Volume

1

VT ENTERPRISES CONSULTING SERVICES

GitHub Project: P2654Simulations Tooling & Environment

User’s Guide

GitHub Project: P2654Simulations Tooling & Environment

User’s Guide

© VT Enterprises Consulting Services, 2020

Bradford G. Van Treuren

7 Barry Road

Lambertville, NJ 08530

Phone 609.240.3066

Revision 1.0.0, June 2020

Table of Contents

[Introduction 1](#_Toc47119406)

[How to Use This Manual 1](#_Toc47119407)

[Background Information 2](#_Toc47119408)

[Overview 3](#_Toc47119409)

[Acknowledgements 4](#_Toc47119410)

[Installation 5](#_Toc47119411)

[Installing Python 5](#_Toc47119412)

[Installing Git 5](#_Toc47119413)

[Installing PyCharm Community Edition 6](#_Toc47119414)

[Setting Up PyCharm 6](#_Toc47119415)

[Cloning P2654Simulations Project 6](#_Toc47119416)

[Adding Additional Packages to the Project 8](#_Toc47119417)

[Virtual ATE 10](#_Toc47119418)

[Device Testing 13](#_Toc47119419)

[myHDL Process Decorators 14](#_Toc47119420)

[@always\_comb 15](#_Toc47119421)

[@always\_seq 15](#_Toc47119422)

[@always 15](#_Toc47119423)

[@instance 15](#_Toc47119424)

[myHDL Documentation 15](#_Toc47119425)

[JTAGController class 17](#_Toc47119426)

[JTAGController2 class 19](#_Toc47119427)

[I2CController class 19](#_Toc47119428)

[SPIController class 20](#_Toc47119429)

[GPIO Access 20](#_Toc47119430)

[Initializing, Configuring, and Terminating the ATE in Testbenches 21](#_Toc47119431)

[Initializing, Configuring, and Terminating the ATE in Applications 22](#_Toc47119432)

[AbstractBoard Class 30](#_Toc47119433)

[Interface Classes 30](#_Toc47119434)

[BoardGPIOInterface class 30](#_Toc47119435)

[BoardJTAGInterface class 30](#_Toc47119436)

[BoardI2CInterface class 30](#_Toc47119437)

[BoardSPIInterface class 31](#_Toc47119438)

[Specialized Board Designs 31](#_Toc47119439)

[Schematic Diagram 31](#_Toc47119440)

[RTL Representation of Schematic 32](#_Toc47119441)

[Specialized Device Designs 38](#_Toc47119442)

[RTL Design Test Bench 38](#_Toc47119443)

[RTL Design ATE Adapter Board 39](#_Toc47119444)

[Specialized IP Core Designs 42](#_Toc47119445)

[RTL Design ATE Adapter Board 48](#_Toc47119446)

[Use Monitor instances 49](#_Toc47119447)

[simservice 51](#_Toc47119448)

[Telnet Service 51](#_Toc47119449)

[Telnetsrv package implementation 52](#_Toc47119450)

[ATE class 52](#_Toc47119451)

[WishboneMaster class 52](#_Toc47119452)

[DisplayPanel 52](#_Toc47119453)

[xsvftool 53](#_Toc47119454)

[GPIOTest Board 54](#_Toc47119455)

[JTAGTest Board 55](#_Toc47119456)

[JTAG2Test Board 56](#_Toc47119457)

[I2CTest Board 56](#_Toc47119458)

[SPITest Board 56](#_Toc47119459)

[I2CDeviceSEP 57](#_Toc47119460)

[SPIDevice 57](#_Toc47119461)

[General Instruments 59](#_Toc47119462)

[Clock Counter 59](#_Toc47119463)

[Clock Generator 59](#_Toc47119464)

[Comparator 59](#_Toc47119465)

[LED 59](#_Toc47119466)

[Noise Maker 59](#_Toc47119467)

[Power Supply Monitor 59](#_Toc47119468)

[PseudoLED 59](#_Toc47119469)

[SimulatedBIST 59](#_Toc47119470)

[Thermometer 59](#_Toc47119471)

[IEEE 1149.1 Instruments 59](#_Toc47119472)

[STD1149\_1\_TAP 59](#_Toc47119473)

[TIR 59](#_Toc47119474)

[TDR 59](#_Toc47119475)

[IEEE 1500 Instruments 59](#_Toc47119476)

[SELWIR 59](#_Toc47119477)

[WBY 59](#_Toc47119478)

[WDRMUX 59](#_Toc47119479)

[WIR 59](#_Toc47119480)

[WIRMUX 59](#_Toc47119481)

[WSP 59](#_Toc47119482)

[WSReg 59](#_Toc47119483)

[IEEE 1687 Instruments 59](#_Toc47119484)

[SIB\_MUX\_PRE 59](#_Toc47119485)

[SIB\_MUX\_POST 59](#_Toc47119486)

[SReg 59](#_Toc47119487)

[Index 60](#_Toc47119488)

Chapter

1

Introduction

This chapter provides an introduction to the use of the P2654Simulations Tools & Environment to aid in understanding the purpose and scope of the GitHub Project and respective code base.

t

he P2654Simulations GitHub Project was created as a repository for experiments supporting IEEE P2654 and P1687.1 working groups. The environment consists of a hardware simulation environment, an Automated Test Equipment (ATE) emulated interface, a Telnet Service to interact with the simulation environment, and device driver libraries allowing access to control the ATE interfaces to the simulated hardware.

icon key

1. Valuable information

☹ Cautions

⮱ User Required Actions

1. Programming Examples
2. Semantic

# How to Use This Manual

The “icon key” at left identifies various symbols you will find in the margin of the text. These symbols will provide a key as to the role of each paragraph description. Valuable information categorizes information worth noting as useful for your own applications. Cautions are important issues that could impact the way your system operations while testing with the simulation. User Required Actions are issues or procedures the user must perform for the correct behavior to take place. Programming Examples are Python code segments that describe how to perform an operation. Code that is left out of an example is represented by ellipses (…). Lastly, Semantic Information is the information describing the semantics behavior of the Python code and the intent for the described directive.

# Background Information

🗁

Hardware simulation environments typically consist of simulation tools for VHDL, Verilog, or SystemC hardware description languages that are used to describe the behavior in a form of a Register Transfer Language (RTL) defining the behavior of hardware signals in a design. These simulation tools contain features to allow for developing test benches, using the same description languages, to perform unit and design verification testing of a particular design module (entity). The designer would apply the test bench and ensure the assertions are satisfied. The designer also has the ability to view waveforms of the signal states over time to ensure timing between signal interactions is occurring at the correct time. The test benches also provide means to validate signals are changing at the appropriate time using various forms of wait delays in separate processes running in parallel to the main design under test.

Design and test in these environments is targeted for device or intellectual property (IP) hardware designers specifically managing the behavior of the signals inside a hardware design. Thus, interaction with application code supplied by a user of a device is not considered. There has been some development of such interfaces recently. Verilog Procedural Interface (VPI) builds on top of the Verilog PLI. Verilog PLI (Programming Language Interface) is a mechanism to invoke C or C++ functions from Verilog code. The Verilog Procedural Interface (VPI) is a C-programming interface for the Verilog Hardware Description Language (HDL). VPI consists of a set of access and utility routines that you call from standard C programming language functions. These routines interact with the instantiated simulation objects contained in your Verilog HDL design.[1] (Tala, 2014)

Unfortunately, these interfaces are defined at a low level close to the IP design and do not reflect the target environment the IEEE P2654 and IEEE P1687.1 are targeting. IEEE P1687.1 builds on the IEEE Std 1687 to provide test access to instrumentation inside devices. This standard describes how to communicate with register of an IP design from the device edge pins. IEEE P2654 is targeting the management and control of the interfaces at the board and system levels connecting to such device pin interfaces. Thus, VPI requires quite a bit of coding supporting such environments.

P2654Simulations leverages the RTL defined using the myHDL Python package. myHDL uses constructs in Python to perform the simulation of signals similar to that of VHDL and Verilog. The design is written is a special semantic form with Python that is also able to generate VHDL and Verilog code from the myHDL code to migrate a design to a hardware environment, like and FPGA or CPLD. myHDL is also able to produce the signal waveform diagrams for evaluation by designers of the IP module being used.

The main interface to the P2654Simulations environment is through standardized test bus interfaces access by a virtual Automated Test Equipment controller. The virtual ATE provides a Wishbone hardware bus where JTAG, I2C, SPI, and GPIO host controllers are connected via the myHDL simulation environment. The virtual ATE provides access to the Wishbone bus using a set of Telnet commands that control read and write operations on the Wishbone bus. A Telnet service interface was chosen as most languages support some means of communications as a Telnet Client to a Telnet Server. This eliminates the dependence on C/C++ as the host language in VPI. The Telnet interface allows any product to be able to connect up to the simulation environment by writing their own driver interfaces to the test bus controllers using Telnet commands.

# Overview

This guide is organized with the following sections:

1. Introduction
2. Installation
3. Architectural Concepts
4. Basic Concepts
5. Virtual ATE
6. Defining a Board Design
7. Defining a Device Design
8. Defining IP Cores and Instruments
9. Programming Suggestions
10. Applications and Tools
11. Appendix A: Use Case Examples
12. Appendix B: Device Examples
13. Appendix C: Instrument Examples

# Acknowledgements

The concept for the P2654Simulations Project came from working with the IEEE P2654 and IEEE P1687.1 working groups. As a member of these groups, it became apparent to me that the groups needed a way to be able to test out ideas on specific use cases found in industry as examples to ensure the concepts identified by the group were going to work. There are various vendors participating in the working groups who use various different languages in their tooling from C, C++, C#, Python, Tcl, and some proprietary languages. It was obvious a single language requirement to the environment was not going to allow vendors to begin testing their tooling with the defined use cases. I would like to thank Michele Portolan, Jeff Rearick, Ian McIntosh, and Heiko Ehrenberg for their expert advice on the use case selection and interaction level required by tool vendors and end users. Many other members of these working groups also shared important insight to the standards that refined what was built as the GitHub project.

Chapter

2

Installation

This chapter describes how to install the simulation environment and tools required running the simulations.

t

he P2654Simulations GitHub Project is based on a Python programming language implementation. The version of Python to use is version 3.6 or newer. It is easier to implement new board designs if one installs a development environment to manage the project and additions.

# Installing Python

The Python interpreter may be downloaded from:

<https://www.python.org/downloads/>

At the time of this writing, the latest version available is 3.8.3. Download the installer for Windows 10 or Linux, depending on what operating system you are installing it on. Follow the directions at this download site to correctly install the interpreter.

# Installing Git

The GitHub repository is managed by the Git version control process. Therefore, the local computer needs to install the Git version control system to be able to clone the project from the repository. Git may be downloaded from:

<https://git-scm.com/downloads>

Follow the instruction on that site to install Git.

# Installing PyCharm Community Edition

The recommended Integrated Development Environment (IDE) is the JetBrains PyCharm Community Edition. This IDE may be downloaded from:

<https://www.jetbrains.com/pycharm/download/>

Follow the installation instructions listed on this website.

# Setting Up PyCharm

To make the installation easier, it makes more sense to install the Python interpreter as a virtual environment (venv) of the project. This way the packages may be tailored for the project without inclusion into the global interpreter. This is especially important for a Linux installation where the user may not have administration privileges

## Cloning P2654Simulations Project

After the PyCharm IDE is installed, it is time to clone the P2654Simulations project from the GitHub repository. Open a web browser and select the following URL to open the GitHub project repository.

<https://github.com/bradfordvt/P2654Simulations>

Make sure the master branch is selected (should be the default branch when you first enter the page). On the right hand side click on the green button that says *Clone or download*. When needing to copy the clone URL in PyCharm, this button will pop-up a window containing the URL to copy the repository. When needed, click on the clipboard to copy the to obtain the location of the project to clone into the PyCharm IDE.

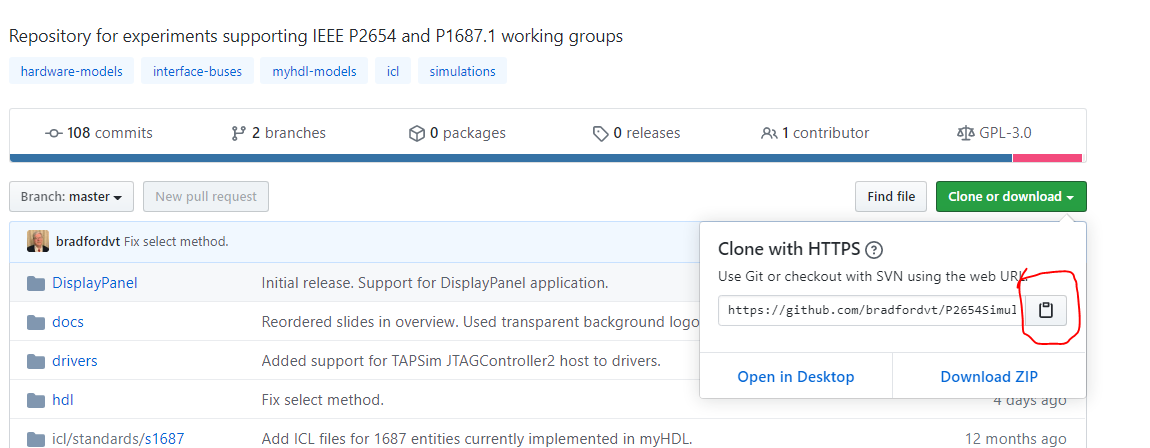


Figure 1 - Copy clone URL from GitHub

Open the PyCharm application. Select File from the menu and New sub-menu. Rename the untitled name to P2654Simulations. Also, make sure the option for venv is chosen and that PyCharm detected the proper location of the installed Python interpreter installed in step one.

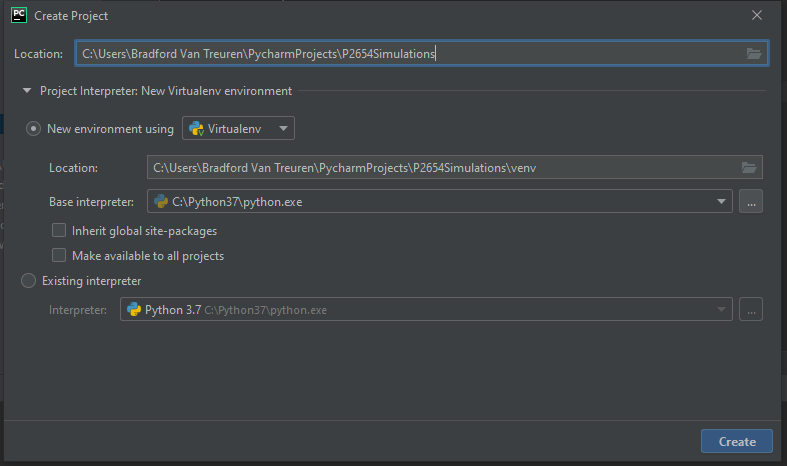


Figure 2 - Create Project Pop-Up

After clicking on the *Create* button, a pop-up will ask where you want to open the project. Choose a new window.

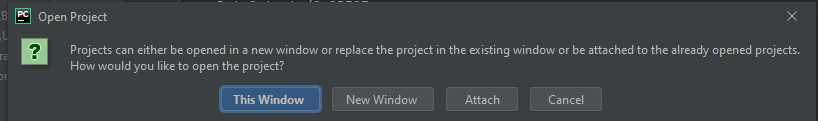


Figure 3 - Open Project Pop-Up

From the menu, select *VCS* and *the Get from version control…* option to clone the GitHub project to the local computer in the IDE.

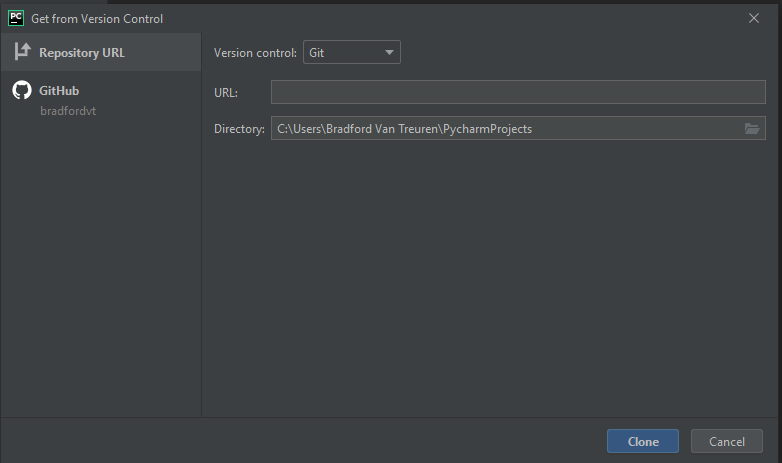


Figure 4 - Get from Version Control Pop-Up

From the repository website, click the clipboard icon to copy the repository URL to be used in the URL text box for the *Get from Version Control* pop-up window. Paste that URL into the text box and click *Clone*.

## Adding Additional Packages to the Project

There are a series of Python packages that will also have to be installed to the Python installation in order to successfully run your simulations. These are:

* myhdl >= 0.11
* Pillow >= 6.0.0
* telnetsrv >= 0.4

These packages are specializations to the Python default environment installed by the previous step. These should be added to the project venv environment.

To add the packages, select *File* from the menu and then *Settings*. In *Settings*, select *Project: P2654Simulations* and *Project Interpreter*. To add the packages, click on the ‘+’ button to the right of the *Latest Version* column and choosing each package from the list. Select the package and click on the Install package button at the bottom of the page.

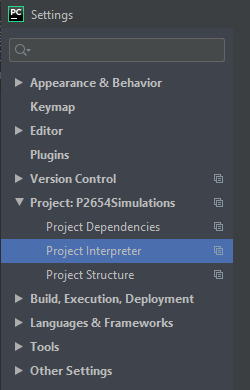


Figure 5 - Settings Pop-Up

Chapter

3

Architectural Concepts

*This chapter describes the composition of the project architecture. Each of the essential components will be described separately.*

t

he P2654Simulations Project is broken into several operational parts attempting to simulate each segment of a system design. Each segment is integrated using interfaces based on standards or well-known industrial designs and interfaces.

System Design Architectural Segments

The P2654Simulations environment attempts to mimic a test system. The top interface provides access to the equivalent of the Automated Test Equipment (ATE) connections to the virtual system via standardized test buses.

The rest of the segments are implemented with myHDL Python code whose edge signals are wired to the ATE test buses for test operation. The ATE test buses wire to a System, Board, or Device interface to be exercised from the ATE Wishbone interface via messages to the Telnet Server from a host application.

