School of Electronics and Computer Science
Faculty of Engineering and Physical Sciences
UNIVERSITY OF SOUTHAMPTON

Ricki S. Tura
June 28, 2019

ELEC3200 Industrial Studies Final Report

Industrial Supervisor: Melvin Cheah
Academic Supervisor: Harold Chong

Word count: 6528

A project report submitted for the award of
MEng Electronic and Electrical Engineering with Industrial Studies

Abstract

During my placement with UltraSoC Technologies Ltd, I have completed a variety of projects that have developed my skill set. This includes script writing in Python and Perl, making use of web frameworks such as Django, source code management, testing on FPGA platforms, and hardware verification. This report outlines the projects undertaken and the skills developed.

Contents

List of Figures

Abbreviations

AXI	Advanced eXtensible Interface
CSV	Comma Separated Values
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
IMC	Integrated Metrics Center
IP	Intellectual Property
RTL	Register Transfer Level
SoC	System on Chip
TCL	Tool Command Language
TFTP	Trivial File Transfer Protocol
UVM	Universal Verification Methodology
YAML	YAML Ain't Markup Language

Acknowledgements

I would like to thank my Industrial Supervisor Melvin Cheah and my Academic Supervisor Harold Chong for their guidance and support throughout the industrial placement.

Chapter 1

Introduction

My industrial placement was with UltraSoC Technologies Ltd, a Cambridge based start-up company that designs semiconductor IP for SoCs. Their products enable the SoC with debugging and analytics capabilities; this helps to reduce the chip's development time, increase performance, and provides security at the hardware level. As an IP company almost all of the work done at UltraSoC is on design, no physical products are made.

The engineers at UltraSoC are loosely split into three groups: hardware, software, and applications. I have been working with the hardware team during my placement and I reported to my line manager and Industrial Supervisor Melvin Cheah. This involved having weekly meetings where I explained the progress I have made, outlined my next steps, and asked questions. The company is based in an open plan office, which had made it very easy for me to speak to my manager outside of these meetings about any queries I have.

I also attended weekly hardware team meetings that were led by the VP of Engineering. These meetings were for key notices, and allowed everyone in the hardware team to present their progress over the week. Lastly, weekly "All Hands" meetings took place where I gained an insight into the other parts of the company including developments from the software team, applications, sales, and marketing.

I have worked on seven projects; they have been self-contained and it has been solely my responsibility to complete them. I have been able to work at my own pace which has tested my time management and project planning skills. Many of the projects have built upon the work done by other engineers, so despite working individually I have also consulted my colleagues when necessary to ensure my work doesn't impact the function of any previously built tools.

To plan my work and keep track of progress, I used a logbook. This follows on from the logbook maintenance learnt from my previous years of study at ECS.

The projects I have worked on are:

- Enhancement of a regression testing system:
    - Coverage merging of tests.
    - Automatic Bugzilla reporting.
- Creation of an XML to DOCX tool for IP user guides.
- Verification of an UltraSoC module.
- Creation of a GUI to improve the release flow of IP.
- Integration verification flow for UltraSoC IP in customer SoC.
- Constraining a traffic generator to provide better stimulus when verifying a module.
- Creation of an FPGA regression testing system.

I will discuss each project in the remaining chapters.

Chapter 2

Regression System Enhancement

As UltraSoC's IP blocks are developed they undergo daily regression testing. This involves running tests with randomly generated seeds which determine the values of the input stimuli. This allows IP designers and verifiers to iteratively develop the IP and identify when a change has caused an issue.

Regression testing helps to fix problems as and when they arise instead of later on in the development cycle when such an issue might take more time and resources to fix. For example, if a test has been consistently passing in previous regression tests and suddenly starts failing after a new feature has been added to the IP, the user can narrow down the cause of the failed test to the newly introduced feature.

The regression system is built in house, and is made up of several Python scripts that interact with tables in a MySQL database and generate bash scripts which run the tests. The results from the tests are viewable on web pages written in Perl CGI, which uses SQL queries to obtain the information and then renders the information in a HTML template.

I already had some knowledge of Python and SQL prior to this project, which put me in a good position to develop on my skills and complete the following tasks.

2.1 Coverage Merging of Tests

2.1.1 The Problem

Coverage is a metric used to determine whether all of the functions of a piece of IP have been effectively tested, which can be separated into two main types: functional coverage and code coverage.

Functional coverage is a measure of how much of the IP's functionality has been tested and code coverage is a measure of how much of the code that makes up the IP has been executed in the test [1]. By writing tests that exercise different parts of the IP and merging the coverage from these results, we can gain a better understanding of whether the entirety of the IP has been tested and 100% coverage has been achieved.

An IP block can have multiple tests, with each test made up of multiple test runs with different initial conditions. These initial conditions are determined by the test's seed value which is randomised for each test run.

Coverage data is created by the Cadence simulator. Each test run produces a `.ucd` file which contains all of the coverage information achieved by the test, and a `.ucm` file is generated from the IP block which contains information on what coverage can be achieved. My task was to improve the regression system so that it:

- Merges the coverage data from all of the tests' `.ucd` files into a single `.ucd` file.
- Generates text reports and HTML reports on the coverage.
- Displays text report on the results web page.
- Allows the `.ucm` and `.ucd` files to be downloaded from the results web page.
- Obtains the overall coverage value from the report, and stores this value in the MySQL database.

