OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

PYNQ SoC Builder

Luke Canny

19339166

ME Electronic and Computer Engineering

May 2024

Project Supervisor: Dr. Fearghal Morgan

Project Co-Supervisor: Prof. Liam Kilmartin

Abstract

This thesis presents an extension to the HDLGen-ChatGPT project, aimed at enhancing user experience by automating the deployment and use of field programmable gate array (FPGA) overlays on local and remote hardware. This project is comprised of two key elements: a standalone Python-based FPGA overlay and driver application generation tool, and a methodology to enable remote connection to Xilinx PYNQ (Python for Zynq) FPGA hardware. The PYNQ SoC Builder automates the deployment of HDLGen-ChatGPT designs onto PYNQ hardware. Additionally, a Jupyter Notebook driver application is automatically generated to replicate testbench cases on real hardware, and to reduce the time-to-deploy. The PYNQ SoC Builder targets the Vivado electronic design suite (EDA) and supports both Verilog and VHDL.

Declaration of Originality

I hereby certify that this dissertation is entirely my own work. Neither the work nor parts thereof have been published elsewhere in either paper or electronic form unless indicated otherwise through referencing

Date: 10/May/2024 Signature: Luke Canny

Acknowledgements

I would like to thank Dr. Fearghal Morgan for his support and guidance, and Roshan George, JP Byrne and Abishek Bupathi for their previous work on HDLGen-ChatGPT, PYNQ and RISC-V.

Contents

List of Figures

List of Tables

List of Acronyms

SoC – System on Chip

HDL – Hardware Description Language

FPGA – Field Programmable Gate Array

Tcl – Tool Command Language

EDA – Electronic Design Automation

I/O – Input/Output

IP – Intellectual Property

GUI – Graphical User Interface

IDE – Integrated Development Environment

HLS – High-Level Synthesis

ALU – Arithmetic Logic Unit

RTL – Register Transfer Logic

XML – Extensible Markup Language

MUX – Multiplexer

RISC – Reduced Instruction Set Computer

1. Introduction

1.1 Project Introduction

PYNQ System-on-Chip (SoC) Builder is an open-source Python-based wizard for automated deployment of hardware description language (HDL) models to AMD PYNQ field programmable gate array (FPGA) hardware, and generation of Python driver applications based on the HDL model and testbench. PYNQ SoC Builder is an extension of an existing University of Galway project called HDLGen-ChatGPT, a Python-based wizard for the fast capture of digital design specifications and generation of HDL models, testbenches, Tool Command Language (Tcl) scripts and Electronic Design Automation (EDA) project files. This project targets HDLGen-ChatGPT projects configured for the Xilinx Vivado EDA design suite (version 2023.2).

Using the PYNQ SoC Builder, users can:

- Make connections between board input/output (I/O) and component ports
- Configure SoC Builder program options
- Execute a full build of a HDLGen-ChatGPT projects
- Generate a Jupyter Notebook component driver applications
- Upload and launch to a local FPGA hardware
- Launch remote FPGA hardware
- Load remote web-camera (to view remote FPGA LED I/O)

In figure 1, an overview of the new project workflow is illustrated. It describes the discrete steps taken from capturing a design specification to running the design on a PYNQ FPGA.

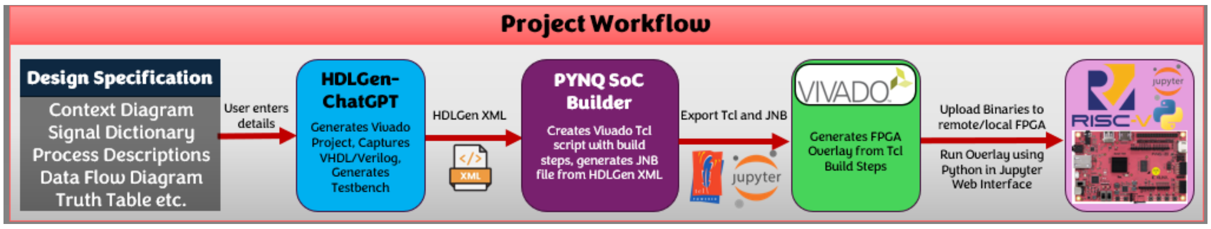
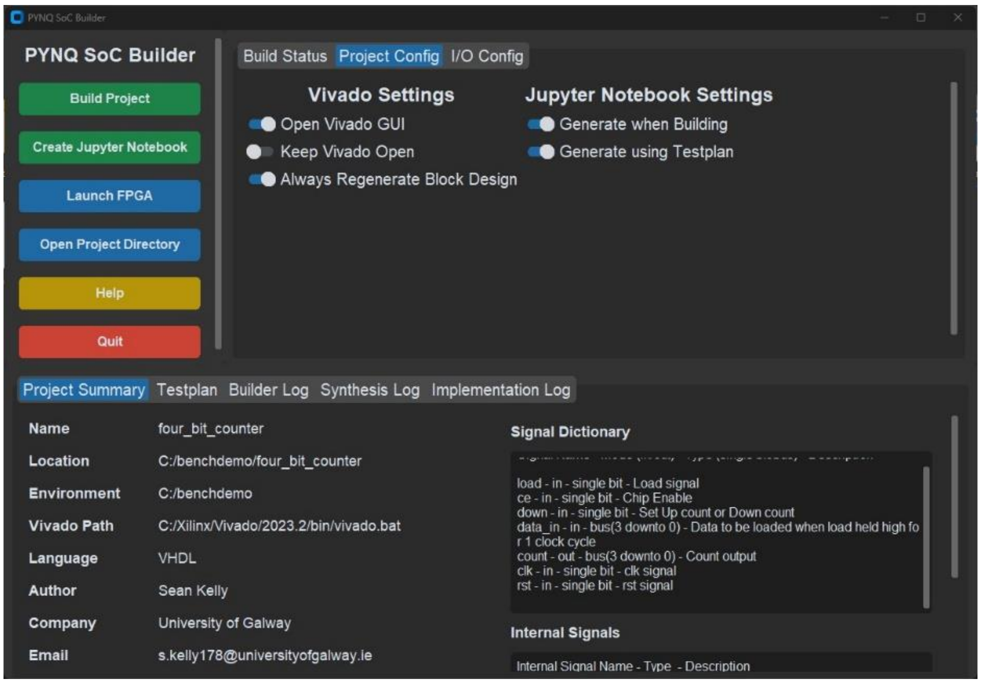

Project Workflow
Design Specification
Context Diagram
Signal Dictionary
Process Descriptions
Data Flow Diagram
Truth Table etc.
User enters details
HDLGen-ChatGPT
Generates Vivado Project, Captures VHDL/Verilog, Generates Testbench
HDLGen XML
XML
PYNQ SoC Builder
Creates Vivado Tcl script with build steps, generates JNB file from HDLGen XML
Export Tcl and JNB
jupyter
VIVADO
Generates FPGA Overlay from Tcl Build Steps
Upload Binaries to remote/local FPGA
Run Overlay using Python in Jupyter Web Interface
RISC-V
jupyter

Figure 1 Top Level Overview of a HDLGen-ChatGPT and PYNQ SoC Builder Project Workflow

Figure 2 presents the graphical user interface (GUI) developed for the PYNQ SoC Builder. Once a HDLGen-ChatGPT design has been completed, the user will use this interface to build their projects.



Vivado Settings

- Open Vivado GUI
- Keep Vivado Open
- Always Regenerate Block Design

Jupyter Notebook Settings

- Generate when Building
- Generate using Testplan

Name	four_bit_counter
Location	C:/benchdemo/four_bit_counter
Environment	C:/benchdemo
Vivado Path	C:/Xilinx/Vivado/2023.2/bin/vivado.bat
Language	VHDL
Author	Sean Kelly
Company	University of Galway
Email	s.kelly178@universityofgalway.ie

Signal Dictionary

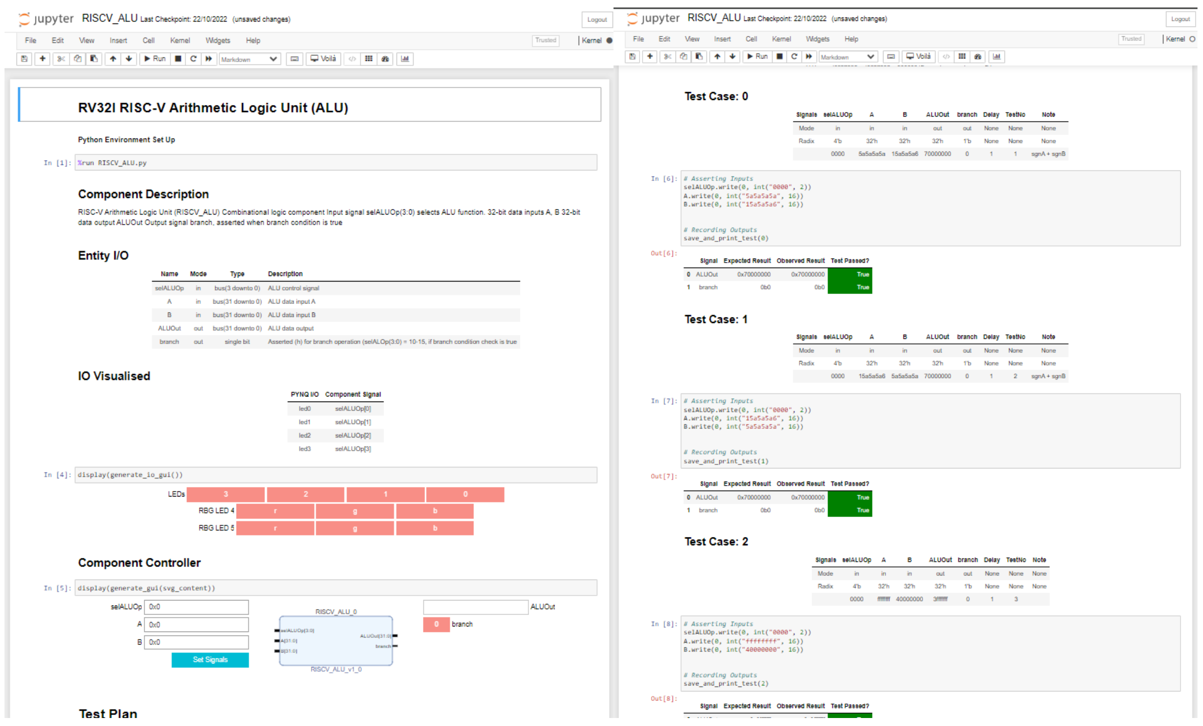
```
load - in - single bit - Load signal
ce - in - single bit - Chip Enable
down - in - single bit - Set Up count or Down count
data_in - in - bus(3 downto 0) - Data to be loaded when load held high fo
r 1 clock cycle
count - out - bus(3 downto 0) - Count output
clk - in - single bit - clk signal
rst - in - single bit - rst signal
```

Internal Signals

```
Internal Signal Name - Type  - Description
```

Figure 2 PYNQ SoC Builder Graphical User Interface

Figure 3 presents a Jupyter Notebook-based driver application produced by the PYNQ SoC Builder for a sample project. It contains a GUI-based controller and testbench execution code.



RV32I RISC-V Arithmetic Logic Unit (ALU)

Python Environment Set Up

```
In [1]: %run RISCV_ALU.py
```

Component Description

RISC-V Arithmetic Logic Unit (RISCV_ALU) Combinational logic component Input signal selALUOp(3:0) selects ALU function. 32-bit data inputs A, B 32-bit data output ALUOut Output signal branch, asserted when branch condition is true

Entity I/O

Name	Mode	Type	Description
selALUOp	in	bus(3 downto 0)	ALU control signal
A	in	bus(31 downto 0)	ALU data input A
B	in	bus(31 downto 0)	ALU data input B
ALUOut	out	bus(31 downto 0)	ALU data output
branch	out	single bit	Asserted (h) for branch operation (selALOp(3:0) = 10-15, if branch condition check is true

IO Visualised

PYNQ I/O	Component Signal
led0	selALUOp[0]
led1	selALUOp[1]
led2	selALUOp[2]
led3	selALUOp[3]

```
In [4]: display(generate_io_gui())
```

Component Controller

```
In [5]: display(generate_gui(svg_content))
```

Test Plan

Test Case: 0

Signals	selALUOp	A	B	ALUOut	branch	Delay	TestNo	Note
Mode	in	in	in	out	out	None	None	None
Radix	4'b	32'h	32'h	32'h	1'b	None	None	None
	0000	5a5a5a5a	15a5a5a6	70000000	0	1	1	sgnA + sgnB

```
In [6]: # Asserting Inputs
        selALUOp.write(0, int("0000", 2))
        A.write(0, int("5a5a5a5a", 16))
        B.write(0, int("15a5a5a6", 16))


        # Recording Outputs
        save_and_print_test(0)
```

Out[6]:

	Signal	Expected Result	Observed Result	Test Passed?
0	ALUOut	0x70000000	0x70000000	True
1	branch	0b0	0b0	True

Test Case: 1

Signals	selALUOp	A	B	ALUOut	branch	Delay	TestNo	Note
Mode	in	in	in	out	out	None	None	None
Radix	4'b	32'h	32'h	32'h	1'b	None	None	None
	0000	15a5a5a6	5a5a5a5a	70000000	0	1	2	sgnA + sgnB

```
In [7]: # Asserting Inputs
        selALUOp.write(0, int("0000", 2))
        A.write(0, int("15a5a5a6", 16))
        B.write(0, int("5a5a5a5a", 16))


        # Recording Outputs
        save_and_print_test(1)
```

Out[7]:

	Signal	Expected Result	Observed Result	Test Passed?
0	ALUOut	0x70000000	0x70000000	True
1	branch	0b0	0b0	True

Test Case: 2

Signals	selALUOp	A	B	ALUOut	branch	Delay	TestNo	Note
Mode	in	in	in	out	out	None	None	None
Radix	4'b	32'h	32'h	32'h	1'b	None	None	None
	0000	ffffffff	40000000	3fffffff	0	1	3	

```
In [8]: # Asserting Inputs
        selALUOp.write(0, int("0000", 2))
        A.write(0, int("ffffffff", 16))
        B.write(0, int("40000000", 16))


        # Recording Outputs
        save_and_print_test(2)
```

Out[8]:

Signal	Expected Result	Observed Result	Test Passed?

Figure 3 Example Jupyter Notebook generated for RISC-V ALU complete with IO configuration and Test case execution

1.2 Goals and Justification

The actions necessary to generate FPGA overlays in the Vivado EDA tool can be a very lengthy, tedious, and repetitive task. Particularly for HDLGen-ChatGPT projects, where the build process is largely the same for each project. This build process can be very prone to human error as the build steps must be completed in a specific sequence, otherwise a build may fail, or the created overlay can behave unexpectedly. Additionally, it can be very challenging for students and beginners to learn how to build their own projects manually. Students and beginners can be discouraged from experimenting and iterating their HDL designs if designs contain many errors after labour-intensive build process. The repetitive nature of the build process can become very time-consuming, limiting the time and focus beginners can spend developing the core functionality of their HDL models. This process can deter students from iterating on their designs.

The aim of this project is to create an automation tool which enables fast FPGA overlay generation, deployment and testing of HDL models. This project has been created as an extension of the University of Galway HDLGen-ChatGPT project. This report has the goal of being both informative and useful to potential users of the PYNQ SoC Builder or remote environment and hence a series of videos are linked throughout chapter 3, 4 and 5. A complete list of online resources including videos, sample projects and documentation produced for this project is available in appendix A.

Secondly, this project aims to develop an online remote FPGA environment intended for students to be able to test their designs on real hardware. Although students have access to various simulation tools available in Vivado and are expected to use simulation to verify their designs, providing access to real FPGA hardware can encourage greater motivation, engagement and learning in universities.

1.3 Project Management and Timeline

Git is the preferred choice for this project, and hence a GitHub repository was established in early mid-October. GitHub's kanban feature is used for task tracking and scheduling throughout the project. This project is owned by the HDLGen-ChatGPT organisation where the HDLGen-ChatGPT git repository also resides.

Weekly meetings, lasting one hour each, were held each week between the project supervisor and student. These sessions were dedicated to brainstorming, planning, feedback exchange and problem-solving. These meetings proved integral to the project's success, as the project could deviate to account for unforeseen developments.

The project commenced by researching the build process for overlays using the Vivado EDA for the PYNQ-Z2 platform. In early October, a proof-of-concept code was developed to determine project feasibility. In late October, early remote platform feasibility testing took place. By January, a basic GUI was created. From January to March, the PYNQ SoC Builder was continuously improved upon and tested rigorously using a wide range of designs and architectures. In March, a new, more intuitive GUI was developed to effectively communicate how to use the program and to provide useful feedback to the user.

1.4 Structure of Thesis

The structure of the thesis is as follows:

- Chapter 1 introduces this thesis and provides a background of the project.
- Chapter 2 is a literature review and details related work in this field, such as HDLGen-ChatGPT, the use of FPGA software and hardware, and related existing remote FPGA infrastructure
- Chapter 3 details the software implementation of the PYNQ SoC Builder, focusing on both the user-interface and backend logic of the software package.
- Chapter 4 describes the implementation and functionality of the remote FPGA environment and details the steps required to implement the system.
- Chapter 5 presents five HDL designs generated using the HDLGen-ChatGPT package and outlines the configuration and output of the PYNQ SoC Builder for each project
- This report concludes with a chapter comprised of conclusions and outlines details for future work.

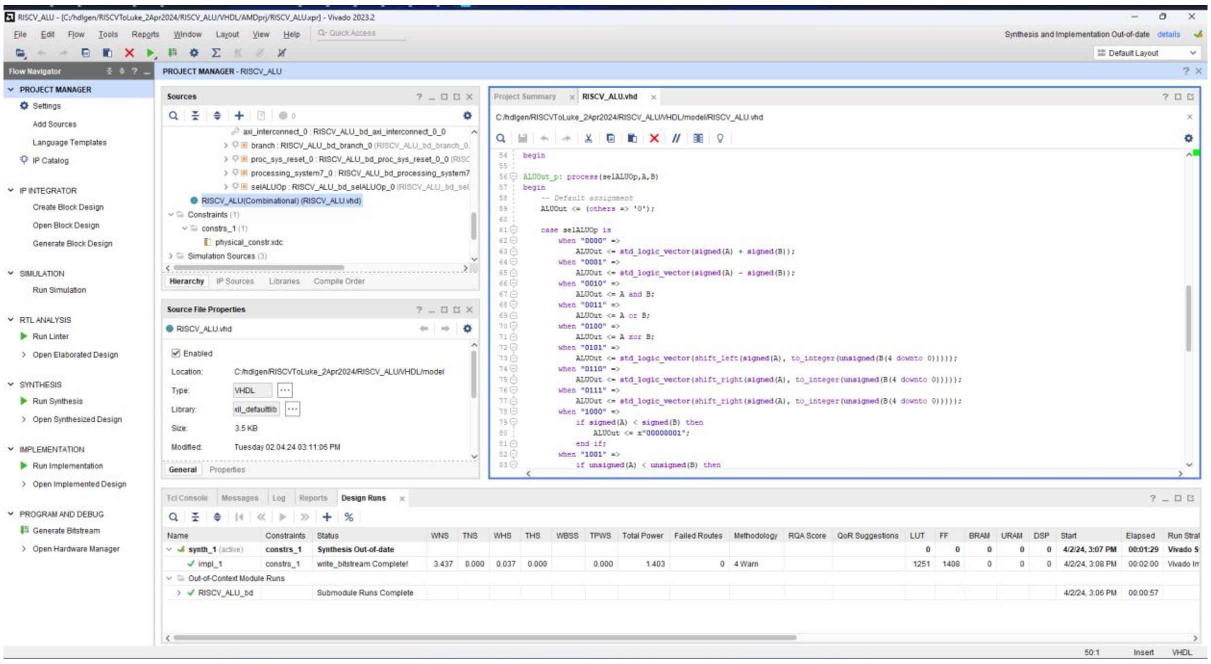
2. Background

2.1 Overview

In this section, related research and technologies are described in detail. In section 2.2, a background of the AMD Vivado Design Suite, an EDA tool suite for digital hardware design is provided and its relevance with respect to this project. As this project is an extension of HDLGen-ChatGPT, its current implementation is explored in section 2.3. In section 2.4, the TuL PYNQ Z2 FPGA is examined in detail describing its various features and benefits to this project. Finally, in section 2.5, an existing platform which provides remote access to FGPA hardware to students, called Vicilogic, is detailed.

