces: The `ap_clk` and `ap_rst` signals are automatically added. `ap_start`, `ap_done`, and `ap_idle` are block-level signals used as handshaking signals to indicate when the design can accept the next computation command (`ap_idle`), when the next computation is started (`ap_start`), and when the computation is completed (`ap_done`).

Port level I/O interfaces: The arguments of the top level functions are synthesized as data ports with respective address (`argName_address0`), data (`argName_q0` or `argName_d0`), chip enable (`argName_ce0`) and write enable (`argName_we0`) controls. Observe how the three inputs `a`, `b` and `res` are synthesized as

above mentioned ports.

Step 4: Analyzing the Design

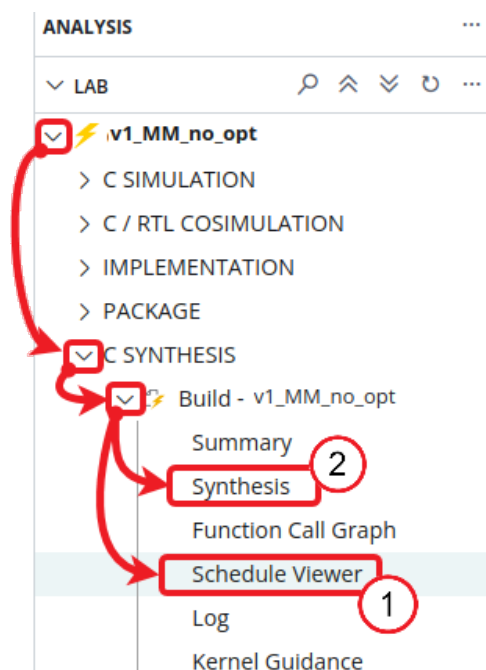
With the synthesis phase now complete, you will analyze the synthesis reports in more detail to help you understand and evaluate the performance of the implementation. These reports include the **Synthesis Summary** report, **Schedule Viewer**, **Function Call Graph**, and **Dataflow Viewer**.

These reports are accessed from the **Flow** view:

- **Schedule Viewer**: Shows each operation and control step of the function and the clock cycle that it executes in.
- **Dataflow Viewer**: Shows the dataflow structure inferred by the tool, allowing you to inspect the channels (FIFO/PIPO) and examine the effect of channel depth on performance.
- **Function Call Graph Viewer**: Displays your full design after C synthesis or C/RTL cosimulation to show the throughput of the design in terms of latency and initiation interval (II).

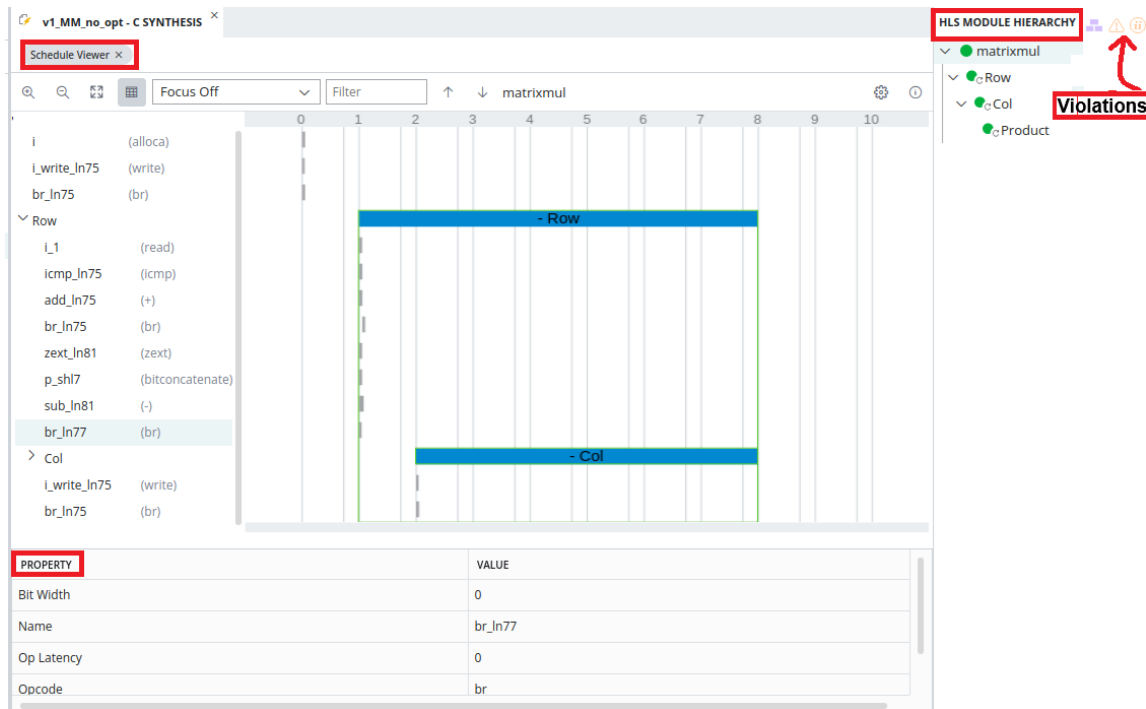
4.1 Schedual Viewer

- Select **View > Analysis** from the toolbar to open the **Schedule Viewer**.
- When the Analysis view opens, expand **v1_MM_no_opt > C Synthesis > Build-v1_MM_no_opt** and click **Schedule Viewer** (1).
- Similarly, click **Synthesis** (2)



