Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Kevin Aleksander Vinding

Automatic Extraction of Complex Bus Structures from RTL

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Per Gunnar Kjeldsberg
Co-supervisor: Berend Dekens, Eivind Fylkesnes
June 2021

NTNU
Norwegian University of
Science and Technology

Kevin Aleksander Vinding

Automatic Extraction of Complex Bus Structures from RTL

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Per Gunnar Kjeldsberg
Co-supervisor: Berend Dekens, Eivind Fylkesnes
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Abstract

Modern system on chip is becoming more complex, and as a result, harder to design, test, and verify. Consequently, the bus structure, connecting the various components in systems on chips, is more complex. This thesis aims to aid the development and verification of bus structures by extract information on the complex bus structures and present it in a beneficial and readily available way. The information is to be extracted directly from the RTL design, representing the actual implementation instead of an abstract representation from the documentation. Furthermore, the solution should be as simple and automated as possible to lessen the burden of implementing the solution in both new and existing designs. All this allows the users to explore and validate the bus system and the modules connected to the bus system, improving the workflow.

The thesis' solution is to exploit the simulation engine to extract bus-related information during simulation. The solution introduces small snippets of extraction code to enhance the already existing design. During simulation, the extraction code collects the information and exports it. A separate processing tool imports the extracted information and processes it into more user-friendly data. The user can then explore the bus structure and generate diagrams depicting connections and relationships between modules in the system.

A case study was performed on the solution, implementing it on a complex bus system provided by Nordic Semiconductor. The results are promising, as the solution can extract information quickly and easily. The extracted information matches the available documentation for the system as well. The solution requires access to the modules connected to the bus system and the bus system itself and is best suited for systems with the source code available for the user. The solution has the potential for expansion with additional functionality and types of information it can extract.

Sammendrag

Moderne system on chip blir mer og mer kompliserte, og av den grunn, vanskeligere å designe, teste, og verifisere for. En konsekvens av dette er at busstrukturene som kobler de forskjellige komponentene i chipen sammen har også blitt mer komplisert. Formålet med avhandlingen er å ekstrahere informasjon rundt komplekse buss strukturer og presentere det på en nyttig og lett tilgjengelig måte. Informasjonen hentes direkte fra RTL design, noe som representerer den faktiske implementasjonen framfor en abstrakt representasjon hentet fra dokumentasjonen. Dessuten skal løsningen være enkel og automatisk slik at det er mindre arbeid å iverksette løsningen i både nye og eksisterende design. Alt dette lar brukeren utforske og validere bussystem og moduler koblet til bussystemene, noe som forbedrer arbeidsflyten.

Avhandlingens løsning er å utnytte simuleringsmotoren for å hente ut bussrelatert informasjon under simulasjon. Løsningen oppnår dette ved å introdusere små kodebiter for å gjennomføre uthenting av informasjon i designet. Under simulasjon vil koden samle sammen informasjonen og eksportere den. Et separat verktøy importerer denne informasjonen og prosesserer den til mer brukervennlig data. Brukeren kan da utforske bussystemet og generere diagram som beskriver koblinger og forhold mellom moduler i systemet.

En casestudie ble gjennomført, hvor løsningen ble implementert i et kompleks bussystem fra Nordic Semiconductor. Resultatene er lovende, hvor løsningen kan hente ut informasjon raskt og enkelt. Den ekstraherte informasjonen stemmer godt overens med den tilgjengelige dokumentasjonen for systemet. Denne løsningen krever tilgang til modulene som er koblet til bussen, samt bussystemet selv, og passer dermed best for systemer med tilgjengelig kildekode. Løsningen har også potensialet for utvidelse, både for funksjonalitet og hvilken type informasjon som er ekstrahert.

Preface

This thesis was written in the spring of 2021 with no preliminary work in collaboration with Nordic Semiconductor, who provided the thesis with access to their bus systems and designs initially developed for their products. Throughout the thesis, it became clear that the potential for this thesis was more than what could be handled by one person over a semester. The most prominent issue is the case studies, whereas the thesis only considers a single case. The idea was to use an open-source SoC as additional cases, but there were no available open-source SoC to use. With little time left, no additional cases could be presented that would bring value to the thesis. The author has made an effort to describe potential features, additions, and changes in the future works chapter.

The author would like to thank Per Gunnar Kjeldsberg from the Norwegian University of Science and Technology for acting as supervisor for the thesis and supporting the writing of the thesis. The author would also like to thank Berend Dekens and Eivind Fylkesnes from Nordic Semiconductor for guidance and support in developing the solution presented in this thesis.

Contents

Abstract **v**

Sammendrag **vii**

Preface **ix**

List of Abbreviations **xvii**

1 Introduction **1**
- 1.1 Objectives . . . 2
- 1.2 Scope and Limitations . . . 2
- 1.3 Main Contributions . . . 3
- 1.4 Structure of the Thesis . . . 3

2 Background **5**
- 2.1 Busses . . . 5
- 2.2 AMBA . . . 10
- 2.3 SystemVerilog . . . 11
- 2.4 Simulations . . . 15
- 2.5 Parsing . . . 19
- 2.6 Related Work . . . 19

3 Discussion of Potential Solutions **21**
- 3.1 What Information Should Be Extracted . . . 21
- 3.2 Solutions . . . 25
  - 3.2.1 Parsing of HDL Code . . . 26
  - 3.2.2 Netlist Signal Tracing . . . 26
  - 3.2.3 Modified Simulation Tools . . . 27
  - 3.2.4 Extraction Through Simulation . . . 27

4 Extracting Bus Information **31**
- 4.1 Brief Overview of the Extraction Solution . . . 32
- 4.2 Implementing the Extraction Code in Original Modules . . . 35
- 4.3 Extraction Macros for Reading & Writing to Busses . . . 40
  - 4.3.1 Sequence of Operations . . . 41
  - 4.3.2 Write Operations . . . 42
  - 4.3.3 Read Operations . . . 44
  - 4.3.4 Relaying of Signals . . . 45
- 4.4 Extraction Macros for Interconnecting Modules . . . 47
- 4.5 Exporting the Extracted Information . . . 50

5 Processing Bus Information **53**
5.1 Brief Overview of the Processing Solution 53
5.2 Using the Processing Tool 54
5.3 Importing Extracted Information 58
5.4 Processing 59
5.4.1 Obtaining a List of Modules 60
5.4.2 Obtaining a List of Connections Between Modules 60
5.4.3 Obtaining a List of Masters and Slaves 61
5.5 Visualisation 65

6 Case Study - System from Nordic Semiconductor **69**

7 Future Works **75**

8 Conclusion **77**

A Extraction Source Code **79**

B Processing Tool Source Code **85**

Bibliography **103**

List of Figures

2.1 An abstract example of a system that uses a bus for communication (left), and one that does not (right). . . . 6
2.2 Example of a centralised bus implementation. . . . 6
2.3 Example of a distributed bus implementation. . . . 7
2.4 Example of several busses connected by bridges. . . . 8
2.5 Example of an AHB matrix bus system, taken from [1] . . . 8
2.6 A bus split into data-, address-, and control bus, which is further split into read and write busses. . . . 9
2.7 Example of an AMBA bus system with an AHB and APB connected via a bridge. . . . 10
2.8 Example of simulation time being much slower than real time. . . . 16
2.9 All event regions in a single simulation time unit, taken from [2]. . . . 17
2.10 Example of how events in the active region are handled. . . . 18
2.11 Simple example of parsing data into a data structure using a set of rules. . . 19

3.1 Example of connection information from a simple bus structure. . . . 22
3.2 Example of connection information on a bridge between two simple bus structures. . . . 22
3.3 Example of module-related information from a simple bus structure. . . . 23
3.4 Example of bus related information from a simple bus structure. . . . 23
3.5 Example of interconnection information from a simple bus structure that consists of a matrix. . . . 24
3.6 Example of address map information, where only Module B can access a specific part of Memory Module C. . . . 25
3.7 Extraction code is placed in design files, and produces bus information through simulations. . . . 28
3.8 The flow of the split solution, where one part handles extraction and the other handles the processing. . . . 29

4.1 Example of a stimulus signal on a bus causing two of the modules to respond. 32
4.2 A bus with a processor and a couple of RAM modules. . . . 32
4.3 A bus with a processor and a couple of RAM modules, organised in files. . . . 33
4.4 The bus system from Figure 4.3, with extraction code implemented into the modules. . . . 34
4.5 The bus system from Figure 4.4 during simulation. . . . 34
4.6 Simple diagram of a interconnecting module, with two masters and two slaves connected to it. . . . 36
4.7 Sequence of operations during extraction, where each type of operation is assigned to a specific delta-cycle. . . . 41

4.8 The signal is relayed across the bridge by associating the input value with the output value. 46

5.1 The interface of the processing tool when opened. 54
5.2 List of available commands for the processing tool. 54
5.3 Example of the processing tool importing a file at startup. 55
5.4 Example of the processing tool importing a file. 55
5.5 Example of the *listcon* command, listing all connections in the bus structure. 56
5.6 Example of a connection diagram generated from extracted data using the *gencon* command. 57
5.7 Example of a master-slave diagram generated from extracted data using the *genmas* command. 58
5.8 Example of two nodes connected by an edge. 65

6.1 Block diagram of the application core of the nRF5340 Bluetooth SoC, taken from [3] 70
6.2 The resulting block diagram showing connections between masters and slaves for the Nordic Semiconductor subsystem. 73

Source code

2.1 An initial procedure block. 11
2.2 Example of an delay of 10 time units. 11
2.3 An *initial* block with a *fork-join* construct. 12
2.4 An *initial* block with a *fork-join* construct containing two process that run concurrently. 12
2.5 Example of *force* and *release* statements 13
2.6 Example of the *$display* task. 13
2.7 Example of the *$display* task with format specification. 13
2.8 Example of a macro using *'define* 14
2.9 Example of a macro using *'define* and an argument list 15
2.10 A nonblocking assignment that can cause an update event. 15
4.1 Template for where to place extraction code in an arbitrary module. 35
4.2 Example of a interconnect-matrix, defining which port is connected to each other. 37
4.3 Example of extraction code placed in an interconnecting module. 38
4.4 Example of extraction code placed in a master module. 38
4.5 Example of extraction code, where a signal is relayed across a bridge module. 39
4.6 Function for getting a unique integer value. 40
4.7 Macro for writing an value to a data bus. 42
4.8 Macro for releasing a data bus after a *force* assignment. 43
4.9 Collection of force, read, and release operations for a single data bus. 43
4.10 Collection of force, read, and release operations for an array of data busses. . 44
4.11 Macro for reading a data bus and exporting the information. 44
4.12 Macro for reading an array of data busses and exporting the information. . . 45
4.13 Macro for relaying a signal across a module by associating the value on the data bus going in and out. 46
4.14 Macro for relaying signasl across a module by associating the value on the data bus arrays going in and out. 47
4.15 Macro for mapping out the connections between bus arrays. 48
4.16 Macro for marking whether the bus is intended for masters or slaves. 49
4.17 Macro for marking whether the array of busses are intended for masters or slaves. 49
4.18 Template for structuring data points from extraction processes. 50
4.19 Macro for writing a data point to the standard output. Accepts up to two values. 51
4.20 Macro for writing a data point to the standard output. Accepts a value and a string. 51
5.1 Function for converting a string into a data point. 59
5.2 Function for obtaining a list of modules from raw data. 60

5.3 Function for obtaining a list of connection between modules from raw data. . 61
5.4 An excerpt from the *Bus* class detailing the values stored in the class. 61
5.5 First step of *get_master_list*: Obtain a list of interconnecting modules. . . . 62
5.6 Second step of *get_master_list*: Obtain information produced by interconnecting modules and relays. 63
5.7 Third step of *get_master_list*: Group values from relays modules with the interconnecting ports. 63
5.8 Fourth step of *get_master_list*: Determine which masters and slaves are connected to the busses. 64
5.9 Fifth step of *get_master_list*: Determine which slave is connected to which master. 64
5.10 Function for generating a diagram of modules and connection between them. 66
5.11 Function for generating a diagram of masters, slaves, and their relationships. 67
6.1 The extraction code applied to *AHB_multi_layer_top.sv* 71
6.2 The extraction code applied to *cpu_top.sv* 71
6.3 The extraction code applied to *subsystem_top.sv* 72
6.4 The extraction code applied to *AHB_bridge_top.sv* 72
A.1 bus_info_extract.sv 84
B.1 bus_tool.py 92
B.2 bus_info_processor.py 99
B.3 bus_info_parser.py 101
B.4 datapoint.py 102

List of Abbreviations

ACE AXI Coherency Extensions

AHB Advanced High Performance Bus

AMBA Advanced Microcontroller Bus Architecture

APB Advanced Peripheral Bus

API Application Programming Interface

ASB Advanced System Bus

AXI Advanced eXtensible Interface

CHI Coherent Hub Interface

CPU Central Processing Unit

DMA Direct Memory Access

HDL Hardware Description Language

IP Semiconductor Intellectual Property

NBA Nonblocking Assignment

PDF Portable Document Format

RAM Random Access Memory

RISC Reduced Instruction Set Computer

RTL Register-transfer Level

SoC System on Chip

VHDL VHSIC Hardware Description Language

VLN Virtual Logic Netlist

XML Extensible Markup Language

Chapter 1

Introduction

With the increasing popularity of system on chip (SoC), we see more and more modules interconnected in complicated structures. Often these connections are made using busses; They allow data and signals to transfer between modules by having a single connection running between them. The modules have to agree on when and who can use the bus. The benefit, however, is a reduction of wires running between the ever-increasing amount of modules in a system, compared to having individual point-to-point connections [4].

A bus system design usually starts as a specification describing how the different modules should communicate at a relatively high abstraction level. The bus system follows these descriptions to ensure that it behaves correctly. Unfortunately, one specification does not mean just one possible solution; One can have systems that behave the same but are entirely different. Consequently, anyone who wants a more detailed overview of a bus system must review code and simulation reports to piece this information together. With increasingly large projects, the effort necessary to obtain this information escalates with the size of the system. [4, 5, 6].

The fact that there are many people involved in the development of systems amplifies the problem. With extensive and complicated systems, the teams developing them must be larger than ever. Most systems require not only a large team but also people from various disciplines as well. Everyone contributes with their share, translating a description of intended behaviour into an implementation, making it hard to accurately view the bus system's state.

A simple solution could be to manually review the design and extract information around the bus system. The downside of this is that it is costly, slow, and prone to human error. A far more efficient and attractive solution would be to have this task automated. Having a tool at one's disposal to extract the needed information directly from the design could be a great asset to any workflow.

1.1 Objectives

This thesis aims to solve the problem of getting a good grasp of complex bus structures from intricate design. In other words, it needs a way of both acquiring and present information related to bus systems that is beneficial to the user. A good example of bus-related information is the structure of the bus systems, describing which modules can communicate through the bus. This information can, in turn, support the development of the design and evaluate the busses themselves.

The problem presents several issues that a potential solution must address. One issue is obtaining the information related to busses; The potential solution needs a reliable way of extracting the desired information from the intricate designs. It is also essential that the extraction is universal; The solution should be applicable for different designs without significant changes to the solution or the design. What specific information will be of interest to extract needs to be defined as well.

Depending on the extraction solution, there might be a need to process the extracted information as well. There can be various reasons for it: there could be significant amounts of unorganised information, it could be in a non-readable state, and so on. Naturally, this is an issue that is highly dependant on the rest of the solution but is worth considering.

1.2 Scope and Limitations

This thesis will present a solution that addresses the issues described in Section 1.1. The goal is to thoroughly describe how each issue can be solved, with the collective effort recognised as the complete solution. Since there are no known previous attempts at solving this problem, arguing for the solution through a comparison would be difficult. This thesis will perform case studies as an alternative, pitting the solution against various complex bus structures.

A designer creating hardware typically uses hardware description languages (HDLs). There are several languages available, with VHDL and Verilog as the dominant ones. The bus systems used in the case studies, and the system they are implemented in, have been written in SystemVerilog. As such, the thesis will use SystemVerilog as its HDL. Besides, SystemVerilog brings many enhancements to the Verilog language, which can be beneficial to the solution [7, 2].

There are many different types of busses available for the designer, with some more popular than others. As such, this thesis will focus on the AMBA bus; This is an open-source communication standard widely used in the industry [8]. It will also support the thesis to have a consistent reference point throughout the solution. Even though this thesis uses a specific communication standard, the general solution should be translatable to other standards.

1.3 Main Contributions

- Presenting possible solutions for extracting information related to bus systems from RTL design.
- Developed a solution for extracting information related to the bus system by exploiting the simulation engine.
- Developed a solution for processing the extracted data and visualise it through automatically generated diagrams.
- Preformed a case study of the proposed solution using complex bus systems provided by Nordic Semiconductor.

1.4 Structure of the Thesis

Chapter 2 introduces the various background theory and information recommended for understanding the thesis, including related works. Chapter 3 discusses what information would be beneficial to extract, possible solutions for doing so, and what solution this thesis proposes. Chapter 4 presents the first half of the solution: extracting information from RTL. Similarly, Chapter 5 presents the other half of the solution: processing and exploring the extracted data. Chapter 6 performs a case study, evaluating the solution against more complex bus structures in a system provided by Nordic Semiconductor. Chapter 7 describes possible improvements and addition that can be made to the solution in future works. Finally, Chapter 8 presents the conclusion to the thesis.

Appendix A contains the source code for the extraction solution described in Chapter 4. Similarly, Appendix B contains the source code for the processing tool described in Chapter 5.

Chapter 2

Background

This chapter describes the theory behind the various concepts and tools that the thesis utilises. The aim is to provide one with the necessary background for understanding the rest of the thesis.

Section 2.1 describes the theory behind busses and typical components in busses. Section 2.2 describes the AMBA standard and its various protocols for bus communication. Having an overview of the AMBA standard is helpful for the cases study in Chapter 6, which uses AMBA busses. Section 2.3 describes the SystemVerilog language and specific properties in detail, which are extensively used in the thesis. Section 2.4 summarises the theory behind hardware simulation. The proposed solution of the thesis utilises key features of hardware simulation where a thorough understanding is helpful. Finally, section 2.6 presents various papers describing related works to the thesis.

2.1 Busses

Busses are the most used communication type between components on-chip, such as processors, memory modules, and hardware accelerators. It is defined by having all components connect to the same communication system. Thus, one can consider the communication system a shared resource that components can reserve for communicating with other components. From an abstract view, one can consider a single collection of wires that all components take turn using. The benefit of this approach is the significant reduction in wires needed to establish communication between many components. The alternative would be to have a specific connection between each component that should communicate. Figure 2.1 shows both cases, where it is clear that the system not using a bus requires many more connections. In addition, a bus system is easy to expand, where new components only have to connect to the existing bus system, whereas non-bus systems need a connection to each component it wants to talk to [4].

One typically categorise each component connected to a bus as a master or a slave. Master components are the ones that initiate communication with other modules, where it typically wants to supplement or request information. A processor is an excellent example of components that, in most cases, are categorised as a master. The role of the slave, on the other hand, is to do whatever the master wants it to do. An example of a slave could be a memory component, where a master can request data residing in the memory. There are also cases where a component can be considered both a master and a slave. A good example is a DMA; It requires instructions to know what data to fetch, hence a slave, but also be able to initiate communication with the memory, hence a master [4].

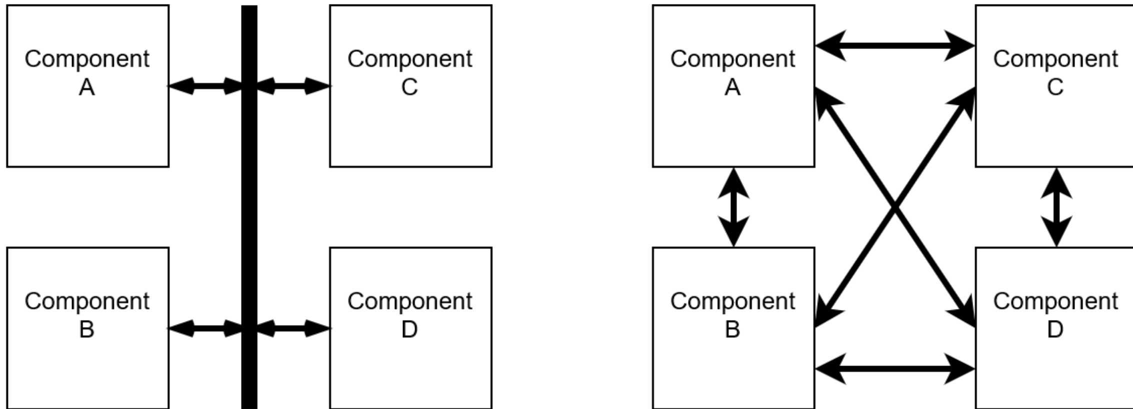

Figure 2.1: An abstract example of a system that uses a bus for communication (left), and one that does not (right).

The idea of all components connecting to the same wires is more of an abstract representation of a bus. Such a solution would rely heavily on the components to not interrupt each other. In practice, a bus system will consist of several other components to take care of the access and arbitration; Two prominent ones are the arbiter and decoder.

The arbiters job is to allocate access to the bus for masters. When multiple masters request access simultaneously, it is up to the arbiter to mediate. An arbiter typically follows a scheme for determining access, such as a priority scheme where one master can have a higher priority than others. The decoder's job is to determine which slave the master is trying to communicate with; This is typically a case of translating the destination address from the master and picking the slave corresponding to that address [4].

Figure 2.2 shows a simple bus system that uses an arbiter and a decoder. Bus systems that have a single pair of arbiter and decoder that supports all components, as in Figure 2.2, is called a centralised implementation. An alternative is a distributed implementation, shown in Figure 2.3, where each component has an arbiter or decoder; The advantage is that modules need fewer wires going in and out, but it requires more logic to implement.

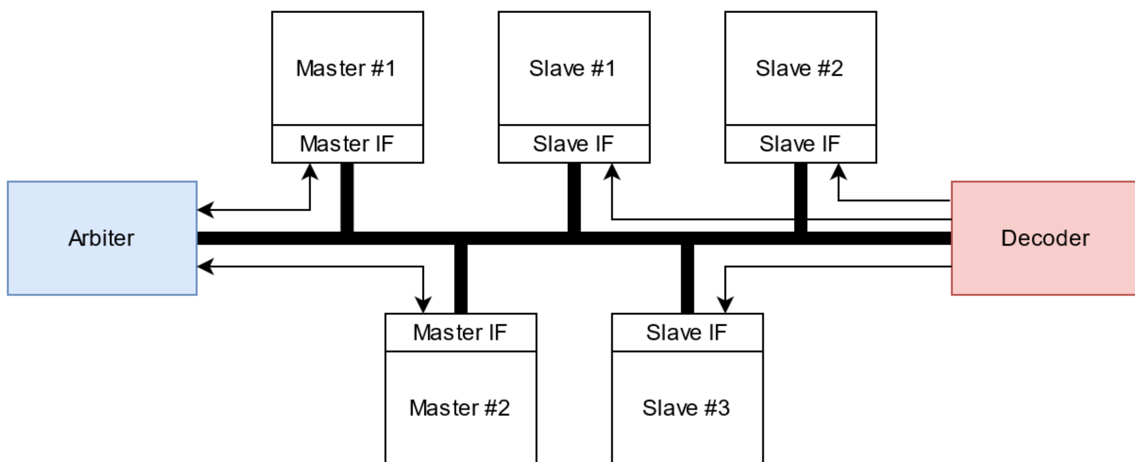

Figure 2.2: Example of a centralised bus implementation.

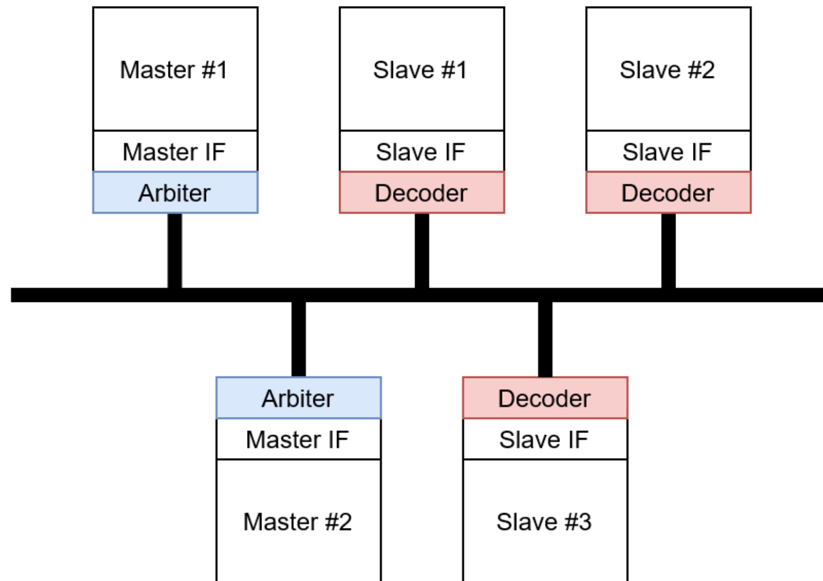

Figure 2.3: Example of a distributed bus implementation.

At its simplest, a bus system can be a single bus connected to all components. Only a single master can use the bus, and thus all other components must wait their turn. If there is little traffic on the bus or requests are evenly distributed, such a system is probably enough. In other cases, one can have components that need frequent access to specific slaves or have long periods of communications at the time. It can be beneficial to move those components to their own busses, freeing up the original bus to work in parallel [4].

Consider the case where one have several busses, and components on different busses need to communicate. The solution is to use an intermediate component called a bridge. Its purpose is to translate and transmit communication between different busses. Figure 2.4 shows an example where multiple busses are connected using bridges, also referred to as a hierarchical topology. A great feature of using bridges is that busses based on different protocols and implementations can communicate. For example, it is not unusual to have components residing in different domains with different frequencies or voltages. Another example is when specific components use a power-saving bus, while others use a bandwidth-optimised one. A bridge allows one to translate the communication from one domain or protocol to another [4].

An alternative to structure the bus system hierarchically, typically in cases that desire high bandwidth, is the matrix topology. In short, the components connects to multiple busses that run in parallel. Access to busses is allocated as needed, allowing communication to occur in parallel [4, 1]. Figure 2.5 shows an example using an AHB matrix (from the AMBA protocol [8]). The matrix topology allows both processors to access various slaves simultaneously since the bus system can allocate them to different busses.

There are nowadays many bus protocols defining the properties and operations of a bus system. There can be many different properties defined in a protocol; The width of the busses, how data transfer, and how communication is initiated and performed are a few examples. There are many different bus protocols available, with some more popular than other [9]. Some examples are the Wishbone bus made by Silicore Corporation [10] and the various AMBA standards made by ARM [8].