2.2 Vivado Design Suite

The Vivado Design Suite by Xilinx/AMD is an EDA tool suite for design entry, synthesis, place and route, verification, and simulation [x]. The Vivado integrated design environment (IDE) is intended for the development of SoCs and FPGA overlays. Vivado IDE supports both VHDL and Verilog HDL languages. Vivado also supports C, C++ and SystemC using its high-level synthesis (HLS) platform. Figure 4 is a screen capture of the Vivado IDE when designing a RISC-V arithmetic logic unit (ALU). The sidebar on the left offers access to the various tools available in Vivado.



```
axi_interconnect_0 : RISCV_ALU_bd_axi_interconnect_0_0
  > branch : RISCV_ALU_bd_branch_0 (RISCV_ALU_bd_branch_0
  > proc_sys_reset_0 : RISCV_ALU_bd_proc_sys_reset_0_0 (RISC
  > processing_system7_0 : RISCV_ALU_bd_processing_system7
  > selALUOp : RISCV_ALU_bd_selALUOp_0 (RISCV_ALU_bd_sel
● RISCV_ALU(Combinational) (RISCV_ALU.vhd)
v Constraints (1)
  v constrs_1 (1)
      physical_constr.xdc
> Simulation Sources (3)
```

RISCV_ALU.vhd

Enabled: ✓
Location: C:/hdlgen/RISCVToLuke_2Apr2024/RISCV_ALU/VHDL/model
Type: VHDL
Library: xil_defaultlib
Size: 3.5 KB
Modified: Tuesday 02.04.24 03:11:06 PM

C:/hdlgen/RISCVToLuke_2Apr2024/RISCV_ALU/VHDL/model/RISCV_ALU.vhd

```vhdl
begin

ALUOut_p: process(selALUOp,A,B)
begin
    -- Default assignment
    ALUOut <= (others => '0');

    case selALUOp is
        when "0000" =>
            ALUOut <= std_logic_vector(signed(A) + signed(B));
        when "0001" =>
            ALUOut <= std_logic_vector(signed(A) - signed(B));
        when "0010" =>
            ALUOut <= A and B;
        when "0011" =>
            ALUOut <= A or B;
        when "0100" =>
            ALUOut <= A xor B;
        when "0101" =>
            ALUOut <= std_logic_vector(shift_left(signed(A), to_integer(unsigned(B(4 downto 0)))));
        when "0110" =>
            ALUOut <= std_logic_vector(shift_right(signed(A), to_integer(unsigned(B(4 downto 0)))));
        when "0111" =>
            ALUOut <= std_logic_vector(shift_right(signed(A), to_integer(unsigned(B(4 downto 0)))));
        when "1000" =>
            if signed(A) < signed(B) then
                ALUOut <= x"00000001";
            end if;
        when "1001" =>
            if unsigned(A) < unsigned(B) then
```

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR Suggestions	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strat
v ✓ **synth_1** (active)	**constrs_1**	**Synthesis Out-of-date**												**0**	**0**	**0**	**0**	**0**	**4/2/24, 3:07 PM**	**00:01:29**	**Vivado S**
✓ impl_1	constrs_1	write_bitstream Complete!	3.437	0.000	0.037	0.000		0.000	1.403	0	4 Warn			1251	1408	0	0	0	4/2/24, 3:08 PM	00:02:00	Vivado Im
v Out-of-Context Module Runs																					
> ✓ RISCV_ALU_bd		Submodule Runs Complete																	4/2/24, 3:06 PM	00:00:57	

Figure 4 Screen Capture of Vivado IDE 2023.2

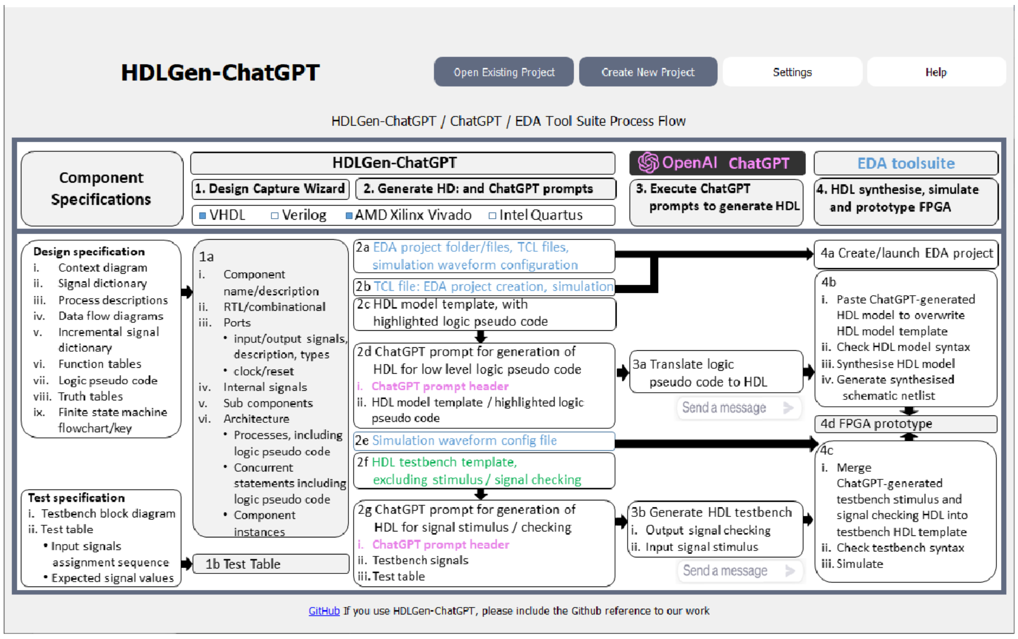
To create an FPGA overlay in the Vivado IDE, the following steps are executed in order:

1. An HDL model (in Verilog or VHDL) is captured, from scratch or using HDLGen-ChatGPT.
2. (Optional) An HDL testbench is designed to simulate the model using the simulation tool.
3. A block design is created using the IP Integrator tool. The ZYNQ FPGA processing system, the user-defined component, and all other required intellectual property (IP) is imported.
4. An HDL wrapper is generated which translates the block design into a source file which can then be used to build the actual design. [x]
5. The design is synthesized, translating the high-level representation into Register-Transfer Level (RTL) netlist. The netlist uses HDL to describe the design's functionality using combination circuits (e.g. AND, OR, NOT, XOR) and sequential elements (e.g. D-flip flops).
6. The design is implemented for FPGA hardware. Implementation is the placement and routing tool for the Xilinx platform. It is the process of configuring the target FPGA resources and routing connections. A bitstream file (.bit) is generated from this step which is then used by the PYNQ board to configure the FPGA.
7. Finally, the block design Tcl file and hardware handoff files, required by the PYNQ, are exported.

2.3 HDLGen-ChatGPT

HDLGen-ChatGPT is an open-source "client-based Python application that automates HDL based SoC capture and implementation" (Byrne, J.P. 2023). This application has been developed to greatly reduce the time required to write FPGA software using VHDL or Verilog HDL programming languages. This project is notable as it has great potential for use in education with respect to SoC and digital system design.

The HDLGen-ChatGPT project was conceived with the goal of simplifying and automating the FPGA prototyping development process. It uses ChatGPT to convert pseudocode (a notation resembling a generic programming language) into HDL code. This program greatly reduces the time-to-prototype as less time is spent writing complex HDL code. The homepage of HDLGen-ChatGPT presents the full flow of the application in figure 5.


HDLGen-ChatGPT
Open Existing Project
Create New Project
Settings
Help
HDLGen-ChatGPT / ChatGPT / EDA Tool Suite Process Flow
Component Specifications
HDLGen-ChatGPT
1. Design Capture Wizard
2. Generate HD: and ChatGPT prompts
■ VHDL □ Verilog ■ AMD Xilinx Vivado □ Intel Quartus
OpenAI ChatGPT
3. Execute ChatGPT prompts to generate HDL
EDA toolsuite
4. HDL synthesise, simulate and prototype FPGA

Design specification
i. Context diagram
ii. Signal dictionary
iii. Process descriptions
iv. Data flow diagrams
v. Incremental signal dictionary
vi. Function tables
vii. Logic pseudo code
viii. Truth tables
ix. Finite state machine flowchart/key

Test specification
i. Testbench block diagram
ii. Test table
• Input signals assignment sequence
• Expected signal values

1a
i. Component name/description
ii. RTL/combinational
iii. Ports
• input/output signals, description, types
• clock/reset
iv. Internal signals
v. Sub components
vi. Architecture
• Processes, including logic pseudo code
• Concurrent statements including logic pseudo code
• Component instances

1b Test Table

2a EDA project folder/files, TCL files, simulation waveform configuration
2b TCL file: EDA project creation, simulation
2c HDL model template, with highlighted logic pseudo code
2d ChatGPT prompt for generation of HDL for low level logic pseudo code
i. ChatGPT prompt header
ii. HDL model template / highlighted logic pseudo code
2e Simulation waveform config file
2f HDL testbench template, excluding stimulus / signal checking
2g ChatGPT prompt for generation of HDL for signal stimulus / checking
i. ChatGPT prompt header
ii. Testbench signals
iii. Test table

3a Translate logic pseudo code to HDL
Send a message
3b Generate HDL testbench
i. Output signal checking
ii. Input signal stimulus
Send a message

4a Create/launch EDA project
4b
i. Paste ChatGPT-generated HDL model to overwrite HDL model template
ii. Check HDL model syntax
iii. Synthesise HDL model
iv. Generate synthesised schematic netlist
4d FPGA prototype
4c
i. Merge ChatGPT-generated testbench stimulus and signal checking HDL into testbench HDL template
ii. Check testbench syntax
iii. Simulate

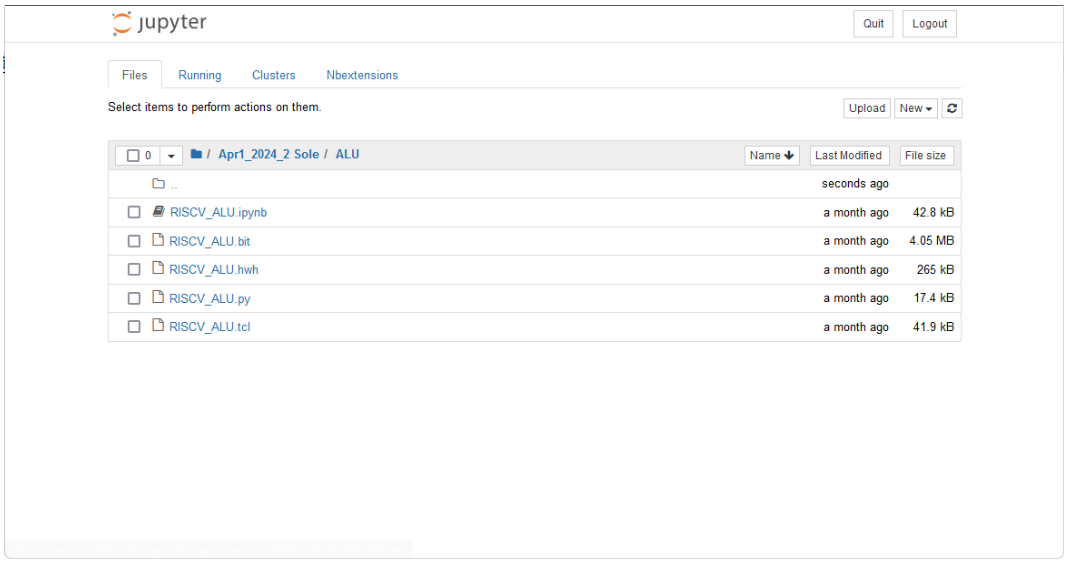
GitHub If you use HDLGen-ChatGPT, please include the Github reference to our work

Figure 5 HDLGen-ChatGPT Homepage

However, HDLGen-ChatGPT does not currently capture the complete development cycle for deploying projects to a target FPGA device. Currently, the user is required to export the finalised project from HDLGen-ChatGPT and complete synthesis, implementation and bitstream generation steps manually.

2.4 PYNQ FPGA

PYNQ (Python Productivity for ZYNQ) is an open-source project that combines productivity of Python with the capabilities of programmable logic devices, specifically the Xilinx Zynq SoCs. PYNQ provides a framework that simplifies the development process by offering an interface accessible through Jupyter Notebooks.



Select items to perform actions on them.

0 / Apr1_2024_2 Sole / ALU

Name	Last Modified	File size
..	seconds ago	
RISCV_ALU.ipynb	a month ago	42.8 kB
RISCV_ALU.bit	a month ago	4.05 MB
RISCV_ALU.hwh	a month ago	265 kB
RISCV_ALU.py	a month ago	17.4 kB
RISCV_ALU.tcl	a month ago	41.9 kB

Figure 6 Jupyter Notebook running on PYNQ FPGA

A plan view of the PYNQ-Z2 board is presented in figure 7. The PYNQ-Z2 is the second revision in the PYNQ board family. It is lower cost than its predecessor, and more powerful. Developed by TuL, it boasts the following features:

- Dual-Core ARM Cortex-A9 Processor.
- 1.3 reconfigurable gates.
- 512MB DDR3 Memory and 128Mbit Flash.
- HDMI In/Out, 10/100/1000 Ethernet, Arduino, and Raspberry Pi Shield Connectors.
- 4 binary LEDs, 2 RBG LEDs, 4 Push-Buttons and 2 Slide Switches.

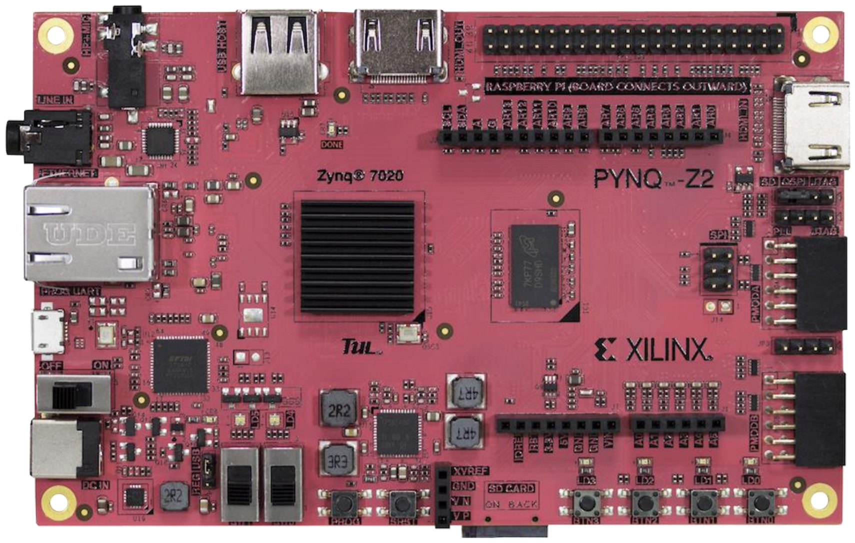


Figure 7 Plan View of PYNQ Z2 FPGA Board

2.5 viciLogic

Remote laboratories have become popular within engineering education, and other fields (Viegas, C. et al, 2018). Remote laboratories can give students access to more hardware and software enabling a hands-on approach to education. In 2014, viciLogic, was created. viciLogic is described as an “effective pedagogical solution and scalable solution for online technology enhanced learning” (Morgan et al 2014) viciLogic consists of several GUIs for students to interact and learn with. viciLogic offers an intuitive approach to learning and assessing digital logic hardware design using real FPGA hardware in real-time. viciLogic is unique in this regard as it does not simulate digital logic like other online learning tools. Although viciLogic does not currently offer students and users the ability to deploy their own HDL models to the FPGA hardware, there is a large library of digital components. viciLogic currently offers users ability to interact with a full RISC-V core implementation using C or RISC-V assembly, and all sub-components such as memory modules, ALU and more. viciLogic also has elementary models such as counters, registers, adders and first-in first-out (FIFO) modules.

3. PYNQ SoC Builder Software Implementation

3.1 Overview

This chapter will discuss the HDLGen-ChatGPT project data and the software implementation of the PYNQ SoC Builder. The PYNQ SoC Builder relies on the HDLGen-ChatGPT project file to build projects. This chapter will describe the software logic and the files produced by the PYNQ SoC Builder. A combination of flow diagrams and explanatory text will be used to describe the application's architecture.

3.2 Python Environment, Dependencies, and Installation

The PYNQ SoC Builder has been designed to work seamlessly with HDLGen-ChatGPT Python environments. This is achieved by prioritizing compatibility and avoiding dependency conflicts. The tool is designed to support the latest supported Python version by HDLGen-ChatGPT (Python 3.10.0) and all subsequent versions as well (until Python 3.12.3).

All dependencies are specified in a *requirements.txt* text file in the root folder of the PYNQ SoC Builder repository. SoC Builder utilises the following libraries:

- **Custom Tkinter:** Custom Tkinter is a modified edition of the Tkinter library. Custom Tkinter is a GUI toolkit for Python developers to design aesthetically pleasing user interfaces. It extends the functionality of Tkinter by adding more widgets and tools.
- **Psutil:** is a library for process and system monitoring in Python. It is used in this application to monitor the Vivado executable and to close it.
- **Tkhtmlview:** is a library for displaying HTML pages in Tkinter
- **Markdown:** is a library for parsing and displaying Markdown text. It is used in combination with Tkhtmlview to display Markdown to users.
- **Nbformat:** Notebook Format is a Python library for the creation of Jupyter Notebook files in Python. It is used by the PYNQ SoC Builder to produce Jupyter Notebook applications for the PYNQ platform.
- **Pysftp:** provides a simple interface for the secure file transfer protocol (SFTP). It offers a layer of abstraction and allows files to be transferred quickly and easily using Python.

The PYNQ SoC Builder can be installed and executed with ease using Git and Python 3.10. The steps are largely the same as HDLGen-ChatGPT. Table 1 lists the available resources related to application installation.

Description	Link
Text Installation Guide	Project README (GitHub)
Video Installation Guide	Video (YouTube)

Table 1 SoC Builder Installation Guides

3.3 HDLGen-ChatGPT Data

The PYNQ SoC Builder loads a HDLGen-ChatGPT project by parsing the hierarchal Extensible Markup Language (XML) data saved by HDLGen-ChatGPT. This data is stored as a *.hdlgen* file in the *HDLGenPrj* folder inside a project's file structure. Figure 8 presents the structure of the HDLGen-ChatGPT project XML file.

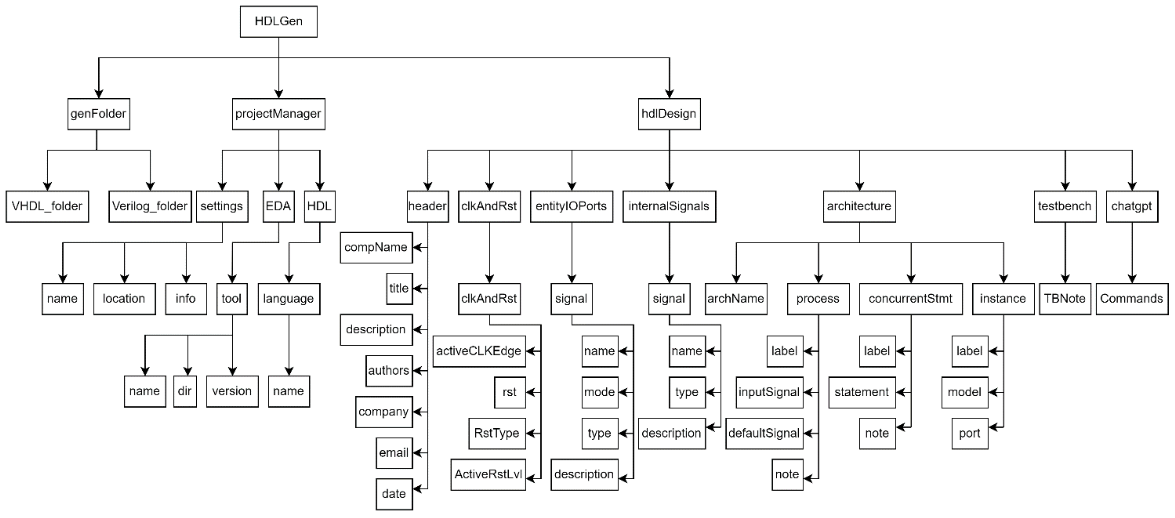


Figure 8 Simplified File Structure of a Typical HDLGen-ChatGPT project (Byrne, 2023)

The 'HDLGen/projectManager/settings' tag contains human-readable text description of the component which is displayed within the SoC Builder's menu for user reference. External and internal connections of the design are defined under the 'entityIOPorts' and 'internalSignals' tags. Finally, the test plan is stored under the 'testbench' tag, and it utilised by the SoC Builder later in the build process to generate a companion driver application for the component.

Data under the 'HDLGen/projectManager/settings' tag provides a text-based description of the component. This data is presented inside the builder menu to remind the user. Data under the 'hdlDesign/entityIOPorts' and 'hdlDesign/internalSignals' tags describe the external and internal connections of the component. The internal operation (architecture of the design) of the component is not relevant to the SoC Builder and is not parsed. Test plan data is stored at 'hdlDesign/testbench' and it used by the SoC Builder to generate a driver application later in the build process.

3.4 Software Architecture

The software is comprised of two distinct components, front-end components, and back-end components. Front-end components relate to the GUI and are responsible for recording input from the user and returning feedback. The main.py file is the entry-point of the application. The main.py file is responsible for loading initial variables and classes required and loading the "Open Project" GUI window. Figure 9 visualises the code structure, separating GUI elements and backend elements.

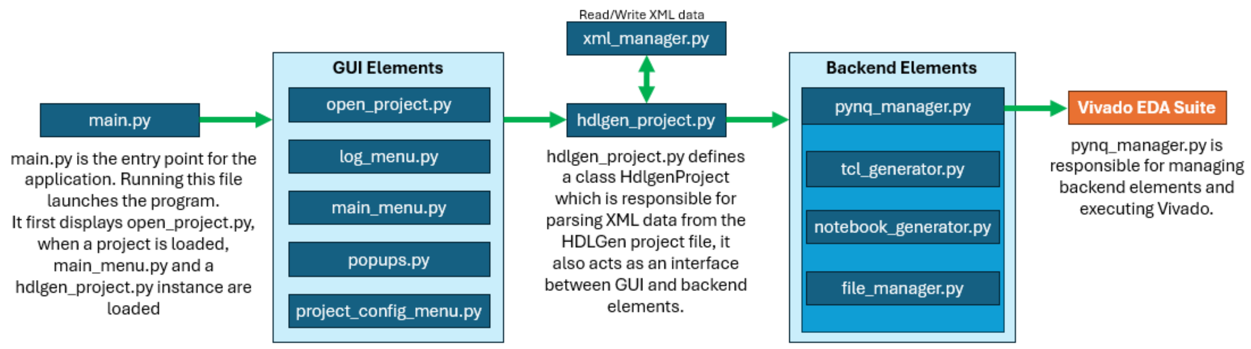

Read/Write XML data
xml_manager.py
main.py
main.py is the entry point for the application. Running this file launches the program. It first displays open_project.py, when a project is loaded, main_menu.py and a hdlgen_project.py instance are loaded
GUI Elements
open_project.py
log_menu.py
main_menu.py
popups.py
project_config_menu.py
hdlgen_project.py
hdlgen_project.py defines a class HdlgenProject which is responsible for parsing XML data from the HDLGen project file, it also acts as an interface between GUI and backend elements.
Backend Elements
pynq_manager.py
tcl_generator.py
notebook_generator.py
file_manager.py
Vivado EDA Suite
pynq_manager.py is responsible for managing backend elements and executing Vivado.

Figure 9 Architecture of PYNQ SoC Builder

Figure 10 is a visual representation of the project's file structure. The main.py file is at the topmost level of the project as is executed to start the program. All application source code is stored in the application folder. Board files provided by TuL (the manufacturer of the PYNQ Z2 board) utilised by Vivado are stored in the board_files folder.