2.1.2 The Solution

By editing the regression system's Python scripts, I created an extra Bash script that would run after each test which opens Cadence's IMC program to generate the merged coverage files and reports. The interaction with IMC was done through a TCL script which sends commands to IMC through a command line interface (CLI). TCL was used as it was the recommended way to use IMC according to the user guide. It is quicker than interacting with IMC through the GUI and using the CLI means that interactions can be made into a script that can be ran by programs which removes the need for human involvement.

I then used a combination of SQL, Perl CGI, and HTML to allow the user to view the results, access reports, and download files. The existing regression results pages are written in Perl CGI and I made improvements to these pages so that a user can request coverage reports. This would trigger an SQL database query, which obtains the report, and the related Perl code to display the report on the screen.

Accessing the `.ucm` and `.ucd` files was also done using Perl. Once the files were retrieved from the database, they were compressed into a tarball and passed through to the HTML template for the user to download.

A side task that spun out from this project was a script that could merge test coverage across multiple regressions. I created this script from scratch, which meant I could embed formatted comments throughout the script so that documentation for the script could be auto-generated using NaturalDocs, which will help with script maintenance.

2.1.3 Evaluation and Skills Learned

This was an essential task that needed to have been completed. This and the other improvements I made, such as allowing for regressions on local files, has provided more detailed coverage data and a reduced test run time. This has benefited UltraSoCs Hardware Team of 10 engineers who interact with the regression tools daily.

From this task, I learnt about various Python techniques such as using APIs to communicate with MySQL databases and generating Bash scripts that get run on a grid engine. I used TCL for the first time, and gained experience interacting with EDA tools using TCL which saves time compared to using a GUI. Perl was also learnt for the first time and it was used to manipulate file directories, interact with databases, and generate HTML web pages to display the information extracted from these databases.

2.2 Automatic Bugzilla Reporting

2.2.1 The Problem

As a mature IP block gets closer to being fully developed and has been successfully tested over many iterations, failing tests become rare. However, any tests that do fail are a result of the random seed for that test creating a set of conditions that exposes a corner case for the IP. These situations are of key concern for verifiers, and when they occur they should be acted upon. Identifying the few failing tests amongst many passing tests was time consuming.

The Hardware Team already uses a bug tracking system called Bugzilla to record and track known bugs in IP blocks, but there was nothing in place to automatically file Bugzilla tickets when a test failed on a module. The task was to edit the regression testing system to detect when tests on "stable" IP blocks failed, generate the appropriate bug tickets, and notify the owners of the affected IP.

2.2.2 The Solution

After conducting some research I found a Python module that handles most of the interactions between Python and a Bugzilla server through a REST API, which is a way of communicating between two different applications through the use of GET requests and POST requests [2].

By adding extra code to the regression scripts, a check was made in the SQL database to search for failed tests. This was cross checked against a list of "stable" IP blocks defined in a YAML file hosted on an SVN repository and if any failed tests were associated with one of these blocks then the failed test was filed in Bugzilla. The Python script would create a dictionary object containing all of the information required to create a bug ticket, and send this to the Bugzilla server using the REST API.

The same approach was also used to check the regression tests for any tests that have been improperly killed, and for tests that have exceeded a time out.

2.2.3 Evaluation and Skills Learned

The completion of this task has made it easier for the Hardware Team to keep track of corner cases identified by the regression system. This will decrease verification time, and ensure that any failed tests do not pass undetected.

From this task I learnt about the REST API and how to communicate between two different frameworks, the YAML format for storing information, and the basics of bug reporting in a project. I further developed my skills in Python scripting.

Chapter 3

User Guide Generation Tools

Each IP block has an accompanying User Guide which is formatted as a Word document (DOCX file). Part of the User Guide contains information on how to set various registers in the IP block, this information is presented in register tables. These tables contain rows for each group of bits that make up the register, and the columns of each row contain the name, the width of the field in bits, and information on what each bit means.

3.1 DOCX vs XML Comparison Tool

3.1.1 The Problem

The register tables are copied by hand from the source XML file, which is a long and error-prone process. There was nothing in place to confirm if the tables in the DOCX matched those in the XML, and my task was to create a script that achieved this.

3.1.2 The Solution

There was already code in place to convert the tables defined in the source XML into tables defined in a CSV file named `messages.csv`. My script would have to convert the DOCX User Guide into the same CSV format and then compare the tables in the two CSV files for discrepancies. This script, `wordtocsv.py`, made use of the ElementTree Python module.

Using this module the script stepped through the XML and extracted the information from any tables present. By using this data the script created a CSV file that matched the format of `messages.csv`. Another script was made, `csvcompare.py`, that would take in the XML CSV and the DOCX CSV and make the comparisons line by line. The output was recorded in a text file that the user could then read.

3.1.3 Evaluation and Skills Learned

These scripts have since been used and have already detected issues with the current documentation. They have made it easier and faster to check the validity of the user guides before they are published.

3.2 DOCX Generator Tool

3.2.1 The Problem

The next task was to create a tool that would generate the register tables directly from the source XML. Depending on the IP block, the source XML could be single or multiple XML files, and the tool would have to extract the information and generate the tables correctly. Furthermore the tool should be able to read in a previous version of the User Guide, extract the information from the old tables, and combine this with the new data in the source XML.

3.2.2 The Solution

To complete this task I had to familiarise myself with XSL transformations, XPath, and the internal structure of a DOCX file. XSL transformations, or XSLT, are pieces of code that can be applied to an XML file to transform it. The transformation is done using an XSLT processor. XPath is a language used to navigate through an XML document, and is used when writing XSLT. A DOCX file is a zipped file containing many XML files that make up the document.