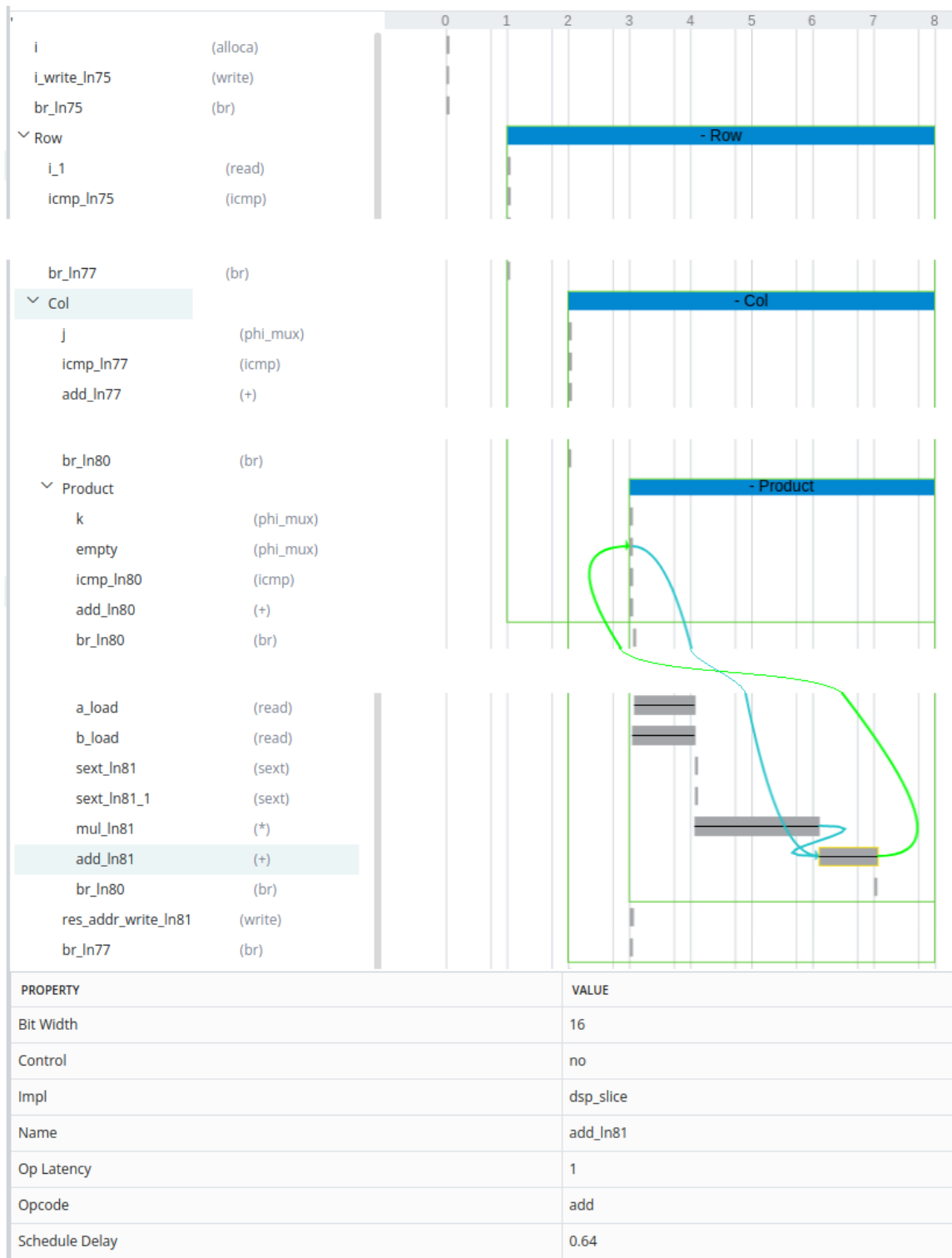
When the Schedule Viewer opens, the following windows appear:

- **Schedule Viewer**: Graphically represents how data moves through the design.
- **HLS Module Hierarchy**: Shows the function hierarchy and the performance characteristics of the current hierarchy.
- **Property**: Shows the properties of the currently selected control step or operation in the Schedule Viewer.

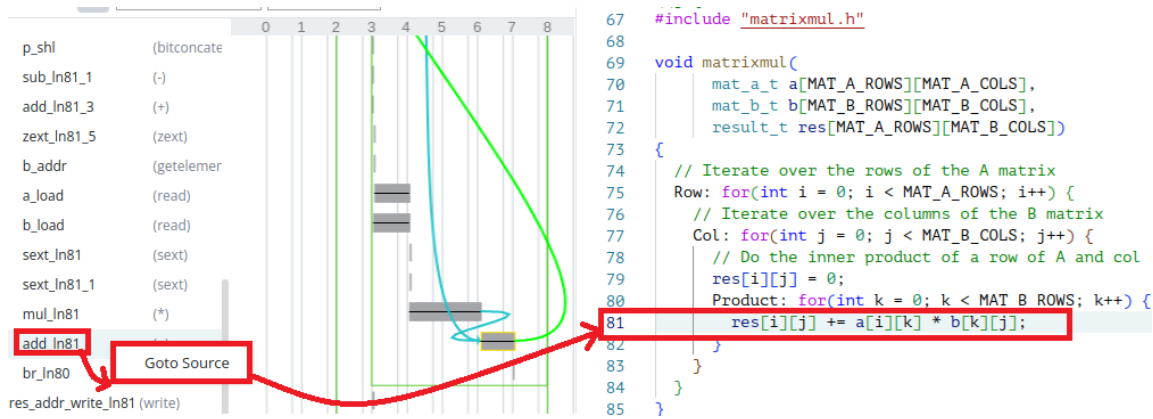


On the far right, the HLS Module Hierarchy window is displayed alongside the Schedule Viewer to let you quickly navigate through the hierarchy of the design. The HLS Module Hierarchy window provides an overview of the entire RTL design. It shows the resources and latency contribution for each block in the RTL hierarchy. This view directly indicates any II or timing violations. In case of timing violations, this hierarchy window will also show the total negative slack observed in a specific module.