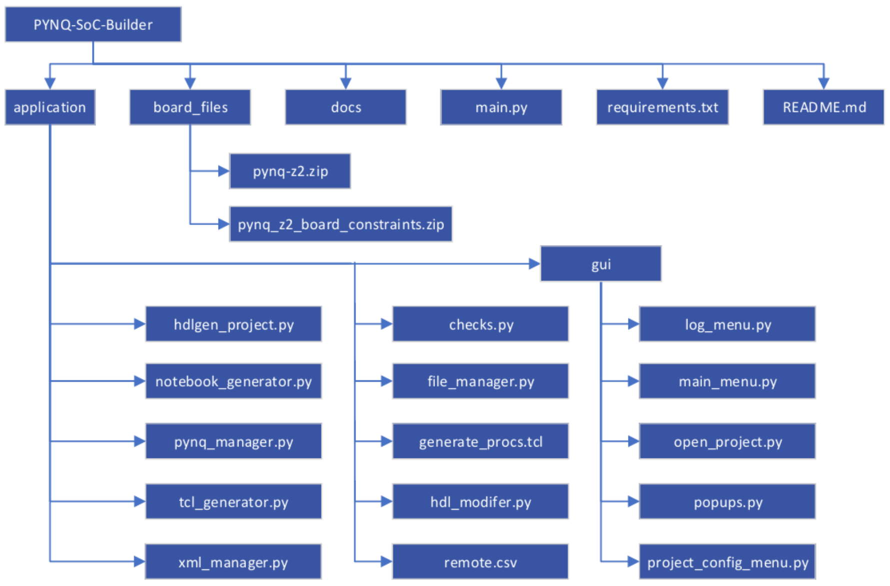


Figure 10 File Structure of the PYNQ SoC Builder Application

Table 2 lists all new classes created in the PYNQ SoC Builder application and contains a description of each class.

Class	Description
Application (main.py)	Entrypoint for application. Launches Tkinter toolkit.
LogTabView (log_menu.py)	Tabbed frame for presenting project summary, test plan, and logging information
LogBoxTab (log_menu.py)	Logging window class with API for adding to log box
SummaryTab (log_menu.py)	Renders project data for user reference
OpenProjectPage (open_project.py)	Renders Open Project Prompt
SidebarMenu (main_menu.py)	Scrollable frame containing labels and buttons of sidebar menu
ConfigMenu (main_menu.py)	Parent frame of Config Tab View Menu
LogMenu (main_menu.py)	Parent frame of Log Tab View Menu
MainPage (main_menu.py)	Master frame containing all widgets once a project is loaded
Alert_Window (popups.py)	Renders an alert pop-up window
Dialog_Window (popups.py)	Renders a dialog pop up window (Yes/No prompt)
MarkdownWindow (popups.py)	Renders Markdown text and presents to user
ConfigTabView (project_config_menu.py)	Frame which resides inside the ConfigMenu object, and renders a tabbed frame, parent object of each tab.
BuildStatusTab (project_config_menu.py)	Renders the build status information in the GUI under Build Status Tab. Child frame of ConfigTabView
PortConfigTab (project_config_menu.py)	Renders the Project Config Tab in the GUI. Child frame of ConfigTabView
IOConfigTab (project_config_menu.py)	Frame intended for use in future if board IO is expanded to use more ports – HDMI, Switches etc.

ConfigMenu (project_config_menu.py)	Parent Frame of all frames in Config Menu region
DashesInHDLFileError (checks.py)	Exception object for ensuring ChatGPT output is copied to HDL model.
File_Manager (file_manager.py)	Responsible for handling output files, can find, rename and copy files from Vivado project, upload over SFTP to PYNQ Board
HdlgenProject (hdlgen_project.py)	Class which loads and parses HDLGen-ChatGPT projects.
Pynq_Manager (pynq_manager.py)	Wrapper for backend logic components. Creates abstraction so that functions can be called easily from GUI.

Table 2 All Class Names and Descriptions in the SoC Program

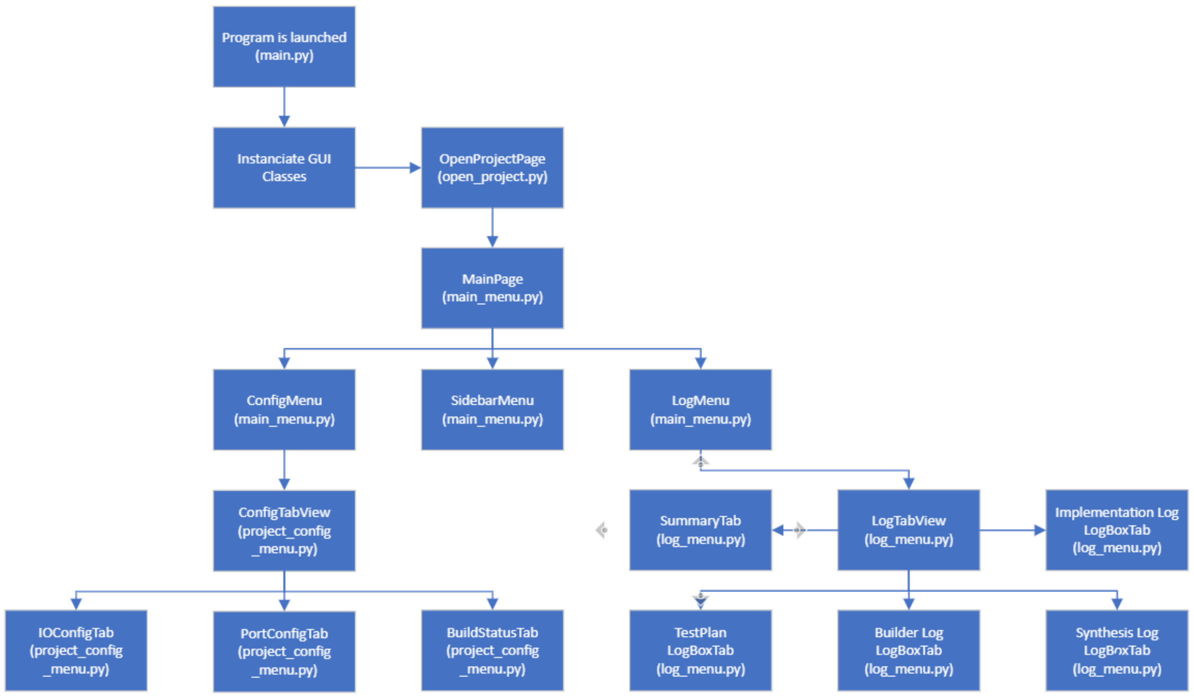
3.5 Front End Logic

3.5.1 Overview

In this sub-section, the front-end behaviour and logic of the PYNQ SoC Builder is detailed. There are two primary windows which are viewed in the GUI, “Open Project” and “Main Menu”. The open project window is presented when the application is first loaded. It prompts the user to select a HDLGen-ChatGPT project. The main menu screen is the second window, which is only made visible once a valid project has been loaded.

3.5.2 Open Project

When the program is launched all the GUI classes are created. Figure 11 describes the front-end hierarchy of the program. The front-end GUI utilises object-oriented principles such as inheritance and polymorphism as per Custom Tkinter documentation recommendations.



Program is launched (main.py)
Instanciate GUI Classes
OpenProjectPage (open_project.py)
MainPage (main_menu.py)
ConfigMenu (main_menu.py)
SidebarMenu (main_menu.py)
LogMenu (main_menu.py)
ConfigTabView (project_config _menu.py)
SummaryTab (log_menu.py)
LogTabView (log_menu.py)
Implementation Log LogBoxTab (log_menu.py)
IOConfigTab (project_config _menu.py)
PortConfigTab (project_config _menu.py)
BuildStatusTab (project_config _menu.py)
TestPlan LogBoxTab (log_menu.py)
Builder Log LogBoxTab (log_menu.py)
Synthesis Log LogBoxTab (log_menu.py)

Figure 11 GUI Class Hierarchy

Once the program has fully loaded each front-end object, the ‘Open Project’ menu is rendered. Figure 12 presents a screen capture of this menu. Figure 13 presents the dataflow diagram of this menu. It describes the behaviour of the front-end elements. Users are prompt to open a HDLGen-ChatGPT project of their choice to proceed.

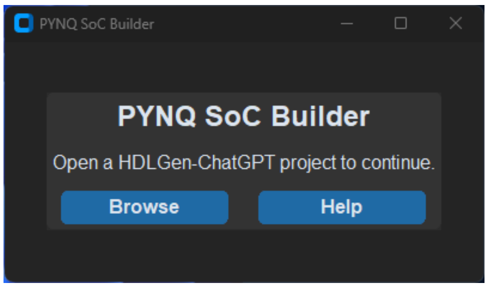
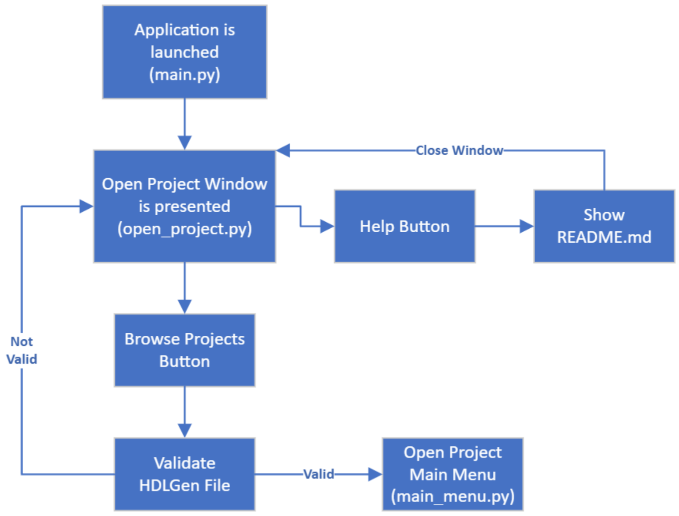


Figure 12 'Open Project' menu of the PYNQ SoC Builder GUI

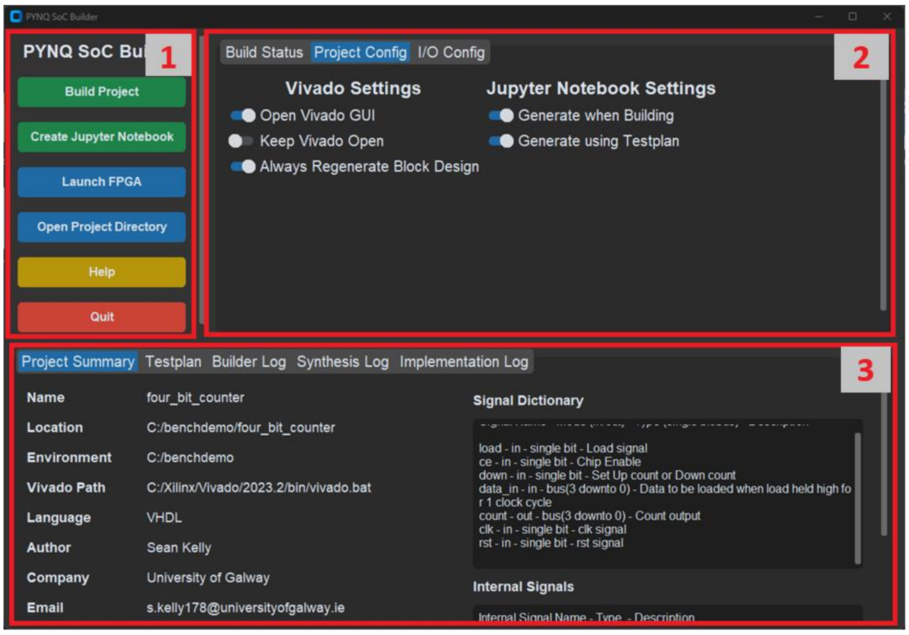

Application is launched (main.py)
Open Project Window is presented (open_project.py)
Close Window
Help Button
Show README.md
Browse Projects Button
Not Valid
Validate HDLGen File
Valid
Open Project Main Menu (main_menu.py)
Figure 13 Open Project Dialog Logic

3.5.3 Main Menu

If a valid HDLGen-ChatGPT project is selected in the 'Open Project' menu, the PYNQ SoC Builder will proceed to the main menu. The main menu consists of multiple menus which serve different purposes. In figure 14, the three distinct regions of the main menu are indicated. The three sections are:

1. **Sidebar menu** is responsible for controlling the entire application – triggering builds or notebook generation, deploying projects to FPGA hardware, opening the project, opening documentation, and quitting the application.
2. **Build menu** is responsible for configuring settings related to the build status, build configuration and board IO configuration.
3. **Project information region** is responsible for presenting key information to the user about the HDLGen-ChatGPT project for the user's reference (such as the component summary and project test plan) as well as presenting logging information from both the SoC Builder and Vivado.

The build menu section has three tabs: Build Status, Project Config, and I/O Config. Figure 15 and 16 present screen captures of both the build status, and I/O configuration menus respectively.

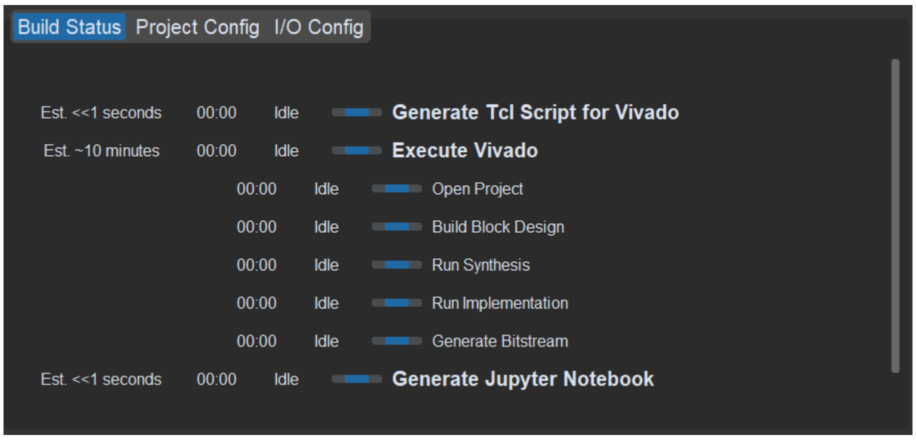


Name	four_bit_counter
Location	C:/benchdemo/four_bit_counter
Environment	C:/benchdemo
Vivado Path	C:/Xilinx/Vivado/2023.2/bin/vivado.bat
Language	VHDL
Author	Sean Kelly
Company	University of Galway
Email	s.kelly178@universityofgalway.ie

```
load - in - single bit - Load signal
ce - in - single bit - Chip Enable
down - in - single bit - Set Up count or Down count
data_in - in - bus(3 downto 0) - Data to be loaded when load held high fo
r 1 clock cycle
count - out - bus(3 downto 0) - Count output
clk - in - single bit - clk signal
rst - in - single bit - rst signal
```

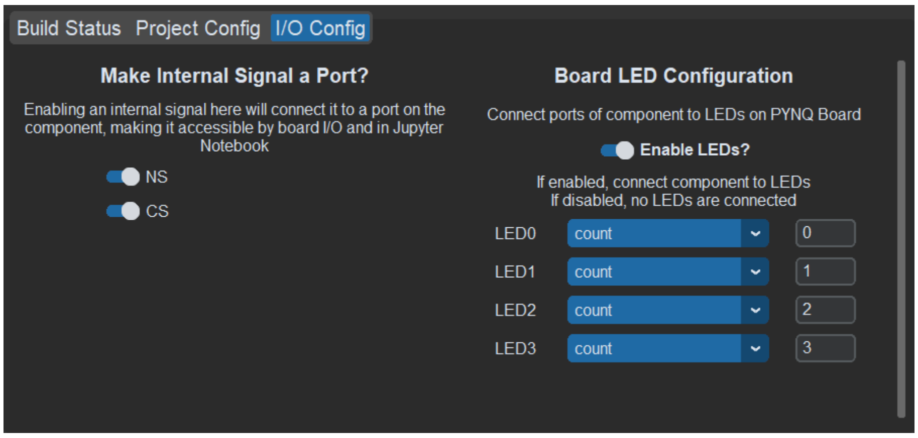
```
Internal Signal Name - Type  - Description
```

Figure 14 Labelled screen capture of Main Menu



```
Est. <<1 seconds   00:00   Idle   Generate Tcl Script for Vivado
Est. ~10 minutes   00:00   Idle   Execute Vivado
                        00:00   Idle   Open Project
                        00:00   Idle   Build Block Design
                        00:00   Idle   Run Synthesis
                        00:00   Idle   Run Implementation
                        00:00   Idle   Generate Bitstream
Est. <<1 seconds   00:00   Idle   Generate Jupyter Notebook
```

Figure 15 Build status menu in PYNQ SoC Builder


Build Status Project Config I/O Config

Make Internal Signal a Port?

Enabling an internal signal here will connect it to a port on the component, making it accessible by board I/O and in Jupyter Notebook

NS ●

CS ●

Board LED Configuration

Connect ports of component to LEDs on PYNQ Board

Enable LEDs? ●

If enabled, connect component to LEDs
If disabled, no LEDs are connected

LED0	count	0
LED1	count	1
LED2	count	2
LED3	count	3

Figure 16 I/O configuration menu in PYNQ SoC Builder

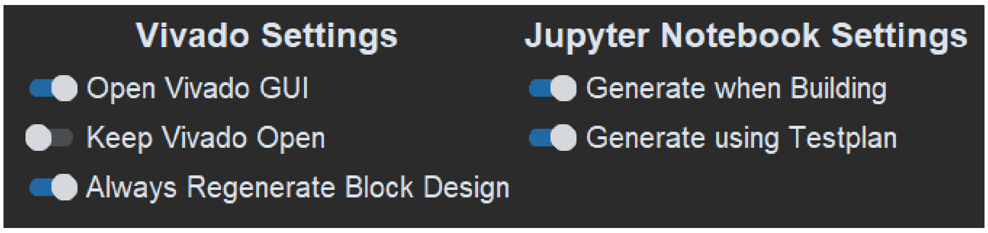
3.6 Configuring a Project

3.6.1 Overview

This section provides a guide to configuring projects within the PYNQ SoC Builder GUI. It details the functionality of each configurable option and its impact on the build process. All configurable settings are accessible within the build menu, specifically under the "Project Config" and "I/O Config" tabs.

3.6.2 Project Config

Figure 17 presents the project configuration tab, and default configuration. It has five options which change the behaviour of the build process.


Vivado Settings

● Open Vivado GUI

○ Keep Vivado Open

● Always Regenerate Block Design

Jupyter Notebook Settings

● Generate when Building

● Generate using Testplan

Figure 17 Options present under Project Config tab with default configuration

Vivado Settings

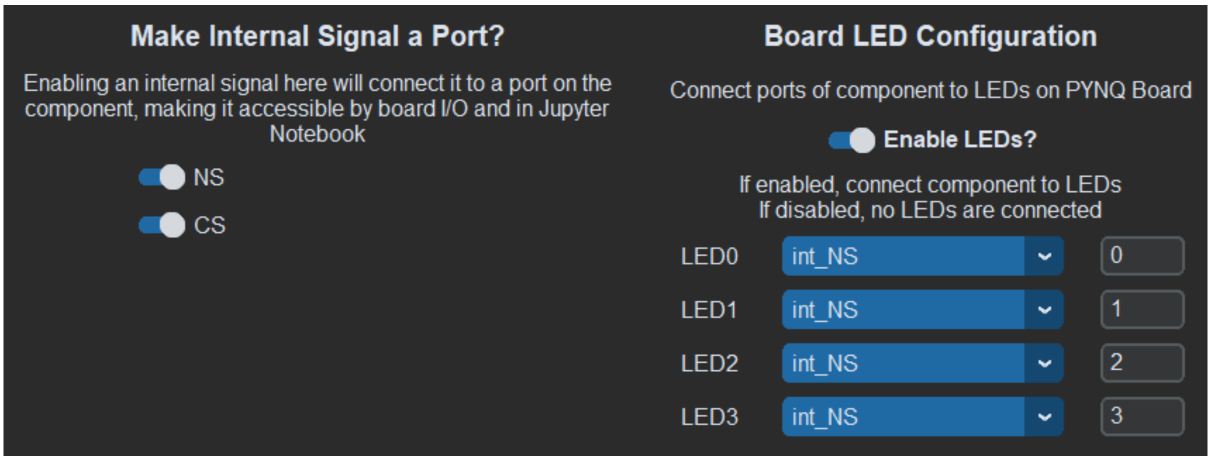
- **Open Vivado GUI:** This option determines if the Vivado GUI should be visible or not. Hiding the GUI can improve performance on lower power hardware.
- **Keep Vivado Open:** This option, if enabled, will keep Vivado open after the build process is completed. This option is intended for debugging. For instance, the user may wish to validate the generated block design before continuing to the next step.
- **Always Regenerate Block Design:** If this option is disabled, and a block design exists, it will use the existing block design. Otherwise, a new block design is generated.

Jupyter Notebook Settings

- **Generate when Building:** If enabled, the SoC Builder will automatically generate a Jupyter Notebook driver application when building. If disabled, no notebook is generated.
- **Generate using Testplan:** If enabled, and a valid test plan is found, the Jupyter Notebook driver application will add cells to run each test case. If disabled or an invalid test plan is found, no test cases are generated, however, markdown descriptions and GUI-controller will generate.

3.6.3 IO Config

Figure 18 presents the 'IO Config' tab using a sample counter project. Under this tab, two major features are available: the ability to make internal signals available at the top level, and ability to connect component to board LEDs.


Make Internal Signal a Port?

Enabling an internal signal here will connect it to a port on the component, making it accessible by board I/O and in Jupyter Notebook

NS

CS

Board LED Configuration

Connect ports of component to LEDs on PYNQ Board

Enable LEDs?

If enabled, connect component to LEDs
If disabled, no LEDs are connected

LED0	int_NS	0
LED1	int_NS	1
LED2	int_NS	2
LED3	int_NS	3

Figure 18 Options present under IO Config tab

Internal Signal Configuration

Standard internal signals (buses of 1 or more bits) can be made available at the top level of the component. If a signal is compatible, a switch is presented in the GUI. Once enabled, internal signals can also be connected to LEDs, as demonstrated in figure 18.

