Tampere University

Kalle Jyrä

AUTOMATED REGISTER GENERATION FROM IP-XACT

Generating VHDL file from IP-XACT formatted XML using Python3 and its' libraries

Master's thesis
Faculty of Information Technology and Communication Sciences
Examiners: Timo D. Hämäläinen
Esko Pekkarinen
April 2021

ABSTRACT

Kalle Jyrä: Automated Register Generation from IP-XACT
Master's thesis
Tampere University
Master's Degree Programme in Embedded Systems
April 2021

The System on Chip (SoC) field relies on tools and processes. Tools are used for design, automation and verification purposes. Automation tools do repetitive tasks and one of these tasks would be generation of register banks. Register banks are fast memory of SoC modules and each SoC module requires their own set of registers to suit module's needs. Register bank generation tools enable developers to quickly generate registers for a SoC module according to a register configuration file.

In this thesis project there is an existing register generator tool flow that uses IP-XACT as intermediate file format. This IP-XACT file is interpreted and VHDL is generated based on its' data. The previously developed tool that does the conversion has become very fragile to changes and its' programming language differs from the rest of the flow. This thesis describes the update to the IP-XACT to VHDL flow. A database is planned to be used as an intermediate format between IP-XACT and VHDL. A database approach would enable easy modifications, big and small, and the database is constructed in such a way that IP-XACT can be regenerated from it.

Tool was developed for the flow and is available alongside the old implementation. The development was successful since the tool is able to transfer the information from IP-XACT file to database losslessly and then generate a VHDL file according to a register description from the IP-XACT file stored in the database. The tool has gone through verification and continues to be in development for extended functionality. Otherwise, the tool is in support mode and provides its' intermediate database for other flows.

Keywords: IP-XACT, VHDL, Python, SQLite, automation

TIIVISTELMÄ

Kalle Jyrä: Rekisterien Automatisoitu Luonti IP-XACT Formaatista
Diplmityö
Tampereen yliopisto
Maisterin Tutkinto Sulautetut Järjestelmät
Huhtikuu 2021

Järjestelmäpiirien ala on riippuvainen työkaluista ja prosesseista. Työkaluja käytetään suunnitteluun, automaatioon ja verifiointiin. Automaatiotyökalut tekevät toistuvia tehtäviä, jotka pystytään automatisoimaan. Yksi näistä automatisoitavista tehtävistä on rekisteripankkien luominen. Rekisterit ovat SoC moduulien nopeaa ajonaikaista muistia, missä rekisteripankki toimii alimoduulina (engl. submodule) SoC moduulin sisällä. Rekisteripankkien luontityökalut antavat kehittäjille mahdollisuuden nopeasti luoda sopivat rekisterikuvaustiedoston mukaiset rekisterit SoC moduulille.

Tässä diplomityön projektissa on työn alla rekisterien luontityökalun vuo, joka käyttää IP-XACT tiedostoa välimuotona. IP-XACT tiedosto tulkitaan ja VHDL tiedosto luodaan sen datan perusteella. Tämän diplomityöprojektin tavoitteena on päivittää vuota IP-XACT tiedostosta VHDL tiedostoon. Vuon päivityksen motivointina on, että IP, joka aikasemmin on kehitetty ja tekee muunnoksen, on alkanut olemaan herkkä muutoksille ja sen ohjelmointikieli on eri kuin kieli, jota muualla vuossa käytetään. Tarkoitus on myös käyttää tietokantaa välimuotona IP-XACT tiedoston ja VHDL tiedoston välillä. Tietokanta lähestymistapa mahdollistaisi helpot muutokset, pienet ja suuret, ja tietokanta olisi myös rakennettu niin, että IP-XACT voidaan myös luoda uudelleen.

Työkalu kehitettiin vuolle ja se on saatavilla vanhemman toteutuksen ohella. Kehitys oli onnistunut, sillä kehitetty työkalu kykenee luomaan VHDL tiedoston rekisterikuvaustiedoston mukaan. Työkalu on käynyt verifikaatiossa ja kehitys harvemmin käytetyille toiminnoille jatkuu. Muuten työkalu on tukitilassa ja tarjoaa sen tietokantavälimuotoa muille työkaluille.

Avainsanat: IP-XACT, VHDL, Python, SQLite, automatisointi

PREFACE

I would like to thank my company, especially my colleagues, employer and supervisor, for giving me this opportunity. Along with this thesis I have found a workplace that, I hope, will suit me for a long time. I would also like to thank my family, girlfriend and friends. From them I have gotten the support when there has been a lack of content or fatigue. Attending university has been a long journey and writing this feels like the end is finally in sight. Thankfully I have made new friends and built a happy life for myself along the way.

Tampere, 5th April 2021

Kalle Jyrä

CONTENTS

LIST OF FIGURES

LIST OF PROGRAMS AND ALGORITHMS

LIST OF SYMBOLS AND ABBREVIATIONS

ASIC	Application Specific Integrated Circuit
CLI	Command Line Interface
CPU	Central Processing Unit
DTD	Document Type Definition
DUT	Design Under Test
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IP-XACT	IP-XACT is an XML format specifically to decribe electronic circuit designs.
IC	Integrated Circuit
IP	Intellectual Property
Perl	Perl is an interpretted high-level programming language.
Python	Python is an interpretted high-level programming language.
RegEx	Regular Expression
RTL	Register Transfer Level
SQLite	SQLite is a program which uses SQL language to create local databases rather than client-server databases.
SQL	Structured Query Language is a language used to send queries to create and manage databases.
SoC	System on Chip is an integrated circuit or the field devoted to developing integrated circuits.
SGML	Standard Generalized Markup Language
SML	Service Modelin Language
ICU	Interconnect interface unit gives common interface protocol between SoC's modules.
VENV	Virtual ENVironment is an isolated environment from the host machine.
VHDL	Very High Speed Integrated Circuit Hardware Description Language which is a programming language used to describe electronic hardware.

VLNV	VLNV is an acronym for Vendor, Library, Name, Version. VLNV is a string of information used identify component by its' vendor, library, name and version.
W3C	World Wide Web Consortium
XML	eXtensible Markup Language is a set of rules for data to be encoded in specific format.
XSD	XML Schema Definition

1 INTRODUCTION

In the development of electronic devices an important factor to company's profitability is their time to market. The goal is to have a product finished and on the market before one's competitors. To increase efficiency the same outcome is wanted in minimal time and automate tasks as much as possible. With automated generation tools hundreds of work hours can be used for other important tasks rather than in those which could be automated.

This thesis shows how automated tools can be customized for company's needs and how standardized formats are used to ease design flows. In the project associated with this thesis an automation tool is developed which generates a VHDL file for a register bank from a previously generated IP-XACT standard XML file. This tool is then distributed in a form which is easily usable.

To generate a VHDL file, Python scripts and SQLite databases are used. Figure 1.1 shows the tool flow currently in place. Register generator tool is used to automatically generate VHDL for a register bank based on a register description file. Excel, IP-XACT XML, info and VHDL are either input, output or intermediate files. IP-XACT as a vital part of the flow is an XML standard used to store information about a design. Excel2xml, xml2info and info2vhdl are tools between the file formats. The goal of this thesis is to improve this flow. The objective of this project is to update current tool, convert it to widely used and easily maintainable programming language and have easily accessible and readable intermediate file formats.

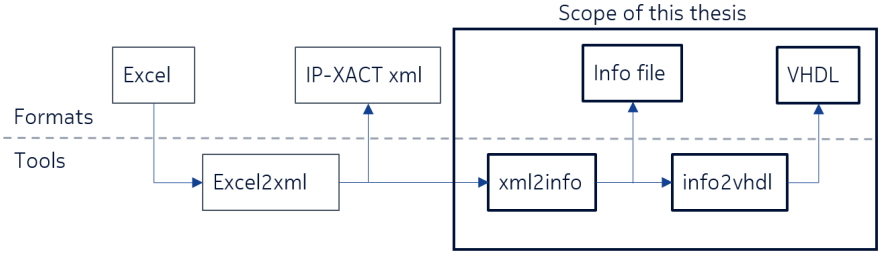

Figure 1.1. *Flow of the current register generator tool.*

The research questions in this thesis are:

- How register information is identified from IP-XACT standard XML data and be converted to hardware description language?
- How to make an easy to use command line automation too which can be used from any Linux environment?
- How to keep intermediate file formats in easily readable and modifiable format?

In chapters through 2 to 6 the content is as follows: First in the Chapter 2 tools, standards and work flows related to the project associated with this thesis are gone through. Then in the Chapter 3 problems with current implementation are analyzed, how it could be improved and what are the possible solutions to the research problems. The developed tool, its' workflow and usage is showcased in Chapter 4. In the Chapter 5 the success of this thesis project is analyzed, what kind of information and thoughts this thesis brought up and what are the steps for future development. In the final Chapter 6 the most important points of the thesis are reviewed and the thesis is summarized.

This thesis is a good read for engineers and students in hardware development or verification, especially those in system on chip field. Although, readers from other fields are most welcome too. For readers from other fields the concepts might be slightly hard to grasp without previous experience. To follow and understand this thesis work the reader should be familiar with intermediate level Python scripting, have basic understanding of SQLite or SQL databases, know VHDL at intermediate level and have basic knowledge of the XML data format.

2 SYSTEM ON CHIP AUTOMATION TOOL DEVELOPMENT

In this chapter related topics and theory are gone through. The chapter explains the contents which are relevant in the scope of this thesis and what kind of technologies are used in automation tool development. First the field to which the tool is developed is introduced and then a little insight is given to the used technologies.

2.1 System on Chip

System on Chip (SoC) is a chip which has multiple different components on same silicon. These components can be processing units, memories or other functions. These SoCs are Integrated Circuits (ICs) but in larger scale and SoCs are usually considered to have more functionality where ICs usually are considered to have one specialized function. [1]

The key is the combination of software and hardware. Hardware is very fast and has low power consumption but is not flexible, adaptable and is hard design and test. Software is very flexible, adaptable, easy to write and test but also slow and has high power consumption. This means that with programmability we lose performance and with performance we lose adaptivity. SoCs are a good middle ground when both programmability and performance are wanted. [2]

SoCs communicate internally with interconnects and externally with communication protocols. For example, an SoC for smart phone would need a communication protocol to be able to communicate with peripheral devices. These devices can be cameras, screens or other chips. Communication protocols can be used internally as well between the different components of the SoC. Figure 2.1 shows an example of a high level representation of an SoC. The figure shows possible modules inside an SoC and how they are connected with an interconnect. [2]

SoCs are becoming more complex and contain more functionality. Complexity of design means more transistors but short time to market forces designers to use transistors less efficiently. The efficient usage of transistors comes from optimization for which the designers do not have enough time. One way to battle this is the design re-usability where same design would be used for multiple applications. These designs usually leave room for some customization in architectural parameters. [3]

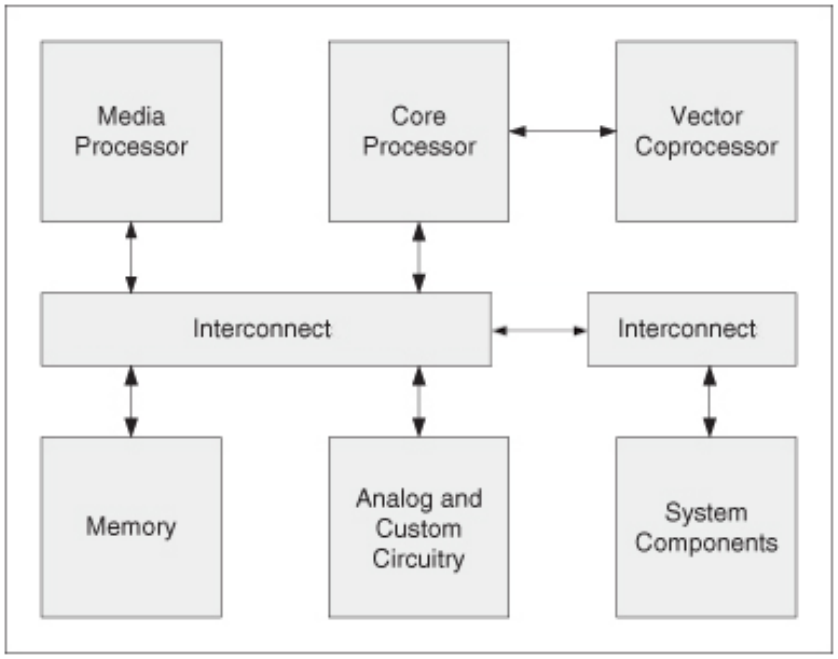

Figure 2.1. *High level design architecture of SoC chips. [2, modified from Figure 1.5]*