- The top horizontal axis shows the clock cycles in consecutive order.
- The vertical dashed line in each clock cycle shows the reserved portion of the clock period due to clock uncertainty. This time is left by the tool for Vivado Design Suite backend processes, like place and route.
- Each operation is shown as a gray box in the table. The box is horizontally sized according to the delay of the operation as a percentage of the total clock cycle. For function calls, the provided cycle information is equivalent to the operation latency.
- Multicycle operations are shown as gray boxes with a horizontal line through the center of the box.
- The Schedule Viewer also displays general operator data dependencies as solid blue lines. As shown in the following figure, when selecting an operation, you can see solid blue arrows highlighting the specific operator dependencies. This gives you the ability to perform detailed analysis of data dependencies. A green dotted line indicates an inter-iteration data dependency. Memory dependencies are displayed using golden lines.



From above figure we can notice that multiply `mul_ln81` and addition `add_ln81` operations are performed inside Product loop which relates to multiply and accumulate operation of Product loop. We can confirm this. Right click `add_ln81` operations and select Goto Source. The source code pane will be opened, highlighting line 81 where accumulation is performed.



Similarly, you can observe that in the first cycle of the Row the loop exit condition is checked and there is an add operation performed. This addition is likely the counter to count the loop iterations, and we can confirm this. Right click on `add_in75(+)`, and select Goto Source. The source code pane will be opened, highlighting line 75 where the Row loop index is being tested and incremented. In the next state (C2) it starts to execute the Col loop (indicated by the blue block with the text -Col).

4.2 Function Call Graph

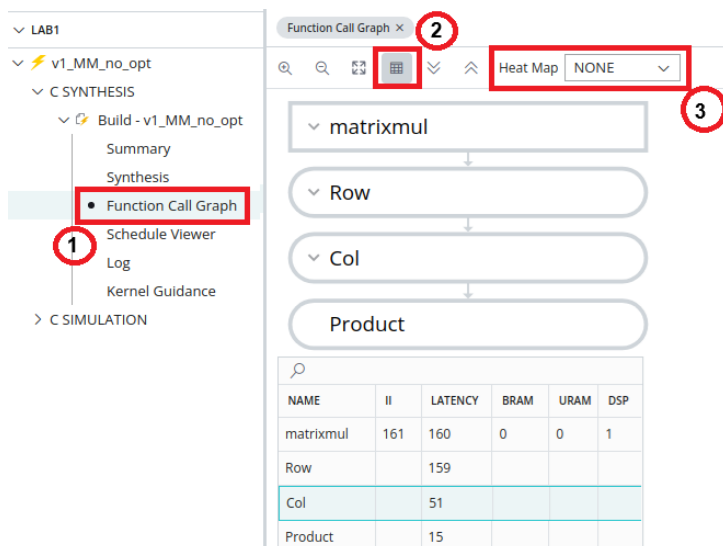
Function Call Graph shows the throughput of the design in terms of latency and initiation interval (II) after C synthesis against each hierarchy level.

- From the Analysis view, select Function Call Graph under **v1_MM_no_opt > C Synthesis > Build-v1_MM_no_opt > Function Call Graph** (1).
- Ensure that the Toggle Table icon is enabled in order to view design analysis values (2).
- Observe the latency and II values for each function and loop.
- Click the Heat Map drop-down arrow and try changing the heat map options and observing the effects (3).

The heat map feature highlights several metrics of interest; II (min, max, avg), Latency (min, max, avg), Stalling Time Percentage.

The heat map uses color coding to highlight problematic modules with a color scale of red to green, where red indicates a high value (bad) of the metric (that is, the highest initiation interval or highest latency) while green (good) indicates a low value of the metric in question. The colors that are neither red nor green represent the range of values that are in between the highest and lowest values.

- Close the window once done.



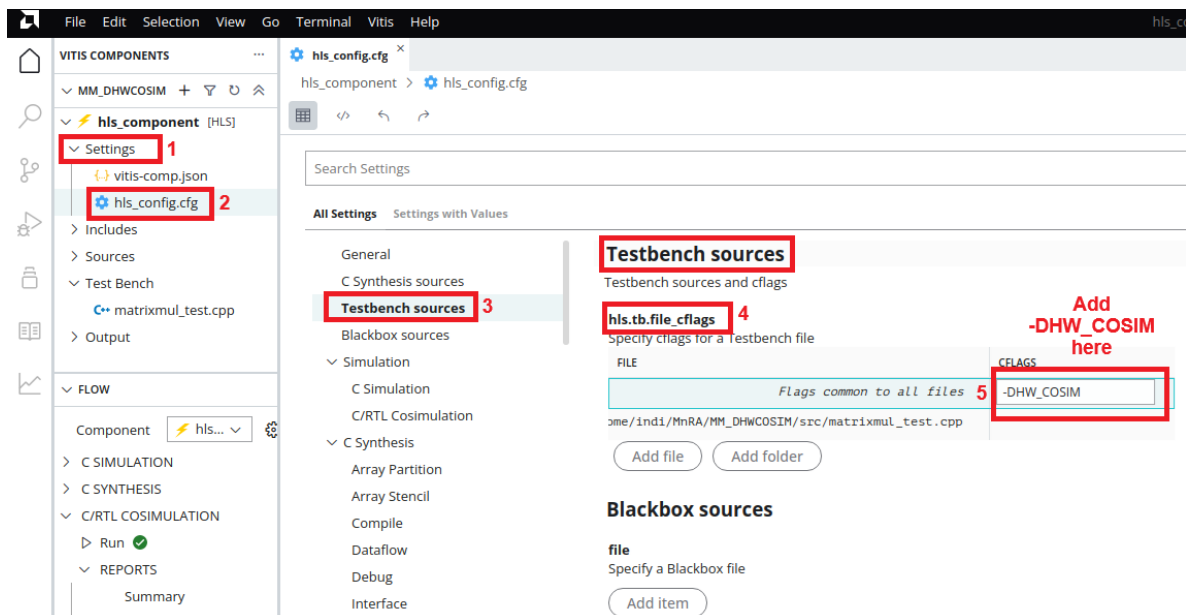
Step 5: Performing C/RTL Cosimulation

Having conducted high-level synthesis on the C design, you will perform RTL cosimulation on the generated RTL using the C test bench. Cosimulation (or more accurately to what you are doing here, simulation verification) compares the behavior of the software (C/C++ code) with the behavior of the RTL code. The hope is that identical results are produced.