Board LED Configuration

To configure LED connections, the user selects a signal using the dropdown menus. If the signal selected is greater than 1-bit wide, an input box prompting the user to enter a number is rendered. If the user does not enter a value, or the value is invalid, the signal bit will default to zero.

3.7 Build Process

3.7.1 Overview

In this chapter, the build procedure (backend logic) is outlined and described in detail. This chapter will describe the various actions undertaken by the PYNQ SoC Builder to produce PYNQ Z2 FPGA Overlays.

The PYNQ SoC Builder build process has five primary stages:

1. Save SoC Builder configuration
2. Generation of a Tcl script to be executed by Vivado containing all build steps.
3. Vivado is prompted to execute the Tcl script produced in step 1.
4. A Jupyter Notebook driver application is generated.
5. All required files are copied to the /PYNQBuild/output folder for user to access easily.

3.7.2 Generate Tcl Script

To complete the Vivado build steps outlined in section 2.2, the tool command language (Tcl) is used. A procedurally executed script can be generated using Python and then executed by Vivado. Figure 19 presents the dataflow diagram of the Tcl Generator function (which is located at tcl_generator.py). The various decisions are determined by the user's configuration of the PYNQ SoC Builder before starting a build.

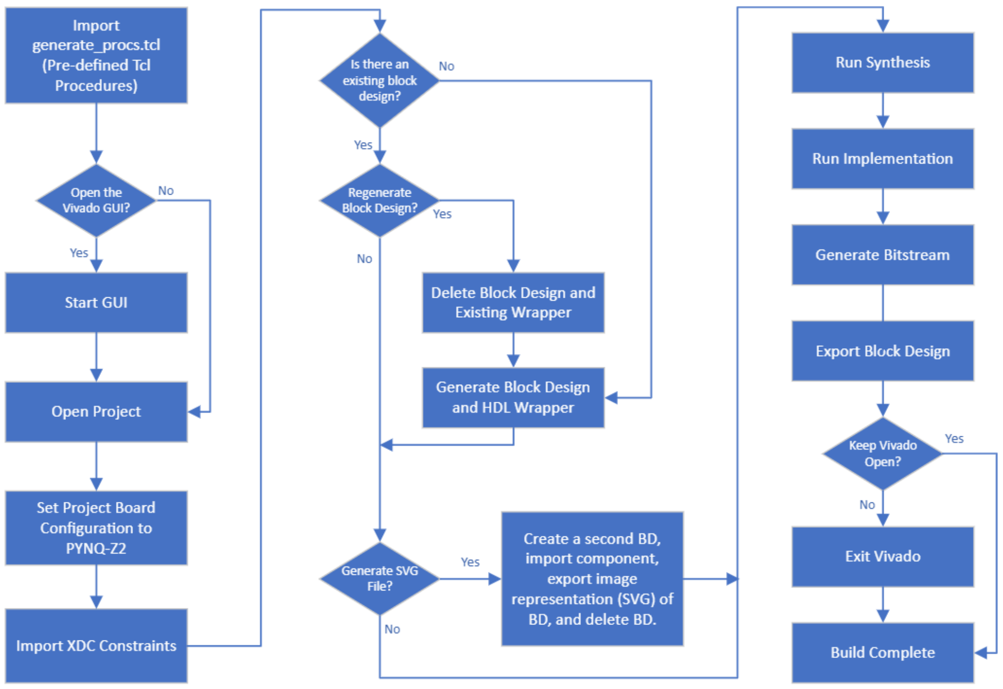

Import generate_procs.tcl (Pre-defined Tcl Procedures)
Open the Vivado GUI?
No
Yes
Start GUI
Open Project
Set Project Board Configuration to PYNQ-Z2
Import XDC Constraints
Is there an existing block design?
No
Yes
Regenerate Block Design?
Yes
No
Delete Block Design and Existing Wrapper
Generate Block Design and HDL Wrapper
Generate SVG File?
Yes
No
Create a second BD, import component, export image representation (SVG) of BD, and delete BD.
Run Synthesis
Run Implementation
Generate Bitstream
Export Block Design
Keep Vivado Open?
Yes
No
Exit Vivado
Build Complete

Figure 19 Dataflow diagram of the Tcl Generator function

Figure 20 presents a flowchart describing the steps required to generate a new block design.

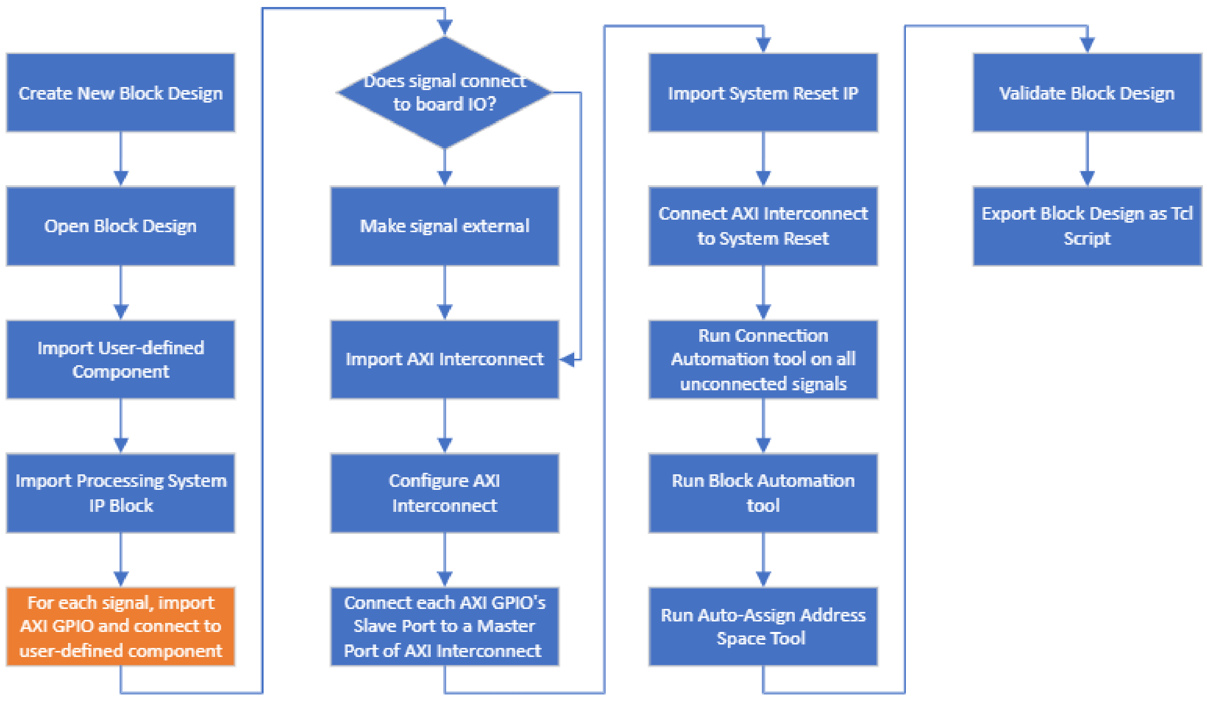

Create New Block Design
Open Block Design
Import User-defined Component
Import Processing System IP Block
For each signal, import AXI GPIO and connect to user-defined component
Does signal connect to board IO?
Make signal external
Import AXI Interconnect
Configure AXI Interconnect
Connect each AXI GPIO's Slave Port to a Master Port of AXI Interconnect
Import System Reset IP
Connect AXI Interconnect to System Reset
Run Connection Automation tool on all unconnected signals
Run Block Automation tool
Run Auto-Assign Address Space Tool
Validate Block Design
Export Block Design as Tcl Script

Figure 20 Flowchart of block design creation steps

Figure 21 presents the steps required to connect the user-defined component to the ZYNQ processing system in greater detail. The AXI GPIO component is limited to a maximum bus width of 32 bits, and hence, for buses greater than this upper limit, multiple AXI GPIO IP blocks are imported.

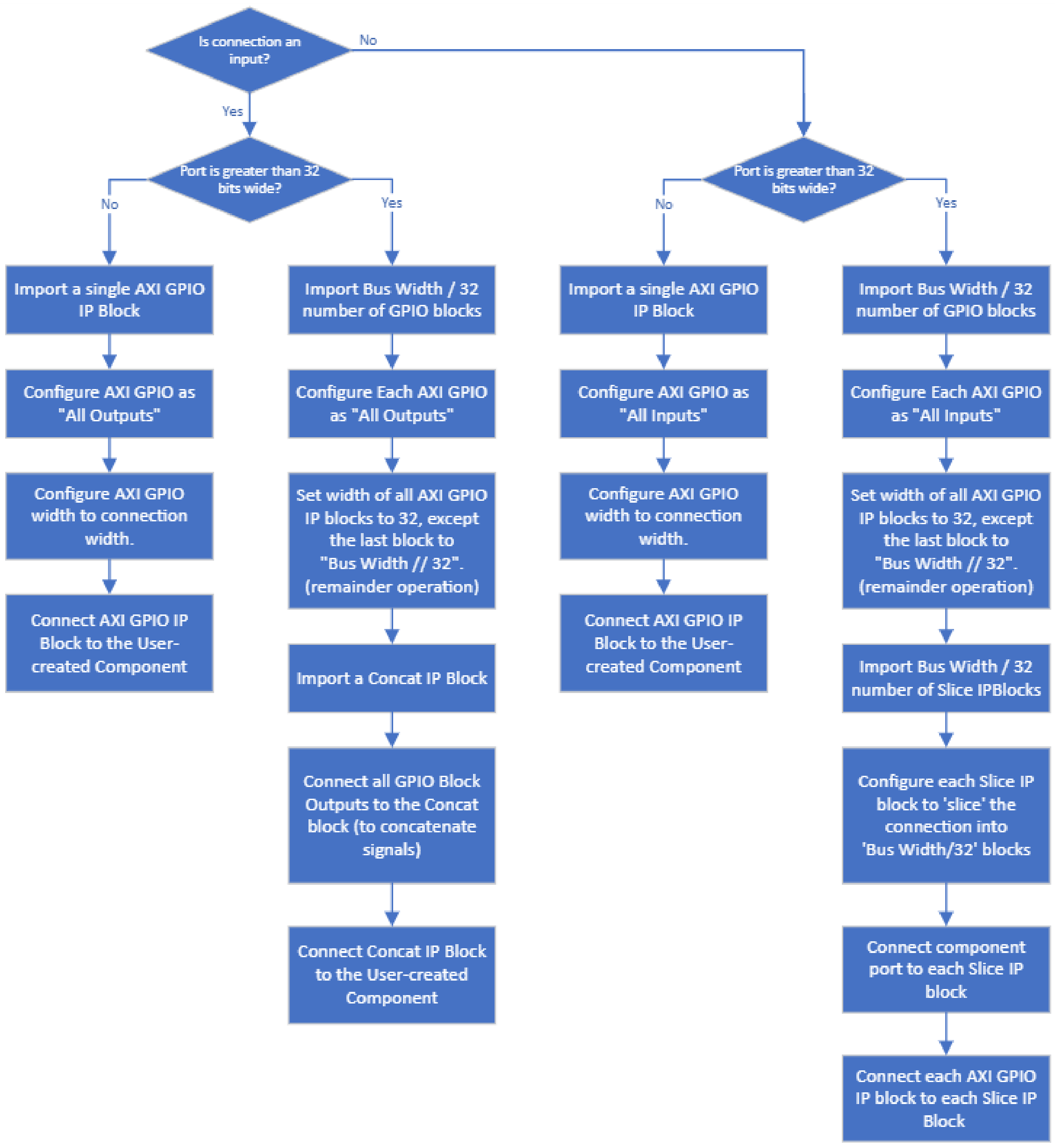
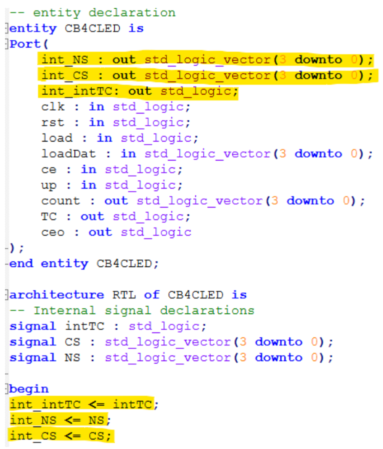

Is connection an input?
No
Yes
Port is greater than 32 bits wide?
No
Yes
Port is greater than 32 bits wide?
No
Yes
Import a single AXI GPIO IP Block
Configure AXI GPIO as "All Outputs"
Configure AXI GPIO width to connection width.
Connect AXI GPIO IP Block to the User-created Component
Import Bus Width / 32 number of GPIO blocks
Configure Each AXI GPIO as "All Outputs"
Set width of all AXI GPIO IP blocks to 32, except the last block to "Bus Width // 32". (remainder operation)
Import a Concat IP Block
Connect all GPIO Block Outputs to the Concat block (to concatenate signals)
Connect Concat IP Block to the User-created Component
Import a single AXI GPIO IP Block
Configure AXI GPIO as "All Inputs"
Configure AXI GPIO width to connection width.
Connect AXI GPIO IP Block to the User-created Component
Import Bus Width / 32 number of GPIO blocks
Configure Each AXI GPIO as "All Inputs"
Set width of all AXI GPIO IP blocks to 32, except the last block to "Bus Width // 32". (remainder operation)
Import Bus Width / 32 number of Slice IPBlocks
Configure each Slice IP block to 'slice' the connection into 'Bus Width/32' blocks
Connect component port to each Slice IP block
Connect each AXI GPIO IP block to each Slice IP Block

Figure 21 Elaborated Block design flowchart describing how HDL model AXI GPIO connections are generated

3.7.3 Execute Tcl in Vivado

Before executing in Vivado, the target HDL model may need to be backed up and modified if internal signals are configured to be exposed as external signals. This injection occurs immediately before Vivado opens, and restored immediately when Vivado exits to ensure the model is modified for the shortest period. This reduces the chance of the model being corrupted. Figure 22 presents an example of internal signal configuration from a VHDL perspective for a simple 4-bit counter.



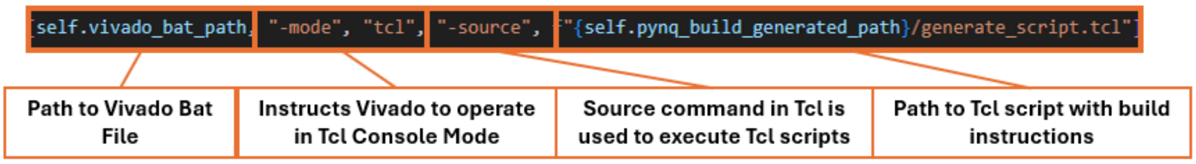
```
-- entity declaration
entity CB4CLED is
Port(
    int_NS : out std_logic_vector(3 downto 0);
    int_CS : out std_logic_vector(3 downto 0);
    int_intTC: out std_logic;
    clk : in std_logic;
    rst : in std_logic;
    load : in std_logic;
    loadDat : in std_logic_vector(3 downto 0);
    ce : in std_logic;
    up : in std_logic;
    count : out std_logic_vector(3 downto 0);
    TC : out std_logic;
    ceo : out std_logic
);
end entity CB4CLED;

architecture RTL of CB4CLED is
-- Internal signal declarations
signal intTC : std_logic;
signal CS : std_logic_vector(3 downto 0);
signal NS : std_logic_vector(3 downto 0);

begin
int_intTC <= intTC;
int_NS <= NS;
int_CS <= CS;
```

Figure 22 Example HDL code injection to make three internal signals available as external signals with a 4-bit counter design

To execute the Tcl script generated in the previous section, the *subprocess* Python library is used. The *subprocess* module enables new processes to be initiated and their return codes recorded. Figure 23 presents how the command-line instruction is formulated to initiate the build process in Vivado:



```
[self.vivado_bat_path, "-mode", "tcl", "-source", f"{self.pynq_build_generated_path}/generate_script.tcl"]
```

Path to Vivado Bat File
Instructs Vivado to operate in Tcl Console Mode
Source command in Tcl is used to execute Tcl scripts
Path to Tcl script with build instructions

Figure 23 Image presenting shell command used to trigger a new Vivado process

Data returned by Vivado in the shell instances is used to determine the status and progress of the build and is reflected under the build status tab and under the builder log tab. Figure 24 presents the block design generated for a RISC-V ALU where the four last bits of the output (32-bit bus) are connected to LEDs (and hence are connected as external ports).

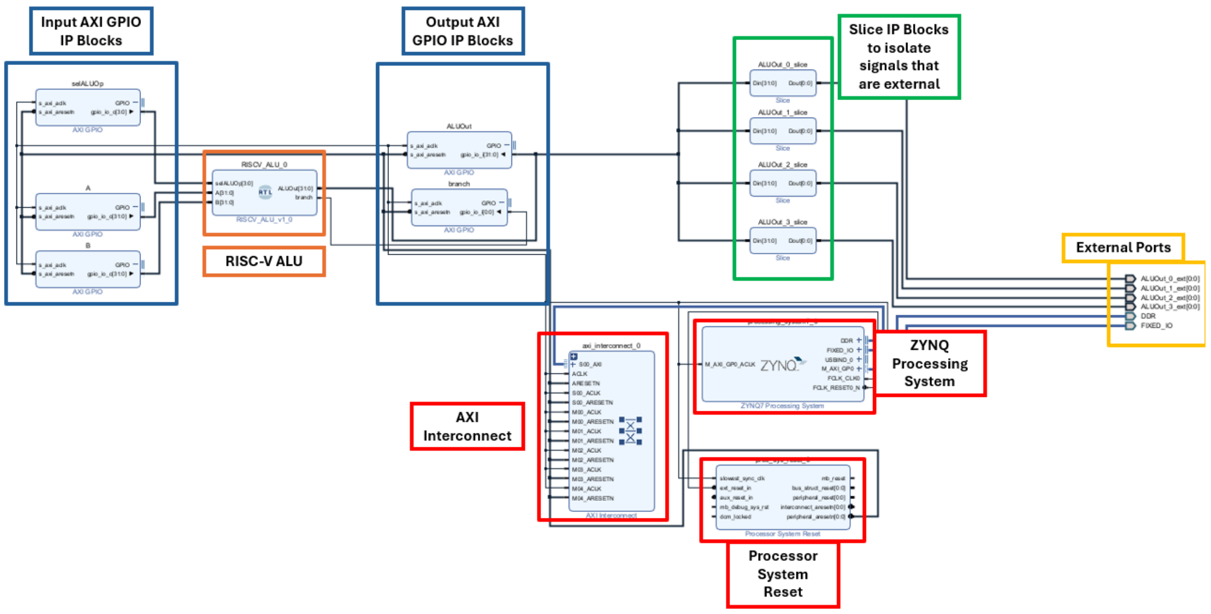
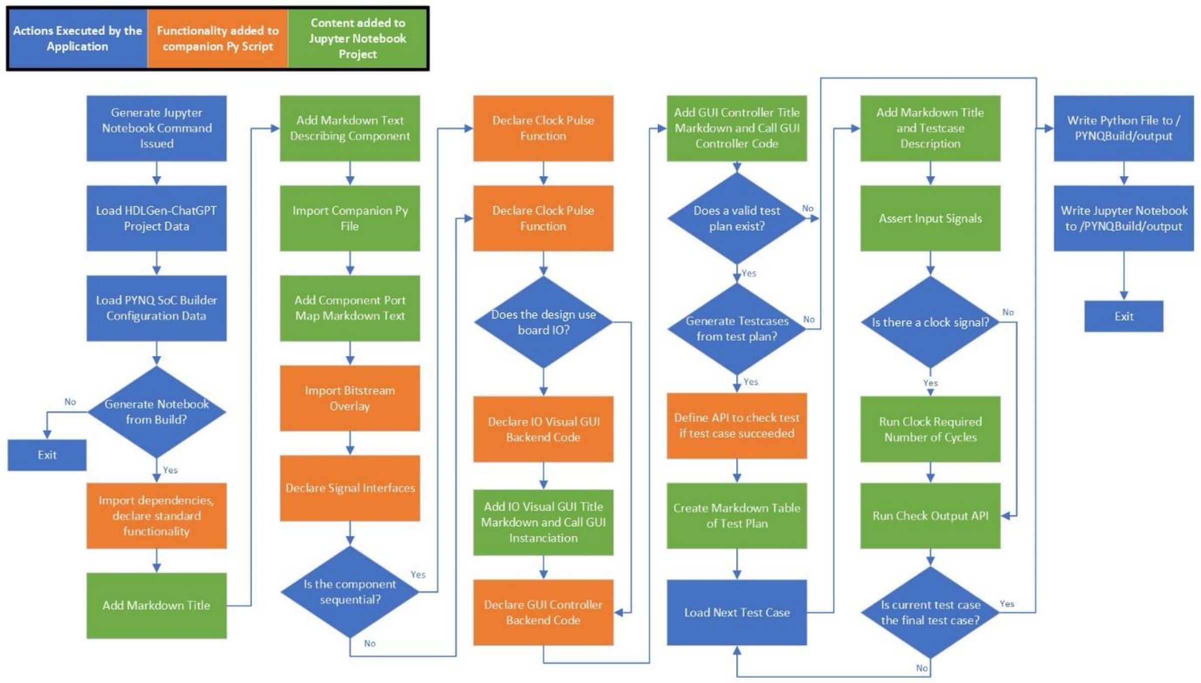


Figure 24 Block Diagram produced for a RISC-V ALU with 4 external ports

3.7.4 Generate Jupyter Notebook Application

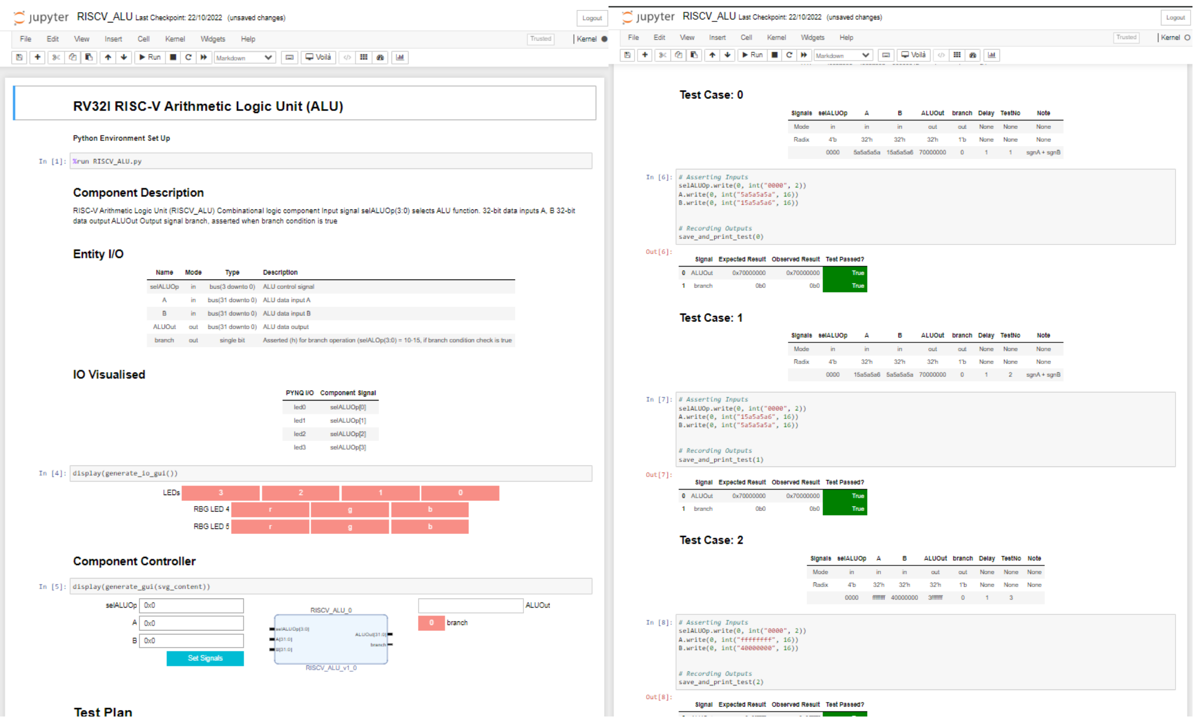
A fundamental component of the PYNQ SoC Builder's functionality is the ability to automatically produce Jupyter Notebook python-based driver applications. Figure 25 presents a flowchart which visualizes the behaviour of the "Generate Notebook" functionality.