The tool was written in Python, which acted as a wrapper to run other commands. This script, `messages.py`, was made to accept arguments such as the name of the IP block, version number, and the path of an existing user guide. The script would then generate a customised XSLT file that would generate the register tables of the provided IP block. This XSLT was applied on the source XML to produce an XML file that resembles that of what you would find inside a DOCX zip. The script then inserts this XML file into an extracted DOCX to create the completed document.

I also created a "How To" guide on the company Wiki to explain how to use the tool and how it works to help with maintenance after I complete my placement.

3.2.3 Evaluation and Skills Learned

This tool reduces the amount of errors in the documentation, the amount of ensuing customer issues (16 of which have already occurred this year due to inaccurate documentation), and the time wasted identifying and correcting these errors.

This project has taught me about manipulating XML files using XSLT, and how to navigate through XML files with XPath.

Chapter 4

Verification of a Module

4.1 The Problem

One of UltraSoC's IP blocks, the Message Lock, was defined in RTL but was not verified. The verification of IP is done using UVM, and my task was to verify this module and achieve 100% coverage. Coverage is an indicator to prove that the section of code in the IP was exercised.

4.2 The Solution

UVM is a highly modular framework. It makes use of the object-oriented nature of SystemVerilog by creating many different components that are inherited from each other [3] [4]. These components make up the testing environment, for example uvm_drivers can send the stimulus to the IP block, uvm_monitors can observe the IP block outputs, uvm_predictors take in the stimulus and determine what the IP block should output, and the uvm_scoreboard compares the uvm_predictor output with the IP block's output to see if the IP block is working correctly. The UVM environment was created to test the Message Lock, and several different tests were made to exercise all of the features of the block.

A standard verification flow was followed. The module specification was analysed in a program called vManager. This involved annotating all of the module's features and identifying the tests that should be made to verify that these features work.

Tests were made to exercise these features, and coverage was collected. The generated coverage files were then linked back to the highlighted parts of the specification in a process known as back-annotation.

4.3 Evaluation and Skills Learned

This task vastly improved my knowledge of UVM and how to use it to verify a piece of IP. It also heavily built on the SystemVerilog learned in digital electronics modules and the object-oriented style of programming learned throughout the course. Lastly, this gave me an introduction to the roles and responsibilities of a Verification Engineer.

Chapter 5

Release Flow GUI

5.1 The Problem

When an IP block has been developed to a stage where it is ready to be added to the list of modules that can be released to customers, a snapshot of the block and any other related code or IP has to be isolated from the main working branch of the version controlled repository using a tag [5]. Tagging an IP block is a way of recording which version of the repository corresponded with the version sent to the customer. This process of releasing a module was a manual, time consuming, and error prone process; it required the running of several scripts, remembering to follow a certain naming convention, and the updating of XML files.

When IP blocks are being prepared to be sent to a customer, an XML file `cfg.xml` is prepared which outlines which blocks are being sent, their properties and version numbers, any supporting files, and any other related information. This file along with the bill of materials was made by hand and is another error prone process.

Both of these tasks are menial and had the potential to be automated to save time.

5.2 The Solution

5.2.1 Module Releases

The proposed solution was a web based GUI that would include forms for data input and back end code that would handle the running of scripts and XML file editing. The GUI would communicate with Jenkins, an automation server, which would handle the creation of the releases. Jenkins was chosen as it was already set up within the company and was being used by the software team to automate their processes.

The web GUI was written in Python Django, a high level web framework that allows you to quickly develop web pages in Python. I chose Python Django over other web frameworks as I had used this framework at UltraSoC in a previous work placement and was familiar with its structure. Many existing UltraSoC web pages have also been made using Django, therefore proceeding with this framework would save time and make deployment and future maintenance easier.

The GUI was linked to a MySQL database which held all of the information regarding each release. This database mirrored what was present in the release area of the engineering team's SVN repository. MySQL database calls are much faster than carrying out `svn` commands, which is why it was decided that the GUI would have its own record of what modules have been released in a MySQL database to reduce page loading times. This database was updated using a Python script that would be automatically called using an SVN hook. This hook would run a script every time a change was committed to the SVN repository. The script, `database_updater.py`, would compare the revision number of the SVN repository against its own revision number and update the MySQL database as necessary.

A Python-Jenkins add-on module was used to communicate with the Jenkins server to build new releases. The Jenkins server would then run several Python scripts that I created. These scripts would access the variables provided by the user when the GUI's form was filled and perform the necessary actions to release the module.

5.2.2 Customer Releases

Creating another page in the GUI to handle complete customer releases involved reusing a lot of the code used to create the page for module releases which saved time. The creation of `cfg.xml` required making many forms to account for all of the different options that go into making this file. Once the forms have been submitted by the user, the Python code combines all of the information into a Python dictionary and uses the Jinja Python package to create `cfg.xml`. Jinja is a templating tool which lets you pass information to a template file which then generates the completed file with all the information placed where it is specified. Jinja was chosen as it was used in the past by my line manager who will be maintaining the GUI after my placement. The syntax in Jinja is also based on the Django templating language which I was already familiar with. By making a `cfg.jinja.xml` template and giving it the information provided by the user from the forms, the GUI produces `cfg.xml` and saves it in a shared area.