# Virtual ATE

The virtual ATE contains two IEEE Std 1149.1 JTAG interfaces, an I2C interface, a SPI interface, and a 16 port input and 16 port output GPIO interface. These test buses are controlled with host controllers implemented with myHDL that are wired to a Wishbone bus. The Wishbone bus is controlled via messages from a Telnet Server thread to control the write and read cycles on the Wishbone bus.

User applications access control of the Wishbone bus using the device driver library for the target application language. The libraries provide high level functions to support each of the test bus interfaces. The JTAG interfaces contain functions to scan vectors as IR or DR vectors in both binary and string representations and a runtest fuction. The I2C functions to write to a register and read from a register along with multi-byte read and multi-byte write functions. The I2C protocol implemented corresponds to a device id followed by a register address before the data transactions occur. The SPI functions of a write and read of a 32-bit value. The GPIO functions are not provided as they are available using direct access to the ATE write and read functions as a single ATE write and ATE read for each operation.

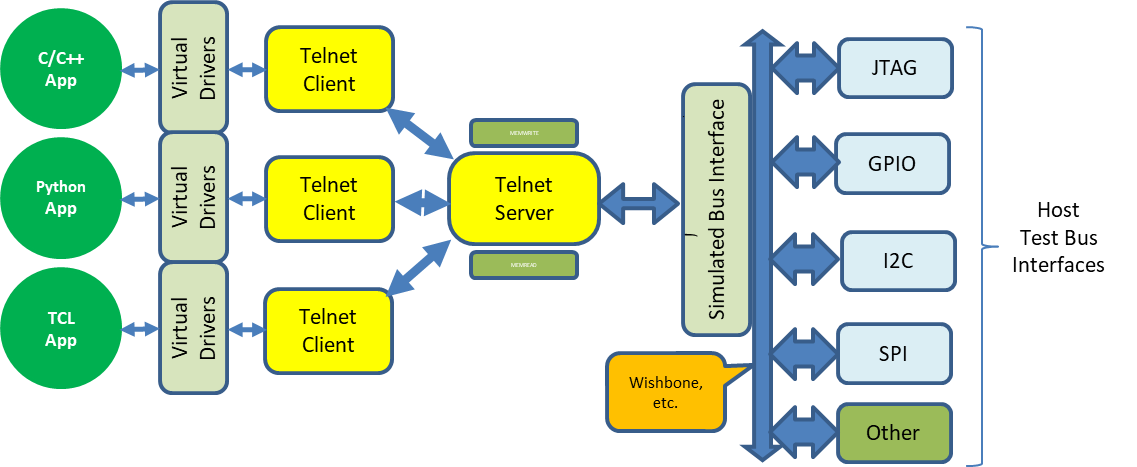


Figure 6 - Virtual ATE Architecture

Board Testing

To configure a test session for a board with the ATE, one needs to define the connections between the board design and the ATE interfaces. This is accomplished in a board design file. A board test application may then exercise the board through the ATE library interfaces. The following diagram describes the connection of the board to the ATE system.

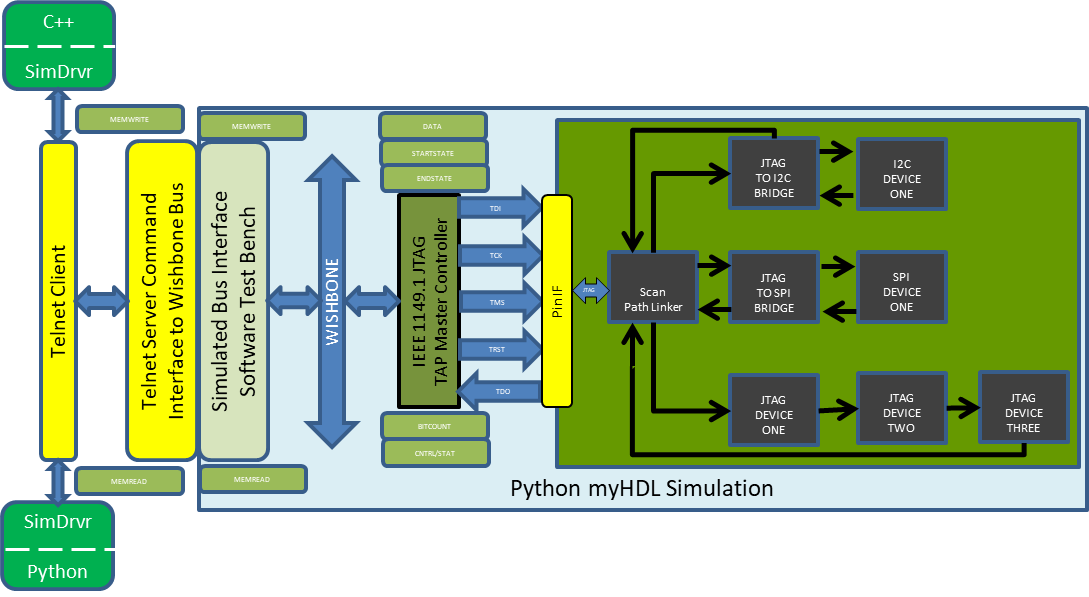


Figure 7 - Board Testing Architecture

Multiple Test Buses for Complex Board Designs

Some board designs require multiple test buses to be connected to the board design. During testing, sequences on each test bus are required to be synchronized from the application controlling the test. This may also be supported by this environment. The following diagram describes an example of such a design.

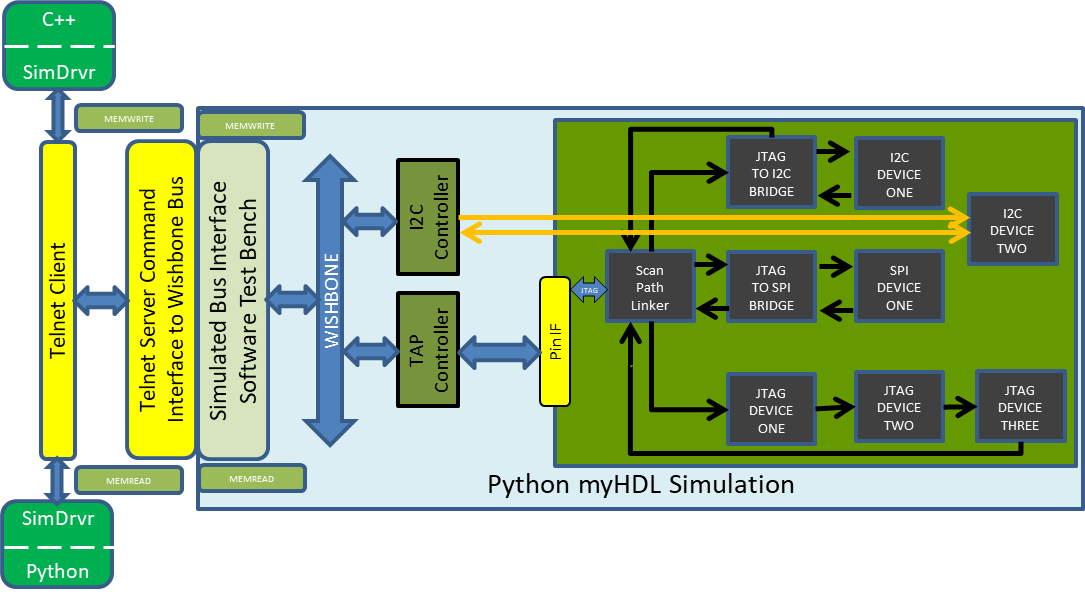


Figure 8 - Complex Board Testing Architecture

# Device Testing

To configure a test session for a device with the ATE, one needs to define the connections between the device design and the ATE interfaces. This is accomplished in a device design file. A device test application may then exercise the device through the ATE library interfaces. The following diagram describes the connection of the device to the ATE system.

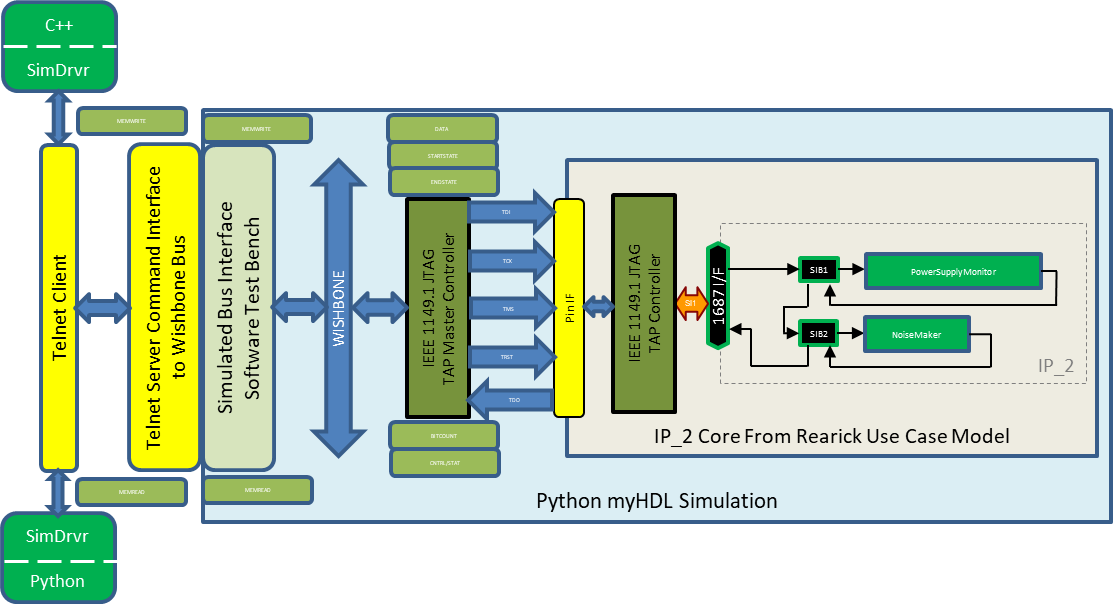


Figure 9 - Device Testing Architecture

Chapter

4

Basic Concepts

*This chapter describes the basic concepts for describing a design and how the integration into the simulation test system is made.*

t

he primary component of the P2654Simulations Project is the myHDL simulation thread. myHDL provides the logic behavior of the design under test. The purpose of the P2654Simulations Project is to provide a means for vendors and working groups to assess and demonstrate control and management concepts used by the IEEE P2654 and IEEE P1687.1 standardization efforts. The project provides common use case benchmarks available for demonstration verification for different customer use cases identified by the working groups.

myHDL Signal representation

myHDL implements the Signal class to define a signal to the simulator. To assign a new value to the signal, myHDL implements the *next* attribute that is assigned the value to be assigned to the signal on the next clock cycle.

A Signal may be assigned a value that is Boolean, an integer, or a binary string. This is typically represented as an intbv type (Integer or Bit Vector type class) defined by myHDL.

# myHDL Process Decorators

myHDL takes advantage of Python decorators to provide wrappers around functions to provide insight for the simulator as to when a function should run based on the transitions of key signals. These decorators mimic the behaviors provided by the combinatorial and sequential processes found in VHDL and Verilog.

## @always\_comb

The *@always\_comb* decorator is used to specify a function is a combinatorial method that is to run whenever any of the signals on the right hand side of an assignment statement changes.

## @always\_seq

The *@always\_seq* decorator is used to specify a function is a sequential method that is sensitive to clock signals and reset signals. When a clock signal changes, the function will be called by the simulator. If a reset signal is detected, the function will also be called by the simulator.

## @always

The *@always* decorator is used to specify a function is a sequential method that is sensitive to general signals used by the right side of assignment statements in the function.

## @instance

The *@instance* decorator is used to specify a function where sensitivity to signals will be managed manually inside the function. Typically, this is performed by using the generator yield on a particular signal trigger. The @instance decorator is used in testbench programs for the test stimulus function.

# myHDL Documentation

There is extensive documentation for the myHDL package. The documentation may be found at:

<http://docs.myhdl.org/en/stable/manual/index.html>

Chapter

5

Virtual ATE

*This chapter describes the composition of the Virtual ATE.*

t

he P2654Simulations Project Virtual ATE provides the tester interface to the simulation environment. The Virtual ATE consists of a Telnet Server that interacts with the application interface, a display application, a command processing thread, and the simulator thread. The following diagram describes each of the modules.

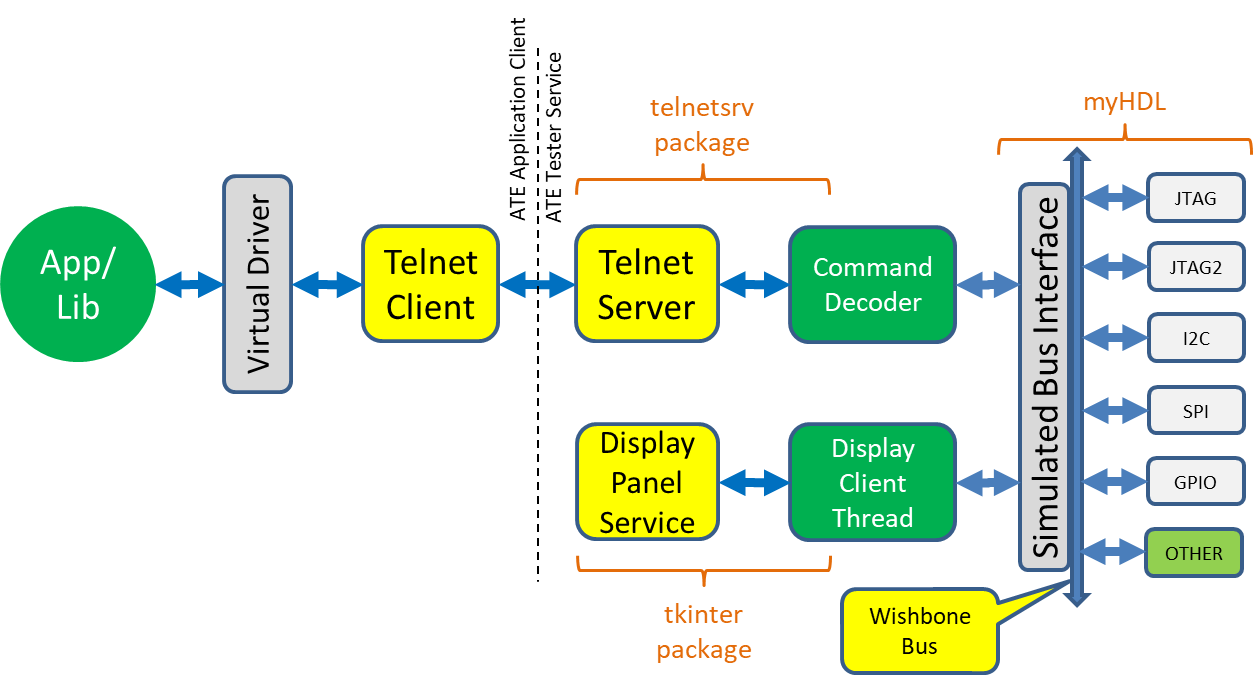


Figure 10 - ATE Architectural Modules

Application Library

The Application Library is implemented as a set of classes in Python and C++ containing high level functions to support the test bus interfaces. These high level functions communicate with the Telnet Client using functions available from the ATE class.

## JTAGController class

class JTAGController {  
public:  
 enum JTAGStates {  
 TEST\_LOGIC\_RESET=0,  
 RUN\_TEST\_IDLE,  
 SELECT\_DR,  
 CAPTURE\_DR,  
 SHIFT\_DR,  
 EXIT1\_DR,  
 PAUSE\_DR,  
 EXIT2\_DR,  
 UPDATE\_DR,  
 SELECT\_IR,  
 CAPTURE\_IR,  
 SHIFT\_IR,  
 EXIT1\_IR,  
 PAUSE\_IR,  
 EXIT2\_IR,  
 UPDATE\_IR  
 };  
 JTAGController(ATE& ate) : ate\_inst(ate) { };  
 ~JTAGController() { };  
 byte\_array ba\_scan\_ir(byte\_array& tdi\_vector, int count);  
 byte\_array ba\_scan\_dr(byte\_array& tdi\_vector, int count);  
 std::string scan\_ir(int count, std::string tdi\_string);  
 std::string scan\_dr(int count, std::string tdi\_string);  
 void runtest(int ticks);  
private:  
 void \_\_write\_vector\_segment(std::uint32\_t adr, byte data);  
 byte \_\_read\_vector\_segment(std::uint32\_t adr);  
 void \_\_set\_bit\_count(std::uint16\_t count);  
 void \_\_set\_state\_start(std::uint8\_t start);  
 void \_\_set\_state\_end(std::uint8\_t end);  
 void \_\_set\_control\_register(std::uint8\_t value);  
 std::uint8\_t \_\_get\_status\_register();  
 byte\_array \_\_scan\_vector(byte\_array& tdi\_vector, int count, std::uint8\_t start, std::uint8\_t end);  
 byte \_\_hex(char ch);  
 const char\* \_\_hex\_to\_char(byte data);  
 ATE& ate\_inst;  
 std::string tdo\_string;  
 byte\_array tdo\_vector;  
};

Figure 11 - JTAGController Listing

byte\_array ba\_scan\_ir(byte\_array& tdi\_vector, int count);

*ba\_scan\_ir( )* is the interface to send a vector to the Instruction Register (IR) scan chain of the JTAG bus. In C++, the *tdi\_vector* argument is a *byte\_array* where the vector right-most bit is the right-most bit of the first byte. Each additional byte contains the next eight bits in the vector. Unused bits are left padded with a zero. The *tdi\_vector* in Python follows the same rules, but is represented as a built-in Python *bytearray* type. The *count* argument is the number of bits in the vector. The maximum size of a vector may be 65,536 bits long per scan operation. The TDO vector response is returned from this function.

byte\_array ba\_scan\_dr(byte\_array& tdi\_vector, int count);

*ba\_scan\_dr( )* is the interface to send a vector to the Data Register (DR) scan chain of the JTAG bus. In C++, the *tdi\_vector* argument is a *byte\_array* where the vector right-most bit is the right-most bit of the first byte. Each additional byte contains the next eight bits in the vector. Unused bits are left padded with a zero. The *tdi\_vector* in Python follows the same rules, but is represented as a built-in Python *bytearray* type. The *count* argument is the number of bits in the vector. The maximum size of a vector may be 65,536 bits long per scan operation. The TDO vector response is returned from this function.

std::string scan\_ir(int count, std::string tdi\_string);

*scan\_ir( )* is the interface to send a vector to the Instruction Register (IR) scan chain of the JTAG bus. In C++, the *tdi\_string* argument is a *std::string* of hexadecimal digits where the vector right-most bit is the first bit to be scanned. Unused bits are left padded with a zero. The *tdi\_string* in Python follows the same rules, but is represented as a built-in Python *string* type. The *count* argument is the number of bits in the vector. The maximum size of a vector may be 65,536 bits long per scan operation. The TDO vector response is returned from this function as a hexadecimal string.

std::string scan\_dr(int count, std::string tdi\_string);

*scan\_dr( )* is the interface to send a vector to the Data Register (DR) scan chain of the JTAG bus. In C++, the *tdi\_string* argument is a *std::string* of hexadecimal digits where the vector right-most bit is the first bit to be scanned. Unused bits are left padded with a zero. The *tdi\_string* in Python follows the same rules, but is represented as a built-in Python *string* type. The *count* argument is the number of bits in the vector. The maximum size of a vector may be 65,536 bits long per scan operation. The TDO vector response is returned from this function as a hexadecimal string.

void runtest(int ticks);

runtest( ) is the interface to move to RUN\_TEST\_IDLE of the JTAG bus state and clock TCK for ticks counts while remaining in this state. The *ticks* argument is an *integer* with a maximum value of 65,536 ticks.