Typical design flow for SoCs begins from application's requirements and end in IC tape-out. This design flow in whole can be seen in Figure 2.2. The figure also shows the hierarchy level of a phase meaning an item further in the list is more complicated representation of the previous item. Briefly the whole flow is as follows: In functional model the inputs, the outputs and the functions that produce output are defined. Then IPs (Intellectual Property) which means abstract property owned by the company, in this case components that produce the outcome are connected together at high-level in an architectural model. The communication model defines communication methods between different IPs. The implementation model is more detailed version of the communication model as it shows the signals required for each module. RTL model is the gate-level netlist and generated from the implementation model. In RTL, registers storing information are connected together with logical elements forming the design's functionality. In the GDSII phase layering and connections are made between all the blocks resulting in a GDSII file. This file is then sent to the factory where the ICs are produced. [3]

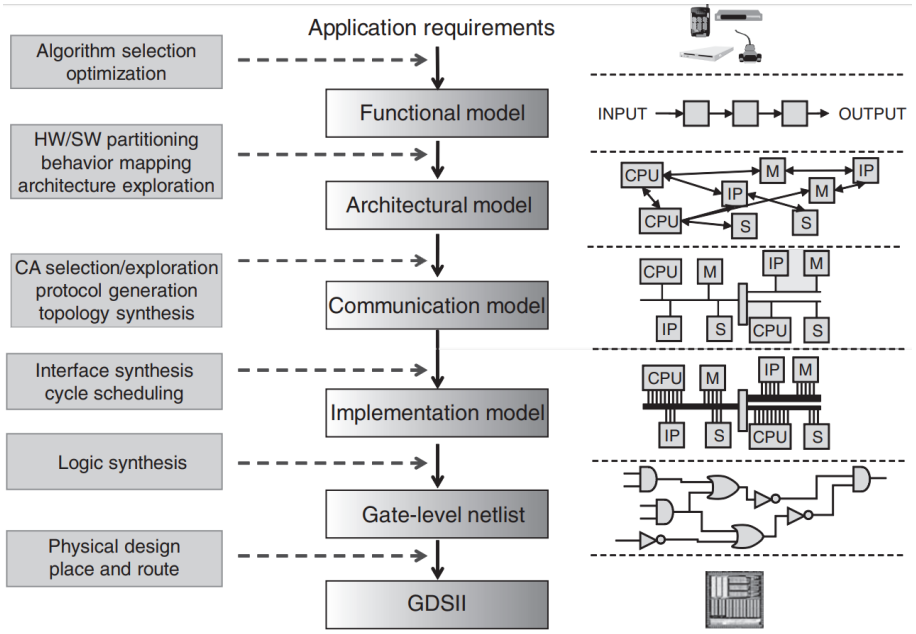

Figure 2.2. *Design flow for SoC chips. [3]*

2.1.1 Registers and Addressing

Register banks are the run time memory of the SoCs. A CPU accesses it's internal registers with read and write operations. With read and write the CPU is able to load and store values. Read and write operations are a part of instructions which are CPUs' own internal commands depending on CPUs' architecture to do certain operations. Register banks outside CPU are accessed via interconnects which follow some communication protocol. Registers are an extremely fast type of memory and are made of flip-flops [1]. Most typical flip-flops are called D-type flip-flops and their function is that the input to output route is delayed by one clock cycle.

When register banks are used with a communication protocol they belong to some address. The function of a communication protocol is to transfer data. For example CPU wants to read data from a specific address. Communication protocol tells the device which owns the address to retrieve the data and if it succeed data is read to the databus and CPU is able to access it.

Registers have either real or virtual address space. Having a virtual address space means that addressing starts from 0x0 in the register bank. In the real address space each virtual address space device has an offset where their address space start. So address of a register for the CPU would be $register\ bank's\ offset\ +\ register's\ virtual\ address$. Virtual memory is used when larger storage than the available physical memory is required. Virtual memory space is used also to protect other memory regions from unauthorized access since programs are allowed only to run in their own virtual address space. [2]

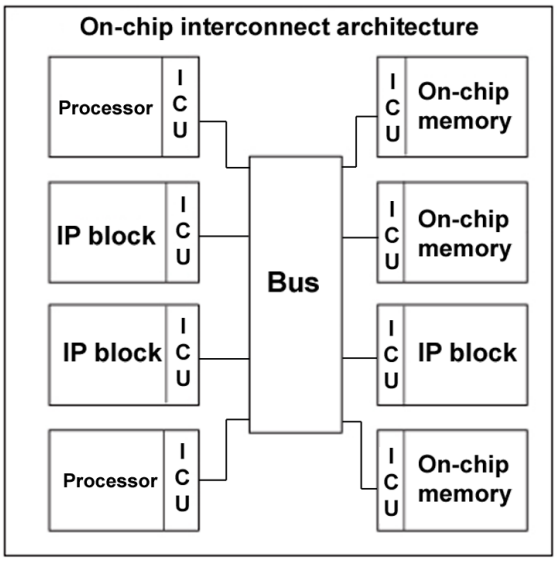

Figure 2.3. *ICUs enable communication between blocks with a protocol. [2, modified from Figure 5.1]*

Decision has to made between real and virtual memory on a communication level. All memory is still in the end physical. While virtual memory has it's advantages it is slower, more expensive and difficult to implement. The cost comes from requiring a translation unit to be implemented between the physical memory and virtual memory. Having smaller real memory can result in *all memory on-die* system which is a true system-on-chip. Where all modules that the SoC requires are on the chip. [2]

2.1.2 Communication Protocols

Communication between SoC modules happen through interconnects. These interconnects are usually a data bus which follows a communication standard, usually a bus-protocol. The bus-protocol can be a licensed protocol like AXI4-Lite from AMBA specification or an open source protocol like Wishbone. The idea is to be able to transfer data to all SoC modules or external devices in a standardized format. Bus-protocols contain data bus, addressing and status signals. One chip can have multiple busses with different protocols. If these busses communicate to each other, a bus bridge is required which does the conversion from one protocol to the other. [2]

For SoC modules to be able to access data from interconnects they may need an ICU (InterConnect interface Unit). An ICU is an submodule which provides an interface between an interconnect and SoC modules. The function of an ICU is to interpret the protocol of the interconnect, process the data and feed it to the SoC module. This way, if the interconnect protocol changes, the SoC module does not have to be modified but the ICU which is less of an effort. The location of ICUs can be seen in the Figure 2.3. If the ICU has to perform protocol translation, they are also called *bus wrappers*, but ICUs can have other functionality also [2].

2.1.3 Hardware Description

Hardware description is expressing the details and functionality of hardware in a human readable form. It would be hard if not impossible to figure out the functionality of hardware if someone were to be given a plate of silicon. HDL languages such as VHDL and Verilog provide a design entry which is the first step in logic design. In design entry the architecture and functionality of a design are described. The design entry can be compiled and simulated to verify the functionality. [4]

After design entry and simulation phases the design is synthesized into physically representable circuit [4]. In synthesis the functionality of each part of the design is replaced with a set of cells from a cell library. These cells of logical elements built from transistors are called standard cells and they are usually the de facto way of implementing logical function. This type of synthesis is done for ASIC designs but synthesis tools can also produce bitstreams for programmable devices like FPGAs. FPGAs (Field Programmable Gate Array) are circuits which can be configured by the user. In ASICs the RTL design of component is set in stone but with FPGAs the RTL design can be reconfigured with reprogramming.

After synthesis place and route, also known as physical design, is done. In this final phase of the physical design interconnects between are logical elements are made and after it is finished so is the hardware description. [4] Files from physical design phase are the final product of the hardware description. The produced files can be used to create an IC in and IC fabrication laboratory.

2.1.4 Verification

The point of verification is to prove that the IP does what it was designed to do. The design, usually referred to as Design Under Test (DUT), is verified by testers and testbenches. Signals generated by testers or testbenches are fed to the inputs of DUTs and resulting outputs are compared against a golden design which is the expected result. These testers or testbenches can be HDL simulations where signals are observed in a simulation tool. In a more mature stage of a project, where physical wafers or components are ready, testers with probes are used. These testers probe the physical input ports of the component, feed electrical signals and read results from probed output ports. [5]

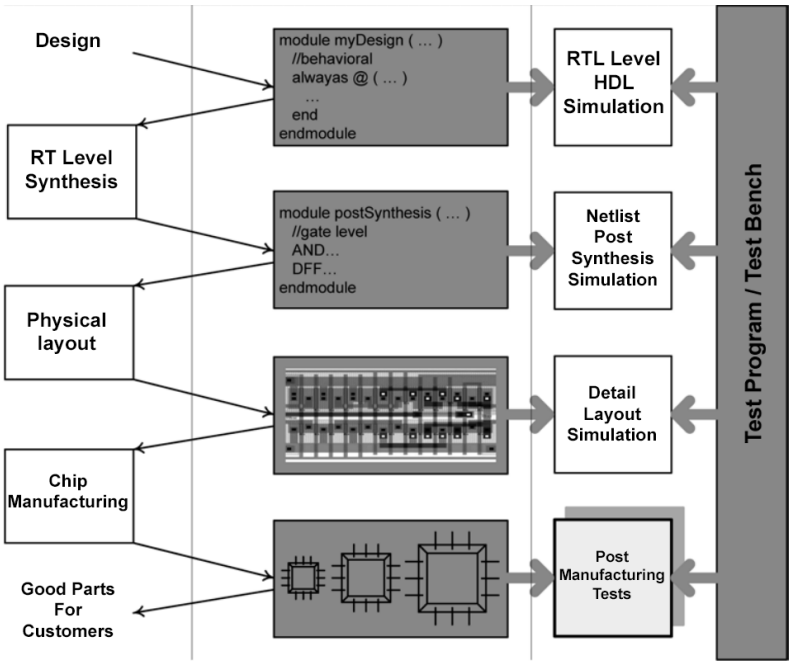

Figure 2.4. *Verification in different phases of SoC development. [5, modified from Figure 1.1]*

Even though the design is proved to work the component may not function properly. The manufacturing may result in defects, so every component must be tested individually. Defects might cause catastrophic failure which means that component does not work the way it should or does not work at all. In the case catastrophic failure the component is always discarded. In less unfortunate case defects may cause only some reduction in performance. In this case the chip can be sold as a different device at a lower price. These failures can be result of problems in the layout or masks. [5] Modifications at this stage are very costly but can improve reliability, performance and quality.

Verification has an important role in each phase of SoC development. Things that may seem good on a higher level of abstraction but may not work at all on the physical layout. In big projects each phase is done by their own team and they may have their own vision where they want to take the project. To be sure that customers receive a functioning good product testing and verification must be done in each phase. In Figure 2.4 can be seen what kind of different testing is done in each of the design phases. Each design phase is linked with a testing methodology suitable for that phase.

With the amount of transistors on chips, a throughout testing of the chip can be a challenging task. For example a chip with can have 64 inputs and outputs and 12 nanosecond internal delay. A tester can have 1 GHz operating frequency and 4 nanosecond latency to apply a new test vector to the DUT. Running a test for every test vector in this case would take approximately 11 700 years. So testing this way would simply be impossible. The amount of test vectors need to be cut down by using a set of algorithms and methods. Time can also be cut down by using another method of verification. One these is formal equivalency checking. In this method two circuits are tried to be proven to be the same by using some mathematical proof. Therefore no test data is required [5].

2.2 Python

Python is an high-level interpretable programming language. Python widely used in data science and machine learning but it also gives tools for quick application development and distribution. In the documentation it is said that the Python interpreter can also be extended with C, C++ and others. Also, the Python interpreter can be embedded to other languages. [6] Python with C extensions are usually used in very data management intensive applications. There are packages for Python which implement new vector datatypes in C which are more efficient than those in Python.

There are many implementations of Python. Most common versions are written in C and they are referred to as CPython. Python is both compiled and interpreted language. First the source code is compiled behind the scenes and then interpreted but for end-user it looks like interpreted language. Python files are regular text files but usually have a *".py"* suffix. The interpreter actually does not care about the suffix but it is there to ease users to identify Python files. [7]

If a method is not a part of the Python standard library it is a part of an imported package. These packages extend the functionality of the standard library. One downside is that often Python or package versions are depended on each other meaning package version A required Python version B and the other way around. This can result in tedious situation where updating breaks something and user needs to maintain multiple versions of the same package. This can be avoided by using virtual environments which are enclosed environments with their own set of specific versions of packages and appropriate Python interpreter. [6]

2.2.1 Virtual Environments

Python virtual environments are a way to create isolated environments where there are only a specific version of Python and packages installed. [8] This way the required packages are for sure in the environment and at the same time no changes have to be made on the host machine. On the host machine Virtual ENVironment (VENV) is just a folder which contains Python interpreter and the installed packages. Packages have to be in-

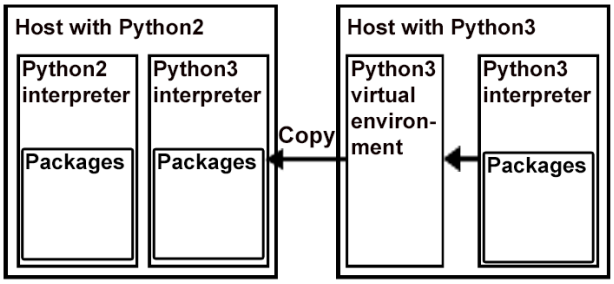

Figure 2.5. *Creating Python3 virtual environment for incompatible host.*

```
1 python3 -m venv /path/to/new/virtual/environment
```

Program 2.1. *An example of VENV creation.*

stalled to the VENV when the VENV is activated for them to actually be installed to the VENV and not to the host environment. Afterwards the VENV folder can just be deleted and recreated at will.