You will now run the C/RTL cosimulation using Verilog. This could also be done using VHDL; however, for the sake of time, only one flow is shown. At the end of this process, you want to see that both simulations generate the same results.

If you missed adding any CFLAG (required for co-simulation) during the component creation step, you can add it later by editing the configuration settings file. Our test bench contains a flag (`-DHW_COSIM`) to control the invocation of the `matrixmul()` function during hardware co-simulation. In case you have not added the `-DHW_COSIM` flag, you can follow these steps to add it now:

1. From the Component view, go to **settings** -> **hls_config.cfg**.
2. The **hls_config.cfg** file will open in the information pane.
3. Click on the **Testbench sources**.
4. Add the flag `-DHW_COSIM` in the **CFLAG** field.

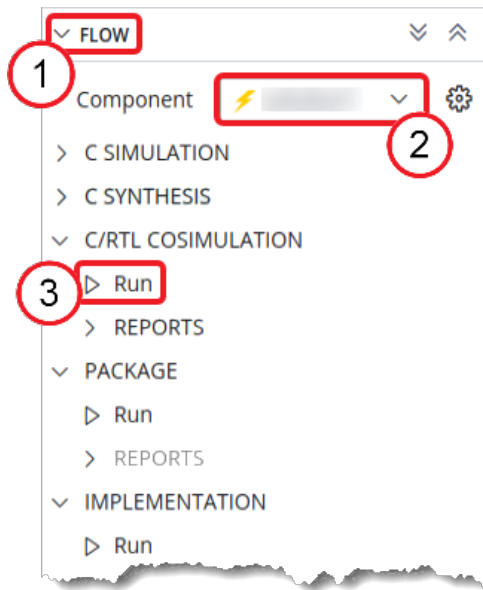


5.1 Run C/RTL cosimulation

- a. Select the **Vitis Components** option in the toolbar to return to the Vitis Components view.
- b. Go to the **Flow** view (1).
- c. Select the **active HLS component** (`v1_MM.no_opt`) to simulate from the Component drop-down list (2).
- d. Click **Run** under C/RTL Cosimulation (3).

You can view the simulation log in the Output tab.

Note: The Console pane shows the message "Test Passed", which is generated by the test bench C code.



The C/RTL Co-simulation will run, generating and compiling several files, and then simulating the design. It goes through three stages.

- First, the VHDL test bench is executed to generate input stimuli for the RTL design.
- Second, an RTL test bench with newly generated input stimuli is created and the RTL simulation is then performed.
- Finally, the output from the RTL is re-applied to the VHDL test bench to check the results.

In the console window you can see the progress and also a message that the test is passed. This eliminates writing a separate testbench for the synthesized design.

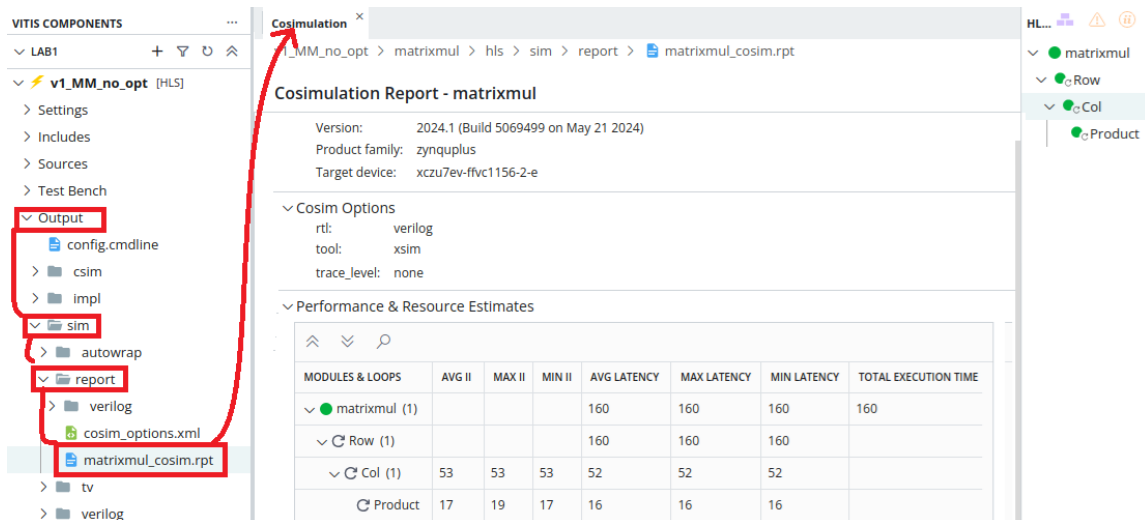
5.2 View the Co-simulation report

The information generated by the Vitis Unified IDE can be found in two places, both described here. The first is the **Output tab**, which reports not only the output produced by the code being simulated but all the simulation engine messages as well.

The second is the **simulation log**, which provides only a few simulation engine messages and the simulated code output.

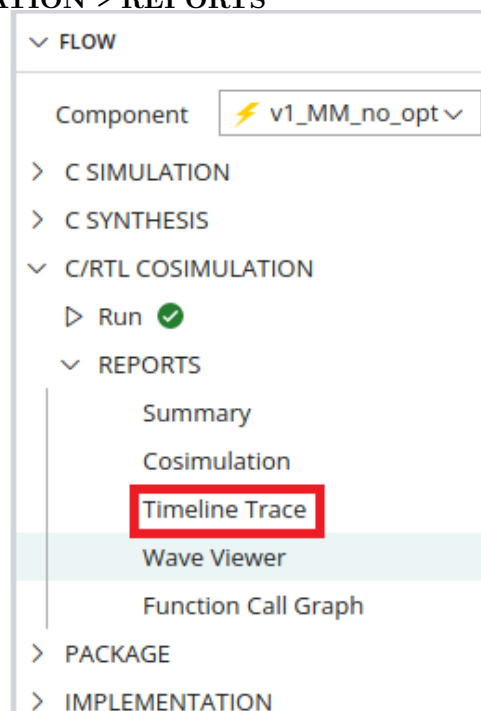
- Option-1: Select the **Output** tab to see the simulation messages.
- Option-2: Expand `v1_MM_no_opt [HLS]` > Output > sim > report in the Vitis Components window.
- Click `matrixmul_cosim.rpt` to open it.
Alternatively, the report can also be found in the Flow window under C/RTL COSIMULATION > REPORTS > Cosimulation.