## JTAGController2 class

The second JTAG Controller class contains the same interfaces are available.

## I2CController class

class I2CController {  
public:  
 I2CController(ATE& ate) : ate\_inst(ate) { };  
 ~I2CController() { };  
 void i2c\_write\_reg(byte dev\_address, byte reg\_address, byte value);  
 byte i2c\_read\_reg(byte dev\_address, byte reg\_address);  
 bool i2c\_multibyte\_write(byte dev\_address, byte reg\_address, uint32\_t data);  
 uint32\_t i2c\_multibyte\_read(byte dev\_address, byte reg\_address);  
private:  
 void \_\_write\_transmit\_register(byte value);  
 byte \_\_read\_transmit\_register();  
 void \_\_write\_receive\_register(byte value);  
 byte \_\_read\_receive\_register();  
 void \_\_write\_control\_register(byte value);  
 byte \_\_read\_control\_register();  
 void \_\_write\_status\_register(byte value);  
 byte \_\_read\_status\_register();  
 ATE& ate\_inst;  
};

Figure 12 - I2CController Listing

void i2c\_write\_reg(byte dev\_address, byte reg\_address, byte value);

i2c\_write\_reg( ) is the interface to write a *byte* of data (*value*) to the I2C register in the device having the identification *dev\_address* and the register address of *reg\_address*.

byte i2c\_read\_reg(byte dev\_address, byte reg\_address);

i2c\_read\_reg( ) is the interface to read a *byte* of data from the I2C register in the device having the identification *dev\_address* and the register address of *reg\_address*.

bool i2c\_multibyte\_write(byte dev\_address, byte reg\_address, uint32\_t data);

i2c\_multibyte\_write( ) is the interface to write multiple byte of *data* to the I2C register in the device having the identification *dev\_address* and the register address of *reg\_address*. The data argument represents four bytes to be sent to the register using 4 merged write cycles. data[31:24] is sent first. data[23:16] is sent second. data[15:8] is sent third. data[7:0] is sent last.

uint32\_t i2c\_multibyte\_read(byte dev\_address, byte reg\_address);

i2c\_multibyte\_read( ) is the interface to read multiple byte of *data* from the I2C register in the device having the identification *dev\_address* and the register address of *reg\_address*. The return value represents four bytes to be read from the register using 4 merged read cycles. data[31:24] is read first. data[23:16] is read second. data[15:8] is read third. data[7:0] is read last.

## SPIController class

The SPIController embodies the ability to write out a 32-bit value while simultaneously capturing 32-bits scanned out of the SPI interface. The spi\_write( ) function initiates a write/read SPI cycle. The spi\_read( ) function is used to obtain the value read from the SPI bus that is stored in the SPIController.

class SPIController {  
public:  
 SPIController(ATE& ate) : ate\_inst(ate) { };  
 ~SPIController() { };  
 void spi\_write(uint32\_t value);  
 uint32\_t spi\_read();  
private:  
 void \_\_spi\_write\_transmit\_register(uint32\_t value);  
 uint32\_t \_\_spi\_read\_transmit\_register();  
 void \_\_spi\_write\_receive\_register(uint32\_t value);  
 uint32\_t \_\_spi\_read\_receive\_register();  
 ATE& ate\_inst;  
};

Figure 13 - SPIController Class Listing

void spi\_write(uint32\_t value);

spi\_write( ) is the interface to write a 32-bits of data (*value*) to the SPI register in the selected SPI device. The Controller implements a single SS signal to indicate a write or read cycle is occurring and expects a separate addressing module to handle multiple SPI devices.

uint32\_t spi\_read();

spi\_read( ) is the interface to read a 32-bits of data from the SPI register in the selected SPI device. The Controller implements a single SS signal to indicate a write or read cycle is occurring and expects a separate addressing module to handle multiple SPI devices.

## GPIO Access

The GPIO interface implements 16 bits of output and input signal buses. Access to the GPIO signals is directly to the Wishbone memory mapped register located at address 0x00001800. Thus, access is performed directly using the ATE class write( ) and read( ) methods. The read( ) method places the response in an internal variable. To obtain the value, the get\_value( ) method must be called after a read( ) call. If an error occurs during the write( ) or read( ) operations (indicated by an exception), the value of the error may be obtained using the get\_error( ) method of the ATE class.

# GPIO Test  
self.assertTrue(ate\_inst.write(0x00001800, 0x00000000))  
self.assertTrue(ate\_inst.read(0x00001800))  
self.assertTrue(ate\_inst.get\_value() == 0x00000000)  
self.assertTrue(ate\_inst.write(0x00001800, 0x00000015))  
self.assertTrue(ate\_inst.read(0x00001800))  
self.assertTrue(ate\_inst.get\_value() == 0x00150015)  
self.assertTrue(ate\_inst.write(0x00001800, 0x0000000A))  
self.assertTrue(ate\_inst.read(0x00001800))  
self.assertTrue(ate\_inst.get\_value() == 0x000A000A)  
self.assertTrue(ate\_inst.write(0x00001800, 0x00000000))  
self.assertTrue(ate\_inst.read(0x00001800))  
self.assertTrue(ate\_inst.get\_value() == 0x00000000)

Figure 14 - GPIO Access Listing example

To help programming, the GPIOController class may be used to perform the same operations that may be more readable. The GPIOController class implements the following.

class GPIOController {  
public:  
 GPIOController(ATE& ate) : ate\_inst(ate) { };  
 ~GPIOController() { };  
 bool write(std::uint32\_t val);  
 bool read( );  
 std::uint32\_t get\_value();  
 const char\* get\_error();  
};

Figure 15 - GPIOController Class Listing

# Initializing, Configuring, and Terminating the ATE in Testbenches

The ATE may be used as part of testbenches for the RTL code in myHDL directly without directly using the Telnet interface. Any testbench using the ATE interface directly is required to initialize the ATE tester prior to interacting with the Unit Under test (UUT). First, an instance of the ATE class must be created using the instance of the board to be tested as the argument. An example of this in Python is:

def P2654Board1\_tb():  
 gpio\_if = BoardGPIOInterface()  
 jtag\_if = BoardJTAGInterface()  
 board\_inst = P2654Board1("TOP", "P2654Board1")  
 board\_inst.configure\_jtag(jtag\_if)  
 board\_inst.configure\_gpio(gpio\_if)  
 ate\_inst = ATE(board\_inst)  
 ate\_inst.configure\_jtag(jtag\_if)  
 ate\_inst.configure\_gpio(gpio\_if)  
 ate\_inst.start\_simulation()  
 sleep(5)

# Test Bench code here ...

ate\_inst.terminate()

Figure 16 - Example Initialization of ATE with P2654Board1 instance as a Testbench

The first line creates an instance of the GPIO test bus to be shared between the ATE and the board (gpio\_if). The shared JTAG bus interface is instantiated next as jtag\_if. Next, the P2654Board1 instance is created passing in the parent name “TOP” and the instance name “P2654Board1” to be used in diagnostic messages from the simulator. Calls to link the gpio\_if and jtag\_if buses to the board instance are made next using the configure\_jtag( ) and configure\_gpio( ) methods of the board class. The ATE instance is created as ate\_inst by calling the constructor of the ATE class and passing in the instance of the board to be tested by the ATE. The test gpio\_if and jtag\_if test buses are bound to the ATE using the configure\_gpio( ) and configure\_jtag( ) methods of the ATE class. The simulator of the board and ATE RTL logic is started using the start\_simulation method of the ATE class. The testbench would then make calls to the ATE write( ), read( ), and get\_value( ) methods to perform the test. Upon ending the test, the testbench must call the ATE terminate( ) method to shut down the simulator.

# Initializing, Configuring, and Terminating the ATE in Applications

When using the ATE from applications, the *simservice* application must be running as the Telnet Server the application interfaces to. The board to be tested must be defined in the *simservice* program in order to be available to the application. This is defined in the *BoardFactory* inside BoardFactory.py.

class BoardFactory:  
 def \_\_init\_\_(self):  
 self.clk\_o = Signal(bool(0))  
 self.rst\_o = Signal(bool(0))  
 self.gpio\_if = BoardGPIOInterface()  
 self.i2c\_if = BoardI2CInterface()  
 self.spi\_if = BoardSPIInterface()  
 self.jtag\_if = BoardJTAGInterface()  
 self.jtag2\_if = BoardJTAGInterface()  
  
 def get\_gpio\_if(self):  
 return self.gpio\_if  
  
 def get\_i2c\_if(self):  
 return self.i2c\_if  
  
 def get\_spi\_if(self):  
 return self.spi\_if  
  
 def get\_jtag\_if(self):  
 return self.jtag\_if  
  
 def get\_jtag2\_if(self):  
 return self.jtag2\_if  
  
 def get\_clk\_o(self):  
 return self.clk\_o  
  
 def get\_rst\_o(self):  
 return self.rst\_o  
  
 def make\_board(self, board\_name):  
 board = None  
 if board\_name == "GPIOTest":  
 board = GPIOTest()  
 board.configure\_gpio(self.gpio\_if)  
 elif board\_name == "I2CTest":  
 board = I2CTest()  
 board.configure\_gpio(self.gpio\_if)  
 board.configure\_i2c(self.i2c\_if)  
 elif board\_name == "SPITest":  
 board = SPITest()  
 board.configure\_gpio(self.gpio\_if)  
 board.configure\_i2c(self.i2c\_if)  
 board.configure\_spi(self.spi\_if)  
 board.configure\_jtag(self.jtag\_if)  
 elif board\_name == "JTAGTest":  
 board = JTAGTest()  
 board.configure\_gpio(self.gpio\_if)  
 board.configure\_jtag(self.jtag\_if)  
 elif board\_name == "JTAG2Test":  
 board = JTAG2Test()  
 board.configure\_gpio(self.gpio\_if)  
 board.configure\_jtag2(self.jtag2\_if)  
 else:  
 board = None  
 if board is not None:  
 board.configure\_syscon(self.clk\_o, self.rst\_o)  
 return board

Figure 17 - Example make\_board from BoardFactory.py

Notice how the test bus interfaces from the ATE are configured specifically by calling the appropriate configuration method. To add the board P2654Board1, the *make\_board* method would be changed as:

class BoardFactory:  
 def \_\_init\_\_(self):  
 self.clk\_o = Signal(bool(0))  
 self.rst\_o = Signal(bool(0))  
 self.gpio\_if = BoardGPIOInterface()  
 self.i2c\_if = BoardI2CInterface()  
 self.spi\_if = BoardSPIInterface()  
 self.jtag\_if = BoardJTAGInterface()  
 self.jtag2\_if = BoardJTAGInterface()  
  
 def get\_gpio\_if(self):  
 return self.gpio\_if  
  
 def get\_i2c\_if(self):  
 return self.i2c\_if  
  
 def get\_spi\_if(self):  
 return self.spi\_if  
  
 def get\_jtag\_if(self):  
 return self.jtag\_if  
  
 def get\_jtag2\_if(self):  
 return self.jtag2\_if  
  
 def get\_clk\_o(self):  
 return self.clk\_o  
  
 def get\_rst\_o(self):  
 return self.rst\_o  
  
 def make\_board(self, board\_name):  
 board = None  
 if board\_name == "GPIOTest":  
 board = GPIOTest()  
 board.configure\_gpio(self.gpio\_if)  
 elif board\_name == "I2CTest":  
 board = I2CTest()  
 board.configure\_gpio(self.gpio\_if)  
 board.configure\_i2c(self.i2c\_if)  
 elif board\_name == "SPITest":  
 board = SPITest()  
 board.configure\_gpio(self.gpio\_if)  
 board.configure\_i2c(self.i2c\_if)  
 board.configure\_spi(self.spi\_if)  
 board.configure\_jtag(self.jtag\_if)  
 elif board\_name == "JTAGTest":  
 board = JTAGTest()  
 board.configure\_gpio(self.gpio\_if)  
 board.configure\_jtag(self.jtag\_if)  
 elif board\_name == "JTAG2Test":  
 board = JTAG2Test()  
 board.configure\_gpio(self.gpio\_if)  
 board.configure\_jtag2(self.jtag2\_if)  
 elif board\_name == "P2654Board1":  
 board = P2654Board1()  
 board.configure\_gpio(self.gpio\_if)  
 board.configure\_jtag(self.jtag\_if)  
 else:  
 board = None  
 if board is not None:  
 board.configure\_syscon(self.clk\_o, self.rst\_o)  
 return board

Figure 18 - Example of adding P2654Board1 to the simulator

Once a board is defined to the *simservice*, the application may link to the board to apply test patterns to it. The SPITest board listed for the following example has the GPIO test bus looped back as outputs to inputs, the I2C bus looped back, the JTAG bus looped back, and the SPI bus looped back. Drivers are provided to connect to the *simservice* as a series of controller classes. An application would link to the driver library or import the driver package to use.

The first thing any application must do is to connect to the *simservice* and bind to the board being tested as the UUT (SPITest in this example). This is done as follows:

ip = "127.0.0.1"  
port = 5023  
ate\_inst = ATE(ip=ip, port=port)  
sleep(0.05)  
self.assertTrue(ate\_inst.connect("SPITest"))  
sleep(0.05)

Figure 19 - Connection to simservice and binding to the UUT

Upon completion of the test code in the application, the ATE terminate( ) method must be called to stop the simulator. Finally, the ATE close( ) method must be called to break the connection to the *simservice*.

self.assertTrue(ate\_inst.terminate())  
ate\_inst.close()

Figure 20 - Ending a session by calling terminate( ) and close( )

An example Unit Test case of the GPIO bus is shown below:

def test\_simserviceATE002(self):  
 ip = "127.0.0.1"  
 port = 5023  
 ate\_inst = ATE(ip=ip, port=port)  
 sleep(0.05)  
 self.assertTrue(ate\_inst.connect("SPITest"))  
 sleep(0.05)  
 # GPIO Test  
 self.assertTrue(ate\_inst.write(0x00001800, 0x00000000))  
 self.assertTrue(ate\_inst.read(0x00001800))  
 self.assertTrue(ate\_inst.get\_value() == 0x00000000)  
 self.assertTrue(ate\_inst.write(0x00001800, 0x00000015))  
 self.assertTrue(ate\_inst.read(0x00001800))  
 self.assertTrue(ate\_inst.get\_value() == 0x00150015)  
 self.assertTrue(ate\_inst.write(0x00001800, 0x0000000A))  
 self.assertTrue(ate\_inst.read(0x00001800))  
 self.assertTrue(ate\_inst.get\_value() == 0x000A000A)  
 self.assertTrue(ate\_inst.write(0x00001800, 0x00000000))  
 self.assertTrue(ate\_inst.read(0x00001800))  
 self.assertTrue(ate\_inst.get\_value() == 0x00000000)  
 sleep(0.05)  
 self.assertTrue(ate\_inst.terminate())  
 ate\_inst.close()

Figure 21 - Unit Test case for the GPIO test bus

The lower 16-bits of the GPIO register represent the 16-bits of output. A read of the register returns the value of the lower bits showing what is written out. The upper 16-bits represent the value of the input ports.

To access the other test buses, the driver code provides specialized Controller classes. Each controller class is initialized with the instance of the ATE targeted for the test. This is done by passing the instance of the ATE to the constructor of each Controller class.

An example of a Unit Test case for the JTAG bus is shown below (Note: Indicator LEDs have been added to the SPITest board on the GPIO lines to provide visual feedback to where the test case is during execution.):

def test\_simserviceATE003(self):  
 ip = "127.0.0.1"  
 port = 5023  
 ate\_inst = ATE(ip=ip, port=port)  
 sleep(0.05)  
 self.assertTrue(ate\_inst.connect("SPITest"))  
 sleep(0.05)  
 # JTAG Test  
 jtag = JTAGController(ate\_inst)  
 self.assertTrue(ate\_inst.write(0x00001800, 0x00000001)) # Turn on WHITE LED to indicate scan start  
 tdo = jtag.scan\_ir(8, '55')  
 # print("tdo = ", tdo)  
 self.assertTrue(tdo == '55')  
 self.assertTrue(ate\_inst.write(0x00001800, 0x00000002)) # Turn on RED LED to indicate scan start  
 tdo = jtag.scan\_ir(12, '0A55')  
 self.assertTrue(tdo == 'A55')  
 self.assertTrue(ate\_inst.write(0x00001800, 0x00000004)) # Turn on GREEN LED to indicate scan start  
 tdo = jtag.scan\_ir(12, '5AA')  
 self.assertTrue(tdo == '5AA')  
 self.assertTrue(ate\_inst.write(0x00001800, 0x00000008)) # Turn on YELLOW LED to indicate scan start  
 tdo = jtag.scan\_dr(8, '55')  
 self.assertTrue(tdo == '55')  
 self.assertTrue(ate\_inst.write(0x00001800, 0x00000010)) # Turn on BLUE LED to indicate scan start  
 tdo = jtag.scan\_dr(12, 'AAA')  
 self.assertTrue(tdo == 'AAA')  
 self.assertTrue(ate\_inst.write(0x00001800, 0x00000011)) # Turn on BLUE & WHITE LEDs to indicate scan start  
 tdo = jtag.scan\_dr(12, 'A55')  
 self.assertTrue(tdo == 'A55')  
 self.assertTrue(ate\_inst.write(0x00001800, 0x00000012)) # Turn on BLUE & RED LEDs to indicate scan start  
 tdo = jtag.scan\_dr(12, '5AA')  
 self.assertTrue(tdo == '5AA')  
 self.assertTrue(ate\_inst.write(0x00001800, 0x00000014)) # Turn on BLUE & GREEN LEDs to indicate scan start  
 tdo = jtag.scan\_dr(16 \* 4, '0123456789ABCDEF')  
 self.assertTrue(tdo == '0123456789ABCDEF')  
 sleep(0.05)  
 self.assertTrue(ate\_inst.terminate())  
 ate\_inst.close()

Figure 22 - Example JTAG test case as a Unit Test

You will notice the use of the JTAGController class from the drivers package. This class provides high level context based utilities for communicating with the test bus. The first argument for scan\_ir( ) and scan\_dr( ) is the number of bits in the vector. The second argument is a hexadecimal string representing the TDI vector to be scanned into the UUT. The right most bit is scanned out first. The captured vector is returned as a hexadecimal string.