First a Python VENV is created with a specific version of Python and required packages. Then the VENV is activated and Python scripts can be ran with the environment that the VENV provides. Results are received wherever the VENV was activated and the Python script was ran. This is especially useful in cases where the host machine's Python version is not compatible with script to ran. Python3 and Python2 for example are not compatible so Python3 scripts cannot be ran on the host machine which has only Python2. One option would be to install Python3 on the host machine but often in companies' development environments work is done on an imaged machine on a server somewhere where the user has no privileges and installing software, although not impossible, is not trivial. The second option is to use VENV where Python3 is installed on a host machine which has only Python2. Because now the script is ran inside the VENV with Python3 no problems follow and nothing needs to be installed on the host machine. In the Figure 2.5 VENV is created with portability option and is then copied and used in other environment.

The version of Python in VENV matches the version that the VENV is created with [8]. This means that one machine has to exist with the version of Python that we want to create our VENV for. So Python3 environment cannot be created with a host machine that has only Python2. VENV has to be first created on a machine with Python3 and then transferred to the machine with only Python2.

Creating the VENV is a one line shell script which can be seen in Program 2.1. This creates VENV of the default Python version to the path defined. Afterwards VENV can be activated with multiple different interpreters including bash, csh and fish. They have their own activation scripts under the <venv>/bin/ folder. [8]

To ensure compatibility one has to ensure that the virtual environment is a truly isolated environment. Virtual environment is created as an isolated system for the host machine creating the environment. This means that to save space and time in creating the environ-

```
1 #!/bin/csh -f
2
3 # Generating a package will result in dist folder
4 # Package will be located in dist folder
5 python setup.py sdist
6 pip3 install 'dist/my_package-1.0.tar.gz'
```

Program 2.2. *Creating and installing a Python package.*

ment the tool will use symbolic links to required files on the system. While the resulting virtual environment is much smaller it is not portable. To ensure portability all the files for the virtual environment must be copied using the *--copies* option with VENV version equal or higher that 3.4 [8].

2.2.2 Packages

Packages are programs that extend the functionality of the Python standard library. After installation modules from the package can be imported. Methods from module can be called with module's name as namespace or with a self-defined namespace. User can also specify the imported methods so they do not have to be used with the namespace specifier but this brings no performance advantage. The whole package is always imported.

Pip is a tool for installing packages or libraries. However, only after Python3.4 pip has been included in the standard library. Obviously pip is not needed to install pip. Installing pip in such case is done by a separate script specifically made for pip installation. [7] PyPI, The Python package Index is the standard repository to install Python packages from [9]. Pip can also be used to install locally stored packages. Local packages can be made with another package called setuptools. With setuptools a developer can make a setup.py file and run a command on that file to create a package which pip can then install to the Python environment. Creating and installing the package can be seen in the Program 2.2. One might want to activate their virtual environment in between setup.py and pip commands. If virtual environment is not activated in between the package will be installed on the host system and not in the virtual environment.

The format of a setup file can be seen in Program 2.3. In Program 2.3 there is generic information that the package will have but also all the modules that will be included in the package which is very important. When packaging command is ran on the setup file the default output will be a folder named "dist" containing the compressed package in format <package_name>-<version>.tar.gz.

```
1  # -*- coding: utf-8 -*-
2  from setuptools import setup
3  setup(
4      name = '<name_of_the_package',
5      version = '<version>',
6      description = '<description>',
7      author = '<Authors>',
8      author_email = '<Authors_emails>',
9      url = '<Linked_urls>',
10     py_modules = ['<Modules>', '<to>', '<be>', '<included>'],
11     )
```

Program 2.3. *Setup file of a package.*

The advantages of packages are easy installation and version management. When a tool is packaged all the source files are compressed and there is no need for copying files. Then any setup scripts do not need to be altered since additional modules can be added to the setup.py file. Also, when installing a package with pip all the files will be placed in the correct locations. Now a whole tool can be packaged, distributed, installed and imported for easy usage.

2.2.3 Distributing Python Tools

Python scripts can be turned into binary and ran without having an interpreter installed on the system. One of the Python tools which enable this is PyInstaller. First PyInstaller gathers all required files and dependencies. Then PyInstaller converts all that bundled up with an interpreter to a single binary file. This way there is no need to install a specific Python interpreter. [10]

PyInstaller binaries are compatible with the environment in which they are created. This means that Linux binaries cannot be run in a Windows environment and vice versa. New binary has to be created for different Python version, operation system and alternative bit versions of an operating system (e.g. 32 and 64 bit). [10] Kernels might require their own versions also. It is tedious work to generate binaries for each development platform but it motivates to keep all development platforms uniform. With binaries which can be ran out of the box nothing needs to be installed. This eases the user experience in an environment where the user does not have privileges. Binaries are stored in a general place for all users and can be ran from there.

2.2.4 Regular Expression

Regular expression is a string pattern matching technology. It is used to search a pattern in a string and then return a match object. Regular expression, or alternatively RegEx, is not a Python specific technology but is widely used in other programming languages as well. In Python RegEx comes in the re Python package. [11]

```
1 import re
2
3 myString = "My fridge is 10 years old."
4 myPattern = "My\s+(\w+)\s+is\s+(\d+)\s+(?:years|weeks)\s+old."
5 match = re.match(myPattern, myString)
6
7 if match:
8     appliance = match.group(1)
9     age = match.group(2)
```

Program 2.4. *An example of RegEx used in Python.*

Match object is returned if a match is found. In RegEx a pattern is compared against a string and if the pattern matches then a match object is returned. This match object can be treated as a boolean because when a match is found it is treated as a True value and when no match is found then None is returned which corresponds to False. Patterns can also have matching or non-matching subgroups. Matching subgroups create groups to the match object first group is the whole matching pattern and following groups are substrings of the first group according to matching subgroups that the pattern has. [11]

Patterns make regular expression special. With Regular Expression user does not have to know the contents of the string specifically but only know the structure of the string. RegEx has generic pattern matching elements like match characters and numbers \w and for only digits \d. These also have opposite elements like match anything that is not a character or a number \W. The elements by themselves match only one character from that set but multipliers can be added after the element. For example multiplier + matches one or more and * matches zero or more characters. Capturing a group is done with parentheses. [11]

In the Program 2.4 is an example of how RegEx is used in a Python program. First there is a string and then the pattern. After defining a pattern it is matched against the string and if match is found then the script reacts to it and perhaps stores or manipulates the string. In the example Program 2.4 there can be statements about ages of appliances and only years or weeks are accepted, no seconds for example. RegEx makes parsing complicated strings easy.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <entity>
3   <firstTag>Data in tag firstTag.</firstTag>
4   <secondTag>Second tag's data.</secondTag>
5   <firsttag attribute="an attribute">Tags are case sensitive</firsttag>
6 </entity>
```

Program 2.5. *An example of an XML document.*

2.3 XML

XML is an markup language for storing data in entities. In ISO 8879 standard SGML is defined of which XML is a derivative of. The goal of XML is to be easily constructed, widely supported and have the minimum amount of features. [12] In Program 2.5 an example of XML document can be seen. XML files are constructed from elements. Elements have starting tag and ending tag. Between start and end tag there can be more enclosed elements with similar pattern. Elements can have either element content or text content. In the example tag "entity" has element content while other elements have text content. The first line is optional but useful for defining the encoding if contents have UTF-8 or other special characters.

The XML definition is loose so XML documents can be used very flexibly. XML documents are used usually data storage and are especially useful when multiple data sets have the same formatting. Similarly formatted data can be then parsed by a program capable of XML parsing and data can be extracted from the document. Since there are no formatting requirements for XML the document could contain anything, more specifically unwanted or malformed data. If the data should be formatted in specific way XML schemas should be used.

2.3.1 XML Schema Definition

XML Schema is used for conveying constraints for an XML document. XML Schema is an language of it's own and very different from an XML file and there are multiple of XML Schema languages. [13] XML Schema Definition, XSD, is a validation language developed and recommended by World Wide Web Consortium, W3C. Unlike DTD, XSD language is similar to XML language so it could be easier for a person who knows XML to write XSD than to write DTD.

XML Schemas are used to write validation files. XML documents are then run against the validation files to see if they are well-formed or not. This is called validation of an XML file. Validation is done to preserve quality and ensure that XML content is in a specific format. With validation more strict versions of XML can be created. [13]

In an XSD file definition about the presumable contents of the XML files are made. XSD defines what kind elements and data types can be inside the file under validation. Multiple XSD files can also be linked together by inclusion to form even more complex validation.

The framework for XSD file relation is provided by SML. [13] XSD file itself does not do validation. For the actual validation a program is needed which tests the XML against the XSD file. There are Python libraries (e.g. *xmlschema*) and online services which do the validation. Only reputable services should be used since the company files will contain sensitive information.

Namespaces are used in XSD to avoid element naming errors. XSD can accidentally or on purpose redeclare elements. This can be avoided by using namespaces. With namespaces two or more similarly named elements can co-exist with prefixes. These prefixes are the name given to a namespace in the inclusion of the namespace followed by a colon (e.g. '*<namespace>:*'). [14]

2.3.2 IP-XACT

IP-XACT is an XML Schema standard to provide a consistent and standardized way to access information about an IP [15]. From 2003 onwards SPIRIT Consortium developed the standard by increasingly adding features. In June of 2009 IP-XACT was submitted for approval to the IEEE organization. Then one year later it became available for free, after which Accelera formed a group to work on the IP-XACT Schema. [16]

IP-XACT standard is used to provide electronic information about an IP. With IP-XACT an IP can be reused without files for specific implementation because all the information is stored in IP-XACT file. IP-XACT validated XML documents can be component, design, design configuration, bus definition or abstraction definition related. For example IP-XACT document about a component contains naming and port information. [17]

Proprietary and open-source tools are used to process IP-XACT and create content from IP-XACT meta-data. Some tools also can create IP-XACT from the content. For example HDL for RTL register implementation can be generated from IP-XACT with a generator tool and backwards IP-XACT can be created from the HDL file. [17]

To describe information that does not fit into IP-XACT vendor extensions can be used. Vendor extension accept anything which is valid XML. This means that the tool that processes the IP-XACT either needs to know how to process specific cases of vendor extensions or leave them intact. If the tool does not know how to process vendor extension some other tool needs to be used for them. [17]

Table 2.1. *An example of an account balance table within a database.*

index	clientID	name	surname	accountBalance
1	31415	Kalle	Jyra	1672.42
2	82317	Byron	Bernstein	9132.92
3	29872	Rania	Morosan	5211.36
4	77417	Matti	Meikalainen	34 881.55

2.4 Databases

Databases are an electronic way of storing indexable information. SQL language is used to create, delete and manipulate these databases by sending queries to them. The advantages of SQL databases come from fast and efficient data retrieval. With queries the content of a retrieval can be specified and broken down to smaller sets rather than all of the data. Goal of databases is also to keep the data consistent whether data is added, deleted or updated. This is done by having unique identifiers which is an ID for each data. This means that the database can have two or more datas which are the same by content but are identified as different datas. [18]

Databases are very inflexible in ways they storage data which means that data is stored in raw format. Text files and spreadsheets can have analyses, graphs and such. With text files or spreadsheets there are no rules. Any data can be inserted anywhere. With databases there is always a defined structure. Data cannot be added without a structure in the database and specifying what data is being inserted. [18]

2.4.1 Managing a Database

An example of a database could be bank's details of account balances. This kind of database could contain client's name, surname, client ID and their balance. Client ID in this case would be unique identifier between clients because two or more clients can have the same name. With these unique identifiers same person can be identified in another database for example loan amount database. In the Table 2.1 an example database can be seen. In the database index is an unique identifier along with clientID. Other column names are name, surname and accountBalance. On the same row there is information related to each other. So for example person with clientID 31415 is named Kalle Jyra and their account balance is 1672.42.

To access the data a query is sent to the database. The data within a database is held inside so called tables so that the database can have multiple tables and we do not have to create multiple databases. The database table in Table 2.1 could be given any name, for example "*information*". To receive data from the database a table and column name have to be issued within a query along with optional specifiers. In Program 2.6 a query is sent to the connected database. The query selects the accountBalances from the table information which are in between values 5000 and 10000. This query would then

```
1 SELECT accountBalance
2 FROM information
3 WHERE accountBalance BETWEEN '5000' AND '10000';
```

Program 2.6. *An example of an SQL query sent to the database.*