Actions Executed by the Application
Functionality added to companion Py Script
Content added to Jupyter Notebook Project
Generate Jupyter Notebook Command Issued
Load HDLGen-ChatGPT Project Data
Load PYNQ SoC Builder Configuration Data
Generate Notebook from Build?
No
Exit
Yes
Import dependencies, declare standard functionality
Add Markdown Title
Add Markdown Text Describing Component
Import Companion Py File
Add Component Port Map Markdown Text
Import Bitstream Overlay
Declare Signal Interfaces
Is the component sequential?
Yes
No
Declare Clock Pulse Function
Declare Clock Pulse Function
Does the design use board IO?
Declare IO Visual GUI Backend Code
Add IO Visual GUI Title Markdown and Call GUI Instanciation
Declare GUI Controller Backend Code
Add GUI Controller Title Markdown and Call GUI Controller Code
Does a valid test plan exist?
No
Yes
Generate Testcases from test plan?
No
Yes
Define API to check test if test case succeeded
Create Markdown Table of Test Plan
Load Next Test Case
Add Markdown Title and Testcase Description
Assert Input Signals
Is there a clock signal?
No
Yes
Run Clock Required Number of Cycles
Run Check Output API
Is current test case the final test case?
Yes
No
Write Python File to /PYNQBuild/output
Write Jupyter Notebook to /PYNQBuild/output
Exit

Figure 25 Flowchart describing the behaviour of the 'Generate Notebook' functionality

Figure 26 presents an example Jupyter Notebook driver application generated for a RISC-V ALU design.


jupyter RISCV_ALU Last Checkpoint: 22/10/2022 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Kernel
Run
Markdown
Voilà

RV32I RISC-V Arithmetic Logic Unit (ALU)

Python Environment Set Up

```
In [1]: %run RISCV_ALU.py
```

Component Description

RISC-V Arithmetic Logic Unit (RISCV_ALU) Combinational logic component Input signal selALUOp(3:0) selects ALU function. 32-bit data inputs A, B 32-bit data output ALUOut Output signal branch, asserted when branch condition is true

Entity I/O

Name	Mode	Type	Description
selALUOp	in	bus(3 downto 0)	ALU control signal
A	in	bus(31 downto 0)	ALU data input A
B	in	bus(31 downto 0)	ALU data input B
ALUOut	out	bus(31 downto 0)	ALU data output
branch	out	single bit	Asserted (h) for branch operation (selALOp(3:0) = 10-15, if branch condition check is true

IO Visualised

PYNQ I/O	Component Signal
led0	selALUOp[0]
led1	selALUOp[1]
led2	selALUOp[2]
led3	selALUOp[3]

```
In [4]: display(generate_io_gui())
```

Component Controller

```
In [5]: display(generate_gui(svg_content))
```

Test Plan

Test Case: 0

Signals	selALUOp	A	B	ALUOut	branch	Delay	TestNo	Note
Mode	in	in	in	out	out	None	None	None
Radix	4'b	32'h	32'h	32'h	1'b	None	None	None
	0000	5a5a5a5a	15a5a5a6	70000000	0	1	1	sgnA + sgnB

```
In [6]: # Asserting Inputs
        selALUOp.write(0, int("0000", 2))
        A.write(0, int("5a5a5a5a", 16))
        B.write(0, int("15a5a5a6", 16))


        # Recording Outputs
        save_and_print_test(0)
```

Out[6]:

	Signal	Expected Result	Observed Result	Test Passed?
0	ALUOut	0x70000000	0x70000000	True
1	branch	0b0	0b0	True

Test Case: 1

Signals	selALUOp	A	B	ALUOut	branch	Delay	TestNo	Note
Mode	in	in	in	out	out	None	None	None
Radix	4'b	32'h	32'h	32'h	1'b	None	None	None
	0000	15a5a5a6	5a5a5a5a	70000000	0	1	2	sgnA + sgnB

```
In [7]: # Asserting Inputs
        selALUOp.write(0, int("0000", 2))
        A.write(0, int("15a5a5a6", 16))
        B.write(0, int("5a5a5a5a", 16))


        # Recording Outputs
        save_and_print_test(1)
```

Out[7]:

	Signal	Expected Result	Observed Result	Test Passed?
0	ALUOut	0x70000000	0x70000000	True
1	branch	0b0	0b0	True

Test Case: 2

Signals	selALUOp	A	B	ALUOut	branch	Delay	TestNo	Note
Mode	in	in	in	out	out	None	None	None
Radix	4'b	32'h	32'h	32'h	1'b	None	None	None
	0000	ffffffff	40000000	3fffffff	0	1	3	

```
In [8]: # Asserting Inputs
        selALUOp.write(0, int("0000", 2))
        A.write(0, int("ffffffff", 16))
        B.write(0, int("40000000", 16))


        # Recording Outputs
        save_and_print_test(2)
```

Out[8]:

	Signal	Expected Result	Observed Result	Test Passed?

Figure 26 Example Jupyter Notebook generated for RISC-V ALU complete with IO configuration and Test case execution

3.7.5 Copy Output Files

The final step in the build process is to copy all required files produced to an easily accessible output folder in the project's file structure. The contents of the output folder and its structure are detailed in the next section.

3.8 PYNQ SoC Builder Data

The PYNQ SoC Builder produces a collection of directories and files required for preserving user configurations, building a design, and storing the output of the full build process. Figure 27 presents the file structure created by the SoC Builder in a project's directory. The PYNQ SoC Builder creates a new folder at the project's root directory, PYNQBuild, which stores all related data.

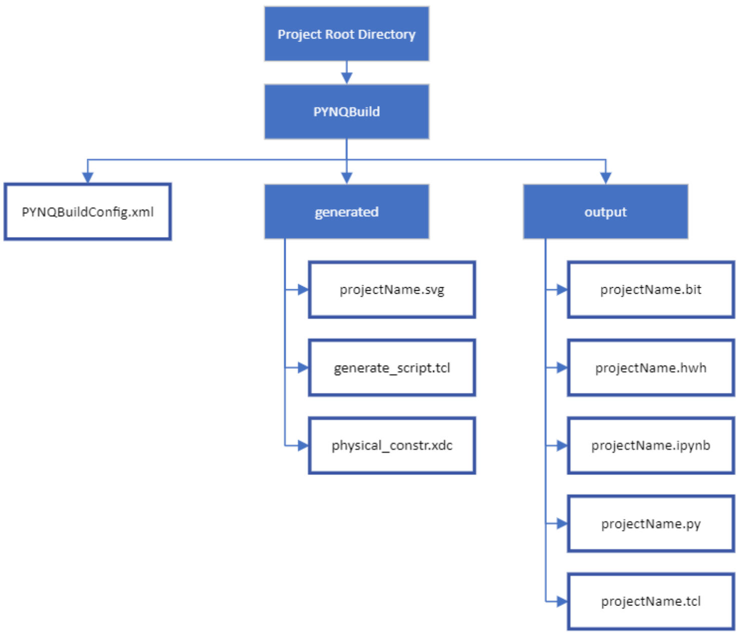


Figure 27 Structure of data produced by PYNQ SoC Builder

The *PYNQBuildConfig.xml* file is an XML-encoded document which stores the project build configuration as specified by the user using the SoC Builder GUI. Figure 28 presents the structure of this XML file. The structure consists of four main categories: settings, ioConfig, internalSignals and flags.

- **settings:** The settings tag stores the Boolean value of each toggle (on/off) switch in the SoC Builder's GUI. Each entry consists of the variable name, and Boolean value.
- **ioConfig:** The *ioConfig* tag defines user-specified connections between components and PYNQ board LEDS. Each connection is comprised of three values: *io, signals* and *pin*. The *io* variable assigns a specific I/O for the connection. Conversely, the *signal* variable selects the corresponding port on the component. The *pin* variable selects the exact pin of the signal to connect to the board I/O.
- **internalSignals:** This tag defines which internal signals should be extended to the top-level of the component. This tag is comprised of two crucial variables for the build process: *name* and *width*. The name variable identifies the internal signal, and the *width* variable defines its bit-width.
- **flags:** The *flags* tag stores Boolean variables indicating the current state of the PYNQ SoC Builder. Currently, the only implemented flag is *hdl_modified.* This flag signifies if the HDL model has been modified by the SoC Builder. This flag is used to ensure that HDL models are not corrupt and can be restored if the SoC Builder quits unexpectedly.

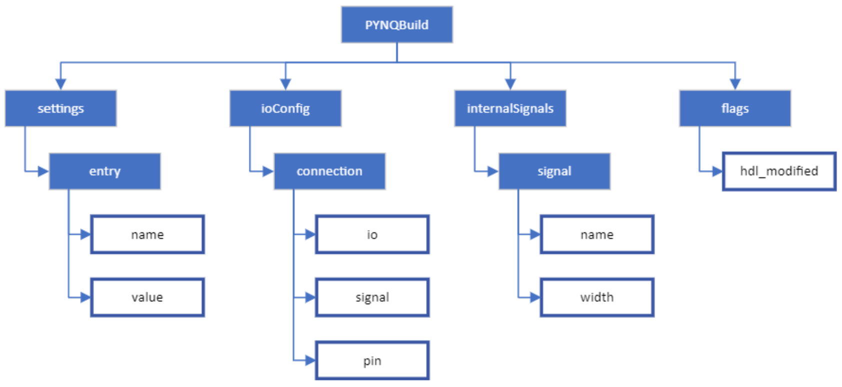


Figure 28 XML structure of PYNQBuildConfig.xml

The *generated* folder contains all temporary files used by the PYNQ SoC Builder to complete the build process. The *generate_script.tcl* is generated by the Tcl Generator step in the build process. Vivado runs this script to generate an FPGA overlay. The *physical_constr.xdc* file is a physical constraints file used by Vivado to define connections between the PYNQ board's I/O and the component. It is generated in parallel with *generate_script.tcl*. The *projectName.svg* file is a scalable vector graphics file generated by Vivado. It is a visual representation of the component which is later presented to the user in the Jupyter Notebook driver application.

Finally, the *output* folder contains all files which are required by the PYNQ hardware to operate. There are five files in total:

- **projectName.bit:** is a bitstream file containing a description of hardware logic, routing, initial values for registers and on-chip memory. It is used to configure the FPGA.
- **projectName.hwh:** is a hardware handoff file. It is automatically generated by Vivado when generating a bitstream. It contains a description of the Zynq system configuration and hardware intellectual property (IP) used in the block design.
- **projectName.tcl:** is a Tcl script representation of the block design. This Tcl file offers the same functionality as the hardware handoff. It is generated to serve the same purpose.
- **projectName.ipynb:** is a Jupyter Notebook file. This driver application is the main interface between the user and component. It contains markdown descriptions, a GUI-based component controller, and test plan execution code (if applicable).
- **projectName.py:** is a companion Python script to the Jupyter Notebook driver application. This file contains lengthy code blocks that are not relevant to the end-user, such as backend logic of the GUI-controller. The driver application calls functions from this script, reducing the interface complexity and offering a more streamlined experience to users.

4. PYNQ FPGA Set Up

4.1 Overview

This chapter will outline the requirements and steps taken to replicate the configurations used throughout this project. The PYNQ Z2 boards can be configured for both local and remote development depending on the specific requirements of the end-user. The goal of each environment is to make the PYNQ board's Jupyter Notebook environment accessible over local network and/or the internet to the user's PC.

4.2 Local Environment

This section will outline how a PYNQ Z2 board must be configured locally to facilitate a remote connection. The two ways in which a PYNQ Z2 can be configured locally are:

1. Direct Connection to the User's PC
2. Local Area Network (LAN) Connection

Whilst a direct connection to a user's PC offers simplicity, it restricts the PYNQ board's access to the internet. Conversely, a LAN connection to the PYNQ board enables both communication with the host computer and internet access for the PYNQ, making it the preferred choice for this project, where internet connectivity is crucial. Figure 29 presents the architecture of such a configuration.

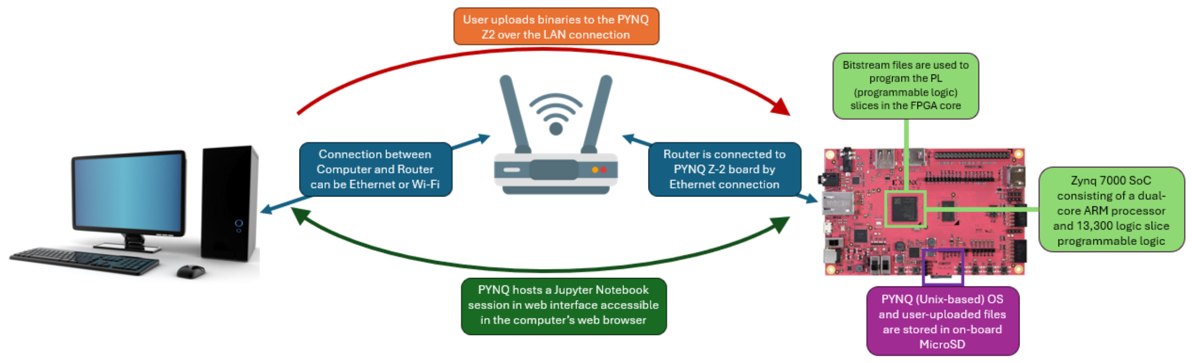
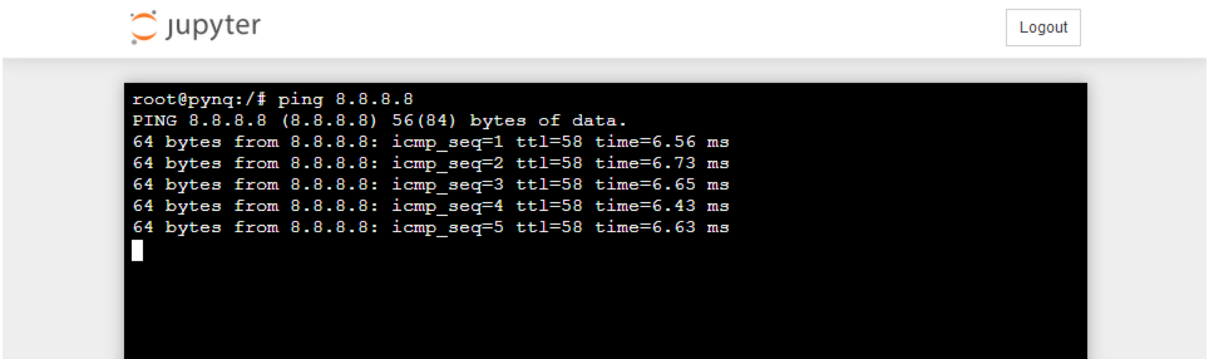

User uploads binaries to the PYNQ Z2 over the LAN connection
Bitstream files are used to program the PL (programmable logic) slices in the FPGA core
Connection between Computer and Router can be Ethernet or Wi-Fi
Router is connected to PYNQ Z-2 board by Ethernet connection
Zynq 7000 SoC consisting of a dual-core ARM processor and 13,300 logic slice programmable logic
PYNQ hosts a Jupyter Notebook session in web interface accessible in the computer's web browser
PYNQ (Unix-based) OS and user-uploaded files are stored in on-board MicroSD

Figure 29 Architecture of LAN connection

4.3 Enabling Remote Connection

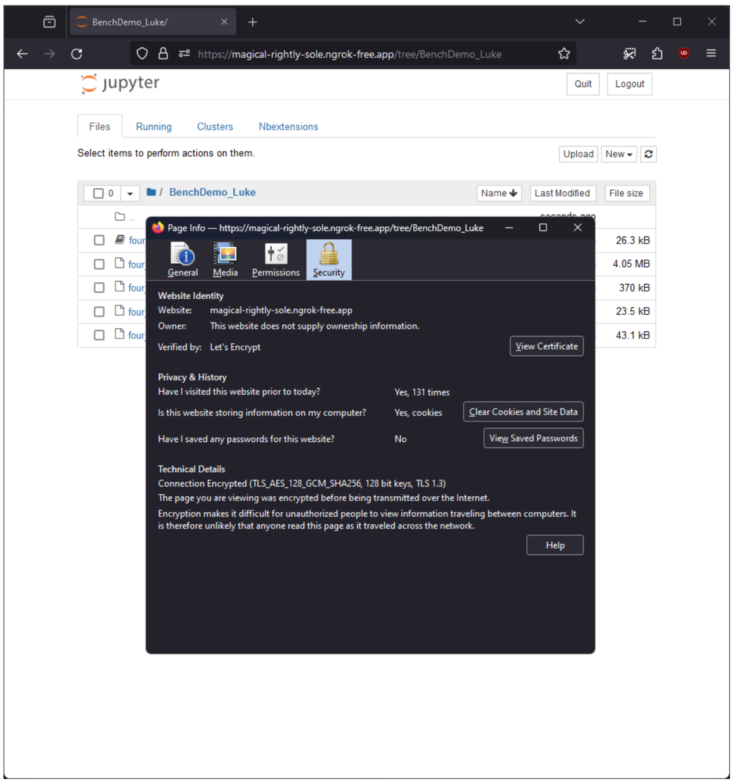
This section details the implementation of remote access to a PYNQ Z2 board over the internet. To achieve this, the local configuration detailed in section 4.2 must first be implemented. To verify the PYNQ board has an internet connection, the ping command may be used to reach Google DNS servers as in figure 30.



```
root@pynq:/# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=58 time=6.56 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=58 time=6.73 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=58 time=6.65 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=58 time=6.43 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=58 time=6.63 ms
```

Figure 30 PYNQ board pinging Google DNS servers from terminal

Ngrok, a unified ingress platform, is utilised to facilitate remote connection to the FPGA. Ngrok functions by reverse-proxying the Jupyter Notebook webserver host on the PYNQ board. By default, the Jupyter Notebook webserver does not provide any encryption or security features, exposing network traffic to man-in-the-middle attacks. Conversely, Ngrok secures all data using a TLS-encrypted connection between the host computer and PYNQ board, ensuring confidentiality and integrity of all data. Figure 31 presents the security information for a remote PYNQ board accessed via Ngrok.



Select items to perform actions on them.

/ BenchDemo_Luke

..	seconds ago
four	26.3 kB
four	4.05 MB
four	370 kB
four	23.5 kB
four	43.1 kB

Page Info — https://magical-rightly-sole.ngrok-free.app/tree/BenchDemo_Luke

Website Identity
Website: magical-rightly-sole.ngrok-free.app
Owner: This website does not supply ownership information.
Verified by: Let's Encrypt

Privacy & History
Have I visited this website prior to today? Yes, 131 times
Is this website storing information on my computer? Yes, cookies
Have I saved any passwords for this website? No

Technical Details
Connection Encrypted (TLS_AES_128_GCM_SHA256, 128 bit keys, TLS 1.3)
The page you are viewing was encrypted before being transmitted over the Internet.
Encryption makes it difficult for unauthorized people to view information traveling between computers. It is therefore unlikely that anyone read this page as it traveled across the network.

Figure 31 Security information when visiting PYNQ Jupyter Notebook server through Ngrok.

Figure 32 illustrates the architecture of a remote connection using Ngrok between a host computer and PYNQ Z2 board.