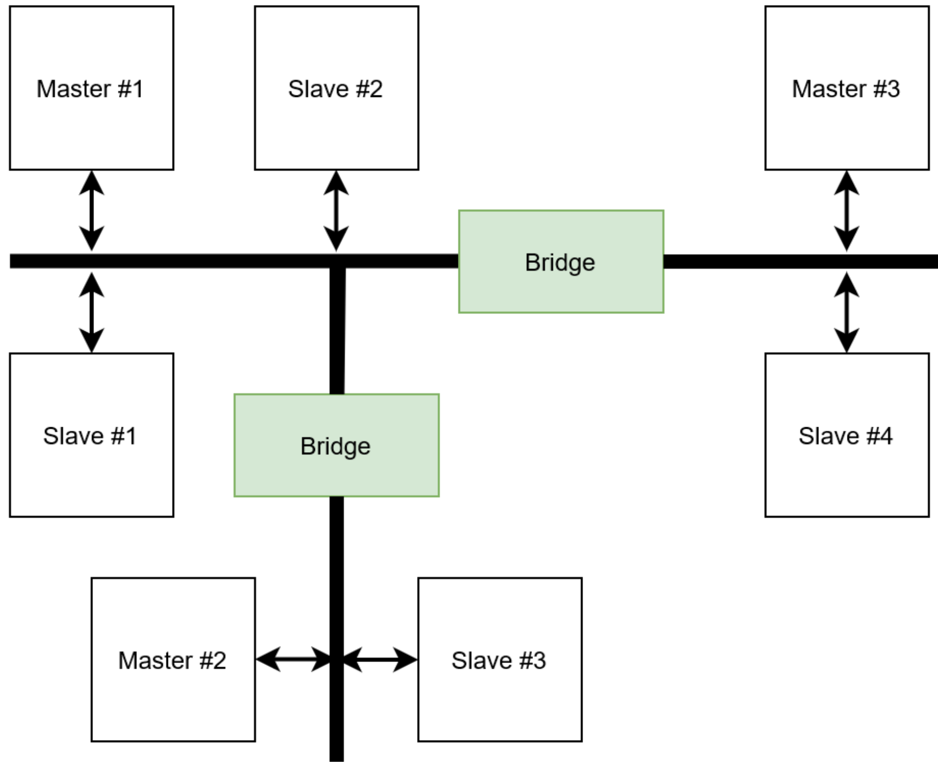

Figure 2.4: Example of several busses connected by bridges.

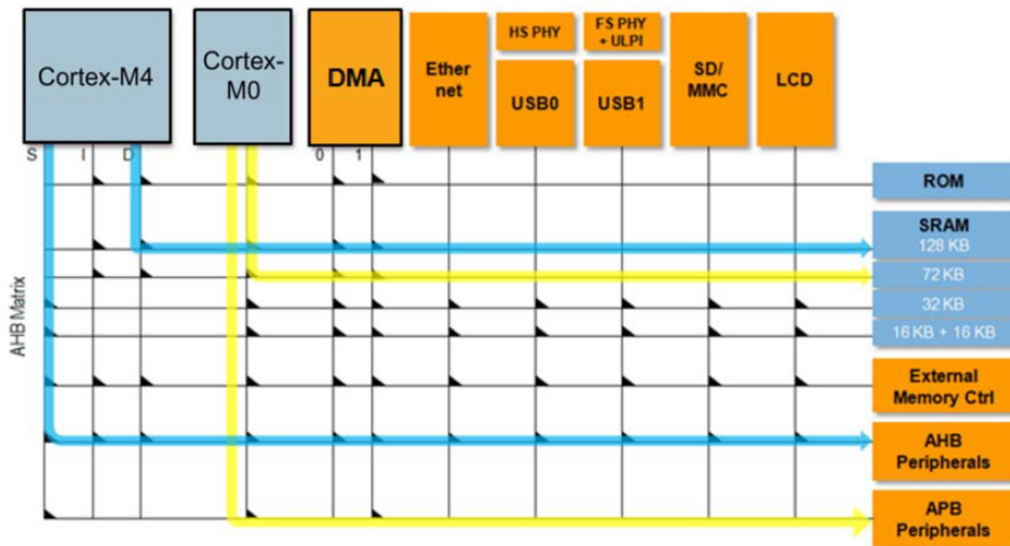

Figure 2.5: Example of an AHB matrix bus system, taken from [1]

While one can consider the bus as a single collection of wires, it is not unusual to split the bus into three smaller sub-busses, each with its purpose. They are typically named data-, address-, and control bus, as depicted in Figure 2.6. The data bus purpose is to transmit data values between components. The address bus purpose is to transmit the destination of the data values. Finally, the control bus contains all signals that do not fit with the other two sub-busses. The protocol typically defines what the control bus contains, for instance, signals for requests from masters and acknowledgement from slaves. It is not unusual for protocols to split the data- and address bus further into individual read- and write busses, improving concurrency [4]; This is also shown in Figure 2.6.

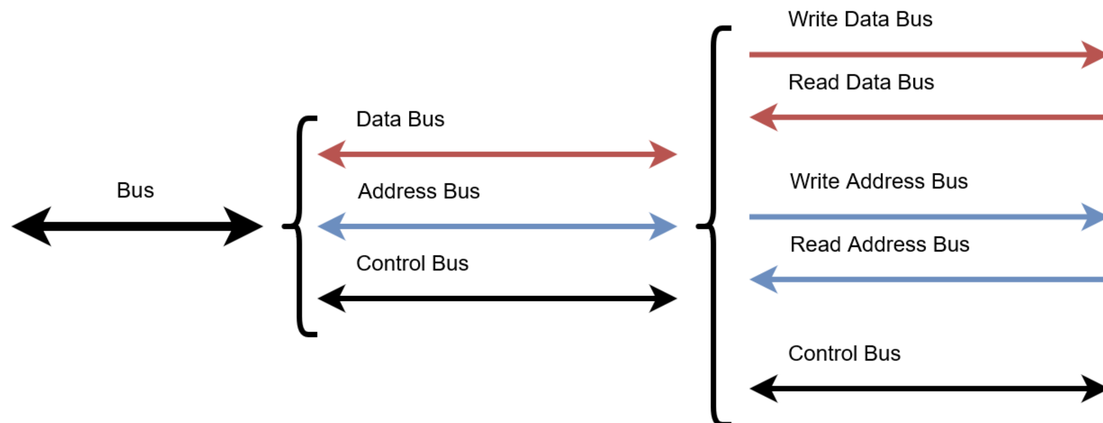

Figure 2.6: A bus split into data-, address-, and control bus, which is further split into read and write busses.

2.2 AMBA

AMBA [8], or Advanced Microcontroller Bus Architecture, is a widely used, open, on-chip communication standard. Arm developed the first version of the standard in 1995 and has since had several iterations with AMBA 5 as its newest. The core idea behind AMBA is reusability, addressing the problem of using ad hoc solutions for communication between modules. Utilising a modular design with flexible interfaces, AMBA helps reducing design time and allows the design to be more portable [11, 12].

The AMBA 2.0 standard introduces the AHB (Advanced high-performance bus) and APB (Advanced peripheral bus) protocols. AHB is designed for high-performance modules, acting as the backbone of the bus system, while APB is intended for low-power peripherals. The system can save power by placing peripherals on an APB and use bridges for communication with the more high-performing AHBs [4, 13]. Figure 2.7 shows an example of an AHB and APB connected through a bridge.

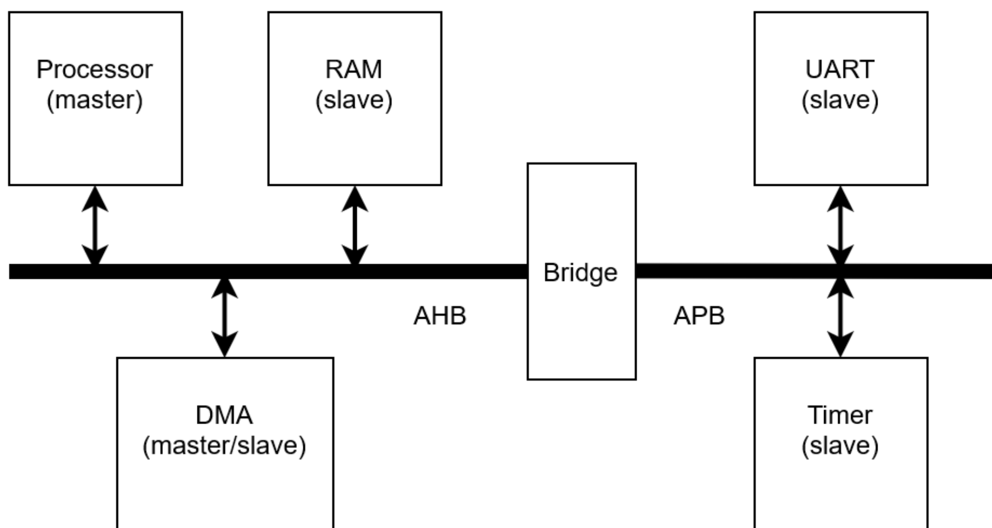

Figure 2.7: Example of an AMBA bus system with an AHB and APB connected via a bridge.

A variant on the AHB protocol is the Multi-layer AHB [14]. It is based on the AHB protocol and implements a matrix for concurrent communication; This allows for multiple transactions to occur concurrently, as described in Section 2.1 and Figure 2.5.

2.3 SystemVerilog

This thesis uses the SystemVerilog language to describe hardware. Since it contains many valuable features used heavily in the thesis, it will be beneficial to explore these features in detail. SystemVerilog is an extension of the Verilog language. Its purpose is to expand on the language and add several valuable features, such as improved verification and assertions. The Accellera Standards Organization first developed SystemVerilog before IEEE Standards Association adopted it [7].

It is important to note that SystemVerilog, along with Verilog, is a language intended to describe digital circuits and systems. They differ from ordinary programming languages in a few aspects, but the most noteworthy is the execution of the code. Traditional languages, such as C, execute one statement at a time, while Verilog and SystemVerilog allow for statements to execute in parallel. This sense of concurrency in HDL is to emulate hardware behaviour, which is concurrent by nature. However, note that code within blocks (e.g., *initial*, *always*) are still executed sequentially. Since SystemVerilog and Verilog have been made with hardware in mind, it is less suitable for more traditional, high-level task associated with languages such as C [15, 16].

Initial procedure

The *initial* construct declares a procedure that runs at the start of the simulation and only runs once. A *initial* procedure can last beyond the zero time slot through delays. Typically, one would use this procedure to initialise the system and prepare for the rest of the simulation. Listing 2.1 shows how an *initial* procedure block will look like; Code that should run from the start of the simulation is placed within the block [2].

```
1 initial begin
2     // Code that should run at the start of the simulation, and only once.
3 end
```

Listing 2.1: An initial procedure block.

Delay

The delay control allows the simulation to postpone the execution of a process. It is defined by the # symbol, followed by the delay value in simulations time units. Listing 2.2 shows an example where an initial process is postponed by 10 time units.

```
1 initial begin
2     // Code executed at time 0
3     #10;
4     // Code executed at time 10
5 end
```

Listing 2.2: Example of an delay of 10 time units.

There is a special case for delay control when the delay value is zero; This is called a zero delay. The code after a zero delay is still executed within the same time slot, but the simulation postpones the process by a delta cycle. In other words, the execution of the process is postponed until other processes within the same time slot is finished executing (or delayed themselves) [2]. Section 2.4 explains delta cycles and the benefit of zero delays in more detail.

Fork-Join

Statements inside a block are executed sequentially by default. However, one can create concurrent processes inside a block by the use of the *fork-join* constructs. As an example, consider Listing 2.3; An *initial block* has, in addition to its statements, a *fork-join* block that also have statements inside. The statements outside of the *fork-join* will execute sequentially, while the statements inside the *fork-join* will execute concurrently.

```
1  initial begin
2      // Statements that run sequentially
3      fork
4          // Statements that run concurrently
5      join
6  end
```

Listing 2.3: An *initial* block with a *fork-join* construct.

It is possible to take it even further and create concurrent processes inside the *fork-join* blocks, whose statements run sequentially. Listing 2.4 shows an example where two separate processes are executed concurrently inside the *fork-join* block. The statement run sequentially inside the process, but concurrently with the other process [2].

```
1  initial begin
2      // Statements that run sequentially
3      fork
4          begin : process_1
5          // Statements that run sequentially, but concurrently with process_2
6          end
7          begin : process_2
8          // Statements that run sequentially, but concurrently with process_1
9          end
10     join
11 end
```

Listing 2.4: An *initial* block with a *fork-join* construct containing two process that run concurrently.

Force and Release

The *force* assignment is a type of continuous procedural assignment that has the special property of overriding any value already present on variables or nets. Once a variable or net receives a forced value, it can only obtain a new value by another *force* assignment. To remove the effects of the *force* assignment, one has to use the statement *release*. Once a variable or net is released, it will keep its assigned value until another assignment is executed. However, should the variable or net be under the influence of another continuous assignment, the release will cause it to revert to its original value. Listing 2.5 shows an example where a variable var_1 is assigned a value through the *force* assignment, before it is released again [2].

```
1 force var_1= val_1;
2 // var_1 is assigned the value val_1 until it is released.
3 release var_1;
```

Listing 2.5: Example of *force* and *release* statements

Display

The *$display* task prints text and values to the standard output of the simulation, which is typically a console or a log. It also appends a newline-character at the end of the information, as opposed to the *$write*, which appends nothing. It is a pretty simple task, as one only has to provide the information to print. Listing 2.6 shows an example of a string that is printed to the standard output.

```
1 $display("This string is printed to the standard output.");
```

Listing 2.6: Example of the *$display* task.

It is possible to inject values into the string printed by the *$display* task. Many different escape sequences allow one to inject various data types. Consider the example in Listing 2.7; Using the escape sequences *%s* and *%h*, we can inject a string and a hexadecimal value into the printed string, respectively [2].

```
1 logic [31:0] val_1 = 32;
2 $display("The variable %s has the value %h.", "value 1", val_1);
3 //This will print: The variable value 1 has the value 00000020.
```

Listing 2.7: Example of the *$display* task with format specification.

Macro

SystemVerilog includes a variety of compiler directives to include code into existing files. One that is of significant interest for this thesis is a text macro substitution facility called *‘define*. Text is associated with a macro name, and the macro name is substituted with the text during compilation. In other words, it behaves similar to copying and pasting code into source files. A handy application is that one can use *‘define* to associate blocks of code with a macro name. Listing 2.8 shows an example of a macro used to display a string. We associate the *$display* task with the name *print_a_string*, which allows us to inject the code wherever we want by calling that macro. Note that line 7 to 9 shows how line 2 to 4 will look like when the compiler has made the substitution.

```
1 `define print_a_string $display("Hello World");
2 initial begin
3     `print_a_string
4 end
5
6 //Equivalent to:
7 initial begin
8     $display("Hello World");
9 end
```

Listing 2.8: Example of a macro using *‘define*

The text macros in SystemVerilog also has an argument feature, where the user can substitute parts of the macro with the arguments. It is important to understand that the argument list does not work like the argument list of a function; The arguments passed are not a value or a pointer but text interpreted as code. Consider the example in Listing 2.9; It shows a multi-line text macro with three arguments intended for summing two terms and printing the sum. When calling the macro, we also provide arguments, which in our case are two integers. Line 13 to 17 shows that the macro name is substituted with the code also substitutes the argument names with the integer variables. In essence, macros allow the user to create template-like code, where the user can "fill in the blanks" by passing variables and even pieces of code as arguments [2].

```
int value_1;
int output;
`define add_together(term, sum) \
    sum = term + term; \
    $display("The sum of %d and %d is %d", term, term, sum);

initial begin
    value_1 = 2;
    `add_together(value_1, output);
end

//Equivalent to:
initial begin
    value_1 = 2;
    output = value_1 + value_1; \
    $display("The sum of %d and %d is %d", value_1, value_1, output);
end
```

Listing 2.9: Example of a macro using *'define* and an argument list

2.4 Simulations

Simulations are an essential tool for the verification of a design's behaviour. It is a software that translates design and test benches into predicted behaviour in the shape of a series of scheduled events or processes. The simulation allows one to verify that everything works as intended and to get a sense of the design without having to implement it as hardware [17, 18].

It is important to emphasise that it is not a hardware implementation but software intended to simulate the hardware. In many cases, simulation is the first step of verification during the design process, as it allows the user to make changes and get reasonably quick feedback. In addition, simulation is limited only by the available computer resources and time; Any design can be simulated. The downside of simulation is that it is a slow process [18].

SystemVerilog uses events and processes in its simulations. Processes are typically born from primitives in the SystemVerilog language, such as the *initial*-block, and are supposed to represent a part of the hardware. Note that all processes are concurrent, as in they execute simultaneously. The reason for concurrency is because processes are supposed to represent hardware, which runs concurrently by nature. Events are updates or evaluations of variables and net values of the design. In other words, events are what happens inside processes. As an example, consider the nonblocking assignment in Listing 2.10. The value on *reg2* is assigned to *reg1*. *reg1* will cause an update event if its value changes due to the assignment [19, 2].

```
reg1 <= reg2;
```

Listing 2.10: A nonblocking assignment that can cause an update event.

A vital part of simulations is the management of time. As mentioned, the processes are supposed to run concurrently, which is impossible on a single processor. Instead, we define a time unit called simulation time. It represents the time relative to the progress made by the simulated hardware. As an example, consider Figure 2.8; By the time the simulation time has reached its first nanosecond, 45 ns has already been spent in "reality" by the processor running the simulation. With simulation time defined, we can simulate concurrency by simply not allowing "time" to move before the simulation has handled all events that occur at that moment [19, 2].

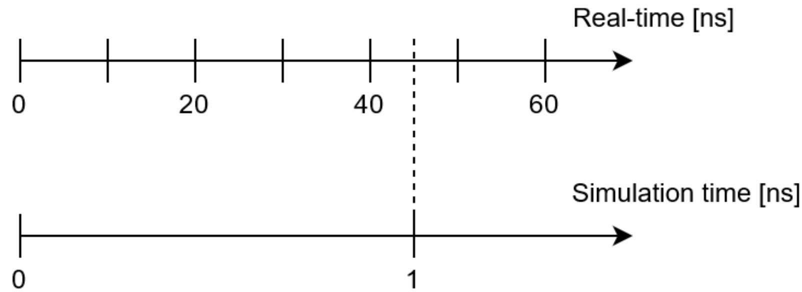

Figure 2.8: Example of simulation time being much slower than real time.

The easiest way of describing an event-based simulation is through a series of regions containing the events to happen. For each time slot, the simulation software iterates through all the events in the regions. By definition, the order events are handled inside a region is random. Once all events have been handled, the simulator moves on to the next time slot. Each region has its purpose, and the simulator handles them in a specific sequence, as shown in Figure 2.9. Which region an event gets scheduled to depends on the context and what statement it is; For instance, updates to the nonblocking assignment in Listing 2.10 would be assigned to the NBA region. Note that the simulator can loop back into previous regions, as events can be rescheduled [19, 2].

This thesis is only interested in two of the regions: the active region and the inactive region. The active region contains events that are to occur immediately, while the inactive region contains events from processes subject to a zero-delay (i.e., *#0*). During simulation, events from the active region will execute until the active region is empty. Then, any events scheduled in the inactive region is moved to the active region, leaving the inactive region empty; This repeats until both regions are empty, moving the simulation forwards and eventually into a new simulation time slot [19, 2].

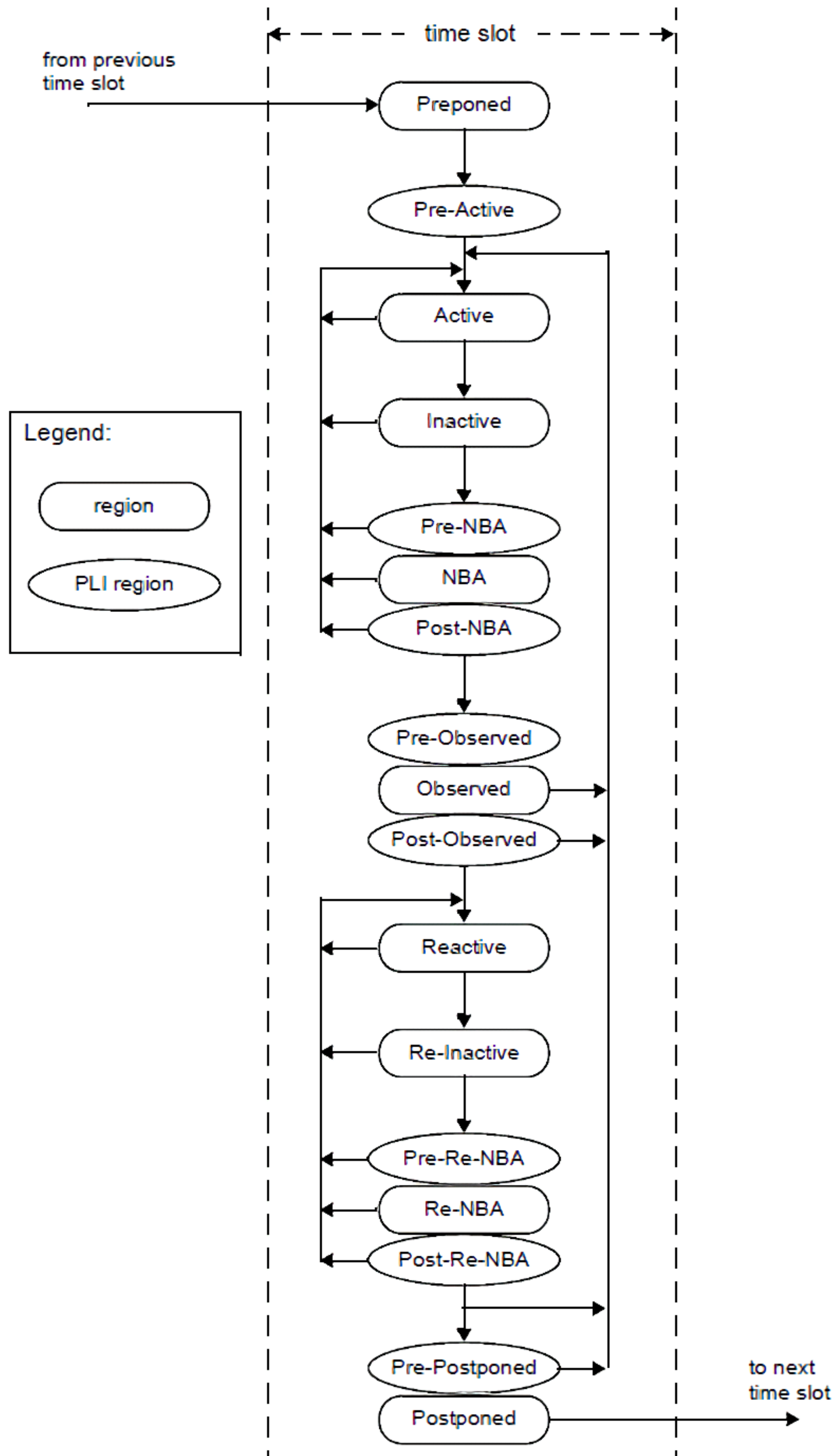

Figure 2.9: All event regions in a single simulation time unit, taken from [2].

As an example, consider Figure 2.10, where the following transpire:

1 Both process A and B are in the active region. However, process B reschedules to the inactive region.

2 Process B is now in the inactive region. Eventually, the remaining processes in the active region, i.e., process A, finishes and the region become empty.

3 All processes in the inactive region, i.e., process B, are moved back to the active region.

4 When process B finishes, both regions will be empty, and the simulation can move on.

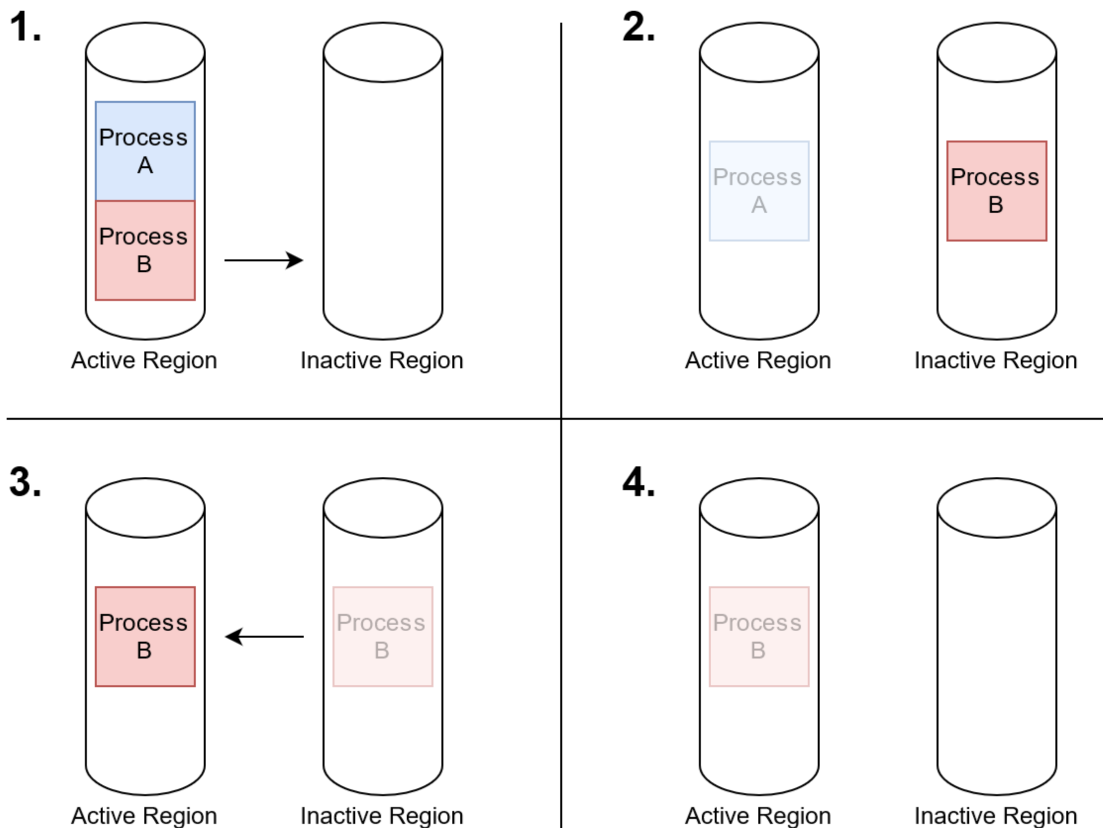

Figure 2.10: Example of how events in the active region are handled.

An important feature is that of rescheduling a process into the inactive queue. As mentioned, imposing a zero-delay using *#0* causes a process to become inactive. We call it a zero delay because while we postpone the execution, it is still happening within the simulation time unit. Doing so allows the user to pause the execution of a process, but not to the extent of progressing simulation time. Each iteration through the events in the active region is called a delta cycle. In other words, when using *#0*, we can say the process is postponed by a delta cycle [19, 2, 18].

We have focused on the simulation theory of SystemVerilog, as this thesis uses SystemVerilog as its HDL. However, other popular languages, such as VHDL and SystemC, also uses the process-oriented approach. Thus, we can find much of the discussed theory in other languages in some shape or form [18].

2.5 Parsing

Parsing is the act of translating regular text into more manageable data. It typically takes the shape of a data structure, made up of bits of information extracted from the text. The parser's job is to determine what parts of a text, or string, is viable information and where they fit in the data structure. The concept of parsing is simple at the surface. Data is feed to a parsing program, typically as a sequence of characters. Following specified rules, the parsing program analyses this data and converts it into a data structure. Figure 2.11 shows an example of a simple parser. It uses a simple rule of looking for keywords in the data, which is "age" and "height", and produces a data structure with the values associated with the keywords.

