Departamento de Engenharia Electrotécnica e de Computadores

Instituto Superior Técnico

Universidade de Lisboa

Electronic Systems of Computers
Hardware/Software Co-Design

Laboratory Guide

José T. de Sousa, Horácio Neto

April 2023

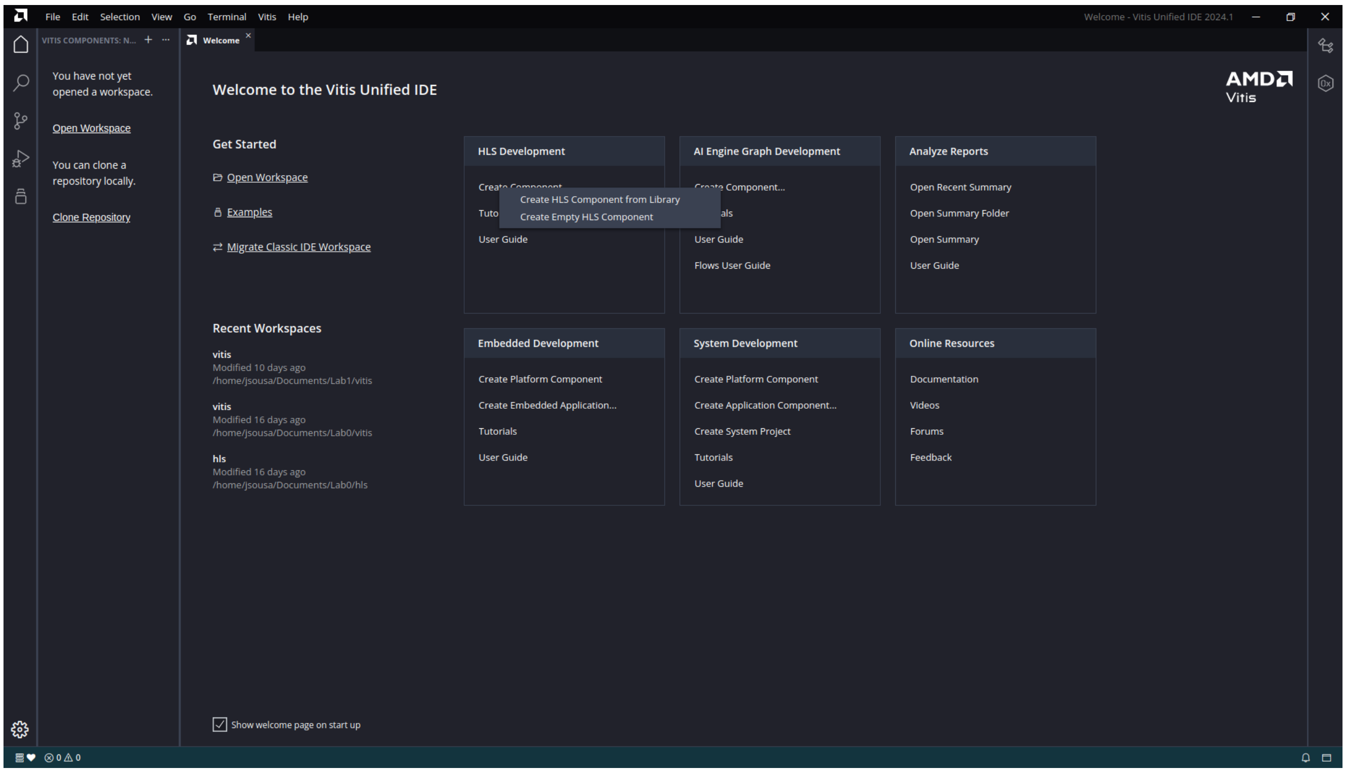
1. Introduction

The objective of this lab is to introduce you to the Xilinx Vitis+Vivado design tools and to the development boards to be used in the labs, by implementing a simple HW/SW system. (Note that the interface figures and results included in this guide were obtained using version 2021.2, so there may be some minor differences to the newer versions)

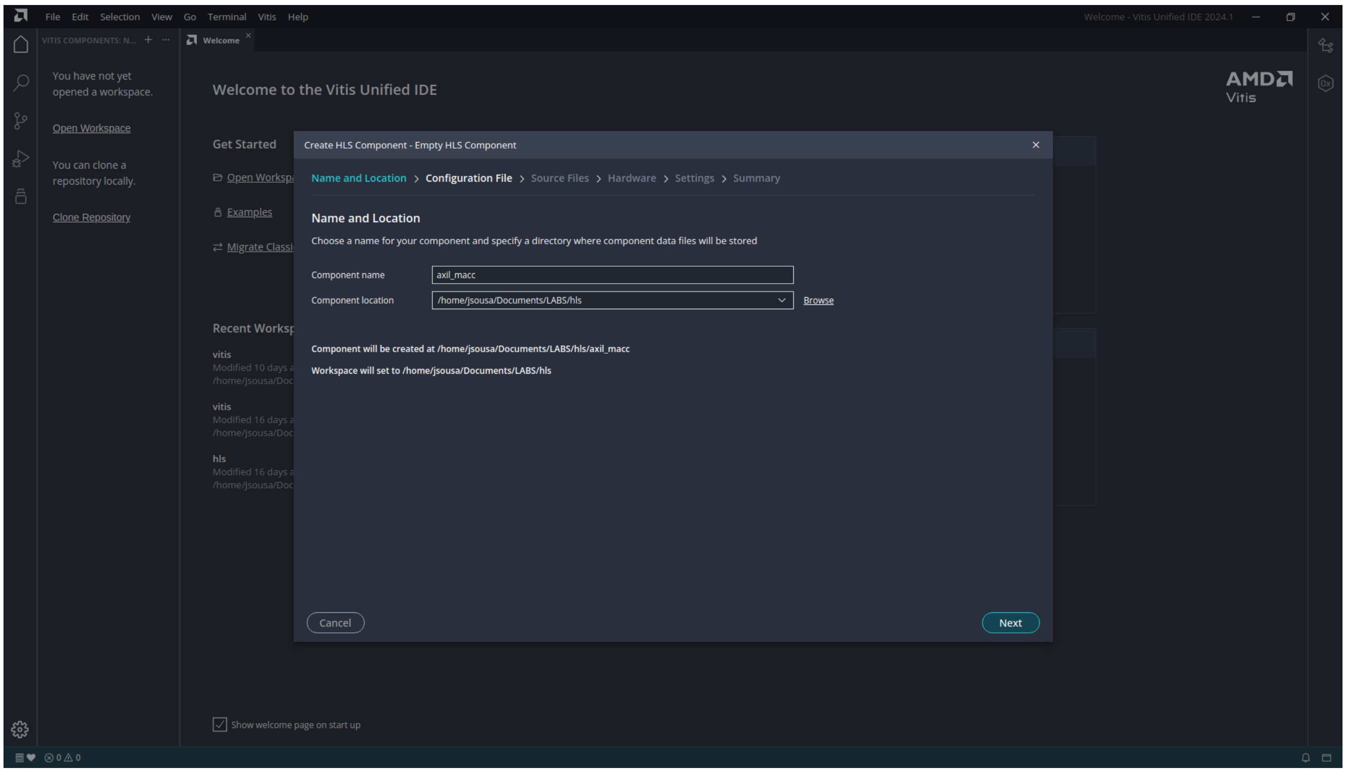
2. Design one AXI-Lite IP using HLS

2.1. Create a new component in Vitis targeting your device (or board)

1. Start the Vitis program from the desktop shortcut or shell and select HLS Development/Create Component/Create Empty HLS Component.


File Edit Selection View Go Terminal Vitis Help
Welcome - Vitis Unified IDE 2024.1
VITIS COMPONENTS: N...
Welcome
You have not yet opened a workspace.
Open Workspace
You can clone a repository locally.
Clone Repository
Welcome to the Vitis Unified IDE
AMD Vitis
Get Started
Open Workspace
Examples
Migrate Classic IDE Workspace
Recent Workspaces
vitis
Modified 10 days ago
/home/jsousa/Documents/Lab1/vitis
vitis
Modified 16 days ago
/home/jsousa/Documents/Lab0/vitis
hls
Modified 16 days ago
/home/jsousa/Documents/Lab0/hls
HLS Development
Create Component
Create HLS Component from Library
Create Empty HLS Component
Tuto
User Guide
AI Engine Graph Development
Create Component...
als
User Guide
Flows User Guide
Analyze Reports
Open Recent Summary
Open Summary Folder
Open Summary
User Guide
Embedded Development
Create Platform Component
Create Embedded Application...
Tutorials
User Guide
System Development
Create Platform Component
Create Application Component...
Create System Project
Tutorials
User Guide
Online Resources
Documentation
Videos
Forums
Feedback
Show welcome page on start up
0 0

2. Choose the component name and location and press Next.


File Edit Selection View Go Terminal Vitis Help
Welcome - Vitis Unified IDE 2024.1
Welcome
AMD Vitis
Open Workspace
Clone Repository
Get Started
Recent Workspaces
Show welcome page on start up
Cancel
Next

Welcome to the Vitis Unified IDE

You have not yet opened a workspace.

You can clone a repository locally.

Create HLS Component - Empty HLS Component

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

Name and Location

Choose a name for your component and specify a directory where component data files will be stored

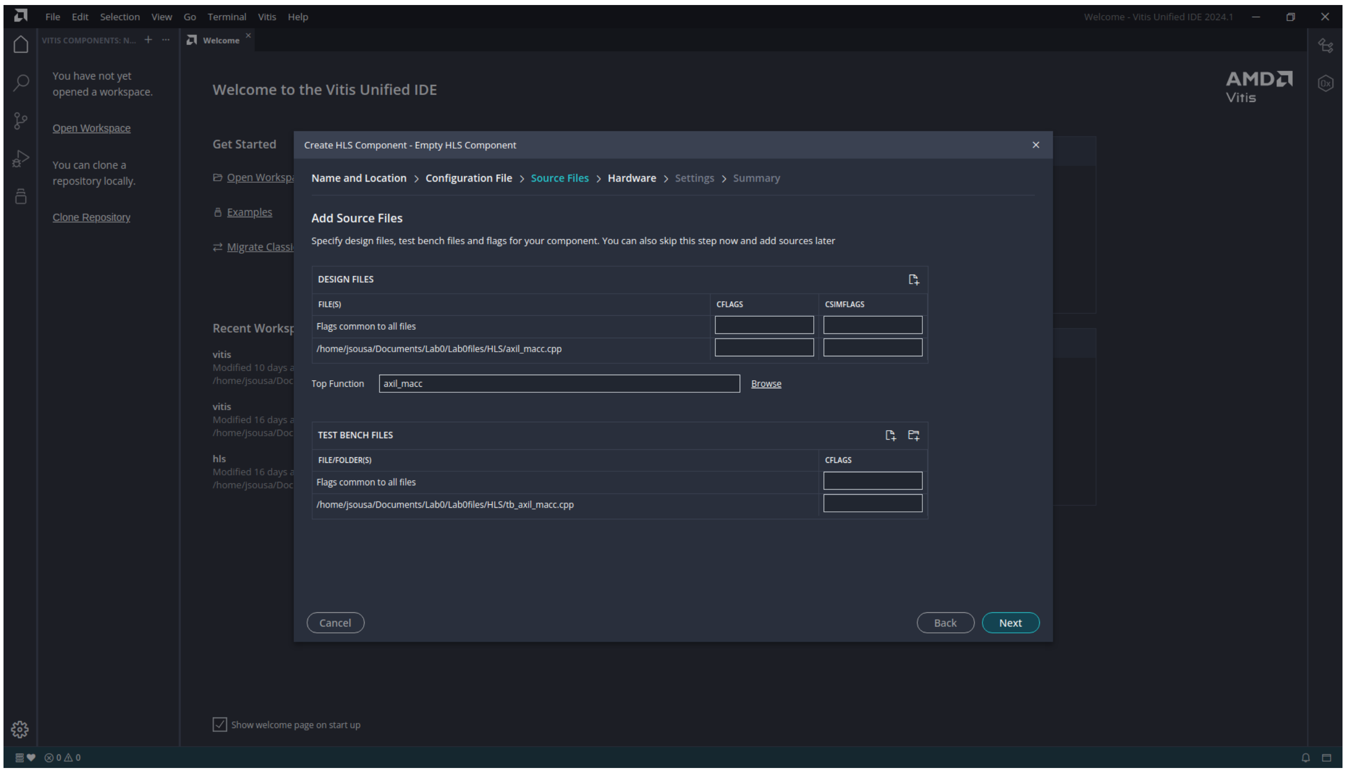
Component name: axil_macc

Component location: /home/jsousa/Documents/LABS/hls

Component will be created at /home/jsousa/Documents/LABS/hls/axil_macc

Workspace will set to /home/jsousa/Documents/LABS/hls

3. In the Configuration File tab choose Empty File, leave the default configuration file name and press Next.
4. In the Source Files tab, add axil_macc.cpp as design file tb_axil_macc.cpp as testbench file; press Browse to select the top function and press Next.