```
1 CREATE TABLE IF NOT EXISTS [information]
2 ( index integer PRIMARY KEY, clientID integer,
3 name text, surname text, accountBalance real );
4
5 INSERT INTO [information]
6 ( clientID, name, surname, accountBalance )
7 VALUES(31415, 'Kalle', 'Jyra', 1672.42);
```

Program 2.7. *An example of creating a table and inserting data to it.*

return values 9132.92 and 5211.36 in that order because that is the order they are in the database sorted by index which is the default sort method. Other sort methods can be defined with sort keyword. Here keywords are capitalized in the query.

To create and manipulate a database we need a database tool. An example of database tool could be SQLite3. Database tool is used to create the database after which we can issue queries to create tables and insert data to the database. The query language is specific to the tool so they can differ a little but generally are very similar. Some tools can omit or add to list of SQL features and how to interpret them [19]. In Program 2.7 first query creates a table named "information" with columns clientID, name, surname and accountBalance. The second query then inserts a row of data to the table for client whose clientID is 31415, name is Kalle, surname is Jyra and their accountBalance is 1672.42.

Database security must be taken very seriously. SQL command are efficient and do a lot with just a few words. That is why one must very careful when issuing a command on a live database. With just one line the whole database with millions of lines can be destroyed or everything from a table can be deleted. Commands affecting multiple lines or tables should be always ran first on a copy or in a way which can be reversed. User can also be given privileges to the database to avoid misuse [18].

All SQLite changes and queries are ACID (atomic, consistent, isolated and durable). This means that even if a program crashes or a transaction is interrupted etc. transaction is still ACID compliant. This means that even in the worst case the database cannot end up in an unstable state and transactions when committed are permanent. When a transaction is started the database is not modified until the commit command. This way unwanted pending changes can be rolled back using the rollback command. [20]

2.4.2 SQLite Database Management Tool

SQLite is the most used database management system in the world. It is small, highly reliable and self-contained, which is why it is used on phones, computers and bundled with applications. What makes SQLite special is that the database is a local file and not a remote server. [19] Therefore it is easy to create a local database and manipulate it. There is no need to make connections to anywhere, not server needs to ran and no client process to access it. SQLite database is either operated with functions or from shell environment.

It is good to use SQLite when the database and application are on the same device. Since SQLite does not need administration it is very suitable for embedded devices, application data or cache. SQLite also supports unlimited number of readers at the same time but only one writer at a time. SQLite which provides a local based solution and server/client database engines are there to solve different problems Therefore they do not compete against each other. [19]

2.5 Related Work

SoC IP information is usually kept under non-disclosure agreement. Companies either write their own VHDL, use their own, licensed or open-source tools to create VHDL for register banks. Related work for other than open-source tools is hard to come by. However with each SoC module connected to the interconnect bus there has to be a submodule which acts as interface between the bus and the SoC module. These submodules are required because the data from the bus has to be interpreted and assessed to run time memory of the SoC module.

These submodules are highly used but very application specific. To enforce SoC module reusability following challenges must be overcome:

- The need for a specific interconnect protocol.
- SoC module's own needs for an interface.

A solution to these problems is to generate a submodule known by many names like a buswrapper, an interconnect interface unit or a hardware socket. When this submodule is done for each needed protocol the SoC module can be reused despite the used protocol. This submodule is able to translate the protocol and communicate with the SoC module.

In [21] a similar tool for a similar case is demonstrated. In their case they have managed to implement a tool that creates a register file with AHB bus interface. From AHB signals they are able to access register fields and they say that by changing only the AHB logic another standard can be supported. Their tool takes register file description, register field template and AHB register file template as input. As output the tool gives HDL files for registers and testbench as well as a C header file for accessing the registers. They say that the tool reduces design errors and accelerates the design and verification.

Comparing to the tool mentioned in the last paragraph to the tool in this project. The tool mentioned in the paper seems to be quite limited in the functionality that it provides. Its' job is to provide simple registers with different access types for AHB bus interface. Also they use application specific register description file which means that it is not standardized. Advantages of the tool in this thesis project is that the input file is standardized IP-XACT and from it registers with more complicated functionality can be generated for multiple protocols.

Rggen [22] is also a similar register generator tool which generates registers with access types. This tool is implemented in Ruby programming language and takes in a register map specification file. It says that the possible specification file can be one of many formats YAML, JSON, Spreadsheet, SiFive DUH or Ruby's own API can be used to provide the register map information. Special functionality includes special bit field types. Generated source files support multiple simulation tools like Synopsys VCS and Xilinx Vivado Simulator and synthesis tools like Synopsys Design Compiler and Inter Quartus. There is no mention about support for protocols. The generated HDL is Verilog and there is no support for VHDL.

Most similar tools just provide latches with protocol interfaces and access types. This kind of functionality is simple and not so interesting. Much more needs to be applied on top of that to achieve performance better than the synthesis tools can automatically provide. Possibilities of self added functionality are pretty much limitless. What makes this tool better than those available is the added functions to the registers.

IP-XACT is used to store information about components and systems. Companies use IP-XACT to collect attributes which can be linked to a specific component such as memory maps, registers, bus interfaces, ports, views and file sets. Multiple components files can be then be connected to produce a design file [23]. IP-XACT can be used to configure different views for a component e.g. simulation or synthesis. These views can differ so that in each different submodules from different libraries are used. For example in simulation a component can use generic models for submodules but for synthesis more advanced model. This way IP-XACT can be included in multiple flows i.e. verification uses another view than implementation does.

Flows are based on IP-XACT. The standard can be used at different parts of the SoC design flow which are IP packaging, platform assembly and flow control. In IP packaging components are gathered in XML files. Those XML describe IPs' attributes such as ports, interfaces, parameters, generics and register map. In platform assembly components can be imported, configured and integrated. In platform assembly design assembly, issue resolving and design automation tasks are done. In flow control design activities are linked around a key IP-XACT document [24].

3 PROBLEM ANALYSIS

In this chapter the state of the project before the start of this thesis is presented. In the presentation the current state of the tool is reviewed and points are made how it could be improved. Afterwards the project brought up points which could be revised and changed for the better. Improvement proposals are introduced here and in the next chapter their implementation is presented.

3.1 Current State

Register banks are an essential part of SoC modules. Register banks in this project are run time memory of modules and act as software interface from CPU to the module. Interface is either a bus protocol or direct ports to the registers. The register bank can translate read and write requests to the registers from bus protocol. This submodule is essential for the reusability of SoC modules since interface that CPU uses to access the module can be changed without modifying the module itself.

The goal of register generator project is to automatically generate suitable software interfaces and registers for SoC modules. In the Figure 1.1 were the phases of the current flow and for the sake of clarity phase from Excel to IP-XACT will be called phase 1, the phase from IP-XACT to Info file phase 2 and the phase from Info file to VHDL phase 3. The initial format is an Excel spreadsheet constructed in a specific way to represent register bank's specifications. The Excel sheet contains register naming, addressing, functionality etc. In phase 1 a tool converts the spreadsheet into IP-XACT which holds the register information. Then in phase 2 a licensed tool is used to generate the intermediate Info file. In phase 3 company tool is used to read the Info file and construct a VHDL representation of the register bank along with the wanted protocol interface. This flow many steps but the key is the intermediate IP-XACT file which is then used in other flows. Info file was used more previously but it has become a legacy issue. The project work of this thesis focuses on the conversion from IP-XACT to VHDL.

Register banks can have more functionality than just being flip-flops with access rights. Register functions and interfaces are illustrated in the Figure 3.1. Register fields can have input functions which interact with the data when it is written to the register. Output functions which do the same but when data is read from the register. Then there are internal functions that react to conditions of ports and other registers. Internal functions can for example set or clear register content when a condition is met. There are even

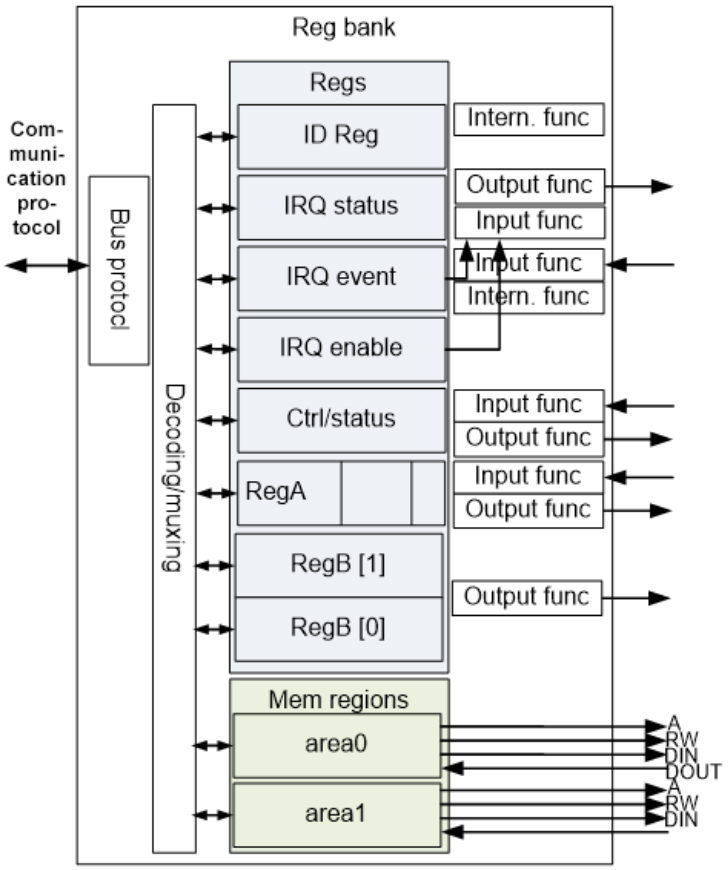

Figure 3.1. *Demonstration of register interface and functionality.*

more functions than shown in the figure but the most basic case is where there are no functions set for a register. Then the register can be only accessed with the read and write commands from the bus protocol.

In Figure 3.1 a register bank with one register block and one memory region can be seen. Register banks can have multiple addressblocks, multiple memory regions within those addressblocks and also external memory interfaces. Registers are unique named within an addressblock so there may be registers with a same name in different addressblocks. Memories can be addressblock mapped into registers or they can be addressblocks. With memory addressblocks the address is decoded and if it hits the memory address space then the data is passed to the external memory. With register addressblock memories the memory structure represents register fields where parts of the memory are mapped to register fields. This means that when data is written to the register field it is actually written to the external memory.

Currently the phase from IP-XACT to VHDL is implemented in Perl with intermediate file formats. The intermediate file formats are not modifiable and modifying the IP-XACT file is not wanted. This means that every version of the component requires their own version of the Excel and later modifications would have to made into every Excel. If the IP-XACT is modified in the process then it obviously does not represent what is in the initial format anymore. Even if modifying the IP-XACT was not a poor practice the modifier must then

know how to do the modification and what the result would be. The tool should use easy file formats like Excel or APIs to make changes. This way the users use the least amount of time and effort learning to use the tool and more time is left for their actual tasks.

Changes to register files are made by changing the Excel file. This means that even small changes require running the flow from beginning which does take some time and resources. Even in some cases the resources might not be available. To elaborate, the tool requires IP from a vendor and this IP requires licensing. Earlier mentioned phase 2 requires functionality of this licensed tool and if no license is available at the time of running the tool the generation will fail and designer will be stuck after phase 1. Also these small changes could be a difference between two versions. So both Excel files are kept in case of version difference.

The phase from IP-XACT to VHDL is being improved. The phase from an Excel to an IP-XACT file of the tool is implemented in Python so conversion of the latter IP-XACT to VHDL phase to Python as well would improve the version control and sustainability of the tool. Also the intermediate file formats are should be more human readable and such that modifications are easy to make. The ultimate goal is to have an API for intermediate file format with which changes can be made and IP-XACT could be generated without modifying the Excel file.

3.1.1 Perl as a Programming Language

The current implementation is done in Perl. Perl is known for its' effectiveness in string manipulation but also notorious for its' readability. Perl programs tend to get complicated really fast because a lot can be achieved with one line of code. This quickly leads to a situation where even the owner does not understand what their code is doing or it is very hard to interpret after some time. Another downside is that hard to read and complicated code tend to become fragile. IP being fragile in these case means that implementing something new has a great risk of breaking some other feature.

With another programming language desired improvements could be achieved in work and development flows:

- Implementing new features could be easier.
- Long term maintainability could be improved.
- More popular programming language could have more extensive support.

Implementing the whole tool in just one language problems such as version discrepancies or missing installations could be avoided.

Changing the programming language would mean that before any new features can be implemented all the old functionality must be replicated on the new language. Without a lot of previous experience on both of the languages this could prove to be very difficult task. The difficulty is possibly tied with types of start and goal languages. If the start and

target languages are high-level scripting languages difficulty could be normal because they are similar to each other. From a high-level to a low level language like C it could be very difficult because the same functionality that a high-level language provides could be way harder to implement in a low-level language. Similarly the opposite way from low level language to high-level it could be easy because what is done in a low-level language could be done easier in a high-level language. Both Python and Perl could be considered as high-level programming languages so difficulty of changing the programming language could be considered moderately difficult.

3.1.2 Intermediate File Formats

Currently the intermediate file format is an Info file. Which is not meant to be read by the user in this flow but user is able to find the same information that is in the Excel file from an Info file. It is easier to parse than IP-XACT file is, at least without IP-XACT specific tools. An example of an Info file can be seen in 3.1. This file is a basic text file where information about registers are written sequentially register after register. Information about one register is between start and end statements.

Info file is hard to use and it is not used elsewhere anymore. When information is extracted from the Info file it has to be processed line by line which is the nature of text files. So specific pieces of information are hard to get like fields of a register with given name. Register information has to be extracted into another datastructure at runtime. From such datastructure specific information can then be parsed systematically and turned into VHDL.

The need for intermediate files is debatable. Even if intermediate files would not be used there would be need to interpret the IP-XACT and turn it into VHDL. Also, if the IP-XACT was the sole file format in this phase then modifications through the proposed API would need to modify the IP-XACT directly and modifications into middle of file could be difficult. So there are advantages to intermediate formats. Intermediate format should be easily readable, data should be easily accessible and modification should be be easy to make.

3.1.3 Work Flow Improvements

Work flow improvements aim for more flexible work flow. Currently the flow must be run from the beginning to the end unless some part has been finished previously. The phase 1 can be skipped if the IP-XACT already exists for example. Workflow improvements aim to:

- Provide capability to do modifications in the IP-XACT after the Excel to IP-XACT phase.
- Have flexible and persistent intermediate file format.
- Freeing the tool from vendor locking.