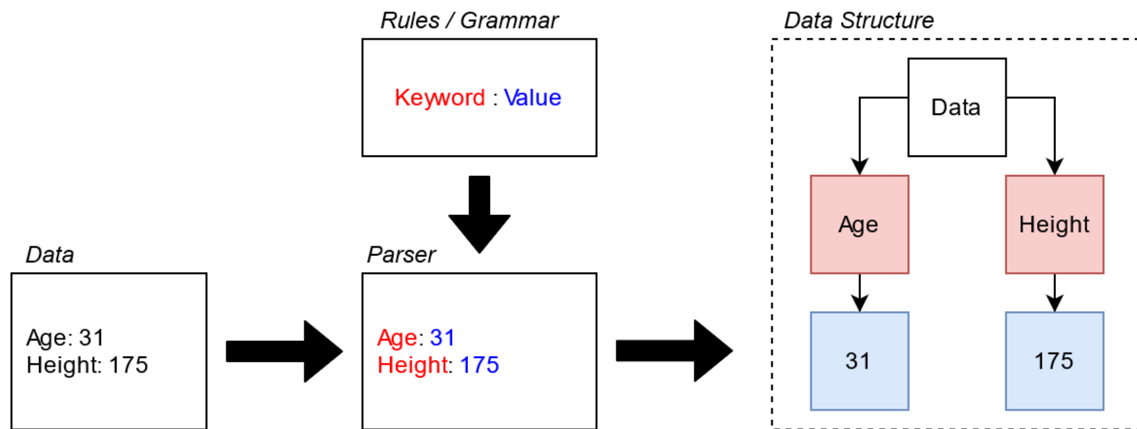

Figure 2.11: Simple example of parsing data into a data structure using a set of rules.

The challenge of parsing lies in determining the rules, or grammar, when both data and its associated structure becomes complex. Consider the case of text written in a human-readable language, such as English. There is a multitude of words and combinations of words that can be produced. They can take on a different meaning depending on context and sentences. Deriving sensible data without a rigid structure to the text and the rules, such as looking for specific keywords and sentence structure, would be a difficult task at best. Similar arguments can be made for parsing text written in computer languages [20].

2.6 Related Work

Safaan et al. [21] proposed two ways of obtaining a connectivity specification. The goal was to provide information to generate connectivity assertions, which could be used for formal verification to validate connections in a system-on-chip. The first approach is to use an IP-XACT library. IP-XACT libraries are documentation for describing IPs, written in XML format. The idea is to compile the library with an XML compiler, then traverse and extract the connectivity specification. However, this approach requires that the design is complete and fully documented or have an older version of the design with the documentation available.

The second approach Safaan et al. proposes is to consider the synthesised netlist of the design. It retrieves the design hierarchy and uses signal tracing between ports to map out the connectivity specification. Since it uses the design files directly, any updates would require new processing. A case study performed with a black-boxed processor showed promising results. There were, however, cases of mismatch between the IP-XACT library and its actual

implementation. Also, the extraction program for the netlist extraction should be tested with more designs.

Rachamalla et al. [22] explores the virtual logic netlist (VLN) concept. A VLN acts as a skeleton version of a regular netlist because it preserves the structural hierarchy of the design but does not contain any of the logic. The purpose of using a VLN is that it avoids the need to synthesising the RTL design before it can be analysed. Using an extraction engine, it extracts a VLN from RTL code in less time than synthesising it. Once the user obtains the VLN, the user can exploit existing back-end tools to analyse the netlist. The solutions were evaluated on several cores of a processor and achieved an error rate of less than 2%. It is worth mentioning that there are limits to what this method can analyse. The user only has the design's structural information; Analysis of metrics such as power consumption would not be feasible.

Große et al. [23] developed a solution for visualising designs written in SystemC code. The tool's motivation is to help the process of understanding and debugging of designs through visualisation, especially in the early stages of development. Their solution's key feature is the ability to extract structural information of a design through a modified SystemC simulation kernel. The tool loads the information into a database that it can use for various operations, such as debug features and custom design rule checkers. The tool also provides a way of including source code references to the SystemC design. The paper also described case studies on a scalable arbiter design and a RISC CPU.

Hosny and Baher [24] also developed a prototype for visualising design similar to Große et al., but for code written in SystemVerilog. The motivation is to visualise the design for easier debugging processes and to ease code maintainability. The solution divides its concept into three stages: extracting data from SystemVerilog code, visualising the data, and query after specific information. A design crawler handles the extraction, written in C and coupled with a SystemVerilog API. It crawls through the design and extracts the desired data. The web application handles the visualisation, showing the design as a mind map. The user can also manipulate the visualisation. The query features allow the user to search for specific information, such as finding continuous assignments from the SystemVerilog code. The prototype, at the moment, can handle basic SystemVerilog constructs, with plans in the future for adding assertions, coverage, and support for VHDL.

These papers all have the common aim of extracting an overview of the hierarchical relations between components from a design. The difference is how they achieve this, and to some degree, its purpose. While this thesis goal is, to some extent, different, the idea is the same: figure out how different components relate to one another. Another way of describing it is that the thesis is not looking for hierarchical relations but information related to bus systems. A point to be made is that while they explore different approaches, few provide thorough studies and experiments to validate the solution. Therefore, one can consider this problem to be at an experimental stage, where a good solution cannot be entirely determined yet. The goal of this thesis is to provide one of these solutions.

Chapter 3

Discussion of Potential Solutions

Section 1.1 presented the problem that this thesis is trying to solve: extracting information about bus systems from intricate designs. It described various issues that relate to the problem that one should consider when developing a solution. In summary, it is what information one would want to obtain, how to do so, and what to do with the obtained information. In some sense, these three issues make up the core of the problem this thesis is trying to solve. This chapter aims to discuss these issues and potential solutions that address these issues, along with the solution chosen for this thesis.

3.1 What Information Should Be Extracted

The first issue the thesis addresses is what information should a potential solution extract. A thorough definition of the various types of information that would be of interest will help define a potential solution. The goal of a solution is, after all, to deliver a good overview of the bus system and its various properties; One would want to extract the information which delivers that. This thesis has determined what information is of value by studying bus system theory and conversations with designers at Nordic Semiconductors. Of course, what is of interest might differ between companies and their designers, but this section should cover a lot of it.

Ideally, a potential solution should be able to obtain all information presented in this section, or at least have the possibility to do so later. However, as this thesis has limited time and resources, it can not cover everything presented in this section. That which is left out is considered for future works and improvements. The information that is of interest is as follows:

Connection between modules

The first type of information to be considered is connections between modules and busses. One of the defining features of a bus structure is that many modules are connected to the bus in order to communicate with one another. Therefore, having a good overview of these connections are essential in managing complex bus structures. Figure 3.1 shows an example that lists the connections from a simple bus structure.

One can further expand on the idea and include another component often used with busses: bridges. As mentioned in Section 2.1, bridges connect two separate busses, allowing modules on one bus to reach other modules on other busses. Understanding which bus connects to which bus will help determine the reach a module has. Figure 3.2 shows a simple example of this, where a bridge is connecting two busses.

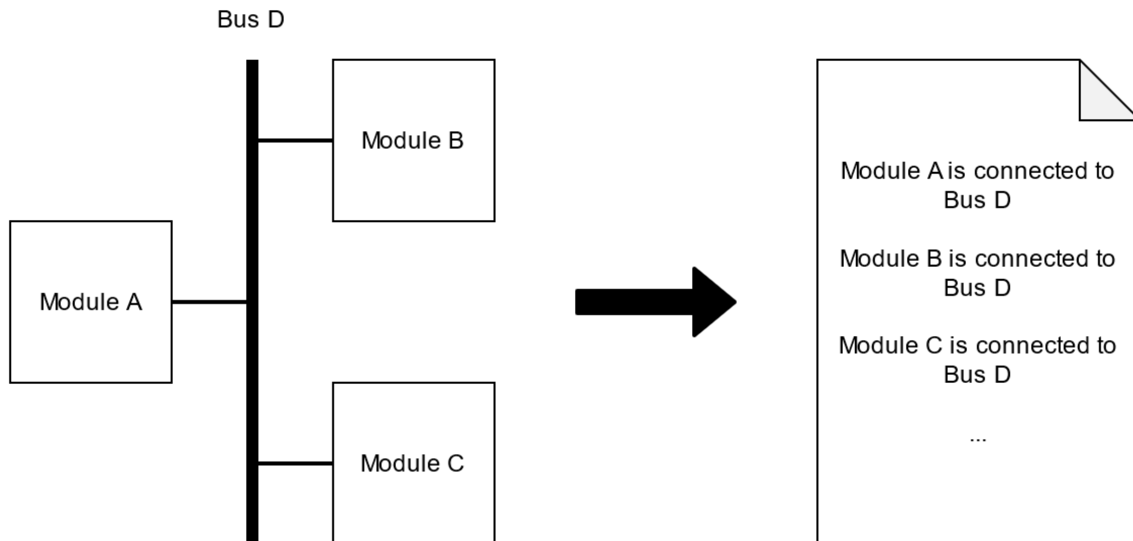

Figure 3.1: Example of connection information from a simple bus structure.

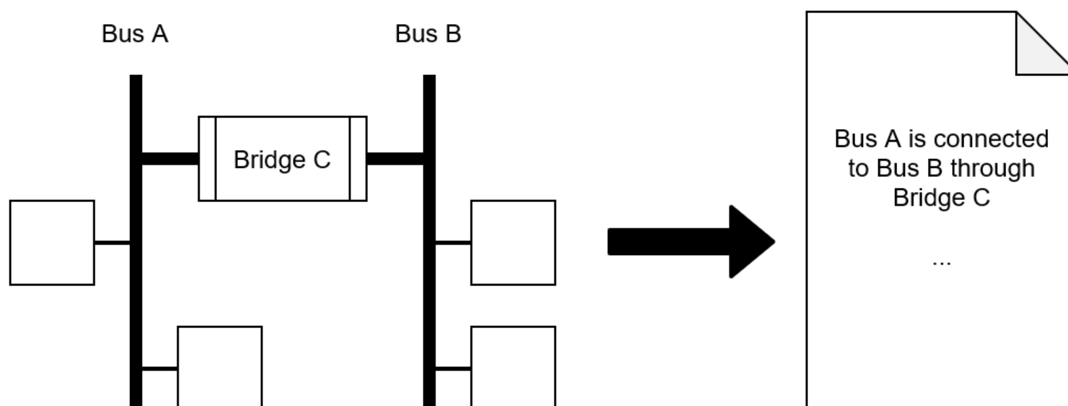

Figure 3.2: Example of connection information on a bridge between two simple bus structures.

Module Properties

The next type of information of interest is the modules themselves. A module is a collective term and can be everything from CPU's and RAM's to tailored hardware. The types and amounts of information that a solution can extract vary widely as well. This thesis deems it beneficial to mainly include that which impacts the bus system, such as whether the module is a master or a slave. Other information unrelated to the bus system is ignored to better focus on the topic at hand.

There are two pieces of information related to modules that would be useful in the context of bus systems. The first one is the name of the module. It allows one to identify the module and, in most cases, help to describe its purpose. For instance, a module called "RAM_1" is almost certainly a memory module consisting of random-access memory. Figure 3.1 used names to identify modules, such as "Module A" and "Module C".

The second module-related information of interest is whether it is a master, a slave, or both. As mentioned in Section 2.1, one can label modules as masters and slaves, based on whether they can request a data transfer or receive such a request. For example, Figure 3.3 shows a case where both name and type is retrieved from a simple bus structure, consisting of a CPU and a RAM module.

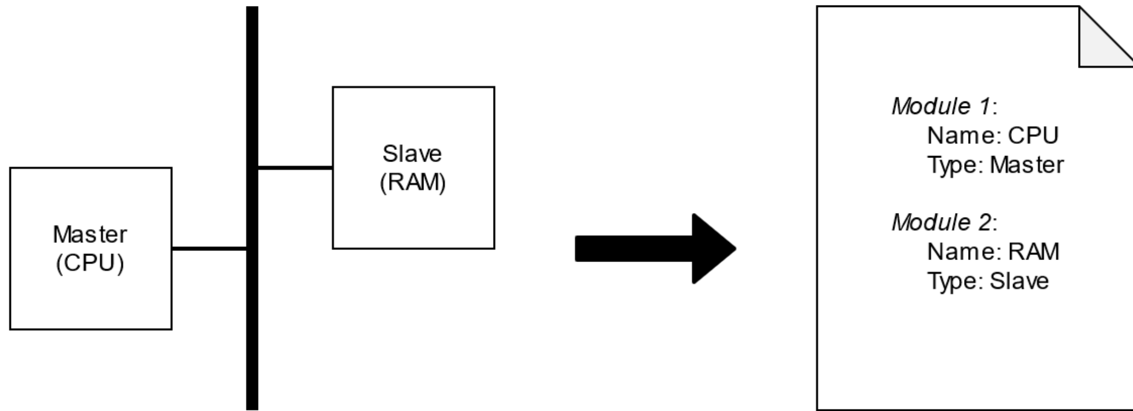

Figure 3.3: Example of module-related information from a simple bus structure.

Bus Properties

It will be beneficial to include information surrounding the busses themselves; This is information that describes the bus wires, such as the bus names and what type of protocol they follow. A bus can also be referenced by different names, depending on the module. Knowing which names a bus is referred to can be beneficial in identifying them, especially in a system with multiple busses. One can also consider properties that can vary between instances of a bus protocol, such as data- and address width. Figure 3.4 shows an example of a bus, where the modules have different names for the bus.

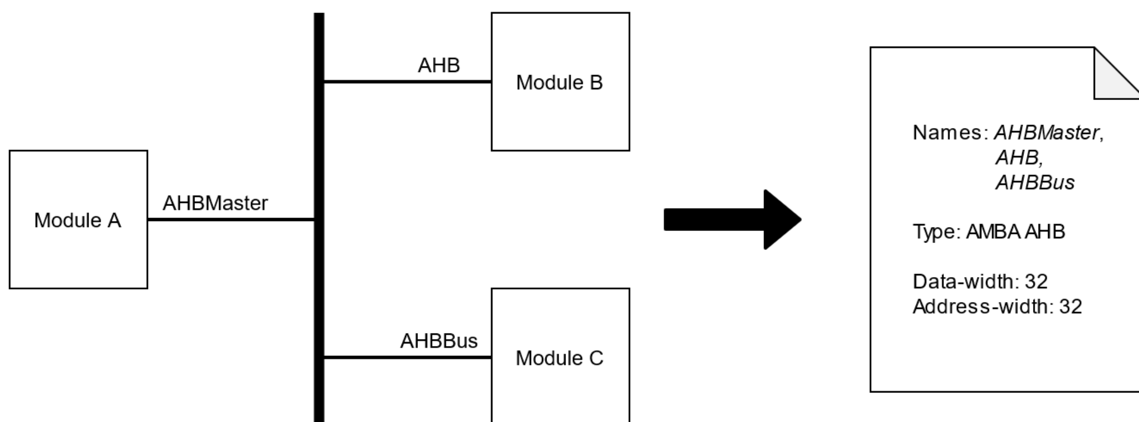

Figure 3.4: Example of bus related information from a simple bus structure.

Including this type of information might be considered excessive considering that the information should be readily available to the designer. However, it has its merits in the broader picture, supporting other information types in creating a complete overview; This benefit is even more apparent in a more complex system that utilises several busses.

Matrices and Interconnection

As one explores more complicated bus protocols, one can run into the matrix topology as described in Chapter 2.1. In this topology, which bus that transfer data between modules can change depending on the bus system's current situation. When a master needs a connection to a slave, the bus systems allocate an available bus. In that sense, it does not have a permanent connection but is "connected" nonetheless. A good example is the AHB multi-layer bus matrix, where several busses run in parallel.

With matrices, one can end up in a situation where it appears, from a top-down view, that everything connects to everything. While it might be the case for some systems, others will place restrictions on which slaves a master can access. It is similar to which module connects to which bus, as described earlier in this section. Extracting information on which slave a master can reach would be helpful. Figure 3.5 shows an example, where a bus that uses a matrix maps its "connections" between each master and slave. Similar arguments can be made for interconnecting modules, which typically defines which master connects to which slave in a bus system. Extracting this information would be beneficial for the user.

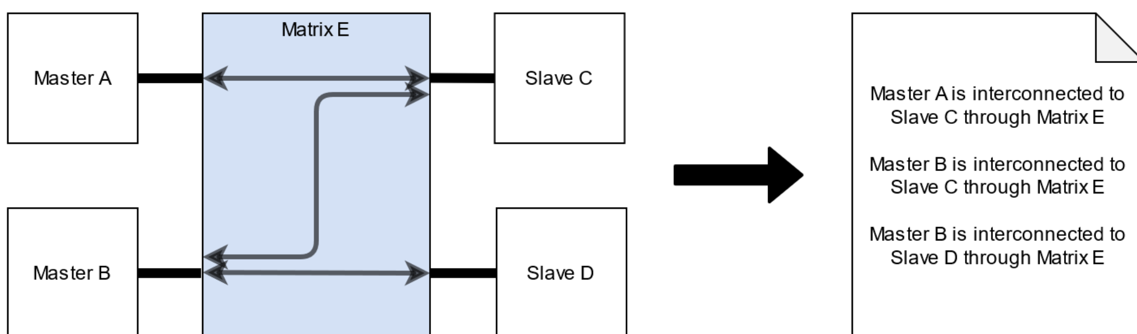

Figure 3.5: Example of interconnection information from a simple bus structure that consists of a matrix.

Address Map

Like interconnections and matrices can have restrictions on which slave a master has access to, there may be restrictions on which address space a master can access. An example of this could be a specific area of a memory module accessible only by the CPU. When such restrictions are present in a bus system, it can be helpful to map them out. Figure 3.6 shows an example of a memory module that splits its available memory into two partitions. Module B has access to both partitions, but Module A has only access to the latter one.

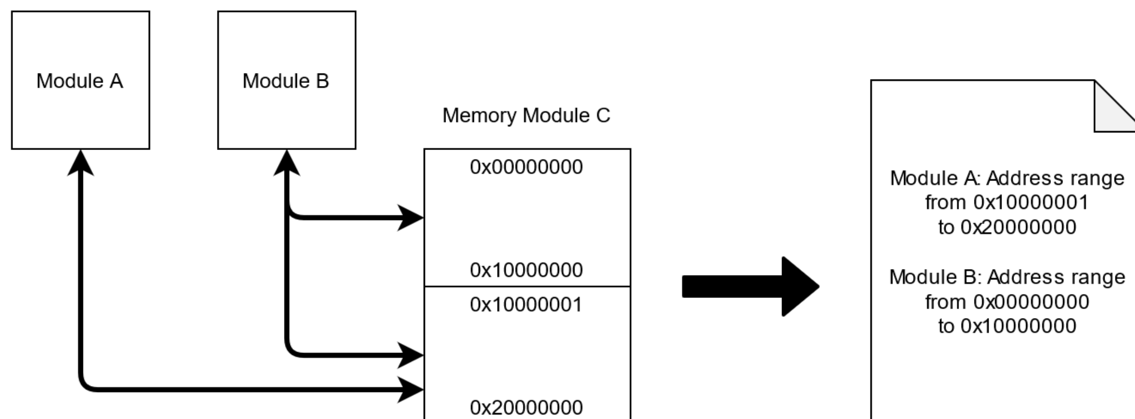

Figure 3.6: Example of address map information, where only Module B can access a specific part of Memory Module C.

3.2 Solutions

The previous section dealt with what one expects of a solution in terms of the results it would produce; In other words, what information should it extract. Naturally, a good solution should produce this information or at least have the possibility to do so in the future. It is an important aspect, but hardly the only one that needs consideration. There is more to a solution than what it can output. A good example is that the solution should be easy and practical to use. It might go without saying that the better solution requires minimal effort to use and understand. However, it also has another benefit to our specific situation. Complex RTL design is seldom conceived by a single person these days. As briefly mentioned in Chapter 1, these designs are a team effort. Consider a situation where the potential solution requires additional code, or changes, to the RTL. One probably has to involve multiple people to achieve this. A confusing solution could lead to a poorly implemented solution, increasing the risk of human error.

The rest of this chapter is dedicated to present the various solutions the thesis have considered for implementation. Most take inspirations from the related works discussed in Chapter 2.6, angled towards the case of extracting bus-related information instead of a hierarchical relationship between modules. Of course, they all have their flaws and advantages, but it is the solution presented in Section 3.2.4, describing extraction through simulation, that is implemented in this thesis.

3.2.1 Parsing of HDL Code

The first and perhaps the most obvious solutions to this thesis's problem is that of using parsing. The idea is to use a parser program to scan through the design source code. The parser would locate and extract information related to the bus, and organise it into a data structure.

Parsing programs are not dependent on a programming language. In other words, a parsing program that parses hardware code can itself be written in a high-level language. The solution can then benefit from all the features a high-level language brings to the table. In our case, input data would be the HDL source code where the parser program would scan through it methodologically. Another benefit is that it is a fairly automatic process; The user decides which files are to parse, and the tool does the job. There was a similar approach by Hosny and Baher [24], as shown in Chapter 2.6. They used a parser program, written in C, to systematically scan through SystemVerilog code and extract the hierarchical design structure. From the prototype they developed, it seemed to be a decent proposition for design extraction.

The biggest drawback to this potential solution is the complexity of the input data. Using HDL source code means the grammar used for parsing would also be quite complex. In the attempt by Hosny and Baher, they tried to use a SystemVerilog API and work with the simulator to extract the desired information. While the solution worked, it lacked the support of more advanced language features at its current state. The issue is further complicated because there are likely different coding conventions between different teams and their implementations. There is likely even a different coding style and preference between people within the same team. Taking this into consideration, developing a solution that works for many different designs will be difficult. It is this high complexity of the parsing grammar prevents this solution from being an optimal one.

3.2.2 Netlist Signal Tracing

Safaan et al. [21] explored the concept of using the netlist as a source for extraction of information (besides exploring IP-XACT documentation) in Chapter 2.6. The concept is as follows: a program extracts modules and their ports from a netlist. This netlist is synthesised from the RTL that is of interest. Afterwards, it uses signal tracing to map out which module is connected to which module.

As with parsing, this solution benefits from being fairly automatic. The user should only need to provide the relevant sources of extraction, which, in our case, is the netlist of our design. The downside of using the netlist is that it requires the user to synthesise the design beforehand. For larger designs, this can be a time-consuming task. Considering that every change that affects the bus system would also require one to re-synthesise the design anew is off-putting.

Another point to be made is that the solution presented by Safaan et al. focused primarily on connections between modules in general. While it did categorise some signals as busses, its definition of a bus was any collection of wires with the same source- and destination module. This thesis has a much more precise definition of busses and what information it wishes to extract; The solution proposed by Safaan et al. will, as such, be insufficient, without some additional functionality aimed at extracting the information types presented in Section 3.1.

3.2.3 Modified Simulation Tools

Große et al. [23] brought up an interesting solution in Chapter 2.6; The idea is that a simulation tool will, at some point, acquire and figure out a lot of the information that they where after. The extraction does not happen from the RTL but by the simulation tools "simulating" the RTL. The way Große et al. extracts the information is by modifying an existing simulation kernel to provide the information at initialisation.

A significant advantage of this solution is that an existing tool can handle a big part of the work. It removes some of the complexity that has troubled previous potential solutions as well. As the extraction happens upon initialisation of the simulation tool, time spent extracting can be reduced by finishing the simulation after initialisation.

There are, however, a downside to modifying a simulation tool: one needs to have a good knowledge of how the tool operates and obtains the information we want. To further complicate things, there might be difficulties in adapting Große et al. solution to fit the thesis' more specific bus-related information. For instance, Große et al. had a problem obtaining the names on ports and resorted to parsing the source files for names. Similar problems would occur in our case as well, where the names of busses would be beneficial to obtain.

3.2.4 Extraction Through Simulation

The final solution, and the one implemented in this thesis, is extraction through hardware simulation. It is inspired by the solution presented by Große et al. [23], but with a significant difference: one does not alter the simulation tool itself. Instead, the solution introduces small code snippets in strategic places in the code describing the RTL. During simulation, these code snippets perform necessary operations and extract the relevant information. In some sense, one could say we are enhancing the RTL with extraction code. Figure 3.7 visualises this solution, where design files with extraction code are simulated and produce information. Note that the solution ignore other simulation results, as they do not contribute to the solution.

A common issue amongst the alternative solutions in the previous solutions was their complexity. For instance, the parsing solution in Section 3.2.1 required a thorough understanding of the HDL syntax to make the correct interpretations. The solution by extraction through simulation reduces this complexity and removes a significant amount of the work as the simulation tools make the necessary interpretations. Furthermore, by defining the extraction behaviour as part of the simulation, one keeps the benefits of allocating work to the simulation tools while avoiding the complexity of modifying the tools themselves. Another benefit is that the solution is not bound to one specific simulation tool, making the solution more universal.

An important characteristic of this solution is that it will only cover the part of the design it is implemented in. In other words, it is up to the user to implement the solution where it needs to be implemented. Such an approach is not without its flaws. Should the user accidentally implement the solution only partially, it might, in the worst case, go unnoticed. The resulting output would then be incomplete. On the other hand, one could argue that this approach gives the user more control, allowing them to choose which part of a design is covered by the solution. Ultimately, it is an issue that should be kept in mind.

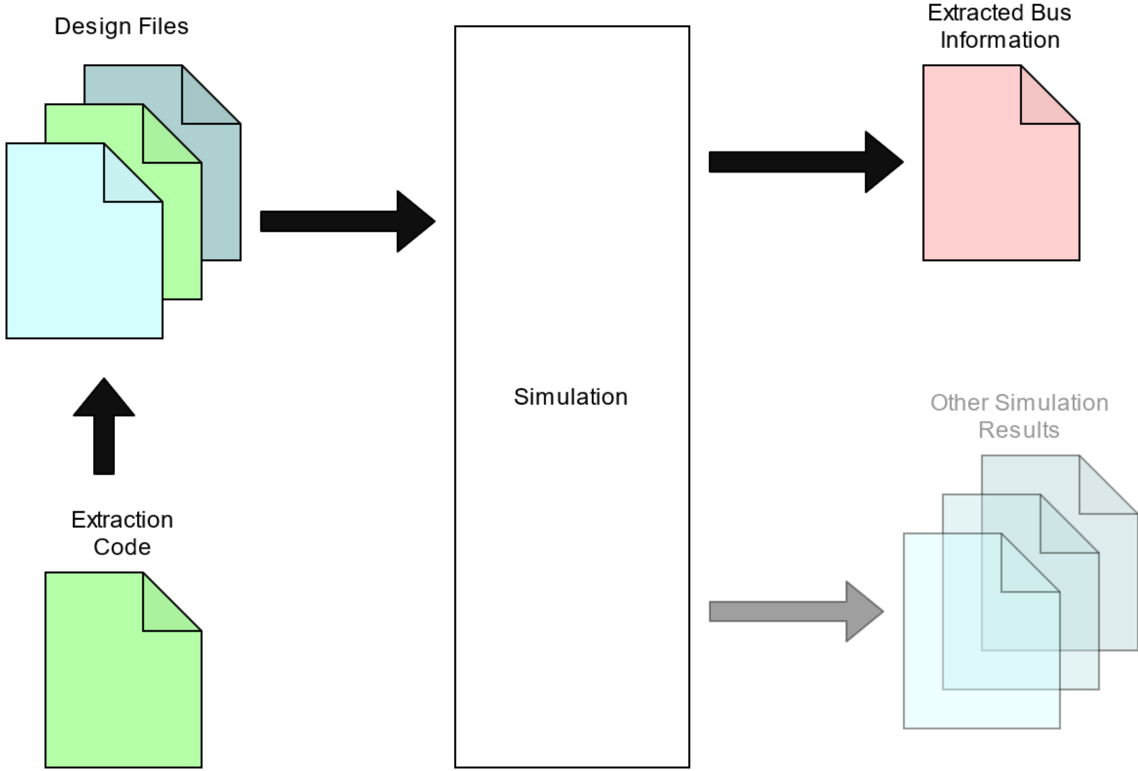

Figure 3.7: Extraction code is placed in design files, and produces bus information through simulations.