File Edit Selection View Go Terminal Vitis Help
Welcome - Vitis Unified IDE 2024.1
Welcome
AMD Vitis
Browse
Cancel
Back
Next

Create HLS Component - Empty HLS Component

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

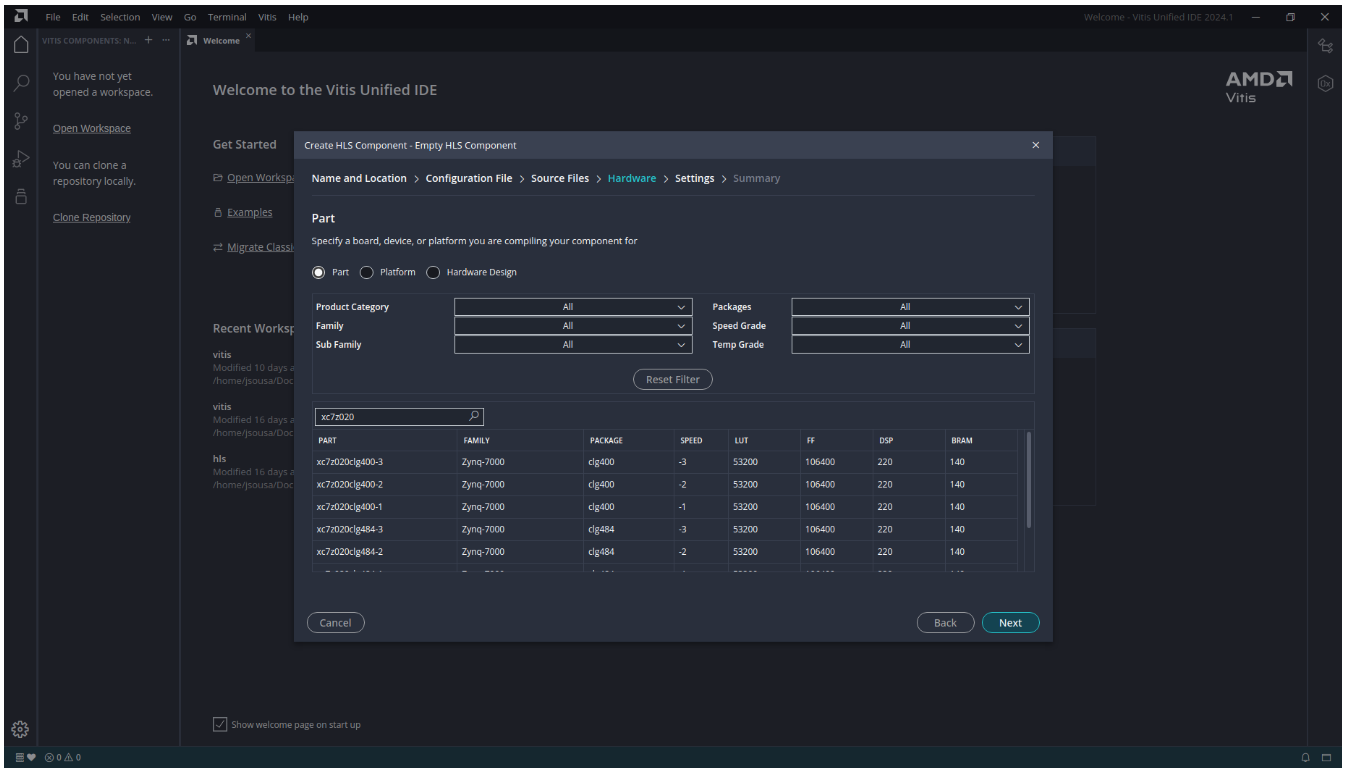
FILE(S)	CFLAGS	CSIMFLAGS
Flags common to all files		
/home/jsousa/Documents/Lab0/Lab0files/HLS/axil_macc.cpp		

Top Function: axil_macc

TEST BENCH FILES

FILE/FOLDER(S)	CFLAGS
Flags common to all files	
/home/jsousa/Documents/Lab0/Lab0files/HLS/tb_axil_macc.cpp	

5. In the Hardware tab, search and select the part xc7z020clg400-1 according to the following figure and press Next.



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

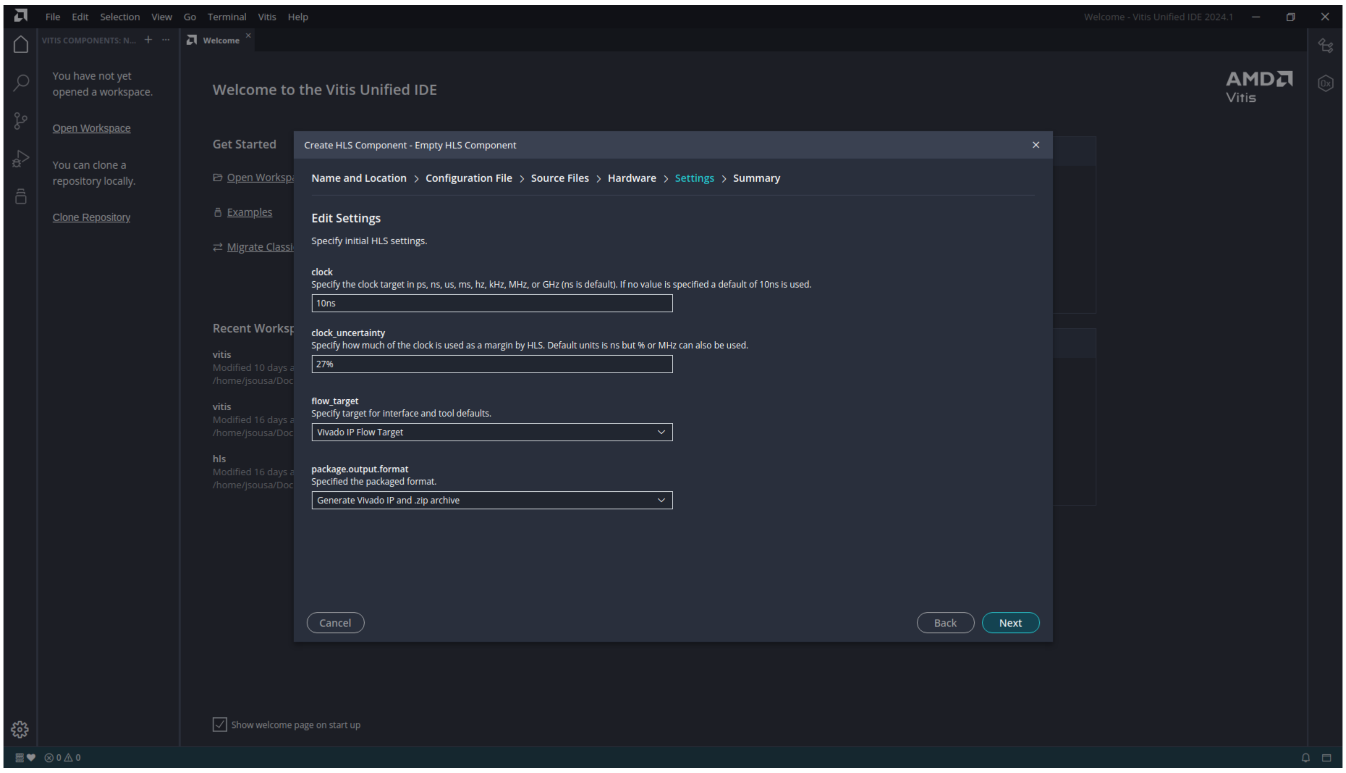
[✓ Part] Platform Hardware Design

Product Category	All	Packages	All
Family	All	Speed Grade	All
Sub Family	All	Temp Grade	All

xc7z020

PART	FAMILY	PACKAGE	SPEED	LUT	FF	DSP	BRAM
xc7z020clg400-3	Zynq-7000	clg400	-3	53200	106400	220	140
xc7z020clg400-2	Zynq-7000	clg400	-2	53200	106400	220	140
xc7z020clg400-1	Zynq-7000	clg400	-1	53200	106400	220	140
xc7z020clg484-3	Zynq-7000	clg484	-3	53200	106400	220	140
xc7z020clg484-2	Zynq-7000	clg484	-2	53200	106400	220	140

6. In the settings tab, choose the settings according to the following figure and press Next.



Create HLS Component - Empty HLS Component

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

Edit Settings

Specify initial HLS settings.

clock
Specify the clock target in ps, ns, us, ms, hz, kHz, MHz, or GHz (ns is default). If no value is specified a default of 10ns is used.
10ns

clock_uncertainty
Specify how much of the clock is used as a margin by HLS. Default units is ns but % or MHz can also be used.
27%

flow_target
Specify target for interface and tool defaults.
Vivado IP Flow Target

package.output.format
Specified the packaged format.
Generate Vivado IP and .zip archive

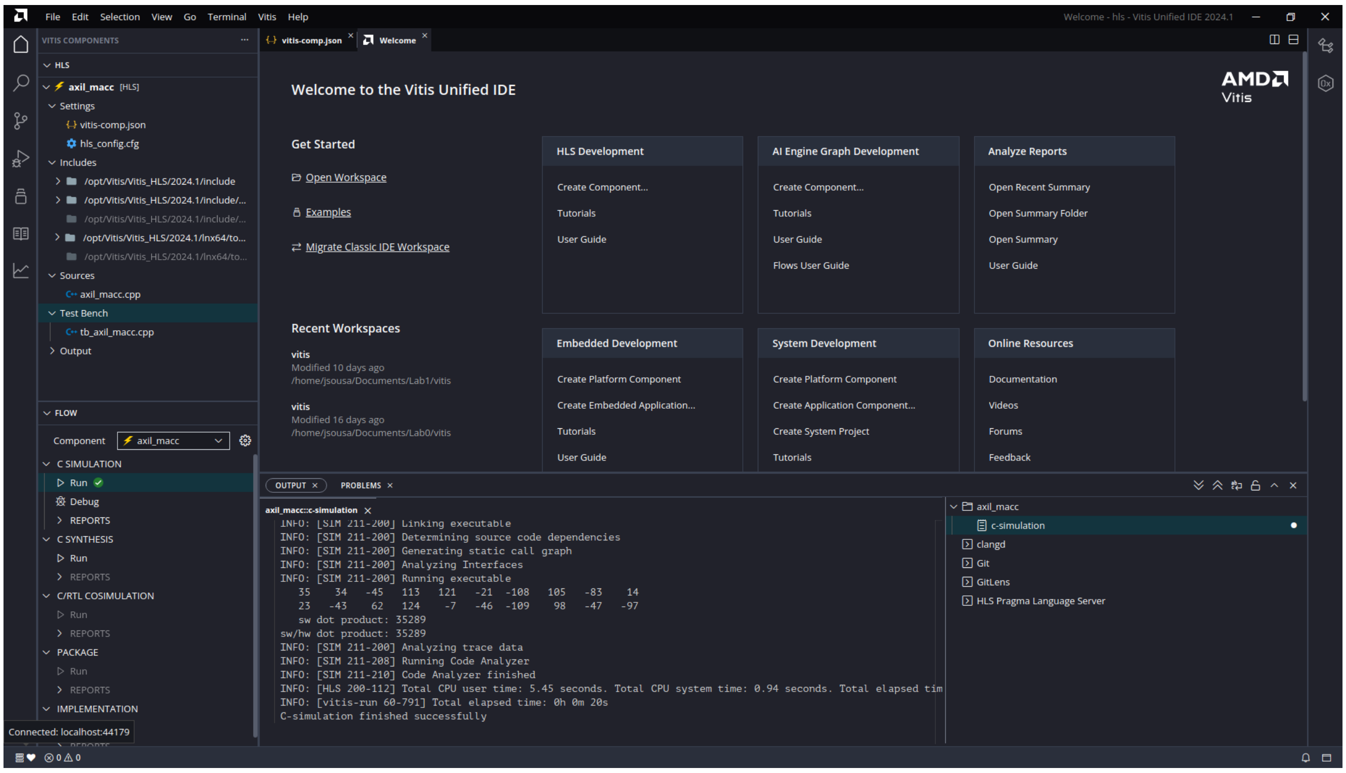
7. In the Summary tab, review your selections and press Finish.

2.2. Verify the functionality of the C-based Specification

The C simulation performs a pre-synthesis validation that the C-based specification correctly defines the intended functionality.

The C testbench includes the *main()* function (for simulation) and other (not for synthesis) sub-functions to help the verification. The testbench provides inputs to the *function to be synthesized* and checks that its results are functionally correct.