There was scope in this project to add many other checks to the `cfg.xml` file that were beyond the time scale of my placement. To help with future development, I clearly signposted within the code the areas in which development can happen.

The accompanying Bill of Materials spreadsheet was made in a similar fashion. By creating a template Excel file and parsing this using a Python module named Openpyxl, the information was written to a copy of this template and saved to the same shared area as `cfg.xml`.

5.3 Evaluation and Skills Learned

Extensive knowledge was gained on code version control, creating a web framework, HTML template generating, CSS for GUI design, SQL database management, and the use of Continuous Integration tools such as Jenkins.

The GUI will save time when preparing a module for a release, and make it easier for the company's Applications Engineers to view all of the different releases for each IP block.

Chapter 6

Integration Verification Flow

Once IP blocks have been designed, they can be connected to other blocks to form a sub-system. These sub-systems are tested to check if IP blocks can correctly interact with other blocks. Unlike single IP blocks where a UVM environment is used to perform the testing, sub-systems are tested using Python scripts which generate the IP block's Verilog code and connect the block's signals to signals found on other blocks and those found in the generated test bench. A test bench is the top level piece of code that instantiates the other IP blocks.

6.1 The Problem

The stimulus for sub-system testing is currently done through a complicated piece of "back-door" Verilog that edits the ROM that contains the transactions that make up the stimulus. This method is complicated and is hard to configure. My task is to replace this with a replayer module that can read in a stimulus in the form of a CSV file, and produce this on the message protocol for the sub-system components to detect.

6.2 The Solution

To obtain the CSV stimulus in the first place, the UVM environment that solely tested one of the sub-system components was edited to record the randomly generated stimulus in a CSV file. This CSV was then placed in the sub-system test bench area.

The replayer module was coded in SystemVerilog, and was designed to work with an AXI protocol [6] which is widely used to communicate between sections of an SoC. The python scripts that control the sub-system testing were edited to replace the “back-door” code with the replayer, and ensure that all of the signals are connected.

After testing that the replayer worked as intended, it was developed further so that it could act as an AXI master. This involved adding extra code so that if any of the bits corresponding to the input ”Ready” signals in the next line of the stimulus are set high, the replayer would only replay these signals once it receives a ”Ready” signal from the slave that is connected to it. Given that in the sub-system the connected slave may respond a few clock cycles before or after what is expected from it in the stimulus CSV file, the replayer had to be able to pause replaying if the slave was late to respond and skip duplicate lines in the stimulus in cases where the slave responded sooner than expected. This was done to better emulate a real sub-system.

6.3 Evaluation and Skills Learned

This AXI replayer and the AXI master version of the replayer both provide a simpler way to provide a stimulus for a sub-system test. The UVM environment can easily produce complex random stimuli, and the use of a CSV means that the sub-system can be separated from the UVM environment which helps with portability. It also improves the support we can provide to our customers who find it easier to create a CSV stimulus over setting up a UVM environment.

The task has taught me about module design using SystemVerilog which has been practised throughout the course. I have also learnt about file manipulation within SystemVerilog, and using Python scripts to generate RTL.

Chapter 7

Improved Traffic Generation

The Bus Monitor is an UltraSoC module that provides a wide variety of functions. It is a passive module that monitors traffic on SoC bus interfaces. One of the features of the Bus Monitor is that it can count a particular metric, for example it can count the number of times a write transaction on the bus occurs for a memory address that falls within a pre-configured range of addresses. When this feature is tested during the verification stage, the range of memory addresses is randomly determined at the start of the test. The UVM environment that contains an instance of the Bus Monitor also contains a traffic generator that produces transactions for the Bus Monitor to read. A UVM sequence is an object that configures the transactions before they get sent to the traffic generator to be put on the bus interface.