The code snippets introduced into HDL uses the same language as the HDL itself. While HDL is suitable for extraction code, more high-level functionalities, such as visualisation, will be challenging. For that reason, this thesis chooses to split the solution in half: one part for extraction and one part for processing and visualisation. The processing and visualisation are better implemented as a separate tool, written in a high-level language. One needs to take into account that the processing tool must import the extracted information as well. Figure 3.8 shows how the extraction part uses the RTL to produce bus information; The processing tool imports the extracted information, processes it, and produces diagrams and analyses.

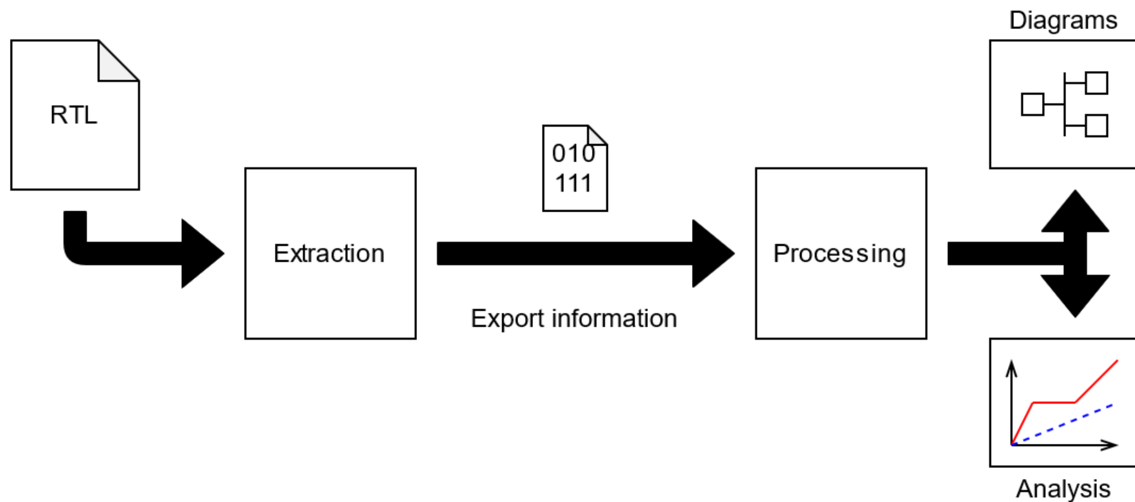

Figure 3.8: The flow of the split solution, where one part handles extraction and the other handles the processing.

To better cover the reasoning and implementation of the solution, the details of each part of the solution are distributed between two chapters. Chapter 4 will describe the implementation of the extraction solution, showing how one can apply the solution to an RTL design to extract bus-related information. Chapter 5 will describe the implementation of the processing tool, importing the information obtained from the extraction solution, and performing various operations on it.

Chapter 4

Extracting Bus Information

The concept of the extraction solution is not that different from a testbench. The basic functionality of a testbench is to generate stimulus, apply it to the device under test, then capture the response and check for correctness [16]. The thesis' solution aims to stimulate the RTL so that the response contains the information that the user wants to extract. The user can then use this information to get better insight into the bus system by, for instance, importing it into the processing tool described in Chapter 5. The issue of extracting information then becomes a question of what stimulus must be applied to generate a response containing this information.

One can categorise the stimuli for the solution into two types: one where one module applies a stimulus and other modules replies, and one where the module returns internal properties and values. Which type of stimulus to use for which type of bus-related information is a matter of using the right tool for the job.

The first type of stimulus can also be considered a series of write- and read operations. Since it has the advantage of readily implemented bus systems in the design, one can exploit this structure between the modules. Typically, one module will write a signal to the bus; Any module connected to the bus would perceive this signal and respond accordingly. Consider the example in Figure 4.1. A signal is stimulating the bus connected to module A and B. When these two modules register the signal on their ports, they respond accordingly. Module C is not connected to the bus and does not respond. Using operations like this can help extract bus-related information, such as which module connects to a specific bus.

The second type of stimulus does not rely on other modules but instead returns values and properties that can be found inside the module. The use of SystemVerilog's macros, explained in Chapter 2.3, simplifies this. It plays on the template-like nature of macros, allowing the user to use port- and variable-names directly into the extraction code without knowing its inner workings. Functionality like this is helpful when developing a solution that different people can use for different systems without significant modifications to the solution.

The following section will provide an overview of the solution before moving on to a more in-depth explanation of how the solution can be implemented in Section 4.2. Afterwards, the rest of the chapter is dedicated to explaining the extraction code and how the various SystemVerilog macros works.

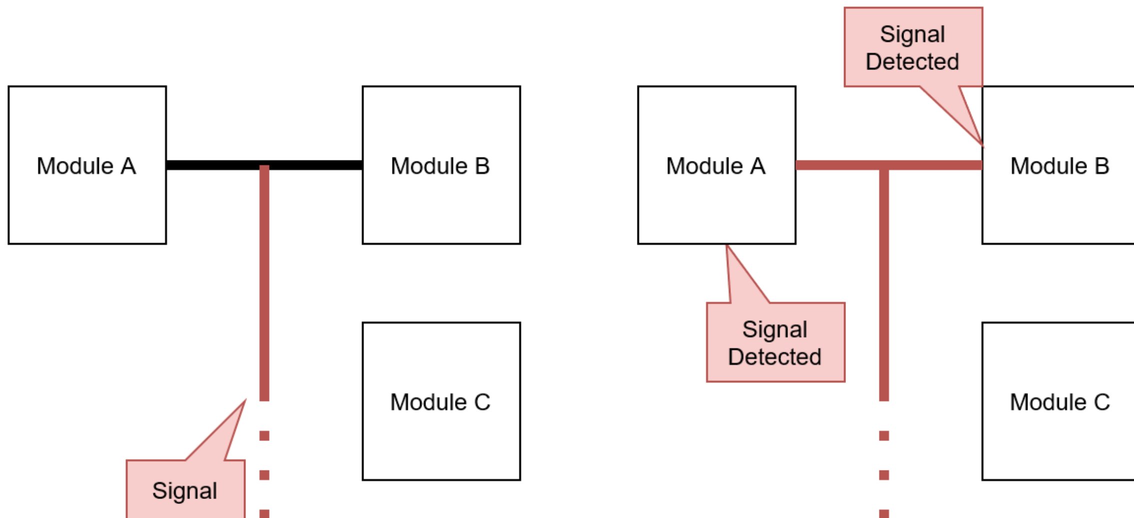

Figure 4.1: Example of a stimulus signal on a bus causing two of the modules to respond.

4.1 Brief Overview of the Extraction Solution

To help better understand this thesis' solution for extracting bus information from RTL design, we will look at a small example. It is a simple bus with three modules: a processor acting as a master module and a pair of RAM modules acting as slaves. The bus and its modules are depicted in Figure 4.2.

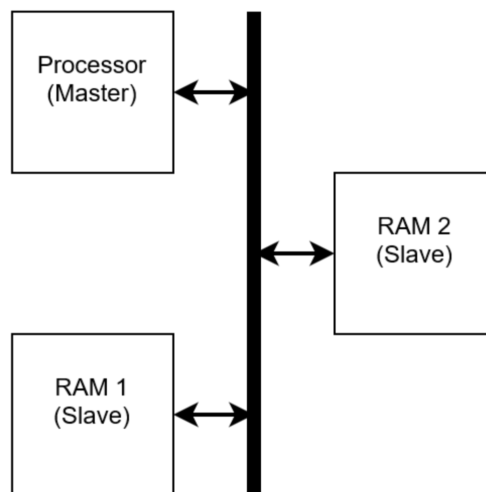

Figure 4.2: A bus with a processor and a couple of RAM modules.

The system in Figure 4.2 is more of an abstract representation of how the modules connect to one another. When looking at it from a source code perspective, it would look more like the diagram in Figure 4.3. The modules and bus are organised in files, written in HDL, and often further divided into sub-modules internally. The RAM modules use the same source code to create two separate instances of the same module. All three modules are connected to the bus system, containing everything needed for a bus to operate.

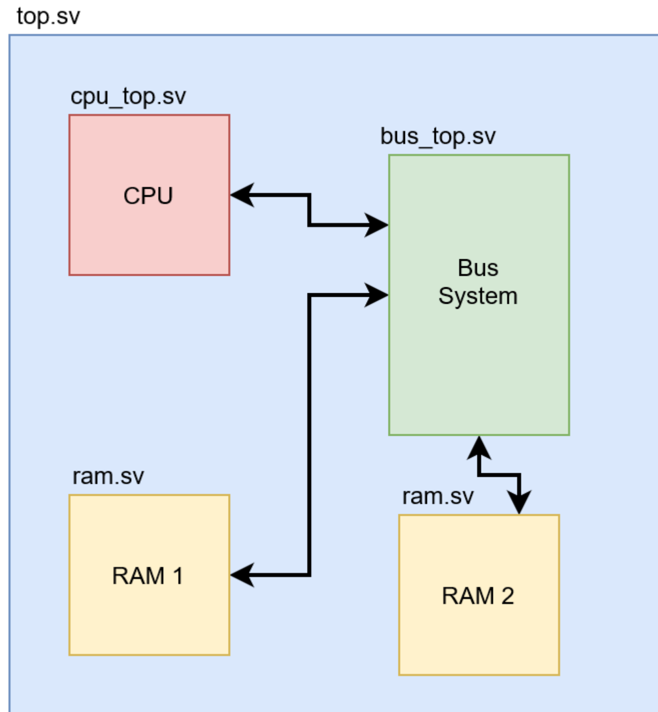

Figure 4.3: A bus with a processor and a couple of RAM modules, organised in files.

The extraction code presented in this thesis can extract the following information:

- Which module connects to a particular bus.
- Whether a module is a master, a slave, or both.
- Which slave modules can a master reach through the bus system.
- The names of the modules and wires.

As described earlier, the solution is implemented by including snippets of extraction code into the modules. Which modules are masters or slaves are defined in the bus system module (i.e., *bus_top.sv* in Figure 4.3), along with which module a master can reach through the bus system. It makes sense to place extraction code into the bus system to extract that information. Further, which module is connected to the bus is defined by both the bus system and the modules connected; This means that there should be extraction code in both the bus module and the modules connected to the bus to extract that information. Figure 4.4 shows the example from Figure 4.3, but now with extraction code snippets added to the modules.

Once the extraction code is in place, one can run the simulation to extract the information. Each code snippet acts on its own, collecting the information and export it to the simulation log; This is depicted in Figure 4.5. The example shown here is quite simple, but more complicated bus systems follow the same implementation idea. The only real difference is that there are more modules to consider, which means more modules that requiring snippets of extraction code. However, the code is autonomous and does not need further intervention once implemented, even when expanding the implementation with additional busses and modules.

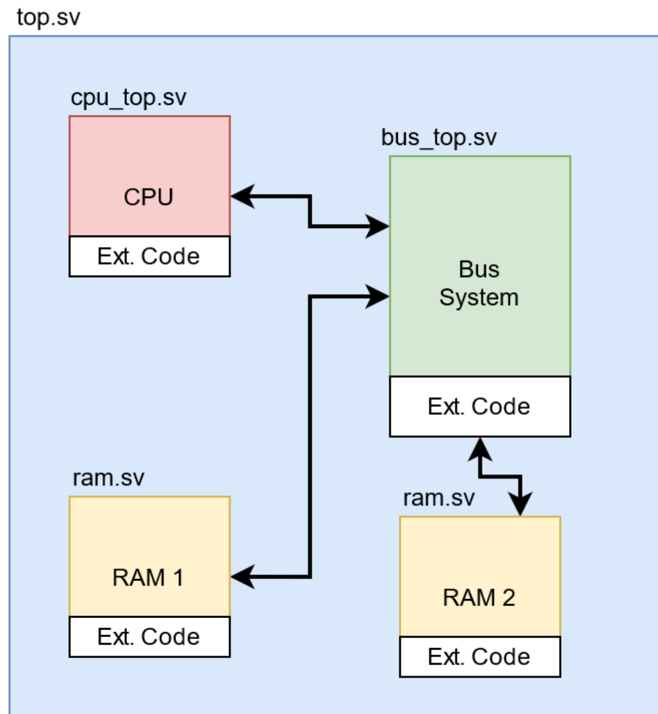

Figure 4.4: The bus system from Figure 4.3, with extraction code implemented into the modules.

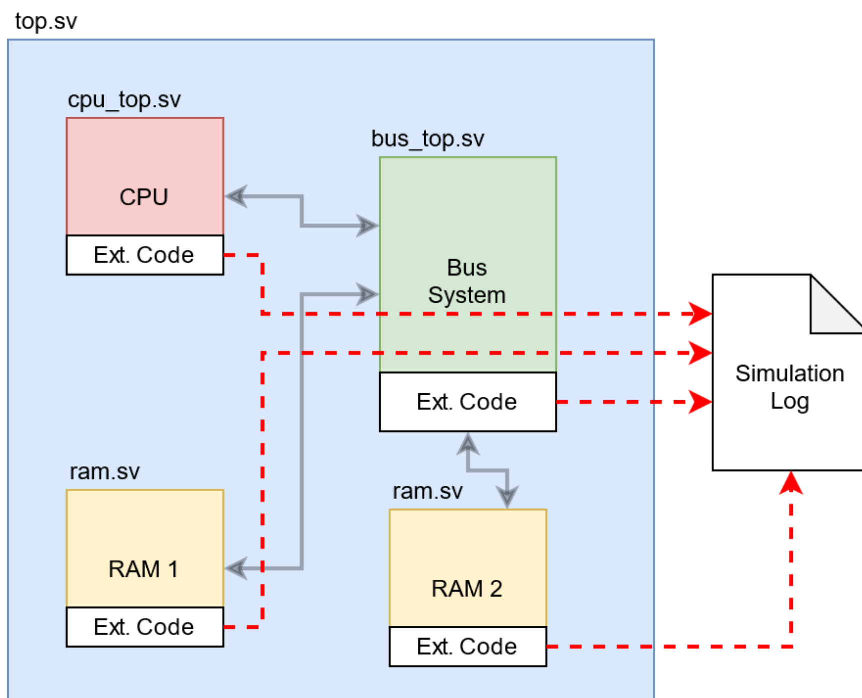

Figure 4.5: The bus system from Figure 4.4 during simulation.

4.2 Implementing the Extraction Code in Original Modules

A single SystemVerilog file contains all extraction code necessary for extracting and supporting the extracting of bus-related information. The thesis includes the file as Appendix A. Keeping the extraction code in a single file makes it easier to include it in projects, where the alternative would be to spread it over several files. However, if the extraction code is expanded greatly in the future, it might be necessary to expand it over multiple files. To use the extraction code, include the file with the other source code of the RTL. The files location is of no importance as long as the other files can reach it, but the main project folder of the RTL is a good place to start.

The extraction code utilises mostly macros, using arguments to alter the code. For instance, a macro designed to read a bus value will accept the instance name of the bus as an argument, injecting it into the code where needed. Whenever the users want a specific operation, they can call on these macros. An interesting feature of our extraction code macros is that they are designed to run concurrently. In addition, the extraction code should run at the beginning of the simulation. These two properties are in order to execute the extraction as early and as fast as possible, reducing the risk of interference from other simulation tasks. Section 4.3.1 goes into further details. When adding extraction code to a module, both properties can be achieved easily by encapsulating the extraction code in an *initial*- and *fork-join* block; Listing 4.1 shows this structure. Simply put, all extraction macros presented in this chapter should be placed within an *initial* and *fork-join* blocks.

```
1  module m (
2      // Input and output of the module
3      );
4
5      initial begin
6          fork
7              // Place extraction code here
8          join
9      end
10
11     // The rest of the module ...
12
13 endmodule
```

Listing 4.1: Template for where to place extraction code in an arbitrary module.

When writing a signal to wires, the solution uses SystemVerilog assignment statement *force*. While it overrides any other values or assignments on the wire, it can not propagate out of a module unless defined as an output. Since the solution uses values from *force* statements for determining connections between modules, the wires written to must be defined as outputs. Luckily, most bus systems have data busses that go both from master to slave or slave to master; This can either be a single bi-directional data bus or a pair of data busses. The thesis will use data busses to carry the forced signals.

Alternatively, one could use other wires present in a module. As long as the module defines them as outputs, they would deliver similar results. The appeal of using data busses is that they are almost guaranteed to be present in the module. In addition, data busses are wide enough to hold a wide variety of values, which Section 4.3.2 explains why it would be helpful. One could also implement a brand new set of wires, specifically for the extraction code. Unfortunately, a separate connection could misrepresent the bus system. Consider a case where a bus is disconnected, either through a bug or intentional, but the separate wires are not; The separate connection would still represent the bus as connected, essentially providing wrongful information.

While not shown in this section, most of the macros are disabled by default. For the extraction code to be enabled, one has to include the definition "*ENABLE_BUS_EXTRACTION*"; This can be done either directly in the SystemVerilog files or as a simulation property. Disabling the extraction code prevents it from interfering with the design outside of the extraction process. The extraction code also has a macro called "*FINISH_SIMULATION_EARLY*" that ends the simulation after the extraction process to reduce the simulation time. Note that the macros that extract information automatically export it by writing to the standard output, which in most cases are the console or log. The reasoning behind this and how the information is structured is detailed in Section 4.5.

Interconnecting Modules

The first step is to implement the extraction code in the interconnecting modules in the bus system; These are the modules that typically contain arbiters and decoders. Essentially, masters and slaves connect to these modules when they use the bus system, as shown in Figure 4.6. One can derive much information from these modules, and thus it forms the basis of the extraction implementation.

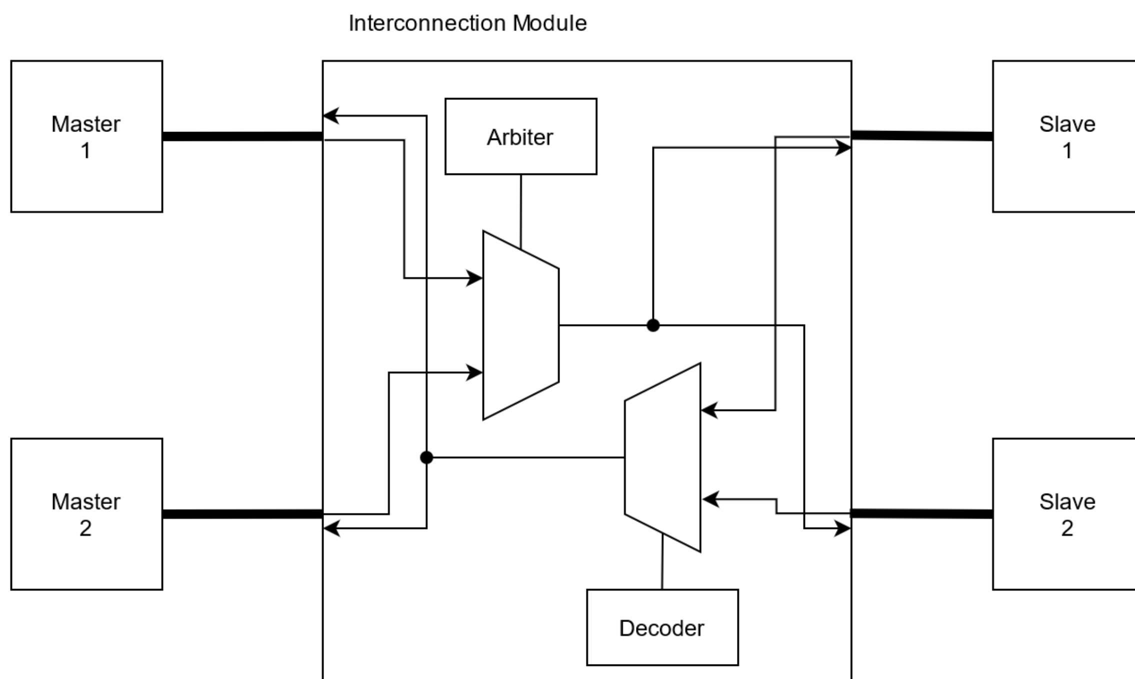

Figure 4.6: Simple diagram of a interconnecting module, with two masters and two slaves connected to it.

There are three operations one would want to execute from this module. The first is to stimulate a signal on the bus wires going out to all the masters and slaves. Doing so allows one to find out which modules connect to the bus system by evaluating which module responds to the signals. The evaluation itself is not done in the extraction code but rather in the processing tool covered by Chapter 5. This operation can be recognised as the "write-and-respond" operation described at the introduction of this chapter, where the interconnecting module writes, and masters and slaves respond.

An alternative is having the masters write to the bus and slaves respond to the signal. The result would be pretty similar to the one achieved by the thesis solution. However, having the masters write to the busses would require one to spread these operations between the master modules. It would increase the designer's risk of forgetting to implement the operation into a module, whereas collecting all the operations in one module would help negate that risk. In addition, the interconnecting module is most likely an IP reused in other RTLs. Placing most of the extraction code in the interconnecting module would mean less work and complexity when implementing the extraction code in later RTLs that reuse the interconnecting module.

The second operation to implement in the interconnecting module is to mark which port goes to masters and which goes to slaves. The interconnecting module needs to keep track of which port goes to what type of module to treat them correctly. That information can be extracted from the module and used to identify what type of module is at the other end of a port. It is important because writing and reading the bus does not determine who is a master and who is a slave by itself.

Finally, one should implement the operation that extracts the interconnecting matrix of the module; In other words, the information that details which slave a master can reach through the interconnecting module. As we can see in Figure 4.6, even the simplest of bus systems does not have a direct connection to the slaves since there is logic in-between to mediate access between modules. In some bus systems, the interconnections are dictated by a matrix, such as the one shown in Listing 4.2. Doing so allows one to make changes to the interconnections with ease. The example in Listing 4.2 shows a matrix where the first master would only connect to the first slave, but the second master can reach both.

```
1 int unsigned NUMBER_OF_MASTERS = 2;
2 int unsigned NUMBER_OF_SLAVES = 2;
3
4 bit CONNECTIONS [NUMBER_OF_SLAVES-1:0][NUMBER_OF_MASTERS-1:0] =
5     '{ '{1'b1, 1'b0}, '{1'b1, 1'b1} };
```

Listing 4.2: Example of a interconnect-matrix, defining which port is connected to each other.

While some bus systems will have stated their interconnection in a clear and concise matrix, others might have their interconnections hard-coded into the design. A typical example would be tool-generated bus systems, where the user provides parameters and a tool generates the bus system from them. As such, there will not be an available matrix like the one shown in Listing 4.2. A solution could be to infer the interconnections from the logic; This would be a much more complicated solution, especially considering that there will not be a practical solution translatable between different bus systems. Unfortunately, the thesis' solution does not have functionality for handling this specific problem and are left for future works.

In Listing 4.3 there is an example of how the extraction code could look implemented in an interconnecting module. All three operations are declared as macros performing the specified behaviour and are explained in detail in Section 4.4. The macro "*FORCE_AND_RELEASE_BUS_ARRAY*" writes a signal on each element of an array, which in this case is two collections of data busses going out of the module. The macro "*MARK_BUSSES_AS_MASTER*" marks the data busses as busses going to a master. The equivalent macro "*MARK_BUSSES_AS_SLAVE*" does the same for slaves. Finally, the macro "*INTERCONNECT_BUSSES*" maps the ports that can reach each other using the *CONNECTION* matrix. Note that since these macros are executing concurrently, the order does not matter as long as they are inside the fork-join block.

```
1  initial begin
2      fork
3          `FORCE_AND_RELEASE_BUS_ARRAY(DataReadMasters, NUMBER_OF_MASTERS,
           ↪  DATA_WIDTH, "DataReadMasters", )
4          `FORCE_AND_RELEASE_BUS_ARRAY(DataWriteSlaves, NUMBER_OF_SLAVES,
           ↪  DATA_WIDTH, "DataWriteSlaves", )
5          `MARK_BUSSES_AS_MASTER(DataReadMasters, )
6          `MARK_BUSSES_AS_SLAVE(DataWriteSlaves, )
7          `INTERCONNECT_BUSSES(CONNECTIONS, DataWriteSlaves, DataReadMasters,
           ↪  )
8      join
9  end
```

Listing 4.3: Example of extraction code placed in an interconnecting module.

Master/Slave Modules

The implementation has the interconnecting module broadcasting signals in every direction there is a bus. It is only natural that there are masters or slave modules to respond at the other end. Since we put most of the extraction work in the interconnecting module, the necessary code for master and slave modules are quite small in comparison. Essentially the module only has to read the busses coming into the module, independent of whether it is a master or slave. Listing 4.4 shows an example of a master module reading its data bus "*DataRead*". The information that is obtained by reading the bus is exported to the simulation log, as detailed in Section 4.3.3.

```
1  initial begin
2      fork
3          `READ_BUS(DataRead, "DataRead", )
4      join
5  end
```

Listing 4.4: Example of extraction code placed in a master module.

Bridges and Relaying Modules

Sometimes, there will not be a direct connection between the master or slave and the interconnecting module. A typical case would be when modules are in different domains, and a bridge is needed to connect modules across domains. It might differ from case to case whether one considers these modules as masters and slaves or not. Either way, it can be helpful to bypass bridges so that signals written by the interconnecting module can reach modules on the other side. In order to do so, the solution provides a macro for "relaying" a signal across a module. It works on the same principle of the interconnect macro "*INTERCONNECT_BUSSES*", seen in Listing 4.3, where both the input and output ports are associated with each other, only here we mark them as relays.

Listing 4.5 shows an example where a master module connects to an interconnecting module through a bridge. The extraction code writes a signal on the *DataMasterRead* bus output, and reads the signal coming in on the *DataSlaveRead* input bus. The macro "*RELAY*" associate the signal coming in from the interconnecting module with the signal itself wrote out to the master or slave module. An external tool can determine a connection between the input and output of the in-between module based on this information, as we will see in Chapter 5.4. Section 4.3.4 explains the macro in further details. Note that the *RELAY* macro does not have built-in read and write operations, so they must be declared separately, as shown in the example.

```
1  module bridge (
2      // ...
3      input   [31:0]  DataSlaveRead,
4      output  [31:0]  DataMasterRead
5      );
6
7      initial begin
8          fork
9              `FORCE_AND_RELEASE_BUS(DataMasterRead, "DataMasterRead", )
10             `READ_BUS(DataSlaveRead, "DataSlaveRead", )
11             `RELAY(DataSlaveRead, DataMasterRead, )
12         join
13     end
14
15     // ...
16 endmodule
```

Listing 4.5: Example of extraction code, where a signal is relayed across a bridge module.

Some Thoughts on the Implementation

The goal of this implementation is to be as small and simple as possible. For that reason, only the master and slaves modules, together with the interconnecting module and bridges, are registered. One could, alternatively, expand on the writing and reading of busses. For instance, one could write and read at both ends, detecting that busses have been appropriately implemented and not partially connected. It is also possible to expand on what information to extract and what modules to include in the extraction; This would most likely take shape as new macros introduced alongside the existing ones. However, the aim was for a simplistic solution, and the thesis has settled for the implementation described. Chapter 7 describes what could be improved and added to the solution in greater details.