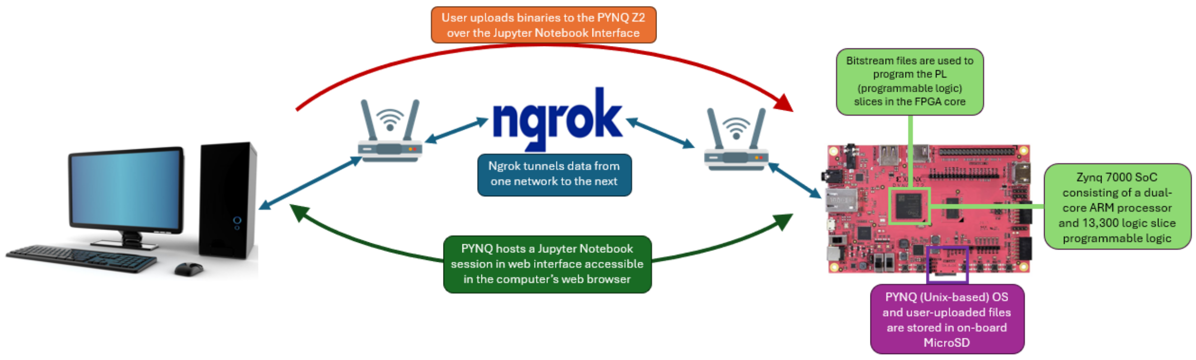

User uploads binaries to the PYNQ Z2 over the Jupyter Notebook Interface
ngrok
Ngrok tunnels data from one network to the next
Bitstream files are used to program the PL (programmable logic) slices in the FPGA core
Zynq 7000 SoC consisting of a dual-core ARM processor and 13,300 logic slice programmable logic
PYNQ hosts a Jupyter Notebook session in web interface accessible in the computer's web browser
PYNQ (Unix-based) OS and user-uploaded files are stored in on-board MicroSD

Figure 32 Architecture of remote connection using Ngrok

To encourage users to implement their own remotely available FPGA boards, online resources in the form of a written guide and video tutorial are available in table 3.

Description	**Link**
Step 1: Configuring your Ngrok account	GitHub PDF
Step 2: Installing Ngrok on PYNQ	Video (YouTube)

Table 3 Set up guide for enabling Ngrok-based remote environment

5. Design Testing

5.1 Overview

This section introduces five projects developed and built using HDLGen-ChatGPT and PYNQ SoC Builder projects. For each project, table 4 provides links to video tutorials and project source files. Each video offers a step-by-step guide to users whilst also emphasizing the SoC Builder's functionality and features.

In this section, a range of designs are tested and documented to both test the project's capabilities and functionality and to act as tutorials for future users or collaborators on this project.

In this chapter, five projects demonstrate the functionality and robustness of the PYNQ SoC Builder. These projects are:

- 2-to-1 bit Multiplexer
- RISC-V Arithmetic Logic Unit (ALU)
- 4-bit Counter
- Single Shot
- RISC-V Register Bank (RB)

Each test project will be developed with VHDL, except for the 2-to-1 bit multiplexer, which is developed using Verilog.

This chapter is structured such that each sample project is accompanied by a video walkthrough tutorial. The documentation in this chapter will provide an overview of each project, and the output of the PYNQ SoC Builder.

Table 4 provides links to both the video and source files. Source files may be used to replicate actions completed within the videos. Each video consists of the following sections:

- Project Introduction and Overview
- Design Capture in HDLGen-ChatGPT
- Building using PYNQ SoC Builder
- Running Designs on PYNQ Hardware

Project	**Content demonstrates:**	**Resources**
2-to-1 Multiplexer	Basic SoC Builder Operation and remote environment use, utilise Verilog	Part 1: YouTube Part 2: YouTube GitHub
RISC-V Arithmetic Logic Unit (ALU)	Advanced test plan and board I/O	YouTube GitHub
4-bit Counter	Reading internal signals	YouTube GitHub
Single Shot	Clock Functionality	YouTube GitHub

Table 4 List of projects and hyperlinks to online resources

5.2 Multiplexer

5.2.1 Overview

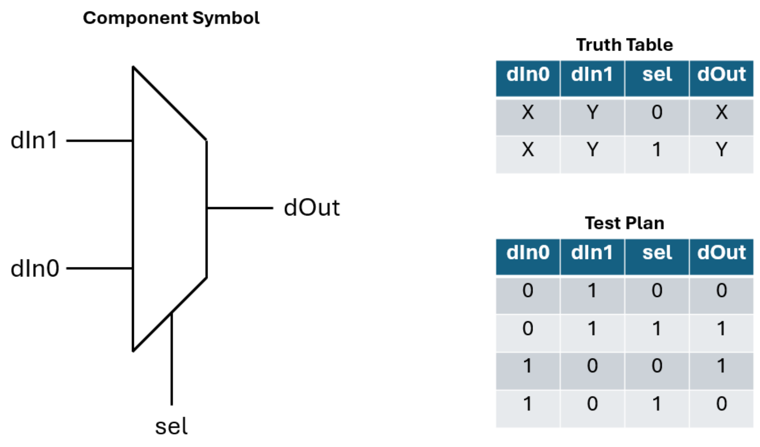
This subsection presents a detailed analysis of a 2-to-1 bit multiplexer within the context of the PYNQ SoC Builder. It covers the multiplexer design itself, the configuration process using the PYNQ SoC Builder, and the obtained results. The multiplexer design is used to demonstrate the most basic functionality of the PYNQ SoC Builder. This project uses Verilog HDL. Links to access video material are provided in table 5.

Description	**Link**
Part 1: Capturing Mux using HDLGen-ChatGPT	YouTube
Part 2: Deploying Mux using PYNQ SoC Builder	YouTube
Source Files	GitHub

Table 5 Link to multiplexer design online resources

5.2.2 Design

Figure 33 presents the component symbol, truth table and test plan of the 2-to-1 mux. There are three single bit inputs to the multiplexer, *dIn1, dIn0*, and *sel*, and one output, *dOut*. The design is captured using the Verilog HDL language in HDLGen-ChatGPT.



Truth Table

dIn0	dIn1	sel	dOut
X	Y	0	X
X	Y	1	Y

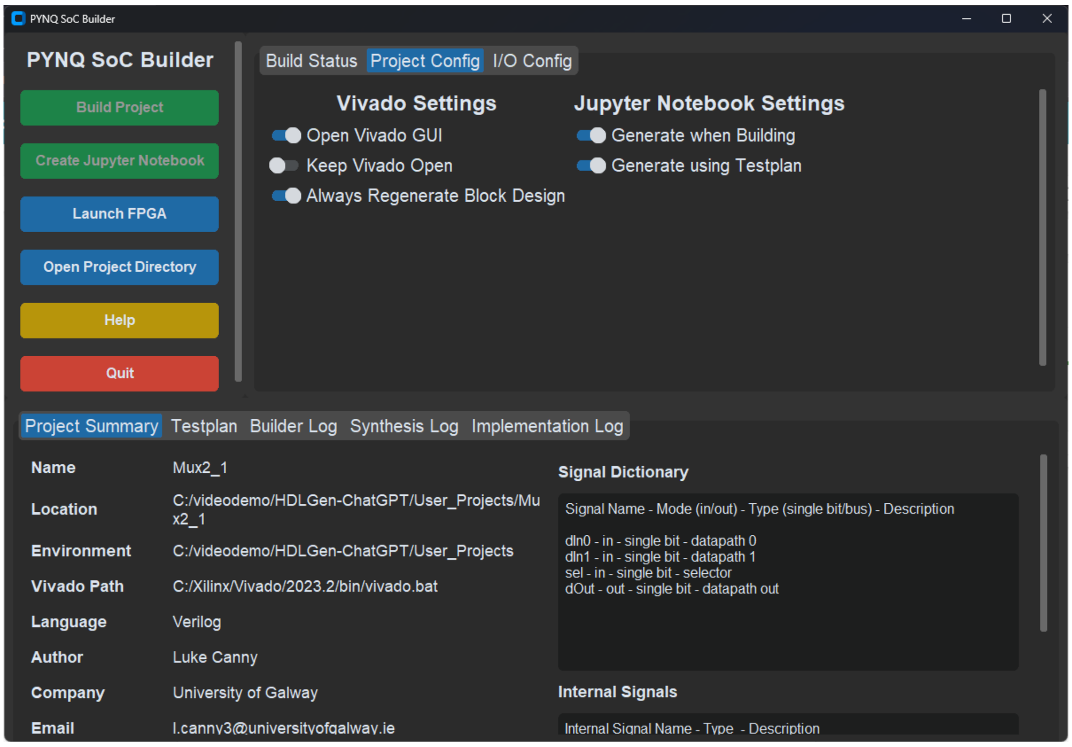
Test Plan

dIn0	dIn1	sel	dOut
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0

Figure 33 Mux2_1 component symbol, truth table and test plan

5.2.3 SoC Builder Configuration

Figure 34 and 35 presents the PYNQ SoC Builder configuration used to build the multiplexer design. Figure 34 presents the project configuration settings used. The default project configuration is used. Figure 35 presents both the IO configuration used, but also the test plan used. The test plan matches the test plan specified in figure 33 previously. There is no I/O or internal signal configuration.



Vivado Settings
- Open Vivado GUI: on
- Keep Vivado Open: off
- Always Regenerate Block Design: on

Jupyter Notebook Settings
- Generate when Building: on
- Generate using Testplan: on

Name	Mux2_1
Location	C:/videodemo/HDLGen-ChatGPT/User_Projects/Mux2_1
Environment	C:/videodemo/HDLGen-ChatGPT/User_Projects
Vivado Path	C:/Xilinx/Vivado/2023.2/bin/vivado.bat
Language	Verilog
Author	Luke Canny
Company	University of Galway
Email	l.canny3@universityofgalway.ie

Signal Dictionary

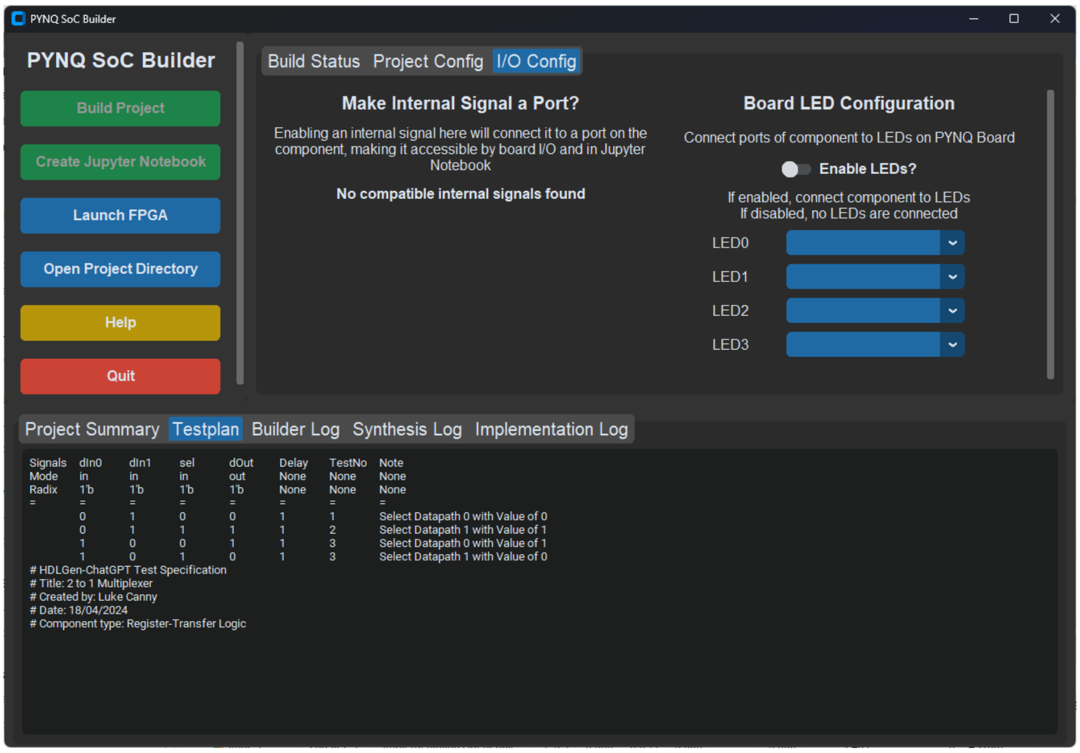
```
Signal Name - Mode (in/out) - Type (single bit/bus) - Description

dIn0 - in - single bit - datapath 0
dIn1 - in - single bit - datapath 1
sel - in - single bit - selector
dOut - out - single bit - datapath out
```

Internal Signals

```
Internal Signal Name - Type  - Description
```

Figure 34 Project Config and Project Summary in PYNQ SoC Builder for 2-to-1 Mux design



Make Internal Signal a Port?

Enabling an internal signal here will connect it to a port on the component, making it accessible by board I/O and in Jupyter Notebook

No compatible internal signals found

Board LED Configuration

Connect ports of component to LEDs on PYNQ Board

Enable LEDs? off

If enabled, connect component to LEDs
If disabled, no LEDs are connected

```
Signals  dIn0   dIn1   sel    dOut   Delay  TestNo  Note
Mode     in     in     in     out    None   None    None
Radix    1'b    1'b    1'b    1'b    None   None    None
=        =      =      =      =      =      =       =
         0      1      0      0      1      1       Select Datapath 0 with Value of 0
         0      1      1      1      1      2       Select Datapath 1 with Value of 1
         1      0      0      1      1      3       Select Datapath 0 with Value of 1
         1      0      1      0      1      3       Select Datapath 1 with Value of 0
# HDLGen-ChatGPT Test Specification
# Title: 2 to 1 Multiplexer
# Created by: Luke Canny
# Date: 18/04/2024
# Component type: Register-Transfer Logic
```

Figure 35 IO Config and Test Plan in PYNQ SoC Builder for 2-to-1 Mux design

5.2.4 Results

Figure 36 presents the block design generated by the PYNQ SoC Builder in Vivado. Figure 37 presents the Jupyter Notebook application generated by the PYNQ SoC Builder, running on remote hardware (accessed using Ngrok).

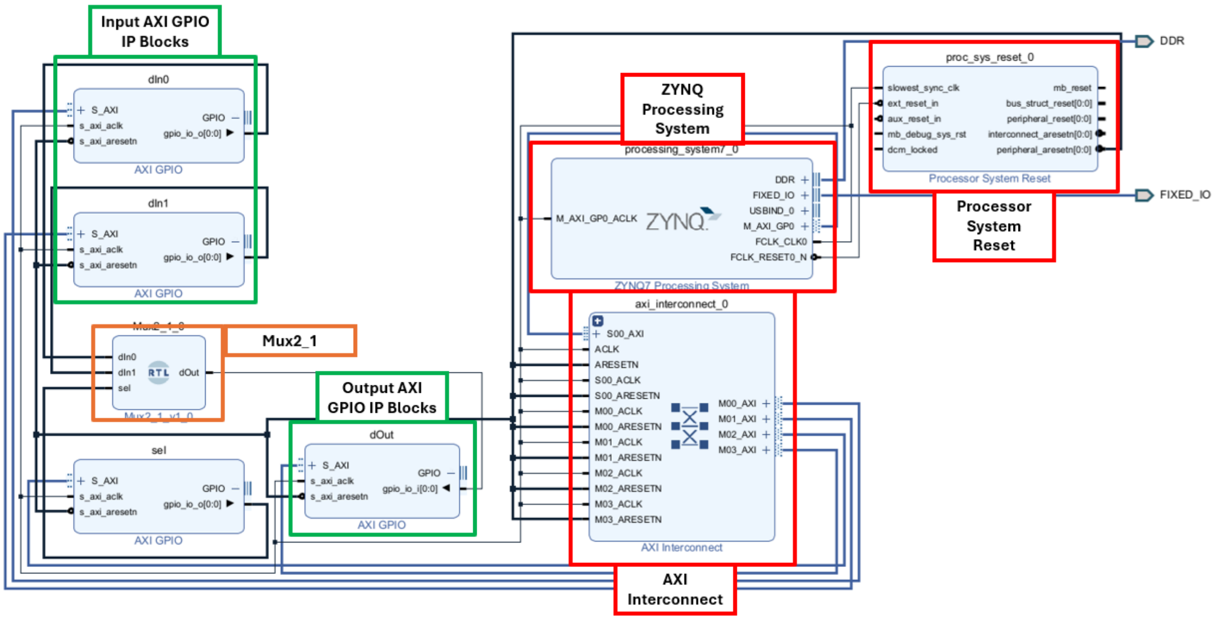
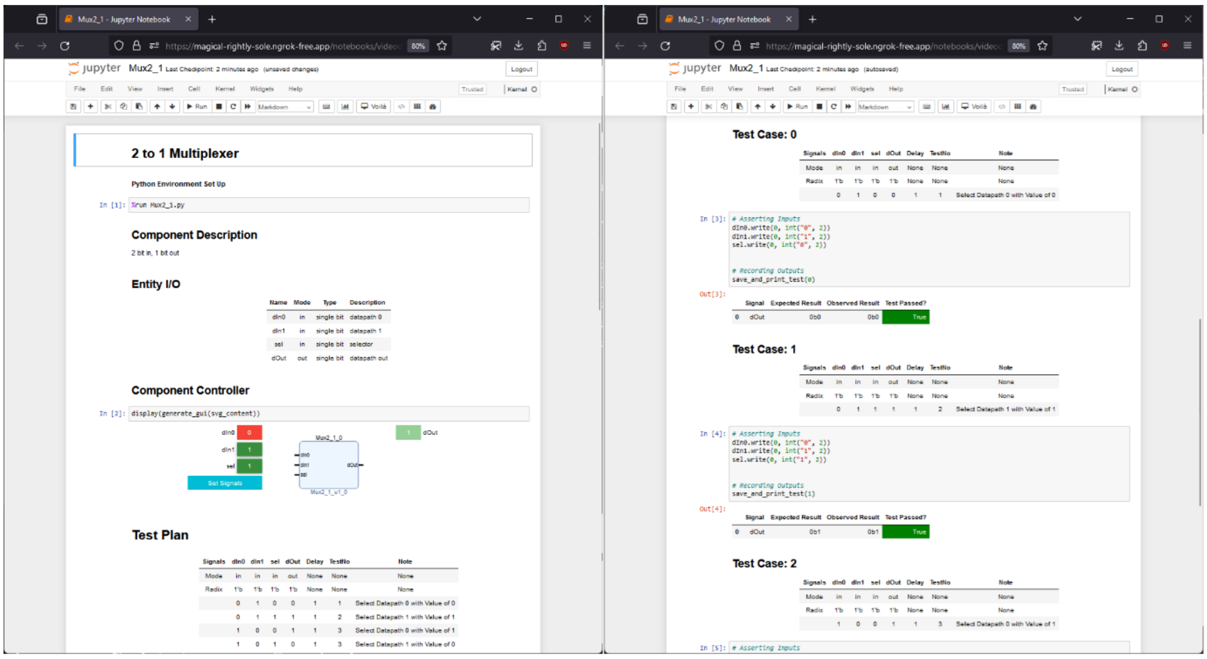


Figure 36 Generated block design for Mux2_1 design



Mux2_1 - Jupyter Notebook
https://magical-rightly-sole.ngrok-free.app/notebooks/videod
80%
jupyter
Mux2_1 Last Checkpoint: 2 minutes ago (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Kernel
Run
Markdown
Voilà

2 to 1 Multiplexer

Python Environment Set Up

```
In [1]: %run Mux2_1.py
```

Component Description

2 bit in, 1 bit out

Entity I/O

Name	Mode	Type	Description
dIn0	in	single bit	datapath 0
dIn1	in	single bit	datapath 1
sel	in	single bit	selector
dOut	out	single bit	datapath out

Component Controller

```
In [2]: display(generate_gui(svg_content))
```

Test Plan

Signals	dIn0	dIn1	sel	dOut	Delay	TestNo	Note
Mode	in	in	in	out	None	None	None
Radix	1'b	1'b	1'b	1'b	None	None	None
	0	1	0	0	1	1	Select Datapath 0 with Value of 0
	0	1	1	1	1	2	Select Datapath 1 with Value of 1
	1	0	0	1	1	3	Select Datapath 0 with Value of 1
	1	0	1	0	1	3	Select Datapath 1 with Value of 0

Mux2_1 - Jupyter Notebook
https://magical-rightly-sole.ngrok-free.app/notebooks/videod
jupyter
Mux2_1 Last Checkpoint: 2 minutes ago (autosaved)
Logout

Test Case: 0

Signals	dIn0	dIn1	sel	dOut	Delay	TestNo	Note
Mode	in	in	in	out	None	None	None
Radix	1'b	1'b	1'b	1'b	None	None	None
	0	1	0	0	1	1	Select Datapath 0 with Value of 0

```
In [3]: # Asserting Inputs
        dIn0.write(0, int("0", 2))
        dIn1.write(0, int("1", 2))
        sel.write(0, int("0", 2))


        # Recording Outputs
        save_and_print_test(0)
```

Out[3]:

	Signal	Expected Result	Observed Result	Test Passed?
0	dOut	0b0	0b0	True

Test Case: 1

Signals	dIn0	dIn1	sel	dOut	Delay	TestNo	Note
Mode	in	in	in	out	None	None	None
Radix	1'b	1'b	1'b	1'b	None	None	None
	0	1	1	1	1	2	Select Datapath 1 with Value of 1

```
In [4]: # Asserting Inputs
        dIn0.write(0, int("0", 2))
        dIn1.write(0, int("1", 2))
        sel.write(0, int("1", 2))


        # Recording Outputs
        save_and_print_test(1)
```

Out[4]:

	Signal	Expected Result	Observed Result	Test Passed?
0	dOut	0b1	0b1	True

Test Case: 2

Signals	dIn0	dIn1	sel	dOut	Delay	TestNo	Note
Mode	in	in	in	out	None	None	None
Radix	1'b	1'b	1'b	1'b	None	None	None
	1	0	0	1	1	3	Select Datapath 0 with Value of 1

```
In [5]: # Asserting Inputs
```

Figure 37 Generated Jupyter Notebook for a 2-to-1 Mux design running on remote PYNQ hardware

5.4 RISC-V Arithmetic Logic Unit

5.4.1 Overview

In this section, a RISC-V ALU is designed and build using the PYNQ SoC Builder. This design is used to demonstrate advanced combinational and the board I/O feature in the SoC Builder application. This design is implemented using VHDL. Online resources are provided in table 6.

Description	**Link**
Video Demonstration	YouTube
Source Files	GitHub

Table 6 Link to RISC-V ALU online resources

5.4.2 Design

As seen in figure 38, the ALU consists of three inputs and two outputs, as per the RISC-V specifications. The ALU is a combination component and therefore does not contain a clock signal. Figure 39 presents the test plan devised for the RISC-V ALU. It tests all 16 different operands and consists of 24 tests total.