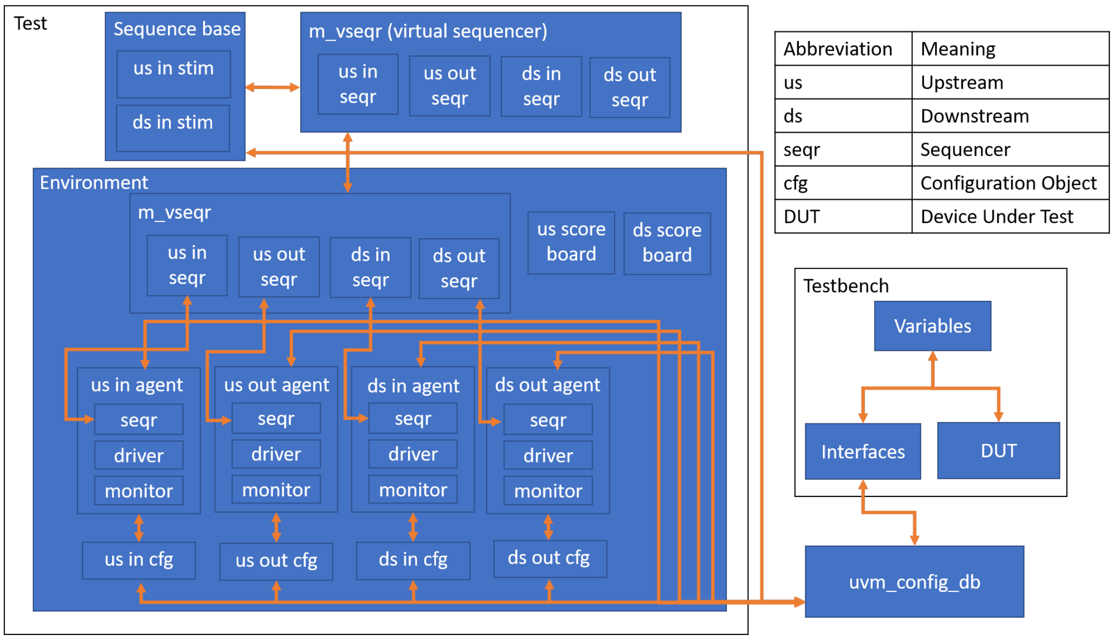
7.1 The Problem

In cases where the randomly determined range of addresses for the Bus Monitor to check for is a narrow range, it will take a very large number of transactions until one of them happens to fall within this address range and increment the counter. To reduce the test time and the number of tests required to verify the Bus Monitor and achieve 100% coverage, it makes sense to give the UVM sequence that is creating the transactions knowledge of how the Bus Monitor is configured at the start of the test and use this knowledge to create transactions that are more likely to fall within the criteria.

Tests have been written to test the Bus Monitor with an AXI bus interface and an OCP bus interface, my improvements to the traffic generator will have to work for both cases.

7.2 The Solution

Whilst this problem sounds simple, in practice it was complex. UVM is a highly modular framework with very specialised components that make up the testing environment.



Abbreviation	Meaning
us	Upstream
ds	Downstream
seqr	Sequencer
cfg	Configuration Object
DUT	Device Under Test

FIGURE 7.1: A block diagram showing all of the different UVM components that make up the verification environment. The Bus Monitor has an upstream channel and downstream channel, with each channel having an input and output. The transactions are determined in the Sequence Base block and are propagated through the virtual sequencer within the Test, to the virtual sequencer in the Environment, to the respective Agent, then to the Agent's Configuration Object which is then finally linked to the interface connected to the DUT (Device Under Test) via the uvm_config_db.

The test file was first edited to store the randomised address ranges for each interface in a structure. A structure is a variable type that contains several variable types within it. The goal was to pass this structure to the `uvm_sequence` so that it knew how the bus monitor was configured in this test and create transactions accordingly. This structure was then stored in the `uvm_config_db`, a configuration database which is globally accessible by any UVM component in the environment. A `uvm_sequencer` manages `uvm_sequence` objects, it retrieves the structure from the `uvm_config_db` and copies the information into a configuration object that is passed down the hierarchy to the `uvm_sequence`. Once the `uvm_sequence` has this information it then randomises an instance of the transaction with the added constraints taken from the configuration object and then passes this transaction to the traffic generator to put on the bus interface. Once this was successful, other metrics besides the transaction address such as the ID number were included so that more of the metrics could be tested.

7.3 Evaluation and Skills Learned

These changes to the way transactions are generated in Bus Monitor tests will reduce the amount of test runs required to achieve 100% coverage and save time. The task taught me the intricacies of sending information throughout a UVM environment and how constrained random testing can is used to verify a piece of IP.

Chapter 8

FPGA Regression Task

FPGA boards are useful for quickly synthesizing RTL designs in physical hardware. FPGAs are made up of a matrix of configurable logic blocks that are connected by programmable interconnects [7]. By programming the interconnects we can create designs in hardware on the fly.

8.1 The Problem

Software simulations of RTL designs in verification environments using UVM are performance limited by many factors such as the simulator and the machine that is running the simulation. A physically implemented design does not have these restrictions, and tests can run several orders of magnitude faster on an FPGA as the test is running on actual silicon rather than a software model. The rationale for testing designs on an FPGA is clear, tests can be run quicker which can help to uncover bugs and corner cases within a shorter amount of time. My final task in this industrial placement was to make something that could implement this.

8.2 The Solution

The proposed solution was to create an automatic FPGA build and regression testing environment. There are several FPGA images that already existed within the company, each image contains a selection of UltraSoC IP along with any other infrastructure. For example there is an image that demonstrates some IP working alongside an ARM core. This build and test environment would randomly select one of these images, synthesize the latest versions of UltraSoC IP that make up the image onto an FPGA and then run a series of tests that correspond to the UltraSoC IP blocks present in the image to check that they work.

The FPGA development board that was used contained an FPGA, two ARM cores, an SD card that contains the software to be run on an ARM core and the FPGA image, and a display. The board also contained two serial port connections, one is used to communicate to the board from a host and load images and the other is used to communicate with a USB communicator that identifies which UltraSoC IP is present in the system. Lastly, a JTAG connection was also present and is used to communicate to the FPGA directly.

FPGA synthesis tools, such as Xilinx Vivado, synthesize IP into `.bit` files. The file format for loading an image directly onto an FPGA over a JTAG interface is the `.elf` format. The Software Team had already made a tool that takes a `.bit` file and creates a `.serial-ui.boot.bin` file which contains the FPGA image and the software that would run on an ARM core. This tool also makes an `.elf` file as a by product. This `.serial-ui.boot.bin` file would then be loaded onto the development board's SD card as `boot.bin`. On power up the board would read the file `boot.bin`, load the contained image onto the FPGA, and run the contained software on the ARM core.

Several milestones were decided upon to track progress:

- Schedule the creation of a randomly selected image's `.bit` file each night.
- Build the accompanying software.
- Load the FPGA image and software onto the development board.
- Run appropriate tests on the development board. These tests have to be self checking and test the basic functionality of the UltraSoC IP blocks present in the FPGA image.