It is also worth noticing that we place the more complex behaviour in modules that typically end up as IPs, such as bridges and interconnecting modules; This is a deliberate decision, as it simplifies the extraction process in future projects. A majority of the extraction code will already be implemented when the IPs are reused in other RTLs.

4.3 Extraction Macros for Reading & Writing to Busses

Section 4.2 briefly mentioned that the way one determine which module connects to the bus is by writing a value to that bus and check which module can read that value. For that purpose, the extraction code has a few macros for writing to- and reading the bus. In both cases, the macros handle any necessary operations, and the user only has to supplement with a few arguments, like the instance-name of the bus wires. Each value that is written to a set of bus wires is to be unique as well. By doing so, one essentially give each bus an identification number. One can determine that a module is connected to the bus wires when the module can read the unique value on the bus. It also lets one associate different sets of wires with one another, as shown in Section 4.3.4 and 4.4.

The extraction code enforces unique numbers on each write operation through a function that returns a unique integer number on every call. As one can see in Listing 4.6, it is simply a function that increments a global integer and returns the value. Each write operation will call on *getNewForceValue* to get a unique value. Since the incrementing global integer is located in the extraction code, one will only have one instance of it, thus enforcing a unique value on each call to *getNewForceValue*.

```
1  integer forceValue = 1;
2
3  function integer getNewForceValue();
4      return forceValue++;
5  endfunction : getNewForceValue
```

Listing 4.6: Function for getting a unique integer value.

An alternative could be to use random values instead of an incrementing value. HDLs such as SystemVerilog have built-in system methods to produce random values intended for test benches and similar; This could produce a smaller and less complex code. However, this thesis wants to enforce unique values on each write operation, and a random-method would not provide that. The property of uniqueness is the only property we are interested in and are vital for the extraction solution. Therefore, using the random-methods is not sufficient for our solution. One could try to create a method that ensures uniqueness amongst the randomly generated values, but it would likely be more complex than the solution presented in Listing 4.6.

4.3.1 Sequence of Operations

Before one can delve into the inner workings of the read and write operations used in the extraction code, one must first understand the execution order of these operations. In short, we allocate specific time slots for certain operations to make sure everything executes as intended. Note that this sequence of operations relies on the theory and properties of RTL simulation, presented in more detail in Chapter 2.4.

One of the properties of our solution is that it should be as little interference with the RTL as possible. The way the thesis solves that problem is by disabling the extraction code when not extracting information. The same is true the other way around; Neither the RTL nor other simulation tools should interfere with the extraction. Ideally, the solution should be universal, without a great need to modify the extraction code for each case. One can not make too many assumptions on how the RTL will behave during simulation, as it will differ between RTLs, and even between iterations of the same RTL. Therefore, it is better to avoid interference altogether.

The thesis' solution is to have all read- and write operations occur in the first time unit of the simulation. Specifically, one asserts (writes) values in the first delta-cycle, read values in the second, and deassert values in the third cycle. Figure 4.7 visualises this, where each operation is assigned to a specific delta-cycle time-slot. One needs a time slot for deassertion since the implementation use the *force* command for assertion. The *force* command will override any other assignments until deasserted using the command *release*. In that sense, one could also label the last time slot as a clean-up slot.

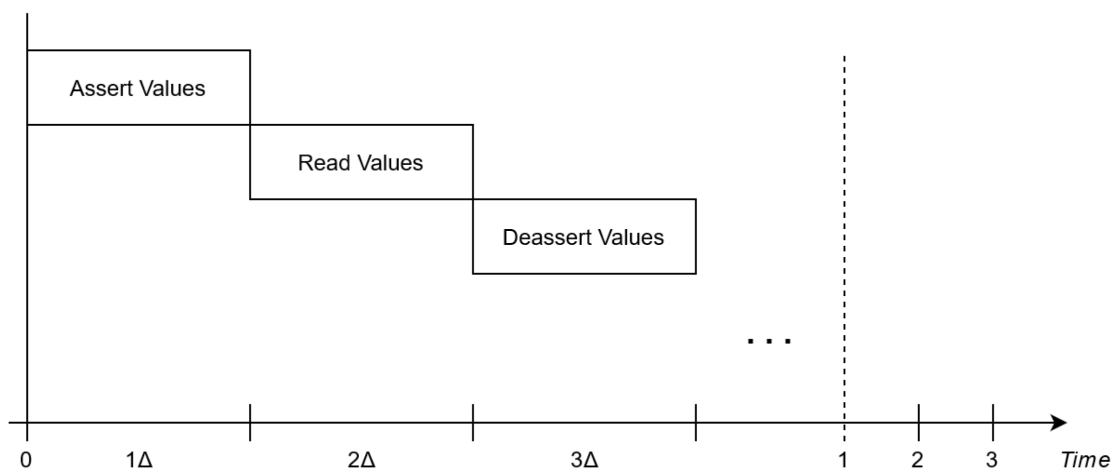

Figure 4.7: Sequence of operations during extraction, where each type of operation is assigned to a specific delta-cycle.

Having all write operations happen before one read ensures that it does not accidentally read an "empty" data bus before writing to it. This solution is also quick and effective, as one do not have to simulate the RTL over a long period. One can end the simulation after the first time unit, as all operations happen on the three first delta cycles.

The obvious downside to this solution is that it is rigorous. Each operation, or macro, must follow the sequence of operations. Each instance of the macros executing the operations must happen at the beginning of the simulation. Additionally, each instance of the macros must execute concurrently to avoid it schedule its operations at a later delta cycle.

An alternative is to not use such a strict sequence of operations within such a small time frame. For example, one could expand the time frame to last more than a single delta cycle. However, there is not much merit to this, as it only serves to make the extraction time longer and increase the possibility of interference from the RTL or simulation. Another alternative could be to avoid assigning a time slot for each operation altogether. Consequently, the extraction code would have to implement some form of a control unit to keep track. This unit would have to make sure modules would not read a bus that has not been written yet. Such a unit would only bring more complexity to the solution without improving beyond what is covered with the thesis' original proposal to limit operations by delta cycles.

4.3.2 Write Operations

As one might guess, write operations enforces a specific value onto a set of wires. A feature for the extraction code is that the value should override all other assignments continuously until the extraction process finishes; This is done by utilising the *force* and *release* statements for SystemVerilog. Any assignments by a *force* statements overrides other assignments and stays that way until deassigned with the *release* statement.

Listing 4.7 shows the macro *FORCE_VALUE_ON_BUS* for forcing a specified value on a data bus and exports the operation as information to the simulation log. It uses the instance name of the data bus, the value written to the bus, and the module's name as arguments. The module name has a default value that is the hierarchical path of the module (or, more specifically, the caller of the macro). In addition, one should release the data busses when finished with the extraction; Listing 4.8 shows the macro *RELEASE_BUS* that does just that.

```
1  `define FORCE_VALUE_ON_BUS(DATA_BUS, VALUE, MODULE_NAME=$sformatf("%m")) \
2  `ifdef ENABLE_BUS_EXTRACTION \
3         force DATA_BUS = VALUE; \
4         `WRITE_VALUES_TO_LOG(MODULE_NAME, `INFO_FORCE, VALUE, ) \
5  `endif
```

Listing 4.7: Macro for writing an value to a data bus.

There are other assignment statements in the SystemVerilog language that could be used as an alternative to *force*, like the *assign* method. However, only the *force* method can deliver both a continuous and overriding assignment. Only a *force* method can override anything already assigned by an *force* method in the SystemVerilog language. For that reason, the thesis considers *force* the most reliable type of assignment for the operations it has in mind, even though it requires one to release the bus afterwards.

```
1 `define RELEASE_BUS(DATA_BUS) \
2 `ifdef ENABLE_BUS_EXTRACTION \
3         release DATA_BUS; \
4 `endif
```

Listing 4.8: Macro for releasing a data bus after a *force* assignment.

In Section 4.3.1 it was specified that macros must follow a strict sequence of operation during execution. In the writing operation, we need to use the *force* method in the first delta cycle and the *release* method in the third delta cycle. In addition, it would be beneficial to read the bus written to in the second delta cycle, as it can verify that the written value is present on the bus. Listing 4.9 shows an macro where all three operations, organises so that they follow the timing defined in Section 4.3.1. Note that it uses *#0* delay to reschedule to the next delta cycle. It also uses the *getNewForceValue* from Listing 4.6 to get a unique value for each write operation.

```
1  `define FORCE_AND_RELEASE_BUS(DATA_BUS, BUS_NAME="",
   ↪  MODULE_NAME=$sformatf("%m")) \
2  `ifdef ENABLE_BUS_EXTRACTION \
3          begin \
4                  static int value_to_force = getNewForceValue(); \
5                  `FORCE_VALUE_ON_BUS(DATA_BUS, value_to_force, MODULE_NAME) \
6                  #0; \
7                  `WRITE_VALUE_AND_TEXT_TO_LOG(MODULE_NAME, `INFO_READ,
                   ↪  DATA_BUS, BUS_NAME) \
8                  #0; \
9                  `RELEASE_BUS(DATA_BUS) \
10         end \
11 `endif
```

Listing 4.9: Collection of force, read, and release operations for a single data bus.

The data busses can sometimes be arranged in an array. Since Listing 4.9 is intended for a single data bus, one can instead use the macro *FORCE_AND_RELEASE_BUS_ARRAY* from Listing 4.10. This macro is a bit more complicated, as it must take certain measures to work around the limitations of the *force* statements. First of all, it needs to create a local variable to contain the values to be written. The reason is that the *force* does not allow automatic variables. The macro also needs more arguments, specifically the size of the array and the data bus's width, to function. Since one should use the force, read, and release operations in all cases where one writes a value to a data bus, it makes sense to use the macros presented in Listing 4.9 and 4.10.

```
1  `define FORCE_AND_RELEASE_BUS_ARRAY(DATA_BUS_ARRAY, ARRAY_SIZE, BUS_SIZE,
   ↪  BUS_NAME="", MODULE_NAME=$sformatf("%m")) \
2  `ifdef ENABLE_BUS_EXTRACTION \
3          begin \
4                  logic [ARRAY_SIZE-1:0][BUS_SIZE-1:0] ValueForBusArray; \
5                  foreach (ValueForBusArray[i]) begin \
6                          ValueForBusArray[i] = getNewForceValue(); \
7                  end \
8                  force DATA_BUS_ARRAY = ValueForBusArray; \
9                  foreach (ValueForBusArray[i]) begin \
10                         `WRITE_VALUES_TO_LOG(MODULE_NAME, `INFO_FORCE,
                           ↪  ValueForBusArray[i], ) \
11                 end \
12                 #0; \
13                 foreach (DATA_BUS_ARRAY[i]) begin \
14                         `WRITE_VALUE_AND_TEXT_TO_LOG(MODULE_NAME,
                           ↪  `INFO_READ, DATA_BUS_ARRAY[i],
                           ↪  $sformatf("%s[%0d]", BUS_NAME, i)) \
15                 end \
16                 #0; \
17                 `RELEASE_BUS(DATA_BUS_ARRAY) \
18         end \
19 `endif
```

Listing 4.10: Collection of force, read, and release operations for an array of data busses.

4.3.3 Read Operations

Reading a value from bus wires is pretty straightforward. The macro reads the value currently present on the data bus, then export that information to the simulation log. Listing 4.11 shows a macro like that, which accepts a data bus instance, a bus name, and a module name as arguments. Note that it uses similar arguments as those for the write macros in Listing 4.7.

```
1  `define INFO_READ "ReadBus"
2
3  `define READ_BUS(DATA_BUS, BUS_NAME="", MODULE_NAME=$sformatf("%m")) \
4  `ifdef ENABLE_BUS_EXTRACTION \
5          begin \
6                  #0; \
7                  `WRITE_VALUE_AND_TEXT_TO_LOG(MODULE_NAME, `INFO_READ,
                   ↪  DATA_BUS, BUS_NAME) \
8          end \
9  `endif
```

Listing 4.11: Macro for reading a data bus and exporting the information.

The first thing to notice is that the only thing that the macro really does is call the export macro, which writes specified value and text to the log with the data bus value as its argument. In addition, it marks the information as "ReadBus"-information, or in other words, a value obtained by reading a data bus. The specifics of the *WRITE_VALUE_AND_TEXT_TO_LOG* are handled in Section 4.5. In Section 4.3.1, we stated that read operations must happen on the second delta-cycle of the simulation. We do so by calling the delay *#0*, effectively postponing execution to the next delta cycle.

The macro *READ_BUS* from Listing 4.11 is intended to be used for a single data-buss. If the busses are organised in an array, one can use the macro *READ_BUS_ARRAY* from Listing 4.12 instead. The principles are pretty much the same as for the *READ_BUS* macro. The only difference is that it iterates through the data bus array, reading one at a time.

```
1 `define READ_BUS_ARRAY(DATA_BUSSES, BUS_NAME="",
  ↪  MODULE_NAME=$sformatf("%m")) \
2 `ifdef ENABLE_BUS_EXTRACTION \
3         begin \
4                 #0; \
5                 foreach (DATA_BUSSES[i]) begin \
6                         `WRITE_VALUE_AND_TEXT_TO_LOG(MODULE_NAME,
                          ↪  `INFO_READ, DATA_BUSSES[i], $sformatf("%s[%0d]",
                          ↪  BUS_NAME, i)) \
7                 end \
8         end \
9 `endif
```

Listing 4.12: Macro for reading an array of data busses and exporting the information.

There are not many alternatives when it comes to how one would execute the read operation. It observes the data bus at a specified time slot and registers the value as an entry in the log. One could argue there are different ways of doing the extraction in general, but that is better to discuss this in more relevant sections.

4.3.4 Relaying of Signals

A type of module one often finds in bus systems is bridges, an interconnecting module that converts bus communication between different domains. For example, there might be domains with different operating frequencies. Some logical operations are often necessary to convert the signal before it can cross between domains and have to use bridges to do so. Whether one would consider them masters and slaves depends on the definition, but it is helpful to allow a signal to pass through the bridge.

The thesis solution is to relay the signal across the intermediate modules, effectively bypassing them. It does so by reading the input signal, writing a new signal to the output, and associating these two values as "connected". Figure 4.8 illustrates this concept. Note that we are not connecting these signals physically but are stating that any signal on the input will eventually reach the output.

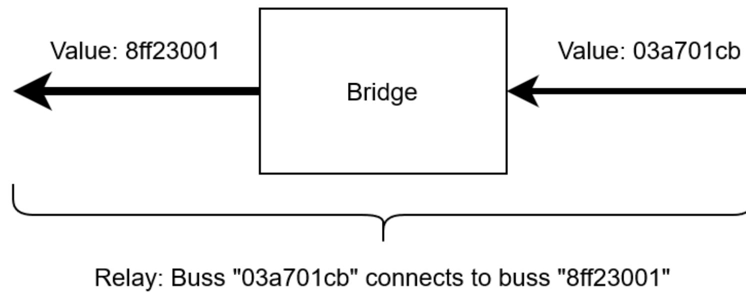

Figure 4.8: The signal is relayed across the bridge by associating the input value with the output value.

Listing 4.13 presents the code for relaying a signal across a module. It is quite similar to the "read"-macros presented in Section 4.3.3. The difference is that it accepts both a data bus going in and a data bus going out. It also categorises the information as a "relay" type of information.

```
1  `define INFO_RELAY "Relay"
2
3  `define RELAY(DATA_BUS_IN, DATA_BUS_OUT, MODULE_NAME=$sformatf("%m")) \
4  `ifdef ENABLE_BUS_EXTRACTION \
5          begin \
6                  #0; \
7                  `WRITE_VALUES_TO_LOG(MODULE_NAME, `INFO_RELAY, DATA_BUS_IN,
                   ↪  DATA_BUS_OUT) \
8          end \
9  `endif
```

Listing 4.13: Macro for relaying a signal across a module by associating the value on the data bus going in and out.

As with the other macros that accept a data bus, one should consider the cases where several busses are arranged into an array. In those cases, one can use the macro in Listing 4.14. It is pretty similar to the macro presented in Listing 4.13, only it iterates through each element of the arrays. Note that the macro in Listing 4.14 require the data bus arrays to be of the same size. Another expectation is that the array elements pairs up sequentially; I.e., DATA_BUSSES_IN[0] signal relays to DATA_BUSSES_OUT[0], DATA_BUSSES_IN[1] signal relays to DATA_BUSSES_OUT[1], and so on.

An alternative solution could be to pass the value in the input directly to the output. Doing so would be more of actually relaying a signal across the module. There is, however, an issue with this approach. The sequence of operations defined in Section 4.3.1 does not allow writing after reading. Relaying the value requires knowing the value written on the input before writing it to the output. Thus, we would have to redefine when one can do what. For example, the implementation could iterate through the sequence of operations as many times as needed to allow the signal to propagate through the bridges. Doing so would bring a whole new layer of complexity to the extraction solution without improving the existing solution.

```
1  `define RELAY_ARRAY(DATA_BUSSES_IN, DATA_BUSSES_OUT,
   ↪  MODULE_NAME=$sformatf("%m")) \
2  `ifdef ENABLE_BUS_EXTRACTION \
3          begin \
4                  #0; \
5                  foreach (DATA_BUSSES_IN[i]) begin \
6                          `WRITE_VALUES_TO_LOG(MODULE_NAME, `INFO_RELAY,
                           ↪  DATA_BUSSES_IN[i], DATA_BUSSES_OUT[i]) \
7                  end \
8          end \
9  `endif
```

Listing 4.14: Macro for relaying signasl across a module by associating the value on the data bus arrays going in and out.

4.4 Extraction Macros for Interconnecting Modules

Section 4.2 described the interconnecting module as the module containing arbiters, decoders, and anything else needed to interconnect the modules. In other words, the interconnecting module contains a large part of the bus system. Consequently, the interconnecting module is an excellent source for bus-related information that we can extract. However, the challenge lies in how and where one can extract the information inside the module; This is further complicated because different bus systems will have different implementations of the interconnecting module. Therefore, basing the extraction on the logic and how the bus system is implemented will not translate well between systems.

The thesis' solution is to exploit the connectivity list that often implemented with the interconnecting module. This list typically takes the shape of an array or matrix, defining which slave a master can access. The way it extracts this information is by associating ports with each other, similar to the relaying in Section 4.3.4. By following the connectivity list, one can map out the connections. Listing 4.15 shows the macro that has a connection matrix (i.e., a multi-dimensional array) and two bus arrays as arguments. It iterates through the busses and exports the pairs that are connected. Typically, one would have busses going to masters in one array and busses going to the slaves in the other. Which array should be considered first or second depending on how the *CONNECTION_MATRIX* is structured, as one can expect the connection matrix to be on the form *[Size_of_first_array-1][Size_of_second_array-1] CONNECTION_MATRIX*.

```
1  `define INFO_INTERCONNECT "Interconnect"
2
3  `define INTERCONNECT_BUSSES(CONNECTION_MATRIX, FIRST_BUS_ARRAY,
   ↪  SECOND_BUS_ARRAY, MODULE_NAME=$sformatf("%m")) \
4  `ifdef ENABLE_BUS_EXTRACTION \
5          begin \
6                  #0; \
7                  foreach (FIRST_BUS_ARRAY[i]) begin \
8                          foreach (SECOND_BUS_ARRAY[j]) begin \
9                                  if (CONNECTION_MATRIX[i][j] != 0) begin \
10                                         `WRITE_VALUES_TO_LOG(MODULE_NAME,
                                           ↪  `INFO_INTERCONNECT,
                                           ↪  FIRST_BUS_ARRAY[i],
                                           ↪  SECOND_BUS_ARRAY[j]) \
11                                 end \
12                         end \
13                 end \
14         end \
15 `endif
```

Listing 4.15: Macro for mapping out the connections between bus arrays.

As with previous macros, one uses the value written to a bus to identify it. That means that one must use the write macros from Section 4.3.2 in addition to the interconnect macro in Listing 4.15. The implementation example, presented in Section 4.2, is a good example of this.

While one can determine which port on the interconnecting module is connected, we can not determine which port leads to masters or slave from interconnections alone. As such, a more manual approach is needed. The thesis' solution provides macros for the user to mark ports and busses manually. Listing 4.16 shows a couple of macros that marks whether a bus wire is intended for a master or a slave. It is similar to the read-macros presented in Section 4.3.3, as it extracts the value on the bus and marks it with the relevant type. In addition, the thesis has defined macros for data busses arranged in arrays in Listing 4.17.

```
1  `define INFO_MARK "MarkBus"
2
3  `define MARK_BUS_AS_MASTER(DATA_BUS, MODULE_NAME=$sformatf("%m")) \
4  `ifdef ENABLE_BUS_EXTRACTION \
5          begin \
6                  #0; \
7                  `WRITE_VALUE_AND_TEXT_TO_LOG(MODULE_NAME, `INFO_MARK,
                   ↪  DATA_BUS, "Master") \
8          end \
9  `endif
10
11 `define MARK_BUS_AS_SLAVE(DATA_BUS, MODULE_NAME=$sformatf("%m")) \
12 `ifdef ENABLE_BUS_EXTRACTION \
13         begin \
14                 #0; \
15                 `WRITE_VALUE_AND_TEXT_TO_LOG(MODULE_NAME, `INFO_MARK,
                   ↪  DATA_BUS, "Slave") \
16         end \
17 `endif
```

Listing 4.16: Macro for marking whether the bus is intended for masters or slaves.

```
1  `define MARK_BUSSES_AS_MASTER(DATA_BUSSES, MODULE_NAME=$sformatf("%m")) \
2  `ifdef ENABLE_BUS_EXTRACTION \
3          begin \
4                  #0; \
5                  foreach (DATA_BUSSES[i]) begin \
6                          `WRITE_VALUE_AND_TEXT_TO_LOG(MODULE_NAME,
                           ↪  `INFO_MARK, DATA_BUSSES[i], "Master") \
7                  end \
8          end \
9  `endif
10
11 `define MARK_BUSSES_AS_SLAVE(DATA_BUSSES, MODULE_NAME=$sformatf("%m")) \
12 `ifdef ENABLE_BUS_EXTRACTION \
13         begin \
14                 #0; \
15                 foreach (DATA_BUSSES[i]) begin \
16                         `WRITE_VALUE_AND_TEXT_TO_LOG(MODULE_NAME,
                           ↪  `INFO_MARK, DATA_BUSSES[i], "Slave") \
17                 end \
18         end \
19 `endif
```

Listing 4.17: Macro for marking whether the array of busses are intended for masters or slaves.

4.5 Exporting the Extracted Information

So far, this thesis has explored how to extract various types of information from RTL. Another vital task is to be able to view and export this information outside of the simulation environment. It would not be of much use to extract information if the user could not view it. In addition, the solution aims to process the extracted information into more useful data in a separate tool, covered in more detail in Chapter 5. All information is exported by writing said information to the simulation log. The idea is that collecting all the extracted information into the log allows the user to import the information by parsing the text. It is also easily achievable by using the SystemVerilog *$display* system task, which prints text to the standard output, which in most cases are the simulation log.

The solution chooses to organise its extracted information into small data points before exporting it to the simulation log. The idea is that each operation described earlier in this chapter will produce a small data point describing the operation and its result. For instance, the read operation from Section 4.3.3 would produce a data point that describes itself as a read operation, which module called the operation, and the value it read (i.e., the result). Since data points would be similar to other operations, we can generalise it into the structure shown in Listing 4.18.

```
#bus#<name of module>#<type of information>#<value 1>#<value 2>#
```

Listing 4.18: Template for structuring data points from extraction processes.

The generalised structure has fields intended for specific information; One field is intended for the module name that calls the operation, one field for defining what type of information it is, and two fields for values resulting from the operation. It also uses the character # as a delimiter and *bus* as a keyword to identify the text as a data point related to the extraction solution.

There are a few benefits to use a generalised template for exporting the information into data points. It allows one to call the same functions or macros whenever it extracts new information. It is also helpful for any tool that imports the information, as it is easier to parse into the tool. The drawback is that the extracted information can be overwhelming without some form of processing. Since our approach divides the information into tiny bits, the results are a long and complex list of data points that can be difficult to interpret. As a consequence, the processing tool in Chapter 5 becomes even more important in order to produce a result that is useful for the user. One does not have to follow the structure we have defined in Listing 4.18 either. It could be both simpler or more complex, without significant changes to the solution complexity. In most cases, it is a matter of personal opinion. However, as stated earlier, the structure chosen for this solution is based on its simplicity and how easy it is to parse.

As mentioned, the solution export the extracted information to the standard output through the *$display* system task. Listing 4.19 shows an macro that uses the generalised data point structure presented in Listing 4.18, along with a *$display* task. It accepts up to two values, in addition to the module name and type of information.

There are cases where one wants to include a string of text as one of the values. It would not be ideal to use the macro in Listing 4.19, as it expects an integer and would interpret it as one. Instead, one can use the macro in Listing 4.20; This macro accepts both a value and a string. The read operation from Section 4.3.3 is a good example, where the name of the bus is included with the value when writing the data point.

```
1 `define WRITE_VALUES_TO_LOG(MODULE_NAME, TYPE_OF_INFORMATION, VALUE_1,
  ↪  VALUE_2 = 0) \
2 `ifdef ENABLE_BUS_EXTRACTION \
3         $display("#bus#%s#%s#%h#%h#", MODULE_NAME, TYPE_OF_INFORMATION,
          ↪  VALUE_1, VALUE_2); \
4 `endif
```

Listing 4.19: Macro for writing a data point to the standard output. Accepts up to two values.

```
1 `define WRITE_VALUE_AND_TEXT_TO_LOG(MODULE_NAME, TYPE_OF_INFORMATION, VALUE,
  ↪  TEXT) \
2 `ifdef ENABLE_BUS_EXTRACTION \
3         $display("#bus#%s#%s#%h#%s#", MODULE_NAME, TYPE_OF_INFORMATION,
          ↪  VALUE, TEXT); \
4 `endif
```

Listing 4.20: Macro for writing a data point to the standard output. Accepts a value and a string.

While we export the extracted information to the standard output, it is also possible to create and use a new file for this purpose. However, it would require that the operations open and close the file every time it is used, impacting the extraction time. Alternatively, it could be opened at the start of the extraction and have a global handle that the operations can reference before closing it at the end of the extraction. Although it is slightly more complicated, it is not that different from using the standard output.

Chapter 5

Processing Bus Information

Chapter 4 dealt with the extraction of bus-related information from RTL. The extraction solution exported that information into a text file, intended for further processing at a higher level; That is the purpose of the tool described in this chapter. The processing tool is a script written in the high-level programming language Python [25], intended for importing, processing, and operating on the extracted data. These operations allow the user to understand better and visualise the information extracted from the RTL. This thesis chose Python, as it is quick to get up and running, and there are many existing libraries, called modules, available to support the solution. However, one could use other high-level languages such as C instead, without any significant impact on the solution. The difference would be that one must use different libraries and possibly implement more of the functionality themselves. In addition, the thesis utilise the graph visualisation tool Graphviz [26] to visualise the extracted information. It is open-source, with python modules readily available for integration into the solution.

The processing tool imports the text file by parsing and structuring it into more usable, raw data. Since the raw data is organised as many small data points, one also needs to process it. The goal of the processing is to make it more compact and manageable. After the raw data has been processed, the tool has everything it needs to operate on and visualise the extracted information. The source code for the processing tool can be found in Appendix B. Section 5.1 will give a brief overview of the solution, before moving on to more detailed explanations.