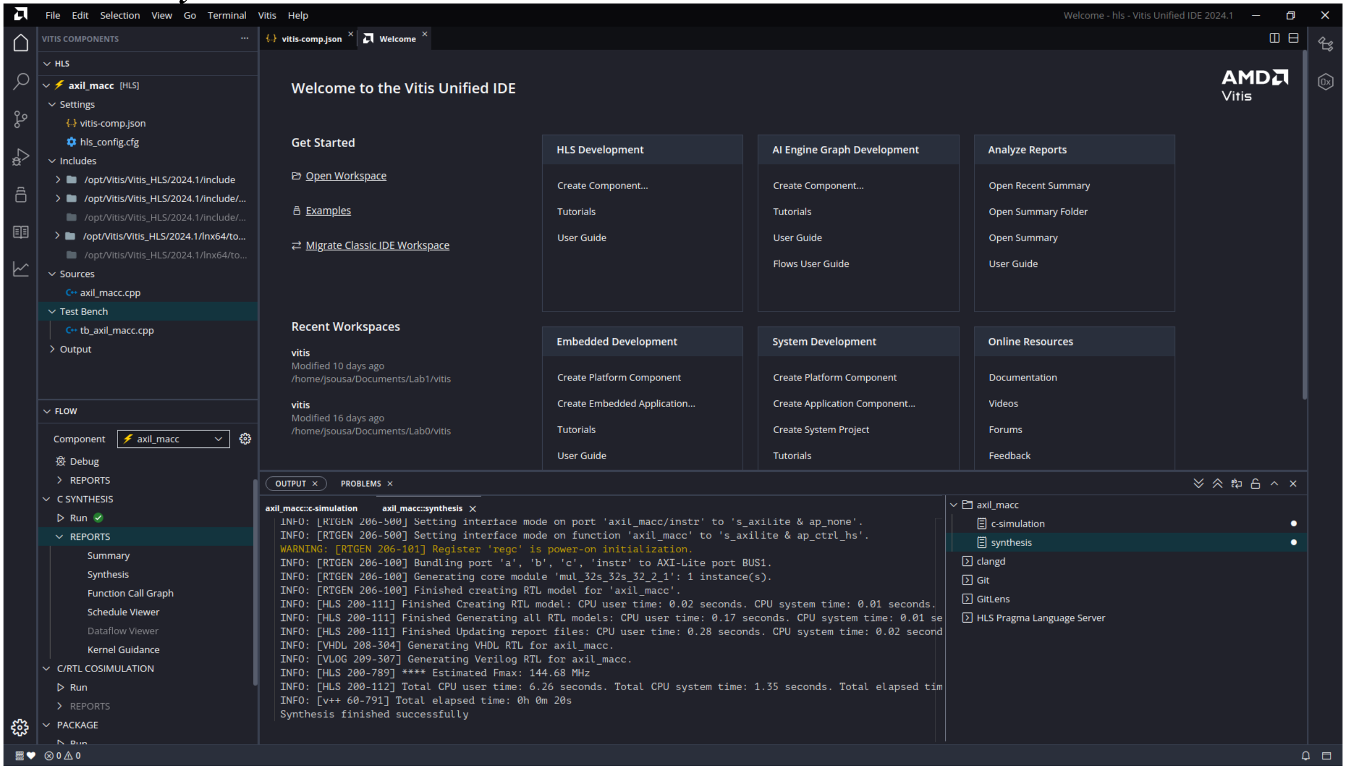
Run ***Flow→C Simulation →Run*** with default settings (no options selected) and check that the function to synthesize the hardware dot-product IP outputs the same result as the software checking function.



```
INFO: [SIM 211-200] Linking executable
INFO: [SIM 211-200] Determining source code dependencies
INFO: [SIM 211-200] Generating static call graph
INFO: [SIM 211-200] Analyzing Interfaces
INFO: [SIM 211-200] Running executable
   35    34   -45   113   121   -21  -108   105   -83    14
   23   -43    62   124    -7   -46  -109    98   -47   -97
   sw dot product: 35289
sw/hw dot product: 35289
INFO: [SIM 211-200] Analyzing trace data
INFO: [SIM 211-208] Running Code Analyzer
INFO: [SIM 211-210] Code Analyzer finished
INFO: [HLS 200-112] Total CPU user time: 5.45 seconds. Total CPU system time: 0.94 seconds. Total elapsed tim
INFO: [vitis-run 60-791] Total elapsed time: 0h 0m 20s
C-simulation finished successfully
```

2.3. Synthesize the Design

Run ***Flow→Synthesis →Run***



```
INFO: [RTGEN 206-500] Setting interface mode on port 'axil_macc/instr' to 's_axilite & ap_none'.
INFO: [RTGEN 206-500] Setting interface mode on function 'axil_macc' to 's_axilite & ap_ctrl_hs'.
WARNING: [RTGEN 206-101] Register 'regc' is power-on initialization.
INFO: [RTGEN 206-100] Bundling port 'a', 'b', 'c', 'instr' to AXI-Lite port BUS1.
INFO: [RTGEN 206-100] Generating core module 'mul_32s_32s_32_2_1': 1 instance(s).
INFO: [RTGEN 206-100] Finished creating RTL model for 'axil_macc'.
INFO: [HLS 200-111] Finished Creating RTL model: CPU user time: 0.02 seconds. CPU system time: 0.01 seconds.
INFO: [HLS 200-111] Finished Generating all RTL models: CPU user time: 0.17 seconds. CPU system time: 0.01 se
INFO: [HLS 200-111] Finished Updating report files: CPU user time: 0.28 seconds. CPU system time: 0.02 second
INFO: [VHDL 208-304] Generating VHDL RTL for axil_macc.
INFO: [VLOG 209-307] Generating Verilog RTL for axil_macc.
INFO: [HLS 200-789] **** Estimated Fmax: 144.68 MHz
INFO: [HLS 200-112] Total CPU user time: 6.26 seconds. Total CPU system time: 1.35 seconds. Total elapsed tim
INFO: [v++ 60-791] Total elapsed time: 0h 0m 20s
Synthesis finished successfully
```

Check the synthesis report, namely the performance estimates (estimated clock period and latency), the utilization estimates, and the interface summary.
Check the schedule view and understand which operations are executed in each clock cycle.

Note that the syn folder (under the hls directory) in the Explorer pane shows report, verilog, and vhdl sub-folders which contain the generated report and RTL hdl files.

2.4. C/RTL Cosimulation

The C/RTL co-simulation performs a post-synthesis (non-definitive) verification that the RTL design generated by HLS is functionally correct.

In the HLS/axil_macc/Settings/hls_config.cfg, look for the cosimulation settings and select trace_level=all and wave_debug.

Run ***Flow→C/RTL Cosimulation***

After the C/RTL cosimulation is complete, the RTL waveforms can be viewed in the Vivado IDE, where you should identify the simulated I/O transactions.

2.5. Generate the IP component

Use ***Package→Run to package the synthesized HW*** component as a standardized IP block (figure 8.a).

Note that when the RTL export finishes a new hls/impl/***ip*** sub-folder appears in the Explorer window, which includes the IP files.

The drivers' folder includes automatically generated C driver files that can be used to access the IP, when connected to a SW/HW system (as will be discussed later).

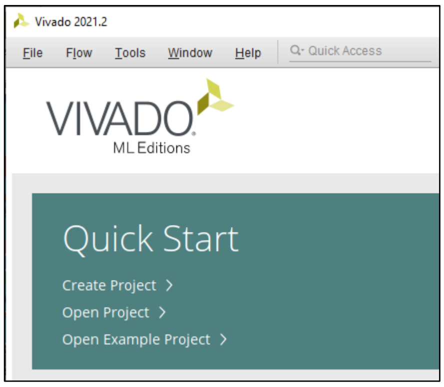
The ip folder contains the IP design that will be imported to the HW/SW design in Vivado.

3. Zynq Project in Vivado

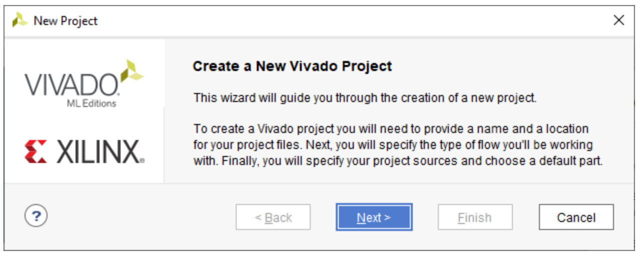
Start the Vivado application from the desktop shortcut or command line without closing Vitis.

3.1. Create New Project targeting the Pynq Z2 board

3.1.1. Open Vivado



3.1.2. Create New Project

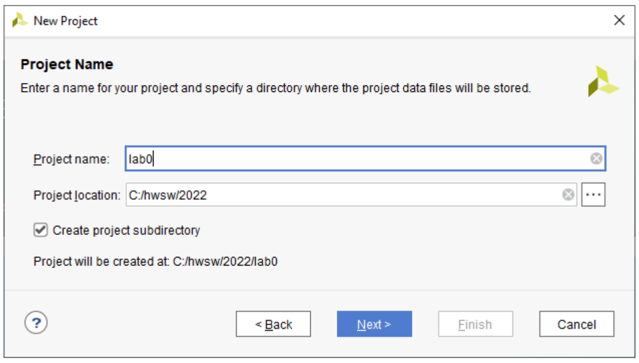


Create a New Vivado Project

This wizard will guide you through the creation of a new project.

To create a Vivado project you will need to provide a name and a location for your project files. Next, you will specify the type of flow you'll be working with. Finally, you will specify your project sources and choose a default part.

3.1.3. Select project name



Project Name

Enter a name for your project and specify a directory where the project data files will be stored.

Project name: lab0

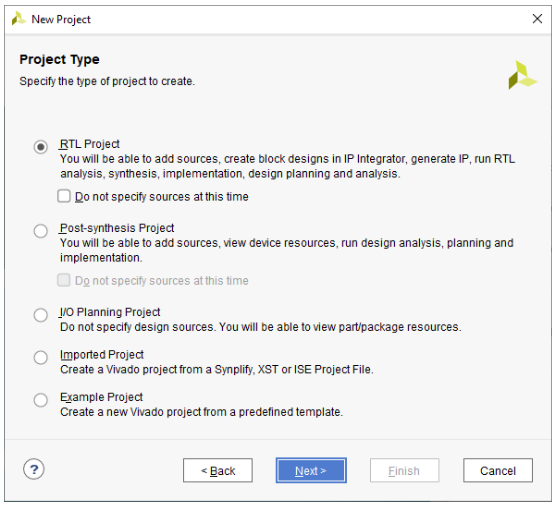
Project location: C:/hwsw/2022

☑ Create project subdirectory

Project will be created at: C:/hwsw/2022/lab0

Note: avoid using spaces and/or special characters on your directory path and on the filenames.

3.1.4. Select project type



(● RTL Project) You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.

☐ Do not specify sources at this time

○ Post-synthesis Project: You will be able to add sources, view device resources, run design analysis, planning and implementation.

☐ Do not specify sources at this time

○ I/O Planning Project: Do not specify design sources. You will be able to view part/package resources.

○ Imported Project: Create a Vivado project from a Synplify, XST or ISE Project File.

○ Example Project: Create a new Vivado project from a predefined template.

3.1.5. Skip Add Sources and Constraints

Click Next twice to skip Add Source and Add Constraints.

3.1.6. Board Selection

In the Default Part form, select Boards and then select the board you are using: Pynq-Z2, (check the version of your board).

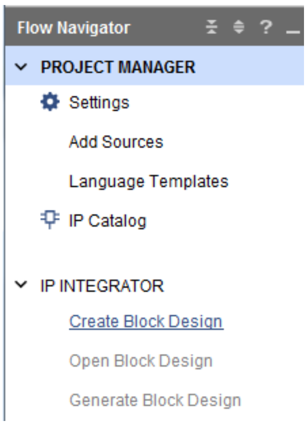
Note: if there is no Pynq-Z2 in the board list, go to Tools/Vivado Store to install it.

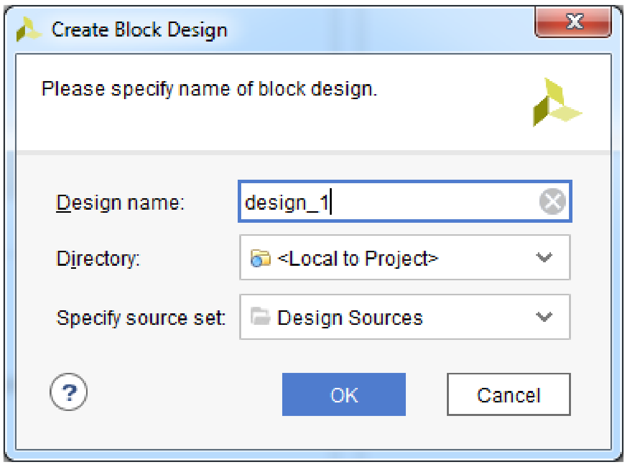
Click Next, check the Project Summary and click Finish to create an empty Vivado project.

3.2. Use IP Integrator to create a new Block Design, with the ZYNQ processing system

3.2.1. Create a Block Design

In the Flow Navigator, create a Block Design (under IP Integrator) with the design name **design_1**.




Create Block Design

Please specify name of block design.

Design name: design_1

Directory: <Local to Project>

Specify source set: Design Sources

3.2.2. Add the Processing System

IP components can be added from the catalog by clicking the Add IP icon in the block diagram side bar, or by right-clicking anywhere in the Diagram workspace and selecting Add IP. Select the Zynq7 Processing System and add it to the design.