The Cosimulation report will be displayed in the main viewing area. You can quickly verify the cosimulation status here.

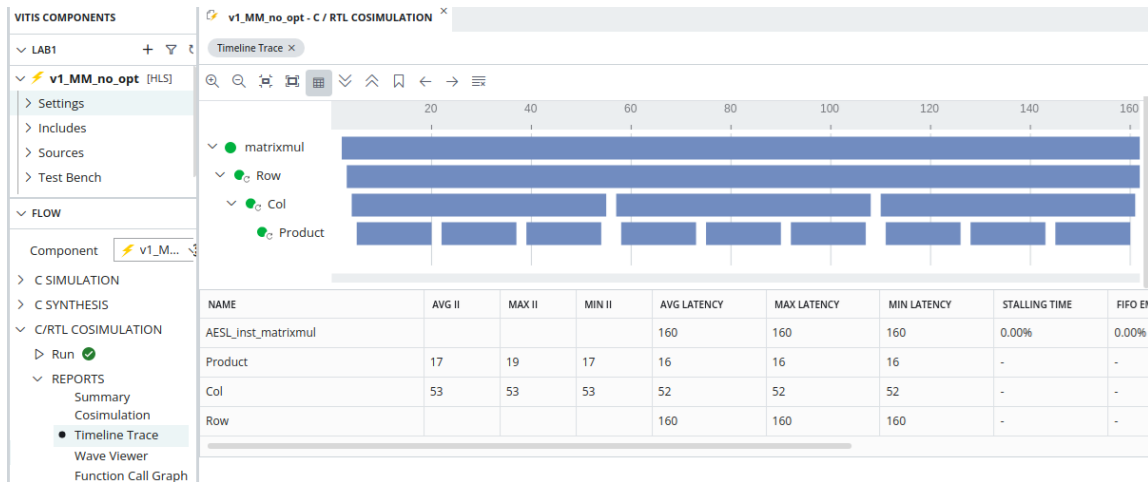


5.3 Timeline Trace Viewer

The Timeline Trace report is available after C/RTL Co-Simulation. The Timeline Trace viewer displays the runtime profile of the functions of your design. It is especially useful to see the behavior of dataflow regions after Co-simulation, as there is no need to launch the Vivado logic simulator to view the timeline. Once the RTL co-simulation is completed, click on **Timeline Trace** under **C/RTL COSIMULATION > REPORTS**



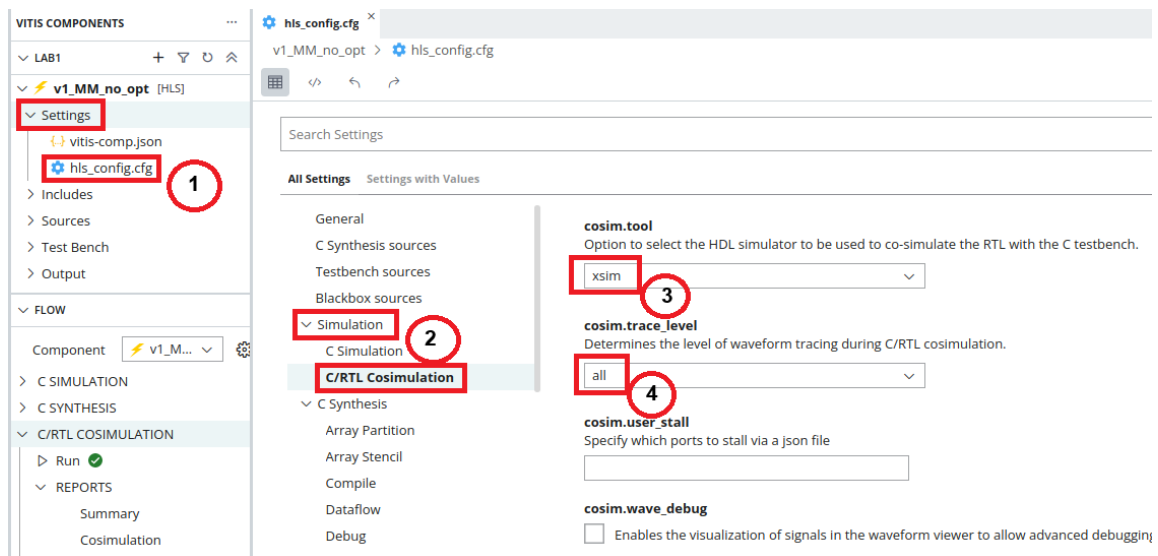
Timeline Trace viewer displays multiple iterations through the various sub-functions of a dataflow region as shown in the preceding figure. It shows where the functions are starting and ending, and displays the Co-simulation data in tables below the timeline.