5.1 Brief Overview of the Processing Solution

The idea behind the processing tool is as follow: after the information in Chapter 4 have been extracted and exported into a text file, the processing tool would import and process it. Once that is done, the user could both look through the information and perform various operations.

When the user imports a text file into the tool, the first thing that happens is that the text is parsed for data points defined in Chapter 4.5. The parser looks for data points in the text and stores them in a list, passed on to the processing part of the tool after parsing. The processor iterates through the data points and creates new lists with more specific information. Some information can be taken directly from the data points, but other types of information must be deduced from the data points. After processing is finished, the tool has some data lists that it can use and operate on.

The tool is written so that all parsing and processing happens when a file is imported. Afterwards, the user can execute any operations they want by giving the relevant command. Section 5.2 describes the available commands in the tool and how one would use them, while the subsequent sections describe how the commands work in detail.

5.2 Using the Processing Tool

As mentioned earlier, the processing tool is a python script containing everything one needs to process and operate on the extracted information. The tool does not use a graphical user interface but instead a command line with an interpreter. The implementation in this thesis uses a Linux computer and command line for the examples. All operations are assigned a specific command for the user to use. In addition, there are a few arguments that can be used upon the startup of the script.

To open the tool, one calls on the main python file *bus_tool.py* together with the interpreter, as shown in Figure 5.1. At this stage, there is no data imported into the tool. The option for reducing the length of the names on modules are on by default and are covered in more detail in Section 5.5.

```
➜  ~/BusExtractionFiles/python_processor git:(master) python bus_tool.py
Processing tool for extracted bus information
---------------------------------------------
Data loaded: None
Reduced Names: True
---------------------------------------------
>> 
```

Figure 5.1: The interface of the processing tool when opened.

Since each operation is assigned a command, having a way of viewing available commands can be helpful. The *help* command allows one to view a list of available commands, as shown in Figure 5.2. It also lists available arguments for the user when starting the processing tool, such as importing a file at startup. To exit the application, one can use either the *quit* or the *exit* command.

```
---------------------------------------------
>> help
----------------------------COMMANDS-------------------------------
    import <filename.txt>       Import meta-information from specified textfile.
    listmod                     Display a list of all modules from extracted data.
    listcon                     Display a list of all connection between modules.
    listmas                     Display a list of all masters, and slaves connected to the master.
    genmas                      Generate a pdf image of the relationship between masters and slave.
    gencon                      Generate a diagram of the connection between modules in the system.
    togname                     Toggle whether name reduction is enabled or not.
    help                        Shows this list of available commands.
    quit                        Quits the tool. (Can also use 'exit'.)
----------------------------ARGUMENTS------------------------------
    -i <filename.txt>           Import file at initialization of the tool
    -n                          Turn off name reduction at startup
    -d                          Start tool in debug mode
-------------------------------------------------------------------
>> 
```

Figure 5.2: List of available commands for the processing tool.

Importing the Information

There are two ways for the user to import information into the processing tool. The first one is to use the argument *-i*, followed by the file name, during startup. The processing tool will then automatically import the file at startup. Figure 5.3 shows an example where the import argument is used to import the file *log_nordic.txt* at startup.

```
→ ~/BusExtractionFiles/python_processor git:(master) python bus_tool.py -i log_nordic.txt
Processing tool for extracted bus information
--------------------------------------------
Parsing log_nordic.txt...
Done
Obtaining list of modules...
Done
Obtaining list of busses...
Done
Obtaining list of masters and their slaves...
Done
Processing finished. Imported 44 modules.
Data loaded: log_nordic.txt
Reduced Names: True
--------------------------------------------
>>
```

Figure 5.3: Example of the processing tool importing a file at startup.

If the tool is already up and running, the user can import a file by using the *import* command, followed by the file name. It works in the same way as using the import argument, as shown in Figure 5.4.

```
→ ~/BusExtractionFiles/python_processor git:(master) python bus_tool.py
Processing tool for extracted bus information
--------------------------------------------
Data loaded: None
Reduced Names: True
--------------------------------------------
>> import log_nordic.txt
Parsing log_nordic.txt...
Done
Obtaining list of modules...
Done
Obtaining list of busses...
Done
Obtaining list of masters and their slaves...
Done
Processing finished. Imported 44 modules.
>>
```

Figure 5.4: Example of the processing tool importing a file.

When the tool imports a file, it also automatically processes the data. Since this process is automatic, the user does not have to supplement with any additional information. Once a file is imported, the user can operate on the data.

Listing the Data

The tool contains several commands that allow the user to browse the processed data in the form of lists. The commands are as follows:

- *listmod*: List all modules involved in the extraction process. For instance, masters, slaves, and bridges.
- *listcon*: List all connections between modules. These connections are coherent to the read and write operations in Chapter 4.3, used for the extraction code. For each connection, it lists the write source (i.e., the module that wrote the signal to the connection) and the read sources (i.e., the modules that observed the value on the connection). In addition, the names given to the connection by modules are included where applicable. Figure 5.5 shows an example of the command. As such, these connections are not equivalent to busses but are a direct connection between modules.
- *listmas*: List all master modules, along with slaves that the masters connect to.

```
>> listcon
---------------------------CONNECTIONS----------------------------
Name(s) of Connection: {'ahbHRDataMaster[1]'}
        Write Source: [illegible]
        Read Source(s): {'u_CpuCore', [illegible] }

Name(s) of Connection: {'ahbHRDataMaster[0]'}
        Write Source: [illegible]
        Read Source(s): {'u_CpuCore', [illegible] }

Name(s) of Connection: {'ahbHWDataSlave[0]'}
        Write Source: [illegible]
        Read Source(s): {'[illegible]}

Name(s) of Connection: {'ahbHRData', 'ahbHRDataMaster[3]'}
        Write Source: [illegible]
        Read Source(s): {'[illegible] }

Name(s) of Connection: {'ahbHRData', 'ahbHRDataMaster[2]'}
        Write Source: [illegible]
        Read Source(s): {[illegible] }
```

Figure 5.5: Example of the *listcon* command, listing all connections in the bus structure.

Visualising the Data

The tool provides two commands for generating diagrams of relevant information, stored as a PDF file. The first one is the *gencon* command that generates a connection diagram; This visualises the various modules and the connections between modules. In a sense, it is a visualisation of the connection list generated from the *listcon* command. Figure 5.6 shows an example of such a diagram, albeit at a low resolution due to its size. While not straightforward, it does provide a visual way of exploring the RTL.

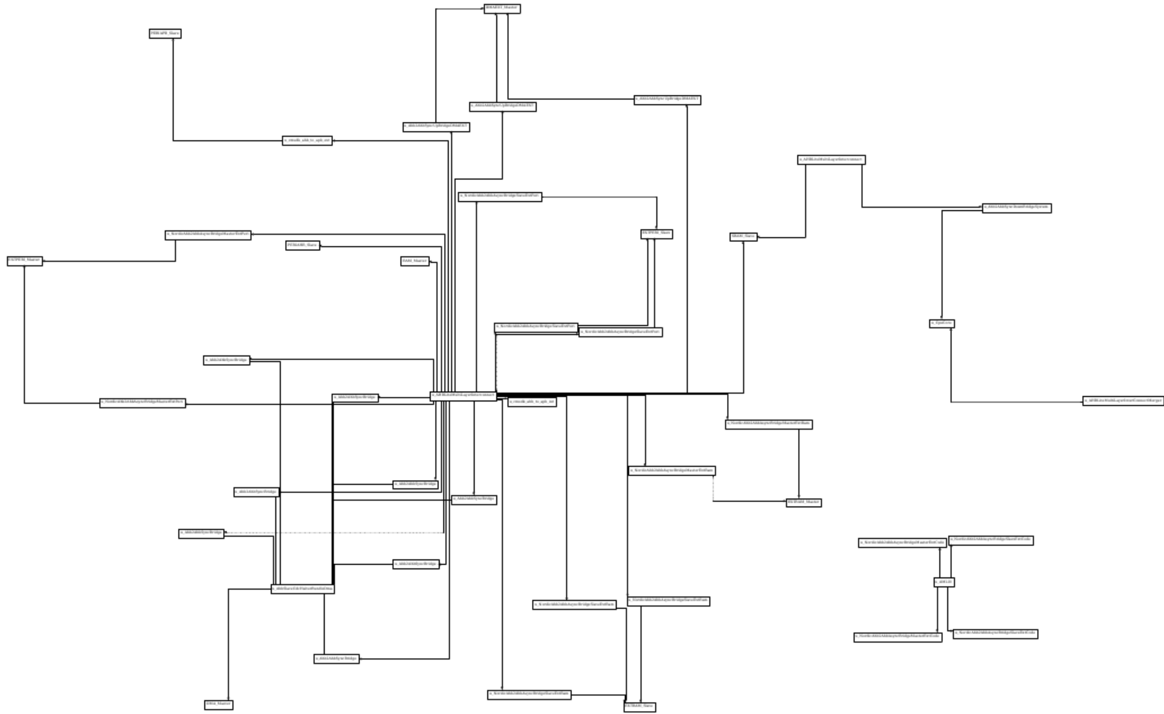

Figure 5.6: Example of a connection diagram generated from extracted data using the *gencon* command.

The other visualisation operation is the *genmas* command. It generates a master-slave diagram depicting which slave module a master module can reach. In some ways, one can consider this a condensed version of the diagram shown in Figure 5.6, with the added feature of detailing who is a master and who is a slave. Figure 5.7 shows an example of a diagram generated from the *genmas* command. The module the arrowhead points at is the slave, while the source of the arrow is the master. Note that this is generated from the same data as the diagram from Figure 5.6.

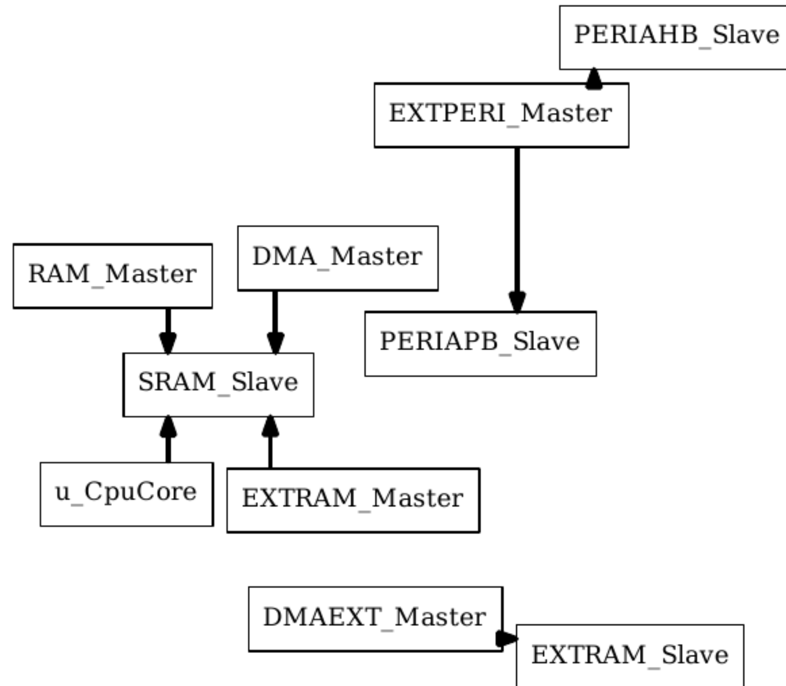

Figure 5.7: Example of a master-slave diagram generated from extracted data using the *genmas* command.

5.3 Importing Extracted Information

The goal of the processing tool is to give the user a platform for processing and to operate on the extracted information from Chapter 4. However, one must first import the information into the tool, preferably in a state that is easy to utilise. The thesis' solution is to parse the information exported from the extraction code through a text file. A text file is easy to write and read from, and the implementation can be flexible in how information is stored and parsed.

An alternative could be to connect the processing tool directly to the extraction code. SystemVerilog has introduced interfaces specifically for communicating with C routines, allowing SystemVerilog to call on the routines as if they were an ordinary SystemVerilog routine [16]. Such a solution would be a great deal more complicated, especially considering that we only need to move data from one platform to another. It is, however, an interesting prospect to consider if one wants to integrate the processing part into the extraction solution from Chapter 4.

Chapter 4.5 defined a strict structure for organising the extracted information. All information is divided into small data points, defined by the template in Listing 4.18. The information making up the data point had to follow a specific order as well. Using a predetermined keyword allows the parser to quickly identify the beginning of a data point from the imported text file; This is especially useful when the extracted information is exported with other redundant information, for instance, through the simulation log. With a predefined delimiter to split apart different fields in the data point, it is easy to parse the strings into usable data. Listing 5.1 shows the function used to convert a string into a data point. It starts by looking for the keyword. If found, it then splits the string into sub-strings and assigns each sub-string to a field in a new data point.

```
1  def string_to_data_point(string):
2      keyword_index = string.rfind(KEYWORD)
3      if keyword_index != -1:
4          string_after_keyword = string[keyword_index + KEYWORD_LENGTH:]
5          split_string = string_after_keyword.split(DELIMITER)
6          new_data_point = dp.DataPoint(split_string[0], split_string[1],
           ↪  split_string[2], split_string[3])
7          return new_data_point
8      else:
9          # No valid data point can be extraced from string
10         return None
```

Listing 5.1: Function for converting a string into a data point.

The benefit of having such a strict structure on the data points when exported is that it knows what information is located where in the string without having to do rigorous test for values and determine its nature. In addition, it gives the implementation a much simpler parser, where the most demanding work is to locate the data points. Note that we are using a class for storing the data points, which can be found in Listing B.4, along with the source code for the parser in Listing B.3.

The end goal of the parser is to create a list of data points that can be processed rather than a long string full of redundant information. Of course, one could pass the string directly to the processing part of the tool, but some form of parsing would still be needed; We would essentially shift the problem to a different part of the code without solving it.

5.4 Processing

After the tool imports and parses the extracted information, it is still a long list of data points that are cumbersome to use. The goal of the processing part of the tool is to reduce the size and compact the information into a more coherent collection of data. As a result, it will be easier to operate on, handle, and view. This section will refer to the parsed but unprocessed information as raw data.

Section 5.2 presented three commands for viewing information as lists. The user can view lists of modules, connections between modules, and master-slave pairs. It is these three lists we want to create from the raw data obtained in Section 5.3.

One can break down the processing operations as a series of iterations through the raw data. In each iteration, the tool will look for specific pieces of information. For instance, in one iteration, it might be looking for all data points classified as read-operations from Chapter 4.3.3, and one operation might be looking for data points classified as write operations from Chapter 4.3.2. The idea is that for each iteration, the tool can collect information from the data points and make deductions, and produce a list of relevant data for the user.

5.4.1 Obtaining a List of Modules

Obtaining a list of modules present in a bus design is a simple matter. The tool iterates through the raw data and records the names of each module before returning it as a list. Listing 5.2 shows a function that does just that. Since sets in python do not allow duplicates, it produces a list of unique names.

```
def get_modules_from_data_points(list_data_points):
    modules = set()
    for data_point in list_data_points:
        modules.add(data_point.name)
    return modules
```

Listing 5.2: Function for obtaining a list of modules from raw data.

5.4.2 Obtaining a List of Connections Between Modules

The purpose of the connections list is to describe the wires going between modules. In other words, the connected modules in this list physically connected with nothing else in between; This should not be confused with the more abstract list detailing which master is connected to which slave, described in Section 5.4.3

Chapter 4.3 looked at how one could use writing and reading operations to map out connections in a bus system. One module would write a unique value to its bus wires, and other modules would detect the value on the same bus. If one module could read what another had written, it is apparent that they are physically connected. The aim of the function in Listing 5.3 is to make that deduction from the raw data.

The function iterates through the raw data list and notes all data points describing a read or write operation. It wants to group operations that read or write the same value, which would mean the modules are connected. Using those values as keys creates entries in a dictionary collection, with a bus-class to store which module read or wrote the value. Listing 5.4 shows the values that can be stored in the bus class. The *__write_source* and *read_source* are intended to hold the names of the modules performing the write and read operations, respectively. The *__value* are intended for holding the value it reads and writes, while *names* holds the names that are given to the bus wires by modules, typical couples with the read operation. Note that there are class functions that are not shown here but can be found in Listing B.2. Once the function has iterated through all raw data points, it returns the dictionary.

```
def get_busses_from_data_points(list_data_points):
    busses = dict()
    # Iterate through all data points that are read- or write operations.
    for data_point in list_data_points:
        if data_point.info_type == INFO_READ or data_point.info_type ==
        ↪  INFO_FORCE:
            if verify_value(data_point.value_1):

                # Create new entry if not in dict
                if data_point.value_1 not in busses:
                    busses[data_point.value_1] = Bus(data_point.value_1)

                # Add info to entry
                if data_point.info_type == INFO_READ:

                    ↪  busses[data_point.value_1].add_read_source(data_point.name)
                    busses[data_point.value_1].add_name(data_point.value_2)
                elif data_point.info_type == INFO_FORCE:

                    ↪  busses[data_point.value_1].set_write_source(data_point.name)
    return busses
```

Listing 5.3: Function for obtaining a list of connection between modules from raw data.

```
class Bus:
    __value = ""
    __write_source = ""
    names = set()
    read_sources = set()
```

Listing 5.4: An excerpt from the *Bus* class detailing the values stored in the class.

5.4.3 Obtaining a List of Masters and Slaves

This function allows the user to obtain a list describing all master modules in a bus system and which slave modules the masters can reach through the bus system. As opposed to the connection list described in Section 5.4.2, this list focuses on providing a simplification over real connections. Obtaining such a list requires considerable amounts of iterations and processing, as it essentially works its way through the connection list to "connect the dots". The function *get_master_list* implements this functionality and can be found in its entirety in Listing B.2. The thesis will show excerpts from this function while explaining its functionality throughout this section.

In summary, the function uses the interconnecting module from Chapter 4.4 as the starting point. It takes the bus wires going out of the interconnecting module and traces them to the master- and slave modules on the other end. If there are relay modules in-between, as described in Chapter 4.3.4, it bypasses them and continues the tracing. Once the function has mapped out the connections, the function can deduce which master is connected to which

slave from information extracted from the interconnecting module. Finally, the function returns the deduced master-slave information as a list.

The first step of the function is to determine which modules are interconnecting modules; The function does this by looking for the modules that produce interconnecting-related information. In Listing 5.5, the function iterates through the raw data and adds modules that produce data points related to interconnections to a set collection.

```
1     interconnect_modules = set()
2     for data_point in datapoints:
3         if data_point.info_type == INFO_INTERCONNECT:
4             interconnect_modules.add(data_point.name)
```

Listing 5.5: First step of *get_master_list*: Obtain a list of interconnecting modules.

In Chapter 4.4, there were three types of information produced by interconnecting modules:

- The values the extraction code writes to the ports and wires going out of the interconnecting module.
- Whether the ports intended for a master or a slave.
- Which ports interconnect with each other. I.e., if a module connected to the port, what other modules on other ports can it reach.

The second step is to extract all this information from the raw data and place them in lists that are easier to access and search through; Listing 5.6 shows this. In addition, the function gathers the information from relay modules and store them in a separate list.

The function has a list of values that each is associated with a port on the interconnecting modules. As mentioned earlier, the way the function determines that a master or slave is connected to the interconnecting module is by reading the same value that the interconnect wrote to one of its ports. However, there might be a relay in-between. The function handles the in-between modules by iteration through the raw data list, looking for relays with a value that matches the interconnecting ports' value. If that is the case, the output of the relay replaces the value we have on the interconnecting module; The function repeat this action until there are no more relays in between. Listing 5.7 shows this functionality.

The fourth step is to figure out which master and slave modules are connected to the interconnect ports or the relay modules. Once again, the function iterates through the raw data list and picks out any read operations that are not already classified as interconnecting- or relay modules. If the value read by a module matches either an interconnect or relay module, it is added to the list. This is shown in Listing 5.8.

Finally, the function has obtained all information in order to determine which masters and slaves are connected. It is now just a matter of collecting the information into an organised list returned to the user. Listing 5.9 shows the function iterating through the master-, interconnect-, and slave lists to add the slaves to the masters' list before returned by the function.

Throughout the function, separate classes. The primary function of these classes is to store data in variables and sometimes bring a bit more elaborate add and append functions to those variables. All classes used in the function can be found in Listing B.2.

```
1     interconnections = list()
2     interconnect_ports = set()
3     relays = list()
4     relay_names = set()
5     for data_point in datapoints:
6         # Get value and type
7         if data_point.info_type == INFO_MARK:
8             interconnect_ports.add(Port(data_point.value_1,
              ↪  data_point.value_2))
9         # Get interconnections
10        if data_point.info_type == INFO_INTERCONNECT:
11            interconnections.append(Connection(data_point.value_1,
              ↪  data_point.value_2))
12        # Get relays
13        if data_point.info_type == INFO_RELAY:
14            relays.append(Connection(data_point.value_1,
              ↪  data_point.value_2))
15            relay_names.add(data_point.name)
```

Listing 5.6: Second step of *get_master_list*: Obtain information produced by interconnecting modules and relays.

```
1     interconnect_updated = True
2     while interconnect_updated:
3         interconnect_updated = False
4         # Check every interconnecting port up against the relay list
5         for p in interconnect_ports:
6             for r in relays:
7                 # On hit, change the value, and change any value in the
                  ↪  interconnecting as well
8                 if p.value == r.source:
9                     interconnect_updated = True
10                    old_value = p.value
11                    p.value = r.target
12                    for i in interconnections:
13                        if i.target == old_value:
14                            i.target = r.target
15                        if i.source == old_value:
16                            i.source = r.target
```

Listing 5.7: Third step of *get_master_list*: Group values from relays modules with the interconnecting ports.

```
1     masters = dict()
2     slaves = list()
3     for data_point in datapoints:
4         if data_point.info_type == INFO_READ and data_point.name not in
          ↪  interconnect_modules and data_point.name not in relay_names:
5            for p in interconnect_ports:
6                 if data_point.value_1 == p.value:
7                     if p.port_type == "Slave":
8                         slaves.append(Slave(data_point.name,
                          ↪  data_point.value_1))
9                     elif p.port_type == "Master":
10                        if data_point.name not in masters:
11                            masters[data_point.name] =
                              ↪  Master(data_point.name)
12
                          ↪  masters[data_point.name].master_values.add(data_point.value_1)
```

Listing 5.8: Fourth step of *get_master_list*: Determine which masters and slaves are connected to the busses.

```
1     # Add interconnection values
2     for m in masters:
3         for i in interconnections:
4             if i.source in masters[m].master_values:
5                 masters[m].slave_values.add(i.target)
6             elif i.target in masters[m].master_values:
7                 masters[m].slave_values.add(i.source)
8     # Add slaves
9     for m in masters:
10        for s in slaves:
11            if s.value in masters[m].slave_values:
12                masters[m].slaves.add(s.name)
13
14    return masters
```

Listing 5.9: Fifth step of *get_master_list*: Determine which slave is connected to which master.

5.5 Visualisation

An important part of the processing tool is generating diagrams of the extracted information. It helps giving the information a visual perspective. In order to do so, the thesis has proposed to automatically generate diagrams based on the list produced in Section 5.4. The processing tool uses Graphviz to generate diagrams based on the extracted information. It is an graph visualisation software that uses an abstract language called the DOT langauge to describe its graphs and diagrams. Using these descriptions, it can generate diagrams [26]. In order to convert the processed lists into a DOT description, the tool uses a python package also called Graphviz. The package provides various functions to interact with Graphviz software and produce DOT descriptions that can be read by the Graphviz software [27].

In short, the DOT language uses nodes and edges to make up its diagrams, as shown in Figure 5.8. The Graphviz python-package provides functions to both create nodes and edges, and enhance them with different visual styles, text, and behaviour during generation of diagrams.

Figure 5.8: Example of two nodes connected by an edge.

The tool offers two operations to generate diagrams. One is for generating a diagram of the connection of modules, as described in Section 5.4.2. Listing 5.10 shows the function for generating the connection diagram. Note that the function uses the *os* python package to call on Graphviz software on the computer, as well as viewing the resulting PDF file.

Similarly, the other operation generates a diagram of the master and slave relations, described in Section 5.4.3; The function in Listing 5.11 describe this behaviour. Aside from using different list as a basis, the functionality is pretty similar to the function in Listing 5.10.

```
1  def generate_connection_diagram():
2      if data_loaded:
3          print('Translating buss list into connection list...')
4          connections = pro.get_connections_from_buss_dict(busses_dict)
5          print('Done')
6          print('Creating DOT file...')
7          dot = Digraph() #Use property engine=<layout_name> for different
           ↪  layout engines
8          dot.attr('graph', splines='ortho')
9          dot = add_modules_to_dot(dot, module_set)
10         dot = add_connections_to_dot(dot, connections)
11         print('Done')
12
13         output_file_name = remove_extension_from_name(data_name)
14         dot.save('{}.gv'.format(output_file_name))
15         os.system('/cad/gnu/anaconda/3-2.4.0/bin/fdp -Goverlap=scale -Tpdf
           ↪  {0}.gv -o {0}.pdf'.format(output_file_name))
16         print('Diagram saved as {0}.pdf and
           ↪  {0}.gv'.format(output_file_name))
17
18         if y_n_input("Do you want to view the file? [y/n] >> "):
19             os.system('okular {}.pdf'.format(output_file_name))
20         return
21     else:
22         print('Error: No data have been loaded.')
23     return
```

Listing 5.10: Function for generating a diagram of modules and connection between them.

```
def generate_master_diagram():
    print('Creating DOT file...')
    dot = Digraph()
    dot.attr('graph', splines='ortho')
    dot.attr('node', shape='box')
    dot.attr('edge', penwidth='3')

    for m in masters:
        reduced_master_name = reduce_hierarchical_path_name(masters[m].name)
        if enable_reduced_names:
            dot.node(masters[m].name, label=reduced_master_name)
        else:
            dot.node(masters[m].name, label=masters[m].name)
        #dot.node(masters[m].name)
        for s in masters[m].slaves:
            reduced_name = reduce_hierarchical_path_name(s)
            if enable_reduced_names:
                dot.node(s, label=reduced_name)
            else:
                dot.node(s, label=s)
            #dot.node(s)
            dot.edge(masters[m].name, s)

    output_file_name = remove_extension_from_name(data_name)
    dot.save('{}.gv'.format(output_file_name))
    print('Done')
```

Listing 5.11: Function for generating a diagram of masters, slaves, and their relationships.

Chapter 6

Case Study - System from Nordic Semiconductor

This case will consider the subsystem provided by Nordic Semiconductor. This is a system encapsulating an Arm Cortex CPU and a bus system to connect the CPU to other modules and peripherals. The thesis will consider how the solution is implemented into the system, and what results can be obtained from the implementation. For the rest of this section, we will refer to the system provided by Nordic Semiconductor as the subsystem. Due to confidentiality, internal signal names and structures have been changed for inclusion in this thesis.

The nRF5340 Bluetooth SoC utilises the subsystem in question. Since the nRF5340 have publicly available documentation, the thesis will use it to verify the extracted information [3]. Figur 6.1 shows the block diagram of the application core for the nRF5340. The diagram shows that the chip uses an AHB Multi-Layer bus from the AMBA standard [14]. In addition, it uses both AHB and APB busses from the AMBA standard [13], but the main focus of this case is the AHB Multi-Layer bus. Chapter 2.2 explains all three protocols in detail.