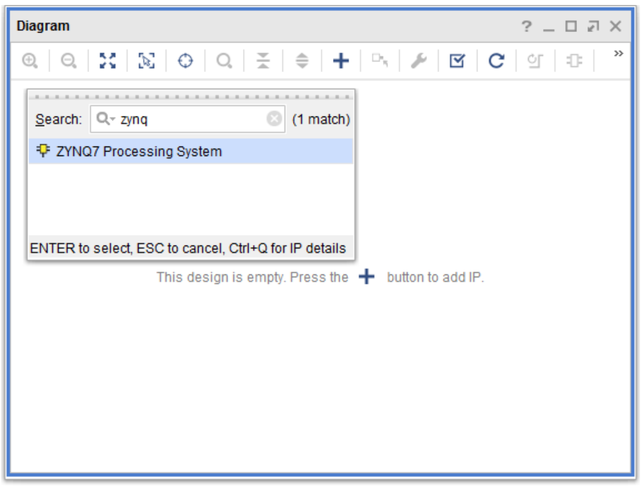
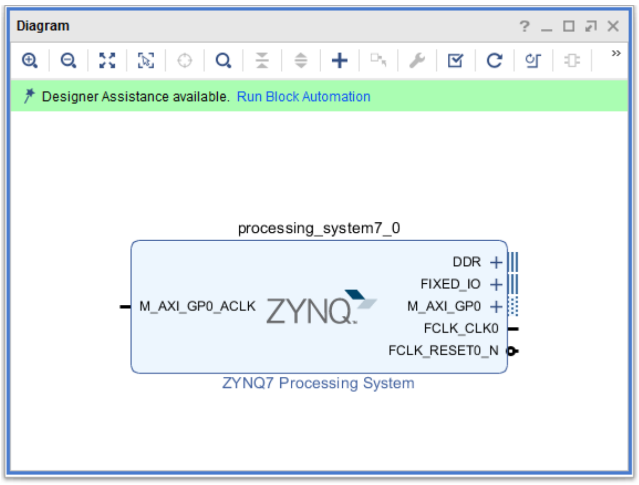
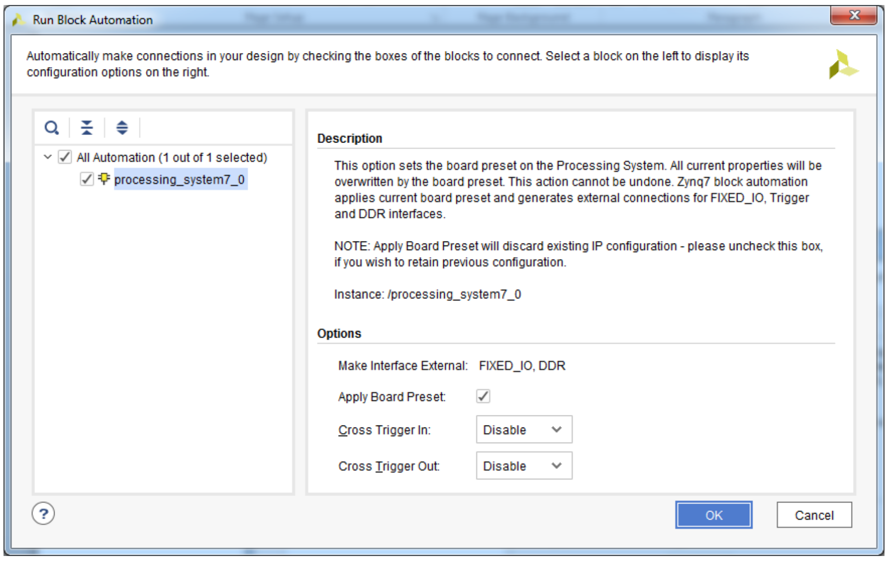


Diagram
Search: zynq (1 match)
ZYNQ7 Processing System
ENTER to select, ESC to cancel, Ctrl+Q for IP details
This design is empty. Press the + button to add IP.



Click on Run Block Automation with the default settings.


Run Block Automation

Automatically make connections in your design by checking the boxes of the blocks to connect. Select a block on the left to display its configuration options on the right.

All Automation (1 out of 1 selected)
processing_system7_0

Description

This option sets the board preset on the Processing System. All current properties will be overwritten by the board preset. This action cannot be undone. Zynq7 block automation applies current board preset and generates external connections for FIXED_IO, Trigger and DDR interfaces.

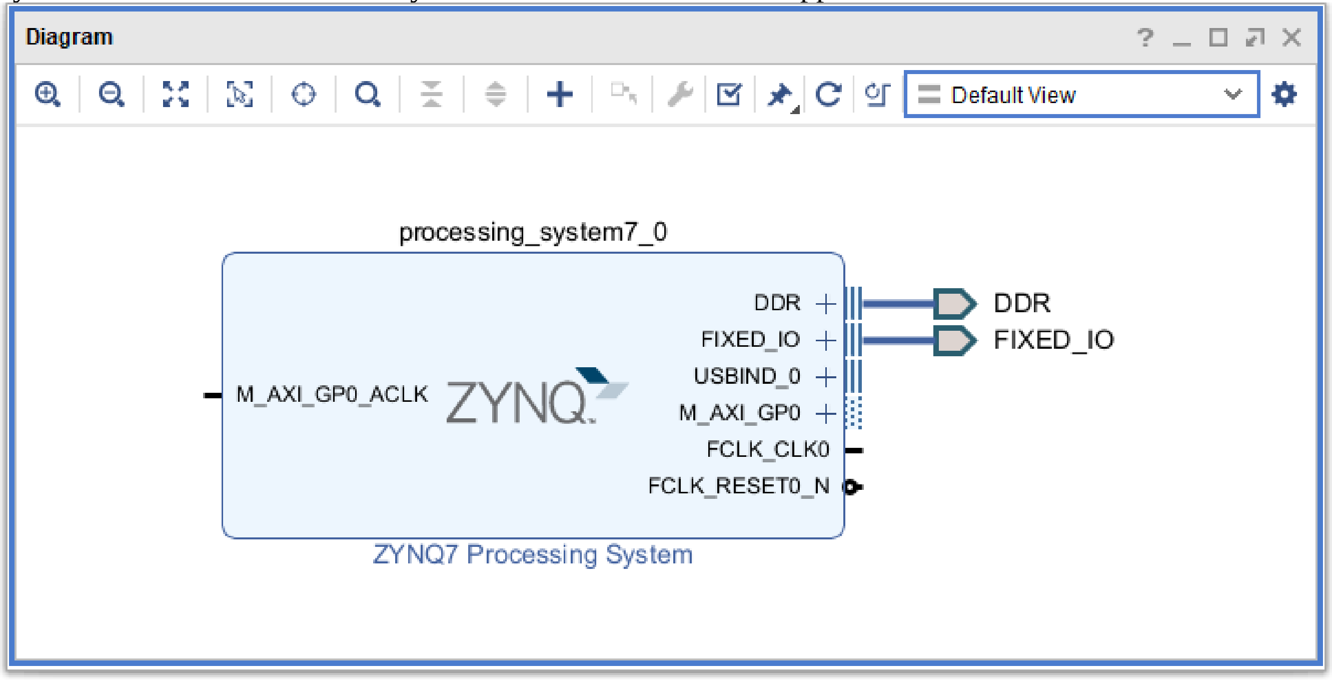
NOTE: Apply Board Preset will discard existing IP configuration - please uncheck this box, if you wish to retain previous configuration.

Instance: /processing_system7_0

Options

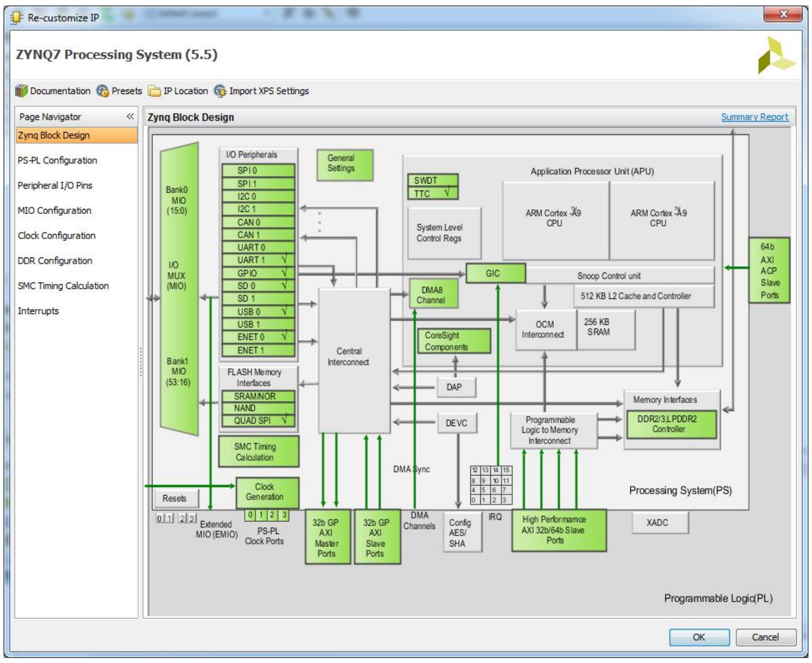
Make Interface External:	FIXED_IO, DDR
Apply Board Preset:	✓
Cross Trigger In:	Disable
Cross Trigger Out:	Disable

Ports are automatically added for the DDR and Fixed IO. An imported configuration for the Zynq PS system related to the Zybo board has been applied which can be customized.

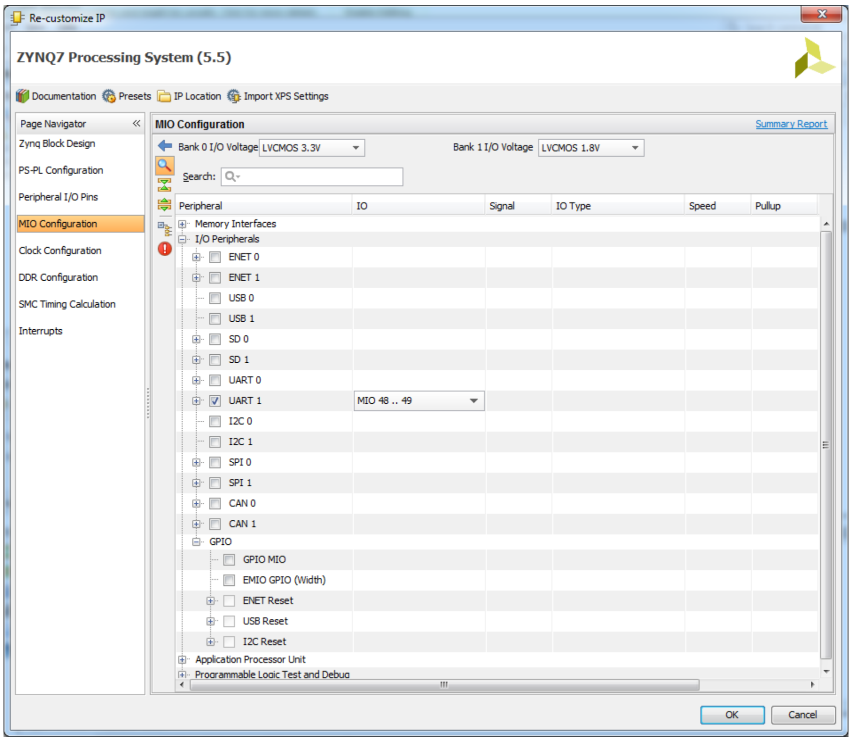


3.2.3. Configure the processing block with only one UART peripheral

Double-click on the PS block to open its customization window.



Open the MIO configuration form and ensure **all the I/O are deselected except UART 1**, i.e. deselect ENET 0, USB 0, SD 0 (I/O Peripherals), GPIO MIO (GPIO), Quad SPI Flash (Memory Interfaces), and Timer 0 (Application Processor Unit).



Re-customize IP

ZYNQ7 Processing System (5.5)

Documentation | Presets | IP Location | Import XPS Settings

Page Navigator: Zynq Block Design, PS-PL Configuration, Peripheral I/O Pins, MIO Configuration, Clock Configuration, DDR Configuration, SMC Timing Calculation, Interrupts

MIO Configuration — Summary Report

Bank 0 I/O Voltage: LVCMOS 3.3V Bank 1 I/O Voltage: LVCMOS 1.8V

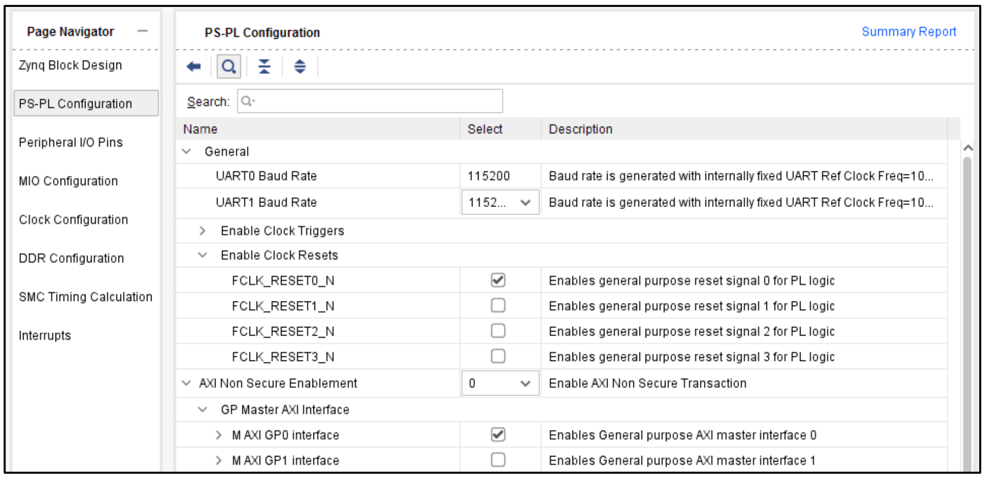
Search:

Peripheral	IO	Signal	IO Type	Speed	Pullup
Memory Interfaces					
I/O Peripherals					
☐ ENET 0					
☐ ENET 1					
☐ USB 0					
☐ USB 1					
☐ SD 0					
☐ SD 1					
☐ UART 0					
☑ UART 1	MIO 48 .. 49				
☐ I2C 0					
☐ I2C 1					
☐ SPI 0					
☐ SPI 1					
☐ CAN 0					
☐ CAN 1					
GPIO					
☐ GPIO MIO					
☐ EMIO GPIO (Width)					
☐ ENET Reset					
☐ USB Reset					
☐ I2C Reset					
Application Processor Unit					
Programmable Logic Test and Debug					

OK | Cancel

3.2.4. PS-PL Configuration

On the PS-PL Configuration window, keep selected the M AXI GP0 interface (AXI Non Secure Enablement → GP Master AXI interface), and the FCLK_RESET0_N option (General → Enable Clock Resets).