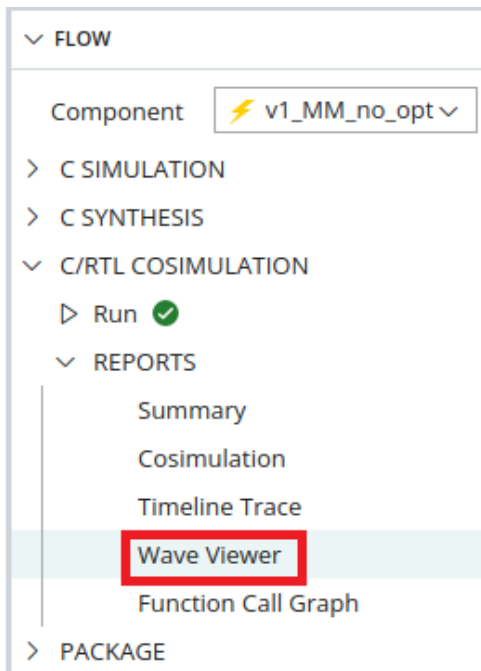
5.4 Analyze the dumped traces

To view waveform data during RTL co-simulation, enable the following in the hls_config file:

- Under `cosim.tool` select `xsim` as the RTL simulator.
- Under `cosim.trace_level` all.



Once the RTL co-simulation is completed, click on **Wave Viewer** under **C/RTL COSIMULATION > REPORTS**

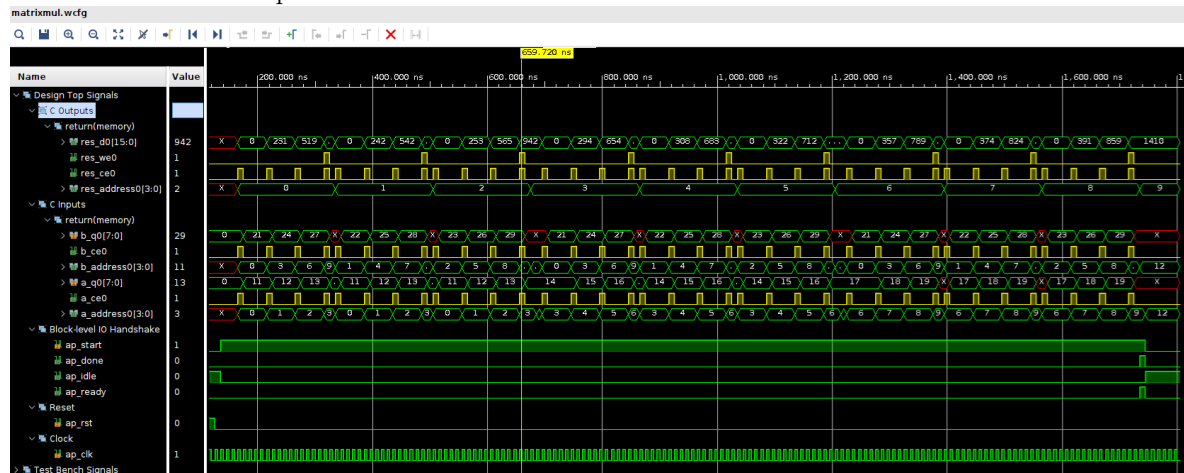


This will start Vivado 2024.1 and open the wave viewer.

NOTE: It can take a while before the start screen of Vivado opens, do not press the button multiple times.

- In the waveform window, expand the Design Top Signals as needed.
- Click on the zoom fit tool button (X) to see the entire simulation of one iteration.
- Select `a_address0` in the waveform window, right-click and select Radix to Unsigned Decimal. Similarly, do the same for `b_address0` and `res_address0` signals.
- Similarly, set the `a_q0`, `b_q0`, and `res_d0` radix to Signed Decimal.

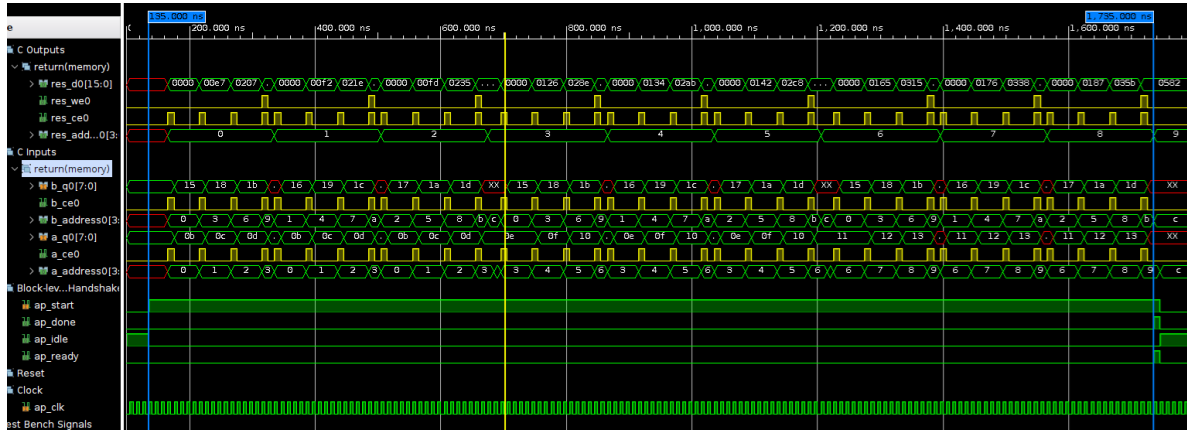
You should see the output similar to shown below.



Observe the following:

- As soon as `ap_start` is asserted, `ap_idle` has been de-asserted indicating that the design is in computation mode.
- The `ap_idle` signal remains de-asserted until `ap_done` is asserted, indicating completion of the process.

- c. Observe that the design expects element data by providing a_address0, a_ce0, b_address0, b_ce0 signals and outputs the result using res_d0, res_we0, and res_ce0.
- d. Observe the 160 clock cycles latency and 161 cycles of interval.
ap_start is asserted at: 135.000 ns
ap_done is asserted at: 1735.000 ns
Total Time taken: 1600.000 ns which corresponds to 160 cycles @ 10 ns clock.
The design will be able to accept next set of inputs on next rising edge after ap_done is asserted.
which makes the Interval 161 cycle.



View various part of the simulation and try to understand how the design works.

When done, close Vivado by selecting File > Exit. Click OK if prompted, and then Discard to close the program without saving.