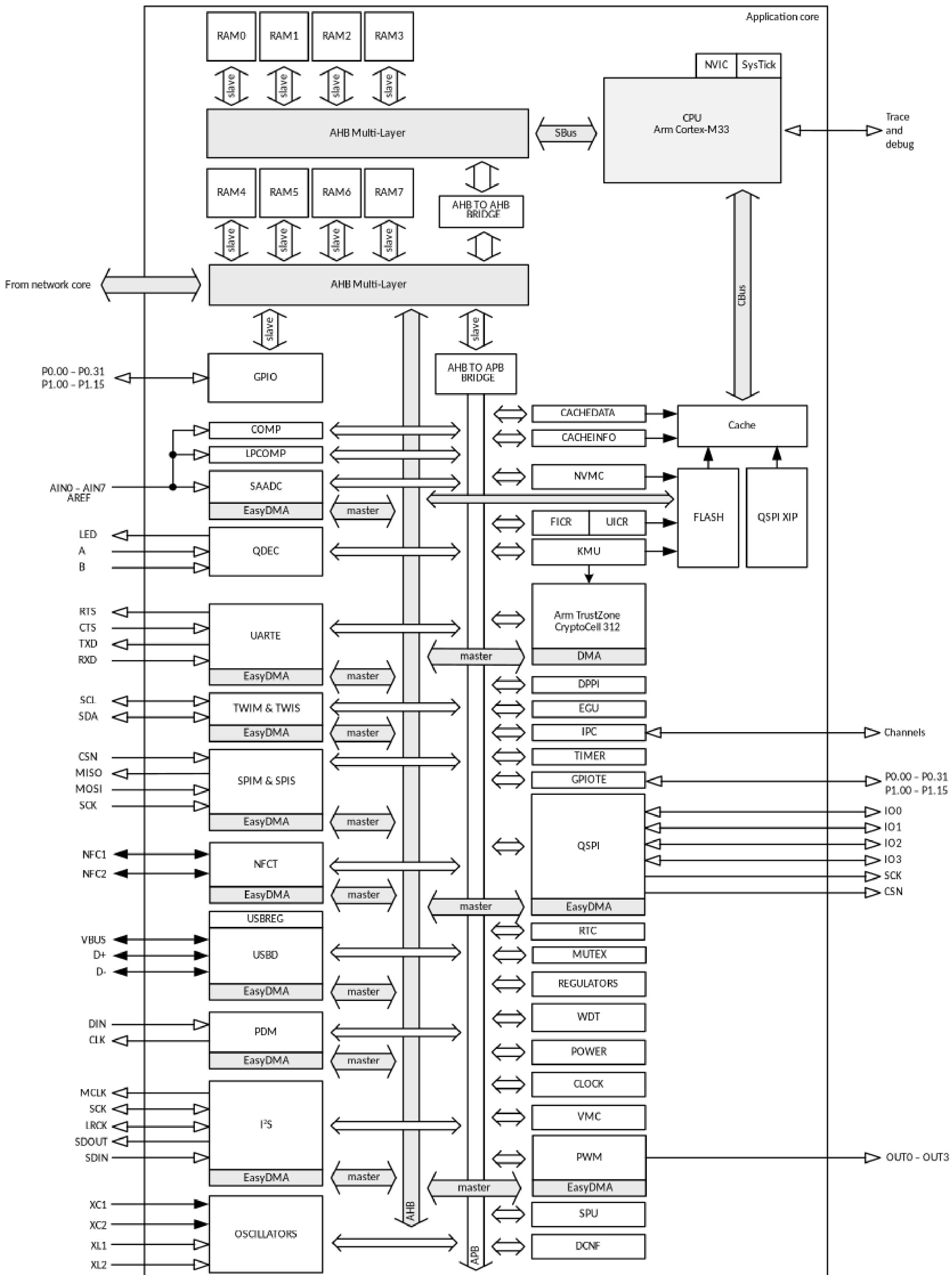

Figure 6.1: Block diagram of the application core of the nRF5340 Bluetooth SoC, taken from [3]

Implementation

The modules used for busses are implemented with reuse in mind. Therefore, one can apply the extraction code to the IP, and all instances of the IP will use it. First of all, the source file for the extraction solution macros, found in Appendix A, is included in the project files for the subsystem. The inclusion is done through config files, referencing the extraction file and making the proper definitions to enable extraction.

Starting with the AHB Multi-Layer bus itself, the thesis implement the extraction code intended for interconnecting modules, as shown in Chapter 4.4, to the top module of the bus files. One would want it to propagate a signal to all modules connected, mark each port as master or slave, and define which master can reach which slave. Listing 6.1 shows the snippet of code containing the extraction code.

```
1  initial begin
2      fork
3      `INTERCONNECT_BUSSES(CONNECTIONS, ahbHWDataSlave, ahbHRDataMaster, )
4      `FORCE_AND_RELEASE_BUS_ARRAY(ahbHRDataMaster, MASTERS,
       ↪  AHBDATAWIDTH,"ahbHRDataMaster", )
5      `FORCE_AND_RELEASE_BUS_ARRAY(ahbHWDataSlave, SLAVES,
       ↪  AHBDATAWIDTH,"ahbHWDataSlave", )
6      `MARK_BUSSES_AS_MASTER(ahbHRDataMaster, )
7      `MARK_BUSSES_AS_SLAVE(ahbHWDataSlave, )
8      join
9  end
```

Listing 6.1: The extraction code applied to *AHB_multi_layer_top.sv*

The processor is one of the modules connected to the bus and should thus read it. Using the macros presented in Chapter 4.3.3, the thesis implements the extraction code in the processor top module as shown in Listing 6.2.

```
1  initial begin
2      fork
3      `READ_BUS_ARRAY(ahbHRDataMaster, "ahbHRDataMaster", )
4      join
5  end
```

Listing 6.2: The extraction code applied to *cpu_top.sv*

The other modules and peripherals (i.e., masters and slaves) shown in Figure 6.1 is not part of the subsystem. Instead, the subsystem has ports that the external modules and busses can use. However, this means one does not have a module one can enhance with extraction code. Instead, the thesis consider each port as a module, applying the extraction code from Chapter 4.3.3. The macros allow custom names so that one can mark the ports as their intended modules and busses. It is necessary since all the ports reside in the same module, and the default name is the module path. The extraction code snippet for the other masters and slaves is shown in Listing 6.3

```
1  initial begin
2      fork
3      `READ_BUS_ARRAY(bridge_1_ahbHWData, "bridge_1_ahbHWData",
       ↪  "EXTRAM_Slave")
4      `READ_BUS_ARRAY(bridge_2_ahbHRData, "bridge_2_ahbHRData",
       ↪  "EXTRAM_Master")
5      `READ_BUS_ARRAY(bridge_3_ahbHWData, "bridge_3_ahbHWData",
       ↪  "EXTPERI_Slave")
6      `READ_BUS_ARRAY(bridge_4_ahbHRData, "bridge_4_ahbHRData",
       ↪  "EXTPERI_Master")
7      `READ_BUS_ARRAY(interface_1.ahbHRData, "interface_1.ahbHRData",
       ↪  "DMA_Master")
8      `READ_BUS_ARRAY(interface_2.ahbHRData, "interface_2.ahbHRData",
       ↪  "DMAEXT_Master")
9      `READ_BUS_ARRAY(interface_3.ahbHWData, "interface_3.ahbHWData",
       ↪  "SRAM_Slave")
10     `READ_BUS_ARRAY(interface_4.ahbHWData, "interface_4.ahbHWData",
       ↪  "PERIAHB_Slave")
11     `READ_BUS_ARRAY(interface_5.ahbHRData, "interface_5.ahbHRData",
       ↪  "RAM_Master")
12     `READ_BUS_ARRAY(interface_6.apbPWData, "interface_6.apbPWData",
       ↪  "PERIAPB_Slave")
13     join
14 end
```

Listing 6.3: The extraction code applied to *subsystem_top.sv*

There are several bridges in the subsystem. As shown in Chapter 4.3.4, one can relay the signals across the bridges. Listing 6.4 shows a code snippet from one of the AHB bridges. While there are several bridges in the subsystem, the implementation is similar to the one in Listing 6.4, only with different names.

```
1  initial begin
2      fork
3      `FORCE_AND_RELEASE_BUS(HRDATAS, "HRDATAS", )
4      `FORCE_AND_RELEASE_BUS(HWDATAM, "HWDATAM", )
5      `READ_BUS(HWDATAS, "HWDATAS", )
6      `READ_BUS(HRDATAM, "HRDATAM", )
7      `RELAY(HWDATAS, HWDATAM, )
8      `RELAY(HRDATAM, HRDATAS, )
9      join
10 end
```

Listing 6.4: The extraction code applied to *AHB_bridge_top.sv*

Notice that signals are relayed both on the master side and the slave side of the bridge. Since the bridge is a module intended for reuse in other designs and instances, it makes sense to implement the extraction code on both sides; One does not know which side of the bridge a signal might come from, so it is better to cover both.

Results

Figure 6.2 shows the diagram of connections between masters and slaves, processed from the extracted data of the subsystem. Note that the arrowhead indicates the slave, and the master is at the other end of the line.

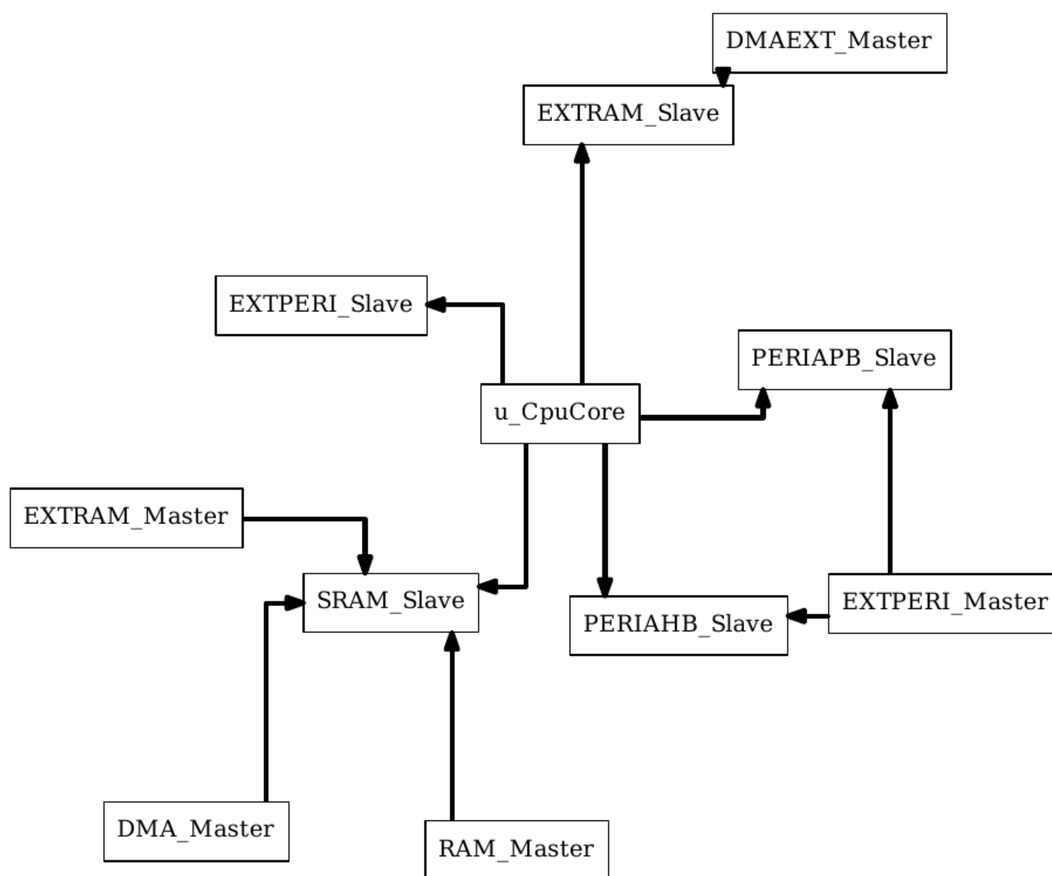

Figure 6.2: The resulting block diagram showing connections between masters and slaves for the Nordic Semiconductor subsystem.

There are promising similarities to the original block diagram from Figure 6.1. As expected, the CPU is the main master module, connecting to most other slaves in the system. Masters that connects from peripherals can also access peripherals on the AHB and APB busses bridged to the multi-layer. The same can be said for DMA masters. Note that the GPIO module is outside of the system, and would be accessed through the PERIAHB_Slave port.

It is relatively simple to implement the extraction code in the subsystem. In addition, a lot of the extraction code is placed in reusable modules with several instances in the subsystem, reducing the amount of work necessary. However, it is difficult to make a proper estimate of how long the implementation took, as the subsystem was used extensively to develop the extraction code.

Chapter 7

Future Works

Throughout the implementation of the solution presented by this thesis, it has become apparent that it has the potential for both changes and additional features. This chapter aims to provide an overview of these potential additions to the solution that could be done in the future when more time and resources are available.

Chapter 3.1 detailed many different types of information that would be interesting to extract from RTL. While some of them are present in the implemented solution, others are not. For instance, the implementation does not include the design's address maps, nor is there much information on the bus wires themselves. Both are a valid proposition for additional information to extract in the future. Of course, there might be even more types of information that would be interesting, depending on the bus system. These have to be worked out in future implementations.

When it comes to the processing tool from Chapter 5 there is much potential. It could be helpful to have more control over the data residing in the tool, such as selecting a subset of the data and editing parameters and values. Introducing new operations is not limited to handling the current information in new ways but should also include ways of handling new types of extracted information. It could be beneficial to expand and change how the information is processed to be more efficient and process new types of information imported into the tool. The tool itself could be more user-friendly as well, perhaps by including a graphical user interface. It could also be considered to use a different graphics generator or even write one from scratch to control how the diagrams and images are generated.

Another helpful addition to the solution would be to consider more cases for the case study. Ideally, the cases should have different bus systems from different sources. More cases would evaluate the quality of the solution and how universal the solution is. It would also help uncover flaws in the solution that are not apparent at first.

An interesting case is that of generated bus systems. These are systems that have been automatically generated from a set of parameters provided by the user. Depending on the tool, the generated source code might not be available to the user; This poses some difficulties for the current solution, as it depends on implementing extraction code into the source code. Future iterations of the solution might change or include extracting information from both automatically generated busses and busses that do not have readily available source code.

Chapter 8

Conclusion

This thesis addressed the issue of obtaining a good grasp of complex bus structures typically found in a more complex system on chips. The goal was to find a solution that could do so automatically and was simple and easy to implement and use. The solution proposed by this thesis was to implement extraction code into the RLT design and exploit the simulation engine to extract the information during simulation. In addition, the thesis conceived a separate tool to process and handle the information after extraction, importing the information from HDL into a more high-level language environment.

The results from the case study with the Nordic Semiconductor systems are promising. The solution managed to extract and process the information on the bus system that corresponded to the documentation of the said system. Considering that the bus system used in the Nordic case is reasonably complex shows that the solution can manage complex bus systems. However, it will be necessary with more and varied cases to evaluate the quality of the solution. One interesting case is that of automatically generated bus systems, where the source code is not necessarily readily available.

In addition to expanding the case study to include more cases, the solution also has potential for expansion. The extraction code can be expanded to include more types of information it can extract, ways of dealing with automatically generated bus systems, and how it handles extraction. The processing tool can be expanded with more additional features as well. In conclusion, the solution shows promise, both in how it can extract information from complex RTL design and the potential expansion of functionality.

Appendix A

Extraction Source Code

```
1  //=================================================================
2  // Name          : Bus Structure Extraction Enhancement Code
3  // Created       : Kevin Vinding
4  // Description   : Macros used for extracting meta-information on
5  //                 bus structures.
6  //=================================================================
7  `ifndef BUS_EXTRACTION_SV
8  `define BUS_EXTRACTION_SV
9
10 // Types of information:
11 `define INFO_READ "ReadBus"
12 `define INFO_FORCE "ForceValue"
13 `define INFO_INTERCONNECT "Interconnect"
14 `define INFO_MARK "MarkBus"
15 `define INFO_RELAY "Relay"
16
17
18 integer forceValue = 10;
19
20 function integer getNewForceValue();
21         return forceValue++;
22 endfunction : getNewForceValue
23
24
25 // -- Writes values to log
26 `define WRITE_VALUES_TO_LOG(MODULE_NAME, TYPE_OF_INFORMATION, VALUE_1,
   ↪  VALUE_2 = 0) \
27 `ifdef ENABLE_BUS_EXTRACTION \
28         $display("#bus#%s#%s#%h#%h#", MODULE_NAME, TYPE_OF_INFORMATION,
           ↪  VALUE_1, VALUE_2); \
29 `endif
30
31 // -- Writes text (with associated value) to log
32 `define WRITE_VALUE_AND_TEXT_TO_LOG(MODULE_NAME, TYPE_OF_INFORMATION, VALUE,
   ↪  TEXT) \
```

```
33 `ifdef ENABLE_BUS_EXTRACTION \
34         $display("#bus#%s#%s#%h#%s#", MODULE_NAME, TYPE_OF_INFORMATION,
        ↪  VALUE, TEXT); \
35 `endif
36
37 // -- Forces the value VALUE on DATA_BUS. Parameter MODULE_NAME defaults to
   ↪  the hierarchical path of the caller.
38 `define FORCE_VALUE_ON_BUS(DATA_BUS, VALUE, MODULE_NAME=$sformatf("%m")) \
39 `ifdef ENABLE_BUS_EXTRACTION \
40         force DATA_BUS = VALUE; \
41         `WRITE_VALUES_TO_LOG(MODULE_NAME, `INFO_FORCE, VALUE, ) \
42 `endif
43
44 // -- Forces a random value on DATA_BUS. Parameter MODULE_NAME defaults to
   ↪  the hierarchical path of the caller.
45 `define FORCE_RANDOM_VALUE_ON_BUS(DATA_BUS, MODULE_NAME=$sformatf("%m")) \
46 `ifdef ENABLE_BUS_EXTRACTION \
47         begin \
48                 static int rand_value = $random(); \
49                 `FORCE_VALUE_ON_BUS(DATA_BUS, rand_value, MODULE_NAME) \
50         end \
51 `endif
52
53 // -- Releases DATA_BUS after a value has been forced by e.g., using
   ↪  FORCE_RANDOM_VALUE_ON_BUS(...).
54 `define RELEASE_BUS(DATA_BUS) \
55 `ifdef ENABLE_BUS_EXTRACTION \
56         release DATA_BUS; \
57 `endif
58
59 // -- Forces a random value on DATA_BUS, waits 2 delta-cycles, then releases
   ↪  DATA_BUS.
60 //    Parameter MODULE_NAME defaults to the hierarchical path of the caller.
61 `define FORCE_AND_RELEASE_BUS(DATA_BUS, BUS_NAME="",
   ↪  MODULE_NAME=$sformatf("%m")) \
62 `ifdef ENABLE_BUS_EXTRACTION \
63         begin \
64                 static int value_to_force = getNewForceValue(); \
65                 `FORCE_VALUE_ON_BUS(DATA_BUS, value_to_force, MODULE_NAME) \
66                 #0; \
67                 `WRITE_VALUE_AND_TEXT_TO_LOG(MODULE_NAME, `INFO_READ,
                ↪  DATA_BUS, BUS_NAME) \
68                 #0; \
69                 `RELEASE_BUS(DATA_BUS) \
70         end \
71 `endif
72
73 // -- Forces random values on all busses in DATA_BUS_ARRAY, and releases
   ↪  them after 2 delta-cycles.
```

```
74  // -- Note: The data-bus array is expected to be on the form:
    ↪  [ARRAY_SIZE-1:0][BUS_SIZE-1:0] DATA_BUS_ARRAY
75  `define FORCE_AND_RELEASE_BUS_ARRAY(DATA_BUS_ARRAY, ARRAY_SIZE, BUS_SIZE,
    ↪  BUS_NAME="", MODULE_NAME=$sformatf("%m")) \
76  `ifdef ENABLE_BUS_EXTRACTION \
77          begin \
78                  logic [ARRAY_SIZE-1:0][BUS_SIZE-1:0] ValueForBusArray; \
79                  foreach (ValueForBusArray[i]) begin \
80                          ValueForBusArray[i] = getNewForceValue(); \
81                  end \
82                  force DATA_BUS_ARRAY = ValueForBusArray; \
83                  foreach (ValueForBusArray[i]) begin \
84                          `WRITE_VALUES_TO_LOG(MODULE_NAME, `INFO_FORCE,
                            ↪  ValueForBusArray[i], ) \
85                  end \
86                  #0; \
87                  foreach (DATA_BUS_ARRAY[i]) begin \
88                          `WRITE_VALUE_AND_TEXT_TO_LOG(MODULE_NAME,
                            ↪  `INFO_READ, DATA_BUS_ARRAY[i],
                            ↪  $sformatf("%s[%0d]", BUS_NAME, i)) \
89                  end \
90                  #0; \
91                  `RELEASE_BUS(DATA_BUS_ARRAY) \
92          end \
93  `endif
94
95  // -- NEw read-bus macro. Change name to READ_BUS when appropiate
96  `define READ_BUS(DATA_BUS, BUS_NAME="", MODULE_NAME=$sformatf("%m")) \
97  `ifdef ENABLE_BUS_EXTRACTION \
98          begin \
99                  #0; \
100                 `WRITE_VALUE_AND_TEXT_TO_LOG(MODULE_NAME, `INFO_READ,
                    ↪  DATA_BUS, BUS_NAME) \
101         end \
102 `endif
103
104 // -- New read-bus-array macro. Change name to READ_BUS_ARRAY when
    ↪  appropiate
105 `define READ_BUS_ARRAY(DATA_BUSSES, BUS_NAME="",
    ↪  MODULE_NAME=$sformatf("%m")) \
106 `ifdef ENABLE_BUS_EXTRACTION \
107         begin \
108                 #0; \
109                 foreach (DATA_BUSSES[i]) begin \
110                         `WRITE_VALUE_AND_TEXT_TO_LOG(MODULE_NAME,
                            ↪  `INFO_READ, DATA_BUSSES[i], $sformatf("%s[%0d]",
                            ↪  BUS_NAME, i)) \
111                 end \
112         end \
```

```
113 `endif
114
115 // -- Finish the simulation early.
116 `define FINISH_SIMULATION_EARLY() \
117 `ifdef ENABLE_BUS_EXTRACTION \
118         #1; \
119         $display("Simulation ends early due to bus structure extraction.");
         ↪  \
120         $finish(); \
121 `endif
122
123 // -- Uses a connection matrix to define which bus in FIRST_BUS_ARRAY array
    ↪  are connected to which
124 //    bus in SECOND_BUS_ARRAY array (after a delta-cycle).
125 // -- Note: CONNECTION_MATRIX first dimension must equal that of the
    ↪  FIRST_BUS_ARRAY. Similarly,
126 //    the second dimension must equal SECOND_BUS_ARRAY.
127 `define INTERCONNECT_BUSSES(CONNECTION_MATRIX, FIRST_BUS_ARRAY,
    ↪  SECOND_BUS_ARRAY, MODULE_NAME=$sformatf("%m")) \
128 `ifdef ENABLE_BUS_EXTRACTION \
129         begin \
130                 #0; \
131                 foreach (FIRST_BUS_ARRAY[i]) begin \
132                         foreach (SECOND_BUS_ARRAY[j]) begin \
133                                 if (CONNECTION_MATRIX[i][j] != 0) begin \
134                                         `WRITE_VALUES_TO_LOG(MODULE_NAME,
                                         ↪  `INFO_INTERCONNECT,
                                         ↪  FIRST_BUS_ARRAY[i],
                                         ↪  SECOND_BUS_ARRAY[j]) \
135                                 end \
136                         end \
137                 end \
138         end \
139 `endif
140
141
142 // -- Defines that DATA_BUS_OUT is an relay-value of DATA_BUS_IN
143 `define RELAY(DATA_BUS_IN, DATA_BUS_OUT, MODULE_NAME=$sformatf("%m")) \
144 `ifdef ENABLE_BUS_EXTRACTION \
145         begin \
146                 #0; \
147                 `WRITE_VALUES_TO_LOG(MODULE_NAME, `INFO_RELAY, DATA_BUS_IN,
                 ↪  DATA_BUS_OUT) \
148         end \
149 `endif
150
151 // -- Defines that DATA_BUSSES_OUT is relay-values of DATA_BUSSES_IN
152 //          Note: DATA_BUSSES_IN must be same size as DATA_BUSSES_OUT
```

```
153 `define RELAY_ARRAY(DATA_BUSSES_IN, DATA_BUSSES_OUT,
    ↪  MODULE_NAME=$sformatf("%m")) \
154 `ifdef ENABLE_BUS_EXTRACTION \
155         begin \
156                 #0; \
157                 foreach (DATA_BUSSES_IN[i]) begin \
158                         `WRITE_VALUES_TO_LOG(MODULE_NAME, `INFO_RELAY,
                            ↪  DATA_BUSSES_IN[i], DATA_BUSSES_OUT[i]) \
159                 end \
160         end \
161 `endif
162 
163 // -- Mark the DATA_BUS as Master
164 `define MARK_BUS_AS_MASTER(DATA_BUS, MODULE_NAME=$sformatf("%m")) \
165 `ifdef ENABLE_BUS_EXTRACTION \
166         begin \
167                 #0; \
168                 `WRITE_VALUE_AND_TEXT_TO_LOG(MODULE_NAME, `INFO_MARK,
                    ↪  DATA_BUS, "Master") \
169         end \
170 `endif
171 
172 // -- Mark the DATA_BUS as Slave
173 `define MARK_BUS_AS_SLAVE(DATA_BUS, MODULE_NAME=$sformatf("%m")) \
174 `ifdef ENABLE_BUS_EXTRACTION \
175         begin \
176                 #0; \
177                 `WRITE_VALUE_AND_TEXT_TO_LOG(MODULE_NAME, `INFO_MARK,
                    ↪  DATA_BUS, "Slave") \
178         end \
179 `endif
180 
181 // -- Mark several DATA_BUSSES as Master
182 `define MARK_BUSSES_AS_MASTER(DATA_BUSSES, MODULE_NAME=$sformatf("%m")) \
183 `ifdef ENABLE_BUS_EXTRACTION \
184         begin \
185                 #0; \
186                 foreach (DATA_BUSSES[i]) begin \
187                         `WRITE_VALUE_AND_TEXT_TO_LOG(MODULE_NAME,
                            ↪  `INFO_MARK, DATA_BUSSES[i], "Master") \
188                 end \
189         end \
190 `endif
191 
192 // -- Mark several DATA_BUSSES as Slave
193 `define MARK_BUSSES_AS_SLAVE(DATA_BUSSES, MODULE_NAME=$sformatf("%m")) \
194 `ifdef ENABLE_BUS_EXTRACTION \
195         begin \
196                 #0; \
```

```
197                 foreach (DATA_BUSSES[i]) begin \
198                         `WRITE_VALUE_AND_TEXT_TO_LOG(MODULE_NAME,
                        ↪  `INFO_MARK, DATA_BUSSES[i], "Slave") \
199                 end \
200         end \
201 `endif
202
203 `endif
```