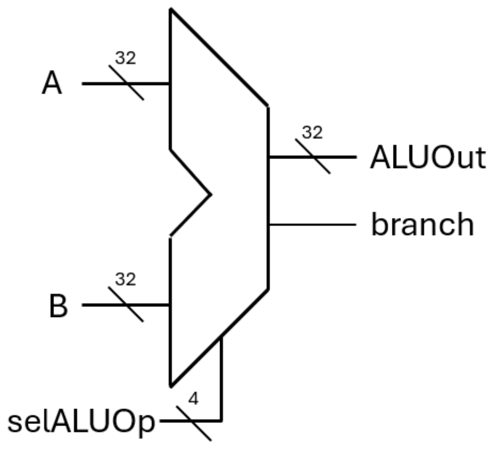


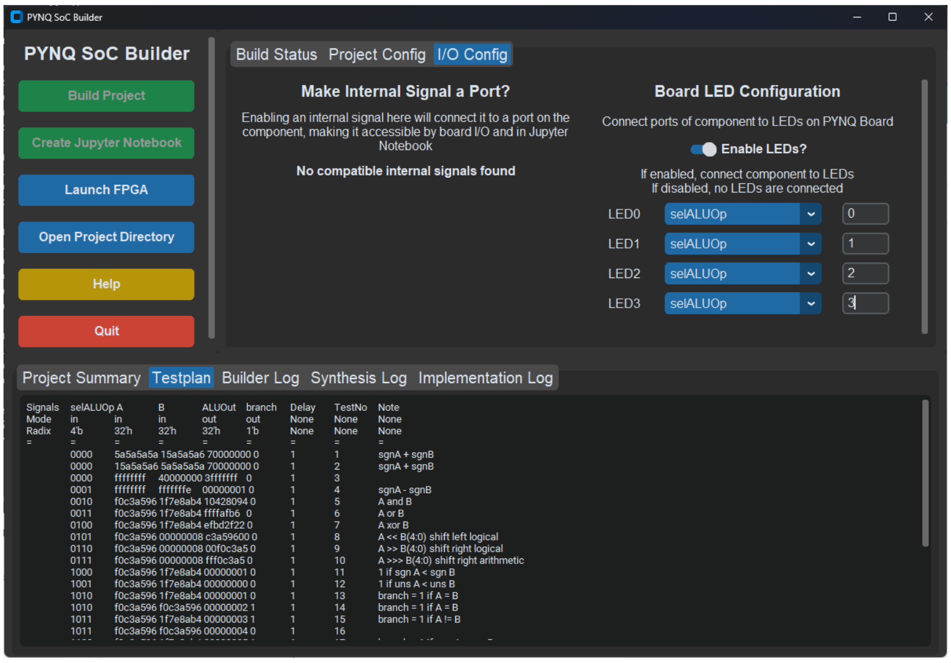
Figure 38 RISC-V ALU component symbol

Signals	selALUOp	A	B	ALUOut	branch	Delay	TestNo	Note
Mode	in	in	in	out	out	None	None	None
Radix	4'b	32'h	32'h	32'h	1'b	None	None	None
=	=	=	=	=	=	=	=	=
	0000	5a5a5a5a	15a5a5a6	70000000	0	1	1	sgnA + sgnB
	0000	15a5a5a6	5a5a5a5a	70000000	0	1	2	sgnA + sgnB
	0000	ffffffff	40000000	3fffffff	0	1	3	
	0001	ffffffff	fffffffe	00000001	0	1	4	sgnA - sgnB
	0010	f0c3a596	1f7e8ab4	10428094	0	1	5	A and B
	0011	f0c3a596	1f7e8ab4	ffffafb6	0	1	6	A or B
	0100	f0c3a596	1f7e8ab4	efbd2f22	0	1	7	A xor B
	0101	f0c3a596	00000008	c3a59600	0	1	8	A << B(4:0) shift left logical
	0110	f0c3a596	00000008	00f0c3a5	0	1	9	A >> B(4:0) shift right logical
	0111	f0c3a596	00000008	fff0c3a5	0	1	10	A >>> B(4:0) shift right arithmetic
	1000	f0c3a596	1f7e8ab4	00000001	0	1	11	1 if sgn A < sgn B
	1001	f0c3a596	1f7e8ab4	00000000	0	1	12	1 if uns A < uns B
	1010	f0c3a596	1f7e8ab4	00000001	0	1	13	branch = 1 if A = B
	1010	f0c3a596	f0c3a596	00000002	1	1	14	branch = 1 if A = B
	1011	f0c3a596	1f7e8ab4	00000003	1	1	15	branch = 1 if A != B
	1011	f0c3a596	f0c3a596	00000004	0	1	16	
	1100	f0c3a596	1f7e8ab4	00000005	1	1	17	branch = 1 if sgn A < sgnB
	1100	f0c3a596	f0c3a596	00000006	0	1	18	
	1101	f0c3a596	1f7e8ab4	00000007	0	1	19	branch = 1 if sgn A >= sgnB
	1101	f0c3a596	f0c3a596	00000008	1	1	20	
	1110	f0c3a596	1f7e8ab4	00000009	0	1	21	branch = 1 if uns A < uns B
	1110	f0c3a593	f0c3a596	00000010	1	1	22	
	1111	f0c3a596	1f7e8ab4	00000011	1	1	23	branch = 1 if uns A >= uns B
	1111	f0c3a596	f0c3a596	00000012	1	1	24	

Figure 39 Test plan of RISC-V ALU

5.4.3 SoC Builder Configuration

Figure 40 presents the IO configuration and test plan used when generating the RISC-V ALU design. The project configuration is set to the default settings. The RISC-V ALU *selALUOp* input signal is connected to each LED, 0-3.



Make Internal Signal a Port?

Enabling an internal signal here will connect it to a port on the component, making it accessible by board I/O and in Jupyter Notebook

No compatible internal signals found

Board LED Configuration

Connect ports of component to LEDs on PYNQ Board

Enable LEDs?

If enabled, connect component to LEDs
If disabled, no LEDs are connected

LED0	selALUOp	0
LED1	selALUOp	1
LED2	selALUOp	2
LED3	selALUOp	3

```
Signals  selALUOp A        B        ALUOut   branch  Delay  TestNo  Note
Mode     in       in       in       out      out     None   None    None
Radix    4'b      32'h     32'h     32'h     1'b     None   None    None
=        =        =        =        =        =       =      =       =
         0000     5a5a5a5a 15a5a5a6 70000000 0       1      1       sgnA + sgnB
         0000     15a5a5a6 5a5a5a5a 70000000 0       1      2       sgnA + sgnB
         0000     ffffffff 40000000 3fffffff 0       1      3
         0001     ffffffff fffffffe 00000001 0       1      4       sgnA - sgnB
         0010     f0c3a596 1f7e8ab4 10428094 0       1      5       A and B
         0011     f0c3a596 1f7e8ab4 ffffafb6 0       1      6       A or B
         0100     f0c3a596 1f7e8ab4 efbd2f22 0       1      7       A xor B
         0101     f0c3a596 00000008 c3a59600 0       1      8       A << B(4:0) shift left logical
         0110     f0c3a596 00000008 00f0c3a5 0       1      9       A >> B(4:0) shift right logical
         0111     f0c3a596 00000008 fff0c3a5 0       1      10      A >>> B(4:0) shift right arithmetic
         1000     f0c3a596 1f7e8ab4 00000001 0       1      11      1 if sgn A < sgn B
         1001     f0c3a596 1f7e8ab4 00000000 0       1      12      1 if uns A < uns B
         1010     f0c3a596 1f7e8ab4 00000001 0       1      13      branch = 1 if A = B
         1010     f0c3a596 f0c3a596 00000002 1       1      14      branch = 1 if A = B
         1011     f0c3a596 1f7e8ab4 00000003 1       1      15      branch = 1 if A != B
         1011     f0c3a596 f0c3a596 00000004 0       1      16
```

Figure 40 RISC-V ALU IO Configuration

5.4.4 Results

Figure 41 presents the block design generated by Vivado as instructed by the SoC Builder. As seen in the previous section, various IP blocks are imported to enable connectivity between the ZYNQ processing system and the RISC-V ALU. Additionally, the *selALUOp* port is connected to external ports. The external ports are connected to the PYNQ's LED pins by means of the master constraints file as seen in figure 42.

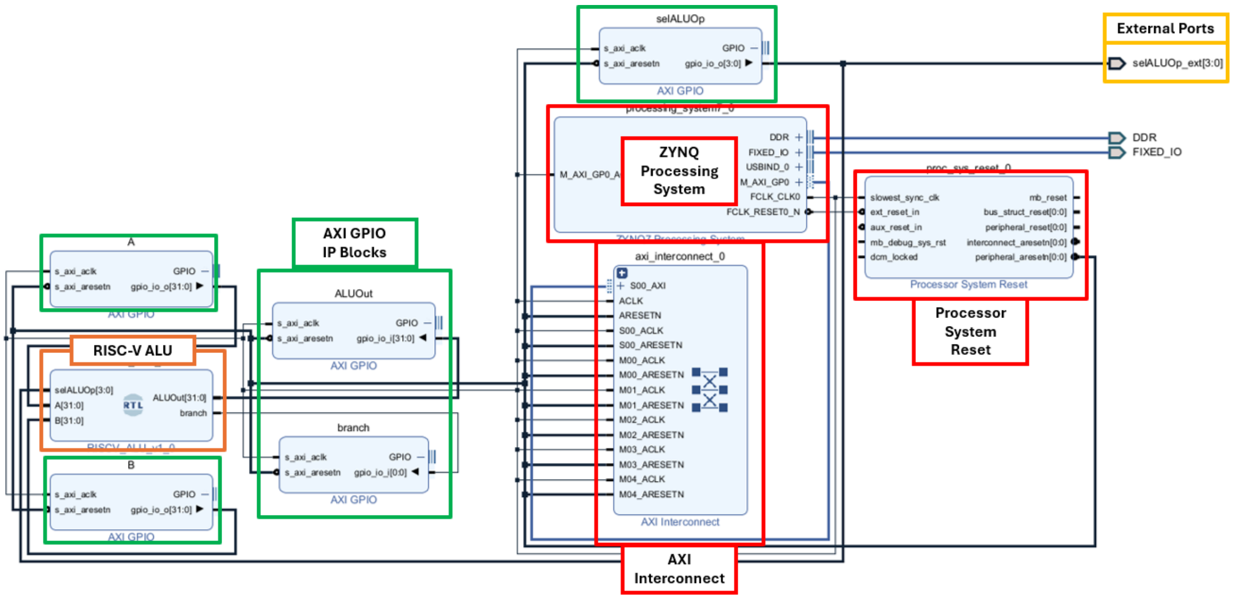
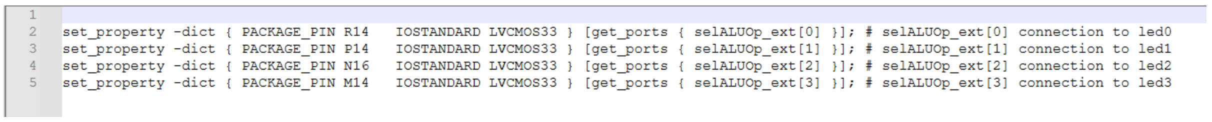


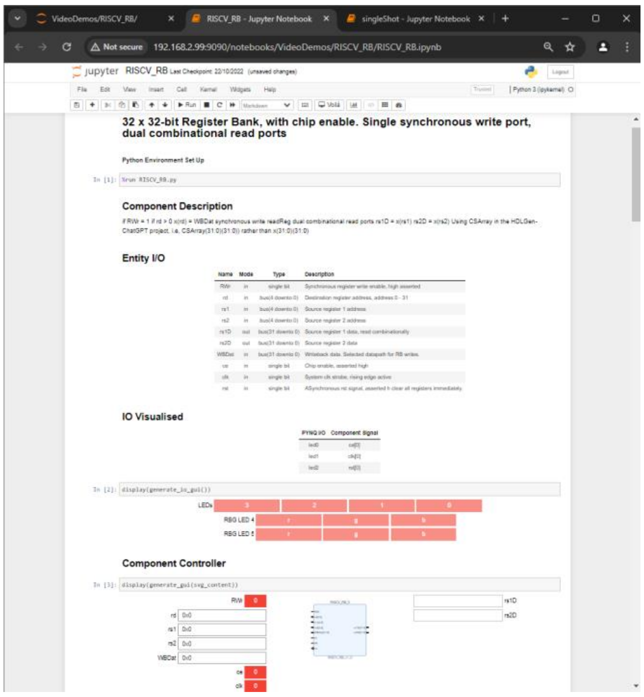
Figure 41 Block design generated for RISC-V ALU



```
set_property -dict { PACKAGE_PIN R14   IOSTANDARD LVCMOS33 } [get_ports { selALUOp_ext[0] }]; # selALUOp_ext[0] connection to led0
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { selALUOp_ext[1] }]; # selALUOp_ext[1] connection to led1
set_property -dict { PACKAGE_PIN N16   IOSTANDARD LVCMOS33 } [get_ports { selALUOp_ext[2] }]; # selALUOp_ext[2] connection to led2
set_property -dict { PACKAGE_PIN M14   IOSTANDARD LVCMOS33 } [get_ports { selALUOp_ext[3] }]; # selALUOp_ext[3] connection to led3
```

Figure 42 Screen capture of master physical constraints file utilised by Vivado to make connections to board LEDs

Figure 43 presents the companion driver application generated by the SoC Builder. The functionality of this application is demonstrated in the accompanying video linked in table 3 previously. Figure 44 presents the results from two independent test cases which have been executed.


VideoDemos/RISCV_RB/
RISCV_RB - Jupyter Notebook
singleShot - Jupyter Notebook
Not secure
192.168.2.99:9090/notebooks/VideoDemos/RISCV_RB/RISCV_RB.ipynb
jupyter RISCV_RB Last Checkpoint: 22/10/2022 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 (ipykernel)
Run
Markdown
Voilà

32 x 32-bit Register Bank, with chip enable. Single synchronous write port, dual combinational read ports

Python Environment Set Up

```
In [1]: %run RISCV_RB.py
```

Component Description

if RWr = 1 if rd > 0 x(rd) = WBDat synchronous write readReg dual combinational read ports rs1D = x(rs1) rs2D = x(rs2) Using CSArray in the HDLGen-ChatGPT project, i.e. CSArray(31:0)(31:0)) rather than x(31:0)(31:0)

Entity I/O

Name	Mode	Type	Description
RWr	in	single bit	Synchronous register write enable, high asserted
rd	in	bus(4 downto 0)	Destination register address, address 0 - 31
rs1	in	bus(4 downto 0)	Source register 1 address
rs2	in	bus(4 downto 0)	Source register 2 address
rs1D	out	bus(31 downto 0)	Source register 1 data, read combinationally
rs2D	out	bus(31 downto 0)	Source register 2 data
WBDat	in	bus(31 downto 0)	Writeback data. Selected datapath for RB writes.
ce	in	single bit	Chip enable, asserted high
clk	in	single bit	System clk strobe, rising edge active
rst	in	single bit	ASynchronous rst signal, asserted h clear all registers immediately

IO Visualised

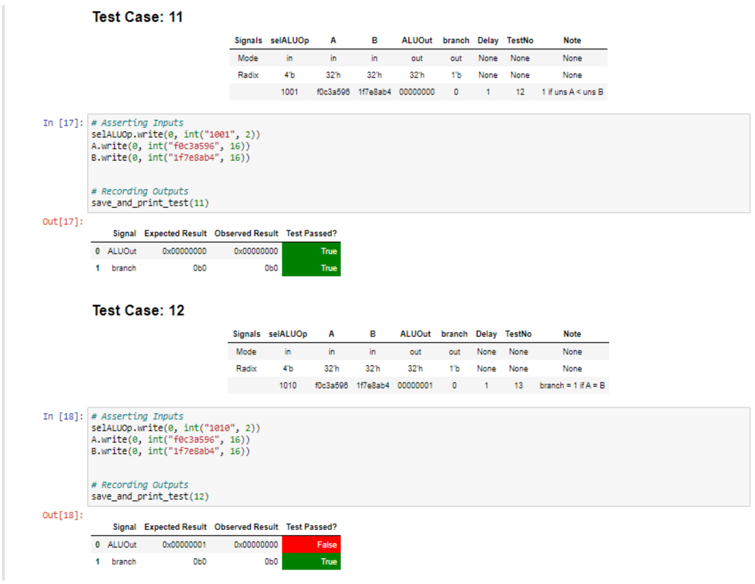
PYNQ I/O	Component Signal
led0	ce[0]
led1	clk[0]
led2	rst[0]

```
In [2]: display(generate_io_gui())
```

Component Controller

```
In [3]: display(generate_gui(svg_content))
```

Figure 43 Jupyter Notebook driver application generated for RISC-V ALU project



Test Case: 11

Signals	selALUOp	A	B	ALUOut	branch	Delay	TestNo	Note
Mode	in	in	in	out	out	None	None	None
Radix	4'b	32'h	32'h	32'h	1'b	None	None	None
	1001	f0c3a596	1f7e8ab4	00000000	0	1	12	1 if uns A < uns B

```
In [17]: # Asserting Inputs
         selALUOp.write(0, int("1001", 2))
         A.write(0, int("f0c3a596", 16))
         B.write(0, int("1f7e8ab4", 16))


         # Recording Outputs
         save_and_print_test(11)
Out[17]:
```

	Signal	Expected Result	Observed Result	Test Passed?
0	ALUOut	0x00000000	0x00000000	True
1	branch	0b0	0b0	True

Test Case: 12

Signals	selALUOp	A	B	ALUOut	branch	Delay	TestNo	Note
Mode	in	in	in	out	out	None	None	None
Radix	4'b	32'h	32'h	32'h	1'b	None	None	None
	1010	f0c3a596	1f7e8ab4	00000001	0	1	13	branch = 1 if A = B

```
In [18]: # Asserting Inputs
         selALUOp.write(0, int("1010", 2))
         A.write(0, int("f0c3a596", 16))
         B.write(0, int("1f7e8ab4", 16))


         # Recording Outputs
         save_and_print_test(12)
Out[18]:
```

	Signal	Expected Result	Observed Result	Test Passed?
0	ALUOut	0x00000001	0x00000000	False
1	branch	0b0	0b0	True

Figure 44 Screenshot from Jupyter Notebook driver of test case execution

5.5 Counter

5.5.1 Overview

This sub-section presents a simple sequential counter project using HDLGen-ChatGPT and PYNQ SoC Builder. The aim of this project is to demonstrate sequential functionality and the ability to make internal signals available externally.

Description	**Link**
Video Demonstration	YouTube
Source Files	GitHub

Table 7 Link to counter design online resources

5.5.2 Design

Figure 45 illustrates a component symbol for a simple 4-bit counter. This 4-bit counter is cascadable, chip-enabled, up/down and is loadable. It also has an asynchronous reset. Figure 46 presents the component symbol generated as part of the Vivado block design. It contains three additional signals on the output, *int_NS, int_CS* and *int_intTC*. The *int_* prefix is intended to denote that these signals are internal signals.

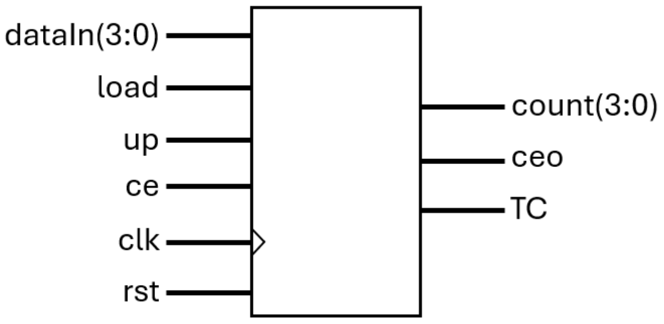


Figure 45 Counter component symbol

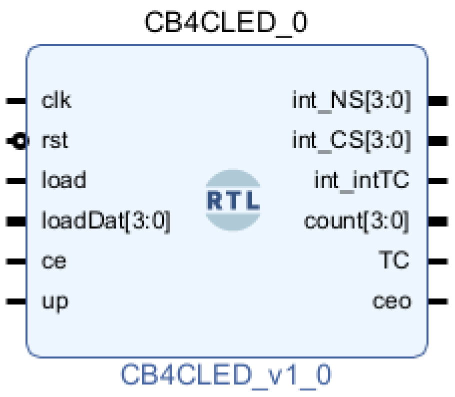
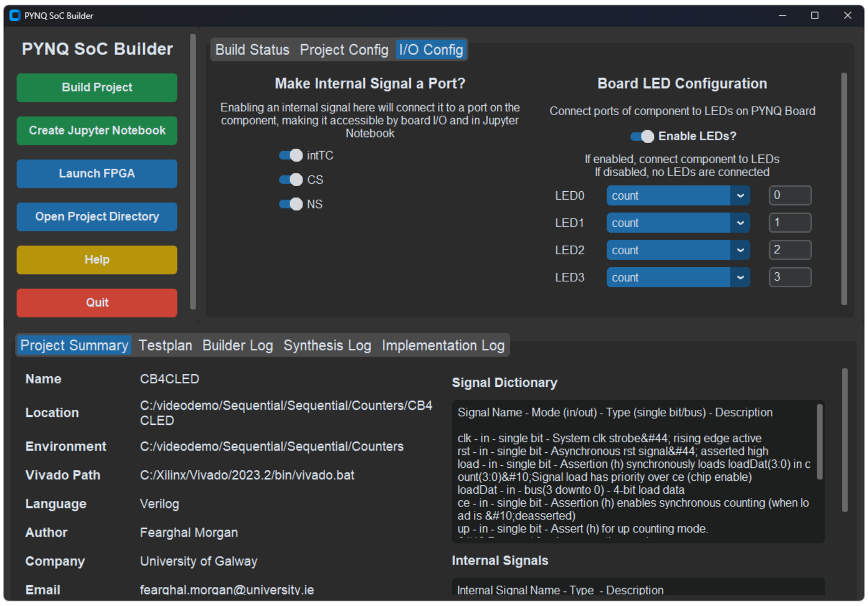


Figure 46 Counter component symbol in Vivado block design

5.5.3 SoC Builder Configuration

The configuration displayed in figure 47 is used to implement the functionality outlined in the previous section, where the NS, CS and int_TC internal signals are exposed as external signals. Additionally, the count output signal is connected to board LEDs.



Make Internal Signal a Port?