```
1  -- Start of register MYREG1 in example_registers --
2  registerName = MYREG1
3  registerDescription =
4  registerAddressOffset = 0x4
5  currentAddress0 = 0x0
6  currentAddress1 = 0
7  currentAddress2 = 0
8  currentAddress = 4
9  registerAddress = 4
10 registerSize = 32
11 registerAccess =
12 registerResetValue = 0
13 registerResetMask = 4294967295
14 registerDimensions =
15 -- Start of field MYFIELD --
16 registerFieldName = MYFIELD
17 -- Start of registerFieldDescription MYFIELD --
18 [Output Function: PORT]
19 -- End of registerFieldDescription MYFIELD --
20 registerFieldBitOffset = 0
21 registerFieldBitWidth = 32
22 registerFieldReadAction =
23 registerFieldAccess = read-write
24 registerFieldModifiedWriteValue =
25 registerFieldParameter_resetValue_ = 0x0
26 registerFieldParameter_customType_ = RW
27 registerFieldParameter_hdlPath_ = reg_MYREG1_s[31:0]
28 -- End of field MYFIELD --
29 -- End of register MYREG1 --
```

Program 3.1. *An example of an intermediate Info file.*

There are so many different versions of the same register bank that it would be a lot of work to change some feature in all of the versions and generate them all over again. By not using vendor's licensed tool the company is more in control of changes and features they want to make and licenses are available for other uses.

Changes could be made to IP-XACT without changing Excel like adding a field to register without adding it to the Excel and generating the IP-XACT again. Other tools use the IP-XACT too, like packaging. Therefore a goal is to have the possibility to return into a valid IP-XACT document. This means that changes could be made after IP-XACT but one could generate the IP-XACT with the new changes. At this point the Excel document would not be needed anymore. It would just act as template for different versions. Project would have IP-XACT files but perhaps no Excel file for each version. However, information is far more readable from the Excel file than from IP-XACT so it is arguably more sensible to have multiple Excel files than IP-XACT files.

3.1.4 Vendor Locks

Vendor locks mean a dependency of a vendor's IP. In other words a tool has to be developed around the IP since the IP cannot be changed. Users of the IP just know what goes in and what comes out. Functionality cannot be changed unless it is discussed with the owners of the IP. Even if a new feature is discussed starting to implement it could take a significant amount of time. However if IP was developed in-house and a new feature is wanted its' development could be started immediately.

Companies must buy or order a license for IP because it saves time and money in short term. Developing a new tool takes a long time which is taken from other projects. Bought IP is ready which means it can be obviously used immediately. Budget is calculated carefully for what should be developed by the company and what should be bought from a vendor.

Vendor IPs, in most cases tools, usually have a license model. This means that when one person is using the tool other people are unable to use it if there is only one license for that tool. Therefore usage of licenses is tried to be kept at the minimum and it is always better if a license use can be removed from some flow to be available somewhere else. Limits in available licenses create bottle necks in work flows where a person cannot continue their daily work because some license is not available. Vendor locks and licensing is not always a bad thing. License means that there will be updates and support for the tool and these tools are way too complicated to update and maintain as a company's side project so they are better off buying them.

3.2 Proposed Changes

In this section proposed changes are introduced. With these changes the goal is to improve the overall quality, usability and availability of the tool. Proposed changes also aim to make future features easier to implement. Since this tool is produced fully in the company it is not bound to some vendor's tool which tackles the vendor lock issue. To elaborate, the tool which uses the licence is no longer used so the licence is no longer used either. Using the database makes the intermediate file more flexible and open for future uses.

3.2.1 Change from Perl to Python3

The choice has been made to move from Perl to Python3. Python3 has been chosen because previous phases of the tool have been implemented in Python as well. Python is also very widely used so there is support for its' usage, libraries and problems. Therefore development for Python applications is quite easy and recommendable. Most notably Python has xmlschema and SQLite3 libraries. Xmlschema is used to parse IP-XACT XML document to Python datastructure tree and SQLite library provides a Python API for SQLite databases. When changing the programming language the functionality of the Perl implementation has to be replicated with Python.

After the replication more functionality can be added and the tool can replace the old implementation in the flow. With the replication of the Perl implementation's functionality with Python control of earlier mentioned phase 2 which uses vendor's tool could be achieved. This means that the phase from IP-XACT to VHDL would be no longer dependent on some vendor's tool so it would not use licences. The tools could be fully developed and maintained inside the company which means full control over it.

3.2.2 SQLite Database Intermediate File Format

The choice has been made to replace intermediate Info file with SQLite database. This SQLite database would then contain the information gathered from the IP-XACT file. Multiple versions of the register bank could be then added to the database having their own vendor-library-name-version (VLNV) identifier. So database could then contain multiple versions of components without keeping multiple Excel files.

SQLite is desirable because of its' usability. SQLite is a local database solution so there is no need for a server and users can easily build multiple of these databases and maintain them in a project folder. Other driving factor for SQLite is that Python has an API for SQLite databases. SQLite is also stable and widely used. There are a lot of tutorials and help for SQLite solutions. Database structure is easily modifiable if new functionality is required.

The use of database targets for more flexible implementation. It enables changes after the IP-XACT phase. SQL commands can be issued to make changes to register banks quickly. Data is also more accessible in a database than in the Info file since any specific data can be selected from the database at any given time with a query. Similarly data can be modified and inserted easily. Overall, the database seems to be better solution to maintain data between tools than text files.

3.2.3 Generating VHDL from SQLite Database

With the proposition of SQLite database being an intermediate file between IP-XACT and VHDL the VHDL file has to be generated from the information in the database. This means than IP-XACT information is stored into the database element by element. Previously only register information was extracted from the IP-XACT to an Info file but now the whole IP-XACT is stored in the database.

In database multiple register banks or multiple versions can be stored and the correct information is then fetched with an unique ID or a VLNV. Both of these suffice since there can only be one component in the database with a specific ID or VLNV. If a design uses the same VLNV as some other design already in the database then the design which is being added will overwrite the existing one. Using a VLNV is more user-friendly because there is no way for the user to know component's ID without looking at the contents of the database.

4 DESIGN AND IMPLEMENTATION OF THE TOOL

In this chapter implementation of the changes are gone through. Each phase of the newly developed tool's explained in their own section together forming the whole flow from IP-XACT to VHDL. In Figure 4.1 comparison between the old and new flows are made. Info file is replaced with a database and the possibility to go back to the IP-XACT is an additional feature.

4.1 Database Generation

In the phase of database generation IP-XACT information is transferred to a database. The tool takes an IP-XACT as an input and optionally the location of the schema files describe the structure of IP-XACT. From the IP-XACT document a database is created which should somehow resembles the structure of the IP-XACT.

Information IP-XACT file is extracted with Python libraries and recursive programming. First the contents of the IP-XACT file are placed in Python's datastructures. Then from datastructure the information is divided into three categories table names, column names and data. After the division database is created from data using Python's database API.

4.1.1 Python and IP-XACT

The Python libraries used to extract the information from elements in IP-XACT are lxml and xmlschema. Lxml turns an XML document into an object which Python can handle and xmlschema library is used to turn XML Python object which is constructed according a schema to Python datastructure. API for schema library takes optionally a location of local or external schema files. Giving the schema file is optional because at the start XML file the location of the schema is given and therefore the library is able to retrieve the schema from its' source. Although, it is encouraged to use local schema files since retrieving the schema from its' source takes significantly longer time than from local files.

Lxml library produces a datastructure from XML and xmlschema makes a dictionary out of lxml's output which is used in this project. The dictionary is constructed in such a way that it resembles the IP-XACT document. Therefore the result is a tree-like datastructure which contains a variety from dictionaries, lists to strings. A short XML example is given in Program 4.1. First there is *components* element which has only element content. The *component* element within *components* element has elements with text content. Those

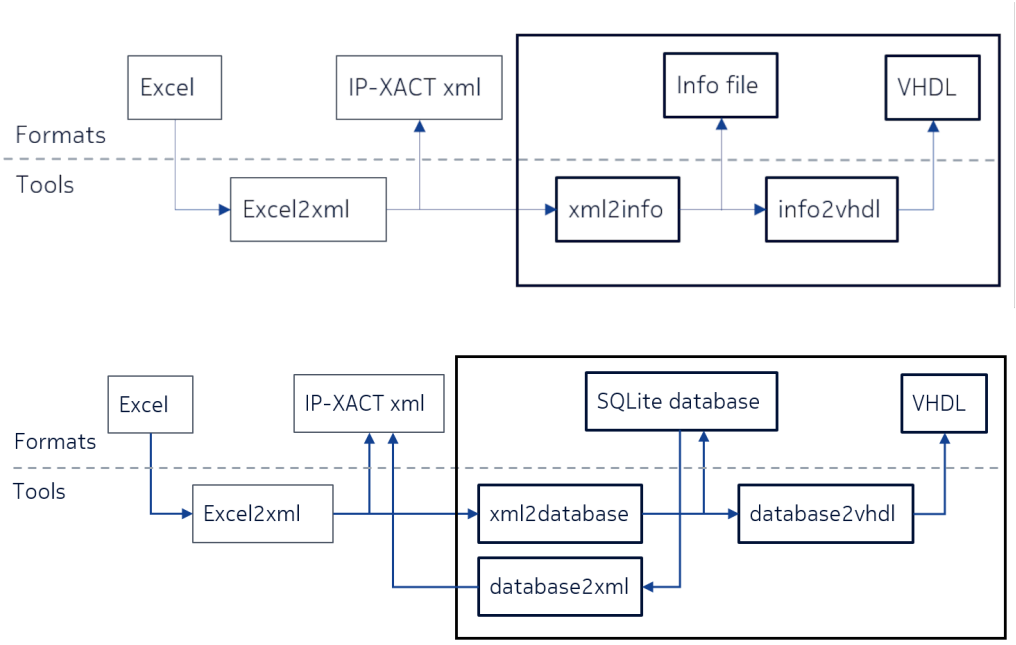

Figure 4.1. *Comparison between old tool flow above and new tool flow below.*

```
1 <components>
2   <component>
3     <name>myComponent</name>
4     <version>0.1</version>
5   </component>
6 </components>
```

Program 4.1. *An example of XML content to be turned into dictionary.*

elements are named *name* and *version*. This example would result following datastructure: *{ components : [{ component : { name : "myComponent", version : "0.1"} }] }*. Element that has elements with text content in them will result a dictionary in which key is the element's name and value is the text content. If element has only element content it results a list containing dictionaries.

There are multiple converters for xmlschema which result in a specific kind of datastructure. Some converters are lossly while others are lossless. Lossless converter means that no data is lost in conversion from XML to the Python datastructure. Other converters may be lossly but give smaller and easier to handle datastructures. In this case the default converter is a lossless converter. A lossless converter is specifically wanted because all of the data is important even more when we want to turn the result back to an XML file. If some data is missing the document obviously is not the same anymore.

The datastructure received from xmlschema API is gone through recursively. The goal in development was to look at the resulting datastructure and identify similarities and repeating structures. First XML and the resulting Python datastructure had to be examined. Then a sensible development could be made accordingly. The result was to check dictionaries' keys' values' type. Then go through the tree structure recursively and act according to values' type.

4.1.2 Database Structure

Next step is to turn the Python datastructure into a database. Independently developed parser will be used to achieve this since there seemed to be a lack of libraries which could convert Python datastructures to SQLite database. Either a need for such application is very niche or results are too dependent of a starting point. There is a need for a developer to decide how Python datastructure will be mapped to the SQLite database. So a parser was needed to be made for this specific application but the resulting application would suit to be general purpose Python datastructure tree to SQLite database converter.

A lot of things have to be taken into account in this phase:

- Database having multiple components with multiple versions.
- How is proper order maintained in database similarly to IP-XACT.
- Which elements are tables, which elements are column names and what is the data.
- How to know which element owns which subsequent element.
- When is a new element with element content opened in database and when is it closed.

First two items are pretty easy to tackle. To track which component certain data belongs to we associate an unique ID number for every component which is linked to all the data that a certain component has. To maintain order the same order as in IP-XACT file all the data placed on the same row in the database must have a running number to identify where it is located in the IP-XACT file. Now every component has an unique identifier as well as a running number to maintain order. This means that every row in the database has some meta-data associated with it.

Tables, columns and data is created based on the datastructure. Database have to start with a table. Similarly XML document starts with an opening element tag. So it is determined that an opening element with only element content like *components* and *component* in Program 4.1 would be tables. Elements can have elements inside which contain text content or more element data. Text content is inside another dictionary because text content is always in a key to value pair where the key is the name of the element and its' value is element's text content.