Page Navigator: Zynq Block Design, PS-PL Configuration, Peripheral I/O Pins, MIO Configuration, Clock Configuration, DDR Configuration, SMC Timing Calculation, Interrupts

PS-PL Configuration — Summary Report

Search:

Name	Select	Description
General		
UART0 Baud Rate	115200	Baud rate is generated with internally fixed UART Ref Clock Freq=10...
UART1 Baud Rate	1152...	Baud rate is generated with internally fixed UART Ref Clock Freq=10...
Enable Clock Triggers		
Enable Clock Resets		
FCLK_RESET0_N	☑	Enables general purpose reset signal 0 for PL logic
FCLK_RESET1_N	☐	Enables general purpose reset signal 1 for PL logic
FCLK_RESET2_N	☐	Enables general purpose reset signal 2 for PL logic
FCLK_RESET3_N	☐	Enables general purpose reset signal 3 for PL logic
AXI Non Secure Enablement	0	Enable AXI Non Secure Transaction
GP Master AXI Interface		
M AXI GP0 interface	☑	Enables General purpose AXI master interface 0
M AXI GP1 interface	☐	Enables General purpose AXI master interface 1

Figure 1: PS-PL Configuration.

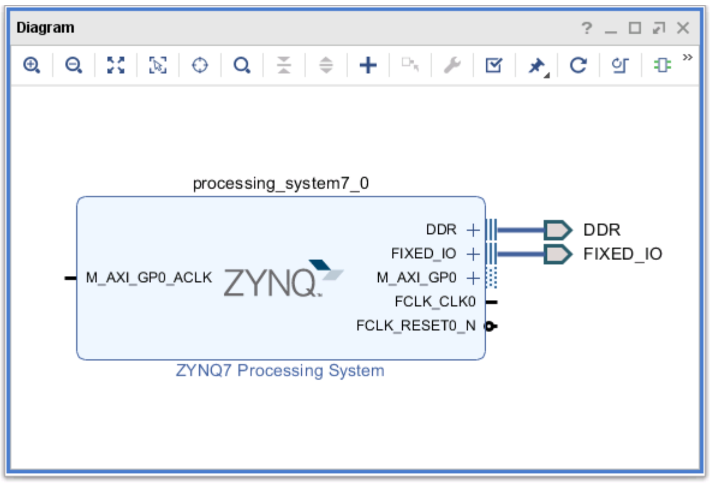
3.2.5. Clock Configuration

On the Clock Configuration window (PL Fabric Clocks), keep the clock configuration with the FCLK_CLK0 clock enabled, and select a requested frequency of 100 MHz.

Do not change the Input Frequency,
which is 50 MHz in the Pynq and Zybo boards and 33.3(3) MHz in the Zybo-Z7-10 boards.

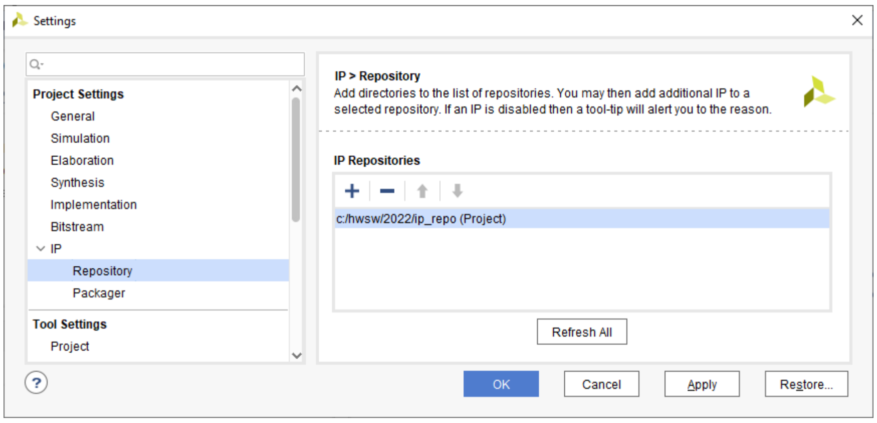
3.2.6. Finalize the PS Configuration

Apply the customization and note that the Zynq PS block includes the GP0, clock and reset ports:



3.3. Specify IP Repository

Add your IP repository folder to the IP repository list in the ***Project Settings***.



Settings

Project Settings: General, Simulation, Elaboration, Synthesis, Implementation, Bitstream, IP (Repository, Packager)

Tool Settings: Project

IP > Repository

Add directories to the list of repositories. You may then add additional IP to a selected repository. If an IP is disabled then a tool-tip will alert you to the reason.

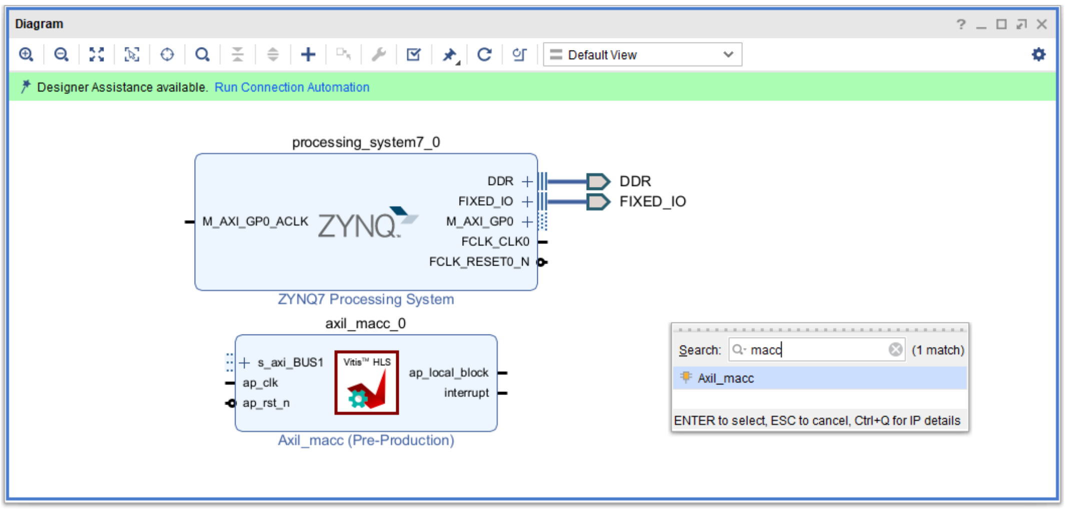
IP Repositories

c:/hwsw/2022/ip_repo (Project)

3.4. Add your IP to the Zynq design

3.4.1. Add New AXI-Lite IP

Click the ***Add IP icon***, search for your new IP and add it to the design.



Diagram

Default View

Designer Assistance available. Run Connection Automation

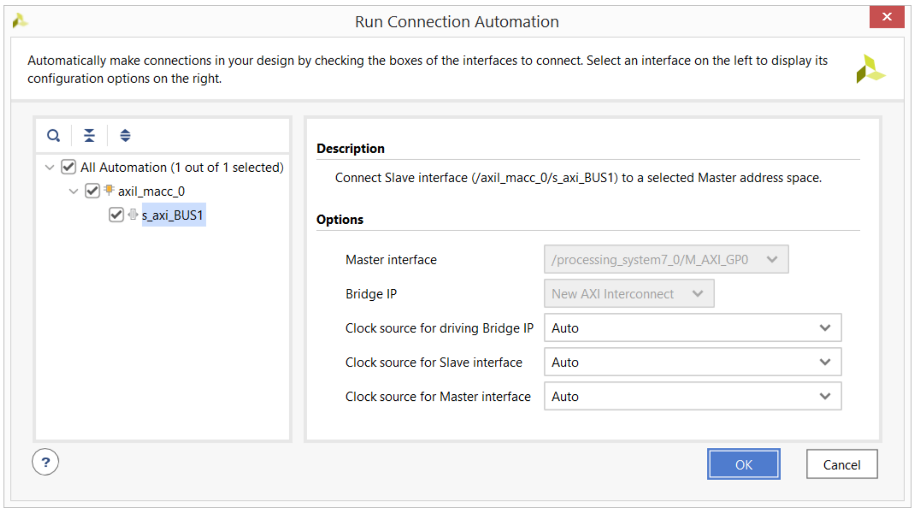
Search: macc (1 match)

Axil_macc

ENTER to select, ESC to cancel, Ctrl+Q for IP details

3.4.2. Connect the blocks

Complete the design by automatically connecting the PS and PL blocks: run ***Connection Automation*** (and select /axil_macc_0/s_axi_BUS1) to automatically make the required connections:



Run Connection Automation

Automatically make connections in your design by checking the boxes of the interfaces to connect. Select an interface on the left to display its configuration options on the right.

- ✓ All Automation (1 out of 1 selected)
  - ✓ axil_macc_0
    - ✓ s_axi_BUS1

Description

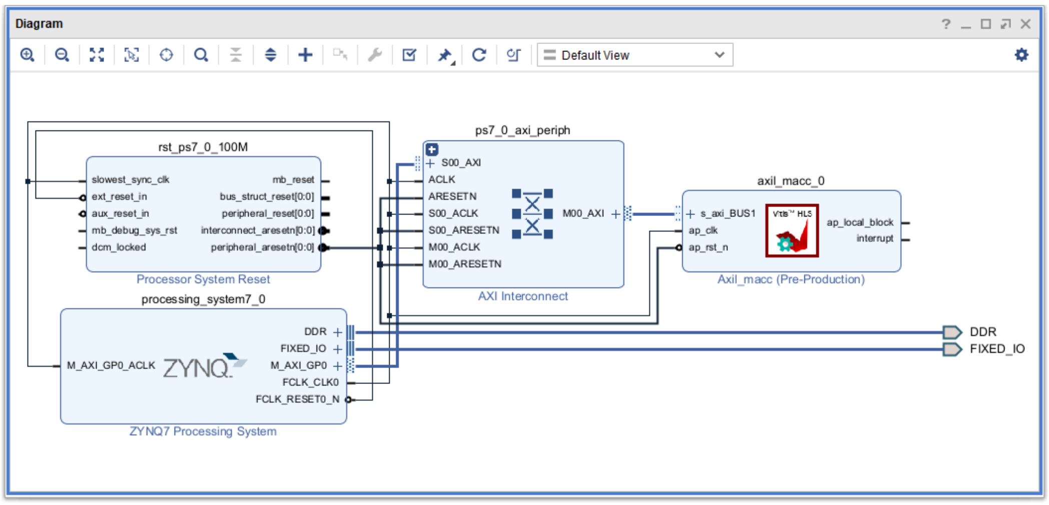
Connect Slave interface (/axil_macc_0/s_axi_BUS1) to a selected Master address space.

Options

Master interface	/processing_system7_0/M_AXI_GP0
Bridge IP	New AXI Interconnect
Clock source for driving Bridge IP	Auto
Clock source for Slave interface	Auto
Clock source for Master interface	Auto

OK Cancel

Note that two additional blocks, Processor System Reset and AXI Interconnect, have been automatically added to the design.

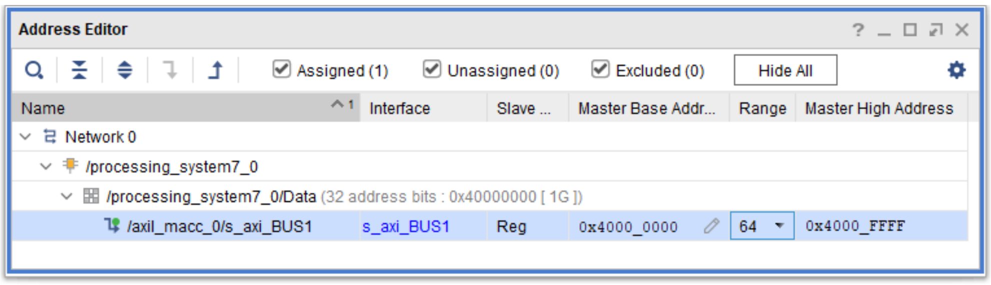


Regenerate the layout.

Validate the design.

Note: if a warning message appears on "PCW_UIPARAM_DDR_DQS_TO_CLK_DELAY_0 has negative value", you may ignore it.

Check the Address Editor tab and check which address region has been assigned to the axil_macc_0 s_axi_BUS_A Registers.
Note: ***do not change*** (manually) the mapping addresses generated by Vivado.

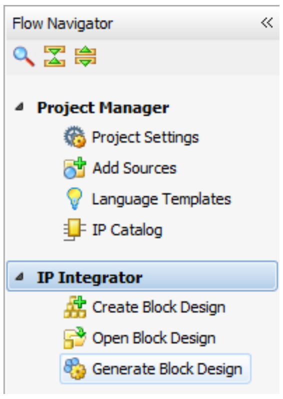


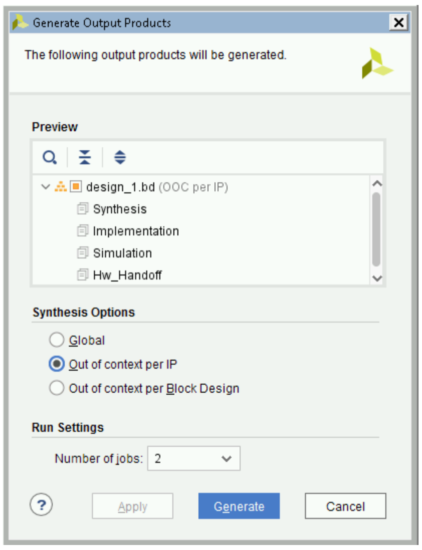
Name	Interface	Slave ...	Master Base Addr...	Range	Master High Address
Network 0					
/processing_system7_0					
/processing_system7_0/Data (32 address bits : 0x40000000 [1G])					
/axil_macc_0/s_axi_BUS1	s_axi_BUS1	Reg	0x4000_0000	64	0x4000_FFFF

3.5. System Hardware generation

3.5.1. Generate Block Design

Click on Generate Block Design (on the Flow Navigator) to generate the Implementation, Simulation and Synthesis files for the design.





The following output products will be generated.