The JTAGController also contains methods to scan bytearray versions of the TDI vector. The returned TDO vector is also in the form of a bytearray. The formatting of the vector in the bytearray is:

* The first byte to be scanned out or captured is the byte with the index of zero.
* The first bit scanned out or captured is the right most bit of the zero index byte.
* Padding of unused bits in the last byte is a value of zero padding to the left.

An example of testing the I2C test bus is shown in the following example:

def test\_simserviceATE004(self):  
 ip = "127.0.0.1"  
 port = 5023  
 ate\_inst = ATE(ip=ip, port=port)  
 sleep(0.05)  
 self.assertTrue(ate\_inst.connect("SPITest"))  
 sleep(0.05)  
 # I2C Test  
 i2c = I2CController(ate\_inst)  
 # I2C Test set i2c master clock scale reg PRER = (48MHz / (5 \* 400KHz) ) - 1  
 i2c.i2c\_write\_reg(0x3C, 0x01, 0xA5)  
 self.assertTrue(i2c.i2c\_read\_reg(0x3C, 0x01) == 0xA5)  
  
 i2c.i2c\_multibyte\_write(0x3C, 0, 0x89abcdef)  
 self.assertTrue(i2c.i2c\_multibyte\_read(0x3C, 0) == 0x89abcdef)  
 self.assertTrue(i2c.i2c\_multibyte\_read(0x3C, 4) == 0x12345678)  
 sleep(0.05)  
 self.assertTrue(ate\_inst.terminate())  
 ate\_inst.close()

Figure 23 - Example I2C testcases as a Unit Test

The I2CController class provides the high level access utilities for the I2C test bus. The first argument for all interfaces is the address of the target device. The second argument is the target register address in the target device. For i2c\_write\_reg( ) third argument is the byte value to write to the target register. For the i2c\_multibyte\_write( ), the third argument is a 32-bit number representing the four bytes to be written. Depending on the I2C Client used, the addressing could be the same with multiple writes to the same register or it could be auto incrementing the register address for each write or read operation.

The access to the SPI bus is provided by the SPIController class. The SPIController assumes the SPI bus is 32-bits long. The first bit shifted out is the right most bit of the 32-bit value passed to the spi\_write( ) method. The spi\_read( ) method returns the value the SPIController captured during the last write cycle. The selection of which SPI Slave to connect to is handled outside of the SPIController. The SPI Host Controller drives the SS signal on the SPI interface, but selection gating must be added to this SS signal. The following example shows a Unit Test for the SPI test bus.

def test\_simserviceATE005(self):  
 ip = "127.0.0.1"  
 port = 5023  
 ate\_inst = ATE(ip=ip, port=port)  
 sleep(0.05)  
 self.assertTrue(ate\_inst.connect("SPITest"))  
 sleep(0.05)  
 # SPI Test  
 spi = SPIController(ate\_inst)  
 spi.spi\_write(0x01345678)  
 spi.spi\_write(0x00BADEDA)  
 self.assertTrue(spi.spi\_read() == 0x01345678)  
 spi.spi\_write(0x02BEEFED)  
 self.assertTrue(spi.spi\_read() == 0x00BADEDA)  
 spi.spi\_write(0x01345678)  
 self.assertTrue(spi.spi\_read() == 0x02BEEFED)  
 sleep(0.05)  
 self.assertTrue(ate\_inst.terminate())  
 ate\_inst.close()

Figure 24 - Example SPI testcases as a Unit Test

Chapter

6

Defining a Board Design

*This chapter describes the methodology for designing a board design using myHDL to configure the netlist interconnecting devices.*

t

he P2654Simulations Project board interface defines how the ATE test buses are connected to the rest of the circuits assembled on a circuit board. The board design file defines an Application Programming Interface (API) aiding the description of the connections.

Board Design Philosophy for Simulation

The P2654Simulations environment attempts to mimic a test system. The top interface provides access to the equivalent of the Automated Test Equipment (ATE) connections to the virtual system via standardized test buses. The virtual ATE contains two IEEE Std 1149.1 JTAG interfaces, an I2C interface, a SPI interface, and a 16 port input and 16 port output GPIO interface. These test buses are controlled with host controllers implemented with myHDL that are wired to a Wishbone bus. The Wishbone bus is controlled via messages from a Telnet Server thread to control the write and read cycles on the Wishbone bus. By stimulating the Wishbone Bus correctly, through the Telnet link, an application may send information through each of the ATE test buses to stimulate the unit under test (UUT). The ATE test buses are the only connections supported between the application software and the UUT.

The myHDL designs for boards and devices are implemented as hierarchical object oriented classes that allow for common attributes, like test bus interface, to be abstracted into a common base class. New board designs may be implemented leveraging the common attributes from the base class. The base class for a board is the AbstractBoard class.

## AbstractBoard Class

The AbstractBoard class defines the test bus signal that mate to the ATE tester signals. This class also provides methods the ATE configuration routine calls to define the specific instances of myHDL Signals implementing the state values of these interfaces in the simulator. These signals are defined and initialized by the ATE instance of the simulator.

# Interface Classes

There are a series ofinterface classes implementing new instances of the test bus signals provided by the ATE environment that define the signals as a collection of signals in a single wrapper interface class.

### BoardGPIOInterface class

This class defines the 16-bit input and 16-bit output signals making up the GPIO test bus for the ATE to Board connection.

class BoardGPIOInterface:  
 def \_\_init\_\_(self):  
 self.i\_gpio = Signal(intbv(0)[16:])  
 self.o\_gpio = Signal(intbv(0)[16:])

Figure 25 - BoardGPIOInterface class

## BoardJTAGInterface class

The BoardJTAGInterface defines the signals to be used between the ATE and board JTAG interfaces. It consists of the five TAP signals defined by IEEE Std 1149.1.

class BoardJTAGInterface:  
 def \_\_init\_\_(self):  
 self.TCK = Signal(bool(0))  
 self.TMS = Signal(bool(1))  
 self.TRST = Signal(bool(1))  
 self.TDO = Signal(bool(1))  
 self.TDI = Signal(bool(0))

Figure 26 - BoardJTAGInterface class

## BoardI2CInterface class

To connect the I2C test bus, the BoardI2CInterface class is used to define and wrap the bus signals. Since myHDL does not contain a construct to support an open collector bus, the I2C bus signals are implemented as a triplet of signals. Each primary signal is represented by individual output, input, and enable signals that may be used to wire up to external open collector interfaces. For the simulation, the use of these triplets is equivalent and sufficient for testing. myHDL has implemented a Tristate Signal in the past couple releases, but it does not fully represent the attributes of an open collector implementation.

class BoardI2CInterface:  
 def \_\_init\_\_(self):  
 self.SCL\_O = Signal(bool(1))  
 self.SCL\_I = Signal(bool(1))  
 self.SCL\_E = Signal(bool(1))  
 self.SDA\_O = Signal(bool(1))  
 self.SDA\_I = Signal(bool(1))  
 self.SDA\_E = Signal(bool(1))

Figure 27 - BoardI2CInterface class

## BoardSPIInterface class

The BoardSPIInterface class instantiates the signals used by the SPI test bus. This interface may be used directly to connect to a single client as only one SS signal is supported. If other SPI devices are needing to be controlled, external logic must be used to gate the SS signal to each device.

class BoardSPIInterface:  
 def \_\_init\_\_(self):  
 self.SCLK = Signal(bool(0))  
 self.MOSI = Signal(bool(0))  
 self.MISO = Signal(bool(0))  
 self.SS = Signal(bool(0))

Figure 28 - BoardSPIInterface class

# Specialized Board Designs

##### P2654Board1 Working Example

## Schematic Diagram

The schematic for P2654Board1 is shown below. It consists of a GPIOInterface via connector J1. The JTAG TAP interface is represented by connector J2. The power is provided by connector J3. The main logic for the board is U1, a SN74ABT8244A buffer device. U1 buffers the GPIO signals to the LEDs. To limit the current flow from U1 to the LEDs, the resistor network RN1 is wired between U1 and the LEDs. The lowest bits of the O\_GPIO bus are wired to the A side inputs of U1. The B side outputs of U1 are wired to the lowest bits of the I\_GPIO bus and RN1. The other side of RN1 is wired to each of the LEDs, D1 – D8.

The LEDs may be lit by driving the O\_GPIO signals or using the JTAG test bus and control the pins of U1 using the EXTEST instruction and the Boundary Scan Register of U1 to drive the B side output pins from the scan chain. The I\_GPIO bus is able to monitor the signals being sent to the LEDs to be able to verify automatically the correct LED is being lit.

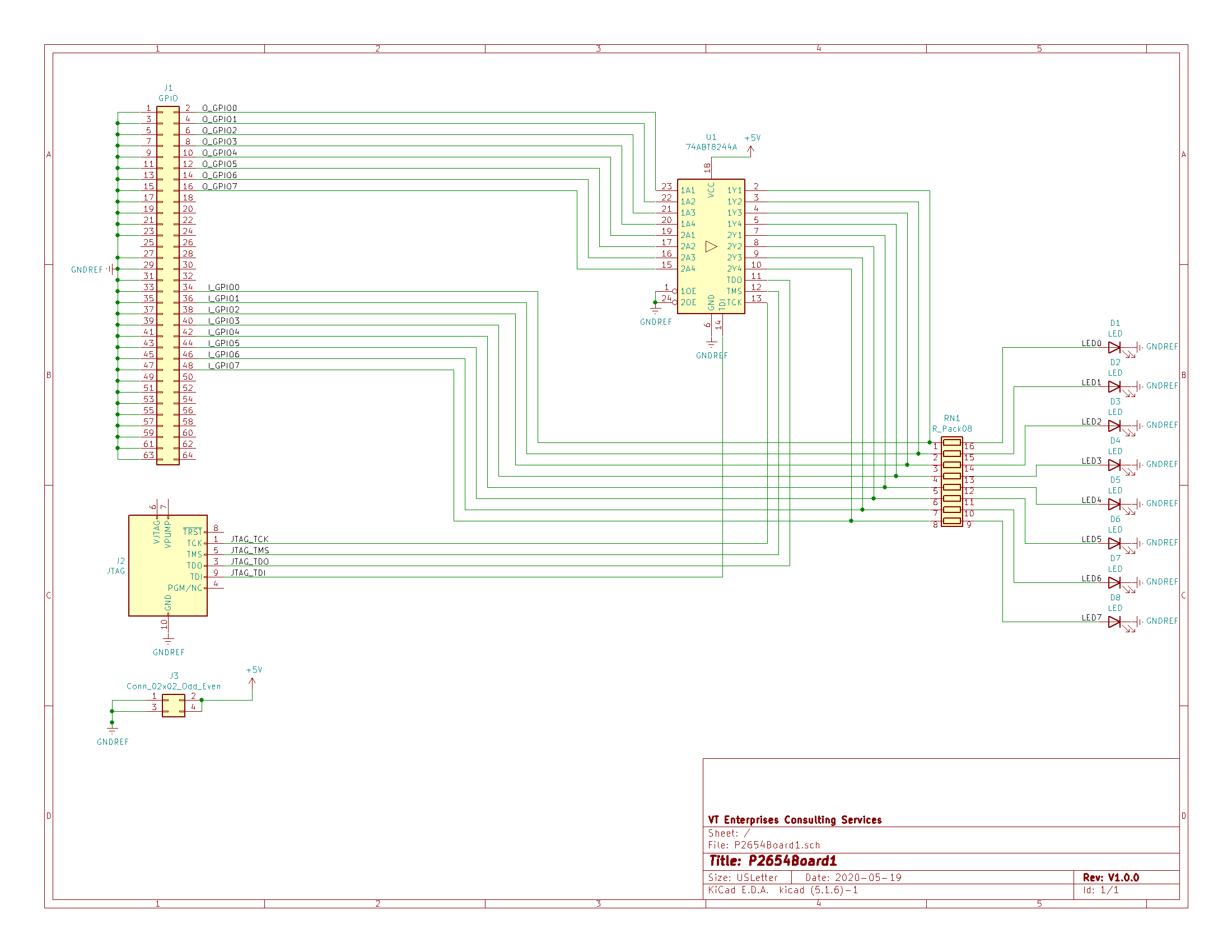


Figure 29 - Schmatic of P2654Board1 Example Specialized Board

## RTL Representation of Schematic

The schematic representation needs to be converted into an RTL equivalent circuit using myHDL to be able to be used by the simulator. A specialized class of the AbstractBoard base class is used to define the board representation. The P2654Board1 class is shown below. The file imports the definition of the functional components assembled on the board: N74ABT8244A and PseudoLED. A specialized board class is created for the board design as P2654Board1. This specialized class is derived from the AbstractBoard class to leverage the test bus interfaces defined by the AbstractBoard class. The parent and name members are used in diagnostic messages to better understand from what code a message is sent from. Next placeholders for the instances of the devices assembled on the board are created.

The method configure\_syscon( ) is used by the ATE to bind a reference to the Wishbone clock and reset signals. These signals are required by some board designs `

from myhdl import \*  
from hdl.boards.common.AbstractBoard import AbstractBoard  
from hdl.devices.SN74ABT8244A.SN74ABT8244A import SN74ABT8244A  
from hdl.instruments.PseudoLED.PseudoLED import PseudoLED  
  
  
class P2654Board1(AbstractBoard):  
 def \_\_init\_\_(self, parent, name):  
 super().\_\_init\_\_()  
 self.parent = parent  
 self.name = name  
  
 self.sn74abt8244\_device = None  
 self.led0\_inst = None  
 self.led1\_inst = None  
 self.led2\_inst = None  
 self.led3\_inst = None  
 self.led4\_inst = None  
 self.led5\_inst = None  
 self.led6\_inst = None  
 self.led7\_inst = None  
 self.syscon = None  
  
 def configure\_syscon(self, clk\_o, rst\_o):  
 self.clk\_o = clk\_o  
 self.rst\_o = rst\_o  
  
 @block  
 def rtl(self):  
 width = 4  
 Y1 = [TristateSignal(False) for \_ in range(width)]  
 Y2 = [TristateSignal(False) for \_ in range(width)]  
 A1 = [Signal(bool(0)) for \_ in range(width)]  
 A2 = [Signal(bool(0)) for \_ in range(width)]  
 oe\_neg1 = Signal(bool(0))  
 oe\_neg2 = Signal(bool(0))  
 tdo\_padoe\_o = Signal(bool(0))  
 led0 = Signal(bool(0))  
 led1 = Signal(bool(0))  
 led2 = Signal(bool(0))  
 led3 = Signal(bool(0))  
 led4 = Signal(bool(0))  
 led5 = Signal(bool(0))  
 led6 = Signal(bool(0))  
 led7 = Signal(bool(0))  
  
 self.sn74abt8244\_device = SN74ABT8244A("TOP", "SN74ABT8245", oe\_neg1, Y1, Y2, A1, A2, oe\_neg2,  
 tdo\_padoe\_o, self.tdi, self.tck, self.tms, self.tdo)  
 self.led0\_inst = PseudoLED("TOP", "LED0", led0, color="RED")  
 self.led1\_inst = PseudoLED("TOP", "LED1", led1, color="GREEN")  
 self.led2\_inst = PseudoLED("TOP", "LED2", led2, color="YELLOW")  
 self.led3\_inst = PseudoLED("TOP", "LED3", led3, color="ORANGE")  
 self.led4\_inst = PseudoLED("TOP", "LED4", led4, color="BLUE")  
 self.led5\_inst = PseudoLED("TOP", "LED5", led5, color="VIOLET")  
 self.led6\_inst = PseudoLED("TOP", "LED6", led6, color="INDIGO")  
 self.led7\_inst = PseudoLED("TOP", "LED7", led7, color="WHITE")  
  
 self.sn74abt8244\_device.configure\_jtag(self.tdi, self.tck, self.tms, self.trst, self.tdo)  
  
 # build up the netlist for the board here  
 @always\_comb  
 def netlist():  
 # Wire the LED to the buffer  
 if Y1[0].val is None:  
 led0.next = False  
 self.i\_gpio.next[0] = False  
 else:  
 led0.next = Y1[0]  
 self.i\_gpio.next[0] = Y1[0]  
 if Y1[1].val is None:  
 led1.next = False  
 self.i\_gpio.next[1] = False  
 else:  
 led1.next = Y1[1]  
 self.i\_gpio.next[1] = Y1[1]  
 if Y1[2].val is None:  
 led2.next = False  
 self.i\_gpio.next[2] = False  
 else:  
 led2.next = Y1[2]  
 self.i\_gpio.next[2] = Y1[2]  
 if Y1[3].val is None:  
 led3.next = False  
 self.i\_gpio.next[3] = False  
 else:  
 led3.next = Y1[3]  
 self.i\_gpio.next[3] = Y1[3]  
 if Y2[0].val is None:  
 led4.next = False  
 self.i\_gpio.next[4] = False  
 else:  
 led4.next = Y2[0]  
 self.i\_gpio.next[4] = Y2[0]  
 if Y2[1].val is None:  
 led5.next = False  
 self.i\_gpio.next[5] = False  
 else:  
 led5.next = Y2[1]  
 self.i\_gpio.next[5] = Y2[1]  
 if Y2[2].val is None:  
 led6.next = False  
 self.i\_gpio.next[6] = False  
 else:  
 led6.next = Y2[2]  
 self.i\_gpio.next[6] = Y2[2]  
 if Y2[3].val is None:  
 led7.next = False  
 self.i\_gpio.next[7] = False  
 else:  
 led7.next = Y2[3]  
 self.i\_gpio.next[7] = Y2[3]  
 A1[0].next = self.o\_gpio[0]  
 A1[1].next = self.o\_gpio[1]  
 A1[2].next = self.o\_gpio[2]  
 A1[3].next = self.o\_gpio[3]  
 A2[0].next = self.o\_gpio[4]  
 A2[1].next = self.o\_gpio[5]  
 A2[2].next = self.o\_gpio[6]  
 A2[3].next = self.o\_gpio[7]  
  
 return netlist, \  
 self.led0\_inst.rtl(), self.led1\_inst.rtl(), self.led2\_inst.rtl(), \  
 self.led3\_inst.rtl(), self.led4\_inst.rtl(), self.led5\_inst.rtl(), \  
 self.led6\_inst.rtl(), self.led7\_inst.rtl(), \  
 self.sn74abt8244\_device.rtl()

Figure 30 - Specialized board class for P2654Board1 circuit

The rtl( ) method is wrapped with an @block decorator because this method implements the logic the simulator will use to simulate the board function. Individual logic processes are defined as child functions of rtl( ). These child processes are wrapped in other myHDL decorators to define the sensitivity of signals each function describes.