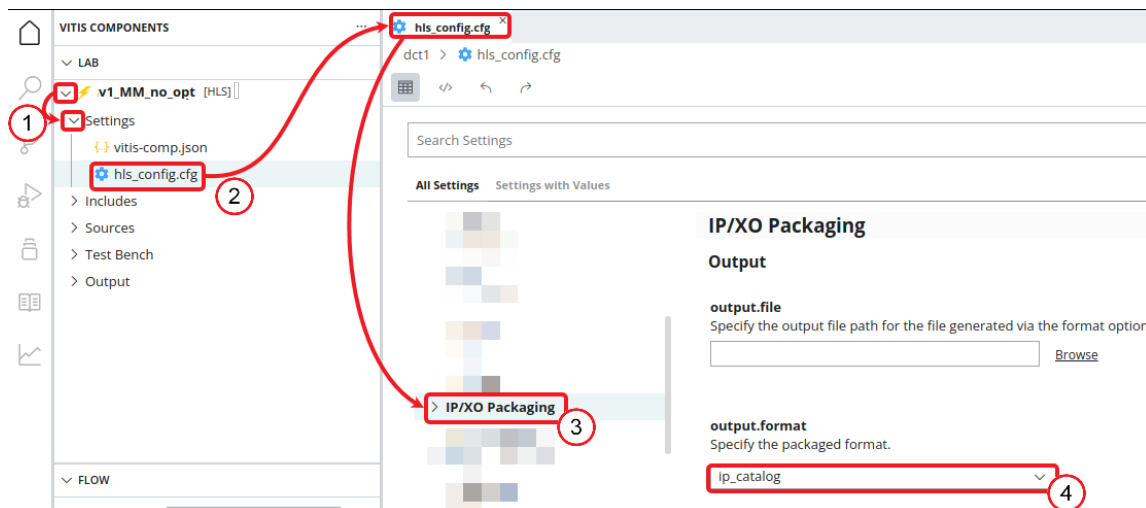
Step 6: Converting the Design into an IP Core

The final step in the HLS component flow is to package the RTL design into a form that can be used by other tools in the design flow, such as in the Vivado Design Suite as part of a larger system design.

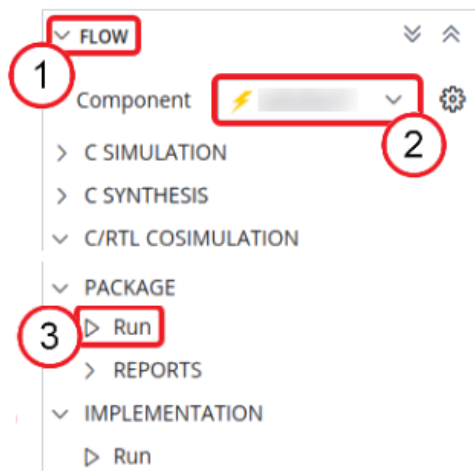
6.1 Packaging the RTL Design

Explore the packaging configuration settings.

- a. Expand v1_MM_no_opt [HLS] > Settings in the Vitis Components window (1).
- b. Click hls_config.cfg to open the configuration settings (2).
- c. Locate and select **IP/XO Packaging** to open the configuration settings for packaging (3).
- d. Ensure that **ip_catalog** is selected from the **output.format** drop-down list (4).
- e. Other settings can be left at their default



- Once you are ready, close the configuration file.
- Go to the **Flow** view (1).
- Select the active HLS component to package from the Component drop-down list (2).
- Click **Run** under Package (3).



You can view the log in the Output tab. High-level synthesis is limited in terms of the estimations it can provide about the RTL design that it generates. It can project resource utilization and timing of the end result, but these are just projections.

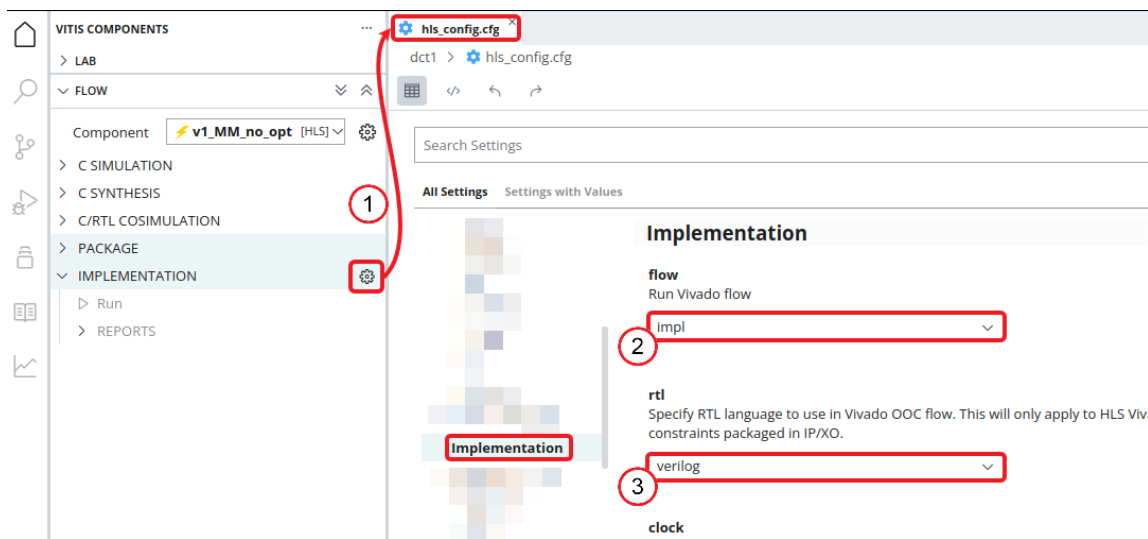
To get a better view of the RTL design, you can actually run Vivado synthesis and place and route on the generated RTL design and review the actual results of timing and resource utilization.