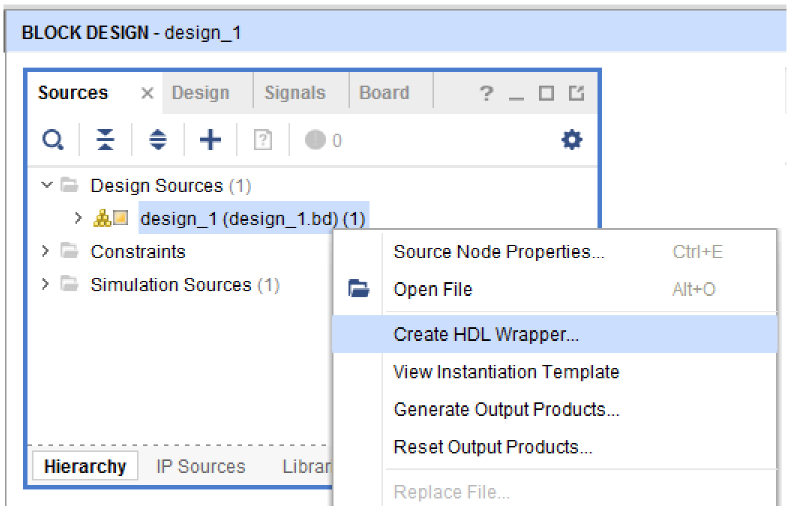
design_1.bd (OOC per IP)
Synthesis
Implementation
Simulation
Hw_Handoff

Synthesis Options
Global [✓ Out of context per IP] Out of context per Block Design

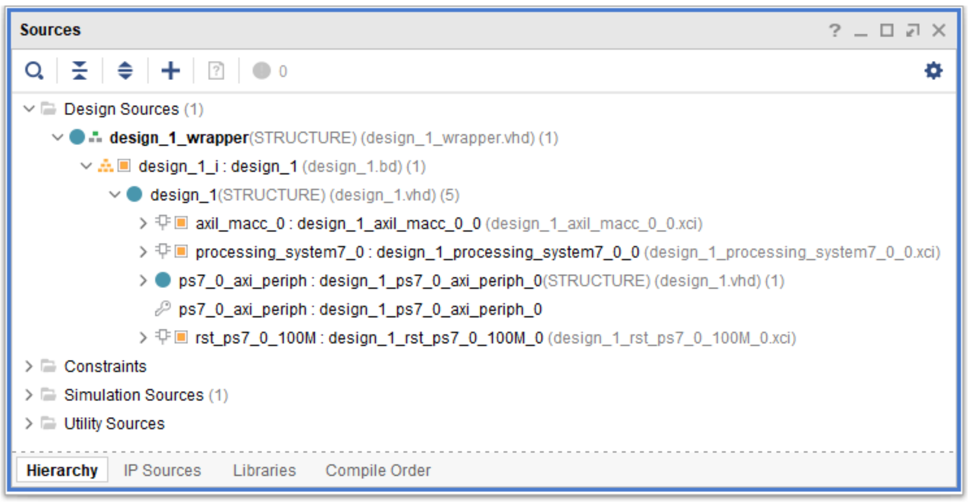
Run Settings
Number of jobs: 2

3.5.2. Create HDL Wrapper

In the sources panel, right-click on *design_1.bd* and select Create HDL Wrapper… to generate the top-level VHDL model. Leave the "*Let Vivado manager wrapper and auto-update*" option selected.



The *design_1_wrapper.vhd* file is created, added to the project, and set as the top module in the design. (You can double-click on the file name to see the content in the Auxiliary pane).


Sources

- Design Sources (1)
  - design_1_wrapper(STRUCTURE) (design_1_wrapper.vhd) (1)
    - design_1_i : design_1 (design_1.bd) (1)
      - design_1(STRUCTURE) (design_1.vhd) (5)
        - axil_macc_0 : design_1_axil_macc_0_0 (design_1_axil_macc_0_0.xci)
        - processing_system7_0 : design_1_processing_system7_0_0 (design_1_processing_system7_0_0.xci)
        - ps7_0_axi_periph : design_1_ps7_0_axi_periph_0(STRUCTURE) (design_1.vhd) (1)
        - ps7_0_axi_periph : design_1_ps7_0_axi_periph_0
        - rst_ps7_0_100M : design_1_rst_ps7_0_100M_0 (design_1_rst_ps7_0_100M_0.xci)
- Constraints
- Simulation Sources (1)
- Utility Sources

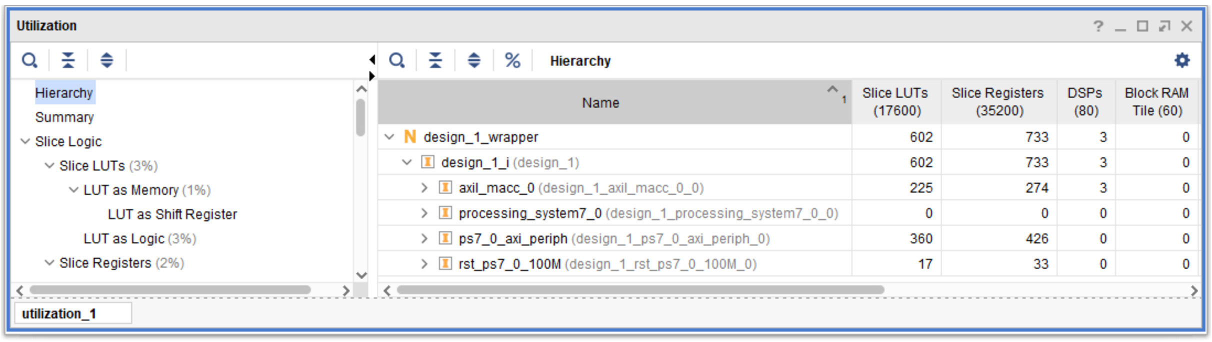
3.5.3. Design Implementation

Implement the Programmable Logic components.

Synthesize the design (***Run Synthesis***),
implement it (***Run Implementation***),
and generate the bitstream (***Generate Bitstream***).

The hardware system has been generated and can now be exported to the Vitis IDE to develop the embedded software and verify the application on the development board.

After implementation, you may check the Vivado reports, namely the *Utilization Report* and the *Timing Summary Report*.



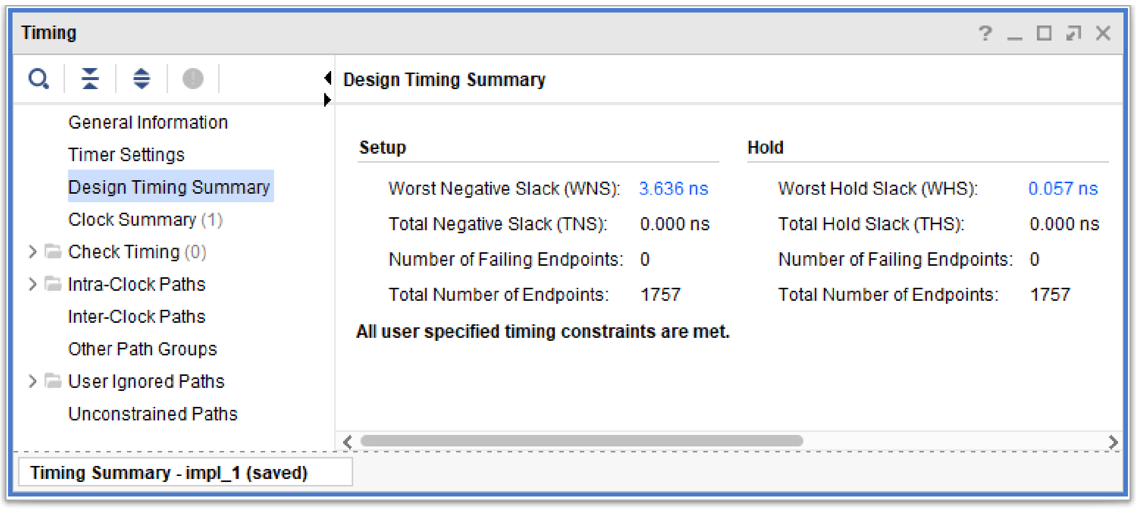
Utilization

- Hierarchy
- Summary
- Slice Logic
  - Slice LUTs (3%)
    - LUT as Memory (1%)
      - LUT as Shift Register
    - LUT as Logic (3%)
  - Slice Registers (2%)

Hierarchy

Name	Slice LUTs (17600)	Slice Registers (35200)	DSPs (80)	Block RAM Tile (60)
design_1_wrapper	602	733	3	0
design_1_i (design_1)	602	733	3	0
axil_macc_0 (design_1_axil_macc_0_0)	225	274	3	0
processing_system7_0 (design_1_processing_system7_0_0)	0	0	0	0
ps7_0_axi_periph (design_1_ps7_0_axi_periph_0)	360	426	0	0
rst_ps7_0_100M (design_1_rst_ps7_0_100M_0)	17	33	0	0

utilization_1



Timing

- General Information
- Timer Settings
- Design Timing Summary
- Clock Summary (1)
- Check Timing (0)
- Intra-Clock Paths
- Inter-Clock Paths
- Other Path Groups
- User Ignored Paths
- Unconstrained Paths

Design Timing Summary

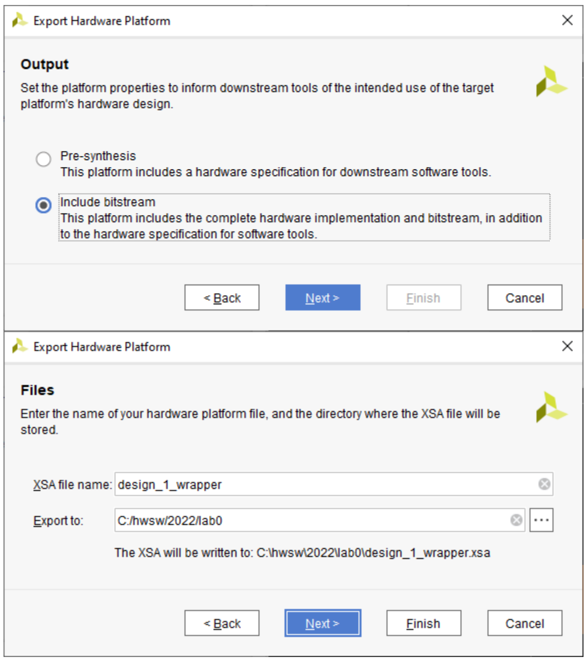
Setup		Hold	
Worst Negative Slack (WNS):	3.636 ns	Worst Hold Slack (WHS):	0.057 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1757	Total Number of Endpoints:	1757

All user specified timing constraints are met.

Timing Summary - impl_1 (saved)

3.5.4. Export Hardware to the Vitis Software Platform

In Vivado, click ***File → Export → Export Hardware***.
This design has hardware in the Programmable Logic (PL), therefore you must include the bitstream to be generated and included.
Make sure that the export path corresponds to your project folder.


Export Hardware Platform

Output

Set the platform properties to inform downstream tools of the intended use of the target platform's hardware design.

Pre-synthesis
This platform includes a hardware specification for downstream software tools.

[✓ Include bitstream]
This platform includes the complete hardware implementation and bitstream, in addition to the hardware specification for software tools.

Export Hardware Platform

Files

Enter the name of your hardware platform file, and the directory where the XSA file will be stored.

XSA file name: design_1_wrapper

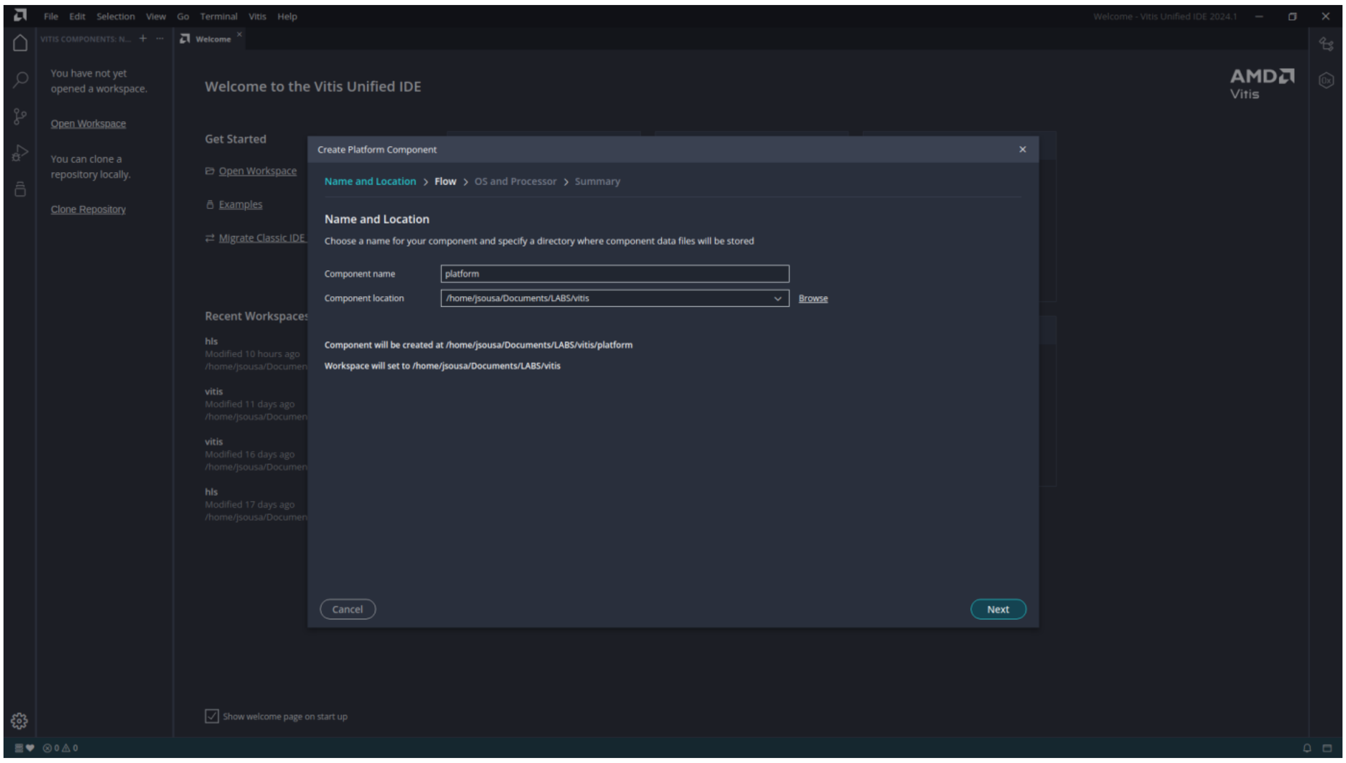
Export to: C:/hwsw/2022/lab0

The XSA will be written to: C:\hwsw\2022\lab0\design_1_wrapper.xsa

4. Software Application

Now return to the Vitis application you left open before.