The first block of code in rtl( ) defines the nets of the P2654Board1 circuit board. These nets have not yet been bound to the devices they are wired to, but initialized as myHDL Signals so the simulator may change their state over time.

Next, the instance of U1, the SN74ABT8244A, as sn74abt8244\_device. The arguments passed to the constructor are the nets (Signals) wired to the board by the device. This binds the RTL of the board to the RTL of the device.

Then, eight instances of the LEDs is instantiated using the led signals/nets defined between RN1 and the LED. Since RN1 is a passive device, it may be bypassed in the behavioral RTL description of the circuit. Thus, the LED is wired directly to the sn74abt8244\_device instance.

Just because the signals are bound to the devices assembled on the board, the behavior of these signals still needs to be defined. That is handled by the behavioral processes defined as child functions of the rtl( ) function. Before the behavior is defined, the configure\_jtag( ) method of the sn74abt8244\_device instance is called to bind the ATE JTAG TAP signals bound to the board with the device instance. If there was more than one Boundary Scan device, there would need to be additional TDI/TDO signals defined for daisy chaining the serial data path between the devices and the end devices would be bound to the BoardJTAGInterface TDI and TDO Signals.

For this board, only a combinatorial process is necessary to define the behavior of the nets on the board. The sn74abt8244 contains tristate outputs on its A and B ports. Thus, that has to be accounted for in the description. The child function, netlist( ), defines the netlist behavior. If an output is tristated, myHDL represents that as a value of None for the Signal. The LED is defined to be off by setting its next value to False. At the same time, the I\_GPIO signal needs to be updated by setting the next value to False. If the output is not tristated, it is either True or False. Thus, set the next value of the LED and I\_GPIO Signal to the value of the output Signal of the sn74abt8244\_device.

Lastly, the input to the sn74abt8244\_device has to be stimulated. To accomplish that, the values from the O\_GPIO connector are assigned to the next value of each of the buffer inputs.

The rtl( ) method return all of the instances of the generators created for myHDL defined by the method. This consists of the netlist function (because it is @always\_comb) and the rtl( ) methods of the device instances created in the rtl( ) method.

Chapter

7

Defining a Device Design

*This chapter describes the methodology for designing a device design using myHDL to configure the netlist interconnecting the ATE interface to a device for testing.*

t

he P2654Simulations Project device interface defines how the ATE test buses are connected to the device under test. The device design file defines an Application Programming Interface (API) aiding the description of the connections.

Device Design Philosophy for Simulation

The P2654Simulations environment attempts to mimic a test system. The top interface provides access to the equivalent of the Automated Test Equipment (ATE) connections to the virtual system via standardized test buses. The virtual ATE contains two IEEE Std 1149.1 JTAG interfaces, an I2C interface, a SPI interface, and a 16 port input and 16 port output GPIO interface. These test buses are controlled with host controllers implemented with myHDL that are wired to a Wishbone bus. The Wishbone bus is controlled via messages from a Telnet Server thread to control the write and read cycles on the Wishbone bus. By stimulating the Wishbone Bus correctly, through the Telnet link, an application may send information through each of the ATE test buses to stimulate the unit under test (UUT). The ATE test buses are the only connections supported between the application software and the UUT.

The myHDL designs for devices are implemented as myHDL block functions or object oriented classes.

# Specialized Device Designs

##### SN74ABT8244A Working Example

## RTL Design Test Bench

A test bench is a test case for the RTL design. This can be a separate file or implemented as an embedded function as part of the RTL design file. Typically, the test bench is implemented as an included function that is called as the main( ) function executed if the design file is executed stand-alone. An example of such a test bench is shown below.

@block  
def SN74ABT8244A\_tb(monitor=False):  
 *"""  
 Test bench interface for a quick test of the operation of the design* ***:param*** *monitor: False=Do not turn on the signal monitors, True=Turn on the signal monitors* ***:return****: A list of generators for this logic  
 """* width = 4  
 Y1 = [TristateSignal(False) for \_ in range(width)]  
 Y2 = [TristateSignal(False) for \_ in range(width)]  
 A1 = [Signal(bool(0)) for \_ in range(width)]  
 A2 = [Signal(bool(0)) for \_ in range(width)]  
 tdi = Signal(bool(0))  
 tdo = Signal(bool(0))  
 tms = Signal(bool(0))  
 tck = Signal(bool(0))  
 trst = Signal(bool(0))  
 oe\_neg1 = Signal(bool(1))  
 oe\_neg2 = Signal(bool(1))  
 tdo\_padoe\_o = Signal(bool(0))  
  
 inst = SN74ABT8244A("TOP", "SN74ABT8244", oe\_neg1, Y1, Y2, A1, A2, oe\_neg2,  
 tdo\_padoe\_o, tdi, tck, tms, tdo)  
  
 @instance  
 def clkgen():  
 while True:  
 tck.next = not tck  
 yield delay(period // 2)  
  
 @instance  
 def stimulus():  
 *"""* ***:return****:  
 """* disabled = [TristateSignal(None) for \_ in range(4)]  
 A1[0].next = True  
 for i in range(1, 4):  
 A1[i].next = False  
 for i in range(4):  
 A2[i].next = False  
 A2[2].next = True  
 oe\_neg1.next = False  
 oe\_neg2.next = False  
 yield delay(1)  
 print("Y1 = ", Y1)  
 print("A1 = ", A1)  
 assert (Y1 == A1)  
 print("Y2 = ", Y2)  
 print("A2 = ", A2)  
 assert (Y2 == A2)  
 A1[0].next = False  
 A1[1].next = True  
 A2[2].next = False  
 A2[3].next = True  
 yield delay(1)  
 assert (Y1 == A1)  
 assert (Y2 == A2)  
 oe\_neg1.next = True  
 oe\_neg2.next = True  
 yield delay(1)  
 print("Y1 = ", Y1)  
 print("disabled = ", disabled)  
 assert (Y1 == disabled)  
 assert (Y2 == disabled)  
  
 raise StopSimulation()  
  
 return inst.rtl(), clkgen, stimulus

Figure 31 – Integrated Device Test Bench Example

The test bench is called from code at the end of the SN74ABT8244A.py design RTL file. The code is shown below.

def main():  
 tb = SN74ABT8244A\_tb(monitor=True)  
 tb.config\_sim(trace=True)  
 tb.run\_sim()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

Figure 32 - Code to instrument the calling of the Test Bench

## RTL Design ATE Adapter Board

To stitch a device to the ATE Interfaces, a board design containing the single device UUT needs to be created. This is similar to what would need to be implemented for testing a device on a real ATE. This adapter board is used to provide the glue necessary for defining the behavior of the signals used by the device and the ATE. The implementation will follow the process shown in the previous chapter. An example of this is shown below.

class AdapterBoard8244(AbstractBoard):  
 def \_\_init\_\_(self, parent, name):  
 super().\_\_init\_\_()  
 self.parent = parent  
 self.name = name  
  
 self.sn74abt8244\_device = None  
  
 def configure\_syscon(self, clk\_o, rst\_o):  
 self.clk\_o = clk\_o  
 self.rst\_o = rst\_o  
  
  
 @block  
 def rtl(self):  
 width = 4  
 Y1 = [TristateSignal(False) for \_ in range(width)]  
 Y2 = [TristateSignal(False) for \_ in range(width)]  
 A1 = [Signal(bool(0)) for \_ in range(width)]  
 A2 = [Signal(bool(0)) for \_ in range(width)]  
 oe\_neg1 = Signal(bool(0))  
 oe\_neg2 = Signal(bool(0))  
 tdo\_padoe\_o = Signal(bool(0))  
  
 self.sn74abt8244\_device = SN74ABT8244A("TOP", "SN74ABT8245", oe\_neg1, Y1, Y2, A1, A2, oe\_neg2,  
 tdo\_padoe\_o, self.tdi, self.tck, self.tms, self.tdo)  
  
 self.sn74abt8244\_device.configure\_jtag(self.tdi, self.tck, self.tms, self.trst, self.tdo)  
  
 # build up the netlist for the board here  
 @always\_comb  
 def netlist():  
 # Wire the LED to the buffer  
 if Y1[0].val is None:  
 self.i\_gpio.next[0] = False  
 else:  
 self.i\_gpio.next[0] = Y1[0]  
 if Y1[1].val is None:  
 self.i\_gpio.next[1] = False  
 else:  
 self.i\_gpio.next[1] = Y1[1]  
 if Y1[2].val is None:  
 self.i\_gpio.next[2] = False  
 else:  
 self.i\_gpio.next[2] = Y1[2]  
 if Y1[3].val is None:  
 self.i\_gpio.next[3] = False  
 else:  
 self.i\_gpio.next[3] = Y1[3]  
 if Y2[0].val is None:  
 self.i\_gpio.next[4] = False  
 else:  
 self.i\_gpio.next[4] = Y2[0]  
 if Y2[1].val is None:  
 self.i\_gpio.next[5] = False  
 else:  
 self.i\_gpio.next[5] = Y2[1]  
 if Y2[2].val is None:  
 self.i\_gpio.next[6] = False  
 else:  
 self.i\_gpio.next[6] = Y2[2]  
 if Y2[3].val is None:  
 self.i\_gpio.next[7] = False  
 else:  
 self.i\_gpio.next[7] = Y2[3]  
 A1[0].next = self.o\_gpio[0]  
 A1[1].next = self.o\_gpio[1]  
 A1[2].next = self.o\_gpio[2]  
 A1[3].next = self.o\_gpio[3]  
 A2[0].next = self.o\_gpio[4]  
 A2[1].next = self.o\_gpio[5]  
 A2[2].next = self.o\_gpio[6]  
 A2[3].next = self.o\_gpio[7]  
  
 return netlist, \  
 self.sn74abt8244\_device.rtl()

Figure 33 - Adapter board for testing SN74ABT8244A RTS Design

Chapter

8

Defining IP Cores and Instruments

*This chapter describes the essential process for defining IP Cores and Instruments using myHDL RTL.*

t

he P2654Simulations Project uses the Python package myHDL to define the RTL logic for all cores and instruments. A detailed manual for myHDL may be found at:

<http://docs.myhdl.org/en/stable/>

IP Core and Instrument Design Philosophy for Simulation

IP Cores and Instruments follow the same design philosophy as that for devices. Each IP Core and Instrument is treated as a design module that gets instantiated as a child instance to the parent module. There is no limit to the depth of nesting of modules that may be supported by myHDL.

# Specialized IP Core Designs

##### Rearick IP\_2 Core Example

The Rearick IP\_2 IP Core includes access to two instruments: PowerSupplyMonitor and NoiseMaker. Each instrument is wrapped behind an IEEE Std 1687 Segment Insertion Bit (SIB) to allow the bypassing of an instrument during testing. The 1687 I/F represents the input and output port Signals for the IP\_2 Core. This interface represents an IEEE Std 1687 Client interface to the rest of the circuit in the device.

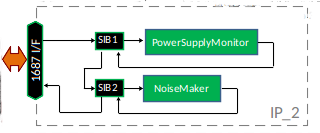


Figure 34 - Pictoral diagram of IP\_2 Core logic

RTL Design Test Bench

A test bench is a test case for the RTL design. This can be a separate file or implemented as an embedded function as part of the RTL design file. Typically, the test bench is implemented as an included function that is called as the main( ) function executed if the design file is executed stand-alone. An example of such a test bench is shown below. At first, the signals used to integrate between the instruments and client interface are defined. Next, a set of test vectors is created to be used as test stimulus. Following the vectors is code to set up a file to record the state of key signals in CSV format at each clock tick for diagnostics. The child function, print\_data( ), is used to print out the diagnostic state data. The fast\_ckgen( ) function is used to create a fast reference clock as a sampling clock for when to sample the voltage value of VDD. ckgen( ) is used to create a normal speed reference clock signal to define the window to apply the noise in. ijtagckgen( ) is used to create the scan clock for the IEEE Std 1687 network. Finally, the stimulus( ) function is used to implement the actual test plan. Since it is wrapped in a myHDL @instance decorator, the yield statement may be used to trigger a wait event for a specific state event change to take place that is implemented as a Python generator method. When the test is complete, the StopSimulation( ) exception is raised. The test bench returns the functions and module instantiations created inside this function as children of the module.

@block  
def IP\_2\_tb(monitor=False):  
 *"""  
 Test bench interface for a quick test of the operation of the design* ***:param*** *monitor: False=Do not turn on the signal monitors, True=Turn on the signal monitors* ***:return****: A list of generators for this logic  
 """* H = bool(1)  
 L = bool(0)  
 ijtag\_si = Signal(L)  
 ijtag\_so = Signal(L)  
 from\_ijtag\_interface = IJTAGInterface()  
 fast\_ck = Signal(L)  
 ck = Signal(L)  
 pu\_mbist1 = Signal(intbv(0, min=0, max=101))  
 pu\_mbist2 = Signal(intbv(0, min=0, max=101))  
 pu\_mbist3 = Signal(intbv(0, min=0, max=101))  
 pu\_mbist4 = Signal(intbv(0, min=0, max=101))  
 pu\_mbist5 = Signal(intbv(0, min=0, max=101))  
  
 sib1\_0 = '0'  
 sib1\_1 = '1'  
 sib2\_0 = '0'  
 sib2\_1 = '1'  
 reg0 = '0000000000000000'  
 delta0 = '00000000'  
 delta1 = '10110010'  
 nf0 = '0'  
 stat0 = '00'  
 width0 = len(sib1\_0) + len(sib2\_0)  
 width = len(sib1\_0) + len(reg0) + len(delta0) + len(nf0) + len(stat0) + len(sib2\_0)  
 sibs00 = Signal(intbv(sib1\_0 + sib2\_0)[2:])  
 sibs10 = Signal(intbv(sib1\_1 + sib2\_0)[2:])  
 si1 = Signal(intbv(sib1\_1 + reg0 + delta0 + nf0 + stat0 + sib2\_0)[width:])  
 so1 = Signal(intbv(sib1\_1 + reg0 + delta0 + nf0 + stat0 + sib2\_0)[width:])  
 si2 = Signal(intbv(sib1\_1 + reg0 + delta1 + nf0 + stat0 + sib2\_0)[width:])  
 so2 = Signal(intbv(sib1\_1 + reg0 + delta0 + nf0 + stat0 + sib2\_0)[width:])  
 si3 = Signal(intbv(sib1\_1 + reg0 + delta0 + nf0 + stat0 + sib2\_0)[width:])  
 so3 = Signal(intbv(sib1\_1 + reg0 + delta1 + nf0 + stat0 + sib2\_0)[width:])  
  
 ip2\_inst = IP\_2("TOP", "IP[2]", ijtag\_si, ijtag\_so, from\_ijtag\_interface,  
 fast\_ck, ck,  
 pu\_mbist1, pu\_mbist2, pu\_mbist3, pu\_mbist4, pu\_mbist5,  
 monitor=monitor)  
  
 # print simulation data to file  
 file\_data = open("IP\_2.csv", 'w') # file for saving data  
 # print header to file  
 print("{0},{1},{2},{3},{4},{5}".format("si", "ce", "se", "ue", "sel", "so"),  
 file=file\_data)  
 # print data on each tap\_interface.ClockDR  
 @always(from\_ijtag\_interface.CLOCK.posedge)  
 def print\_data():  
 *"""  
 """* # print in file  
 # print.format is not supported in MyHDL 1.0  
 print(ijtag\_si, ",", from\_ijtag\_interface.CAPTURE, ",", from\_ijtag\_interface.SHIFT, ",",  
 from\_ijtag\_interface.UPDATE, ",", from\_ijtag\_interface.SELECT, ",", ijtag\_so, file=file\_data)  
  
 @instance  
 def fast\_ckgen():  
 while True:  
 fast\_ck.next = not fast\_ck  
 yield delay(period // 10)  
  
 @instance  
 def ckgen():  
 while True:  
 ck.next = not ck  
 yield delay(period)  
  
 @instance  
 def ijtagckgen():  
 while True:  
 from\_ijtag\_interface.CLOCK.next = not from\_ijtag\_interface.CLOCK  
 yield delay(period // 2)  
  
 @instance  
 def stimulus():  
 *"""  
 Perform instruction decoding for various instructions* ***:return****:  
 """* # Reset the network  
 from\_ijtag\_interface.RESET.next = H  
 yield delay(1)  
 from\_ijtag\_interface.RESET.next = L  
 yield delay(1)  
  
 # Shift deselect pattern through SIB1 and SIB2 to ensure network is in bypass state  
 # Start the Capture transition operation  
 # First C, S(00), U is so == bin(00)?  
 print("First C, S(00), U is so == bin(00)?")  
 yield from\_ijtag\_interface.CLOCK.posedge  
 yield from\_ijtag\_interface.CLOCK.negedge  
 # Write Capture value  
 from\_ijtag\_interface.CAPTURE.next = H  
 from\_ijtag\_interface.SELECT.next = H  
 yield from\_ijtag\_interface.CLOCK.posedge  
 yield from\_ijtag\_interface.CLOCK.negedge  
 for i in range(width0):  
 # Write Shift value  
 from\_ijtag\_interface.CAPTURE.next = L  
 from\_ijtag\_interface.SHIFT.next = H  
 print("ijtag\_si.next = sibs00[i], sibs00[", i, "] = ", sibs00[i])  
 ijtag\_si.next = sibs00[i] # ########################################################### SHIFT  
 print("sibs00[", i, "] = ", sibs00[i], ", ijtag\_so = ", ijtag\_so)  
 yield from\_ijtag\_interface.CLOCK.posedge  
 assert(ijtag\_so == sibs00[i])  
 yield from\_ijtag\_interface.CLOCK.negedge  
 # Update  
 from\_ijtag\_interface.SHIFT.next = L  
 from\_ijtag\_interface.UPDATE.next = H  
  