6.2 Implementation

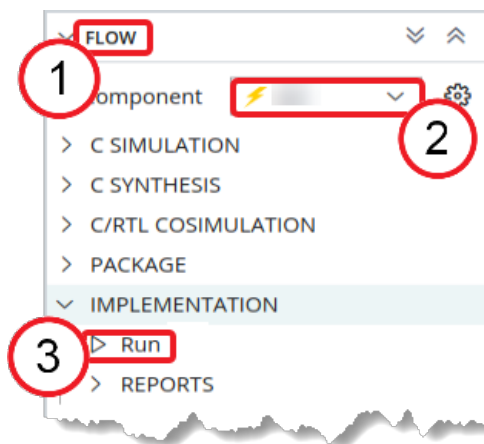
When the HLS compiler reports the results of high-level synthesis, it provides an estimate of the results with projected clock frequencies and resource utilization (LUTs, DSPs, BRAMs, etc). These results are only estimates because the tool cannot know what optimizations or routing delays will be in the final synthesized or implemented design. To get a better analysis of the RTL design you can actually run Vivado synthesis and place and route on the generated RTL design, and review the results of timing and resource utilization.

You can configure Vivado synthesis and implementation prior to running it by using the Implementation section of the Config File.

- a. From the Flow view, hover over **Implementation** and click the **Settings** icon (gear like) that appears (1).
The hls_config.cfg configuration file opens in the Information pane.
- b. Ensure that **impl** is selected from the flow drop-down list (2).
This setting specifies whether to run only synthesis or both synthesis and implementation. Synthesis alone will run faster than both synthesis and implementation but will lack some of the details of the implementation run.
- c. Ensure that **verilog** is selected from the rtl drop-down list (3).
This specifies the language to use when running a Vivado out-of-context flow you can also use VHDL.
- d. The other settings can be left at their default.



- a. Once you are ready, close the configuration file.
- b. Go to the **Flow** view (1).
- c. Select the active HLS component to simulate from the Component drop-down list (2).
- d. Click **Run** under Implementation (3).



You can view the implementation log in the Output tab.

Note: Implementation will take 5-7 minutes to complete.

This will perform Vivado synthesis and implementation on the generated IP. Implementation is run

to evaluate and provide confidence that the RTL will meet its estimated timing and area goals and that these results are not included as part of the exported package.

When the run is complete, the Implementation report is displayed in the Information pane. The final timing of the implemented design has been achieved.

Look for the "Timing met" and "Implementation finished successfully" messages in the **Output** tab and under the Final Timing report section.

The exported IP is available in the Output > impl > ip folder in the Vitis Components window.

- e. Expand the **Output > impl > ip** folder in the Vitis Components tab.
Observe the various generated folders and files (IP, hardware files, software drivers, and IP documentation).
- f. Expand **dct > Output > impl > report**.
- g. Click **export_impl.rpt** to open the Implementation report in the Information pane.
- h. Go to the **Fail Fast** section of the Implementation report.
The Fail Fast report provided by Vivado Design Suite can guide your investigation into specific issues encountered by the tool.
In the Fail Fast report, you should look into anything with the status of REVIEW to improve implementation and timing closure.

VITIS COMPONENTS

- LAB1
 - v1_MM_no_opt [HLS]
 - Settings
 - Includes
 - Sources
 - Test Bench
 - Output
 - config.cmdline
 - csim
 - hls_data.json
 - impl
 - ip
 - misc
 - report
 - export_impl.rpt
 - export_impl.xml

Implementation (Place & Route) x

v1_MM_no_opt > matrixmul > hls > impl > report > vhd > export_impl.rpt

Implementation Report (Place & Route) - matrixmul

Fail Fast

Created on Tue Feb 18 15:50:24 CET 2025 with report_failfast (2023.09.14)

Design Summary

impl_1

xczu7ev-ffvc1156-2-e

CRITERIA	GUIDELINE	ACTUAL	STATUS
LUT	70%	0.04%	OK
FD	50%	0.02%	OK
LUTRAM+SRL	25%	0.00%	OK
CARRY8	25%	0.03%	OK
MUXF7	15%	0.00%	OK
DSP	80%	0.00%	OK
RAMB/FIFO	80%	0.00%	OK
URAM	80%	0.00%	OK

- i. Go to the **Timing Paths** section of the Implementation report.
The Timing paths report shows the timing-critical paths that result in the worst slack for the design. By default, the tool will show the top 10 worst negative slack paths.
Each path in the table has detailed information that shows the combination path between one flip-flop to another. Breaking these long combinational paths may be required to address the timing issues.

VITIS COMPONENTS

- LAB1
 - v1_MM_no_opt [HLS]
 - Settings
 - Includes
 - Sources
 - Test Bench
 - Output
 - config.cmdline
 - csm
 - hls_data.json
 - impl
 - ip
 - misc
 - report
 - vhdl
 - export_impl.rpt
 - export_impl.vml

Implementation (Place & Route)

v1_MM_no_opt > matrixmul > hls > impl > report > vhdl > export_impl.rpt

Implementation Report (Place & Route) - matrixmul

Timing Paths

Worst Negative Slack: 7.700ns(met)

Total Negative Slack: 0.000ns(met)

Max levels: 7

Max fanout: 19

Full Timing Report: vhdl/report/matrixmul_timing_routed.rpt

NAME	VALUE
> Path 1 (4)	slack=7.700ns(met) levels=7 fanout=19
> Path 2 (4)	slack=7.707ns(met) levels=7 fanout=19
> Path 3 (4)	slack=7.715ns(met) levels=7 fanout=19
> Path 4 (4)	slack=7.727ns(met) levels=7 fanout=19
> Path 5 (4)	slack=7.798ns(met) levels=6 fanout=19

Note: Values may slightly vary depending upon your system.

6.3 Close the Vitis Unified IDE.

Select **File > Close Window** to close the tool.

Summary

You just created a piece of hardware IP from a C-program following the HLS component development flow in the Vitis Unified IDE. This included simulating, synthesizing, cosimulating, and packaging the design as an IP.

You examined some of the generated reports and determined how the design was implemented. Other labs will expand on what you have learned here.