1. Click ***Embedded Development → Create Platform Component to create a new platform project from the Xilinx Shell Archive (XSA) previously created in*** Vivado. Enter the platform component name and choose a suitable location. Click Next.



Create Platform Component

Name and Location > Flow > OS and Processor > Summary

Name and Location

Choose a name for your component and specify a directory where component data files will be stored

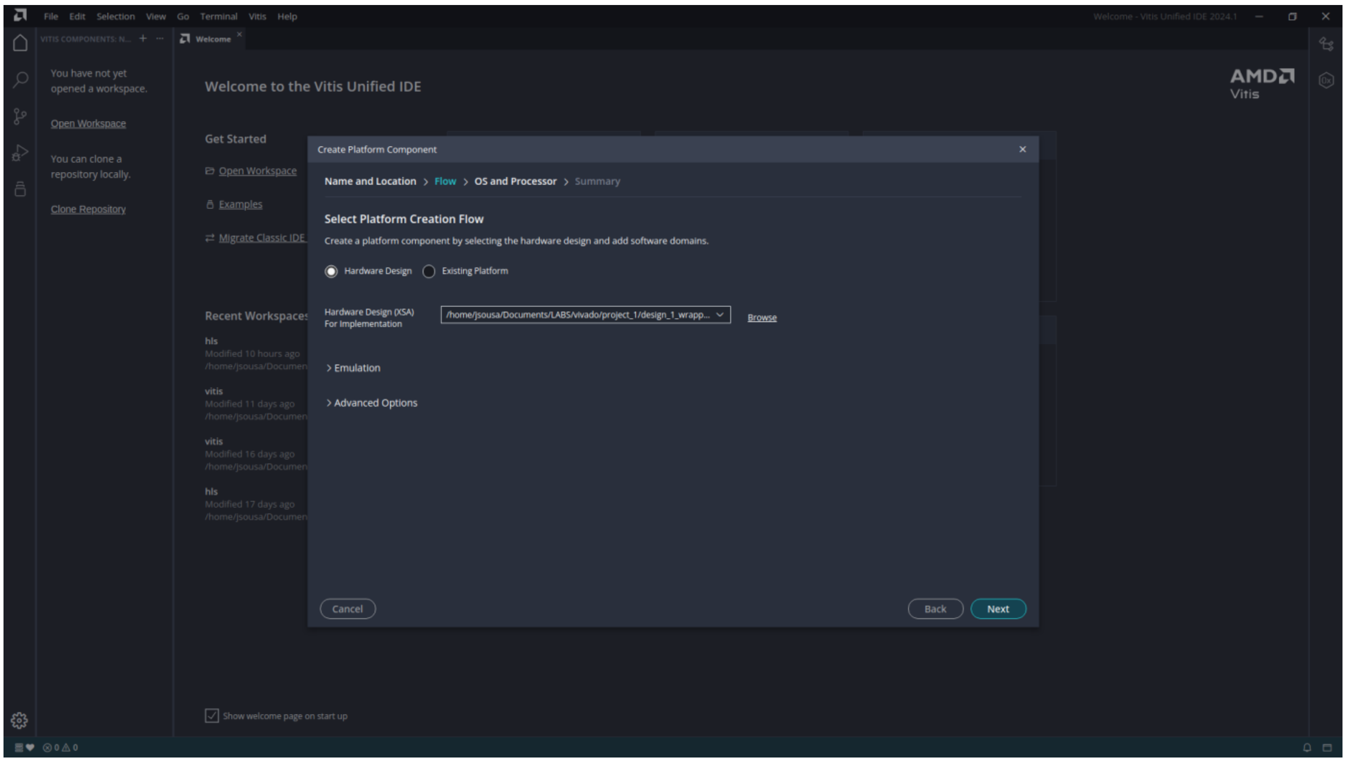
Component name: platform

Component location: /home/jsousa/Documents/LABS/vitis

Component will be created at /home/jsousa/Documents/LABS/vitis/platform

Workspace will set to /home/jsousa/Documents/LABS/vitis

2. Browse to the hardware specification file (*design_1_wrapper.xsa) and select it. Click Next.*



Create Platform Component

Name and Location > Flow > OS and Processor > Summary

Select Platform Creation Flow

Create a platform component by selecting the hardware design and add software domains.

[✓ Hardware Design] Existing Platform

Hardware Design (XSA) For Implementation: /home/jsousa/Documents/LABS/vivado/project_1/design_1_wrapp... Browse

> Emulation

> Advanced Options

3. The *Software Specification* fields (Operating system and Processor) are updated to *standalone* and *ps7_cortexa9_0* respectively. Click Next, review the settings in the next screen and click Finish.
4. Build the platform project by clicking Flow/*Build.*

After the project builds, you can now start developing your software application.

4.1. Create your C Project

In this example, you will use a previously coded C program "*dotprod_v0.c*" that multiplies two vectors (software-only).

1. Create a new C project by clicking ***File → New Component → Application Project***. Choose a name and suitable location for your software component, click Next. Target the existing hardware platform you have previously created, click Next. Leave the default Domain, click Next. Review and click Finish. The C project will be created and shown in the Vitis Project Components window.
2. Now you can import C source files, using the C code provided as in this example. To import a new C source file, select the *src* folder under in the Components view, right-click, and select ***Import->Files..***. Browse and import file *dotprod_v0.c*.

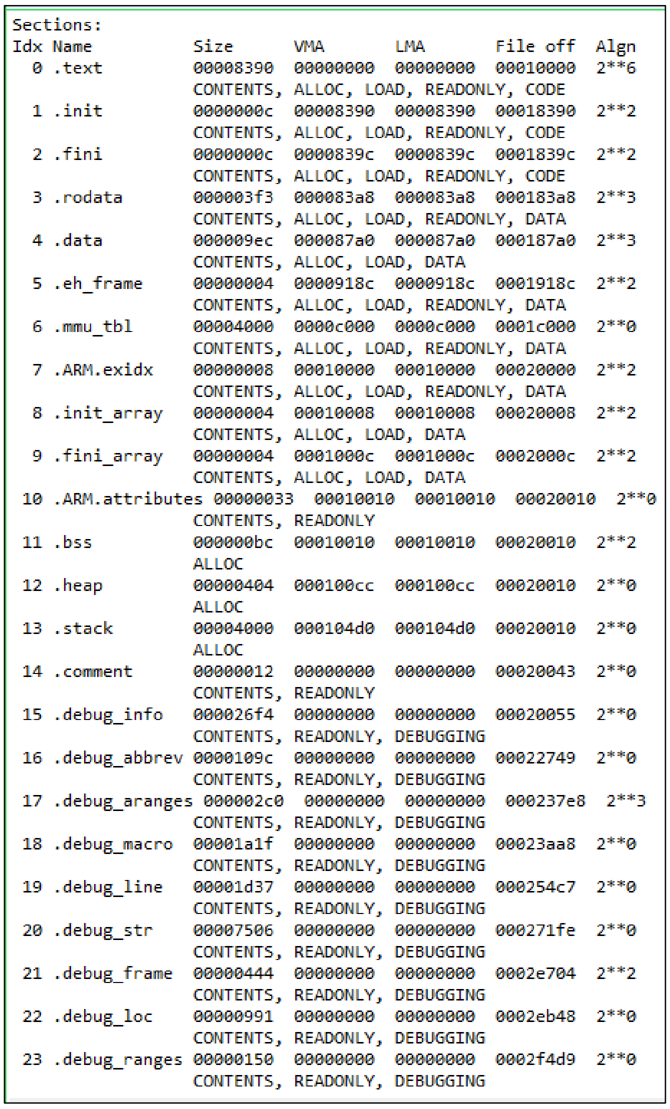
You can set the gcc compiler settings, including the optimization level, by selecting ***Settings*** (below your *application)* and clicking on *UserConfig.cmake* item. *For now, just leave them as default, namely leave the Optimization Level set to None (-O0).*

You can also view the Linker Script by double clicking on *Sources/src/lscript.l*d. Here you can define where the data and instruction sections of your code will be stored. Place all the sections of your program on the OCM and leave the heap and stack sizes as 1K and 10K bytes, respectively.

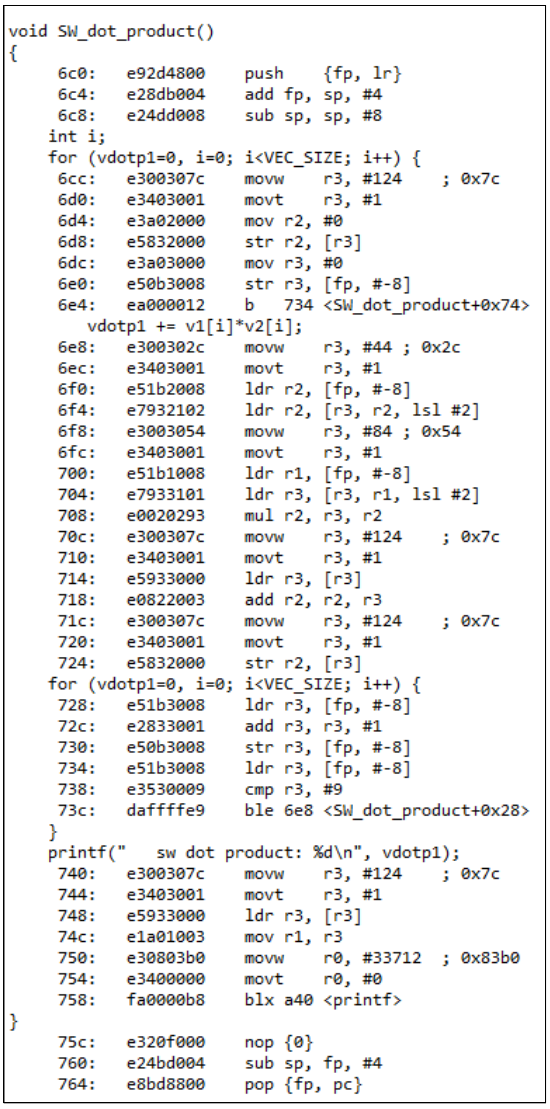
Note: if you use the printf() function extensively, you may need to increase the heap size (e.g. to heap size = 10kB)

3. *Build* the project to generate *the* executable file. The linker combines the compiled applications, including libraries and drivers, and produces an executable file, in Executable Linked Format (ELF), that is ready to execute on your processor hardware platform.