```
{'component':
  {'spirit:busInterfaces':
    {'spirit:busInterface':
      [{'spirit:busType':
        {'spirit:slave':
          {'spirit:memoryMapRef':
            {'@spirit:memoryMapRef': 'memory_map'}}}]},
 'spirit:library': 'ip',
 'spirit:memoryMaps':
    {'spirit:memoryMap':
      [{'spirit:addressBlock':
        [{'spirit:baseAddress': '0x0',
          'spirit:name': 'myRegBank',
          'spirit:range': '0x184',
          'spirit:register':
            [{'spirit:addressOffset': '0x0',
              'spirit:description': 'Identification',
              'spirit:field':
                [{'spirit:access': 'read-write',
                  'spirit:bitOffset': 0,
                  'spirit:bitWidth': 32,
                  'spirit:description': 'This is for ....',
...
...
...
...
  'spirit:vendor': 'myVendor',
  'spirit:version': '1.0'}}
```

Program 4.2. *An example of a dictionary received from xmlschema API.*

Program 4.2 shows an example of IP-XACT data turned into a dictionary though it is shortened to be an example. In the example also the usage of lists is demonstrated. Elements, which there can be many of according to the XSD, produce a list. If the schema allows there to be a multiple of some entity it is placed in a list. For example in an address space there can be multiple register banks so there is an address block in a list inside memory map and again register bank *myRegBank* is inside another list. Also attributes in this case have a prefix *@* but it can be changed in the xmlschema API.

To maintain same order and form in the database as in the IP-XACT multiple things have to be done. Matching every row of data with an unique running number was already discussed. However the running number is not enough to maintain proper form. This means there needs to be a way to track elements, parenting elements and indentation. Element names are either table or column names. Attributes are a special case where a column is created with a name *<element_name>@<attribute>*. Parenting elements meaning the element which encloses subsequent elements are tracked by their own specific column which is called *reference_table*. Reference table reports the table in which the table in question is enclosed in. In Program 4.1 *component*'s reference table would be *compo-*

nents. Therefore this reference table is a link between tables. These reference tables are used to keep track when to open or close a element and increase or decrease the indentation. This is done by checking are the reference tables for certain data same. Special case is the first table which references itself to point that it is the first and has no parenting elements.

In previous paragraphs it is explained which dictionary structure creates tables, columns and content. Yet elements with only element content, like *components* in Program 4.1, cannot be represented in this way. Without representing elements with only element content the conversion to the database would be lossly so they need to be represented in the database. In IP-XACT these elements are usually plurals of their nesting elements e.g. addressblocks element contains multiple *addressblock* elements and so on. If an element has only element content meaning it does not have any text content, elements with text content or attributes it would create no table. The reason for not creating a table is that there simply would not be any text content to insert. However, there needs to be some way to identify these element content elements. This is done by creating tables with only the meta data. These tables will have only the default information that every table has which is a component unique id, a running number and a reference table.

4.1.3 SQL commands in a file

Database is created and manipulated through Python's SQLite3 library's API. The SQLite3 API works similarly within Python like the SQLite3 engine would through terminal. In Python script a connection to a database and a cursor for that connection are created. Cursor can then be used to issue SQL queries to database using cursor's execute command. Executed instruction can contain a single or multiple queries. Execute is used to issue one query and executemany is used with a sequence of parameters. Queries can also be read and executed from a file or a string.

To enforce good practices with SQL databases long sequences of commands should be issued within a transaction. SQL queries generated in IP-XACT to database phase are first printed in a temporary SQL file. Depending on IP-XACT file the resulting temporary file can be tens of thousands of lines long. Contents of the temporary SQL file are then ran in a single transaction to increase efficiency. Transactions could save one also from disasters. When a transaction is started the database is not saved before a commit command. If one would somehow mess up the database within a transaction it could be easily undone with a rollback command. Rollbacks are only possible within transactions. Rollbacks here are not used because the tool does not use databases interactively and transactions are used only for efficiency.

SQL query may require some input sanitation. Apostrophes need to doubled when inputting them into SQL database meaning *'* has to be *''* which is a double apostrophe, not to be confused with a quotation mark (*"*). Line change (*\n*) is a tricky one as it needs to be done with concatenation. Line change is replaced with *' || CHAR(10) || '*. The first apos-

trophe ends the string before line change (CHAR(10)) and the other apostrophe starts the string after the line change. Then these two strings are concatenated by || with a line change character between them.

4.2 IP-XACT Generation

In this phase a conversion is done from the database back to the original IP-XACT. If the database is well structured and the conversion from IP-XACT to database was lossless returning from database to the IP-XACT document should be possible.

The database's structure was build with having the order in mind so now it is easy to keep the same order for the IP-XACT file. All the data for a component is fetched using the running number. Elements are then fetched from the database using the running number. Correct indentation, opening and closing elements is done with reference with combination of the reference tables, the running number and database structure ruling created by the developer.

4.3 VHDL Generation

The result of this phase is VHDL file which can be verified and synthesized. Register information is combined with a wanted protocol. The result is a source file for a register bank that can be accessed by a specific communication protocol and direct ports described in the register bank's description file. Registers can also have more special functionality than saving data which is described for each register field in the register description file.

Generated VHDL is achieved by manipulating the previously generated database, analyzing the information and printing it out. All the information associated with a specific component is fetched using an unique identifier ID of that component. Data of that component is then manipulated and saved again in different form to the database so it is easily printable. After this the information is written to a file.

Only in this phase the choice of protocol makes a difference. The protocol does not affect the registers themselves but only the way they are accessed. Therefore registers can have different protocol interfaces in different instances and not too many changes have to be made to the processes when changing the protocol. Possibility to choose a protocol interface greatly increases the reusability possibilities of a component. The component does not become unusable if another protocol is needed because it can be generated again with another interface.

4.3.1 Gathering Information

VHDL generation is divided into two phases:

- Gathering information into easily printable format.
- Interpreting the gathered information and writing the VHDL file.

The division is done because in the database the information is scattered because all information and proper structure of IP-XACT are kept. So to ease writing the VHDL file the database has to be parsed and information is placed in temporary tables within the same database. If the IP-XACT standard is updated some changes have to be made to the parsing here to make it more compatible. For example there are differences in IP-XACT namespaces where older standard uses *spirit:* and newer uses *ipxact:*. Structural changes in standard are less likely but if they happen then the locations of data are different and data must be selected from different place in the database according to the standard but the automatic database generation should work regardless of the standard. Also support ending for the Python XML parser library is unlikely but possible. If support for the xmlschema library ended it would not be a big deal because a specific Python version is used anyways. A disaster would be if support would have ended, a new standard was introduced and old library version would not be able to support it.

Temporary tables contain only closely related data. Their structure was driven by the VHDL structure and a thought of what information is required at some given point. For example when outputting the entity declaration generics and ports should be known so there should be a table for generics and ports. Each row in tables would contain a generic or a port with relating information. These tables are but not limited to ports, signals, generics, process information and register information. Ports table has port's name, type and left and right boundaries. Signals table has signal's name, type and left and right boundaries. Generics table has generic's name, type and default value. Process information table contains information for each process. Register information table has each register their addresses, ranges, registers' addressblock and possible register file. From these tables different parts of the VHDL document are easy to print out.

In the tool there are functions for filling each of these tables. Scattered information is fetched from the tables which were automatically generated earlier in the flow, properly combined and then placed to the temporary table. Filling the tables requires some pre-known information. This pre-known information is names of the tables and columns. So filling the tables is at the moment is pretty dependent on the namespace and version of the IP-XACT standard. Currently there is no idea how to avoid this because the database is automatically generated based on the structure of the IP-XACT XML. Solution would require different kind of approach to transforming the IP-XACT to database. However, there are workarounds to this because there are ways to check if table or column exists. Trying out which tables exist could help to identify the version of IP-XACT. This functionality is already used with description columns or similar. The fact is that some columns or tables do not always exist means a check has to be made if a table or a column exists.

Some cleaning has to be done with the temporary tables between runs. These temporary tables are removed at the end of run but removal is tried also in the beginning. The removal in the beginning removes the possibility of the database being in an unknown state. Perhaps the last run could have been interrupted in the middle of a run resulting the temporary tables staying in the database. If the temporary tables remained in the database next run would have information from last run which would give unknown results.

Tables should have columns which connect tables together to make selecting needed information easier. This means that when there is a table for register fields there should be a column for register in which that field belongs to. This becomes especially useful when calculating addresses for the registers. Register can belong to register file, register file to address block or there can be a register file within a register file. If table about registers has information about in which register file a register belongs and a table about register files has information in which addressblock a register file belongs to it is then easy to calculate $address\ block\ offset\ +\ register\ file\ offset\ +\ register\ offset$.

In most cases gathering information is pretty straight forward because all the information is from the same file. However, some ports may require input or output synchronization which is done with synchronization components. In trials generic synchronizer models are used which are them replaced with the correct technology in the implementation phase of the design. When a design uses a synchronizer the generic model is fetched from its' source. Then the entity of that generic model is read, parsed and saved to the database. In the register bank these synchronizers need to be first introduced as components and later used in the design by mapping ports and signals into their inputs and outputs. The difficult thing is that component introduction and instantiation have to be parsed from the entity. Since there are many ways in which the entities can be declared and they can have different amounts of generics and ports the parsing can be really difficult. This would need a real VHDL parser but regular expression will do for now.

4.3.2 Generating Entity

The VHDL file is printed from top to bottom. First some information about the tool and used command line options or environment variables are printed. Tool information contains version and date. It is good to keep track of different versions of the tool with version numbers. When the tool is used the exact version is known if result contains some errors. Users may be using an old version of the tool and reported errors could have been fixed already. There should be also another access to the version number than the produced VHDL file in case that the tool crashes in VHDL generation and does not produce anything. A solution is to have version number also in the beginning of the --help option's print. Before entity there are used libraries introduced. Used libraries are constant in the scope of this tool.

Fetching information from the database in Python results a list of tuples. Issuing a *SELECT*-query to the database with name and direction in selection for the port table would return a list of tuples where data from each row would form a tuple in the list (e.g. [(name1, direction1), (name2, direction2), (name3, direction3), ...]). These are easy to go through with Python's loops and tuple unpacking.

Entity describes the external interface of the component. Entity has a name which can be used then to invoke this component. Entity's name is a combination of component's name, used protocol and description (e.g. *myRegs_axi4_registers*). Next are generics. Generics are used to make the design more flexible and reusable. In this case they are mostly used to forward information to the register bank since the component is modified through the tool's register description file.

After generics are component's interface's ports. At this point all the ports are known because of the information gathering phase. The database contains ports' names, directions, widths and types, even those from the protocol. The combination of width and type proved to be difficult in some cases. Difficulty was that std_logic_vector can be width of 1 (e.g. 0 downto 0) and at the same time std_logic type is width of 1 but are not indexable. At first ports were stored without type and when they were width of 1 they were expected to be std_logic. This did not work since in some cases vectors of width 1 were required because of indexing. Std_logic is used where signal cannot be wider than 1 bit. In the end type was also stored in the database and declaration to be either a std_logic_vector or a std_logic is done based on that. The decision between std_logic_vector and std_logic is made in the gather phase based on the source and description in the register description file. Now that all the port names, directions, their widths and types are known it is very easy to print them out.

4.3.3 Generating Signals and Functions

Next job is to generate the actual functionality of the component. Functionality is written in an architecture block. Before the body of the architecture which means before the begin keyword signals and functions are declared. Signals are processes' memory and create registers in RTL. Used signals are known beforehand because they are generated to the database in gather phase based on register description file. Unfortunately this way some signals are not always used and are optimized away by the synthesis tool. So here is some room for improvement. The tool could more intelligently interpret the register description file and not generate redundant signals. Similarly to ports, all the signals are easy to write out when all the information is collected.

Each row in signals table is independent signal. There are columns for name, type, left boundary and right boundary. First this table was made without type column. Type was interpreted so that signals which were wider than one would be *std_logic_vector* and those with width of one would be *std_logic*. Quickly this perception was proven to be faulty. There is a difference in VHDL compilation between *std_logic* and vector which

is "*0 downto 0*". Type column was then added and it is then used to identify difference between *std_logic* and vectors with width of one. This same thing is controlled on ports with an option.