Enabling an internal signal here will connect it to a port on the component, making it accessible by board I/O and in Jupyter Notebook

- intTC ✓
- CS ✓
- NS ✓

Board LED Configuration

Connect ports of component to LEDs on PYNQ Board

Enable LEDs? ✓

If enabled, connect component to LEDs
If disabled, no LEDs are connected

LED0	count	0
LED1	count	1
LED2	count	2
LED3	count	3

Name	CB4CLED
Location	C:/videodemo/Sequential/Sequential/Counters/CB4CLED
Environment	C:/videodemo/Sequential/Sequential/Counters
Vivado Path	C:/Xilinx/Vivado/2023.2/bin/vivado.bat
Language	Verilog
Author	Fearghal Morgan
Company	University of Galway
Email	fearghal.morgan@universitv.ie

Signal Dictionary

```
Signal Name - Mode (in/out) - Type (single bit/bus) - Description

clk - in - single bit - System clk strobe&#44; rising edge active
rst - in - single bit - Asynchronous rst signal&#44; asserted high
load - in - single bit - Assertion (h) synchronously loads loadDat(3:0) in c
ount(3:0)&#10;Signal load has priority over ce (chip enable)
loadDat - in - bus(3 downto 0) - 4-bit load data
ce - in - single bit - Assertion (h) enables synchronous counting (when lo
ad is &#10;deasserted)
up - in - single bit - Assert (h) for up counting mode.
```

Internal Signals

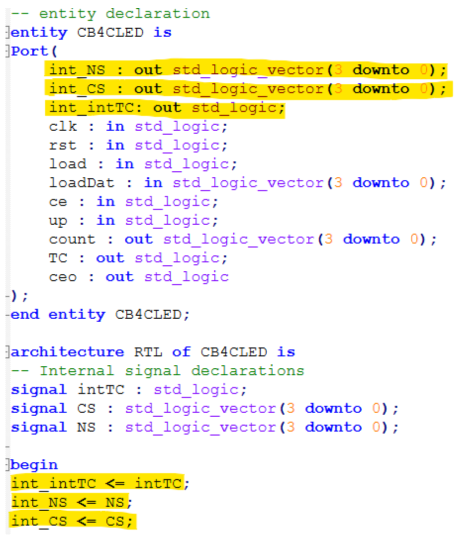
```
Internal Signal Name - Type  - Description
```

Figure 47 PYNQ SoC Builder configuration for 4-bit counter project

5.5.4 Results

Figure 48 highlights the changes made to the user's HDL model automatically to expose internal signals as outputs of the model. Two additions are made to the HDL model to achieve the desired functionality:

1. Additional ports are declared in port map
2. Concurrent statements to assign outputs are added



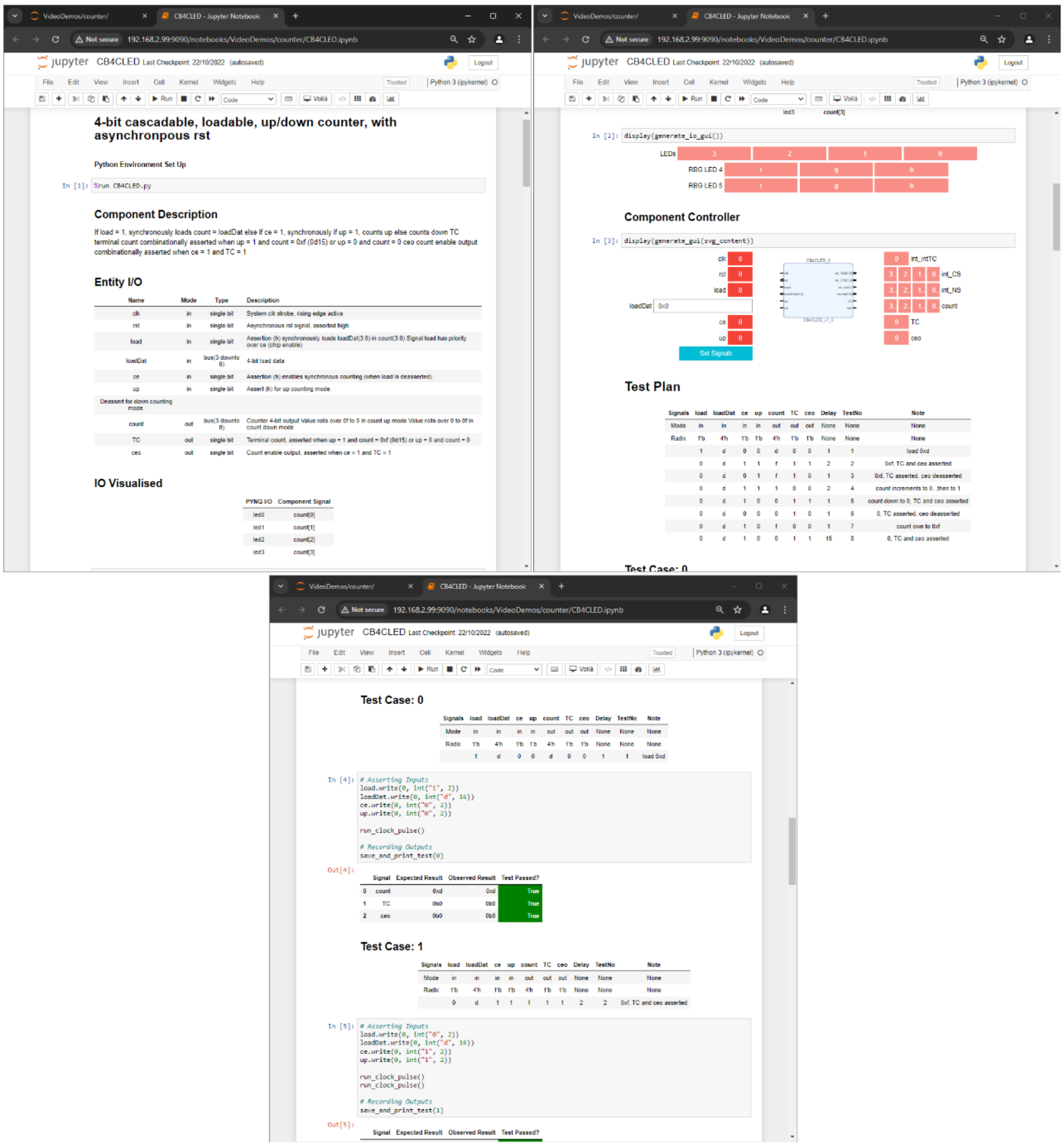
```
-- entity declaration
entity CB4CLED is
Port(
    int_NS : out std_logic_vector(3 downto 0);
    int_CS : out std_logic_vector(3 downto 0);
    int_intTC: out std_logic;
    clk : in std_logic;
    rst : in std_logic;
    load : in std_logic;
    loadDat : in std_logic_vector(3 downto 0);
    ce : in std_logic;
    up : in std_logic;
    count : out std_logic_vector(3 downto 0);
    TC : out std_logic;
    ceo : out std_logic
);
end entity CB4CLED;

architecture RTL of CB4CLED is
-- Internal signal declarations
signal intTC : std_logic;
signal CS : std_logic_vector(3 downto 0);
signal NS : std_logic_vector(3 downto 0);

begin
int_intTC <= intTC;
int_NS <= NS;
int_CS <= CS;
```

Figure 48 VHDL code injected into model to expose internal signals

Figure 49 presents three screenshots from the Jupyter Notebook generated for the counter model.



4-bit cascadable, loadable, up/down counter, with asynchronpous rst

Python Environment Set Up

```
In [1]: %run CB4CLED.py
```

Component Description

If load = 1, synchronously loads count = loadDat else if ce = 1, synchronously if up = 1, counts up else counts down TC terminal count combinationally asserted when up = 1 and count = 0xf (0d15) or up = 0 and count = 0 ceo count enable output combinationally asserted when ce = 1 and TC = 1

Entity I/O

Name	Mode	Type	Description
clk	in	single bit	System clk strobe, rising edge active
rst	in	single bit	Asynchronous rst signal, asserted high
load	in	single bit	Assertion (h) synchronously loads loadDat(3:0) in count(3:0) Signal load has priority over ce (chip enable)
loadDat	in	bus(3 downto 0)	4-bit load data
ce	in	single bit	Assertion (h) enables synchronous counting (when load is deasserted)
up	in	single bit	Assert (h) for up counting mode.
Deassert for down counting mode.			
count	out	bus(3 downto 0)	Counter 4-bit output Value rolls over 0f to 0 in count up mode Value rolls over 0 to 0f in count down mode
TC	out	single bit	Terminal count, asserted when up = 1 and count = 0xf (0d15) or up = 0 and count = 0
ceo	out	single bit	Count enable output, asserted when ce = 1 and TC = 1

IO Visualised

PYNQ I/O	Component Signal
led0	count[0]
led1	count[1]
led2	count[2]
led3	count[3]

```
In [2]: display(generate_io_gui())
```

Component Controller

```
In [3]: display(generate_gui(svg_content))
```

Test Plan

Signals	load	loadDat	ce	up	count	TC	ceo	Delay	TestNo	Note
Mode	in	in	in	in	out	out	out	None	None	None
Radix	1'b	4'h	1'b	1'b	4'h	1'b	1'b	None	None	None
	1	d	0	0	d	0	0	1	1	load 0xd
	0	d	1	1	f	1	1	2	2	0xf, TC and ceo asserted
	0	d	0	1	f	1	0	1	3	0xf, TC asserted, ceo deasserted
	0	d	1	1	1	0	0	2	4	count increments to 0, ,then to 1
	0	d	1	0	0	1	1	1	5	count down to 0, TC and ceo asserted
	0	d	0	0	0	1	0	1	6	0, TC asserted, ceo deasserted
	0	d	1	0	f	0	0	1	7	count own to 0xf
	0	d	1	0	0	1	1	15	8	0, TC and ceo asserted

Test Case: 0

Signals	load	loadDat	ce	up	count	TC	ceo	Delay	TestNo	Note
Mode	in	in	in	in	out	out	out	None	None	None
Radix	1'b	4'h	1'b	1'b	4'h	1'b	1'b	None	None	None
	1	d	0	0	d	0	0	1	1	load 0xd

```
In [4]: # Asserting Inputs
        load.write(0, int("1", 2))
        loadDat.write(0, int("d", 16))
        ce.write(0, int("0", 2))
        up.write(0, int("0", 2))

        run_clock_pulse()

        # Recording Outputs
        save_and_print_test(0)
Out[4]:
```

	Signal	Expected Result	Observed Result	Test Passed?
0	count	0xd	0xd	True
1	TC	0b0	0b0	True
2	ceo	0b0	0b0	True

Test Case: 1

Signals	load	loadDat	ce	up	count	TC	ceo	Delay	TestNo	Note
Mode	in	in	in	in	out	out	out	None	None	None
Radix	1'b	4'h	1'b	1'b	4'h	1'b	1'b	None	None	None
	0	d	1	1	f	1	1	2	2	0xf, TC and ceo asserted

```
In [5]: # Asserting Inputs
        load.write(0, int("0", 2))
        loadDat.write(0, int("d", 16))
        ce.write(0, int("1", 2))
        up.write(0, int("1", 2))

        run_clock_pulse()
        run_clock_pulse()

        # Recording Outputs
        save_and_print_test(1)
Out[5]:
```

Signal	Expected Result	Observed Result	Test Passed?

Figure 49 Generated Jupyter Notebook driver application for Counter

5.5 Single Shot

5.5.1 Overview

In this sub-section, a single shot signal generator is designed and deployed to the PYNQ Z2 hardware. This relatively simple sequential design can appear as a subcomponent to many designs which require pulse assertion such as a button input.

Description	**Link**
Video Demonstration	YouTube
Source Files	GitHub

Table 8 Link to single shot design online resources

5.5.2 Design

The design consists of three inputs and one output, as presented in figure 50. No test plan is written for this design and the timing diagram as seen in figure 51, is performed manually in the accompanying video using the GUI-based controller functionality.

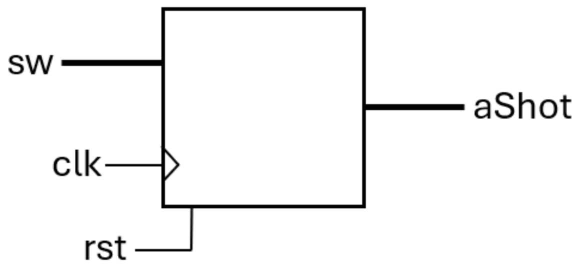


Figure 50 Single shot component symbol

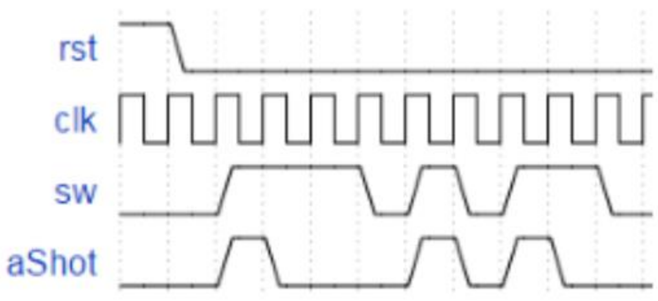
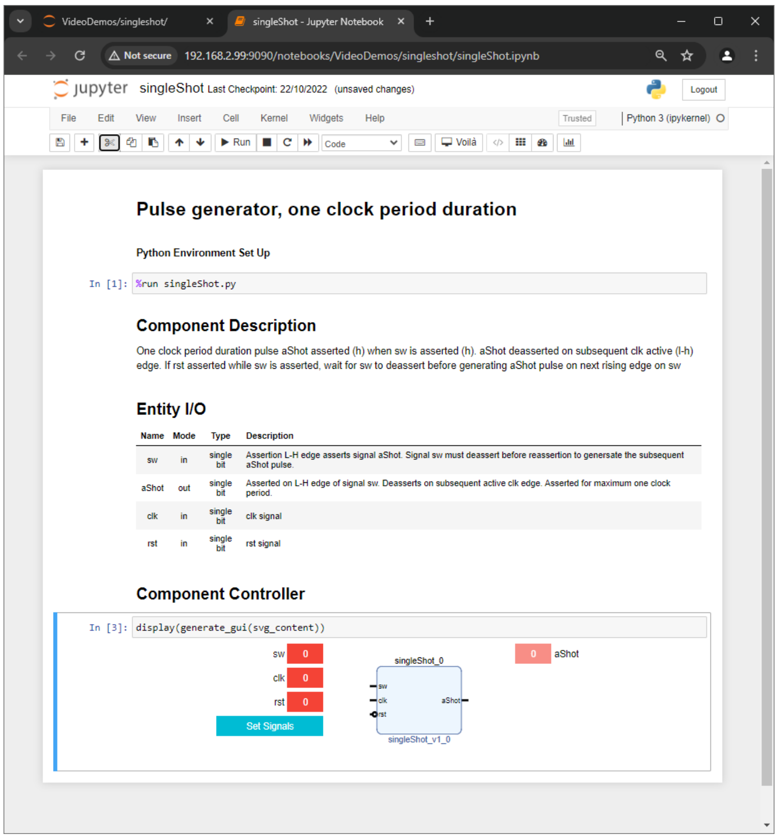


Figure 51 Timing diagram of single shot design

5.5.3 Results

Figures 53 and 54 present the generated Jupyter Notebook driver application and block design respectively. The Jupyter Notebook does not contain any test cases or test plan.



VideoDemos/singleshot/ | singleShot - Jupyter Notebook

Not secure 192.168.2.99:9090/notebooks/VideoDemos/singleshot/singleShot.ipynb

jupyter singleShot Last Checkpoint: 22/10/2022 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help | Trusted | Python 3 (ipykernel)

Pulse generator, one clock period duration

Python Environment Set Up

```
In [1]: %run singleShot.py
```

Component Description

One clock period duration pulse aShot asserted (h) when sw is asserted (h). aShot deasserted on subsequent clk active (l-h) edge. If rst asserted while sw is asserted, wait for sw to deassert before generating aShot pulse on next rising edge on sw

Entity I/O

Name	Mode	Type	Description
sw	in	single bit	Assertion L-H edge asserts signal aShot. Signal sw must deassert before reassertion to genersate the subsequent aShot pulse.
aShot	out	single bit	Asserted on L-H edge of signal sw. Deasserts on subsequent active clk edge. Asserted for maximum one clock period.
clk	in	single bit	clk signal
rst	in	single bit	rst signal

Component Controller

```
In [3]: display(generate_gui(svg_content))
```

Figure 52 Jupyter Notebook driver application for Single Shot - No test bench or I/O Configuration

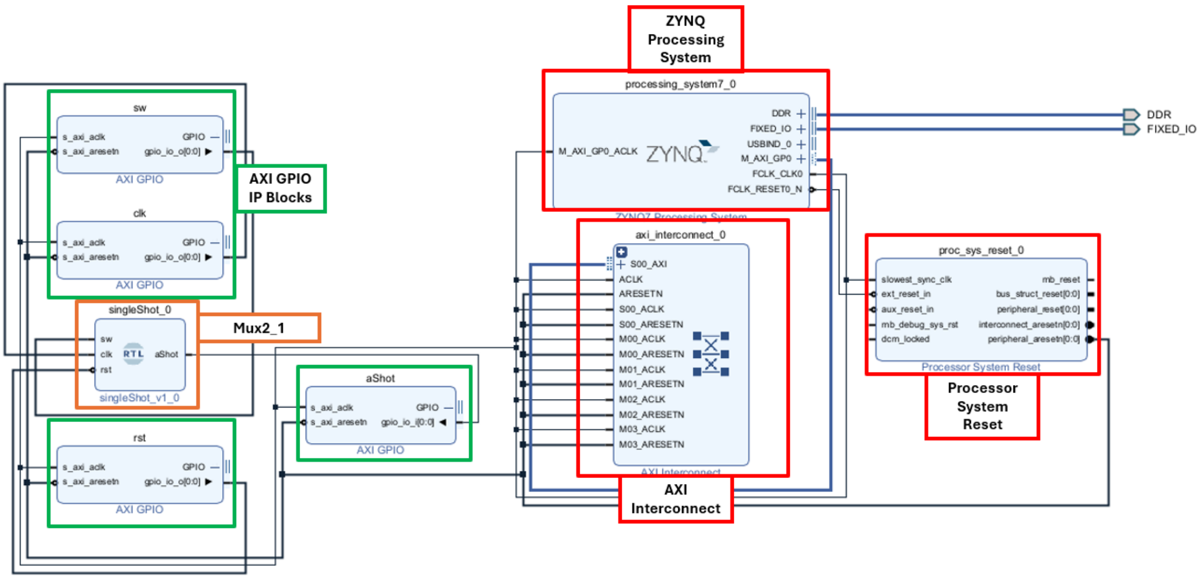


Figure 53 Block design generated for Singleshot Pulse Generator in Vivado

Future Work

5.1 Expanded I/O Connectivity

The PYNQ SoC Builder currently supports I/O connectivity with the four binary LEDs present on the PYNQ Z2 board. The PYNQ Z2 board has much more I/O connectivity to offer, such as support for microcontroller shields for Arduino and Raspberry Pi, input I/O such as switches and more. Greater I/O support and functionality could be implemented with relative ease, although it would require further testing and development of more advanced designs.

5.2 Scalable Remote Platform

As it stands, each PYNQ Z2 added to the remote platform is independent and requires its own Ngrok credentials. A centralised server or server-like facility would be required to increase the scalability of the design such that any number of PYNQ boards can be added or removed, ultimately increasing the accessibility of FPGA hardware in universities.

5.3 Integration with HDLGen-ChatGPT

Integration of the PYNQ SoC Builder application with HDLGen-ChatGPT has the potential to enhance user experience by streamlining the experience by consolidating design and deployment workflows into a single environment. This is particularly advantageous for users or potential users of HDLGen-ChatGPT who may not be aware of the benefits or existence of the PYNQ SoC Builder when deploying designs.

Conclusions

In conclusion, I believe project has been successful in achieving its aims and objectives. The PYNQ SoC Builder application developed throughout this project has been tested using a wide range of projects such as those detailed in chapter 5 and more throughout its development cycle.

I believe that the PYNQ SoC Builder's capability extends beyond the designs detailed in this report and has great potential for use in education settings. I believe the PYNQ SoC Builder has potential to facilitate RISC-V core designs and more intricate advanced designs.

In education settings, while FPGA hardware deployment might not be directly relevant to all student coursework, the ability to interact with designs deployed to real FPGA hardware provides a valuable learning experience when compared to purely simulation-based tools. When paired with a remote working environment, students can gain valuable hands-on experience with FPGA hardware.

References

Byrne, J.P. 2023. A client-based python automation tool for fast capture of HDL based SoC capture and implementation. *University of Galway*

Viegas, C., Pavani, A., Lima, N., Marques, A., Pozzo, I., Dobboletta, E., Atencia, V., Barreto, D., Calliari, F., Fidalgo, A. and Lima, D., 2018. Impact of a remote lab on teaching practices and student learning. *Computers & Education*, *126*, pp.201-216.

Morgan, F., Cawley, S., Coffey, A., Callaly, F., Lyons, D., O'Loughlin, D., Krewer, F., Dwivedi, M., Marechal, K., De Santana, E.G. and Neelen, M., 2014, October. viciLogic: Online learning and prototyping platform for digital logic and computer architecture. In *eChallenges e-2014 Conference Proceedings* (pp. 1-9). IEEE.

Appendix

Appendix A: Online Resources

All online resources related to this project:

Description	**Link**
PYNQ SoC Builder GitHub Repository	GitHub
HDLGen-ChatGPT GitHub Repository	GitHub
Project Kanban	GitHub
Logicademy (HDLGen-ChatGPT GitHub Organisation)	GitHub
Installation Guide (Text)	GitHub README
Installation Guide (Video)	YouTube
Creating an Ngrok account	GitHub PDF
Installing Ngrok on PYNQ FPGA	YouTube
Capturing 2-to-1 Mux in HDLGen-ChatGPT	YouTube GitHub
Building 2-to-1 Mux in PYNQ SoC Builder	YouTube
RISC-V Arithmetic Logic Unit (ALU)	YouTube GitHub
4-bit Counter	YouTube GitHub
Single Shot	YouTube GitHub

Table 9 List of all available online resources produced for this project