Jenkins was chosen as the tool to create this system. I had prior knowledge of Jenkins in previous tasks and its structure makes it suitable for this task. I created five Jenkins jobs that were linked in a chain, the completion of the first job would trigger the next job. The jobs were named:

- fpga_bit_build

- fpga_image_build
- fpga_image_test
- fpga_image_load
- fpga_run_test

The first job `fpga_bit_build` used Bash shell scripts to run the Xilinx Vivado tools which created the `.bit` file. I used Jenkins artefacts to pass the `.bit` file to the next job so that it could be accessed.

The second job `fpga_image_build` accesses the produced `.bit` file that was provided from the previous job and runs the make command to build the software. It produces a `serial-ui.boot.bin` file and an `.elf` file which are also stored as Jenkins artefacts ready to be used by the proceeding job.

The third job `fpga_image_test` runs an initial test where the `.elf` file from the previous job is loaded directly onto the FPGA via a JTAG interface. Then a script I made is run which communicates to the FPGA over a serial connection and runs a simple command which should return the image name and version number. If this passes, then this build is ready for actual testing and the next job is started. If it fails and nothing is returned, this implies that the image is corrupted and that this image made from the latest versions of the IP needs fixing. The board is restarted by running a Bash script that controls the power to the board. This power cycles the board, causing it to boot from the SD card which at this point still contains the old working FPGA image. This job is necessary because if the image was loaded directly onto the SD card and the image turned out to be corrupted, the only way to restore a working image onto the SD card would be to physically remove the card from the board and use a computer to access its files. The development board is located in a server room and in the future where there would be many boards running tests, this manual approach is not feasible. Therefore, by loading the `.elf` file directly onto the FPGA and testing it works is a better solution, as the original working image is on the SD card and ready to be used if the new image does not respond.

The fourth job `fpga_image_load` loads the `serial-ui.boot.bin` file from the `fpga_image_build` job onto the board's SD card over a TFTP connection. The board is then restarted by sending a command to it over a serial connection. On power up, the board loads up the new image from the SD card and then it is ready to run tests.

The fifth and final job `fpga_run_test` is where the IP blocks in the FPGA are tested. This job runs a Python script I created which reads a YAML file that contains information about what tests should run on what FPGA image. This YAML file is hosted on an SVN repository so that anyone within the company can add new tests to the FPGA images. The Python script then opens an instance of connection-utilities, which is a program created by the Software Team to act as a link between a host and the FPGA board, and uses the built in house tool PyLink to run each test. One of the tests is General Discovery, which communicates with the board to identify all of the UltraSoC IP blocks that are present. This is the first test which is run and checks to see if all of the IP blocks that should be present actually respond.

I created several tests for various IP blocks during the final weeks of my placement. I used a consistent structure in the tests which should make it easier to create more tests after my placement. I also created a page in the company's internal Wiki to explain the system and the Jenkins jobs.

8.3 Evaluation and Skills Learned

The solution works and met all set criteria, and there is scope for more boards to be added to the system and for more tests to be included. Running these tests also indirectly tests the software connection-utilities which handles the host-to-chip connection. This is a real customer use case, where customers connect their PC to UltraSoC IP on a chip to run tests. This FPGA regression system mimics what customers would be doing in the field.

From this task I further developed my knowledge of using Jenkins to automate tasks. I also learnt about FPGA development boards, using synthesis and debug tools such as Xilinx's Vivdado tool and their SDK software, and communicating with these boards via serial and JTAG connections.

Chapter 9

Project Management

To keep track of my progress throughout my placement, I used a logbook and a page on the company's Wiki to record notes. During my weekly meetings with my Line Manager Melvin I explained my progress and outlined future steps. Melvin kept a rough schedule of my tasks and how long they should take to complete, the durations were rough estimates based on delivery time. I have adapted and added to this schedule in Appendix A.

In Appendix A we can see that many of the tasks were completed later than expected. This was due to a variety of reasons. Some of the tasks were vague either in the sense that it was unknown how long they would take to complete or in the sense that the complexity was unknown.

For example the completion of the User Guide generating task was very late. I had to make use of XLST. It is a tool that people in the company were not familiar with which led to a conservative estimate on the duration. The task turned out to be more complex than anticipated and many hours were spent creating an XSLT file that satisfied all of the possible options when generating the document.

Another factor which wasn't considered in the time estimates is the time spent on maintaining previous tools from previously completed tasks. I spent a lot of time fixing parts of the regression system throughout the placement as well as adding new features to completed scripts and GUIs based on user feedback. This used up time that would have been spent on completing current tasks.

Lastly, I also completed several minor tasks during the placement. These tasks took a day or two to complete and were usually to do with fixing or creating a script to do something in response to someone within the Hardware Team openly asking the team about a certain problem. As a placement student who's deadlines were not as severe as my colleague's, I regularly put myself forward to solve these problems. Whilst this did use up time that I could have spent on my projects, the time was relatively small and the benefit gained from completing these minor tasks outweighed this.

Despite many projects being completed late, my Line Manager was happy with my progress and given the amount of work that was done to complete these projects he was happy with the amount of projects that I completed during my placement.