Before the body are also functions and procedures. Functions and procedures are there to support processes and are a way to avoid complex repetitive code. These stay constant through uses so they can be written out similarly each time. Some optimization can be done to print them only when they are used but unused functions and procedures should not have effect on synthesis since they just remove repetitive code.

4.3.4 Generating Processes

Processes can be divided into write, read, output and protocol related processes. In write processes data is taken from input ports or protocol's data bus and then placed into signal associated with that register. This means that there is one signal for the whole register and one signal for each field. Inputting data into the register's signal is done by indexing register's signal with boundaries of the register field. Each field can have their own input, enable, clear etc. functionality. Each process is also synchronous which means that data would move every clock cycle from write data to the registers. Activation signals are used for registers so that only one register is activated when writing from protocol interface.

The register field functionality is a specialty of the tool. In register description additional functionality is described. Tool then parses the register description with regular expression and then VHDL is generated according to the description. For example in the description there can be a request for port for input or output. Port functionality is parsed from description and then an internal signal for the register field is directly connected to a port.

Output processes are used to keep output ports or field registers synchronized so there is an output process for each register field. Output processes are sensitive to the signal associated with the register so any change in the register's signal will result changes in the register's field's signal. If there is an output port associated with the register field changes in register's signal are then reflected to register's field's output port. This means that when field has an output port the register's field's signal is replaced with the output port.

In read processes data from register's signal is transferred to protocol's read data. Each register has a signal of same length which is used as protocol's read data signal. If read for a specific register is not active the read data signal is filled with zeros. This means that if read is not active for the protocol contents of this read data is zeros and it does not interfere with the read data from other registers. OR-operation is used to reduce down all the read data signals to one which then is connected to the output port. When read is active register's signal's content is copied to the read data signal from where it is then read.

In addition there are a few miscellaneous like strobe or synchronization processes. In strobe processes there is a strobe port attached to a register field and whenever field's content is accessed it creates a pulse to the strobe port. With synchronization processes registers contents can be synchronized with clock and/or reset signals from different domain than in which the register bank runs. Synchronization processes are also used to create a delay in the data with high clock speeds to reduce the amount incorrect data.

There are different kind of access types for register fields which modify the functionality of the processes. For example if a field is read-only the protocol takes care that the register is given no write access and similarly with write-only. These are done with register field specific active signals. If field is write-only then the read activation signal never goes to one. More exotic access types have more functionality like read-to-clear, RC. RC makes it so that content of the register is cleared one clock cycle after reading. There are also opposite actions to clears which is set. Set sets all the bits to one instead of zero. ReadwriteOnce, W1, gives only one chance to write to the register field etc.

Register mapped memories are done differently. There is a possibility that a memory component is accessed through a register interface or memory has its' own addressblock. Before any processes are printed for a register a check is made if register is actually address block mapped memory i.e. register is actually an interface for a memory. If it is then it is skipped completely because memory read and write processes are for the memory and not for the register. This means that if a memory is register mapped then no processes are printed for that register but afterwards processes for these memories are printed from their own table.

4.3.5 Generating Protocol Interface

Protocol processes are very closely depended on the protocol. This section is based on AXI4-Lite because it was implemented first in this project. Protocols can be parallel or sequential and synchronous or asynchronous. AXI4-Lite is parallel and synchronous. Protocols have a lot of different response signals and busses, which are usually data or address busses. It is tried to keep the effect of a protocol to processes outside protocol at minimum so other processes do not have to be changed if the protocol changes.

To know which register protocol is trying to access address has to be decoded. In the address decode process protocols address bus is checked for an address and if the address hits the address space then either read or write is for that address space is activated. One register bank can have multiple address blocks whose own virtual addressing start from 0 or offset and they have a base address. At the same time address is copied to a signal for protocols read or write address signal. Two signals are used because address meant to be locked when write or read starts.

In protocol processes registers have their own addresses. Address decoding activated an address block and in the read and write processes it is now determined which register is read from or written to. Register addresses are calculated by adding offsets together.

In the most simple case there is an address block with an offset of "0x0", which means no offset, and that address block has registers. Registers within an addressblock have an address offset relative to the address block's offset. This means that two or more registers can have the same offset but they are then in different addressblocks therefore having a different address in the end. Similarly to address blocks' offsets there can be register files within address blocks which then have their own offset and relative addressing within them.

Protocol interface has processes to read and write from the registers. Protocol processes work with state machines where one read or write takes multiple clock cycles. Both processes start similarly. First the state machine is in *not active* state in which the state machine waits for changes in read or write enable signals. When read or write is enabled the address is stored from address bus and state machine moves to next state. Next in write state machine waits for valid write data and then stores it. After receiving valid data the write process activates the right register according to the address. When register is activated it saves the data from data bus. Process then sends a response and moves back to *not active* state. Read works similarly but data comes from register which is activated and is written to the databus.

Read process can easily result in bad RTL. When a result from a register is written to the read databus it is done by using the OR-operation for all read data signals. One of these signals has the value and others are full of zeros. With a small amount of registers this is not a problem but if hundreds and hundreds of register signals are reduced down to one with OR-operation it starts be problematic. One solution to this is to have staged OR-operations where register signals are divided into groups and reduced into a smaller group each stage. Each stage takes one clock cycle but reduces the critical path.

4.4 Using the Tool

The tool is used as command line interface tool. It is invoked in the flow of register generator, has no graphical user interface and most likely never will. However, it can be used independently if previous steps are done, meaning there is an IP-XACT document already available. The tool uses environment variables and command line options to configure it's run. These range from file paths to which kind of capitalization should be used in VHDL generation.

The whole tool is modular. One module does the transformation from a XML to a database. Second does the generation of VHDL from a database. The last is extra in this flow but does the transformation back to IP-XACT from database as that functionality is not used in this flow but it is used in other flows in the future. In register generator tool flow the XML to a database and the database to VHDL tools are served within a wrapper. This wrapper handles environment variables and command line options with a class that uses its' methods to parse the variables and options. All the configuration information is collected to this class and it is passed within the wrapper to the tools which use the information.

During run time the tools give out some information. The outputted information is the phase that is running currently, their input files and the most important configuration items. There is no verbosity option or progress meters because the run time of the tool is quite short. More importantly the tool tries to identify erroneous cases, interrupts the run and informs the user. These in most cases are sanity checks for user input in register descriptions. However invalid cases which have not been identified by the developer can cause a crash or malformed VHDL. In crash the tool abruptly stops working because of faulty coding. Malformed VHDL in the other hand does not cause the tool to crash but causes the generated VHDL file not to compile or pass the verification.

4.4.1 Invoking the Tool in the Flow

Parts of the tool can be used independently or within a wrapper. The package in which this tool comes contains multiple wrappers, xml2database, database2vhdl, database2xml and xml2vhdl. Notably xml2database is used to add additional components to the database but xml2vhdl also does this because it contains xml2database in it. Xml2vhdl replaces existing component if it exists in the database or adds it if it does not. If xml2vhdl is not given an XML file then it tries to find the component from the database and create it from component already existing in the database. That is what database2vhdl does independently. So xml2vhdl is a "smarter" wrapper and combination of xml2database and database2vhdl. Database2xml is planned to be only used independently since it requires the database to already exist. At the moment it has bad user experience because user needs to know what they are doing but it out of the scope of this thesis.

The tool used to to generate a VHDL file from an IP-XACT XML file is distributed as a binary to users. When the tool is distributed as a binary it works independently right away so no installations are needed. There is no need to invoke it with the Python interpreter as Python scripts usually require. Which means *python3 -m ./xml2vhdl.py* for a Python script and *./xml2vhdl* for a binary. The binary can be run without the interpreter as the interpreter is included in the binary which works as far as the environment and kernel are appropriate. This means that Python interpreter within the binary still has some dependencies which have to be satisfied.

4.4.2 Options and Environment Variables

Options are a way to configure the tool. Options can be used to change default values of variables which affect the run of the tool but some options can also be obligatory. Options available to the tool are given in the Table 4.1. In this tool a Python class is used to store option information. The class is then made available for all the functions so every function can check options where it is needed. In flow from a IP-XACT file to a VHDL file the option which provides the path of the XML file is obligatory. Less the obligatory options easier the use and better the user experience is. Used options are printed in the beginning of the generated VHDL file to make solving issues and debugging easier.

Table 4.1. *Options to modify tool's run.*

Option	Feature
addr_bus_width	Define address bus width.
architecture	Define name for the VHDL architecture.
clean	Remove the run time files.
database	Define the database name.
data_bus_width	Define data bus width.
debugfile	Define debug file name and redirect stderr.
debug_mode	Turn on debug prints.
debug_stdout	Redirect debug prints to stdout.
entity	Define register bank's entity name.
field_names	Define field's capitalization.
help	Print help and exit.
max_addr_range	Define maximum address space.
memory_read_wait_cycles	Define clock cycles before memory read starts.
parameter	Define parameter or a generic.
port_name_prefix	Define ports' names' prefix.
protocol	Define protocol that the register bank uses.
register_read_wait_cycles	Define clock cycles before register read starts.
register_names	Define register name capitalization.
reset_level	Define reset level.
reset_type	Define reset type.
vhdlfile	Define name for the generated VHDL file.
vhdl_port_mode	Define type of ports with width of one.
vlnv	Define a VLNV to select a design.
xmlfile	Define input XML file name.
xmlschema	Define location of the XML schema.

Environment variables work similarly to options with this tool. The script which calls this tool sets environment variables with same names as the corresponding options are. Then the tool goes through the set environment variables and checks if there are some that correspond to an option. Used options overwrite environment variables which means that if there is a variable set and then tool is used with the same option then the option is more prominent. It is also good to have environment variables' names' in line with the earlier scripts so there are no multiple environment variables for the same thing.

4.5 Documentation

The whole register generator tool flow has its' documentation but there is also phase specific documentation. Flow documentation gives users an overview of the tool how it is used and for what purposes. It goes through steps in the flow, how they are executed and what are the inputs and outputs.

With the register generator there are also more tool specific documentation provided. These either help with configuring the tool or assist how the Excel file should be constructed. Tool configuration documentation generally is about the options that can be used with the tools and what they do. Excel documentation lists the register functionality and how they should be done.

Code should be well commented. Commenting helps to maintain code modifiability. The longer the tool is in support state and does not require changes the harder it becomes to remember how it functions. Commenting key points is essential to extend the lifespan and usability of the tool. Comments in this tool are used to explain complicated parts of the code. It is no use to overdo commenting when the goal is that most of the code is easily understandable.

Help option is also important documentation. Good help option improves the user experience and it is usually expected to be found from Unix command line tools. Help lists the tool version, usable options and how they should be used. Here help option was made so that if options are added they are also automatically added to help with a default description so that the developer remembers to add the description later for all supported options.

5 RESULTS AND DISCUSSION

In this Chapter, the results of the project are presented and how the project can be developed in the future. Thesis research questions and proposed changes from section 3.2 Proposed Changes are revised and this chapter tries to analyze how well they were answered and solved.

5.1 Analysis

In this section the thesis project will be analyzed. How it was done? How did it succeed? Were there any downfalls or ingenuity? Most of the analysis was done by comparison by determining how the new tool compared against the old where same end result would be a success. However, some features were implemented first in this tool and they were tested in verification.

5.1.1 Progression of Development

The development was done in Unix terminal environment. Code functionality was mostly tested with the traditionally printing to the stdout because no IDE was used as the author was able choose and build their own development environment. There were a lot of register description files available which were used to test the functionality. From a register description file IP-XACT was generated and then that was used as an input to the tool. The description files usually covered some specific type and functionality of register but there was also a few generic description files which covered a wider range of different functions. With the more generic description files coverage checks could be made. To test some function specifically a description file could be made for it and the functionality of the code could often be seen if the resulting VHDL file looked as expected. Testing was, however, limited to test cases which were available and those that were specifically made to test some functionality. There was no test which could test everything at once.