 # Select SIB1 to open secondary network  
 # Second C, S(10), U is so == bin(00)? SIB1 should now be enabled  
 print("Select SIB1 network. Second C, S(10), U is so == bin(00)? SIB1 should now be enabled")  
 yield from\_ijtag\_interface.CLOCK.posedge  
 yield from\_ijtag\_interface.CLOCK.negedge  
 # Write Capture value  
 from\_ijtag\_interface.UPDATE.next = L  
 from\_ijtag\_interface.CAPTURE.next = H  
 yield from\_ijtag\_interface.CLOCK.posedge  
 yield from\_ijtag\_interface.CLOCK.negedge  
 for i in range(width0):  
 # Write Shift value  
 from\_ijtag\_interface.CAPTURE.next = L  
 from\_ijtag\_interface.SHIFT.next = H  
 print("ijtag\_si.next = sibs10[i], sibs10[", i, "] = ", sibs10[i])  
 ijtag\_si.next = sibs10[i] # ########################################################### SHIFT  
 print("sibs00[", i, "] = ", sibs00[i], ", ijtag\_so = ", ijtag\_so)  
 yield from\_ijtag\_interface.CLOCK.posedge  
 assert(ijtag\_so == sibs00[i])  
 yield from\_ijtag\_interface.CLOCK.negedge  
 # Update  
 from\_ijtag\_interface.SHIFT.next = L  
 from\_ijtag\_interface.UPDATE.next = H  
  
 # Scan safe data through secondary network to verify the network is working  
 # Third C, S(10), U is so == bin(00)? 16:8:1:2 bits required from SI to SO for reference:delta:nf:status.  
 print("Scan safe data through SIB1 network. Third C, S(10), U is so == bin(00)?")  
 yield from\_ijtag\_interface.CLOCK.posedge  
 yield from\_ijtag\_interface.CLOCK.negedge  
 # Write Capture value  
 from\_ijtag\_interface.UPDATE.next = L  
 from\_ijtag\_interface.CAPTURE.next = H  
 yield from\_ijtag\_interface.CLOCK.posedge  
 yield from\_ijtag\_interface.CLOCK.negedge  
 print("width = ", width)  
 for i in range(width):  
 # Write Shift value  
 from\_ijtag\_interface.CAPTURE.next = L  
 from\_ijtag\_interface.SHIFT.next = H  
 print("ijtag\_si.next = si1[i], si1[", i, "] = ", si1[i])  
 ijtag\_si.next = si1[i] # ########################################################### SHIFT  
 print("so1[", i, "] = ", so1[i], ", ijtag\_so = ", ijtag\_so)  
 yield from\_ijtag\_interface.CLOCK.posedge  
 assert(ijtag\_so == so1[i])  
 yield from\_ijtag\_interface.CLOCK.negedge  
 # Update  
 from\_ijtag\_interface.SHIFT.next = L  
 from\_ijtag\_interface.UPDATE.next = H  
  
 # Scan delta data through secondary network to verify the network is working  
 # Forth C, S(10), U is so == bin(00)? 16:8:1:2:1:1 bits required from SI to SO for reference:delta:nf:status:sib1:sib2.  
 print("Scan safe data through SIB1 network. Third C, S(10), U is so == bin(00)?")  
 yield from\_ijtag\_interface.CLOCK.posedge  
 yield from\_ijtag\_interface.CLOCK.negedge  
 # Write Capture value  
 from\_ijtag\_interface.UPDATE.next = L  
 from\_ijtag\_interface.CAPTURE.next = H  
 yield from\_ijtag\_interface.CLOCK.posedge  
 yield from\_ijtag\_interface.CLOCK.negedge  
 for i in range(width):  
 # Write Shift value  
 from\_ijtag\_interface.CAPTURE.next = L  
 from\_ijtag\_interface.SHIFT.next = H  
 print("ijtag\_si.next = si2[i], si2[", i, "] = ", si2[i])  
 ijtag\_si.next = si2[i] # ########################################################### SHIFT  
 yield from\_ijtag\_interface.CLOCK.posedge  
 print("so2[", i, "] = ", so2[i], ", ijtag\_so = ", ijtag\_so)  
 assert(ijtag\_so == so2[i])  
 yield from\_ijtag\_interface.CLOCK.negedge  
 # Update  
 from\_ijtag\_interface.SHIFT.next = L  
 from\_ijtag\_interface.UPDATE.next = H  
  
 # Now test if the delta data persisted correctly  
 # Fifth C, S(10), U is so == bin(00)? 16:8:1:2:1:1 bits required from SI to SO for reference:delta:nf:status:sib1:sib2.  
 print("Scan safe data through SIB1 network. Third C, S(10), U is so == bin(00)?")  
 yield from\_ijtag\_interface.CLOCK.posedge  
 yield from\_ijtag\_interface.CLOCK.negedge  
 # Write Capture value  
 from\_ijtag\_interface.UPDATE.next = L  
 from\_ijtag\_interface.CAPTURE.next = H  
 yield from\_ijtag\_interface.CLOCK.posedge  
 yield from\_ijtag\_interface.CLOCK.negedge  
 for i in range(width):  
 # Write Shift value  
 from\_ijtag\_interface.CAPTURE.next = L  
 from\_ijtag\_interface.SHIFT.next = H  
 print("ijtag\_si.next = si3[i], si3[", i, "] = ", si3[i])  
 ijtag\_si.next = si3[i] # ########################################################### SHIFT  
 yield from\_ijtag\_interface.CLOCK.posedge  
 print(">> so3[", i, "] = ", so3[i], ", ijtag\_so = ", ijtag\_so)  
 assert(ijtag\_so == so3[i])  
 yield from\_ijtag\_interface.CLOCK.negedge  
 # Update  
 from\_ijtag\_interface.SHIFT.next = L  
 from\_ijtag\_interface.UPDATE.next = H  
 yield from\_ijtag\_interface.CLOCK.posedge  
 yield from\_ijtag\_interface.CLOCK.negedge  
 yield from\_ijtag\_interface.CLOCK.posedge  
 yield from\_ijtag\_interface.CLOCK.negedge  
 yield from\_ijtag\_interface.CLOCK.posedge  
 yield from\_ijtag\_interface.CLOCK.negedge  
  
 raise StopSimulation()  
  
 return ip2\_inst, ijtagckgen, fast\_ckgen, ckgen, stimulus

-Figure 35 - IP Core Test Bench Example

## RTL Design ATE Adapter Board

To stitch an IP Core to the ATE Interfaces, a board design containing the single core UUT needs to be created. This is similar to what would need to be implemented for testing a device on a real ATE. This adapter board is used to provide the glue necessary for defining the behavior of the signals used by the core and the ATE. The implementation will follow the process shown in the board test chapter.

Chapter

9

Programming Suggestions

*This chapter describes insights discovered while designing and building the P2654Simulations project.*

t

he P2654Simulations Project is a complex collection of software that models the hardware behavior of various instrumentation and digital logic circuits.

Incorporate Test Benches for Digital Modules in Design File

Incorporating the code for the digital module logic test bench in the same file as the behavioral logic makes it easy to locate the correct test bench for the module. Python makes it simple to conditionally call a method if the module is called stand-alone. This is accomplished using the *if \_\_name\_\_ == “\_\_main\_\_”:* conditional. Several examples of this suggestion have been demonstrated already in this document.

# Use Monitor instances

Monitors are generator methods used to monitor the state of a signal and provide feedback to the user about when the state changes. The method is implemented with the *@instance* decorator. It prints the initial state of the signal. Then the method stays in a while loop yielding on the signal being monitored to return when the state of the signal changes. After the yield statement, the state of the signal is then reported again. An example of the monitor for the PseudoLED instrument is shown below.

@instance  
def monitor\_on():  
 print("\t\tLED[{:s}]({:s}): on".format(self.parent + "." + self.name, self.color), self.on)  
 while 1:  
 yield self.on  
 print("\t\tLED[{:s}]({:s}): on".format(self.parent + "." + self.name, self.color), self.on)

Figure 36 - Example Monitor Method

Chapter

10

Applications and Tools

*This chapter describes the individual applications and tools provided by the P2654Simulations project.*

t

he P2654Simulations Project is broken into a simulation service and supporting tools and subordinate services to help present the state of the simulations.

# simservice

The P2654Simulations primary application is the simservice application. It is implemented as a Telnet Server running in one thread while a myHDL simulation instance is running in a separate thread. The two threads communicate with each other through a pair of queues. One is a Python queue and the other is a simulated hardware queue with myHDL elements. Chapters 3, 4, and 5 describe much of the details of the operation of this application. The following are insights into special elements of the simservice to gain insight into important operation and elements of the application.

## Telnet Service

The Telnet Service is implemented using the Python *telnetsrv* package. From the *telnetsrv.threaded* module the *TelnetHandler* class is specialized by the *SimulatorHandler* class to implement each of the command handlers for the Telnet Service. To simplify the construction of each command handler, the *command* decorator is used from the *telnetsrv.telnetsrvlib* module. The *command* decorator provides a simple interface where each command is handled by a specific method of the *SimulatorHandler* class. The *telnetsrv* package provides utility methods to help format response and error messages back to the Telnet Client that are callable in each command handler. For the Wishbone memory read and write commands, the command handler methods delegate the call to the simulation thread to some private helper methods implementing the communications with the shared queues. These helper methods actually leverage the ATE class to interface with the simulation as the ATE class controls the execution of the simulation.

### Telnetsrv package implementation

Not all Python versions implement the telnetsrv package as part of the pypi available packages. Telnetsrv is generally slow to be implemented for the latest versions of Python releases. Sibling to the P2654Simulations project is a telnetsrv project in the same GitHub repository. This version may be used with the latest Python releases.

## ATE class

The *ATE* class in *hdl.ate.ate* module is the glue binding between the Telnet Service and the Simulation environment. The STARTSIM Telnet command is responsible for initializing the ATE instance as well as starting up the simulator thread by calling the start\_simulation method of the ATE class. The ATE class is also used to bind the test bus signals and the various test bus controllers of the virtual ATE instance. The access to the Wishbone bus is delegated through the WishboneMaster implementation. The ATE.write( ) and ATE.read( ) methods delegate to the WishboneMaster write( ) and read( ) methods.

### WishboneMaster class

The WishboneMaster class is the brains of the simservice application. It implements the handoff from the Telnet Service thread and the Simulation thread of the application. It also implements the pair of queues used to communicate between the two threads. This class is a hybrid class consisting of both general Python methods that interface with the Telnet Service side and generator methods executed by the simulator to implement the top entity of the hardware behavioral model.

# DisplayPanel

The DisplayPanel application is used to provide visual feedback to users of the state of indicator instruments (LEDs) toggled by the simulation. The *LED* instrument in *hld.instruments.led.led* is the only instrument interfacing to the DisplayPanel application at this time. Upon instantiation of an LED instance, the LED registers with the DisplayPanel application if it is running and constructs a visual icon LED on the display. The name of the instance is also associated with the icon as a label in the display. When the LED toggles to True, the icon changes from black to the associated color registered for the LED. A value of False for the LED instance will turn the display icon black.

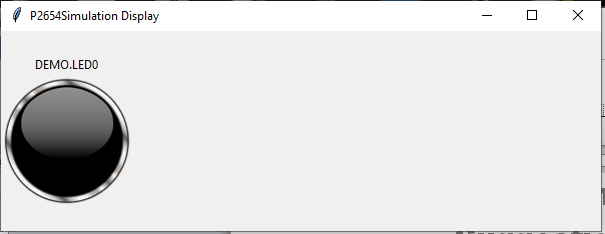
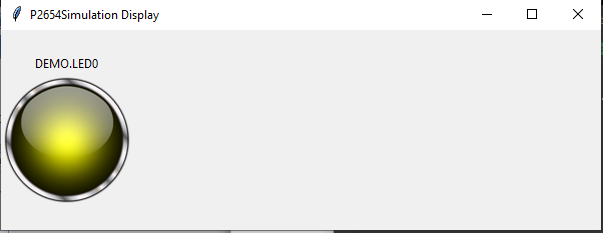


Figure 37 - Example LED in off state

Figure 38 - Example LED in on state

# xsvftool

The xsvftool is an implementation of the xsvftool open source program that has been modified to provide an SVF player for the simservice application. Instead of calling the GPIO or FTDI interfaces as normally supported, the code has been modified to use the JTAGControllers of the ATE interface defined by the CPP driver.

Appendix

A

Appendix A: Use Case Examples

*This chapter describes some of the use cases implemented for testing different P2654 and P1687.1 scenarios. The use cases are implemented as board designs in the P2654 design space.*

t

he P2654Simulations use cases are implemented as board designs that may be plugged into the virtual ATE test environment. The boards contain interfaces to the ATE test busses for access to the circuits being exercised by the use case.

# GPIOTest Board

The GPIOTest Board is used to verify the ATE GPIO test bus is operating properly. The board consists of loopback signals tying an output signal to an input signal to enable automated testing through the ATE environment. The GPIOTest Board also contains 5 PseudoLED indicator instruments to provide feedback with monitor methods as to the state of the signals to provide manual feedback.

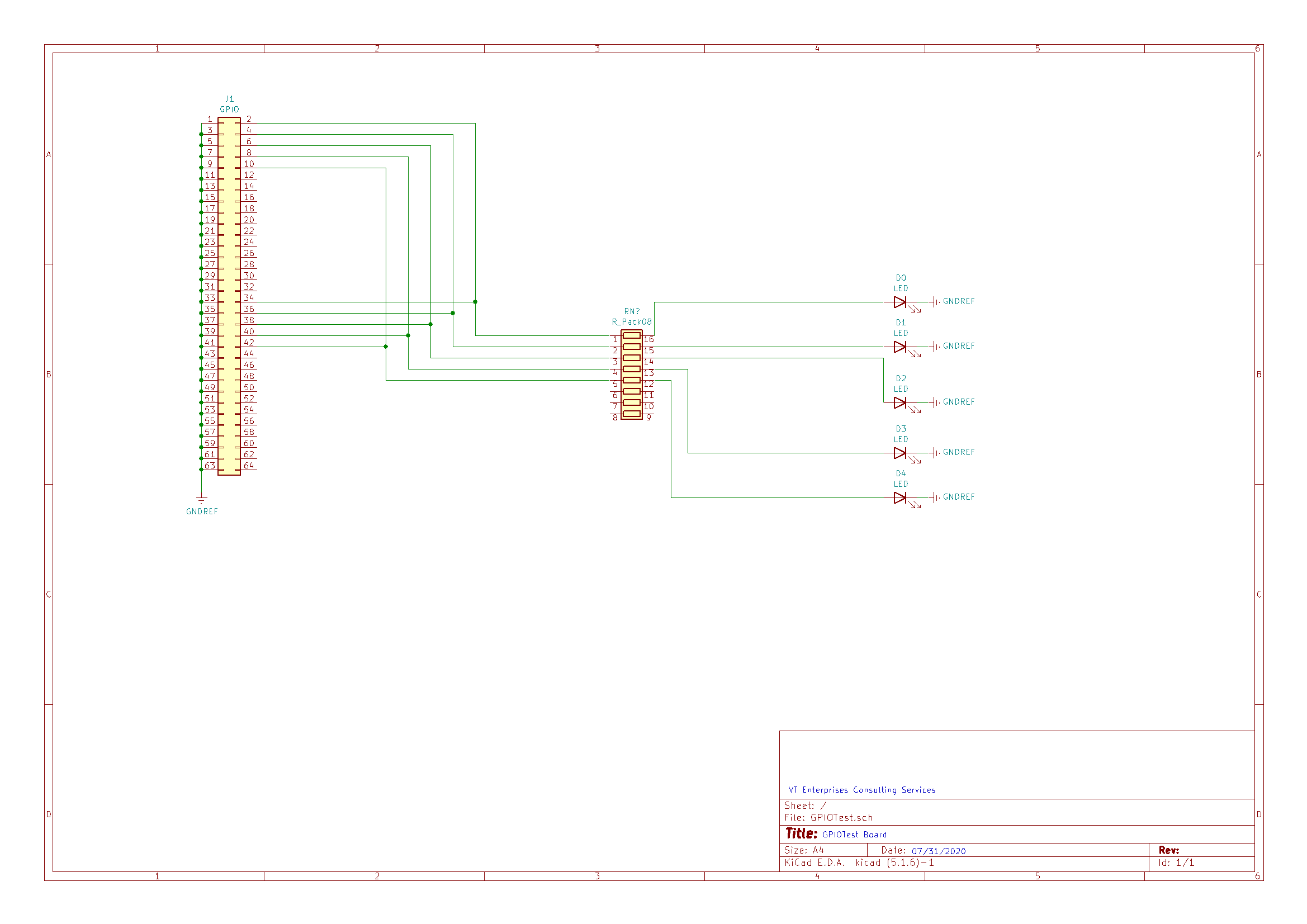
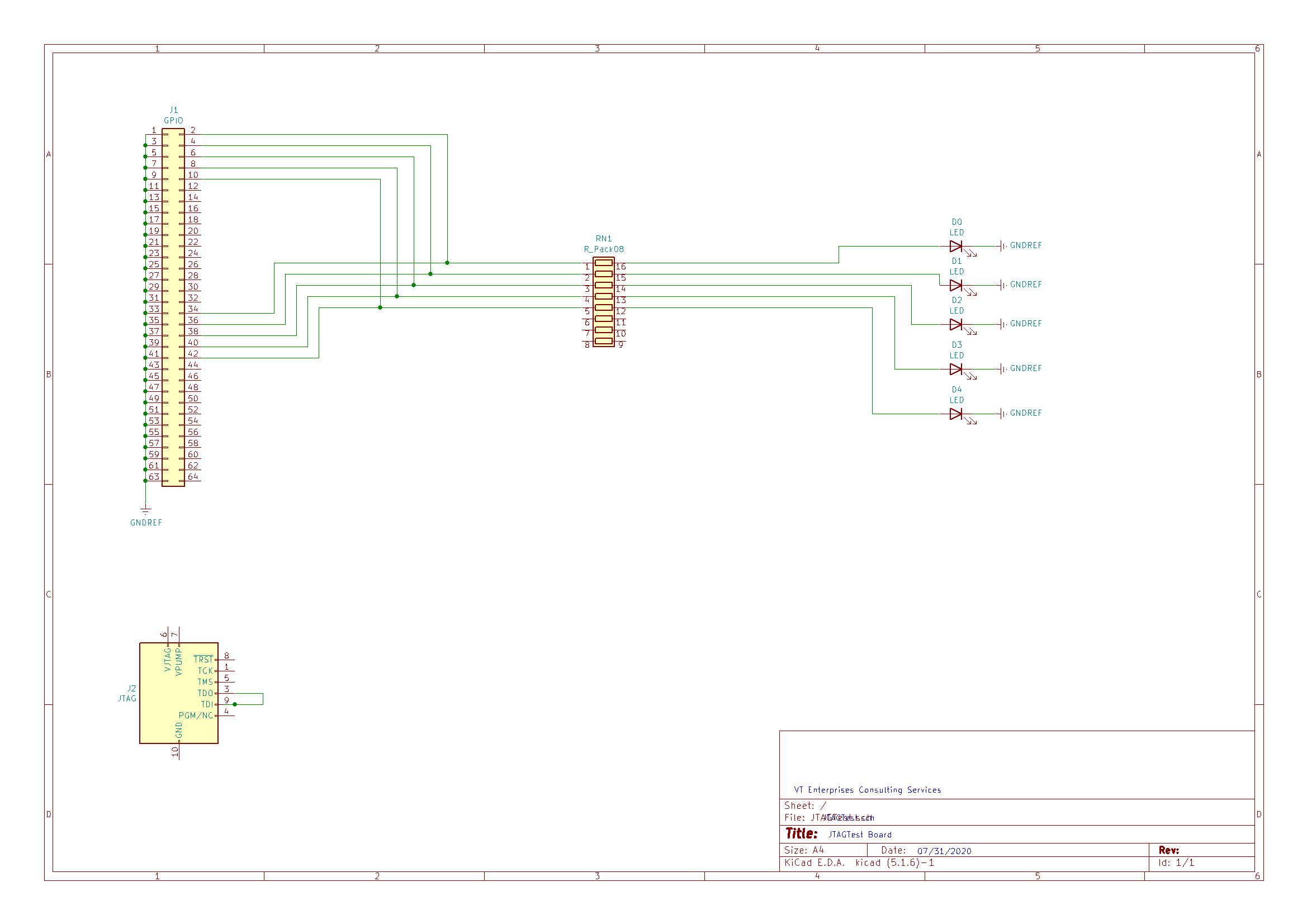


Figure 39 - GPIOTest Board Schematic

# JTAGTest Board

The JTAGTest Board is used to test the JTAGController interface of the virtual ATE. The board contains the same circuit as the GPIOTest Board along with a JTAG connector looping back TDI to TDO.