Chapter 10

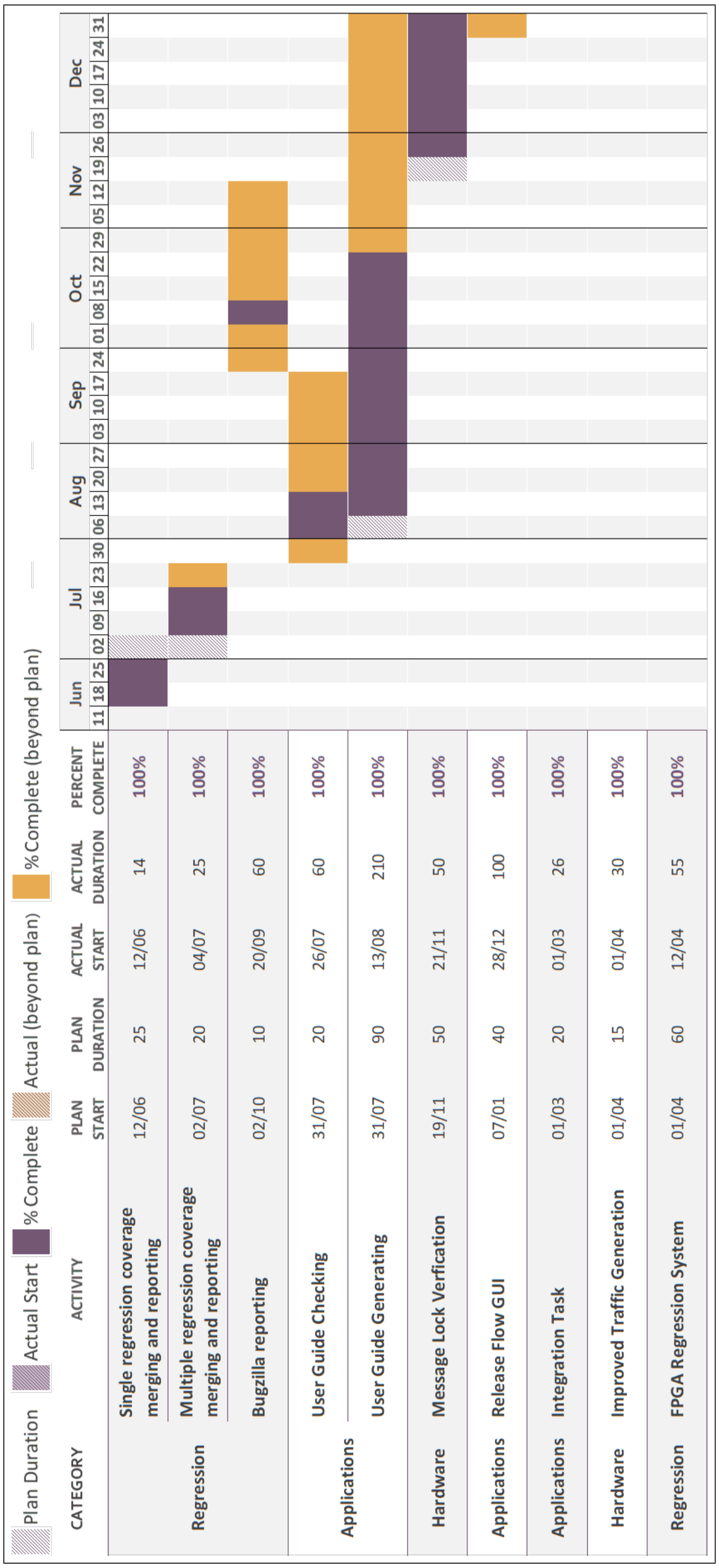
Conclusions

During the past year at UltraSoC I have undertaken a variety of projects. I have developed my knowledge in many areas including scripting, web development, FPGA testing, hardware design, and hardware verification. I have applied many of the skills learnt at university during my placement such as programming and researching solutions. I have managed to create many new scripts, tools, GUIs, and automated Jenkins jobs which will improve the work flows of my colleagues, reduce the amount of errors, and save time. Having completed this placement I feel that I am better prepared to work in the electronics industry after graduating and I look forward to returning to university to complete my final year.