VHDL files were compared at first. Same VHDL was generated with the old tool and the same with the new tool. Then they were first compared manually side by side. This spotted major and easy mistakes like ports, signals and processes missing but missed a lot hard to spot mistakes. These hard to spot mistakes were such as typos, signal boundaries not aligning and register address calculations not being correct. This also led to a lot of assumptions which were later proven incorrect in more extensive testing and when

Table 5.1. *Approximated progression of test cases used to test functionality of the tool.*

Index	Test Case	Feature	Similarity
1	dummyVHDL	Produce a file with port assignments.	100%
2	oneReg	Produce a functional register.	100%
3	regOr	Register functionality for or-operation	100%
4	regWRX	Different read and write modes for registers.	100%
5	protocolAXI	Implementing AXI4-Lite protocol.	100%
6	regFunc	Rest of the register operation functionality.	80%
7	protocolAvalon	Implementing AvalonMM protocol.	new
8	remainingProtocols	Implementing rest of the protocols.	0%

the tool was more familiar to the developer. Comparison testing was a mistake in this project which should have been avoided with automated testing. Although, comparison testing was a rough starting point and then VHDL compilation spotted more mistakes.

After more blatant mistakes were fixed synthesis tools were used to spot the rest. VHDL compilation gave errors for typos and signal misalignments. After successful VHDL compilation testbenches and waveform tools were used to check basic functionality of the registers. Waveform simulation tool was used to check basic functionality and that the design works after turning on clock and toggling reset. RTL simulation tool was used to check that there were no floating pins and the RTL looks sensible. RTL simulation tool also gives the critical path but it can vary from design to another.

Some minor stress testing has been also done. During the development a database was created containing 1000 times the same component and a VHDL file was generated each time. With smaller database the tool runs notably faster but as the size of the database starts to grow the tool seems to slow down. So the time that it takes to fetch information from the database seems to be linear to its' size. The average run over all 1000 runs was 2.983 seconds for xml2database and for database2vhdl it was 3.146 seconds. This equals to 6.129 seconds which is still really fast compared to fetching the XML schema data from source which took over a minute if not multiple in tests. The average has to be taken with a grain of salt since the time is very dependent on the design which in the test 5 from Table 5.1 was pretty complicated. This test was done to prove that even though the performance of the database gets worse as its' size increases the run-time of the tool stays within a sensible limit. Thankfully the run times of the tool are not critical and optimizations are done where they are possible. There are Python libraries for big data and intensive math operations which optimize for performance and those are used in places where it is possible.

Development has progressed well. The development however is still on going and will still continue after this thesis. The basic features which are most commonly used and which produce the scope of this thesis are implemented. After the new version of the register generator is equal or surpasses the current one in features the new register generator can be taken into use as the main solution. Before that users have to explicitly want to use the new register generator.

When the new register generator is taken into use it goes into support phase where the tool sees only a little development. Bug reports come from user experience as do requests for new features. In support phase users submit bug reports or feature requests and then they are given an evaluation and the needed time. New feature requests are evaluated i.e. is the user base for a feature wide enough that it is worth the time used for implementing. After evaluation if the feature is chosen to be implemented development time is distributed for it.

The project has been successful so far without major obstacles. Difficulties in development usually come from some feature needing additions or changes to the database. With those kind of changes one has to be really careful. Changes to the database could create a rippling effect of bugs because the program expects some order from the data fetched from the database. This is because fetching from the database returns a list of tuples and not a list of dictionaries for example.

5.1.2 Verification

During this thesis project a new IP bus protocol AvalonMM was requested and its' interface was implemented first in this new version of the tool. The protocol functionality was implemented and sent to verification for engineers who requested the new protocol. This also worked as a verification for new register generator tool. The results for both, protocol and tool, have been positive meaning there has not been much problems so far. Verification has found test cases which either crashed the tool or produced non-synthesizable VHDL. These reports have been reacted to and fixes have been made accordingly.

Verification for the tool has been done also in development. This means a comparison between produced files on the old register generator and the new one. Comparison can be either by developer looking at and comparing the files or by using formal verification methods. Also simple simulation tests are done that prove that registers function as expected and they can be written to and read from using ports and protocol interface. Most of the verification is done outside development by verification engineers because verification time is time off development.

Occasional coverage tests have been done to verify code functionality and quality of the tool. However, coverage test results are dependent on the test case quality. There should be tests case to try every function of the tool. However, these test would prove to be massive since there are so many options which would require separate runs to test (e.g. registers having synchronous or asynchronous reset). Linters which verify the

quality against some standard were not used during this project but were later taken into use. Example of these linters is Flake8 for Python which tests Python code against the PEP8 standard.

5.1.3 Meeting the Goals

Project has met the goals set in Section 3.2 Proposed Changes. This database based tool is available alongside the old register generator and can be used to provide AvalonMM protocol interface which the old register generator currently cannot do.

The phase from IP-XACT to VHDL contains no more vendor locks. The tool is fully developed by the author of this thesis using Python3 excluding used Python packages. Therefore there are no vendor locks, tool no longer reserves any licenses and the company is now fully in control of the phase from IP-XACT to VHDL. Which is a great success since now those tool licenses which were used in VHDL generation can be used to something else.

Database approach has been successful. An IP-XACT document can be read into a Python datastructure and a lossless database can be generated from it. No template or pre-known information is used in creation of the databases so the database is created completely based on the structure of the IP-XACT document. The resulting database is also lossless so going back to IP-XACT from a database is possible. XML can also be generated back from the database but at the moment no data is modified or inserted to it.

VHDL can generated from the IP-XACT information stored in the database. In VHDL generation pre-known information about the IP-XACT elements is used to fetch the scattered register information. Temporary tables are then constructed which contain the register information in more easily accessible format. The VHDL file is then printed from these temporary tables.

Project associated with the thesis has answered the research questions. First IP-XACT is interpreted by a Python library which uses an XSD to parse the document into a Python datastructure. Good command line interface tools are made with good user experience, version control, options and environment variables. In the scope of this thesis databases were proven to be a good choice to be a intermediate file format through easy management and formatting. Excel is still used as a starting point and IP-XACT is generated from Excel template as in the old flow.

5.2 Future Development

In this section the future development plans are gone through. Future development includes new features, functions and tools around this project. This project itself is always in development.

The database approach enables possibility for a modification API i.e. a tool which takes commands to make modifications to the database. Such API could add, delete or modify contents of a register bank. This would prove especially useful when there are multiple versions of the register bank saved to the database. Then actions like "*Add a register field to every version of this register.*" could be easily possible.

First there needs to be a decision of the user interface. Will the API be completely script based or is there some user interaction during run time? One choice would be to open a template for modifications in an editor and then commit the changes to the database which were made to the template. For example if there are 100 versions of a component and they all have same register field and its' width is 4 bits. In the user's default text editor would then open up template *4*. When the template is corrected to 8, saved and quit then the width would be 8 in all 100 versions. The alternative would be a script approach where changes would be given in options and then the API would make the change. There is a drawback that with script approach there would not be a similar template to the editor approach. User would not possibly know the selected content before it is changed.

Current way that IP-XACT is stored to the database could cause problems in implementation of the API. The reason is the running number which is used in IP-XACT storage. Adding something to the IP-XACT portion of the database would mess up the running numbers. When adding more rows it would be required to keep track of the amount of added rows and in the end then increment all numbers greater than the last one before addition by number of rows added.

The IP-XACT which is an intermediate file format in this flow is also used it other flows. These flows modify the IP-XACT file and pass it forward. These flows would benefit from the IP-XACT being in a database. Needed modification would be quick to do to a database and the modified IP-XACT could be generated from the database. This newly generated IP-XACT could then be used in different flows.

Hopefully in the future there will be IP-XACT models for components which are from outside of the register banks' IP-XACT. Then the components could be interpreted to the database and there would be no need to parse VHDL entities and convert them to component introductions and instantiations. After parsing the IP-XACT to the database for external components they could be parsed similarly to the register bank information.

6 CONCLUSION

Register generation is widely used and important subject in the SoC design. Registers are fast run time memory of the SoC modules. In the register generator project registers have direct port and protocol interface to the registers which means that the register bank acts CPU's interface to the SoC modules. This means that register banks are a submodule within SoC modules. The generated register banks increase the reusability and maintainability of the SoC modules. The protocol can be changed or the contents of the register bank can be altered to fit the needs of a interconnect.

The main purpose of this project was to replace an older implementation of the same tool and at the same time make it more flexible. Register generator is a tool which designers use to create register banks for SoC modules. These register banks will have direct port and protocol interface to access registers. This means that registers can be accessed with a protocol and addressing. The old IP-XACT to VHDL phase was implemented in a programming language which was different from the other tools' languages in the flow while also providing inflexible intermediate file formats. These were the driving factors of re-implementing the phase from IP-XACT to VHDL.

In this thesis project a command line tool was made which can generate VHDL from IP-XACT register description files using Python and SQLite. Python is used as the main programming language and SQLite is used as a intermediate storage for register information. Python provides libraries for IP-XACT parsing, SQLite API and tool distribution. Python tools are installed to the Python interpreter. However in this project a single binary containing a Python interpreter and required Python packages is made for an easy distribution and user experience. SQLite database is used for intermediate file format to keep the data easily readable and modifiable.

Altogether the project was successful and results were satisfying. The tool is able produce a database similarly every run and it also produces the same file subsequently. The database can contain multiple register banks at the same time of which one can be chosen to be generated by an unique identifier. The register generator was tested in verification along with a new protocol implemented in this project and passed test cases from verification. In the end the tool of this thesis project has been taken into use alongside the old implementation.

REFERENCES

[1] Butterfield, A. and Szymanski, J. *A dictionary of electronics and electrical engineering.* eng. 5th ed. Oxford quick reference. Oxford, England: Oxford University Press, 2018. ISBN: 0-19-179271-3.

[2] Flynn, M. J. *Computer System Design System-on-Chip*. eng. Hoboken: Wiley, 2011. ISBN: 1-283-15735-7.

[3] Pasricha, S. *On-chip communication architectures system on chip interconnect*. eng. Systems on Silicon. Amsterdam ; Elsevier / Morgan Kaufmann Publishers, 2008. ISBN: 1-281-37094-0.

[4] Ferdjallah, M. *Introduction to digital systems : modeling, synthesis, and simulation using VHDL*. eng. Hoboken, New Jersey, 2011.

[5] Navabi, Z. *Digital System Test and Testable Design Using HDL Models and Architectures*. eng. 1st ed. 2011. New York, NY: Springer US, 2011. ISBN: 1-4419-7548-9.

[6] *Python 3.6.12 documentation. Python Documentation*. URL: https://docs.python.org/3.6/ (visited on: 21. Aug. 2020).

[7] Cassell, L. *Python projects*. eng. Wrox professional guides Python projects. Indianapolis, Indiana: John Wiley and Sons, 2014. ISBN: 1-119-20758-4.

[8] *Creation of virtual environments*. Python Software Foundation. URL: https://docs.python.org/3/library/venv.html (visited on: 21. Aug. 2020).

[9] *The Python Package Index. PyPi*. URL: https://pypi.org/ (visited on: 21. Aug. 2020).

[10] *PyInsaller Manual. PyInstaller*. URL: https://pyinstaller.readthedocs.io/en/stable/ (visited on: 6. Nov. 2020).

[11] *Regular Expression Operations. Regular Expression*. URL: https://docs.python.org/3/library/re.html (visited on: 20. Nov. 2020).

[12] *Extensible Markup Language (XML) 1.0 (Fifth Edition). XML*. Feb. 7, 2013. URL: https://www.w3.org/TR/2008/REC-xml-20081126/ (visited on: 28. Aug. 2020).

[13] *SCHEMA. XML TECHNOLOGY*. URL: https://www.w3.org/standards/xml/schema (visited on: 4. Sept. 2020).

[14] *XML Schema Part 0: Primer Second Edition. W3C Recommendation 28 October 2004*. Oct. 28, 2004. URL: https://www.w3.org/TR/xmlschema-0/#Intro (visited on: 4. Sept. 2020).

[15] *IEEE Std 1685-2014 (Revision of IEEE Std 1685-2009). IEEE Std 1685-2014 (Revision of IEEE Std 1685-2009): IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows IEEE Std 1685-2014*. eng. IEEE, 2014. ISBN: 0-7381-9226-0.

[16] *IP-XACT Working Group. IP-XACT*. URL: https://www.accellera.org/activities/working-groups/ip-xact (visited on: 4. Sept. 2020).

[17] *IP-XACT User Guide. IP-XACT*. Mar. 2018. URL: https://www.accellera.org/images/downloads/standards/ip-xact/IP-XACT_User_Guide_2018-02-16.pdf (visited on: 4. Sept. 2020).

[18] Wilton, P. *Beginning SQL*. eng. Indianapolis, IN, 2005.

[19] *SQLite. Small. Fast. Reliable. Choose any three.* URL: https://www.sqlite.org/index.html (visited on: 18. Sept. 2020).

[20] *SQLite Tutorial. SQLite Tutorial*. URL: https://www.sqlitetutorial.net/ (visited on: 22. Dec. 2020).

[21] Poulos, K., Adaos, K. and Alexiou, G. Automated Generation of the Register Set of a SOC and its Verification Environment. Oct. 2014, 1–2. DOI: 10.1145/2645791.2645851.

[22] *Rggen. Code generation tool for ASIC/IP/FPGA/RTL engineers.* URL: https://github.com/rggen/rggen (visited on: 30. Oct. 2020).

[23] *Building an IP-XACT Desing and Verification Environment with DesignWare IP. By John A. Swanson, senior marketing manager, DesignWare IP, Synopsys*. URL: https://www.synopsys.com/designware-ip/technical-bulletin/design-verification-environment.html (visited on: 11. Feb. 2021).

[24] Kruijtzer, W., Wolf, P. van der, Kock, E. de, Stuyt, J., Ecker, W., Mayer, A., Hustin, S., Amerijckx, C., Paoli, S. de and Vaumorin, E. *Industrial IP Integration Flows based on IP-XACT™ Standards*. Tech. rep. URL: https://www.cl.cam.ac.uk/research/srg/han/ACS-P35/readinglist/kruijtzer.pdf (visited on: 11. Feb. 2021).