Figure 40 - JTAGTest Board Schematic

# JTAG2Test Board

The JTAG2Test Board is identical to the JTAGTest Board other than it is wired to the JTAG2Controller of the ATE instead.

# I2CTest Board

The I2CTest Board contains the same GPIO circuit of the GPIOTest Board along with an I2C expanded interface with separate in, out, and oe signals for clock and data. It also includes the I2CDeviceSEP device wired to that connector. The I2CDeviceSEP contains 8 r/w registers that may be set and read.

# SPITest Board

The SPITest Board is the overall test board for the ATE. It includes the GPIOTest circuit, the JTAG loopback, the I2CDeviceSEP, and the SPIDevice device wired to a SPI interface. The SPIDevice contains a 32 bit shift register partitioned with 8 address bits and 24 data bits. Three registers are implemented by the slave module.

Appendix

B

Appendix B: Device Examples

*This chapter describes various device designs implemented by the P2654Simulations Project. These devices may be implemented on the board designs to create new use cases. New devices are added as new features for use cases are identified.*

t

he P2654Simulations Project contains individual devices and composite devices consisting of multiple IP cores assembled into a single device package. A device may also contain multiple test bus interfaces.

# I2CDeviceSEP

The I2CDeviceSEP contains an expanded I2C interface with separate in, out, and oe for each signal instead of open collectors. This is because myHDL does not support open-collector, but only tri-state. The device contains 8 r/w register accessible from the interface port.

# SPIDevice

The SPIDevice contains the SPI interface to a 32-bit shift register. The 32-bit register is partitioned into 8-bits for selecting a 24-bit data register and 24-bits of data for that selected register. This device had 3 data registers implemented.

# SN74ABT8244A

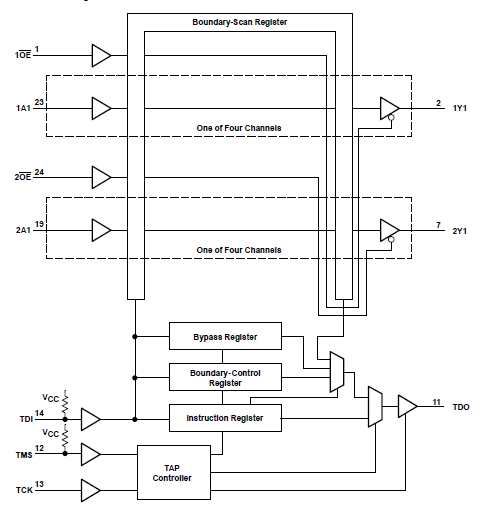


Figure 41 - Architecture of SN74ABT8244A

# SN74ABT8245A

# 

Figure 42 - Architecture of SN74ABT8245A

# SN7404

# 

Figure 43 - Architecture of SN7404

# SN74LS138

# 

Figure 44 - Architecture of SN74LS138

# AS6C6264

# 

Figure 45 - Architecture of AS6C6264 SRAM

# RearickDevice4

# 

Figure 46 - Architecture of RearickDevice4

Appendix

C

Appendix C: Instrument Examples

*This chapter describes various IP cores implemented as instrumentation or compositions of instrumentation used in the demonstration devices. These instrument IP are categorized as generalized or common logic, composite logic, and standards required logic.*

t

he P2654Simulations Project contains a variety of instrumentation logic available to be included in demonstration devices. These instruments may be integrated to different test buses depending on the design hierarchy used to contain them.

# General Instruments

## Clock Counter

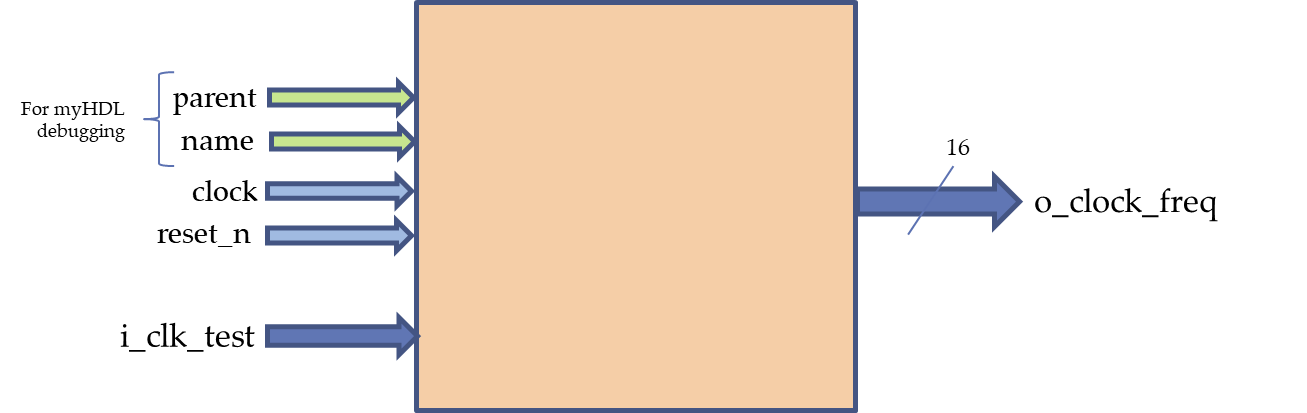


Figure 47 - Architecture of Clock Frequency Counter Instrument

|  |  |  |
| --- | --- | --- |
| **Bit** | **Signal Name** | **Description** |
| 15 | MSB | Most Significant Bit of frequency in Hz. |
| 14 | Bit14 | *Bit of frequency in Hz.* |
| 13 | Bit13 | *Bit of frequency in Hz.* |
| 12 | Bit12 | *Bit of frequency in Hz.* |
| 11 | Bit11 | *Bit of frequency in Hz.* |
| 10 | Bit10 | *Bit of frequency in Hz.* |
| 9 | Bit9 | *Bit of frequency in Hz.* |
| 8 | Bit8 | *Bit of frequency in Hz.* |
| 7 | Bit7 | *Bit of frequency in Hz.* |
| 6 | Bit6 | *Bit of frequency in Hz.* |
| 5 | Bit5 | *Bit of frequency in Hz.* |
| 4 | Bit4 | *Bit of frequency in Hz.* |
| 3 | Bit3 | *Bit of frequency in Hz.* |
| 2 | Bit2 | *Bit of frequency in Hz.* |
| 1 | Bit1 | *Bit of frequency in Hz.* |
| 0 | LSB | *Least Significant Bit of frequency in Hz.* |

Figure 48 - Clock Frequency Counter o\_clock\_freq Register

## Clock Generator

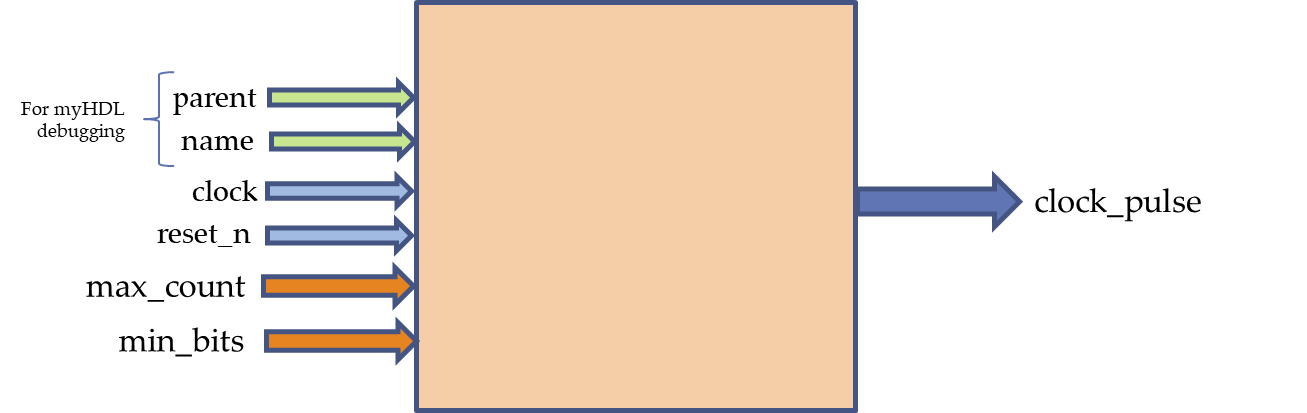


Figure 49 - Architecture of Clock Generator Instrument

## Comparator



Figure 50 - Architecture of Comparator Instrument

|  |  |  |
| --- | --- | --- |
| **Bit** | **Signal Name** | **Description** |
| 8 | MSB | *Most Significant Bit of Temperature in Fahrenheit.* |
| 7 | bit7 | *Data bit of Temperature in Fahrenheit.* |
| 6 | bit6 | *Data bit of Temperature in Fahrenheit.* |
| 5 | bit5 | *Data bit of Temperature in Fahrenheit.* |
| 4 | bit4 | *Data bit of Temperature in Fahrenheit.* |
| 3 | bit3 | *Data bit of Temperature in Fahrenheit.* |
| 2 | bit2 | *Data bit of Temperature in Fahrenheit.* |
| 1 | bit1 | *Data bit of Temperature in Fahrenheit.* |
| 0 | LSB | *Least Significant Bit of Temperature in Fahrenheit.* |

Figure 51 - Comparator Temperature Register

|  |  |  |
| --- | --- | --- |
| **Bit** | **Signal Name** | **Description** |
| 8 | MSB | *Most Significant Bit of low\_register Temperature in Fahrenheit.* |
| 7 | bit7 | *Data bit of low\_register Temperature in Fahrenheit.* |
| 6 | bit6 | *Data bit of low\_register Temperature in Fahrenheit.* |
| 5 | bit5 | *Data bit of low\_register Temperature in Fahrenheit.* |
| 4 | bit4 | *Data bit of low\_register Temperature in Fahrenheit.* |
| 3 | bit3 | *Data bit of low\_register Temperature in Fahrenheit.* |
| 2 | bit2 | *Data bit of low\_register Temperature in Fahrenheit.* |
| 1 | bit1 | *Data bit of low\_register Temperature in Fahrenheit.* |
| 0 | LSB | *Least Significant Bit of low\_register Temperature in Fahrenheit.* |

Figure 52 - Comparator low\_register Register

|  |  |  |
| --- | --- | --- |
| **Bit** | **Signal Name** | **Description** |
| 8 | MSB | *Most Significant Bit of high\_register Temperature in Fahrenheit.* |
| 7 | bit7 | *Data bit of high\_register Temperature in Fahrenheit.* |
| 6 | bit6 | *Data bit of high\_register Temperature in Fahrenheit.* |
| 5 | bit5 | *Data bit of high\_register Temperature in Fahrenheit.* |
| 4 | bit4 | *Data bit of high\_register Temperature in Fahrenheit.* |
| 3 | bit3 | *Data bit of high\_register Temperature in Fahrenheit.* |
| 2 | bit2 | *Data bit of high\_register Temperature in Fahrenheit.* |
| 1 | bit1 | *Data bit of high\_register Temperature in Fahrenheit.* |
| 0 | LSB | *Least Significant Bit of high\_register Temperature in Fahrenheit.* |

Figure 53 - Comparator high\_register Register

|  |  |  |
| --- | --- | --- |
| **Bit** | **Signal Name** | **Description** |
| 7 | Reserved7 | *Reserved. Added to pad read to 8 bits.* |
| 6 | Reserved6 | *Reserved. Added to pad read to 8 bits.* |
| 5 | Reserved5 | *Reserved. Added to pad read to 8 bits.* |
| 4 | Reserved4 | *Reserved. Added to pad read to 8 bits.* |
| 3 | Reserved3 | *Reserved. Added to pad read to 8 bits.* |
| 2 | Reserved2 | *Reserved. Added to pad read to 8 bits.* |
| 1 | above | *1=Temperature above high value, 0=Temperature at or below high value* |
| 0 | below | *1=Temperature fell below low value, 0=Temperature at or above low value* |

Figure 54 - Comparator Status Register

## LED

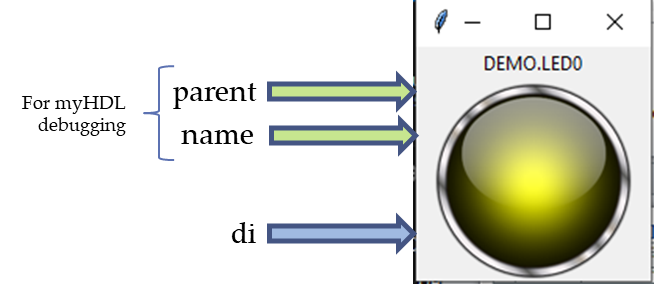


Figure 55 - Architecture of LED Instrument

|  |  |  |
| --- | --- | --- |
| **Bit** | **Signal Name** | **Description** |
| 0 | di | *Data input port: 1=ON, 0=OFF* |

Figure 56 - LED Port Description

## Noise Maker

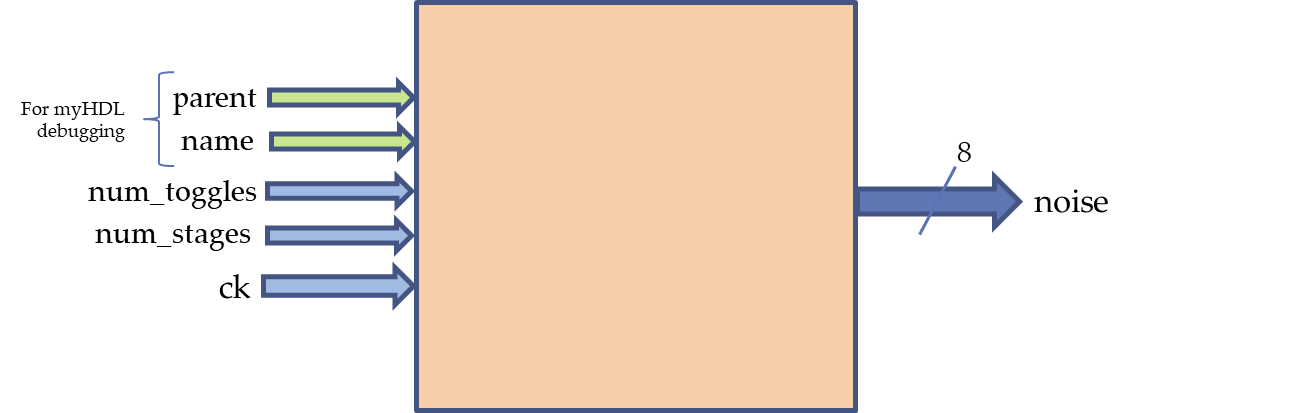


Figure 57 - Architecture of Noise Maker Instrument

|  |  |  |
| --- | --- | --- |
| **Bit** | **Signal Name** | **Description** |
| 4 | MSB | *Most Significant Bit of num\_toggles register.* |
| 3 | Bit3 | *Bit of num\_toggles register.* |
| 2 | Bit2 | *Bit of num\_toggles register.* |
| 1 | Bit1 | *Bit of num\_toggles register.* |
| 0 | LSB | *Least Significant Bit of num\_toggles register.* |

Figure 58 - Noise Maker num\_toggles Register

NOTE: Number of toggles to perform per stage. Signal(intbv(0, min=0, max=MAX\_TOGGLES))

|  |  |  |
| --- | --- | --- |
| **Bit** | **Signal Name** | **Description** |
| 3 | MSB | *Most Significant Bit of num\_stages register.* |
| 2 | Bit2 | *Bit of num\_stages register.* |
| 1 | Bit1 | *Bit of num\_stages register.* |
| 0 | LSB | *Least Significant Bit of num\_stages register.* |

Figure 59 - Noise Maker num\_stages Register

NOTE: Number of stages to perform per clock cycle. Signal(intbv(0, min=0, max=MAX\_STAGES))