Opening the ***.elf*** (see Components/Output) you can see the instructions generated for your C code, including their instruction memory addresses. At the top, you can view a list of the sizes and starting addresses of the various sections of the program, where ***size*** indicates the size of each section and ***LMA*** is the Loadable Memory Address (start address for each section). Note that, in this case, all sections reside in the OCM memory space (starting with base address 0x00000000).



```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00008390  00000000  00000000  00010000  2**6
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .init         0000000c  00008390  00008390  00018390  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .fini         0000000c  0000839c  0000839c  0001839c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .rodata       000003f3  000083a8  000083a8  000183a8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         000009ec  000087a0  000087a0  000187a0  2**3
                  CONTENTS, ALLOC, LOAD, DATA
  5 .eh_frame     00000004  0000918c  0000918c  0001918c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .mmu_tbl      00004000  0000c000  0000c000  0001c000  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .ARM.exidx    00000008  00010000  00010000  00020000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .init_array   00000004  00010008  00010008  00020008  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  9 .fini_array   00000004  0001000c  0001000c  0002000c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 10 .ARM.attributes 00000033  00010010  00010010  00020010  2**0
                  CONTENTS, READONLY
 11 .bss          000000bc  00010010  00010010  00020010  2**2
                  ALLOC
 12 .heap         00000404  000100cc  000100cc  00020010  2**0
                  ALLOC
 13 .stack        00004000  000104d0  000104d0  00020010  2**0
                  ALLOC
 14 .comment      00000012  00000000  00000000  00020043  2**0
                  CONTENTS, READONLY
 15 .debug_info   000026f4  00000000  00000000  00020055  2**0
                  CONTENTS, READONLY, DEBUGGING
 16 .debug_abbrev 0000109c  00000000  00000000  00022749  2**0
                  CONTENTS, READONLY, DEBUGGING
 17 .debug_aranges 000002c0  00000000  00000000  000237e8  2**3
                  CONTENTS, READONLY, DEBUGGING
 18 .debug_macro  00001a1f  00000000  00000000  00023aa8  2**0
                  CONTENTS, READONLY, DEBUGGING
 19 .debug_line   00001d37  00000000  00000000  000254c7  2**0
                  CONTENTS, READONLY, DEBUGGING
 20 .debug_str    00007506  00000000  00000000  000271fe  2**0
                  CONTENTS, READONLY, DEBUGGING
 21 .debug_frame  00000444  00000000  00000000  0002e704  2**2
                  CONTENTS, READONLY, DEBUGGING
 22 .debug_loc    00000991  00000000  00000000  0002eb48  2**0
                  CONTENTS, READONLY, DEBUGGING
 23 .debug_ranges 00000150  00000000  00000000  0002f4d9  2**0
                  CONTENTS, READONLY, DEBUGGING
```



```
void SW_dot_product()
{
     6c0:   e92d4800    push    {fp, lr}
     6c4:   e28db004    add fp, sp, #4
     6c8:   e24dd008    sub sp, sp, #8
    int i;
    for (vdotp1=0, i=0; i<VEC_SIZE; i++) {
     6cc:   e300307c    movw    r3, #124    ; 0x7c
     6d0:   e3403001    movt    r3, #1
     6d4:   e3a02000    mov r2, #0
     6d8:   e5832000    str r2, [r3]
     6dc:   e3a03000    mov r3, #0
     6e0:   e50b3008    str r3, [fp, #-8]
     6e4:   ea000012    b   734 <SW_dot_product+0x74>
        vdotp1 += v1[i]*v2[i];
     6e8:   e300302c    movw    r3, #44 ; 0x2c
     6ec:   e3403001    movt    r3, #1
     6f0:   e51b2008    ldr r2, [fp, #-8]
     6f4:   e7932102    ldr r2, [r3, r2, lsl #2]
     6f8:   e3003054    movw    r3, #84 ; 0x54
     6fc:   e3403001    movt    r3, #1
     700:   e51b1008    ldr r1, [fp, #-8]
     704:   e7933101    ldr r3, [r3, r1, lsl #2]
     708:   e0020293    mul r2, r3, r2
     70c:   e300307c    movw    r3, #124    ; 0x7c
     710:   e3403001    movt    r3, #1
     714:   e5933000    ldr r3, [r3]
     718:   e0822003    add r2, r2, r3
     71c:   e300307c    movw    r3, #124    ; 0x7c
     720:   e3403001    movt    r3, #1
     724:   e5832000    str r2, [r3]
    for (vdotp1=0, i=0; i<VEC_SIZE; i++) {
     728:   e51b3008    ldr r3, [fp, #-8]
     72c:   e2833001    add r3, r3, #1
     730:   e50b3008    str r3, [fp, #-8]
     734:   e51b3008    ldr r3, [fp, #-8]
     738:   e3530009    cmp r3, #9
     73c:   daffffe9    ble 6e8 <SW_dot_product+0x28>
    }
    printf("   sw dot product: %d\n", vdotp1);
     740:   e300307c    movw    r3, #124    ; 0x7c
     744:   e3403001    movt    r3, #1
     748:   e5933000    ldr r3, [r3]
     74c:   e1a01003    mov r1, r3
     750:   e30803b0    movw    r0, #33712  ; 0x83b0
     754:   e3400000    movt    r0, #0
     758:   fa0000b8    blx a40 <printf>
}
     75c:   e320f000    nop {0}
     760:   e24bd004    sub sp, fp, #4
     764:   e8bd8800    pop {fp, pc}
```

You can check the size occupied by your program by opening the file ***dotprod0.elf.size***. In this example it occupies about 70 kB.

```
   text      data       bss       dec       hex   filename
  51111      2548     17600     71259     1165b   dotprod_0.elf
```

Note: your program must fit in with the memory you select to store it in!
Also, you must avoid trying to store multiple programs and/or data in the same memory zones.
All the memory management is done by the designer, you!

5. Verify the Design in Hardware

5.1. Connect the board and establish serial communication from Vitis's Terminal

Connect and power up the Pynq Z2 oard. You will use one cable to connect to the PROG/UART port of the board.

Warning: be careful when connecting the cable, as the board connector is very breakable.

Open a terminal with
Vitis → Serial Monitor Terminal.

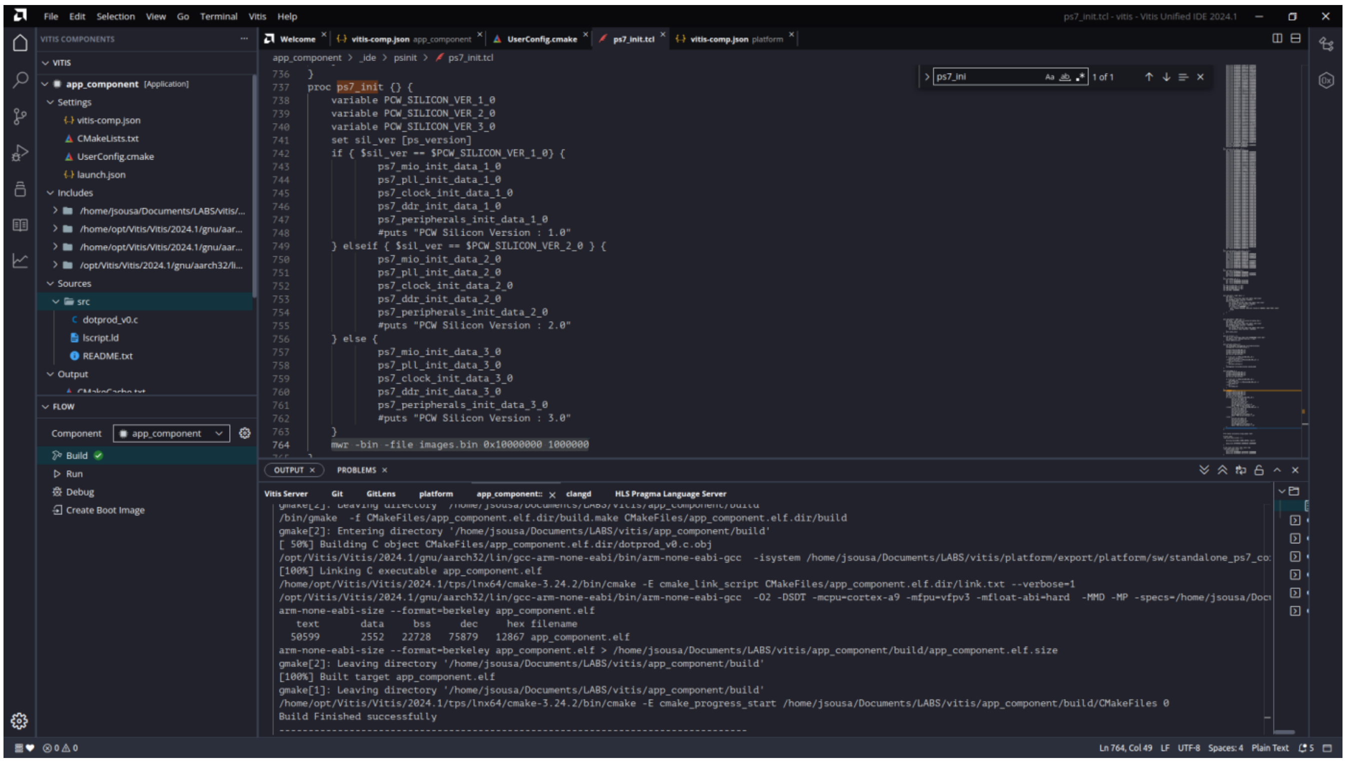
Select the serial port and baud rate. After clicking ***Run***, the application will execute, and you will see the *stdout* output in the Terminal console created before.
Note that this *dotprod_0* application is software-only (only the processor system is used).

5.2. Debug your software using the Application Debugger

To execute the program in debugging mode, select *Debug instead of Run.* The application will start and stop in the program's first instruction, and the IDE automatically changes to Debug view (you can change between Design and Debug view using the top-right buttons).
Breakpoints can be set/unset by double-clicking on the blue bar on the left of the program window.
The program flow can be controlled by the resume and step control buttons on the top taskbar.
The program variables and program memory zones can be monitored on the right-side windows.

6. Memory Initialization from a Binary File

You can debug your applications by initializing part of the memory of your system using data from a binary file in your computer. This basic procedure will be used, in your future projects, to demonstrate your designs. Edit the file ps7_init.tcl as in the figure, where images.bin is the binary file, the first number is the memory address, and the last number is the file size.



```
}
proc ps7_init {} {
    variable PCW_SILICON_VER_1_0
    variable PCW_SILICON_VER_2_0
    variable PCW_SILICON_VER_3_0
    set sil_ver [ps_version]
    if { $sil_ver == $PCW_SILICON_VER_1_0} {
            ps7_mio_init_data_1_0
            ps7_pll_init_data_1_0
            ps7_clock_init_data_1_0
            ps7_ddr_init_data_1_0
            ps7_peripherals_init_data_1_0
            #puts "PCW Silicon Version : 1.0"
    } elseif { $sil_ver == $PCW_SILICON_VER_2_0 } {
            ps7_mio_init_data_2_0
            ps7_pll_init_data_2_0
            ps7_clock_init_data_2_0
            ps7_ddr_init_data_2_0
            ps7_peripherals_init_data_2_0
            #puts "PCW Silicon Version : 2.0"
    } else {
            ps7_mio_init_data_3_0
            ps7_pll_init_data_3_0
            ps7_clock_init_data_3_0
            ps7_ddr_init_data_3_0
            ps7_peripherals_init_data_3_0
            #puts "PCW Silicon Version : 3.0"
    }
    mwr -bin -file images.bin 0x10000000 1000000
```

```
/bin/gmake  -f CMakeFiles/app_component.elf.dir/build.make CMakeFiles/app_component.elf.dir/build
gmake[2]: Entering directory '/home/jsousa/Documents/LABS/vitis/app_component/build'
[ 50%] Building C object CMakeFiles/app_component.elf.dir/dotprod_v0.c.obj
/opt/Vitis/Vitis/2024.1/gnu/aarch32/lin/gcc-arm-none-eabi/bin/arm-none-eabi-gcc  -isystem /home/jsousa/Documents/LABS/vitis/platform/export/platform/sw/standalone_ps7_co
[100%] Linking C executable app_component.elf
/home/opt/Vitis/Vitis/2024.1/tps/lnx64/cmake-3.24.2/bin/cmake -E cmake_link_script CMakeFiles/app_component.elf.dir/link.txt --verbose=1
/opt/Vitis/Vitis/2024.1/gnu/aarch32/lin/gcc-arm-none-eabi/bin/arm-none-eabi-gcc  -O2 -DSDT -mcpu=cortex-a9 -mfpu=vfpv3 -mfloat-abi=hard  -MMD -MP -specs=/home/jsousa/Docu
arm-none-eabi-size --format=berkeley app_component.elf
   text	   data	    bss	    dec	    hex	filename
  50599	   2552	  22728	  75879	  12867	app_component.elf
arm-none-eabi-size --format=berkeley app_component.elf > /home/jsousa/Documents/LABS/vitis/app_component/build/app_component.elf.size
gmake[2]: Leaving directory '/home/jsousa/Documents/LABS/vitis/app_component/build'
[100%] Built target app_component.elf
gmake[1]: Leaving directory '/home/jsousa/Documents/LABS/vitis/app_component/build'
/home/opt/Vitis/Vitis/2024.1/tps/lnx64/cmake-3.24.2/bin/cmake -E cmake_progress_start /home/jsousa/Documents/LABS/vitis/app_component/build/CMakeFiles 0
Build Finished successfully
--------------------------------------------------------------------------------
```

You must be careful to select the memory address zone such that it corresponds to a valid data memory zone in your system and that it is free to store your data. Choose a free memory zone of your system to store your data. Be careful that this zone must have the size adequate for the required storage and must not interfere with the memory zones used for other storage, namely for the program(s) code and data. **Remember that all the memory management is done by you.**

In the figure example, the base address selected, 0x10000000, defines a memory area to be used in the external memory (DDR).

Then and for example, you can set the base address for your vector's storage in your C program, and use it to set constant pointers to address the vectors:

```
#define VEC1_START_ADD 0x10000000
#define VEC2_START_ADD (VEC1_START_ADD+4*VEC_SIZE)

int *v1 = (int *)(VEC1_START_ADD);
int *v2 = (int *)(VEC2_START_ADD);
```

7. Measure the execution time of your application

You can use the time-specific function ***XTime_GetTime (XTime *xtime)*** (available in the standalone BSP) to evaluate the execution time of your application. This function provides direct access to the 64-bit Global Counter in the PMU. This global timer (GT) is a 64-bit incrementing counter with an auto-incrementing feature and is accessible to both Cortex-A9 processors. The global timer is always clocked at 1/2 of the CPU frequency (the counter increments at every 2 processor cycles). The pre-defined value COUNTS_PER_SECOND directly indicates the GC clock frequency (number of counts per second).

```
#include <stdio.h>

#include "xiltimer.h"

int main()

{

  XTime tStart, tEnd;

  // initialize the application

  XTime_GetTime(&tStart);    // start measuring time

  // process the application

  XTime_GetTime(&tEnd);      // end measuring time

  // finalize the application

  printf("Execution took %llu clock cycles.\n",
          2*(tEnd - tStart));

  printf("Output took %.2f us.\n",
          1.0*(tEnd - tStart) * 1000000/(COUNTS_PER_SECOND));

  return 0;
}
```

Note: when measuring the performance of (parts of) your application, be careful not to measure the time of the printfs!