Listing A.1: bus_info_extract.sv

Appendix B

Processing Tool Source Code

```
1  ###############################################################################
2  # Name:         Bus Tool (main file)
3  # Author:       Kevin Vinding
4  # Description:  Main file for analysing and visualising the meta-information
   ↪  obtained from extraction code.
5  ###############################################################################
6  import sys
7  import getopt
8  import os
9  import bus_info_parser as bip
10 import bus_info_processor as pro
11 from graphviz import Digraph
12
13 CMD_PDF = '/cad/gnu/anaconda/3-2.4.0/bin/dot -Tpdf bus_structure.gv -o
   ↪  graph.pdf && okular graph.pdf'
14
15 data_loaded = False
16 enable_reduced_names = True
17 enable_debug_mode = False
18 data_name = 'None'
19 module_set = set()
20 busses_dict = dict()
21 masters = dict()
22
23
24 def add_modules_to_dot(dot, list_of_modules):
25     dot.attr('node', shape='box')
26     for module in list_of_modules:
27         reduced_name = reduce_hierarchical_path_name(module)
28         if enable_reduced_names:
29             dot.node(module, label=reduced_name)
30         else:
31             dot.node(module, label=module)
32     return dot
33
```

```
34
35 def add_connections_to_dot(dot, list_of_connections):
36     for con in list_of_connections:
37         if con.source != "" and con.target != "":
38             if con.bidirectional:
39                 dot.attr('edge', dir='none', style='solid', penwidth='5') #,
                    ↪  penwidth = 2, style='solid')
40             else:
41                 dot.attr('edge', dir='forward', style='solid', penwidth='1')
                    ↪  #, penwidth=1, style='dashed')
42             dot.edge(con.source, con.target)
43     return dot
44
45
46 def reduce_hierarchical_path_name(hierarchical_name):
47     separate_hier_name = hierarchical_name.split('.')
48     return separate_hier_name[len(separate_hier_name)-1]
49
50
51 # Ask a yes/no-question, then return answer as bool
52 def y_n_input(question):
53     while True:
54         answer = input(question).lower()
55         if answer == 'y' or answer == 'yes':
56             return True
57         elif answer == 'n' or answer == 'no':
58             return False
59         else:
60             print('Invalid answer. Use "y" or "n".')
61     return
62
63
64 # Remove extension from file name (e.g., .txt or .pdf)
65 def remove_extension_from_name(name):
66     return name.split('.')[0]
67
68
69 # Generates a diagram of the (imported) data as a .pdf file
70 def generate_connection_diagram():
71     if data_loaded:
72         print('Translating buss list into connection list...')
73         connections = pro.get_connections_from_buss_dict(busses_dict)
74         print('Done')
75         print('Creating DOT file...')
76         dot = Digraph() #Use property engine=<layout_name> for different
            ↪  layout engines
77         dot.attr('graph', splines='ortho')
78         dot = add_modules_to_dot(dot, module_set)
79         dot = add_connections_to_dot(dot, connections)
```

```
80          print('Done')
81
82          output_file_name = remove_extension_from_name(data_name)
83          dot.save('{}.gv'.format(output_file_name))
84          os.system('/cad/gnu/anaconda/3-2.4.0/bin/fdp -Goverlap=scale -Tpdf
            ↪  {0}.gv -o {0}.pdf'.format(output_file_name))
85          print('Diagram saved as {0}.pdf and
            ↪  {0}.gv'.format(output_file_name))
86
87          if y_n_input("Do you want to view the file? [y/n] >> "):
88              os.system('okular {}.pdf'.format(output_file_name))
89          return
90      else:
91          print('Error: No data have been loaded.')
92      return
93
94
95  # Generate a diagram detailing masters and slaves connected to the masters
96  def generate_master_diagram():
97      print('Creating DOT file...')
98      dot = Digraph()
99      dot.attr('graph', splines='ortho')
100     dot.attr('node', shape='box')
101     dot.attr('edge', penwidth='3')
102
103     for m in masters:
104         reduced_master_name = reduce_hierarchical_path_name(masters[m].name)
105         if enable_reduced_names:
106             dot.node(masters[m].name, label=reduced_master_name)
107         else:
108             dot.node(masters[m].name, label=masters[m].name)
109         #dot.node(masters[m].name)
110         for s in masters[m].slaves:
111             reduced_name = reduce_hierarchical_path_name(s)
112             if enable_reduced_names:
113                 dot.node(s, label=reduced_name)
114             else:
115                 dot.node(s, label=s)
116             #dot.node(s)
117             dot.edge(masters[m].name, s)
118
119     output_file_name = remove_extension_from_name(data_name)
120     dot.save('{}.gv'.format(output_file_name))
121     print('Done')
122
123     print('Creating PDF file...')
124     os.system('/cad/gnu/anaconda/3-2.4.0/bin/fdp -Goverlap=scale -Tpdf
        ↪  {0}.gv -o {0}.pdf'.format(output_file_name))
125     print('Done')
```

```
126
127     if y_n_input("Do you want to view the file? [y/n] >> "):
128         os.system('okular {}.pdf'.format(output_file_name))
129     return
130
131
132 # Toggles whether the name reduction is applied or not.
133 def toggle_name_reduction():
134     global enable_reduced_names
135     toggle = False
136     if enable_reduced_names:
137         print('Name reduction is currently ENABLED')
138         toggle = y_n_input('Disable reduction of names? [y/n] >> ')
139     else:
140         print('Name reduction is currently DISABLED')
141         toggle = y_n_input('Enable reduction of names? [y/n] >> ')
142
143     if toggle:
144         enable_reduced_names = not enable_reduced_names
145
146
147 # Checks the command line agruments and takes correspondingly actions
148 def check_command_line_args():
149     global enable_reduced_names
150     global enable_debug_mode
151     import_file_path = ""
152     import_at_init = False
153
154     try:
155         opts, args = getopt.getopt(sys.argv[1:],"ni:d")
156     except:
157         print('Error: Invalid arguments.')
158         return
159
160     for opt, arg in opts:
161         if opt == '-i':
162             import_at_init = True
163             import_file_path = arg
164         elif opt == '-n':
165             enable_reduced_names = False
166         elif opt == '-d':
167             enable_debug_mode = True
168
169     if import_at_init:
170         import_file(import_file_path)
171     return
172
173
174 # Print the set of modules obtained from data
```

```
175 def list_modules():
176     if data_loaded:
177
        ↪  print('--------------------------MODULES-------------------------------')
178         for mod in module_set:
179             if enable_reduced_names:
180                 print(reduce_hierarchical_path_name(mod))
181             else:
182                 print(mod)
183
        ↪  print('----------------------------------------------------------------')
184     else:
185         print('Error: No data have been loaded.')
186     return
187
188
189 #Print the dictionary of busses obtained from data
190 def list_connections():
191     if data_loaded:
192
        ↪  print('--------------------------CONNECTIONS---------------------------')
193         for bus in busses_dict:
194             nb = pro.Bus(busses_dict[bus].get_value())
195             if enable_reduced_names:
196                 for r in busses_dict[bus].read_sources:
197                     nb.add_read_source(reduce_hierarchical_path_name(r))
198                 for n in busses_dict[bus].names:
199                     nb.add_name(reduce_hierarchical_path_name(n))
200
                ↪  nb.set_write_source(reduce_hierarchical_path_name(busses_dict[bus]
                ↪  .get_write_source()))
201             else:
202                 nb = busses_dict[bus]
203
204             if enable_debug_mode:
205                 print(nb)
206                 print("")
207             else:
208                 print('Name(s) of Connection: {}\n\tWrite Source: {}\n\tRead
                ↪  Source(s): {}\n'.format(
209                     nb.names, nb.get_write_source(), nb.read_sources
210                 ))
211
        ↪  print('----------------------------------------------------------------')
212     else:
213         print('Error: No data have been loaded.')
214     return
215
```

```
216 #Print the dictionary of master modules, and what slaves is connected to
    ↪  them.
217 def list_masters():
218     if data_loaded:
219
            ↪  print('--------------------------MASTERS-------------------------------')
220         for m in masters.values():
221             new_m = m
222             if enable_reduced_names:
223                 new_m.name = reduce_hierarchical_path_name(new_m.name)
224                 new_slaves = set()
225                 for s in new_m.slaves:
226                     new_slaves.add(reduce_hierarchical_path_name(s))
227                 new_m.slaves = new_slaves
228
229             if enable_debug_mode:
230                 print(new_m)
231             else:
232                 #print(m.slaves)
233                 print('Master: {}\nSlaves: {}\n'.format(new_m.name,
                    ↪  new_m.slaves))
234             #print(masters[m])
235
            ↪  print('----------------------------------------------------------------')
236     else:
237         print('Error: No data have been loaded.')
238     return
239
240 # Imports a specified textfile
241 def import_file(filename):
242     global data_name
243     global data_loaded
244     global module_set
245     global busses_dict
246     global masters
247
248     print('Parsing {}...'.format(filename))
249     data_points = bip.parse_text_file(filename)
250     if data_points == -1:
251         print('Import failed')
252         return
253     print('Done')
254
255     print('Obtaining list of modules...')
256     module_set = pro.get_modules_from_data_points(data_points)
257     print('Done')
258
259     print('Obtaining list of busses...')
260     busses_dict = pro.get_busses_from_data_points(data_points)
```

```
261     print('Done')
262 
263     print('Obtaining list of masters and their slaves...')
264     masters = pro.get_master_list(data_points)
265     print('Done')
266 
267     data_name = filename
268     data_loaded = True
269     print('Processing finished. Imported {}
        ↪  modules.'.format(len(module_set)))
270     return
271 
272 
273 # Prints a list of all available commands to the user
274 def get_commands():
275 
        ↪  print("""-------------------------COMMANDS-----------------------------
276     import <filename.txt>\tImport meta-information from specified textfile.
277     listmod\t\t\tDisplay a list of all modules from extracted data.
278     listcon\t\t\tDisplay a list of all connection between modules.
279     listmas\t\t\tDisplay a list of all masters, and slaves connected to the
        ↪  master.
280     genmas\t\t\tGenerate a pdf image of the relationship between masters and
        ↪  slave.
281     gencon\t\t\tGenerate a diagram of the connection between modules in the
        ↪  system.
282     togname\t\t\tToggle whether name reduction is enabled or not.
283     help\t\t\tShows this list of available commands.
284     quit\t\t\tQuits the tool. (Can also use 'exit'.)
285 -------------------------ARGUMENTS----------------------------
286     -i <filename.txt>\t\tImport file at initialization of the tool
287     -n\t\t\t\tTurn off name reduction at startup
288     -d\t\t\t\tStart tool in debug mode
289 ----------------------------------------------------------------""")
290     return
291 
292 
293 # main loop
294 print('Processing tool for extracted bus information')
295 print('---------------------------------------------')
296 check_command_line_args()
297 print('Data loaded: {}'.format(data_name))
298 print('Reduced Names: {}'.format(enable_reduced_names))
299 if enable_debug_mode:
300     print('DEBUG MODE')
301 print('---------------------------------------------')
302 
303 while True:
304     command = input('>> ').lower()
```

```
305     if command == 'help':
306         get_commands()
307     elif command == 'quit' or command == 'exit':
308         sys.exit(0)
309     elif command == 'listmod':
310         list_modules()
311     elif command == 'listcon':
312         list_connections()
313     elif command == 'listmas':
314         list_masters()
315     elif command == 'genmas':
316         generate_master_diagram()
317     elif command == 'gencon':
318         generate_connection_diagram()
319     elif command == 'togname':
320         toggle_name_reduction()
321     elif command.split(' ')[0] == 'import':
322         import_file(command.split(' ')[1])
323     else:
324         print('Invalid command. Type "help" for list of available
             ↪  commands.')
```

Listing B.1: bus_tool.py

```
1  ############################################################################
2  # Name:         Bus Info Processor
3  # Author:       Kevin Vinding
4  # Description:  Processes and maps out bus information based on datapoints
   ↪  obtained from bus_info_parser.py
5  ############################################################################
6  import datapoint as dp
7
8  INFO_READ = "ReadBus"
9  INFO_FORCE = "ForceValue"
10 INFO_INTERCONNECT = "Interconnect"
11 INFO_MARK = "MarkBus"
12 INFO_RELAY = "Relay"
13 NAME_DELIMITER = '.'
14
15 # Get a set of modules from a list of data points
16 def get_modules_from_data_points(list_data_points):
17     modules = set()
18     for data_point in list_data_points:
19         modules.add(data_point.name)
20     return modules
21
22
23 # Reduce the name by only include the last region of the name
24 def reduce_name(name):
25     split_name = name.split(NAME_DELIMITER)
26     return split_name[len(split_name)-1]
27
28
29 # Verify that a value is an int and not the value 0
30 def verify_value(string_value):
31     try:
32         value_as_int = int(string_value, 16)
33         if value_as_int == 0:
34             return False
35         else:
36             return True
37     except:
38         return False
39
40
41 # Get a dictionary of busses bewteen modules from a list of data points,
   ↪  with the written value as key
42 def get_busses_from_data_points(list_data_points):
43     busses = dict()
44     # Iterate through all data points that are read- or write operations.
45     for data_point in list_data_points:
```

```
46          if data_point.info_type == INFO_READ or data_point.info_type ==
             ↪  INFO_FORCE:
47              if verify_value(data_point.value_1):
48
49                  # Create new entry if not in dict
50                  if data_point.value_1 not in busses:
51                      busses[data_point.value_1] = Bus(data_point.value_1)
52
53                  # Add info to entry
54                  if data_point.info_type == INFO_READ:
55
                         ↪  busses[data_point.value_1].add_read_source(data_point.name)
56                      busses[data_point.value_1].add_name(data_point.value_2)
57                  elif data_point.info_type == INFO_FORCE:
58
                         ↪  busses[data_point.value_1].set_write_source(data_point.name)
59      return busses
60
61
62  # Update an existing connecton list with a new connection
63  def update_connection_list(connection_list, new_connection):
64      for connection in connection_list:
65          # If it's inside list, discard the new connection
66          if connection == new_connection:
67              return connection_list
68          # If it's the inverted instace, update it to be bidirectional
69          if connection.get_inverted() == new_connection:
70              connection.bidirectional = True
71              return connection_list
72      # Else add to the list
73      connection_list.append(new_connection)
74      return connection_list
75
76
77  # Get a list of connections between modules, based on a buss dictionary
78  def get_connections_from_buss_dict(dict_busses):
79      connections = list()
80      for bus in dict_busses:
81          for read_source in dict_busses[bus].read_sources:
82              if read_source != dict_busses[bus].get_write_source():
83                  new_connection =
                     ↪  Connection(dict_busses[bus].get_write_source(),
                     ↪  read_source, dict_busses[bus].get_ran_name())
84                  connections = update_connection_list(connections,
                     ↪  new_connection)
85      return connections
86
87
88  # Retrieve a list of masters and which slave is connected to them
```

```
89  def get_master_list(datapoints):
90      # 1: get which modules are interconnecting modules
91      interconnect_modules = set()
92      for data_point in datapoints:
93          if data_point.info_type == INFO_INTERCONNECT:
94              interconnect_modules.add(data_point.name)
95
96      # 2: get the values on its ports, which are master/slave
97      #    and which master interconnect to which slave
98      interconnections = list()
99      interconnect_ports = set()
100     relays = list()
101     relay_names = set()
102     for data_point in datapoints:
103         # Get value and type
104         if data_point.info_type == INFO_MARK:
105             interconnect_ports.add(Port(data_point.value_1,
                 ↪  data_point.value_2))
106         # Get interconnections
107         if data_point.info_type == INFO_INTERCONNECT:
108             interconnections.append(Connection(data_point.value_1,
                 ↪  data_point.value_2))
109         # Get relays
110         if data_point.info_type == INFO_RELAY:
111             relays.append(Connection(data_point.value_1,
                 ↪  data_point.value_2))
112             relay_names.add(data_point.name)
113
114     # 3: Update port values to use relay-values if neccessary
115     interconnect_updated = True
116     while interconnect_updated:
117         interconnect_updated = False
118         # Check every interconnecting port up against the relay list
119         for p in interconnect_ports:
120             for r in relays:
121                 # On hit, change the value, and change any value in the
                     ↪  interconnecting as well
122                 if p.value == r.source:
123                     interconnect_updated = True
124                     old_value = p.value
125                     p.value = r.target
126                     for i in interconnections:
127                         if i.target == old_value:
128                             i.target = r.target
129                         if i.source == old_value:
130                             i.source = r.target
131
132     # 4: Get which module reads the value out of interc.
133     masters = dict()
```

```
134     slaves = list()
135     for data_point in datapoints:
136         if data_point.info_type == INFO_READ and data_point.name not in
            ↪  interconnect_modules and data_point.name not in relay_names:
137            for p in interconnect_ports:
138                 if data_point.value_1 == p.value:
139                     if p.port_type == "Slave":
140                         slaves.append(Slave(data_point.name,
                             ↪  data_point.value_1))
141                     elif p.port_type == "Master":
142                         if data_point.name not in masters:
143                             masters[data_point.name] =
                                 ↪  Master(data_point.name)
144
                             ↪  masters[data_point.name].master_values.add(data_point.value_1)
145
146     # 5: Collect everything and return
147     # Add interconnection values
148     for m in masters:
149         for i in interconnections:
150             if i.source in masters[m].master_values:
151                 masters[m].slave_values.add(i.target)
152             elif i.target in masters[m].master_values:
153                 masters[m].slave_values.add(i.source)
154     # Add slaves
155     for m in masters:
156         for s in slaves:
157             if s.value in masters[m].slave_values:
158                 masters[m].slaves.add(s.name)
159
160     return masters
161
162
163 class Master:
164     name = ""
165     master_values = set()
166     slave_values = set()
167     slaves = set()
168
169     def __init__(self, name):
170         self.name = name
171         self.master_values = set()
172         self.slave_values = set()
173         self.slaves = set()
174
175     def __repr__(self):
176         return 'Name: {0}\n\tMaster Value: {1}\n\tSlave Values:
            ↪  {2}\n\tSlaves: {3}'.format(
177             self.name, self.master_values, self.slave_values, self.slaves
```

```
178         )
179 
180     def __str__(self):
181         return 'Name: {0}\n\tMaster Value: {1}\n\tSlave Values:
        ↪  {2}\n\tSlaves: {3}'.format(
182             self.name, self.master_values, self.slave_values, self.slaves
183         )
184 
185 
186 class Slave:
187     name = ""
188     value = ""
189 
190     def __init__(self, name, value):
191         self.name = name
192         self.value = value
193 
194 
195 class Port:
196     value = ""
197     port_type = ""
198 
199     def __init__(self, value, port_type):
200         self.value = value
201         self.port_type = port_type
202 
203 
204 class Bus:
205     __value = ""
206     __write_source = ""
207     names = set()
208     read_sources = set()
209 
210     def __init__(self, value):
211         self.__value = value
212         self.__write_source = ""
213         self.names = set()
214         self.read_sources = set()
215 
216     def __repr__(self):
217         return 'Name: {0}\n\tValue: {1}\n\tWrite Source: {2}\n\tRead
        ↪  Sources: {3}'.format(
218             self.names, self.__value, self.__write_source, self.read_sources
219         )
220 
221     def __str__(self):
222         return 'Name: {0}\n\tValue: {1}\n\tWrite Source: {2}\n\tRead
        ↪  Sources: {3}'.format(
223             self.names, self.__value, self.__write_source, self.read_sources
```

```
224         )
225 
226     def add_read_source(self, name_of_source):
227         self.read_sources.add(name_of_source)
228 
229     def set_write_source(self, name_of_source):
230         if self.__write_source == "":
231             self.__write_source = name_of_source
232             return
233         else:
234             print('Warning: Tried to overwrite write-source "{}" with "{}"
                ↪  (Bus value: {}).'.format(self.__write_source,
                ↪  name_of_source, self.__value))
235             return -1
236 
237     def get_write_source(self):
238         return self.__write_source
239 
240     def get_value(self):
241         return self.__value
242 
243     def add_name(self, name_of_bus):
244         self.names.add(name_of_bus)
245 
246     def get_ran_name(self):
247         if len(self.names) == 0:
248             return "None"
249         else:
250             ran_name = self.names.pop()
251             self.names.add(ran_name)
252             return ran_name
253 
254 class Connection:
255     source = ""
256     target = ""
257     name = ""
258     bidirectional = False
259 
260     def __init__(self, source, target, name=""):
261         self.source = source
262         self.target = target
263         self.name = name
264         self.bidirectional = False
265 
266     def __repr__(self):
267         return "Source: {0}, Target: {1}, Name:{3}, Is Bidirectional:
            ↪  {2}".format(
268             self.source, self.target, self.bidirectional, self.name
269         )
```

```
270
271     def __str__(self):
272         return "Source: {0}, Target: {1},Name:{3}, Is Bidirectional:
         ↪  {2}".format(
273             self.source, self.target, self.bidirectional, self.name
274         )
275
276     def __eq__(self, other):
277         if self.source == other.source and self.target == other.target:
278             return 1
279         else:
280             return 0
281
282     def get_inverted(self):
283         inv_connection = Connection(self.target, self.source, self.name)
284         inv_connection.bidirectional = True
285         return inv_connection
```

Listing B.2: bus_info_processor.py

```
###########################################################################
# Name:         Bus Info Parser
# Author:       Kevin Vinding
# Description:  Opens text file containing extracted bus information, parses
 ↪  the text, and returns it as
#               a list of data points.
###########################################################################
import datapoint as dp
import os

KEYWORD = '#bus#'
KEYWORD_LENGTH = len(KEYWORD)
DELIMITER = '#'


# Parses a text file for data points and returns them as a list
def parse_text_file(file_name):
    data_point_list = list()
    try:
        text_file = open(file_name, 'r')
    except:
        if os.path.isfile(file_name):
            print('Error: File "{}" could not be opened!'.format(file_name))
        else:
            print('Error: File "{}" does not exist!'.format(file_name))
        return -1

    while True:
        current_text_line = text_file.readline()
        if not current_text_line:
            # End of text-file
            break

        new_data_point = string_to_data_point(current_text_line)
        if new_data_point is not None:
            data_point_list.append(new_data_point)

    text_file.close()
    return data_point_list


# Parses string for a data point and converts it into a data point
def string_to_data_point(string):
    keyword_index = string.rfind(KEYWORD)
    if keyword_index != -1:
        string_after_keyword = string[keyword_index + KEYWORD_LENGTH:]
        split_string = string_after_keyword.split(DELIMITER)
```

```
47          new_data_point = dp.DataPoint(split_string[0], split_string[1],
             ↪  split_string[2], split_string[3])
48          return new_data_point
49      else:
50          # No valid data point can be extraced from string
51          return None
```

Listing B.3: bus_info_parser.py

```
1  # Class for organizing information as data points
2  class DataPoint:
3      name = ""
4      info_type = ""
5      value_1 = ""
6      value_2 = ""
7
8      def __init__(self, name, info_type, value_1, value_2):
9          self.name = name
10         self.info_type = info_type
11         self.value_1 = value_1
12         self.value_2 = value_2
13
14     def __repr__(self):
15         return "Name: {0}, Information Type: {1}, Value 1 = {2}, Value 2 =
           ↪  {3}".format(
16             self.name, self.info_type, self.value_1, self.value_2
17         )
18
19     def __str__(self):
20         return "Name: {0}, Information Type: {1}, Value 1 = {2}, Value 2 =
           ↪  {3}".format(
21             self.name, self.info_type, self.value_1, self.value_2
22         )
```

Listing B.4: datapoint.py

Bibliography

[1] T. Martin, *The Designer's Guide to the Cortex-M Processor Family.* Newnes, second ed., 2016.

[2] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, 2018.

[3] Nordic Semiconductor, "nRF5340 Product Specification v1.1." `https://www.nordicsemi.com/Products/Low-power-short-range-wireless/nRF5340`, 2021. [Online; accessed 19-May-2021].

[4] S. Pasricha and N. Dutt, *On-chip communication architectures: system on chip interconnect.* Morgan Kaufmann, 2010.

[5] W. Chen, J. Ray, S.and Bhadra, M. Abadir, and L. Wang, "Challenges and trends in modern SoC design verification," *IEEE Design & Test*, vol. 34, no. 5, pp. 7–22, 2017.

[6] Y. Chen and S. Y. Kung, "Trend and challenge on System-on-a-Chip designs," *Journal of Signal Processing Systems*, vol. 53, pp. 217–229, 2008.

[7] S. Sutherland, S. Davidmann, and P. Flake, *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling.* Springer Science & Business Media, second ed., 2006.

[8] Arm Ltd., "AMBA." `https://developer.arm.com/architectures/system-architectures/amba`, 2021. [Online; accessed 09-Feb-2021].

[9] R. P. Patil and P. V. Sangamkar, "A review of system-on-chip bus protocols," *International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering*, vol. 4, no. 1, pp. 271–281, 2015.

[10] OpenCores, "Wishbone, Revision B.4 Specification." `https://opencores.org/howto/wishbone`, 2010. [Online; accessed 16-May-2021].

[11] D. Flynn, "AMBA: enabling reusable on-chip designs," *IEEE micro*, vol. 17, no. 4, pp. 20–27, 1997.

[12] A. Shrivastav, G. Tomar, and A. K. Singh, "Performance comparison of AMBA bus-based system-on-chip communication protocol," in *2011 International Conference on Communication Systems and Network Technologies*, pp. 449–454, IEEE, 2011.

[13] Arm Ltd., "AMBA Specification (Rev 2.0)," protocol specification, May 1999.

[14] Arm Ltd., "Multi-Layer AHB v.2.0," technical overview, May 2004.

[15] W. J. Dally and R. C. Harting, *Digitak Design: A Systems Approach.* Cambridge University Press, 2012.

[16] C. Spear and G. Tumbush, *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features.* Springer, third ed., 2012.

[17] L. Wang, Y. Chang, and K. T. Cheng, *Electronic design automation: synthesis, verification, and test.* Morgan Kaufmann, 2009.

[18] L. Scheffer, L. Lavagno, and G. Martin, *EDA for IC system design, verification, and testing.* CRC press, 2018.

[19] E. Cerny, S. Dudain, J. Havlicek, and D. Korchemny, *SVA: The Power of Assertions in SystemVerilog.* Springer, second ed., 2015.

[20] D. Grune and C. J. H. Jackobs, *Parsing Techniques: A Practical Guide.* Springer, second ed., 2008.

[21] H. Saafan, M. W. El-Kharashi, and A. Salem, "SoC connectivity specification extraction using incomplete RTL design: An approach for formal connectivity verification," *11th International Design & Test Symposium (IDT)*, pp. 110–114, 2016.

[22] S. Rachamalla, A. Joseph, R. Rao, and D. Pandey, "Virtual logic netlist: Enabling efficient RTL analysis," *Sixteenth International Symposium on Quality Electronic Design*, pp. 571–576, 2015.

[23] D. Große, R. Drechsler, L. Linhard, and G. Angst, "Efficient automatic visualization of SystemC designs.," *FDL*, pp. 646–658, 2003.

[24] S. Hosny and A. Baher, "Design crawler: A web application for digital design metadata analysis," *20th International Workshop on Microprocessor/SoC Test, Security and Verification (MTV)*, pp. 31–34, 2019.

[25] Python Software Foundation, "Python." `https://www.python.org/`, 2021. [Online; accessed 12-Mar-2021].

[26] J. Ellson, E. Gansner, Y. Hu, and S. North, "Graphviz." `https://graphviz.org/`, 2021. [Online; accessed 24-May-2021].

[27] S. Bank, "Graphviz Python Package." `https://github.com/xflr6/graphviz`, 2021. [Online; accessed 31-May-2021].

Kevin A. Vinding

Automatic Extraction of Complex Bus Structures from RTL

NTNU
Norwegian University of
Science and Technology