## Power Supply Monitor

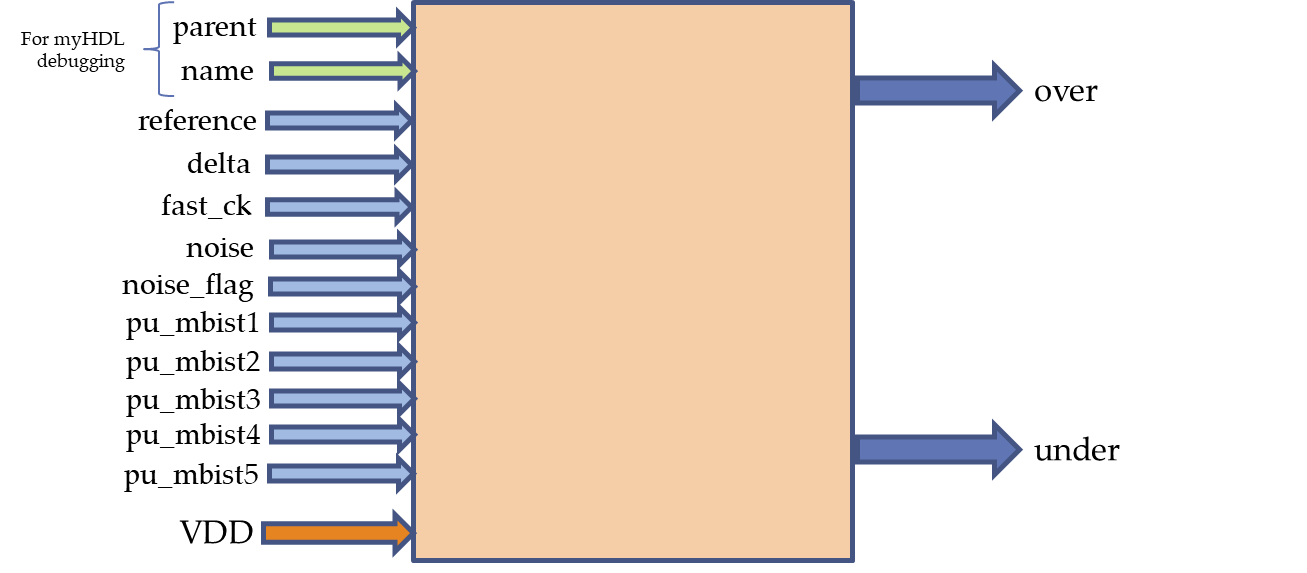


Figure 60 - Architecture of Power Supply Monitor Instrument

|  |  |  |
| --- | --- | --- |
| **Bit** | **Signal Name** | **Description** |
| 15 | MSB | *Most Significant Bit of reference voltage in millivolts.* |
| 14 | Bit14 | *Bit of reference voltage in millivolts.* |
| 13 | Bit13 | *Bit of reference voltage in millivolts.* |
| 12 | Bit12 | *Bit of reference voltage in millivolts.* |
| 11 | Bit11 | *Bit of reference voltage in millivolts.* |
| 10 | Bit10 | *Bit of reference voltage in millivolts.* |
| 9 | Bit9 | *Bit of reference voltage in millivolts.* |
| 8 | Bit8 | *Bit of reference voltage in millivolts.* |
| 7 | Bit7 | *Bit of reference voltage in millivolts.* |
| 6 | Bit6 | *Bit of reference voltage in millivolts.* |
| 5 | Bit5 | *Bit of reference voltage in millivolts.* |
| 4 | Bit4 | *Bit of reference voltage in millivolts.* |
| 3 | Bit3 | *Bit of reference voltage in millivolts.* |
| 2 | Bit2 | *Bit of reference voltage in millivolts.* |
| 1 | Bit1 | *Bit of reference voltage in millivolts.* |
| 0 | LSB | *Least Significant Bit of reference voltage in millivolts.* |

Figure 61 - Power Supply Monitor reference Register

NOTE: Reference value of what the power supply voltage should be as a Signal(intbv(0)[16:]) type. Setting the reference value to zero (0) will reset the under and over signals.

|  |  |  |
| --- | --- | --- |
| **Bit** | **Signal Name** | **Description** |
| 1 | over | *Signal to indicate the monitor detected the voltage exceeded the delta setting. Signal(bool(0)) type.* |
| 0 | under | *Signal to indicate the monitor detected the voltage fell below the delta setting. Signal(bool(0)) type.* |

Figure 62 - Power Supply Monitor Status Register

NOTE: over: Signal to indicate the monitor detected the voltage exceeded the delta setting. Signal(bool(0)) type.   
under: Signal to indicate the monitor detected the voltage fell below the delta setting. Signal(bool(0)) type.

|  |  |  |
| --- | --- | --- |
| **Bit** | **Signal Name** | **Description** |
| 0 | noise\_flag | *Signal to enable(1) or disable(0) noise influence on Power Supply Monitoring. Signal(bool(1)) type.* |

Figure 63 - Power Supply Monitor noise\_flag Register

NOTE: noise\_flag: Signal to enable or disable noise influence on Power Supply Monitoring. Signal(bool(1)) type.

|  |  |  |
| --- | --- | --- |
| **Bit** | **Signal Name** | **Description** |
| 7 | MSB | *Most Significant Bit of delta register.* |
| 6 | Bit6 | *Bit of delta register.* |
| 5 | Bit5 | *Bit of delta register.* |
| 4 | Bit4 | *Bit of delta register.* |
| 3 | Bit3 | *Bit of delta register.* |
| 2 | Bit2 | *Bit of delta register.* |
| 1 | Bit1 | *Bit of delta register.* |
| 0 | LSB | *Least Significant Bit of delta register.* |

Figure 64 - Power Supply Monitor delta Register

NOTE: The amount of mV variance allowed around the voltage reference as a Signal(intbv(0)[8:]) type.

## PseudoLED

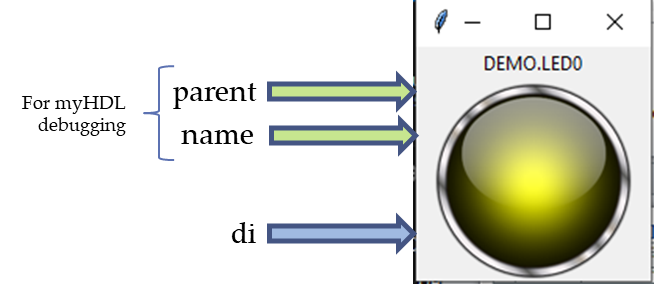


Figure 65 - Architecture of PseudoLED

|  |  |  |
| --- | --- | --- |
| **Bit** | **Signal Name** | **Description** |
| 0 | di | *Data input port: 1=ON, 0=OFF* |

Figure 66 - PseudoLED Port Description

## SimulatedMBIST

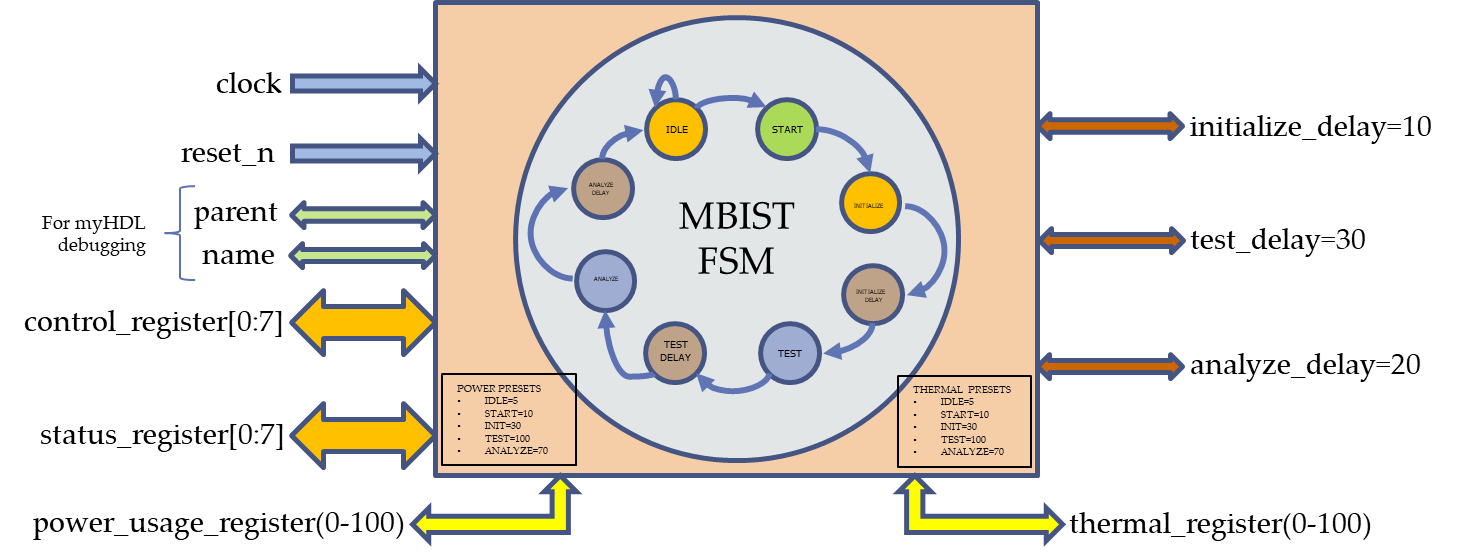


Figure 67 - Architecture of Simulated MBIST IP

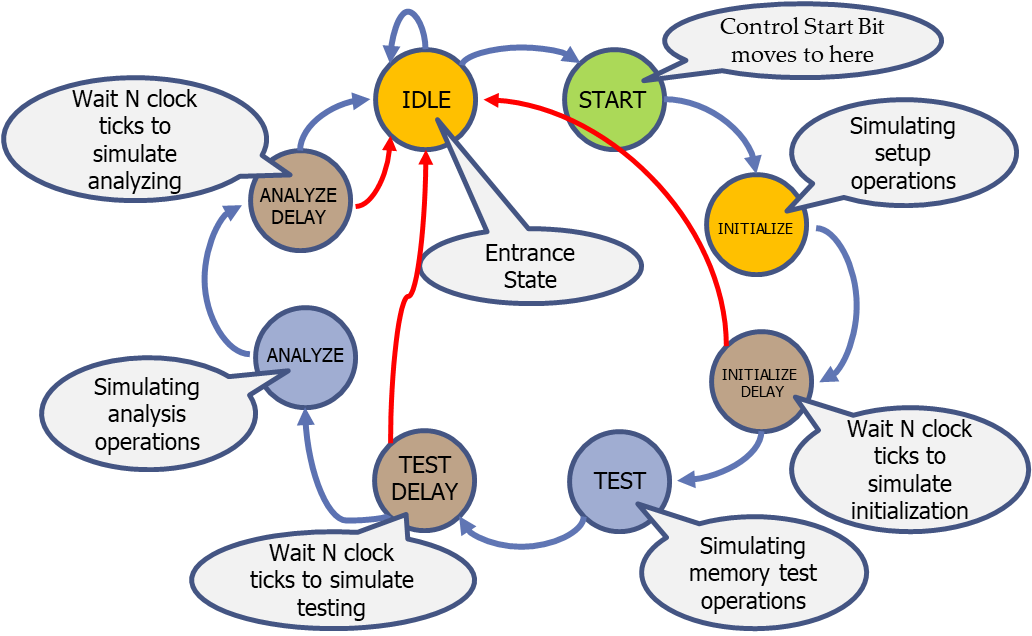


Figure 68 - MBIST Simulation State Machine

|  |  |  |
| --- | --- | --- |
| **Bit** | **Signal Name** | **Description** |
| 6 | analyze\_wait | *1=Double the analyze\_delay time to use at start, 0=Use the specified analyze\_delay* |
| 5 | test\_wait | *1=Double the test\_delay time to use at start, 0=Use the specified test\_delay* |
| 4 | initialize\_wait | *1=Double the initialize\_delay time to use at start, 0=Use the specified initialize\_delay* |
| 3 | inj\_analyze\_err | *1=Inject error during analyze\_delay state, 0=Do not inject error during analyze\_delay state* |
| 2 | inj\_test\_err | *1=Inject error during test\_delay state, 0=Do not inject error during test\_delay state* |
| 1 | abort | *1=Stop the BIST operation and abort, 0=Do not abort the test* |
| 0 | start | *1=Start the BIST operation, 0=NOP for status scans* |

Figure 69 - MBIST Control Register

|  |  |  |
| --- | --- | --- |
| **Bit** | **Signal Name** | **Description** |
| 6 | Reserved1 | *Reserved. Added so status\_register can be capture register and control\_register as update* |
| 5 | Reserved0 | *Reserved. Added so status\_register can be capture register and control\_register as update* |
| 4 | analyze\_error | *1=Error during analyze state detected, 0=No error detected during analyze state* |
| 3 | test\_error | *1=Error during test state detected, 0=No error detected during test state* |
| 2 | unknown\_error | *1=Test aborted due to unknown error, 0=Test did not abort* |
| 1 | status | *1=MBIST test is running, 0=MBIST test is not running* |
| 0 | result | *1=Test passed, 0=Test failed* |

Figure 70 - MBIST Status Register

## Thermometer

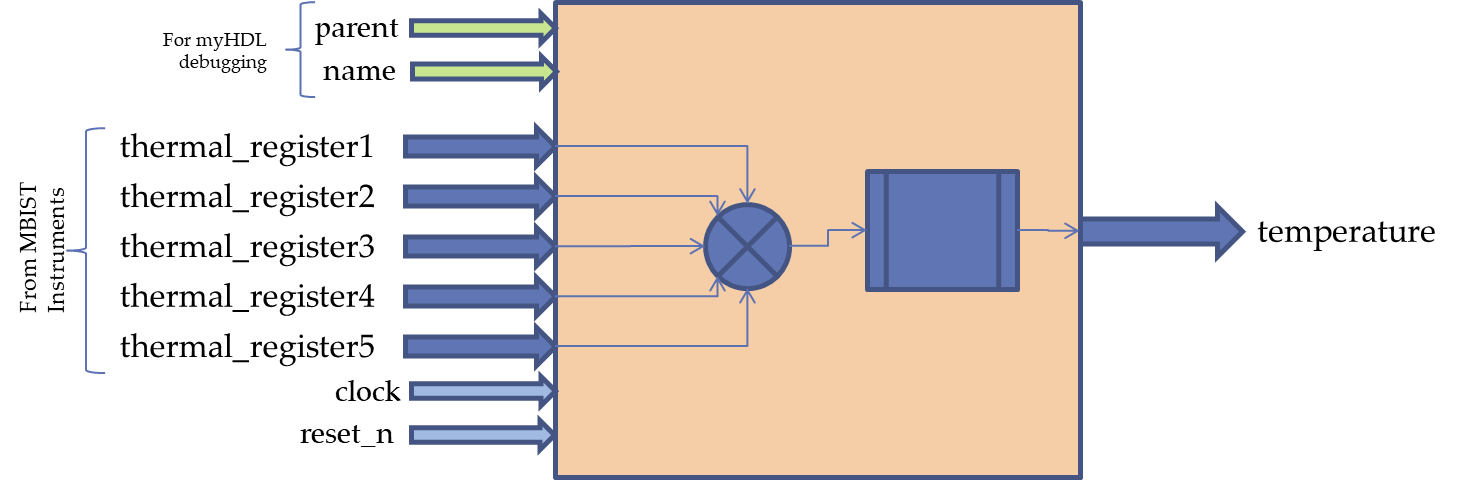


Figure 71 - Architecture of Thermometer Instrument

|  |  |  |
| --- | --- | --- |
| **Bit** | **Signal Name** | **Description** |
| 7 | MSB | Most Significant Bit of thermal\_register from MBISTX. |
| 6 | bit6 | *Data bit of thermal\_register from MBISTX.* |
| 5 | bit5 | *Data bit of thermal\_register from MBISTX.* |
| 4 | bit4 | *Data bit of thermal\_register from MBISTX.* |
| 3 | bit3 | *Data bit of thermal\_register from MBISTX.* |
| 2 | bit2 | *Data bit of thermal\_register from MBISTX.* |
| 1 | bit1 | *Data bit of thermal\_register from MBISTX.* |
| 0 | LSB | *Least Significant Bit of thermal\_register from MBISTX.* |

Figure 72 - Thermometer thermal\_registerX Register

|  |  |  |
| --- | --- | --- |
| **Bit** | **Signal Name** | **Description** |
| 8 | MSB | *Most Significant Bit of Temperature in Fahrenheit.* |
| 7 | bit7 | *Data bit of Temperature in Fahrenheit.* |
| 6 | bit6 | *Data bit of Temperature in Fahrenheit.* |
| 5 | bit5 | *Data bit of Temperature in Fahrenheit.* |
| 4 | bit4 | *Data bit of Temperature in Fahrenheit.* |
| 3 | bit3 | *Data bit of Temperature in Fahrenheit.* |
| 2 | bit2 | *Data bit of Temperature in Fahrenheit.* |
| 1 | bit1 | *Data bit of Temperature in Fahrenheit.* |
| 0 | LSB | *Least Significant Bit of Temperature in Fahrenheit.* |

Figure 73 - Thermometer Temperature Register

# IEEE 1149.1 Instruments

## STD1149\_1\_TAP



Figure 74 - Architecture of Std1149\_1\_TAP Entity

## TDR



Figure 75 - Architecture of TDR

## TIR



Figure 76 - Architecture of TIR

# IEEE 1500 Instruments

## SELWIR



Figure 77 - Architecture of Select Wrapper Instruction Register (SELWIR)

## WBY



Figure 78 - Architecture of Wrapper Bypass Register (WBY)

## WDRMUX



Figure 79 - Architecture of Wrapper Data Register MUX (WDRMUX)

## WIR



Figure 80 - Architecture of Wrapper Instruction Register (WIR)

## WIRMUX



Figure 81 - Architecture of Wrapper Instruction Register MUX (WIRMUX)

## WSP

def \_\_init\_\_(self):  
 self.AUXCKn = Signal(bool(0))  
 self.WRCK = Signal(bool(0))  
 self.WRSTN = Signal(bool(1))  
 self.TransferDR = Signal(bool(0))  
 self.UpdateWR = Signal(bool(0))  
 self.ShiftWR = Signal(bool(0))  
 self.CaptureWR = Signal(bool(0))  
 self.SelectWIR = Signal(bool(0))

Figure 82 - Architecture of Wrapper Serial Port (WSP)

## WSReg



Figure 83 - Architecture of Wrapper Serial Register (WSREG)

# IEEE 1687 Instruments

## SIB\_MUX\_PRE



Figure 84 - Architecture of SIB\_MUX\_PRE

## SIB\_MUX\_POST



Figure 85 - Architecture of SIB\_MUX\_POST

## SReg



# Index

background, 1

border, 3

bullet, 1

caption, 2

color, 2

drawing, 2

drop cap, 1

footer, 3

frame, 3

graphic, 2

group, 2

header, 3

Help, 3

link, 4

margins, 2

normal view, 1

number, 4

picture, 2, 3, 4

print, 1

re-size, 3

section break, 2

shading, 1

style, 1, 2, 3, 4

symbol, 1

Table of Contents, 3

template, 4

ungroup, 2

Wingdings, 1