Appendix A

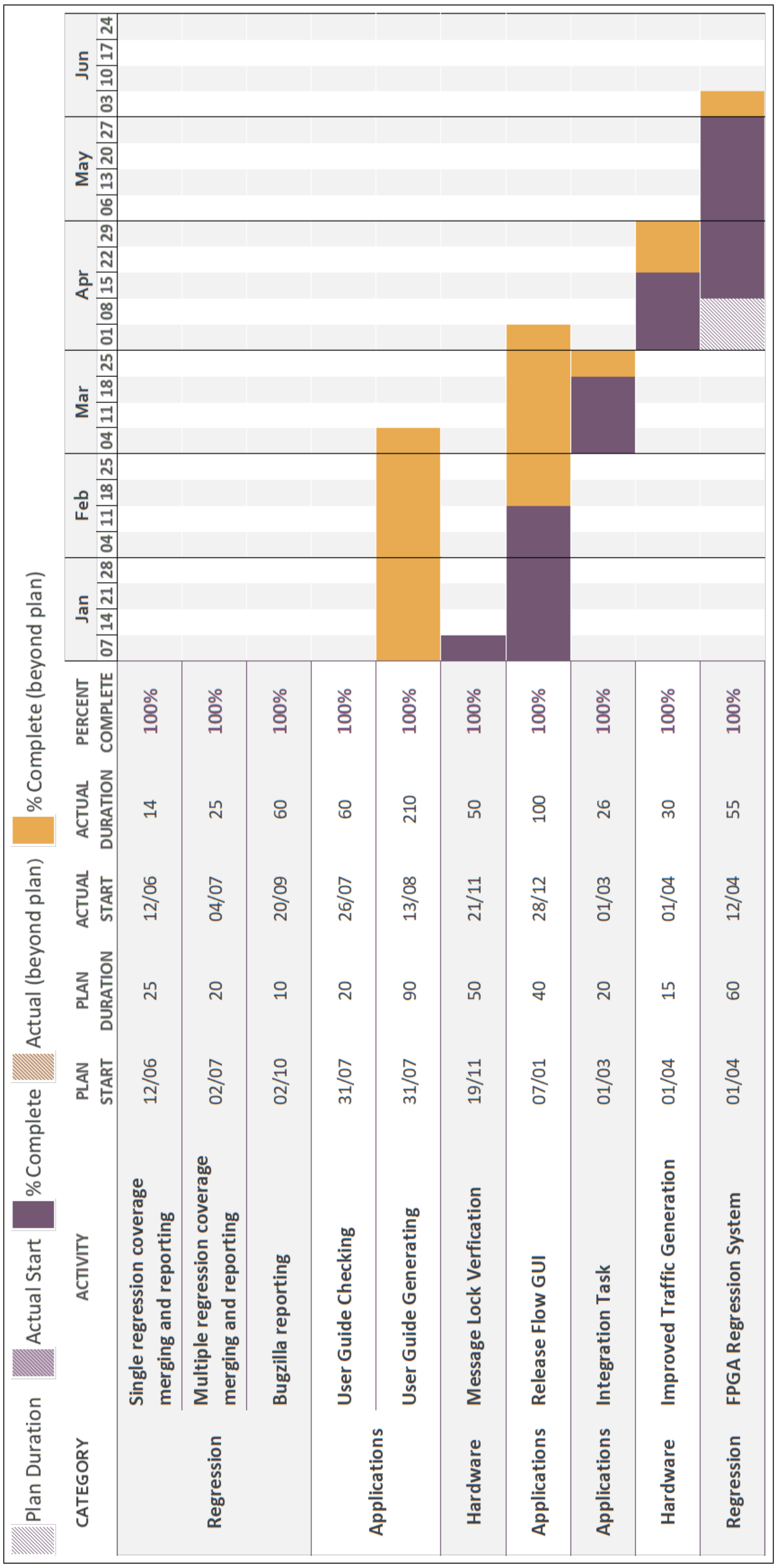
Gantt Chart

FIGURE A.1: A Gantt chart showing the progress of projects from June to December 2018.



CATEGORY	ACTIVITY	PLAN START	PLAN DURATION	ACTUAL START	ACTUAL DURATION	PERCENT COMPLETE
Regression	Single regression coverage merging and reporting	12/06	25	12/06	14	100%
	Multiple regression coverage merging and reporting	02/07	20	04/07	25	100%
	Bugzilla reporting	02/10	10	20/09	60	100%
Applications	User Guide Checking	31/07	20	26/07	60	100%
	User Guide Generating	31/07	90	13/08	210	100%
Hardware	Message Lock Verfication	19/11	50	21/11	50	100%
Applications	Release Flow GUI	07/01	40	28/12	100	100%
Applications	Integration Task	01/03	20	01/03	26	100%
Hardware	Improved Traffic Generation	01/04	15	01/04	30	100%
Regression	FPGA Regression System	01/04	60	12/04	55	100%

Figure A.2: A Gantt chart showing the progress of projects from January to June 2019.



CATEGORY	ACTIVITY	PLAN START	PLAN DURATION	ACTUAL START	ACTUAL DURATION	PERCENT COMPLETE
Regression	Single regression coverage merging and reporting	12/06	25	12/06	14	100%
	Multiple regression coverage merging and reporting	02/07	20	04/07	25	100%
	Bugzilla reporting	02/10	10	20/09	60	100%
Applications	User Guide Checking	31/07	20	26/07	60	100%
	User Guide Generating	31/07	90	13/08	210	100%
Hardware	Message Lock Verfication	19/11	50	21/11	50	100%
Applications	Release Flow GUI	07/01	40	28/12	100	100%
Applications	Integration Task	01/03	20	01/03	26	100%
Hardware	Improved Traffic Generation	01/04	15	01/04	30	100%
Regression	FPGA Regression System	01/04	60	12/04	55	100%

Bibliography

[1] H. Foster, A. Krolnik, and D. Lacey, “Functional Coverage,” in *Assertion-Based Design*. Boston, MA: Springer US, 2003, pp. 123–159. [Online]. Available: http://link.springer.com/10.1007/978-1-4419-9228-4{_}5

[2] “What is REST Learn to create timeless RESTful APIs.” [Online]. Available: https://restfulapi.net/

[3] “TestbenchBasics — Verification Academy.” [Online]. Available: https://verificationacademy.com/cookbook/testbenchbasics

[4] “UVM Introduction - Verification Guide.” [Online]. Available: https://www.verificationguide.com/p/uvm-introduction.html

[5] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato, “Version Control with Subversion For Subversion 1.7 (Compiled from r5930),” Tech. Rep., 2002. [Online]. Available: http://svnbook.red-bean.com/en/1.7/svn-book.pdf

[6] “AMBA ® AXI and ACE Protocol Specification AXI3 , AXI4 , and AXI4-Lite ACE and ACE-Lite ,” Tech. Rep., 2003. [Online]. Available: http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf

[7] “What is an FPGA?” [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html